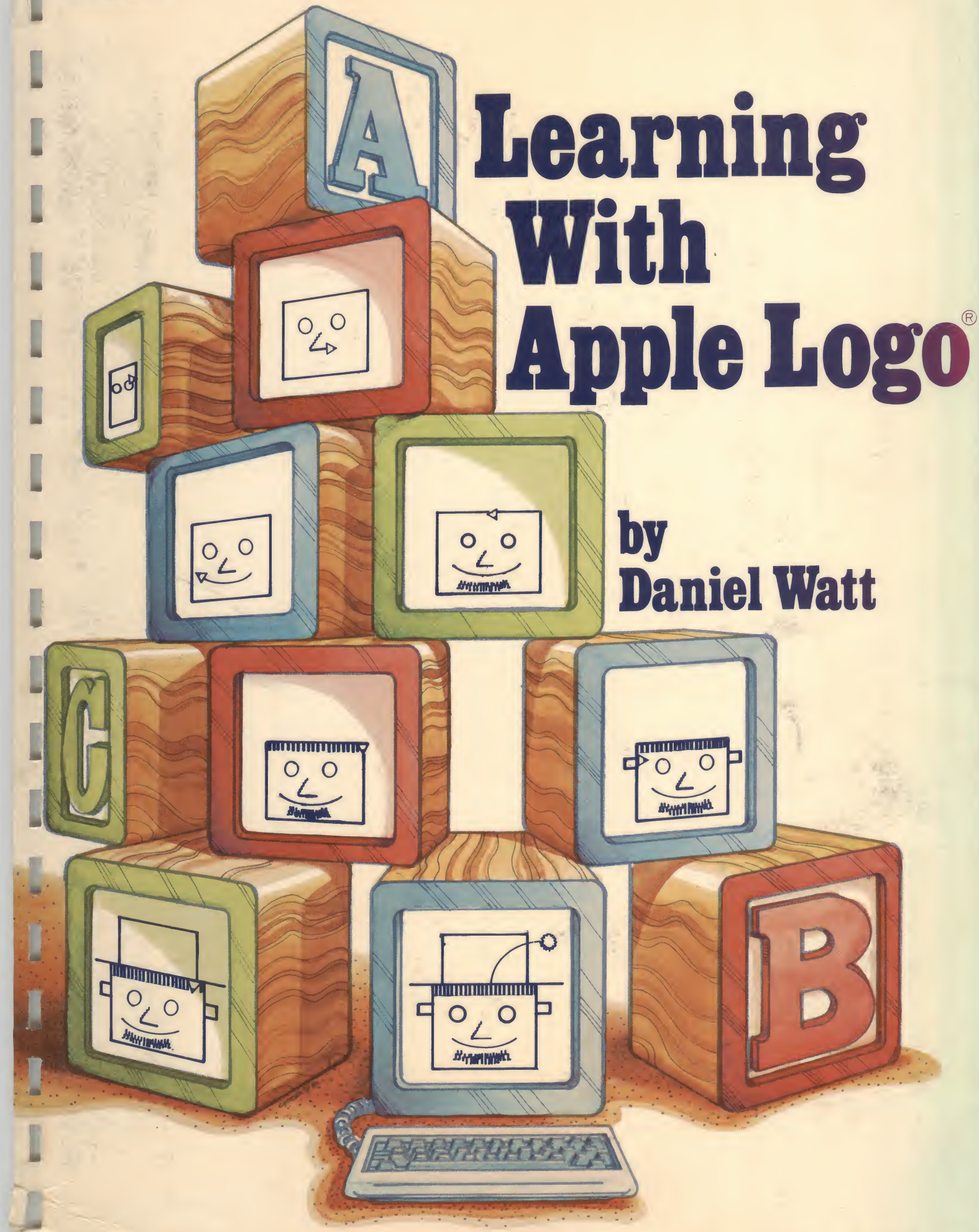
A BYTE BOOK

>$19.95

Learning With Apple Logo®

by Daniel Watt



Learning With Apple Logo

Learning With Apple Logo

by Daniel Watt

Cartoon Illustrations by Paul D. Trap

McGraw-Hill Book Company

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

The author of the programs provided with this book has carefully reviewed them to ensure their performance in accordance with the specifications described in the book. Neither the author nor McGraw-Hill, Inc., however, makes any warranties concerning the programs, or for the consequences of any such errors. The programs are the sole property of the author and have been registered with the United States Copyright Office.

Library of Congress Cataloging in Publication Data

Watt, Daniel.
Learning with Apple Logo.

Includes index.
1. Apple computer—Programming. 2. LOGO (Computer program language) I. Title.
QA76.8.A66W38 1984 001.64'24 83-26828
ISBN 0-07-068571-1

1234567890 HAL/HAL 8987654

ISBN 0-07-068571-1

Cover design by Ellen Klempner-Beguin
Cover illustration by Connie Porter

The editors for this book were Stephen G. Guty, Bruce Roberts, and Esther Gelatt, and the production supervisor was Teresa F. Leaden.
It was set in Times Roman by Byrd Data Imaging.

Printed and bound by Halliday Lithograph.

Table of Contents

Acknowledgments

I want to acknowledge my debts to the many people whose ideas have been incorporated in this book and whose efforts have helped make it a reality.

First and foremost, I want to acknowledge the contributions made by Molly Watt. Virtually every idea in the book has been discussed thoroughly with her and has been strengthened and deepened by her suggestions. Many of the ideas had their first trial use in her classes for teachers at Lesley College and Keene State College and for children at Computer Camps International. Finally, she has had the unenviable experience of having to live with me during the final months of exhausting and all-consuming effort needed to make *Learning with Apple Logo* a reality. She knows how grateful I am for her support during those months.

The ideas in the book go back to my involvement in the late 1960s with Elementary Science Study, a federally supported curriculum development project that made science a real experience for thousands of elementary school children and teachers. It was through interactions with colleagues in that effort and during my seven years of teaching elementary school in Brookline, Massachusetts, that my educational philosophy and practices were shaped and matured.

It was not until 1976, however, when I was welcomed into the MIT Logo Group, that I began to comprehend how the ideals and educational values that had been nurtured through the sixties and seventies could become reality for thousands, possibly millions of people via microcomputers, the Logo language, and the ideas and inspiration of Seymour Papert. Although Logo and the culture that has grown up around it are the work of a large group of people, there are good reasons why Seymour Papert's name is so closely identified with it. Seymour was both the chief theoretician and the chief practitioner of Logo. He gathered the research teams and obtained the funding necessary to create the hardware, software, and philosophy of teaching and learning that have made Logo a reality. Working with Seymour Papert was one of the most fortunate and rewarding experiences in my life.

Many others were part of the Logo group during the five years I was associated with it. My closest associations were with those who were involved with me on the two Brookline Logo Projects between 1977 and 1981. Many of the ideas that found their way into this book came from those people: Hal Abelson, Jeanne Bamberger, Andy diSessa, Greg Gargarian, Ellen Hildreth, Danny Hillis, Bob Lawler, Margaret Minsky, Seymour Papert, Sylvia Weir, and Ursula Wolz. Others who made major contributions to the Logo culture at MIT and to my thinking were Howard Austin, John Berlow, Paul Goldenberg, Ginny Grammer, Marvin Minsky, Cynthia Solomon, and Jose Valente.

Outside of MIT, I want to thank the hundreds of 9- through 14-year-old

students at Lincoln School in Brookline, Massachusetts, whose ideas and experiences form the basis for most of the projects in the book. It was their experiences, efforts, and enthusiasm that transformed Logo from an ivory tower educational philosophy to a living reality for me. Then there were the teachers and administrators who nurtured a Logo hothouse at Lincoln School for four years: Joan Aronson, Gerry Cote, Lisa Hirsch, Bob Lewis, Mary Parkins, Florence Regolino, Ellie Shacter, and Robin Welch, and Brookline's Superintendent of Schools, Bob Sperber, who believed in the viability of Logo when many people thought it was just a pipe dream.

There are some whose ideas have been directly incorporated in the book, and who need to be acknowledged specifically. Many of the turtle geometry projects and design ideas used in Chapters 5, 6, and 7 are based on ones that were originally collected and made available by Ellen Hildreth. Others, especially those in Chapter 8, came from the book *Turtle Geometry* by Hal Abelson and Andy diSessa, published by the MIT Press. Although programs like QUICKDRAW, used in Chapters 3 and 11, were used by many people at MIT, I was first introduced to the idea by Paul Goldenberg. Similarly, my first exposure to a SHOOT game like the one in Chapters 3 and 10 was to the one developed by Bob Lawler for his own children. I learned the idea of animating the turtle from Danny Hillis, and it was elaborated into a variety of games by students at Lincoln School. The POET procedures were borrowed from a BASIC program published by Lloyd Prentice in *Classroom Computer News*, but the idea of sentence pattern procedures goes way back to the earliest Logo experiments, before there ever was a turtle. Finally, I want to single out one student, David Libby, a sixth grader in 1978, who created the design that was used for the cover of this book.

As important as the ideas that went into *Learning With Apple Logo* was the work of those who nurtured, edited, designed, and produced it. It was a great privilege to work with the editorial and production team assembled by Ed Kelly at Byte Books. Ed Kelly, as publisher, nourished the book from the time it was barely a gleam in my eye, until it was out the door, and, not so incidentally, contributed the title, *Learning With Apple Logo*. Ellen Klempner's inspiration as designer and production coordinator has deeply influenced it, as has the thoughtful work of Bruce Roberts as editor. In addition to all her other support, Molly Watt served as "translator" in adapting *Learning With Apple Logo* from its earlier version *Learning With Logo*. Peggy McCauley as production editor also played a key role in making this book a reality. Most important of all was the synergistic way we all worked together. I only hope that the spirit with which the staff of Byte Books lovingly produced *Learning With Apple Logo* will be reflected in the satisfaction of those who read it.

Others made important contributions. I don't think I can overestimate the importance of Paul Trap's cartoons. His wit and intelligence in creating visual explanations of how Logo works have played a vital role in making Logo come alive. Similarly, Tim Taussig's cover design and Connie Porter's execution of it have been critical ingredients of the whole that has finally emerged. Finally, I want to acknowledge the practical help of Art Scottin, President of Orange Micro of Anaheim, California, who loaned me an Epson MX-80 printer and a Grappler interface card which were used to make the graphic illustrations that are essential to the book. And I'm

grateful to my daughter, Kristin Gustafson, who spent part of her summer on the tedious task of printing them out, one at a time.

Somehow a book seems to end right back where it started and so do these acknowledgments, with Molly. Without her love, care, and intelligent, critical support, this book could never have come to be.

Daniel Watt
Antrim, New Hampshire

0 Before You Start Reading This Book

Learning With Apple Logo is designed to help you learn to use a computer. In this chapter I will tell you something about how to use this book. It may seem strange for a book to start with Chapter 0. It may also seem strange to read a chapter *about* a book, before you start reading the book itself. If you want to get started with Logo right away, without thinking too much about it, go right ahead and turn to Chapter 1, "Getting Started." You won't miss much, and you can always come back to this chapter later.

On the other hand, if you like to know something about what you're getting into before you start doing it, read this chapter first.

Section 0.1. Who This Book Is For

Learning With Apple Logo is for anyone who wants to learn Logo, a modern computer language that makes it fun to program a computer. Logo is great for kids. It makes it easy to get started and simple to make a computer do exciting things. Logo is also great for adults. Once you learn Logo, you can make the computer do some wonderfully difficult things—many things that you can't easily do with other computer languages—in a way that is a lot simpler than you'd expect. Logo is for people of all ages.

Learning With Apple Logo is designed for kids and adults to use together, helping each other along. Very young children—even as young as four or five—can use the activities in Chapter 3 with the help of a parent, a teacher, or an older friend. People of about 10 or 11 should be able to read most of Chapters 1 through 6 without much help. Slightly older people should be able to read the rest of the book without much help. Most adults should be able to read the whole book by themselves, too, but unless they have unusually adventurous and playful spirits (for adults, that is), they will probably enjoy it much more if they use it with someone younger.

Teachers or parents may read *Learning With Apple Logo* and use parts of it to help learners who may not read the rest of the book at all. If you are a teacher or parent who reads the book as a way of passing ideas on to your children, I urge you to think of yourself as a learner first. Try as many of the activities, projects and explorations as you possibly can so you'll have an idea of what the learners you are helping may be experiencing.

I really know only one good way to help someone learn Logo. I sit down with a friend at a computer, make suggestions, ask and answer questions, and watch my friend work. As I watch, I try to notice any particular difficulties the person is having as well as any special things that my friend really likes and is good at. As I help someone learn, I use what I know about how other people have learned to help me decide how I can be most helpful to *this* person. Sometimes the best thing I can do is to go away and let my friend work on his own for a while. At other times I try to work very closely with someone, showing her how to do something new or working

with her on a problem that has both of us stumped. Very, very often, I learn something new myself when I help someone solve a problem or just watch someone learn.

When I decided to write a book to help someone learn with Logo, I knew that the hardest thing would be to put everything down in print, in definite words, in fixed order. With this book I want to help people learn *together* and have the same kinds of experiences that have made learning and teaching with Logo so exciting for me.

Section 0.2. How to Use This Book

This book is designed to be used directly with Apple Logo, sold by Apple Computer, Inc. If you have Apple Logo, you can type in all the commands and examples in the book exactly as they are written.

Most activities in the book can be carried out with any version of Logo. However, you may need to make minor changes as you work your way through the book using other versions of Logo. If you have an Apple computer, and are using either Terrapin Logo or Krell Logo, you should get the first edition of this book, called *Learning With Logo*. That edition also has an appendix for use with TI Logo.

Other editions of the book are being made for Commodore Logo, TI Logo, and Atari Logo. Please look for the edition that goes with your version of Logo. Your bookstore or computer store may be able to help you find it.

What's in the Book?

Learning With Apple Logo is divided into three parts. Part I, including Chapters 1 through 6, can be read by anyone from the age of ten or eleven. Part II, Chapters 7–9, can be read by eleven- to thirteen-year-olds who have finished Part I. Part III, Chapters 10–14, can be read by people from about the age of thirteen or fourteen on, or by younger readers with an adult helping them.

Part I

Part I gets people started with Logo and can keep some people busy for a long time. Chapters 1 and 2 introduce the computer keyboard and the *Logo turtle*, a robot that moves around and draws pictures on a TV screen. Chapter 3 shows how to use two preprogrammed Logo activities—a game called SHOOT that helps people learn how to explore the world of the turtle, and QUICKDRAW, which allows someone to make turtle designs, just by pressing keys on the keyboard. Chapter 4 tells how to teach the computer new commands, called *procedures*, how to use the Logo screen editor, and how to save procedures on your *Logo work disk*. Chapter 4 is a long, complicated chapter that you will probably want to come back to as you read the rest of the book. Chapters 5 and 6 show how to do dozens of turtle designs and drawing projects, most of which were taken from the work of 9- to 11-year-old learners.

Part II

Part II introduces *procedures with inputs* and tells how Logo uses *variables* to keep track of information. Chapter 7 tells how to draw turtle shapes with variable sizes and angles, how to make shapes grow or shrink, and how to use *conditional commands* to make the computer *stop* doing something. Chapter 8 introduces the POLY procedure with its many variations and shows how to make exciting mathematical designs with the turtle. Chapter 9 explains how to have conversations with the computer and how to make question-and-answer games and quizzes.

Part III

Part III includes four big projects that can start simple and grow as complicated as you want to make them. In Chapters 10 and 11 you'll learn how to create the SHOOT and QUICKDRAW projects that were used as activities in Chapter 3. Chapter 12 shows how you can make the turtle come alive and keep it moving around the screen by pressing keys on the keyboard. It also shows how to create an action game using a moving turtle as if it were a car racing around a track. Chapter 13 shows how to turn the computer into a poet that creates sentences, poems, and stories using words and patterns that you invent. Finally, Chapter 14 explains the workings of the tool procedures that are used throughout the book.

By the time you finish Part III, you will be ready to tackle just about any Logo project on your own!

Appendices

A series of appendices at the back of the book gives some useful information that you will need at different times while using the book.

Appendix I tells you how to create your own *LWAL Procedures Disk*. It contains listings of all the tool procedures and many of the sample programs used in the book.

Appendix II tells how to initialize and copy Logo work disks.

Appendix III is a list of Apple Logo commands that you can use as a reference while you work with Logo.

Cartoon Characters

As you read the book, certain special cartoon characters will help you understand some of the trickier ideas. Soon you will become very familiar with them, but I'd like to introduce them to you now.

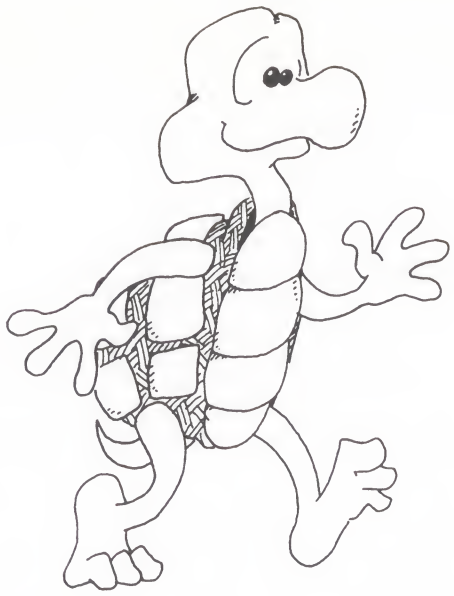


Figure 0.1: The turtle.

The *turtle* is used to represent the Logo turtle. Sometimes the turtle also represents a person reading this book.

Figure 0.2: The Logo wizard.

This is *Logo*, a "wizard" inside the computer who tries to understand and carry out your commands. Logo is shown as a wizard because he has all the powers of the computer. Because Logo wants to help you make the computer do things, he is shown as being very friendly. Because Logo is *not* very smart—he depends entirely on *you* to tell him what to do—he is shown as being easily confused. In fact, he is rather dumb, in spite of all his powers.

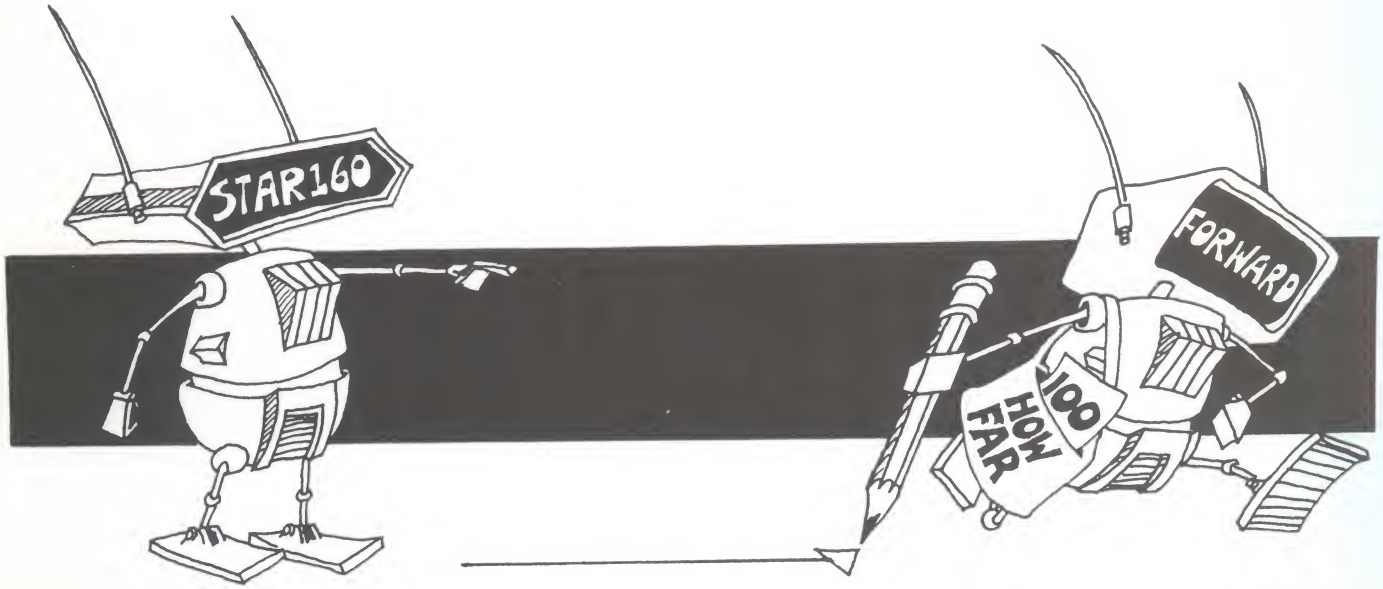


Figure 0.3: Two of Logo's helpers.

These are some of Logo's helpers. They are shown as robots because each one of them does the same job, over and over again, unless you change its instructions. Robots with rectangular heads like FORWARD or RIGHT represent built-in Logo commands, called *primitives*. Robots with diamond-shaped heads like STAR160 represent *procedures*—commands that you teach the computer. Whenever you teach the computer a new command, you create a new helper that can do its own special thing to help Logo help you.

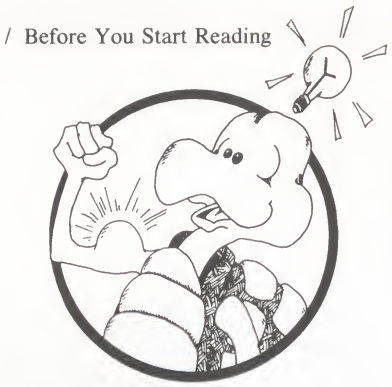
Special Sections of the Book

As you read the book, certain sections are marked with special symbols to help you use the book more easily. These sections are repeated all though the book, along with their symbols.

This symbol represents a *pitfall*, a kind of trap that many people fall into when they learn Logo. When you see this pitfall symbol, it will warn you of a possible trap, help you avoid a trap, or help you get out of one if you have already fallen in.

Don't be afraid of traps. They are almost impossible to avoid completely, and they can be a lot of fun. Think of a trap as part of an adventure—the part that you'll tell people about over dinner in years to come.

A mistake in a computer program is sometimes called a *bug*, and fixing such a mistake is called *debugging*. So you can think of these *pitfall* symbols as a guide to possible bugs, as well as a sign of traps to be avoided.



This symbol represents a *powerful idea*, an idea which makes *you* more powerful when you understand it and use it. The symbol calls your attention to ideas that will help you think more clearly and solve problems with the computer more easily. Some of these ideas will be powerful for you even when you are not using a computer.

This symbol means *exploration*. After you have learned how to do something new with the computer, you are ready to explore and make discoveries on your own. Explorations are *not* usually explained very well. After all, if you already knew the way, you'd miss the fun of exploring. Some explorations are hard and some are easy. You can try as few or as many as you like. No answers are given, except the ones that you discover yourself by using the computer.

This *helper's hint* symbol shows two learners, an older and a younger one, helping each other learn by shining a brighter light on a subject. These hints are for someone who wants to help someone else learn Logo. They may explain difficult points more fully, tell more about how to avoid common pitfalls, or just give a practical suggestion for helping someone learn Logo.

Helper's hints can also be used if you want help understanding more about what *you* are learning with Logo. In other words, you can use them to help *yourself* think more deeply about what you are learning.

Other Materials You Will Need

First, you will need an *Apple Logo Language Disk* from Apple Computer Inc. Of course you will also need an Apple II or IIe computer with a sufficient amount of memory.

As you use the computer to learn Logo, you will need a *Logo work disk* that you can use to save your own procedures as you work.

I suggest that you keep a regular *Logo journal* in which you write down what you do every day. In your journal you can take notes on what you have done, write down ideas for new projects, and keep track of any questions about things you don't understand. Another good use of your journal is to keep copies of all procedures, along with lists of procedure names and

what each procedure does. If you have a printer attached to your computer this can be done very easily—just print out your procedures on the printer and tape the information into your journal.

Another useful kind of information to keep in your journal is a list of bugs (or mistakes) that you have encountered and a record of whether you have been able to debug them or not. If you haven't been able to solve a particular bug, perhaps someone else can help you later. If you have solved a particular bug, by writing it down you may remember it and be able to help someone else at a later time. Even if you don't use your journal very often, it's a good idea to keep it near the computer just for those special times when you do want it.

Finally, when you use this book you will need a special disk of Logo procedures called the *LWAL Procedures Disk*. This disk is described in the next section.

Section 0.3. Using the LWAL Procedures Disk

This book is designed to be used with a preprogrammed disk called the *LWAL Procedures Disk* (LWAL is short for "Learning With Apple Logo").

The disk has several purposes. First, it contains a number of *tool procedures* that can be used just as if they were built-in Logo commands. For example, circle and arc procedures are used in Chapters 2, 5, and 6, as well as in later chapters. Procedures to print pictures on a printer are needed for Chapter 4 and throughout the rest of the book. Other tools are needed for the projects in Chapters 9 through 13.

Second, the LWAL Procedures Disk contains sample procedures for the long projects in Chapters 9–13. Of course, you could type in the procedures as you go along, as you have to do with most computer instruction books. My aim is to help you *understand* the procedures, and the first step in understanding them is *using* them and seeing what they do. I think it best to allow you to use the procedures first, without wasting a lot of time typing them in (and having to debug many typing errors along the way). When you begin to *modify* the procedures you *will* have to type in all the changes yourself, but you will know by that time what the procedure does and how it works.

Third, there are two preprogrammed Logo activities, SHOOT and QUICKDRAW, introduced in Chapter 3. These activities are especially useful for very young learners and for Logo beginners. There is no way to carry out these activities without using the SHOOT and QUICKDRAW procedures from the LWAL Procedures Disk.

Many of the other projects in Chapters 9–13 can also be used by people long before they are ready to understand the procedures themselves. The games, quizzes and activities of these later chapters can give beginning Logo learners a very different sense of what is possible with Logo and of where their learning experiences may be heading.

Obtaining the LWAL Procedures Disk

Because bookstores don't like to sell books with disks in them, and because this book would cost a lot more if it came with a disk, the book and the disk are being sold separately. It is essential that you have a copy of the

LWAL Procedures Disk when you use this book, and you can get the disk in one of three ways:

1. You can order the disk for $15.95 including postage. Use the tear-out order form at the back of the book and send a check made out to Creative Publications, Inc. to: Creative Publications, Inc., PO Box 10328, Palo Alto, CA 94303. **Be sure to indicate that you are using Apple Logo when ordering.**
2. You can copy the procedures from Appendix I, "Making Your Own LWAL Procedures Disk." Although this may sound like the easiest way to obtain the LWAL Procedures Disk, you may find that it is difficult to copy a set of Logo procedures accurately unless you know enough Logo to understand how the procedures work. That is, you may have to know a lot of what's in the book before you can copy all the procedures in Appendix I without making too many errors.
3. You can copy the disk from someone who has purchased it or copied it from Appendix I. Let me make this clear. The procedures on the LWAL Procedures Disk are copyrighted by Educational Alternatives, 1983. However, anyone who has purchased a copy of this book is hereby given permission to make one copy of the LWAL Procedures Disk (and one back-up copy) for his or her personal use.

Similarly, if you have already obtained the LWAL Procedures Disk you may make a copy of it for someone else who has purchased *Learning With Apple Logo*. You may not *sell* anyone a copy of the disk, but you can *give* a copy to anyone who has purchased the book. Please don't give a copy to anyone who has *not* purchased the book or trade or sell the procedures on the disk.

My entire purpose in providing the disk is to make it possible for you to use this book. I hope that you will help anyone who has bought and paid for the book get a copy of the disk. But to protect my rights, I'm asking you not to give or sell the procedures to anyone else.

Section 0.4. What Is Logo?

What *is* Logo? There are really several answers. First, *Logo is a computer language*, that is, a code for translating numbers and symbols that people understand into electronic impulses that the computer "understands."

A longer, more useful answer is that *Logo is like a dumb but powerful mechanical servant*. Its job is to help you make the computer do things. You give it commands, and Logo's helpers—*commands and procedures*—carry them out. If Logo doesn't understand a command it will tell you. You can teach it how to perform new commands by using things it already knows. The things you teach it are called procedures. Every procedure gives Logo more helpers, so that it can do more for you.

When we talk about Logo as if it were a *person* that "knows," "understands," "wants to help you," etc., we do so because it helps us think about solving problems with the computer. We know that Logo isn't *really* a person, but if we can visualize Logo as a personality, it helps us understand its behavior at a number of tricky points. The cartoon drawings throughout the book, showing Logo as a friendly "wizard" and Logo's

helpers as mechanical "robots," are designed to help you visualize the workings of the computer.

There's even a *longer* answer. *Logo isn't just something you learn. It's something you learn with.* When you use Logo to make the computer do things, you're learning about the things you're making it do. Learning with Logo can also help you understand more about yourself and the ways you learn. The rest of this book is designed to help you understand Logo as something to learn with.

If you want to understand more about Logo as a computer language, you should read one of Harold Abelson's books, *Apple Logo, Logo for the Apple II,* or *TI Logo.*[1] If you want to understand more about the background and philosophy of Logo as a learning environment, I strongly recommend Seymour Papert's book, *Mindstorms: Children, Computers, and Powerful Ideas,*[2] which explains how and why Logo came to be, and how it makes the computer into what he calls "an object to think with."

And now, time to get started!

---

[1]Published by Byte Books, McGraw-Hill, New York, 1982, 1983.
[2]Published by Basic Books, New York, 1980.

CHAPTER 1

Command	Short Form	Examples With Inputs
PRINT	PR	PRINT [CATHY PERINI]
CLEARSCREEN	CS	
FORWARD	FD	FORWARD 20, FD 20
BACK	BK	BACK 10, BK 10
RIGHT	RT	RIGHT 90, RT 90
LEFT	LT	LEFT 30, LT30
ERALL		

(No tool procedures or files used.)

1 Getting Started

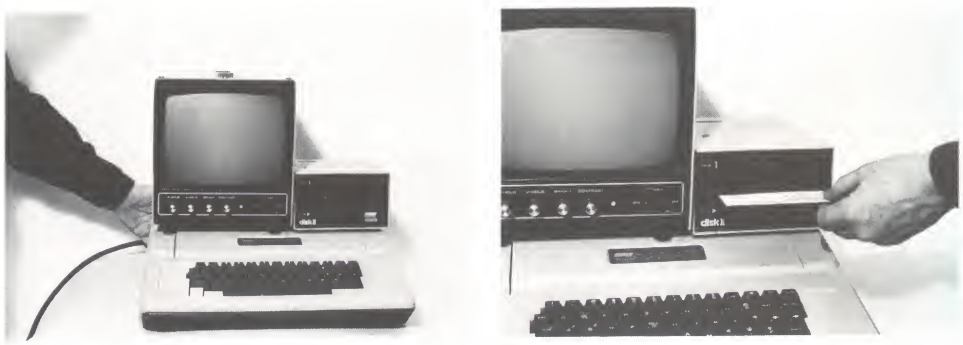
In this chapter you will learn to load Logo from a Logo Language Disk and to use the Apple II computer keyboard. You'll also meet a very famous character, the Logo turtle, and learn some commands that make the turtle move and draw pictures on your TV screen.

To start working with Logo, you'll need an Apple II computer and an Apple Logo Language Disk. It might be a good idea to have your Logo journal and a pen or pencil handy, too, in case you want to write something down.

Section 1.1. Loading Logo from an Apple Logo Language Disk

First things first. Before you can make the computer do anything, you have to *load* the Logo language into the Apple II computer's memory. To do this, follow the steps shown in Figure 1.1.

- Make sure the Apple is turned off.
- Open the disk drive and slide the Apple Logo Language Disk into it.



- Close the cover of the disk drive.
- Turn the Apple on, using the power switch at the rear.

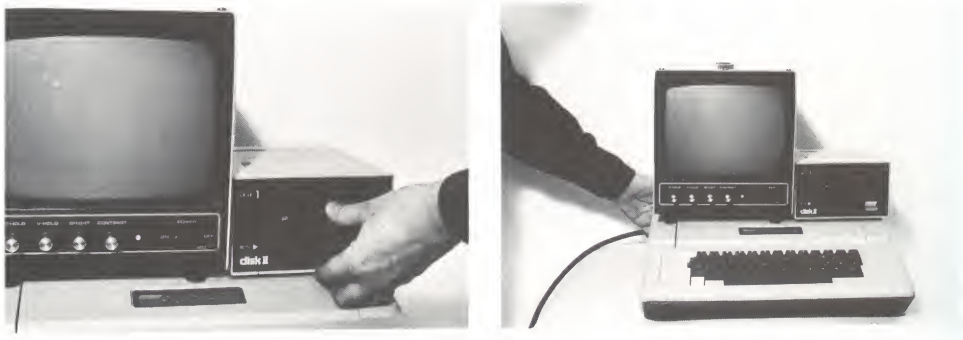
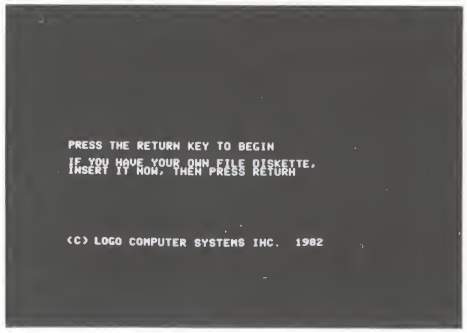


Figure 1.1: The sequence of actions involved in loading Logo.

- Wait . . . you'll hear some soft clicking sounds from the disk drive . . . then wait some more. After a few seconds, a message will appear on the screen. If you have your own Logo work disk, remove the Apple Logo Language Disk from the disk drive, insert your own work disk, and press the *key* labeled **RETURN**. (Don't type the letters R, E, T, U, R, N.) If you do not have your own Logo work disk, just press **RETURN**.

- Wait . . . you'll hear more soft clicking. After a few more seconds, a **WELCOME TO LOGO** message will appear on the screen.



- Remember to put your Apple Logo Language Disk away in a very safe place.

Figure 1.1 (continued).

The symbol appearing to the left is used to help you avoid a possible problem.

If you do not get a beep when you turn on the power, or if the yellow POWER light on the keyboard doesn't come on, check to make sure everything is plugged in properly.

If the red light on the disk drive doesn't go on or if the disk drive doesn't begin making a series of gentle clicking noises, check to see that it is connected properly, that the disk is inserted properly, and that the cover is closed.

If you have difficulty loading Logo, get help from someone who knows how or read the instructions that came with your Apple Logo Language Disk.

How to STOP Using Logo

If you are all finished with the computer, just put the Apple Logo Language Disk away, along with any other disks you have been using. Then turn the power off using the power switch on the back of the computer.

If *you* are finished using Logo but *someone else* is ready to use it, don't turn the power off. Instead, type:

ERALL **RETURN**

This will erase everything, clearing the computer's memory and leaving Logo ready for a new user. When you see a word in dark type like **RETURN** in this book, it means *press the key* labeled "RETURN." Do not *type the word* "RETURN" on the keyboard.

Section 1.2. Using the Apple Keyboard

Before you can give commands to the computer you need to know something about how the Apple's keyboard works. A computer keyboard is a lot like a typewriter. When you press a key, the symbol you type appears on the TV screen instead of on paper. The blinking light on the screen is called the *cursor*. It shows where the *next* letter, number, or symbol will be typed on the screen.

Look at the drawing of the Apple II plus keyboard in Figure 1.2. There are a few special keys on the keyboard that work differently than those on a typewriter. The keyboard of the Apple IIe is described on page 14.

Figure 1.2: The Apple II plus keyboard, including letters, numbers, and special keys.

Special Keys

The special keys on the Apple II plus are the ones marked ←, **CTRL**, **SHIFT**, **RESET**, **REPT**, **RETURN**, and →

The ← key makes the computer backspace and *erase* the letter or number you just typed. This key is called the *left arrow key*.

The **CTRL** key is used in combination with other keys for special commands. ("CTRL" is short for "control.") For example, **CTRL-G**, a combination of the **CTRL** and the **G** keys, makes Logo stop whatever it is doing. Whenever you use the **CTRL** key, hold it down first and keep it down while pressing the other key.

The **SHIFT** key appears twice on the keyboard. It is also used in combination with other keys. For example, **SHIFT-1** makes the computer print the **!** symbol. **SHIFT-/** prints a **?**. Whenever you use the **SHIFT** key, hold it down first and keep it down while pressing the other key.

Apple IIe

If you have an Apple II plus, **SHIFT-N** and **SHIFT-M** print the **[** and **]** symbols on the screen. These are very important symbols for Logo, as you will soon see. **SHIFT-2** prints ", another important Logo symbol.

RESET is a key that *resets* the computer and forces you to load Logo all over again. **Avoid pressing RESET while using Logo!**

On the Apple II plus, **REPT** is a key that is used in combination with other keys. It makes the computer *repeat* the other key you type as long as you keep both keys down.

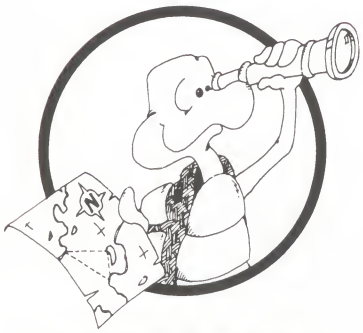
RETURN is the most important special key for Logo. After you have typed a Logo command, **RETURN** tells Logo to *do it*. You must type **RETURN** after every Logo command.

CTRL-B and → are used to move the *cursor* left and right.

The keyboard of the Apple IIe is a bit different from that of the Apple II plus. **[ ]** are marked directly on the keyboard. The " symbol is printed by typing **SHIFT-'**, which is located directly under **[** and **]**. If you hold down any key for a few seconds, that key will keep repeating.

PITFALL

Be sure the **CAPS LOCK** key is *down* when you use Logo. Also, do *not* use **SHIFT** when you type **[ ]** on the Apple IIe.



EXPLORATION

This symbol is an invitation to *explore* the use of the computer, to see what you can find out. It's one of the best ways to learn exactly what the computer will do.

Here are some ways to explore the use of the keyboard:

- Type a lot of different keys. Then press ← and see if you can rub them all out.
- Type **CTRL-G** by pressing **CTRL** and **G** at the same time. Watch what the computer prints on the screen.
- Use **SHIFT** with a lot of other keys. Make sure you can type **[** and **]** using **SHIFT-N** and **SHIFT-M**, if you have an Apple II plus. Make sure you can type " using **SHIFT-2** on an Apple II plus, or **SHIFT-'** on an Apple IIe. **[** , **]** and " are important for typing certain Logo commands.
- If you have an Apple II plus, use **REPT** with other keys and see what happens. If you have an Apple IIe, hold a key down for a few seconds.
- Press the **RETURN** key. Logo may complain and print a strange message, but you'll learn about this later.
- Type anything you like. Then use → and **CTRL-B** to move the cursor. Notice the difference between **CTRL-B** and ←. They both move the cursor back, but only ← rubs out symbols as it moves back. Also notice the difference between → and the **SPACE BAR**. Both move the cursor forward, but only **SPACE BAR** leaves a space.

WARNING! The next exploration will cause the computer to "crash." Its purpose is to show you what *not* to do! When you are finished, you will have to restart Logo by following the directions in Section 1.1 again.

- Press **RESET**. (If nothing happens, press the combination **CTRL-RESET**.) You will see a message on the screen: **UH-OH. YOU HIT THE RESET KEY YOU'D BETTER REBOOT LOGO** This means that the computer has "lost" part of the Logo program. This is called a *system crash* (or just *crash* for short).

When the system crashes the computer may forget everything you have been working on or even forget how to do Logo. You have to load Logo all over again before you can go on to the next section. To load Logo, get out your Apple Logo Language Disk, and follow the directions given in Section 1.1.

HELPER'S HINT

This symbol is for people who want to *help* other people learn Logo or for people who want to think more about what they are learning when they learn Logo.

The last exploration, pressing the **RESET** or **CTRL-RESET** key, is an important one. It is very useful to *explore* a potential disaster before experiencing it accidentally. There are two reasons for this. First, it may help you avoid causing the disaster by accident. Second, it helps you learn the consequences and how to recover.

One of the best ways to learn how to use a computer system is to explore its limits, to see what happens when things are pushed to extremes. Rather than tell a learner "Never press **RESET**!" say, "Press **RESET** *now* and see what happens." A good motto for Logo explorations is "You can't hurt the computer! Try anything you like."

Obviously this does not apply to physical damage like punching the keyboard with your fist or pouring Coke on it. Such "explorations" should be discouraged if you want to have a computer left to explore with. But aside from physical damage, anything goes.

Section 1.3. Typing Logo Commands

Once the computer prints "Welcome to Logo" you are ready to go. The **?** under the "welcome" message is called a *prompt*. This is the computer's way of telling you it's ready for a command. It's a short way of saying, "I'm ready. What do you want me to do next?"

Figure 1.3: The Apple II screen showing the welcome message and prompt.

The blinking light next to the **?** is called a *cursor*. Its job is to help you type. It shows you where the *next* letter, number, or symbol you type will appear on the screen. It's very important when you want to change something you've already typed (more about that later).

Now type something simple—your name if you like. If your name is *Cathy Perini*, you'll see this:

```
?CATHY PERINI■
```

The cursor is still blinking next to the last "I." Now press the **RETURN** key. This tells the computer to "do it!" after typing a command. After pressing **RETURN,** you should see this on the screen:

```
?CATHY PERINI
I DON'T KNOW HOW TO CATHY
?■
```

Logo complained! It doesn't know how to "do" CATHY.

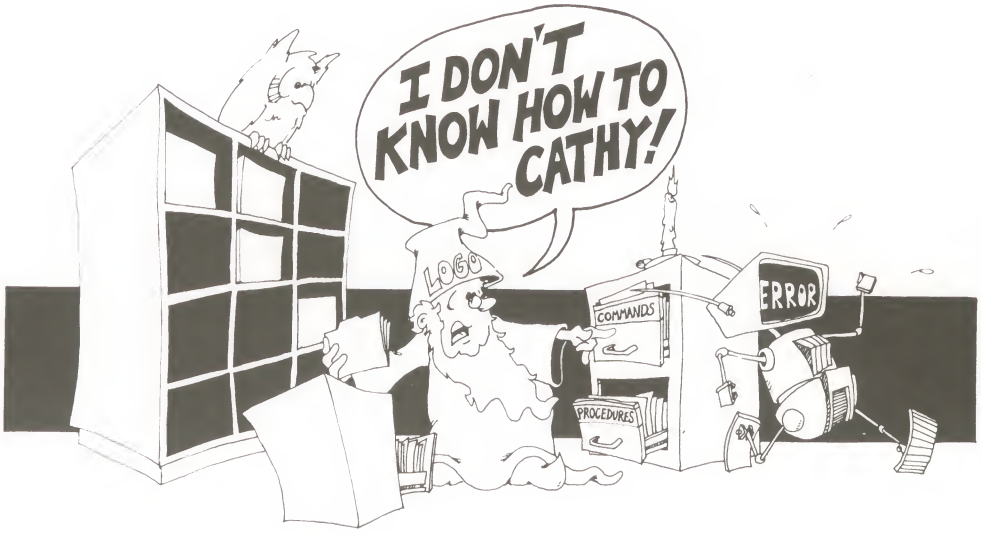


Figure 1.4: Logo complaining about an unknown command.

When Logo prints a **?** it always expects you to type a *command*, something for it to *do*. Type this:

```
PRINT [CATHY PERINI] RETURN
```

Now you should see all of this on your screen:

```
?CATHY PERINI
I DON'T KNOW HOW TO CATHY
?PRINT [CATHY PERINI]
CATHY PERINI
?■
```

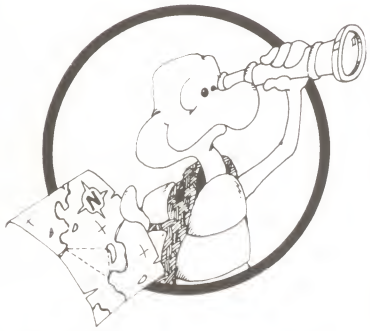
PITFALL

If you have an Apple II plus, **[** and **]** may not be marked on your keyboard. Did you remember to type **SHIFT-N** and **SHIFT-M** to type **[** and **]**? Hold down the **SHIFT** key first and then press the **N** or **M** key.

You should also remember that **RETURN** means press the *key* marked "RETURN."

In the last example, PRINT told Logo to *print* what followed. It will print anything you type between **[** and **]**. [CATHY PERINI] in the example is called the *input* to PRINT. Logo is *very fussy* about *how* to type a command. It's easy to make it complain!

If you make a mistake while you are typing, you can always use the → and **CTRL-B** keys to move the cursor and the ← key to rub out anything that was typed incorrectly. Then just type the correct symbol in its place.



EXPLORATION

Type these examples and see which ones make Logo complain:

```
PRINT   RETURN
PRINT CATHY PERINI   RETURN
PRINTCATHYPERINI   RETURN
PRINT[CATHY PERINI]   RETURN
```

Of course you don't have to use "Cathy Perini." Type your own name or anything you like. Experiment with the PRINT command. Try printing a lot of different things on the screen. See how many ways you can find to make Logo complain. The messages Logo prints when it complains are called *error messages*.

POWERFUL IDEA

This symbol is for *ideas* that give you more *power*. You can't hurt the computer by typing commands! Type anything you like. On the other hand, Logo *is* very fussy! If it doesn't understand what you type, it will complain.

HELPER'S HINT

"Trying to make Logo complain" is an excellent activity whenever you're learning something new. It helps you learn exactly what Logo will and won't accept. It also shows you what kinds of messages Logo prints when it complains. It helps a beginner get used to the idea that Logo is fussy and complains a lot.

Even very experienced Logo users make frequent typing errors or make little mistakes in Logo *syntax*—the exact rules for typing Logo commands. Logo syntax is *not* arbitrary. These are important reasons why PRINT [CATHY PERINI] "works" and PRINT CATHY PERINI doesn't. As you go along, you'll come to understand these better. For now, it's useful to think of them as if they *were* arbitrary and just learn them by rote. But the best way to learn them is to experiment with all kinds of right and wrong ways, if only just to get used to the idea that there are "right ways" and "wrong ways" to type Logo commands.

Section 1.4. Meet the Turtle

Now that you know how to give Logo a command, you're ready to meet the turtle. The turtle is usually used to draw with, so the first command to give Logo is:

CLEARSCREEN **RETURN** or CS **RETURN**

CLEARSCREEN tells Logo, "Give me a blank screen to draw on and put the turtle in the middle of it heading straight up." CS is the short form of **CLEARSCREEN**

Figure 1.5: The screen with the turtle as it appears after typing the CLEARSCREEN command.

You can command the turtle to *move* by typing FORWARD and BACK. You command it to *turn* by typing RIGHT and LEFT. Before the turtle will move or turn, you have to tell Logo how far to move or turn it by typing an *input number*.

Try these commands. Be sure to leave a space between the command and its input number, otherwise Logo will complain. And be sure to press **RETURN** after each command. Otherwise Logo won't do anything.

```
FORWARD 40
RIGHT 30
FORWARD 50
BACK 80
LEFT 90
```

Figure 1.6: The results of a series of turtle commands.

Type CLEARSCREEN again to clear the screen.

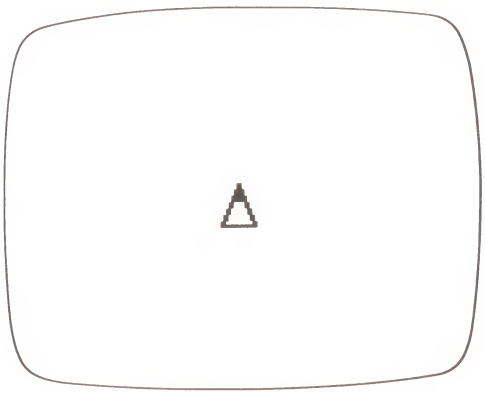


Figure 1.7: The screen after it has been cleared.

Remember to press **RETURN** after each command to make the computer *do it*. If you type FORWARD without an input number or RIGHT30 without leaving a space, Logo will complain. (Try to make Logo complain.)

You can shorten these Logo commands by typing FD, BK, RT, and LT instead of FORWARD, BACK, RIGHT, and LEFT:

```
FD 40
RT 30
FD 50
BK 80
LT 90
```

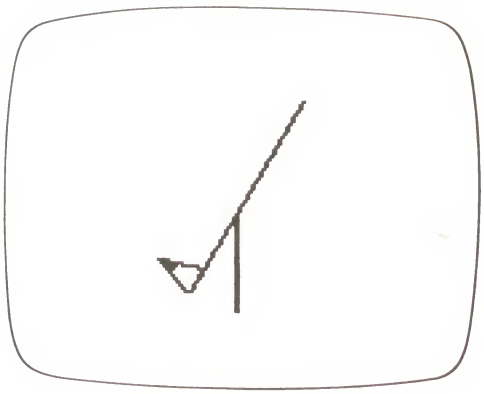
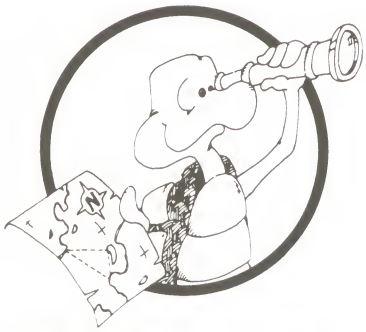


Figure 1.8: The results of a series of shortened turtle commands.

In the next chapter you will have many more chances to explore with the turtle. Try a few things now to help you get used to the basic turtle commands:



EXPLORATION

- Move the turtle all around the screen using FORWARD, BACK, RIGHT, and LEFT with different input numbers.
- Clear the screen by typing CLEARSCREEN or CS. Make the turtle draw a weird shape.
- Pick a point on the screen and make the turtle move there.
- How far is it from the center of the screen to the top edge or to either side? Use the turtle to find out.
- Type FORWARD, BACK, RIGHT, or LEFT with very large input numbers. Then try very small numbers. What are the largest and smallest distances you can make the turtle move?

HELPER'S HINT

The entire purpose of this chapter is to help someone get used to the computer, the keyboard, and the form of Logo commands. One thing you can do to help with this is to post a list of turtle commands somewhere near the computer. As new commands are learned, the list can be expanded. With a list posted near the computer, a learner doesn't have to keep flipping through the book trying to remember how a particular command is spelled or whether it needs an input.

If you are in a school or club where a lot of different people may be using the computer, you might also want to post a list of instructions for loading Logo from a disk and any other "housekeeping" functions. If clear instructions are posted near the computer, students can help each other and there is much less strain on a teacher.

CHAPTER 2

Command	Short Form	Examples With Inputs
PENUP	PU	
PENDOWN	PD	
SETPC		SETPC 3,
SETBG		SETBG 5,
LOAD		LOAD "CIRCLES
HIDETURTLE	HT	
SHOWTURTLE	ST	
WRAP		
FENCE		
WINDOW		
CLEAN		
HOME		
FULLSCREEN	**CTRL-L**	
SPLITSCREEN	**CTRL-S**	
TEXTSCREEN	**CTRL-T**	

LWAL Procedures Disk files used: "CIRCLES

New tool procedures used:

Tool Procedure	Examples With Inputs
RCIRCLE	RCIRCLE 20
LCIRCLE	LCIRCLE 10
RARC	RARC 40
LARC	LARC 20

2 The World of The Turtle

In this chapter you will spend a lot more time exploring the world of the turtle, learning more commands to control it, and learning to make it draw shapes and designs in living color.

Once you have made the turtle draw a design that you like, you might want to write down all the commands in your Logo journal. In Chapter 4 you'll learn how to teach the computer new commands and how to save them on your own Logo work disk.

Section 2.1. Basic Turtle Commands

The best way to find out what the turtle can do is to explore its world. Here are some Logo commands to help you do that:

CLEARSCREEN or CS
starts the turtle in the center of a clear screen. Use CLEARSCREEN whenever you want to clear the turtle's screen for a new design.

TEXTSCREEN
makes the turtle disappear, leaving the entire screen clear for typing.

FORWARD and BACK, or FD and BK
move the turtle forward and back. FORWARD and BACK need *input numbers* to know how far to move the turtle:

```
FORWARD 20
```
or
```
FD 20
```

```
BACK 50
```
or
```
BK 50
```

RIGHT and LEFT, or RT and LT
turn the turtle to its own right or left. RIGHT and LEFT need *input numbers* to know how far to turn the turtle:

```
RIGHT 45
```
or
```
RT 45
```

```
LEFT 100
```
or
```
LT 100
```

PITFALL

Always leave a space between a command and its input. If you type FORWARD100 or RT30, Logo will complain.

PENUP or PU
lifts the turtle's pen up so that it won't draw when it moves. Once the turtle's pen is up, it will stay up until you put it down again.

PENDOWN or PD
puts the turtle's pen down so that it will draw lines when it moves.

PENUP and PENDOWN are used when you want to draw a design that starts somewhere besides the center of the screen or when you want to have one drawing inside of or next to another one.

Try using PENUP and PENDOWN. First type CLEARSCREEN to clear the screen.

```
CLEARSCREEN
PENUP
FORWARD 50
RIGHT 90
PENDOWN
FORWARD 50
PENUP
RIGHT 45
FORWARD 20
PENDOWN
RIGHT 90
FORWARD 50
```

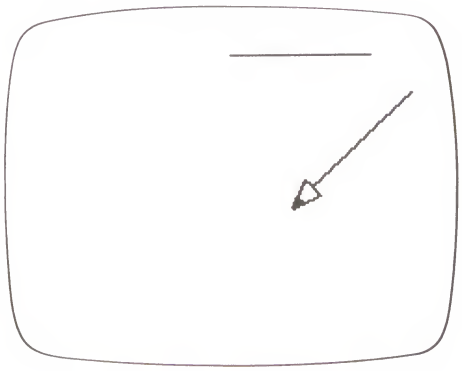


Figure 2.1: A turtle drawing made using PENUP and PENDOWN.

Later you can use PENUP and PENDOWN to make some designs like those in Figure 2.2.

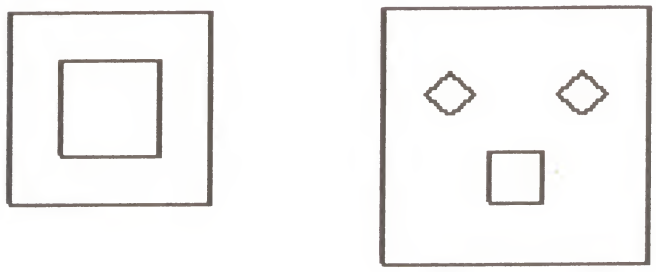


Figure 2.2: Simple drawings made using PENUP and PENDOWN.

Section 2.2. Exploring the Turtle's World

You're going to be spending a lot of time with the turtle. Take some time now to see how much you can find out about the turtle and its world. In the last chapter you began exploring with the turtle. Here are some suggestions for further explorations. Use your own ideas, too.

- Pick a point on the screen and make the turtle move there. Try points at different places on the screen.

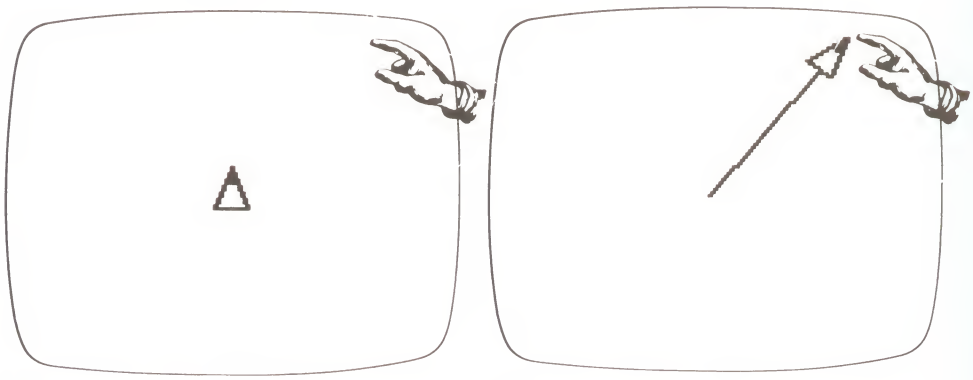


Figure 2.3: Picking a point on the screen **(a)** and moving the turtle to it **(b)**.

- How far is it from the center of the screen to the top edge? Use the turtle to find out. What happens if the turtle goes off the edge of the screen? Can you make it come back?

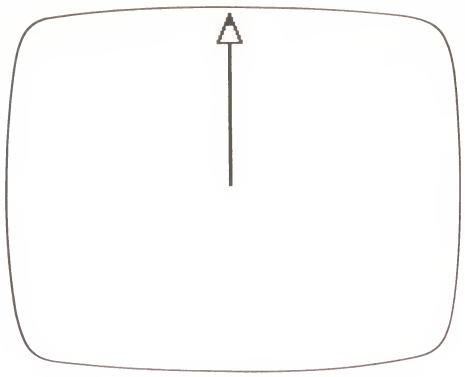


Figure 2.4: Finding the distance from the center to the top of the screen.

- Type CLEARSCREEN to clear the screen. Turn the turtle so that it is heading straight *across* the screen. How far do you have to turn it? When you think it is heading straight across, make it go forward and see.

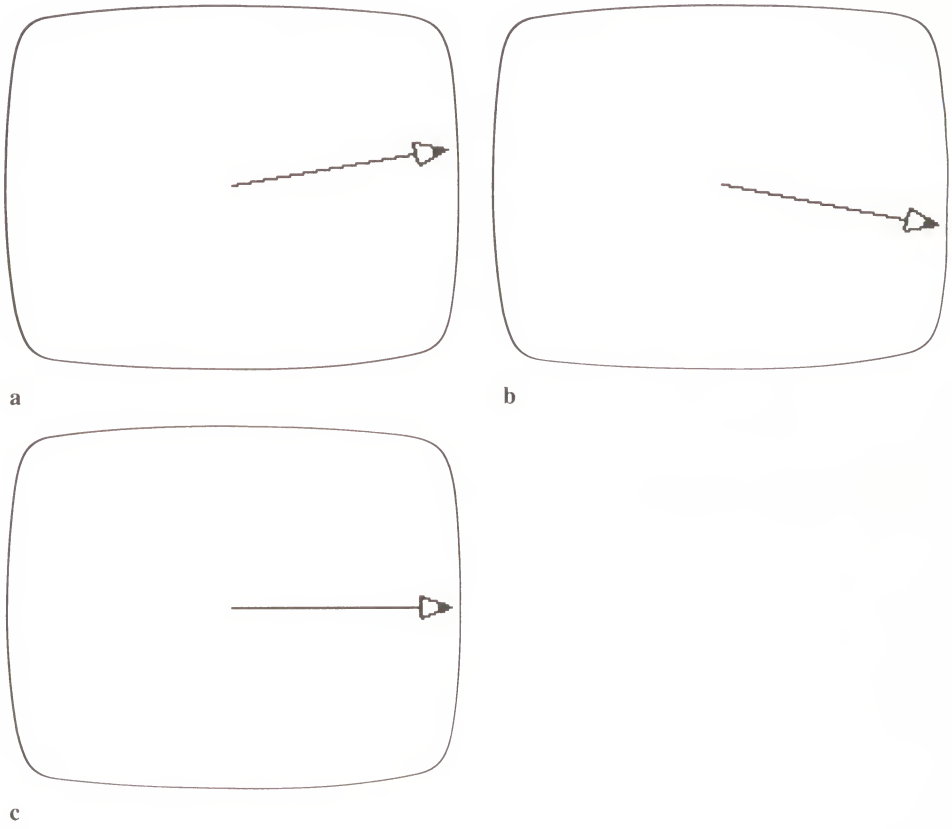


Figure 2.5: Turning the turtle to draw straight across the screen—**(a)** turned too little, **(b)** turned too much, **(c)** turned just right.

- Use the turtle to find out how far it is across the screen.

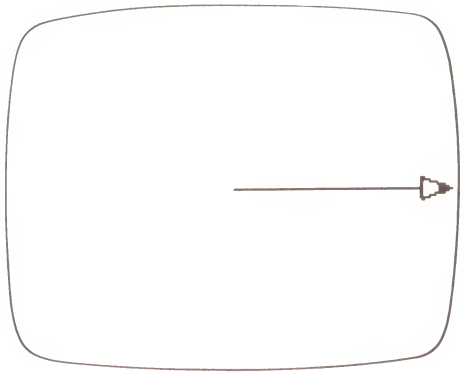


Figure 2.6: Finding the distance from the center to the side of the screen.

- Make the turtle go *off* the edge of the screen. Can you make it *wrap* around so that it comes back to the place it started? How far does it have to go?

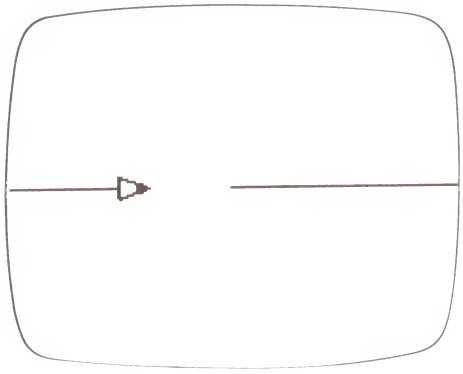


Figure 2.7: A turtle line *wrapped* from the right to the left side of the screen.

When the turtle goes off one edge of the screen and comes back on the opposite edge, we say that it *wraps* around the screen. You can think of it as a piece of string, wrapping around a package.

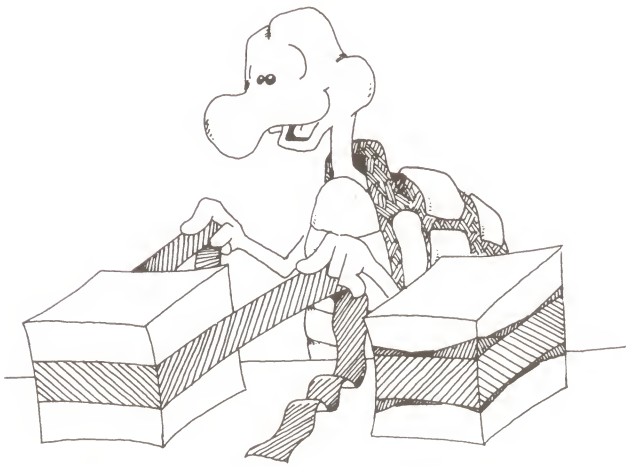
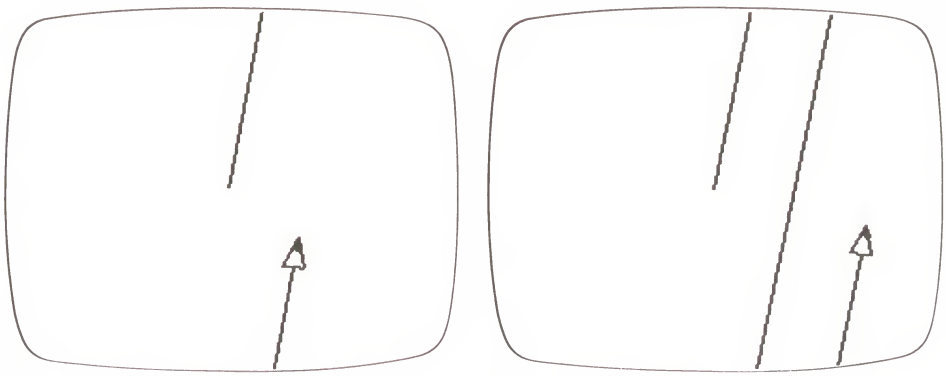


Figure 2.8: Wrapping a package.

- Type CLEARSCREEN. Then turn the turtle just a little. Now type a FORWARD command with a large input number. Can you make it keep wrapping around the screen?



a b

Figure 2.9: A turtle line wrapped around the screen once **(a)**. A line wrapped around the screen several more times **(b)**.

PITFALL

Sometimes the turtle seems to disappear *behind* the printed lines on the bottom of the screen. Even though you can't see it, it's still there.

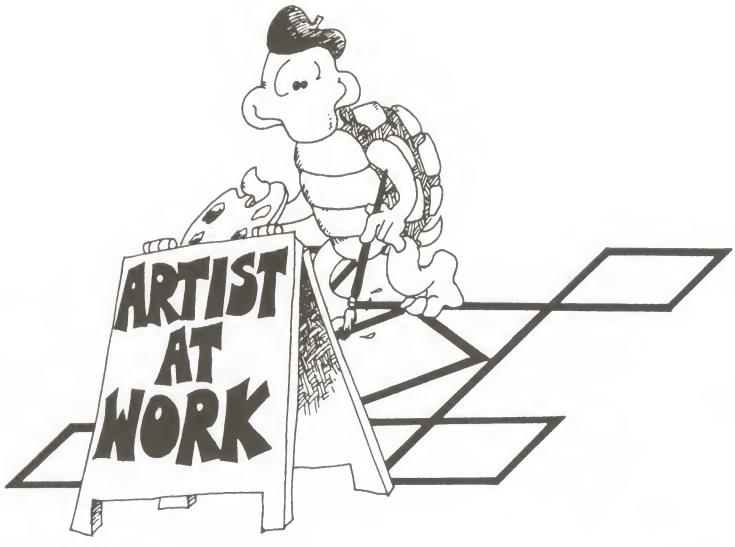


Figure 2.10: The turtle at work behind the print at the bottom of the screen even where you can't see it.

Sometimes, you may want to see the *full* graphics screen. FULLSCREEN or **CTRL-L** lets you see the *full* screen.

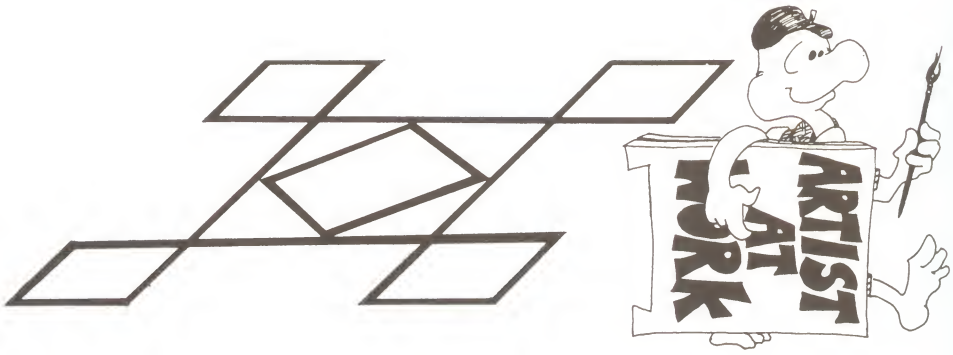


Figure 2.11: The turtle's work is revealed by typing FULLSCREEN.

Another special key command, **CTRL-S**, lets you see the *split* screen when you want to type commands again.

HELPER'S HINT

"Explorations" and "Answers"

Explorations are one important way of learning about the turtle's world. Some people enjoy them a lot, while others seem to prefer more "purposeful" activities, projects to make the turtle draw particular preplanned shapes. A Logo learning experience is made up of a combination of explorations and projects, shifting back and forth between the two modes. An exploration often leads to an idea for a project or to information that will be useful in later projects.

For example, suppose you want to make the turtle draw a square. The critical information you need is the *angle* to turn the turtle at each corner. The angle needed for drawing a square is the same as the angle needed to move the turtle straight across the screen. It happens to be 90 degrees, one fourth of 360 degrees. Turning 360 degrees is a "total turtle rotation." Factors of 360, such as 30, 45, 90, 120, etc., are very important "number facts" in the world of turtle geometry. These angles are often first discovered by learners through explorations.

Important *skills* developed through these explorations include learning to use the keyboard, becoming familiar with the concepts of forward, back, right, and left as they apply to the turtle, and learning to estimate distances and rotations on the turtle screen. These skills can also be practiced in a more structured way using the SHOOT game introduced in Chapter 3.

Learning *strategies* for finding answers to exploratory questions is more important than the answers themselves. If you think of these explorations as learning experiences, the lesson to be learned is not "the distance to the edge of the screen." The most important thing to be learned is that you *can* find this out yourself by using the computer.

There are many different approaches and many possible "answers" to any of these questions. If you're working with a group of students, a good way to emphasize this kind of learning is to pose an exploration problem and ask them to share possible ways of finding the answer. Then, after everyone has worked on the problem, ask people to share what they did and how they thought about it. In the long run, establishing an environment in which a process and the thinking behind it can be discussed (and even argued about) is the most important "learning goal" that you can pursue through explorations.

Another suggestion for working with a group is to make a bulletin board on which students can write their questions, discoveries, exploration suggestions and project ideas. This also helps establish a learning environment in which students can teach and learn from each other.

Section 2.3. Drawing Shapes with the Turtle

The turtle can draw many shapes. Simple ones like squares can be used to build more complicated shapes like those in Figure 2.12.

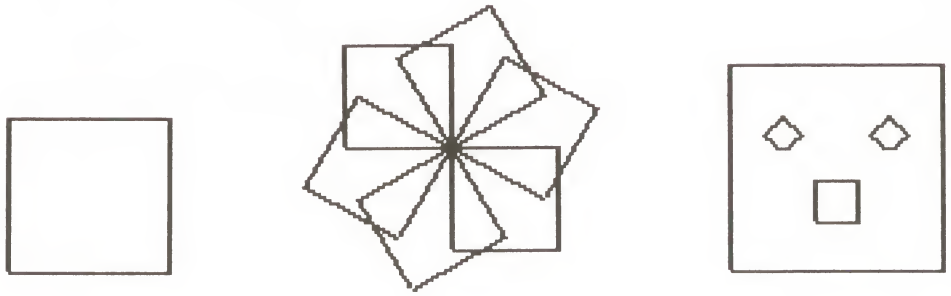


Figure 2.12: Simple turtle drawings made from squares.

You'll learn how to do this kind of thing soon. First, you should start with simple shapes. Squares and rectangles are easy to draw. A square has four equal sides and four equal turns.

You can figure out how to draw a square with the turtle by *pretending to be the turtle yourself* and walking in a square on the floor. For example, walk five steps forward, turn right 90 degrees, and keep repeating that until you get back where you started.

POWERFUL IDEA

Playing Turtle, a learning game in which you pretend to *be* the turtle, is a good way to solve a lot of turtle drawing problems. If you can't decide how to draw something on the screen, stand up and solve it *yourself* by pretending to be the turtle and walking the shape you are trying to draw.

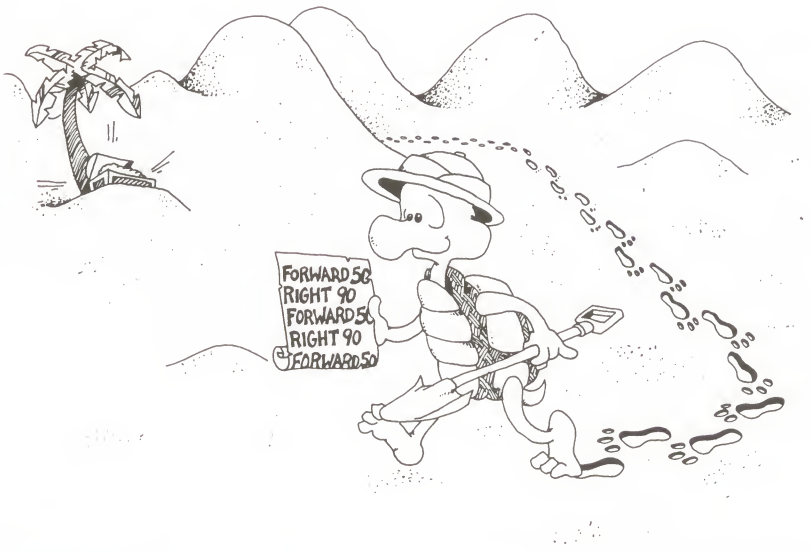


Figure 2.13: Playing turtle.

Now that you've walked in a square, it should be easy to draw one. I'll help you start. First type CLEARSCREEN to clear the screen. Then type

```
FORWARD 50
RIGHT 90
FORWARD 50
```

Figure 2.14: Starting to draw a square: CLEARSCREEN, FORWARD 50, RIGHT 90, FORWARD 50.

Can you finish drawing the square yourself?

There is another way to "play turtle" without getting up and walking. You can do this while sitting and using the computer. Just turn your head and shoulders so that they are lined up with the direction the turtle is heading on the screen. This will help you decide which way the turtle should turn next and how far it has to turn.

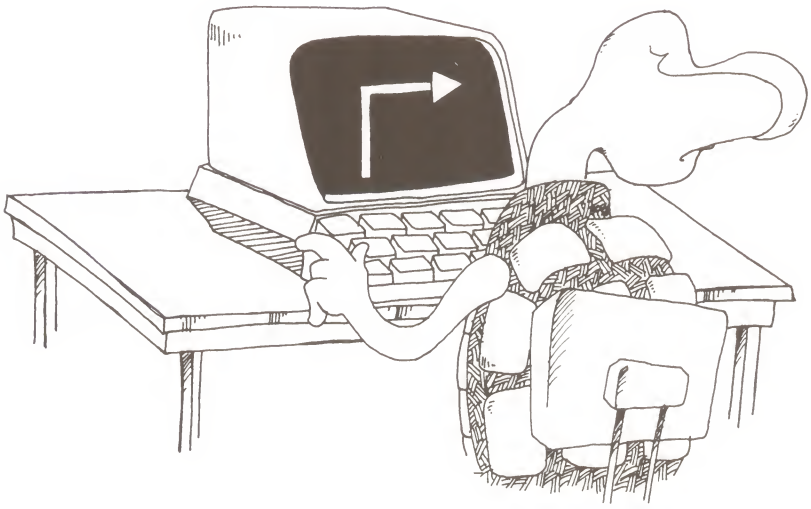


Figure 2.15: Playing turtle while sitting at the screen.

- Draw a large square and a small square.
- Make a square that turns to the *left*.
- Turn the turtle first and draw a tilted square.

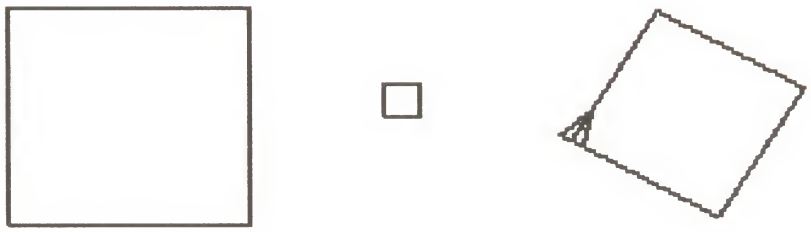


Figure 2.16: Several different squares.

- Now try drawing some rectangles. Rectangles are like squares, but only the pairs of opposite sides are equal. If you need help to draw a rectangle, stand up and walk one first.

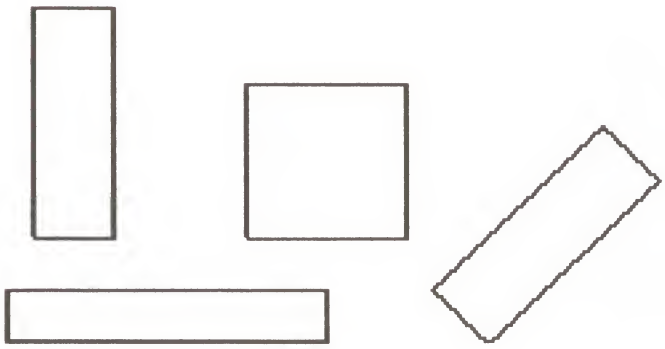


Figure 2.17: Several different rectangles.

- Make the turtle draw your initials.

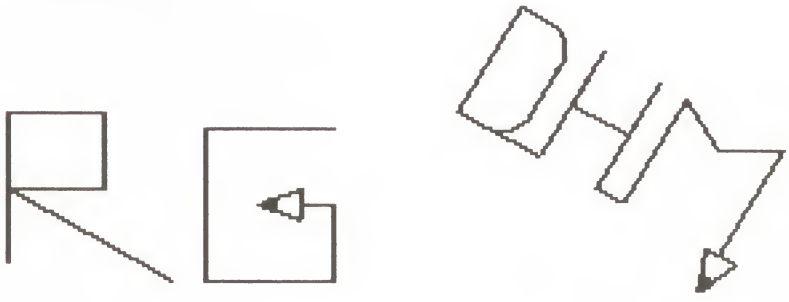


Figure 2.18: Two sets of initials drawn by the turtle.

- Draw any shapes you like.

Figure 2.19: Some of the many different shapes the turtle can draw.

If you want to *teach* the computer a new command so that it will *remember* how to draw one of your shapes, turn to Chapter 4. Be sure to come back and read the rest of Chapter 2 after learning how to teach the computer a new command.

HELPER'S HINT

Playing Turtle

Playing turtle is one of the most useful and important Logo activities. This is because it allows a learner to identify with the turtle and thereby make use of some of the most deeply ingrained and thoroughly understood knowledge that a person has—the knowledge of how to walk and turn with one's own body. By applying this knowledge it is possible to make a turtle problem a concrete and practical one—"How would I do this if I were the turtle?"—rather than an abstract and confusing one—"How can I make the turtle do something on that strange TV screen?" By physically standing up and pretending to *be* the turtle, I can actually walk through the problem that I am trying to solve on the screen. Having solved the problem myself, I can then *transfer* the solution to the screen. If you have any doubts about the effectiveness of this approach, I urge you to put aside any inhibitions you might have and try it yourself. It works remarkably well!

You can play turtle in two ways. First, you can solve a problem physically in an open space as I have just described. Second, you can simply align your head and shoulders with the turtle's position on the TV screen. This second approach is often helpful when you need to discover what the turtle should do *next*. You'll find that you can do this quite effectively without actually getting up and walking through the turtle's steps.

Playing turtle is an excellent activity to use with a group of students. All it requires is a large, open space in which the activity can be carried out. Every turtle action can be simulated as a physical activity. One person plays the turtle and other people give right and left commands and tell it how many steps to take.

The activity of playing turtle can be used to foster group cooperation as well as to learn about turtle behavior. Playing turtle requires that everyone agree on the terminology to be used in commanding the turtle. For example, everyone playing has to agree on how to interpret turtle commands like RIGHT 30 or FORWARD 10. If degrees are used as inputs to right-turn and left-turn commands, the number of possible inputs should probably be limited to multiples of 90, 45, or 30 so the person playing turtle can turn to one of four, eight, or twelve different orientations. The important thing is not so much getting the angles and distances exactly correct as getting all the players to agree on how to follow turtle commands.

Another important thing to establish when playing turtle is that the "turtle" can only follow orders, not interpret them. When the turtle walks forward, it should walk straight ahead and not veer either to the left or right, even though the person playing turtle at the moment may know that the present direction is incorrect.

When playing turtle with a group for the first time, it's usually a good idea for the teacher or group leader to take on the role of the turtle first. This has several important benefits. Students enjoy ordering a teacher around. Also the teacher can accurately model the dumb behavior of the turtle so other people can see how it ought to be done. Finally, playing turtle can seem quite "silly" the first few times it is tried. If the teacher takes on the role, other people may be more comfortable being the turtle later.

The other way of playing turtle is to orient your head in the direction the turtle is heading to help determine which way the turtle should turn or move next. For people who have difficulty knowing right from left or telling which way the turtle should turn when it is facing downward on the screen, this can be a very helpful technique. It is used over and over for solving all kinds of turtle-turn problems. Someone playing turtle this way might also want to stand up and play turtle by walking the steps and turns.

Seeing the Turtle

Some people seem to have difficulty *seeing* where the turtle is pointing. This is because of the difference between what the computer can *calculate*—the exact angle of the turtle—and what can be easily shown on the TV screen, using what is called "high-resolution graphics." The TV image is just not fine enough to show all angles clearly. It helps to remember that the turtle heads in the direction given by the *solid* corner of the triangle.

A related problem is that when the turtle turns a small amount, RIGHT 1 or RIGHT 3, for example, there may not be any *visible* change in the turtle's screen position. This is because there are only 36 different turtle positions that can be *shown* on the screen. Any lines drawn by the turtle will be drawn in the correct direction, however. Try this sequence of commands and watch what happens.

```
CLEARSCREEN
FORWARD 50
BACK 50
RIGHT 5
```

The turtle does not *appear* to turn after the RIGHT 5 command.

```
FORWARD 50
```

The new line is drawn at the corrected angle.

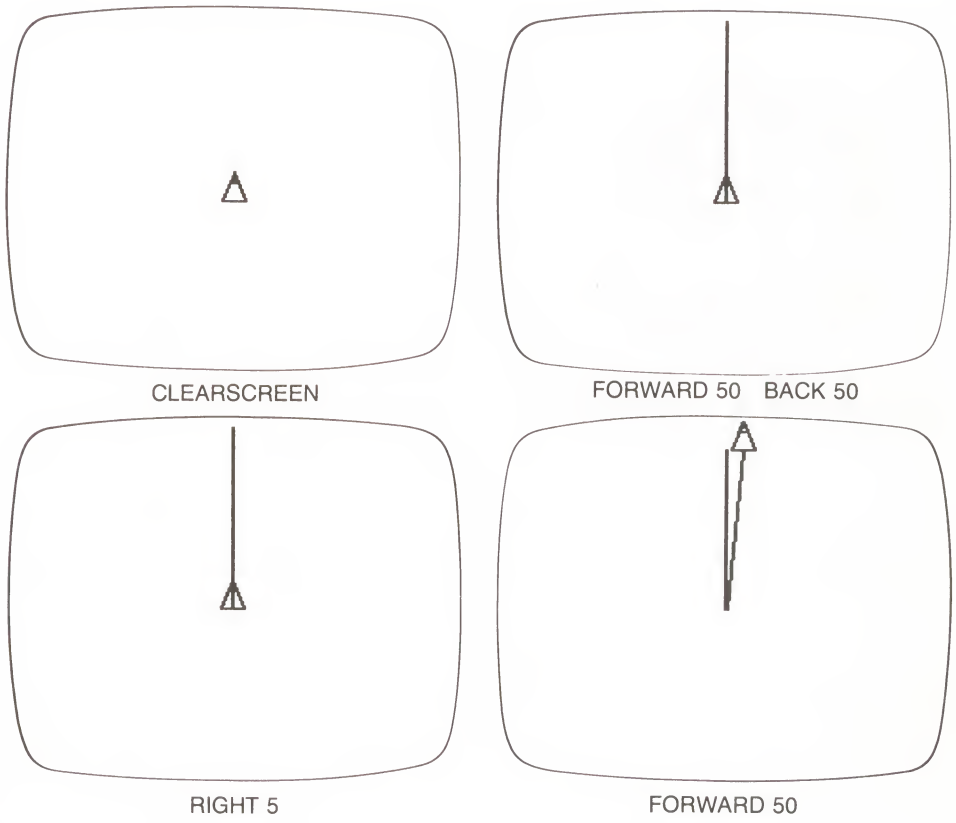


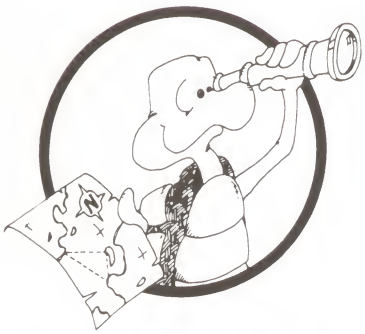
Figure 2.20: Although the turtle does not appear to turn when given the RIGHT 5 command, the final line drawn shows that it *did* turn.

Section 2.4. Living Color

If you have a color TV or monitor, the turtle can draw in six different colors, using the SETPC (set pen color) command to set the color of the turtle's pen. The actual colors you get will depend on the color setting of your TV screen. SETPC needs an input number between 0 and 5.

SETPC 0 draws in black.
SETPC 1 draws in white.
SETPC 2 draws in green.
SETPC 3 draws in violet.
SETPC 4 draws in orange.
SETPC 5 draws in blue.

The screen can also have six different background colors using SETBG (set background color) with input numbers from 0 to 5.



EXPLORATION

Here are some ideas for exploring different pen and background colors.

- Type SETPC 2 and then draw a green square with the turtle. Change to another color, and draw a different-sized square nearby.
- Type CLEARSCREEN to clear the screen and change the background with the SETBG 2. Draw different shapes on this background, using black, white, and violet pens (SETPC 0, SETPC 1, and SETPC 3). What will happen if you draw on background 2 with pencolor 2?
- Draw a black shape on white background.
- Change background colors after drawing a shape.
- Draw a shape with many different colors in it. Here are the commands for a square with black, green, violet, and orange sides on a white background:

```
SETBG 1
SETPC 0
FORWARD 50
RIGHT 90
SETPC 2
FORWARD 50
RIGHT 90
SETPC 3
FORWARD 50
RIGHT 90
SETPC 4
FORWARD 50
```

PITFALL

The Apple II computer does not allow green or violet to be drawn next to orange or blue. You can't have a green or violet line on an orange or blue background, and you can't have an orange or blue line on a green or violet background. Sorry, that's just the way it is.

Section 2.5. Designs with Circles and Arcs

Logo does not have built-in commands to draw circles and arcs. Instead, you will have to load those from your LWAL Procedures Disk. The helper's hint which follows explains how to use the circle and arc procedures in the Apple Logo startup file.

Note: If you do not have an LWAL Procedures Disk, you can ask an adult or an older friend who knows more about Logo to copy the procedures for you from Appendix I. Otherwise you will have to skip this section until after you have read Chapter 4. After you read Chapter 4 you will be able to copy the procedures from the appendix yourself and save them on your own LWAL Procedures Disk.

If you *do* have the circle procedures saved on a disk, put that disk in the disk drive and type

LOAD "CIRCLES **RETURN**

PITFALL

To type " on an Apple II plus, type **SHIFT-2,** that is, press the **SHIFT** key and the **2** key at the same time. To type " on an Apple IIe, type **SHIFT-'**.

Only type " *before* the name of the file. Logo never uses " at the end of a word.

After you type LOAD "CIRCLES the disk drive will whir and click just as it does when you load Logo.

Logo now knows a set of *procedures* for drawing circles and arcs. RCIRCLE and LCIRCLE draw circles. RARC and LARC draw *quarter circles*. Each of these commands needs one input number—the *radius* of the circle or arc you are drawing. (An *arc* is a part of a circle. The *radius* of a circle or arc is the distance from its center to its edge.) RCIRCLE and RARC draw shapes curving to the right. LCIRCLE and LARC curve to the left.

```
RCIRCLE 20
LCIRCLE 10
LARC 20
RARC 10
```

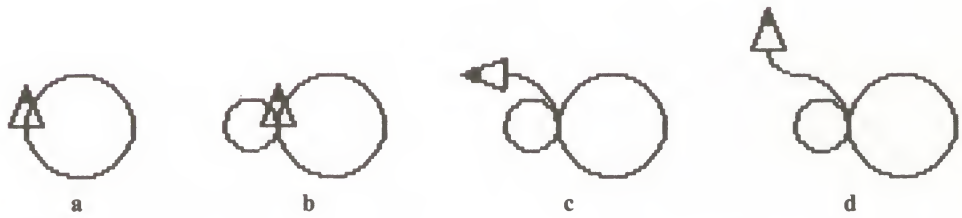
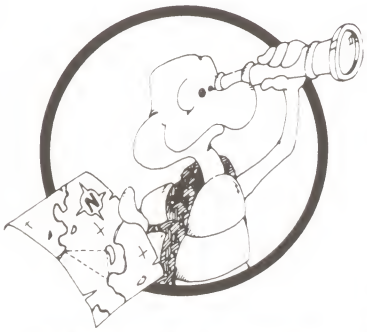


Figure 2.21: Drawings made by circle and arc commands: **(a)** RCIRCLE 20, **(b)** LCIRCLE 10, **(c)** LARC 20, **(d)** RARC 10.

HELPER'S HINT

Apple Logo does have circle and arc procedures in its startup file. If you want to use them, leave the Apple Logo Language disk in the disk drive when you load Logo. You can use the procedures CIRCLER and CIRCLEL wherever you see RCIRCLE and LCIRCLE in this book. You can also use ARCR and ARCL instead of RARC and LARC. ARCR and ARCL need two inputs, a radius and an angle. To use them in the same way as RARC and LARC, you will have to keep the second input, the angle, fixed at 90. For example, use ARCR 10 90 in place of RARC 10. Use ARCL 20 90 in place of LARC 20. Section III.7 of Appendix III explains how to add my circle and arc procedures to the Apple Logo startup file.

I have included my own circle and arc procedures for several reasons. The names RCIRCLE, LCIRCLE, RARC, and LARC are easy to use and remember, and I prefer to use arc procedures with only a radius input, at least at first. In addition, I think that the procedures themselves are easier to understand. The use of ARCR and ARCL is explained on pages 98–100 of the tutorial booklet, *Introduction to Programming through Turtle Graphics*, that comes with Apple Logo.



EXPLORATION

Here are some interesting things you can do with circles and arcs. You will have a chance to do a lot more when you read Chapter 5.

- Draw circles of different sizes curving to the right.

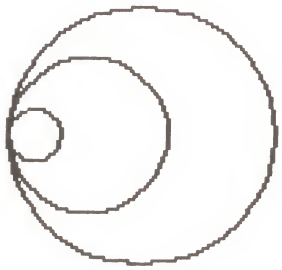


Figure 2.22: Different-sized circles made with RCIRCLE.

- Make the same design with circles curving to the left.

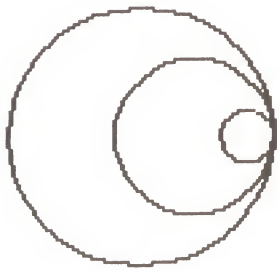


Figure 2.23: Different-sized circles made with LCIRCLE.

- Put them together to make a "butterfly."

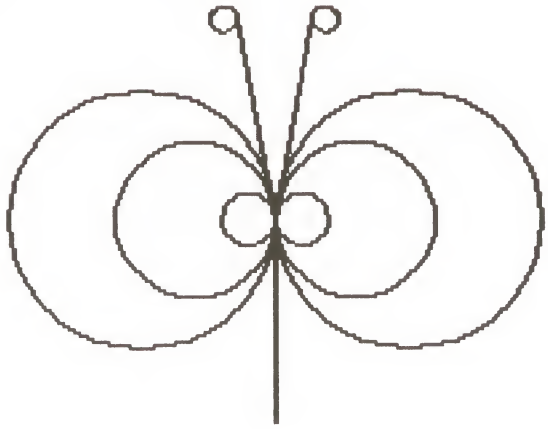


Figure 2.24: A butterfly made with circles.

- One interesting thing to do with circles is to see what happens when they get very big.

```
RCIRCLE 50
RCIRCLE 100
RCIRCLE 200
RCIRCLE 500
```

and so on

- What input number would you need to make a circle that splits up and rejoins itself at the center, like this one?

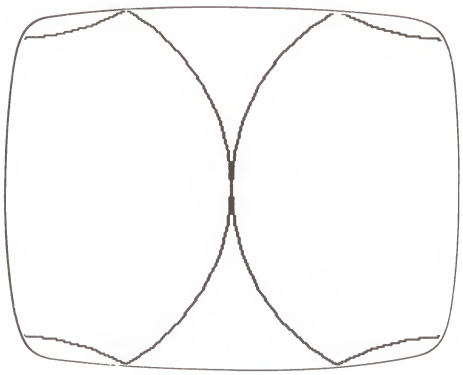


Figure 2.25: A circle big enough to wrap around the screen and touch itself.

- See if you can make a wave pattern like this with RARC and LARC.

Figure 2.26: A wave pattern made with RARC and LARC.

- If you keep making the radius of your arc bigger, you can make a kind of spiral.

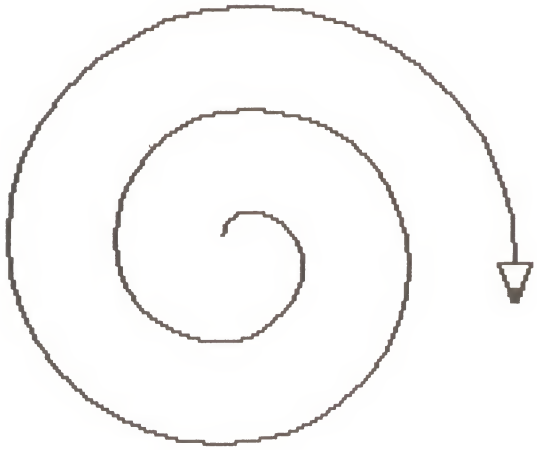


Figure 2.27: A spiral made from bigger and bigger RARC commands.

HELPER'S HINT

Circles and arcs are a good medium for exploring symmetry. There is a general principle at work here. Any turtle drawing can be made into a symmetrical one by interchanging all RIGHT and LEFT commands and then combining the original drawing with the reversed one. It's also possible to make other shapes with right and left-turning versions—RSQUARE and LSQUARE, RSTAR and LSTAR, etc. There will be more about this later in the book.

Section 2.6. More Turtle Commands

Here are a few more turtle commands that can be very useful. They are used in projects later in the book.

HIDETURTLE and SHOWTURTLE or HT and ST
hide and show the turtle without changing its position. A turtle that is hidden doesn't "spoil" a picture:

a **b**

Figure 2.28: A turtle drawing with turtle shown (**a**) and hidden (**b**).

Hiding the turtle *before* drawing a design makes the turtle draw faster. This is very helpful for designs with a lot of different lines in them, like circles. Try this:

```
CLEARSCREEN
RCIRCLE 50
CLEARSCREEN
HIDETURTLE
RCIRCLE 50
SHOWTURTLE
```

See how much faster the circle is drawn with the turtle hidden?

FENCE
forces the turtle to stay *within* the boundary of the screen. If you try to send it across the boundary, Logo will complain. Try this:

```
CLEARSCREEN
FENCE
FORWARD 200
```

Figure 2.29: The message given by Logo when the turtle is commanded to move too far in FENCE mode.

WRAP
brings things back to normal. Try this:

```
FENCE
FORWARD 200
WRAP
FORWARD 200
```

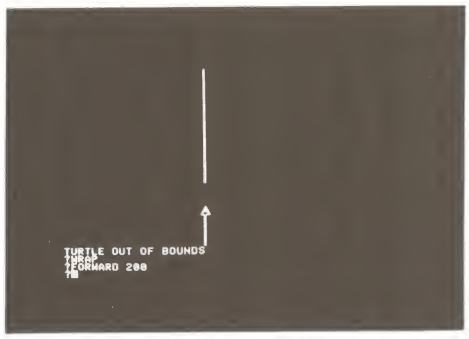


Figure 2.30: The turtle wraps around the screen normally after the WRAP command is given.

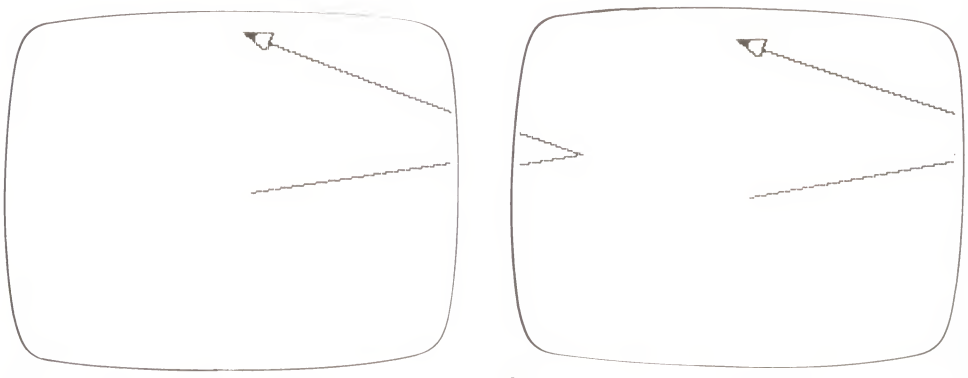
WINDOW
allows the turtle to go off the edge of the screen, but once it leaves the screen you can't see it any more. WINDOW makes the screen seem like a window. You can see anything that's right in front of the window, but you can't see anything beyond the window's edge. Try this:

```
CLEARSCREEN
WINDOW
RIGHT 80
FORWARD 180
LEFT 150
FORWARD 180
```

The result is shown in Figure 2.31a. The turtle has left the screen and returned to it, but you can see only the lines directly in front of the screen. Now try this sequence:

```
CLEARSCREEN
WRAP
RIGHT 80
FORWARD 180
LEFT 150
FORWARD 180
```

This time, the turtle wraps around to the left side of the screen and then back to the right again. The results are shown in Figure 2.31b.



a b

Figure 2.31: The results of the same series of commands in *window mode* **(a)** and *wrap mode* **(b)**.

CLEAN
clears the turtle's screen but leaves the turtle in place. To see the difference between CLEARSCREEN and CLEAN, move the turtle away and then type CLEAN.

```
CLEARSCREEN
RIGHT 30
FORWARD 20
CLEAN
FORWARD 20
```

Figure 2.32: A sequence of commands showing the effect of the CLEAN command.

HOME
sends the turtle to the center of the screen, heading straight up. The combination of HOME and CLEAN does exactly the same thing as the CLEARSCREEN command.

```
CLEARSCREEN
FORWARD 50
HOME
CLEAN
```

Try reversing CLEAN and HOME and see what happens.

```
CLEARSCREEN
FORWARD 50
CLEAN
HOME
```

Can you see why there is a line left on your screen?

The last set of commands I'll show you in this chapter change the way the screen looks.

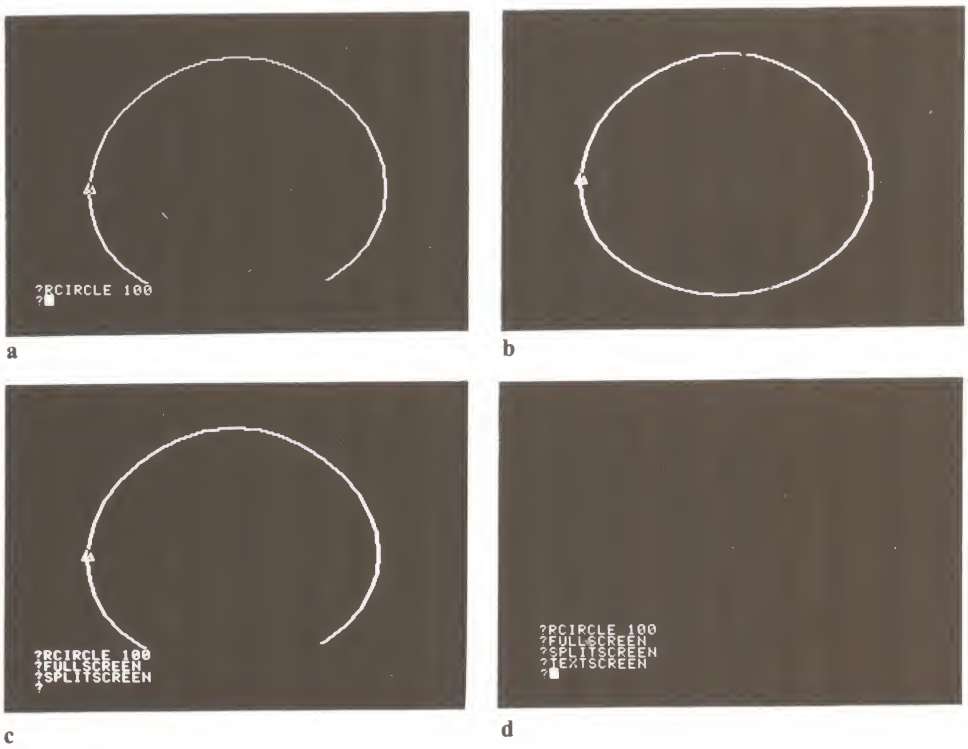
CTRL-L or FULLSCREEN
allows you to see the full turtle screen but hides anything typed or printed on the screen.

CTRL-S or SPLITSCREEN
restores the split screen. You can see four lines of print, but part of the turtle's drawing may be hidden.

CTRL-T or TEXTSCREEN
shows you the full text screen. You can't see any of the turtle's drawing, even though it is still there. To see the turtle again type **CTRL-L** or **CTRL-S**.

Type these commands and watch what happens.

```
CLEARSCREEN
PENUP
LEFT 90
FORWARD 100
RIGHT 90
PENDOWN
RCIRCLE 100
CTRL-L
CTRL-S
CTRL-T
CTRL-S
```



a b c d

Figure 2.33: The screen as it appears after the FULLSCREEN **(b)**, SPLITSCREEN **(c)**, and TEXTSCREEN **(d)** commands.

One nice thing about **CTRL-L**, **CTRL-S**, and **CTRL-T** is that you can use them at any time, without pressing **RETURN**. If you use FULLSCREEN, SPLITSCREEN, or TEXTSCREEN, however, you must use **RETURN** in the normal way. If you type a turtle command such as FD 50 while in *textscreen mode*, the computer will immediately shift to *splitscreen mode*.

HELPER'S HINT

The HOME command always moves the turtle to the center of the screen, regardless of where it was before. This behavior is different than that of commands like FORWARD, BACK, RIGHT, and LEFT, which are based on the turtle's current position.

Logo has other commands that, like HOME, refer to *fixed* positions on the screen. Commands SETX, SETY, and SETPOS move the turtle to a new position on the screen based on a *Cartesian* X and Y coordinate system. Other commands, XCOR and YCOR, output the values of the turtle's current X and Y positions. SETHEADING changes the turtle's heading to a fixed value, and HEADING outputs the value of the turtle's current heading.

Although these commands can be useful for certain purposes, they have been left out of this chapter because the most powerful ideas in turtle geometry come from learning to use the *relative* turtle commands. Fixed coordinate turtle commands are introduced when they are needed for specific activities in later chapters. HEADING is first used in connection with *conditional* commands in Chapter 7. SETHEADING, SETPOS, and Cartesian coordinates are introduced first in Chapter 10, where they are needed for the SHOOT game.

CHAPTER 3

New Commands: none
LWAL Procedures Disk files used: "SHOOT, "QUICKDRAW

3 Special Turtle Activities: SHOOT and QUICKDRAW

In this chapter you will learn to play a game called SHOOT and use a drawing tool called QUICKDRAW. They will help you understand more about using the turtle. SHOOT helps you learn about angles and distances. QUICKDRAW lets you draw interesting designs with the turtle very easily. SHOOT and QUICKDRAW are complicated Logo programs that you will learn about later in this book. To *use* the programs, all you have to do is *read* them from your LWAL Procedures Disk.

If you do not have a complete LWAL Procedures Disk yet, you can ask an adult or an older friend who knows more about Logo to help you by copying the procedures for you from Appendix I. If you don't have someone to help you right now, you can read this chapter quickly and come back to it later, after you have read about the procedures for SHOOT and QUICKDRAW in Chapters 10 and 11.

HELPER'S HINT

SHOOT and QUICKDRAW are activities that should be used at an early stage of Logo learning. In fact, they could be used by very young children to explore the world of the turtle in a structured way. They are examples of what Seymour Papert calls *microworlds* in his book *Mindstorms*. Microworlds are small learning environments which are fun for learners to use and have important skills and powerful concepts embedded in them. An important part of the art of being a successful Logo teacher is to be able to identify and create such environments or to help learners create their own. Not all microworlds involve writing programs. You have already encountered several in the exploration sections of this book.

If you want to provide these experiences for the people you are helping, you will have to either order a copy of the LWAL Procedures Disk from the address given in Chapter 0 or carefully copy the exact procedures given in Appendix I. If you decide to copy the procedures, you should first read Chapter 10 and Chapter 11, in which they are thoroughly explained.

Section 3.1. SHOOT: An Interactive Turtle Game

Before you can play SHOOT, you will have to clear the computer's working memory by typing

ERALL **RETURN**

Then you will have to *load* the game procedures from an LWAL Procedures Disk file called "SHOOT. Put the disk into the disk drive and type

LOAD "SHOOT **RETURN**

After the disk drive clicks and whirs for a while, the logo ? prompt will return.

Here's how the game is played:

1. Type

   START **RETURN**

   The computer will draw a target somewhere on the TV screen and place the turtle somewhere else on the screen. Every time the game is played the computer will put the target and the turtle at different points.
2. Aim the turtle toward the target using RIGHT and LEFT commands.
3. When you think the turtle is pointing directly at the target, type

   SHOOT **RETURN**

4. The computer will ask HOW FAR? You type a number. Then the computer will move the turtle the distance you type and show whether you have hit the target. If you miss the target, the turtle goes back to its original starting point and you can try aiming and shooting the turtle again. That's all there is to it.

I'll show you how one game that I played worked out. I typed

START

The computer placed the target and the turtle on the screen as shown in Figure 3.1.

Figure 3.1: The target and the turtle, after typing the START command.

Next I aimed the turtle. I had to *estimate* how far to turn it. I turned it to the right 100 degrees by typing

RIGHT 100

Figure 3.2: The turtle turned right 100 degrees from its starting position.

It looked close, but not quite right, so I turned the turtle a little more by typing

RIGHT 10

Figure 3.3: The turtle turned 10 more degrees to the right.

Now it looked like it was heading straight at the target, so I was ready to try a shot. I typed

SHOOT

and the computer printed

HOW FAR?

I estimated the distance and typed the number

100

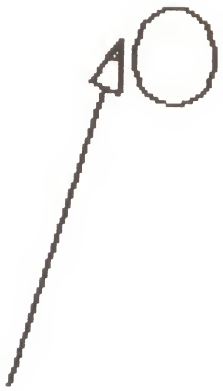


Figure 3.4: The results of the first shot.

My shot carried the turtle near the target, but not quite to it. The turtle drew a line and returned to its original position. I had to aim and shoot all over again. From the line drawn by the turtle, I could see that my first shot had been *too short* and aimed just *a little too far to the left*. Since I had turned the turtle right 110 degrees the first time (100 + 10), I needed to turn it a little bit *more* to the right the second time, so I typed

```
RIGHT 120
```

Figure 3.5: The turtle turned right 120 degrees from its starting position.

I also needed to shoot just a little *farther* than 100, so I tried 125.

```
SHOOT
HOW FAR?
125
```

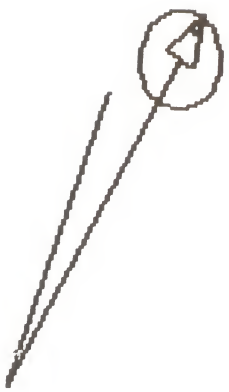


Figure 3.6: A hit!

CONGRATULATIONS! YOU HIT THE TARGET.
IT TOOK YOU ONLY 2 SHOTS.

And that's all there is to playing SHOOT. Try it a few times.

HELPER'S HINT

SHOOT provides a different way of helping someone learn Logo. Research with Logo at MIT has shown that some people—young children in particular—need a lot of practice estimating distances and angles before they can fully control the turtle and make it do what they want. SHOOT focuses on these "basic skills" of turtle control.

A Logo learning environment can and should include activities carefully designed to offer practice in skills which are themselves critical for success in Logo. Think of it as an appropriate form of "computer-assisted instruction" that can easily be modified or extended by a parent or teacher. Anyone who understands Logo can modify the program at a child's request. Seeing someone modify an advanced program can give a child a richer sense of what programming is all about. It is one way of giving a child some control over the computer, even though a more experienced programmer is needed to exercise that control.

I can image a nice synergistic project for a team of learners—one being an older, more experienced programmer, anywhere from age ten to adult, the other a less experienced learner, from age five or six on up. Starting with a game like SHOOT, they could work together to modify it and make it more interesting to both of them. This could be a great project for a class of sixth or seventh graders working with a class of second or third graders. Some suggestions for modifying SHOOT are given in Chapter 10.

Playing SHOOT also offers an opportunity to introduce or reinforce the activity of *playing turtle*. This form of SHOOT can be played outdoors or in a large room, with a real person taking the part of the turtle and a circle drawn on the ground for the target. Every element of the game can be simulated as a physical activity. The person playing the turtle can be spun around as in "blind man's bluff" to simulate the randomness of the turtle's starting point. Then other players can give right and left commands to the turtle and tell it how many steps to take to hit the target.

As described in the *helper's hint* in Section 2.3, playing turtle in this way requires that everyone agree on the words and numbers to be used to command the turtle. It will probably take some practice before everyone agrees on how the game should be played with *people*. The important thing is not to get the two games to be identical, but to set up a situation in which the simulated SHOOT game reinforces the learning involved in the SHOOT microworld, and vice versa, of course.

Section 3.2. QUICKDRAW: Drawing with an "Instant" Turtle

QUICKDRAW lets you draw with the turtle very easily by typing single keys, **F**, **B**, **R**, and **L**. It also lets you give a *name* to a turtle drawing so that you can redraw it any time you want.

Before you can use QUICKDRAW, you have to type ERALL to clear the computer's working memory. Then *load* a file called "QUICKDRAW from your LWAL Procedures Disk. (If you don't yet have a complete LWAL Procedures Disk, read the beginning of this chapter to find out what to do.)

```
ERALL   RETURN
LOAD "QUICKDRAW   RETURN
```

QUICKDRAW lets you command the turtle using only four keys. Here's how it works. To start, type

```
QD   RETURN
```

Then to draw with the turtle you can use single keys. Do not type **RETURN**.

F moves the turtle *forward* 20 turtle steps.
B moves the turtle *back* 20 turtle steps.
R turns the turtle *right* 30 degrees.
L turns the turtle *left* 30 degrees.

To stop drawing and give your picture a name, type

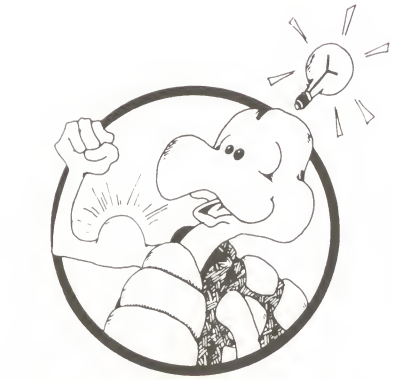
E

The computer will then print

```
PLEASE CHOOSE ONE WORD AS A NAME
FOR THIS DRAWING
TO FORGET IT, JUST TYPE RETURN
```

You type a name—any one word name you like—for this drawing. Then press **RETURN**. The computer will remember the name of your drawing.

Try this. Make a silly drawing using **F**, **B**, **R**, and **L**. When your drawing is finished, type **E** and give your drawing the name SILLY.



POWERFUL IDEA

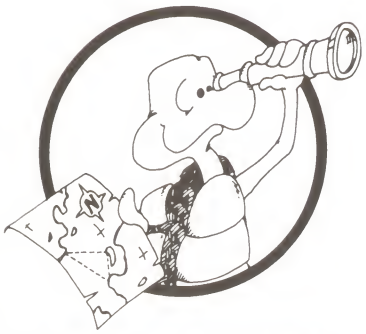
You can pick *any name you want* for your drawing. I just chose the name SILLY for this example. I could have called it DAN (that's my name) or HARRIET (maybe that's your name) or BOX (if it was shaped like a box) or even XQD999 (if I wanted to be really silly).

To *redraw the drawing named* SILLY, just type

RD :SILLY **RETURN**

PITFALL

The : symbol is very important in Logo. RD :SILLY tells the computer to *redraw the drawing whose name is* SILLY. If you forget to type the : in :SILLY or leave a space after the : (: SILLY), Logo will complain.



EXPLORATION

You can use QUICKDRAW to draw almost any drawing or design that you can think of. Here are a few ideas to get you started.

- Make simple shapes in different sizes, like squares or triangles.

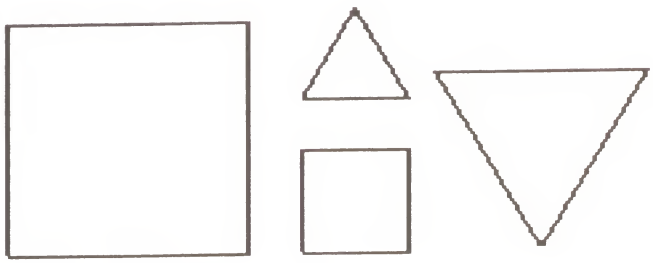


Figure 3.7: Squares and triangles made with QUICKDRAW.

- Make a design by turning the turtle and redrawing a shape. Repeat this over and over until your design is complete.

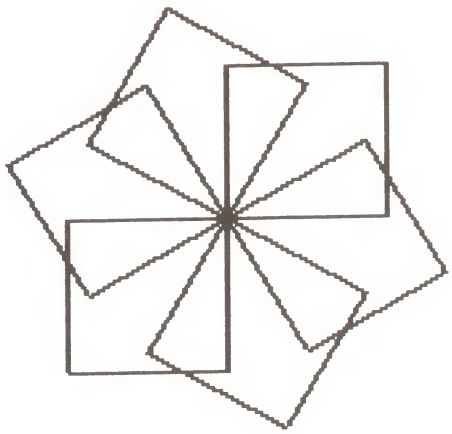


Figure 3.8: A design made by repeatedly drawing squares and turning the turtle.

- Use PENUP and PENDOWN to redraw the shapes in different parts of the screen. Put shapes together to make a cartoon face.

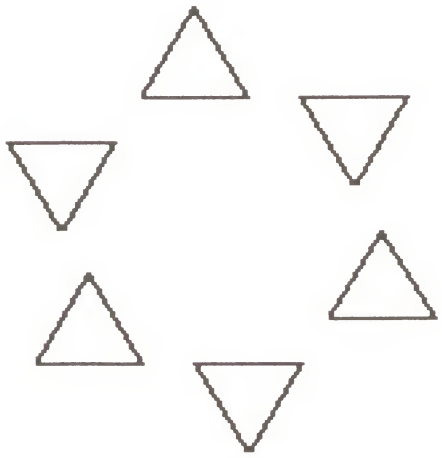


Figure 3.9: Shapes drawn on different parts of the screen.

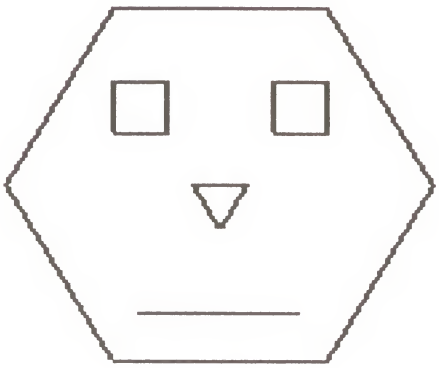


Figure 3.10: Shapes arranged into a cartoon face.

- Make a random shape *without planning it*. Give it a name and then redraw it several times in a row. You can make some really interesting designs this way.

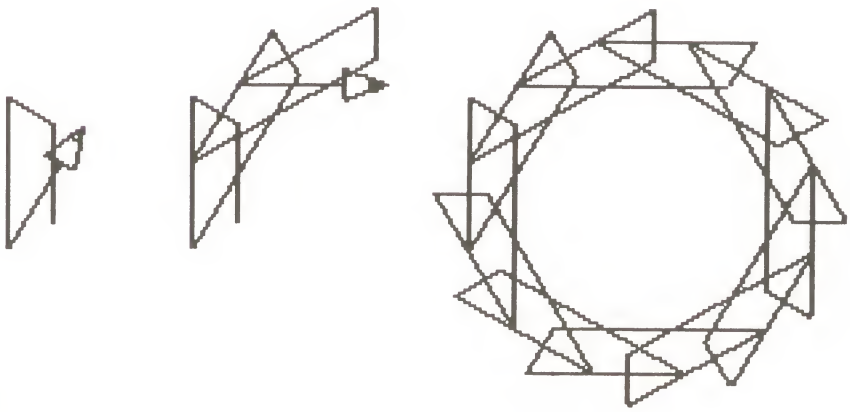


Figure 3.11: A random shape and the design made by repeating it.

- Look ahead to some of the drawing ideas in Chapters 5 and 6. See if you can draw them using QUICKDRAW.

HELPER'S HINT

The QUICKDRAW microworld contains—in limited form—many of the major concepts involved in learning with Logo:

- Controlling the turtle—making it move forward and back and turn right and left.
- Naming a picture and using that name to redraw it.
- Building a complex shape out of simple building blocks.
- Repeating a design until it closes.

QUICKDRAW is introduced here for three reasons. First, it is a learning environment for young children who may not be ready to work with the more complex Logo command structure. The procedures names, QD for QUICKDRAW and RD for REDRAW, are kept short on purpose to minimize any typing. If a family includes children with a range of ages and abilities, QUICKDRAW allows the younger ones to be involved in the same kinds of activities as the older ones. Second, it is a quick way to introduce some turtle drawing ideas that will be developed much more fully in Chapters 5 and 6. Third, it is an example of an intermediate Logo project which is explained in detail in Chapter 11. (Chapter 11 also contains a number of suggestions for extending and modifying QUICKDRAW.)

If you are helping a very young learner—someone between three and six, perhaps—for whom the full Logo command structure is too complex right now, you might want to use the extended version of QUICKDRAW given in Chapter 11. Another alternative might be to use the INSTANT program described in Harold Abelson's *Apple Logo*.

QUICKDRAW makes an excellent collaborative project for a younger and an older learner or a teacher to work on together. The younger learner could suggest ideas for improving QUICKDRAW, and the older learner or teacher could implement them (after he or she has read Chapter 11).

I want to add a couple of cautionary notes here. QUICKDRAW is *not* intended to be any kind of "ultimate" Logo environment for young children. First of all, I think such an environment is best created by children and adults together, as described above. Also, making QUICKDRAW more elaborate would have conflicted with my goal of using it as an intermediate project, as a model of how to begin the process of creating a microworld.

QUICKDRAW is just powerful enough for a learner to do something meaningful as an introductory activity. But it is not so wonderful that someone will want to stay with it forever. I think it is important to go on from QUICKDRAW to Logo itself. It is designed to be an appetizer but not an entire meal.

CHAPTER 4

Command	Short Form	Examples With Inputs
TO		TO BOX
END		
EDIT	ED	EDIT "BOX, ED "BOX
PO		PO "BOX
POTS		POTS
POALL		
ERASE	ER	ERASE "BOX, ER "BOX
ERALL		
SAVE		SAVE "CIRCLES
LOAD		LOAD "CIRCLES
CATALOG		
ERASEFILE		ERASEFILE "OLDSTUFF
.PRINTER		.PRINTER 1, .PRINTER 0

LWAL Procedures Disk files used: "PRINTSCREEN.S, "PRINTSCREEN.G

New tool procedures used:

Tool Procedure	Examples
PRINTSCREEN	PRINTSCREEN, PS
PRINTSCREEN.E	PRINTSCREEN.E, PSE
PRINTSCREEN.BIG	PRINTSCREEN.BIG, PSB
PRINTSCREEN.BIG.E	PRINTSCREEN.BIG.E, PSBE

4 Teaching the Computer

A command that you teach the computer is called a *procedure*. In this chapter you will use the Logo screen editor to teach the computer new commands. You will also learn how to *save* procedures in a *file* on a Logo work disk and how to use a printer to make *hard copy printouts* of Logo procedures and pictures.

From now on you will be teaching the computer new commands all the time and saving them on a disk. After reading this chapter once carefully, you may need to read parts of it again as you use the rest of the book.

You will need a Logo work disk and your Logo journal. Appendix II tells you how to create a Logo work disk if you do not already have one.

Section 4.1. Teaching the Computer How to BOX

You probably already know how to make the turtle draw a square box on the TV screen. (If not, read Chapter 2 again.) Using a computer would get very boring if you had to type in a long series of commands every time you wanted it to draw a box. Logo helps you teach the computer new commands so that you don't have to do this. This is sometimes called *writing a computer program* or just *programming*. Let's *teach* the computer how to draw a box on its own.

First, draw a square with the turtle. You should see something like Figure 4.1 on your TV screen.

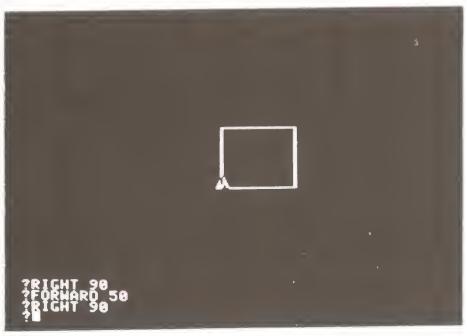


Figure 4.1: A simple square drawn by the turtle.

Pick a name for this shape. You can pick any name you like that is not already a Logo command. I'll choose BOX, because the shape looks like a box. You can pick any name you want: JOHN, CHRISTINA, MOMMY, ME, ET, R2D2, XQ34, or whatever. Now type the name you picked and press **RETURN.**

BOX **RETURN**

Logo will complain

I DON'T KNOW HOW TO BOX

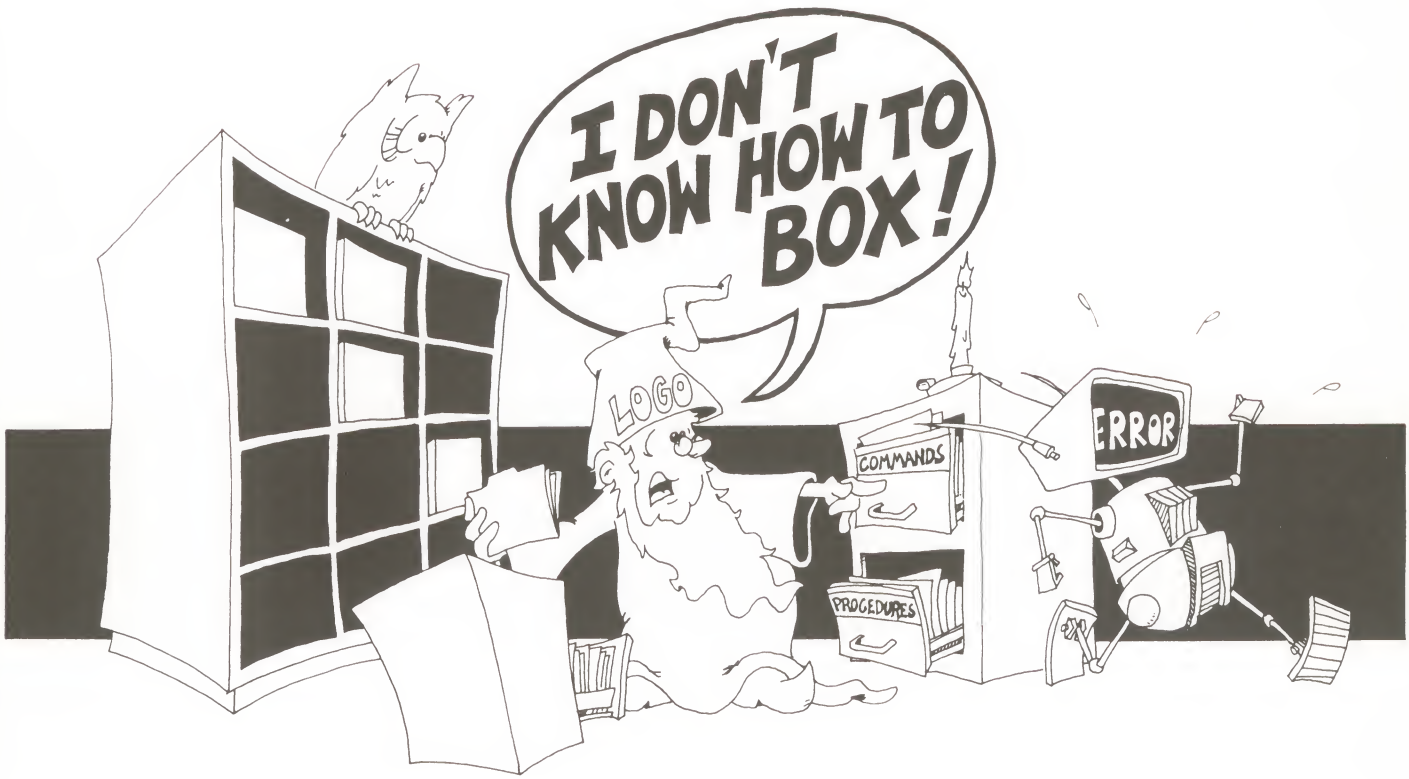


Figure 4.2: Logo complains that it doesn't know how to BOX.

You will have to *teach the computer* how to BOX. There are two ways to do this. The first way is to use the Logo command TO and type the following:

```
TO BOX
> FORWARD 50
> RIGHT 90
> FORWARD 50
> RIGHT 90
> FORWARD 50
> RIGHT 90
> FORWARD 50
> RIGHT 90
> END
```

The first line, TO BOX, tells the computer that you are teaching it a new command called BOX. The next eight lines are the *instructions* for how to BOX. The > symbol is a special *prompt* that the computer prints instead of the usual **?**. This reminds you that you are teaching the computer a new command. The last line, END, tells the computer that you are finished teaching it how to BOX.

Now let's work through the example step by step. Type each line and read the comments.

TO BOX **RETURN**

The computer will print this prompt:

>

and wait for you to type a command. It won't *do* anything, however. Instead, it will just store each command you type until you finish teaching it by typing END.

Now type the rest of the commands one at a time.

> FORWARD 50 **RETURN**
> RIGHT 90 **RETURN**
> FORWARD 50 **RETURN**
> RIGHT 90 **RETURN**
> FORWARD 50 **RETURN**
> RIGHT 90 **RETURN**
> FORWARD 50 **RETURN**
> RIGHT 90 **RETURN**
> END **RETURN**
BOX DEFINED

After you type the command END, the computer knows you are finished and prints the message, **BOX DEFINED**.

BOX is now a Logo command, just like the others you already know. To use the command, type

BOX **RETURN**

A small square just like the one in Figure 4.1 should appear on the screen. If BOX does not do what you expected, you may have made a typing mistake. If so, you can start over and teach the computer to BOX again. Or you can *edit* BOX, using the Logo screen editor.

Teaching a New Command Using the Logo Screen Editor

The second way to teach the computer new commands is using the Logo screen editor. Typing mistakes are easy to fix with the screen editor. Let's work through the same example, using the editor.

First, erase the BOX procedure by typing

ERASE "BOX **RETURN**

To start using the editor, type

EDIT "BOX **RETURN**

PITFALL

When you type ERASE "BOX or EDIT "BOX, be sure to type a " symbol *before* the word BOX, but *not after* it. If you type EDIT BOX and forget the ", or leave a space after the " (EDIT " BOX), Logo will complain.

When you type EDIT "BOX, the computer will clear the screen and enter *edit mode*, as shown in the figure. When the computer is in edit mode, you can type in anything you like. The computer just stores the information and doesn't *do* anything until you leave the edit mode.

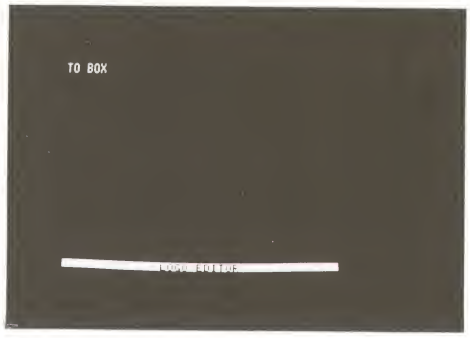


Figure 4.3: The computer screen as it appears in edit mode.

TO BOX appears at the top of the screen. Now type the rest of the commands, one at a time.

FORWARD 50 **RETURN**
RIGHT 90 **RETURN**
FORWARD 50 **RETURN**
RIGHT 90 **RETURN**
FORWARD 50 **RETURN**
RIGHT 90 **RETURN**
FORWARD 50 **RETURN**
RIGHT 90 **RETURN**
END **RETURN**

The screen will now look like the one in Figure 4.4.

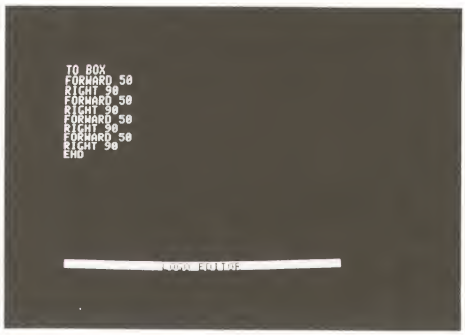


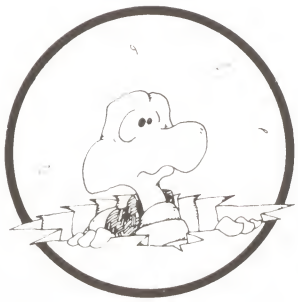
Figure 4.4: The screen as it appears after the BOX procedure has been typed in edit mode.

Use the arrow keys and **CTRL-B** key to correct any typing mistakes.

Even though you just typed END, you haven't quite finished teaching the computer how to BOX. To *leave* edit mode, type **CTRL-C**. (Hold down the **CTRL** ("control") key and then press the **C** key.) The computer will now print

BOX DEFINED

Figure 4.5: The screen as it appears after BOX has been defined.



PITFALL

It is very common for people to forget to type **CTRL-C** when they are finished teaching the computer. Any commands you type while in edit mode will not be carried out until you return to *command mode* by typing **CTRL-C**.

The illustrations in Figure 4.6 show what happens when you type EDIT "BOX.

Figure 4.6a: Logo calls on EDIT and tells it the name of the new command, BOX.

Figure 4.6b: EDIT stores all the commands you type and any changes that you make.

Figure 4.6c: When you type **CTRL-C**, EDIT gives the completed procedure back to Logo.

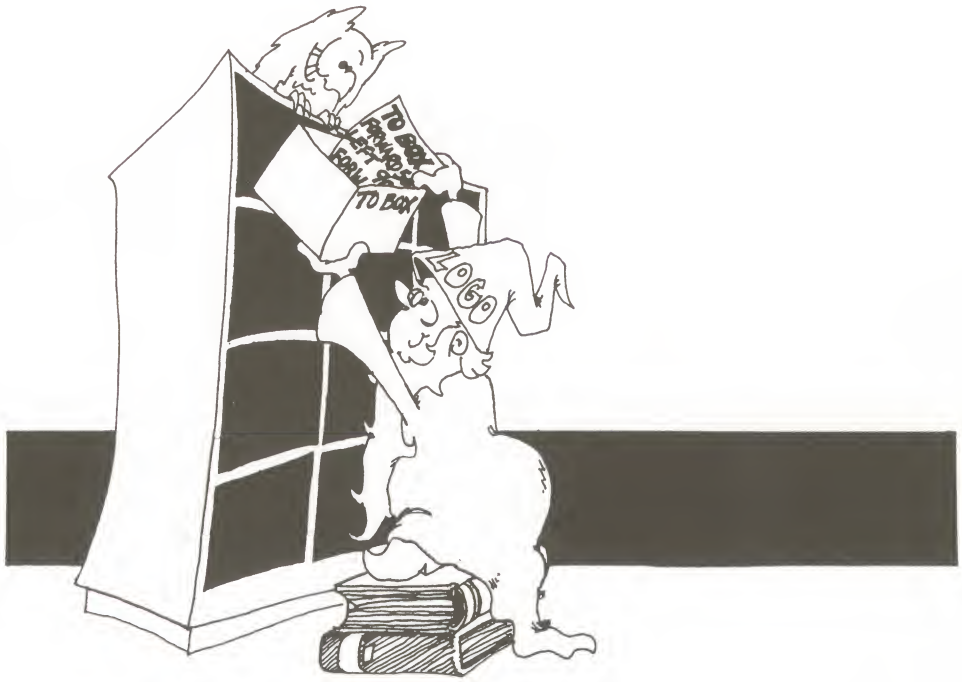


Figure 4.6d: Logo stores the instructions where they can easily be found again.

Figure 4.6e: When you type the command, BOX, Logo calls BOX, giving it the new instructions.

After leaving edit mode by typing **CTRL-C**, you will be back in Logo *command mode*, the normal mode in which Logo carries out the commands you type. Now type the command BOX. You should see this drawing on the screen:

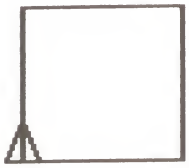


Figure 4.7: A box drawn by typing the new command BOX.

If your procedure does not do what you expected, type

EDIT "BOX **RETURN**

This returns you to edit mode. Use the arrow keys and **CTRL-B** to correct any errors. Type **CTRL-C** when you have corrected all your errors. Now type the command BOX again.

BOX is now a Logo procedure and can be used just like any other Logo command. You have begun to create your own computer language!

If you want to SAVE the box procedure on a disk, Section 4.3 of this chapter will tell you how. Chapter 5 shows how you can use BOX as a *sub-procedure* in teaching the computer more new commands. You can use BOX as a basic shape to make many other designs, as shown in Figure 4.8.

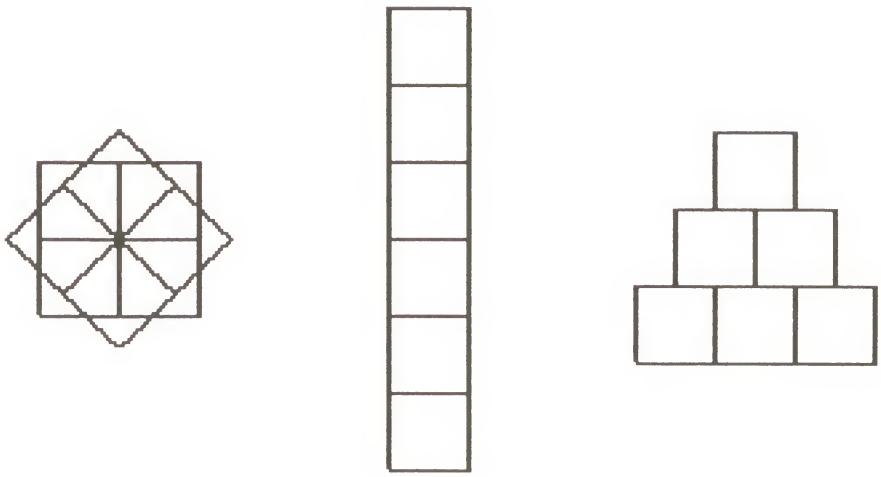


Figure 4.8: Designs made using the BOX procedure.

Because some people get confused about the difference between *command mode* and *edit mode*, Apple Logo has provided two ways of teaching the computer a new command. For these learners, it may be easier to start using TO and to type each command without having to think about the edit mode. However, people should move on to the editor as soon as possible.

For most other versions of Logo, TO and EDIT do exactly the same thing. They shift the computer to edit mode. When I teach Apple Logo, I prefer to use edit mode and the EDIT command from the start, even if it is a bit more difficult to use the " symbol and to learn a strange new word, EDIT. There are two reasons for this: (1) beginners often make typing errors and have to start all over or switch to the edit mode in order to correct them; (2) using the editor is one of the most important parts of using Logo and so it helps to start using it right away. I've seen children as young as 5 or 6 learn to use the editor.

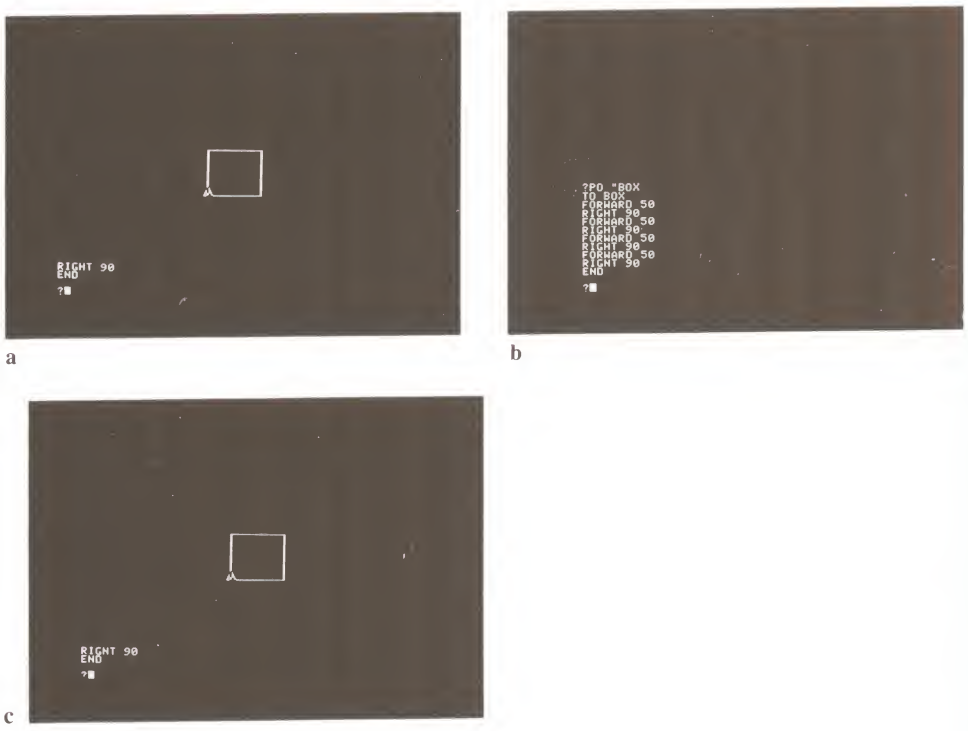
Ways to Examine Your Work

You can print a list of all the steps in the procedure TO BOX by typing

```
PO "BOX
```

PO stands for *printout*.

This will show the procedure on the screen. If you can't see the entire procedure printed on the screen, type **CTRL-T**, which shows you the entire text screen. Later, type **CTRL-S** to return to the normal split screen.



a b c

Figure 4.9: If you can't see the entire procedure after typing PO "BOX **(a)**, type **CTRL-T** to see the entire text screen **(b)**; then return to a normal split screen by typing **CTRL-S** **(c)**.

To printout *all* your procedures, type

```
POALL
```

The Logo command POTS will print out the *titles* (names) of all your procedures. POTS is the short form of PRINTOUT TITLES.

If you want to ERASE a procedure completely, you can type

```
ERASE "BOX
```

or

```
ER "BOX
```

You can also erase all your procedures from the computer's memory by typing

```
ERALL
```

PITFALL

Be careful. Never type ERALL unless you have saved your procedures on a disk or unless you are really and positively sure you don't want them. Section 4.3 tells how to *save* your procedures on a disk.

Section 4.2. Using the Logo Screen Editor

The Logo screen editor gives you an easy way to type in a new procedure or change an old one. It may take you a little while to get used to using the editor because it works a little differently than Logo command mode.

The Basics of Editing

There are a few special words that make it easier to talk about how the Logo screen editor works. The letters, numbers, and symbols that you type on the screen are called *characters*. Everything typed on the screen at any one time is called the *text*. Remember that the *cursor* shows you where the *next* character will be typed on the screen. Four *direction keys* let you move the cursor to any point in the text. You can do all the editing you need with the four direction keys, the **REPT** key, the **RETURN** key, and the **ESC** key.

Direction Keys to Move the Cursor

→ or **CTRL-F**
moves the cursor *right* one space.

CTRL-B
moves the cursor *left* one space.

CTRL-P
moves the cursor *up* one line. ("P" stands for "previous line.")

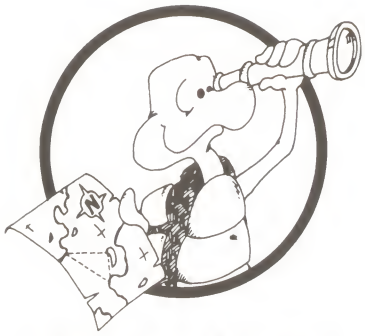
CTRL-N
moves the cursor *down* one line. ("N" stands for "next line.")

Basic Editing Keys to Change the Text

←
erases the character to the left of the cursor and *backspaces* the cursor.

RETURN
creates a *new line* and moves the cursor down to the beginning of that new line.

REPT
if you have an Apple II plus, *repeats* the effect of the key you just typed. For example, to move the cursor across the screen to the right, first press →, then keep holding the → key down while pressing the **REPT** key. If you have an Apple IIe, holding any key down for a while will cause it to repeat.



EXPLORATION

Practice using the editor. Type EDIT "BOX and practice making changes to the BOX procedure. Don't worry about messing it up—you can always fix it later. Try some of the following suggestions.

- Change one of the lines in BOX. Leave the editor by typing **CTRL-C**. Try out the new BOX procedure to see what happens, then edit BOX again and return it to the way it was before.
- Change all the inputs to FORWARD from 50 to another number. Try the procedure and see what happens.
- Change all the angles in BOX from 90 to another number. Try the procedure and see what happens.
- Change the *name* of the procedure from BOX to something else. Type the new name and see what happens. Type BOX to see what happens. Then type POTS to see the names of all the procedures you have taught the computer.
- Edit BOX again to make it just as it was before you started experimenting with the editor.

HELPER'S HINT

Using the editor can be very tricky for any beginner. The most important thing to understand is the function of the cursor. Anything typed on the keyboard will be inserted in the text *at the cursor* and everything else will be moved over to make room for it. If something is erased, the cursor will move back, and all the text to its right will be moved back with it.

There are two kinds of operations involved in editing.

1. *Moving the cursor without changing the text.*
2. *Changing the text* by erasing something, typing new characters, or creating a new line. Changing the text also moves the cursor.

Most beginners' *editing bugs* come from confusing these two kinds of operations or not understanding the function of the cursor. The following are some problems beginners often run into:

- Using **SPACE BAR** instead of → to move the cursor to the right, or trying to use → to insert a space.
- Using **RETURN** instead of **CTRL-N** to move down one line. This happens because people have the habit of typing **RETURN** at the end of every line. When you do this while editing, you add a new line and move all the other lines down. When this happens by accident it can be fixed immediately by pressing ← once.
- Not knowing where to put the cursor to erase or insert something.

People familiar with the computer language BASIC often experience another difficulty. In BASIC, typing over an old line replaces it with new text. In Logo, text is inserted when you type and the old text is moved over, not replaced.

Another common bug, unrelated to these, is forgetting to type **CTRL-C** to return to command mode, after editing.

Logo has a large collection of editing commands to facilitate creation and editing of text. However, I believe that people should learn as few as possible to begin with and then gradually learn others as they become familiar with the editor. All editing can be done with only four keys, → and **CTRL-B** to move the cursor, and ← and **RETURN** to change the text. Using → at the right end of a line moves the cursor down. Using **CTRL-B** or ← at the left end of a line moves the cursor up.

All the other editing keys are ways of doing the same things faster and more conveniently. I show people how to use **CTRL-N**, **CTRL-P**, and **REPT** as soon as they learn the first four. Other editing keys can be posted on a chart near the computer, and people can begin using them whenever they feel comfortable with what they already know.

I also use some special terminology (jargon) to talk about editing. Words like *character*, *text*, *cursor*, *edit mode*, and *command mode* help people make distinctions and think more clearly about what they are doing, even though these words may seem strange at first.

More About Using the Editor

There are several more editing keys that can make editing easier and faster. Until you can easily use the basic editing keys, you should probably skip ahead to Section 4.3, which teaches you how to save procedures on a disk. When you want to learn *more* about editing, come back and read the rest of this section.

More Editing Keys to Move the Cursor

CTRL-E
moves the cursor to the *end* of a line.

CTRL-A
moves the cursor to the *beginning* of a line.

CTRL-V
moves the cursor *forward an entire screen* when the screen is full of text.

ESC-V
moves the cursor *backward an entire screen* when the screen is full of text. (Type **ESC** first; then, release it and type **V**.)

More Editing Keys to Change the Text

CTRL-D
erases the character at the cursor and does *not* backspace. "D" stands for *delete*.

CTRL-K
erases the entire line to the right of the cursor. "K" stands for *kill*.

CTRL-O
creates a new line at the cursor and moves all the rest of the text down one line. "O" stands for *open* a new line.

Even More About Using the Editor

You can use the editor to teach the computer several procedures at once. After typing END, go on to teach it something else. After each new procedure, type END. When you are all finished, type **CTRL-C**.

If you type EDIT **RETURN**, without entering a procedure name, the computer will return to edit mode with whatever was just edited on the screen. If this is the first use of edit after using the turtle, the screen will be blank.

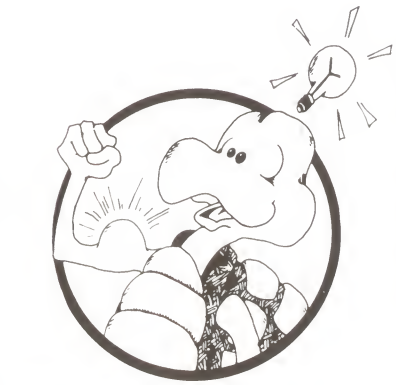
You can edit several procedures at once. If you want to edit three procedures called BOX, RACE, and STAR, you can type

EDIT [BOX FACE STAR] **RETURN**

If you want to learn even more about using the editor, read Harold Abelson's book *Apple Logo* or the Reference Manual that comes with your Apple Logo Language Disk.

Section 4.3. Saving Procedures on a Logo Work Disk

A computer has two kinds of memory. Its *working memory* contains everything it remembers *right now*. Its *permanent memory* is stored on disks and can be *loaded* into the working memory.



You can think of it this way—you have two kinds of memory, too. There are things you remember right now—that's your *working memory*. Then there are things you write down so that you can remember them later, or things you look up from a book. That's your *permanent memory*.

I can't remember more than a few phone numbers at any one time. When I want to remember a new phone number I write it down in a little booklet that I carry with me. There is also a printed phone book that has thousands of phone numbers that I can look up whenever I want to, but I can't ever change those numbers.

The telephone numbers in my head right now are my *working memory*. They are like a bunch of procedures that I have just taught the computer. The phone numbers in my little booklet are my own *personal permanent memory*. They are like procedures that I save on my own Logo work disk. The numbers in the public telephone book are *shared permanent memory*. I can use them, but I can't change them. They are like the instructions for Logo that I can load into the computer's working memory from the Logo Language Disk whenever I want to. The Logo computer language is a form of *shared permanent memory*.

Figure 4.10: Three different kinds of memory.

Once you have taught the computer some procedures, you will want to save them on a Logo work disk. If you do not already have a Logo work disk, Appendix II tells how to create one.

Logo has several special commands for storing information in *files* on a Logo work disk.

SAVE "SALLY
saves all the procedures in the computer's working memory on the Logo work disk in a file named "SALLY.

LOAD "SALLY
reads all the procedures from the file named "SALLY and puts them into the computer's working memory.

CATALOG
prints a list of all the files stored on the Logo work disk.

ERASEFILE "SALLY
erases the file named "SALLY from the Logo work disk. Never erase a file without being absolutely sure either that you don't need the information any more, that you have saved it somewhere else, or that you are about to save an improved version of that file.

Here's how filing works. First you choose a *name* for your file. A good file name to start with is your own name. You can save a lot of different procedures in that one file. To save procedures in a file, make sure your Logo work disk is in disk drive number 1 (if you have two) and the door is closed. See Figure 4.11.

Figure 4.11: Inserting the Logo work disk into the disk drive.

Then type:

POTS **RETURN**
to see the list of procedures that are about to be saved and then

SAVE "SALLY **RETURN**
if your name is Sally.

The red light on the disk drive will come on, and the drive should make soft clicking sounds. When you see a **?** prompt again, the computer will be ready for new commands.

Be sure to type the " symbol before the file name *without a space between the " and the name*. Do not type *" after the name as you do in ordinary English*. Type "SALLY, not " SALLY or "SALLY".

If this process does not seem to be working properly, make sure that the disk is correctly inserted in the disk drive, with the door closed, and that you were using a properly *initialized* Logo work disk. If you still have trouble, ask a more experienced person for help.

Now type the Logo command

CATALOG

The computer will print a list of all the *file names* on the disk. All of the files that contain Logo procedures will have ".LOGO" after their names.

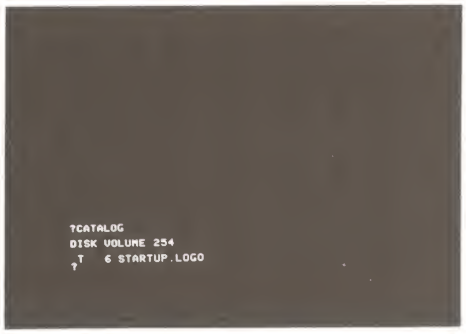
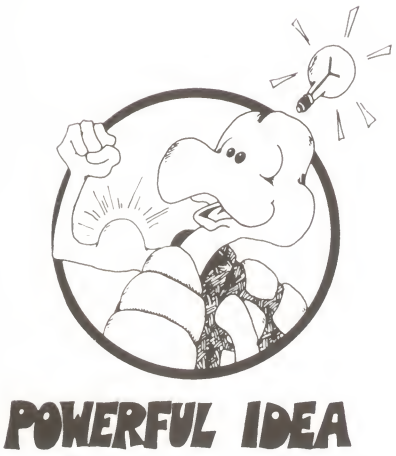


Figure 4.12: The screen showing file names in response to the CATALOG command.



A *file* on a disk is like a file folder in a file drawer. The folder holds many papers. Each paper has one procedure written on it. A file drawer can have many file folders in it, and each file would have a different name so that you can find it when you need it. In the same way that many different people can keep files in the same drawer, many different people can store files on the same disk.

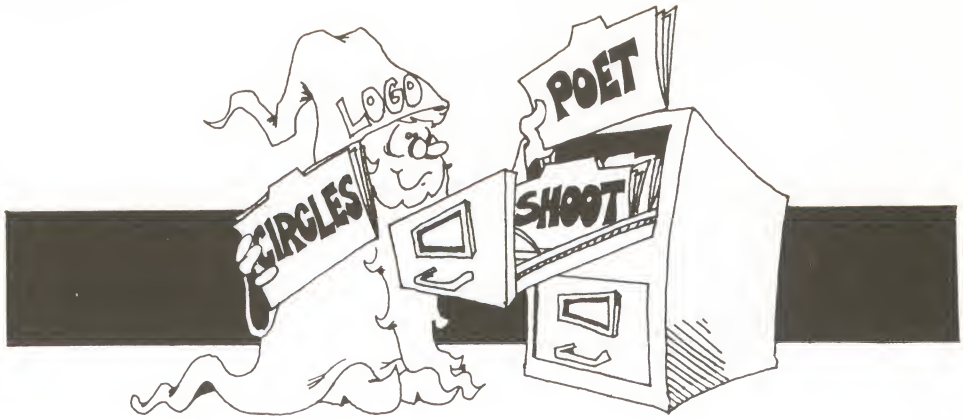


Figure 4.13: A Logo work disk is like a file cabinet filled with different file folders.

If your new file name appears on the screen when you type CATALOG, your procedures have been saved on the disk. You can now read them from the disk whenever you need them. Try typing this series of commands:

POTS
reminds you which procedures you just saved.

ERALL
clears the computer's working memory.

POTS
shows no list of procedures this time—you just erased them!

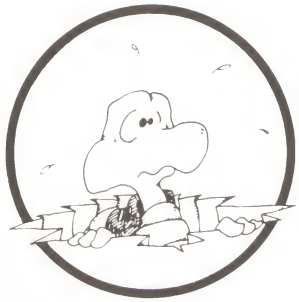
LOAD "SALLY
or whatever file name you used.

Now your procedures should be back in the computer's working memory. Find out for sure by typing

POTS

Whenever you start work, type LOAD to read the file you want to work on. At first, it should be the one with your own name. Here is a good procedure for you to follow:

1. When you start work type ERALL to clear the computer's working memory.
2. Read your file by trying LOAD "SALLY (or whatever file name you are using).
3. After teaching the computer a new procedure and testing it, type ERASEFILE "SALLY (using your own name, of course) to clear the old file from the disk. Then save your new procedures by typing SAVE "SALLY. In this way you will save all your procedures as you go along.



PITFALL

WARNING! If you don't LOAD your file when you start work, your old procedures may be accidentally erased when you save new ones in your files. If you load a file first, before saving anything, then your old procedures will be saved along with any new ones or any changes you have made.

When you are all finished working, type ERALL to clear the computer's working memory for the next person.

HELPER'S HINT

Filing procedures on a disk is a difficult process for some people to understand. I find it helpful to stress the idea of *two* kinds of memory. Short-term or *working* memory includes the things that Logo actively remembers at the present moment; long-term or *permanent* memory includes things that Logo doesn't actively remember but will recall when given the instructions detailed in this section.

I have found the analogies to the telephone book and the file cabinet very useful in helping learners of all ages understand filing. However, there is one major difference between computer files and someone's personal booklet of phone numbers. When you save procedures on a disk, the computer makes a copy of *everything* in the working memory and stores it on the disk. The contents of the working memory are not changed by saving it on the disk. When you load a file, you make a copy of the entire file and place it in the working memory. The information stays in the file also, until you save some other information by erasing the old file and using the same name. This is why it is so important to load the old file *before* starting work. In this way the working memory already contains the old information before any changes or any new procedures are saved.

Despite this kind of precaution, material does sometimes get lost. This is a good reason for keeping *hard copy*, that is, printed files as well. Section 4.4 tells how to print Logo information on paper with a printer. It's also a good idea to make a back-up copy of an entire disk from time to time. The process of copying disks is explained in Appendix II.

More About Filing

You can skip this part for now. Come back to it when your working memory begins to be filled up by procedures or you are ready to create separate files for separate projects.

Just as you would probably keep difficult kinds of papers in different file folders with different names, you can also store computer information in *many different files*. When you have taught the computer a lot of procedures, you may want to use more than one file name. Each file should contain a group of procedures that go together in some way. The group can contain all the procedures needed for a particular drawing (see Chapter 6) or a group of procedures that all make similar designs (see Chapter 5).

Once you begin to have separate files for different projects, you will probably want to keep them as separate as possible. Each time you begin work on a new project, choose a file name for it. Use that name every time you save that group of procedures. When you work on that same project again, start by loading the file before you save anything. Before starting to work on a different project, clear the working memory by typing ERALL. Then read the file for your next project before starting to work on it.

Appendix II explains even more about using Logo files. It tells how to save groups of procedures in "packages" and gives more suggestions about files and file names. To learn still more about Logo files, read Chapter 4 of Harold Abelson's book, *Apple Logo*, and the Reference Manual that comes with Apple Logo.

Section 4.4. Printing Procedures and Pictures with a Printer

If you have a printer connected to your Apple computer, you will be able to print your procedures on it and maybe even print copies of Logo turtle drawings.

Make sure your printer is turned on. The special command .PRINTER is used to send information to the printer. .PRINTER needs an input telling which *slot* the printer is plugged into. If it is plugged into slot 1, type

```
.PRINTER 1
```

This will start the printer. (If the printer were plugged into slot 2, you'd type .PRINTER 2, etc.). Normal Logo printing commands will now print things on the printer paper instead of on the TV screen.

```
PRINT [HELLO, HOW ARE YOU]
```
will print the sentence HELLO, HOW ARE YOU on the printer.

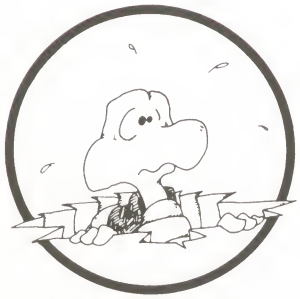
```
POTS
```
will print a list of all your procedures.

```
PO "BOX
```
will print out the commands in the procedure named BOX.

```
POALL
```
will print out all the commands in all your procedures.

```
CATALOG
```
will print a list of all the files on your disk.

```
.PRINTER 0
```
will turn the printer off and make the computer print on the screen again.



PITFALL

WARNING! If you use an incorrect input for .PRINTER, that is, a number that is *not* the correct slot for the printer, Logo will crash. To avoid losing all your procedures, it's a good idea to save them on a disk before using the printer.

Printing Pictures with a Silentype Printer

Some printers can print pictures directly from a TV screen. If you have a *Silentype* printer made by the Apple Computer Company, you can print pictures by typing

```
.PRINTER 1
PRINT CHAR 17
.PRINTER 0
```

The first line, .PRINTER 1, turns on the printer (assuming it is plugged into slot 1). The middle line, PRINT CHAR 17, is a "magic word" which tells the printer to print the screen picture. The last line, .PRINTER 0, turns the printer off.

If you load a file called "PRINTSCREEN.S from your LWAL Procedures Disk, you can use a procedure called PS (short for PRINTSCREEN). If you do not yet have a complete LWAL Procedures Disk, you can copy the procedures from Appendix I.

PS or PRINTSCREEN will print pictures as dark as possible and will *reverse* the screen image so that the pictures are printed as black lines on the white paper (rather than white on black as they are on the TV screen).

Printing Pictures with an EPSON Printer

The Logo illustrations in this book were made using an EPSON MX-80 printer with a *Grappler* interface board supplied by Orange Micro, Inc. If you have this equipment, you can print pictures using some procedures on the LWAL Procedures Disk. Read a file called "PRINTSCREEN.G from the disk or copy the procedures from Appendix I. The following eight procedures are in this file.

PS or PRINTSCREEN
will print a *regular* copy of the picture on the screen.

PSE or PRINTSCREEN.E
will print an *enhanced* or *darker* copy of the picture on the screen.

PSB or PRINTSCREEN.BIG
will print a *larger* copy of the picture on the screen. The picture will also be rotated to make room for it on the paper.

PSBE or PRINTSCREEN.BIG.E
will print a *big enhanced* copy of the picture on the screen.

HELPER'S HINT

Making hard-copy printouts of Logo procedures and pictures adds an important dimension to a Logo learning experience. Having printed procedures to paste in a journal or pictures to post on a bulletin board or pass around can make a huge difference in helping someone enjoy using a computer and understand what he or she is learning.

In a school, you don't have to have a printer with every computer. One printer in a school library is enough to allow many students to print out their work. If you don't have your own printer, it would be worth trying to borrow one. Some computer stores allow regular customers to use a printer in the store from time to time.

CHAPTER 5

Command	Short Form	Examples With Inputs
REPEAT		REPEAT 4 [FORWARD 20 RIGHT 90]

LWAL Procedures Disk files used: "CIRCLES
New tool procedures used: none.

5 Turtle Projects 1: Designs

In this chapter you will see examples of many different designs drawn by the turtle that you can copy or change. You'll also learn how to use procedures and subprocedures to invent your own.

This chapter can be read quickly for its ideas or worked through very slowly as a source of design ideas for many turtle projects. Don't be surprised if something is harder to do than it looks. At the same time, don't be afraid to try something that looks hard at first. Many designs are really a lot simpler than they look, if you build them using the right pieces.

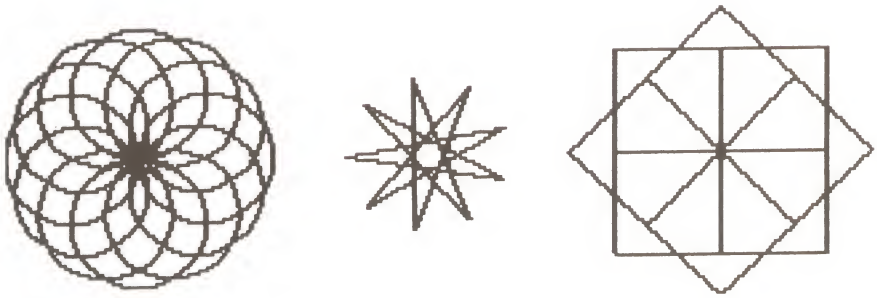


Figure 5.1: Sample designs drawn by the turtle.

Section 5.1. Procedures and Subprocedures

A command that you teach the computer is called a *procedure*. A procedure can be used just as if it were a built-in Logo command. When one procedure is used as part of another procedure, it's called a *sub*procedure. Look at the design made of squares in Figure 5.2.

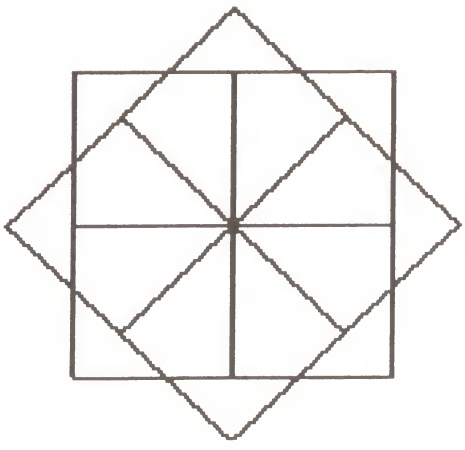


Figure 5.2: A "star" made of rotated "windows."

If you have already taught the computer how to draw a square, this can be a very simple project. If you haven't already taught the computer how to square, teach it this procedure now.

```
TO SQUARE
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
FORWARD 50
RIGHT 90
END
```

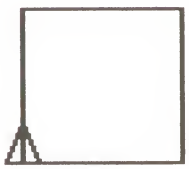


Figure 5.3: A square drawn by the SQUARE procedure.

The SQUARE procedure is made by repeating FORWARD 50 RIGHT 90 until the turtle gets back to where it started. SQUARE can now be used to build BOXES.

```
TO BOXES
SQUARE
LEFT 90
SQUARE
LEFT 90
SQUARE
LEFT 90
SQUARE
LEFT 90
END
```

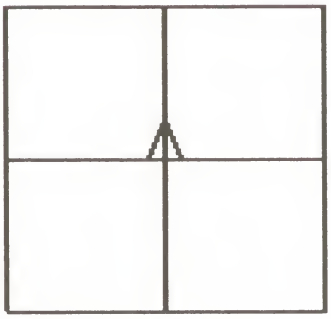


Figure 5.4: The design drawn by the BOXES procedure.

BOXES is made by repeating SQUARE LEFT 90 until the turtle gets back to its starting point. BOXES can now be used to make STAR.

```
TO STAR
BOXES
RIGHT 45
BOXES
END
```

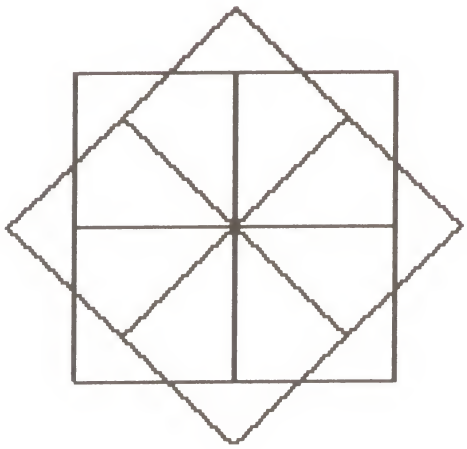
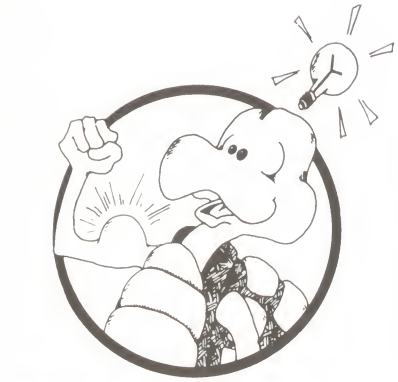


Figure 5.5: The design drawn by the STAR procedure using BOXES as a subprocedure.



Building things with subprocedures is one of the most important, useful, and powerful ideas you can learn with Logo. If you tried to make the turtle draw STAR *without* using SQUARE and BOXES as subprocedures, it would be a long, complicated project. Using subprocedures makes it easy to do and easy to understand. Because each subprocedure is short and because the names make it clear what each subprocedure does, it would also be easy for someone else to understand how to do this.

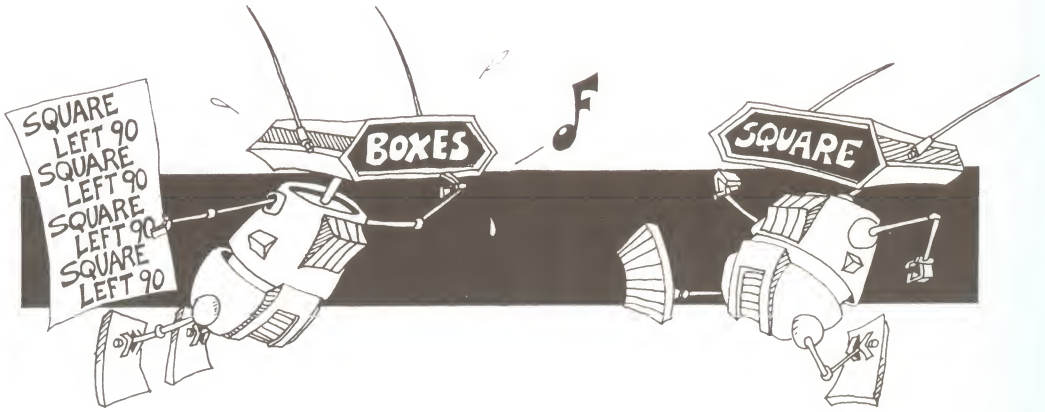


Figure 5.6a: Using subprocedures makes complicated designs simpler.

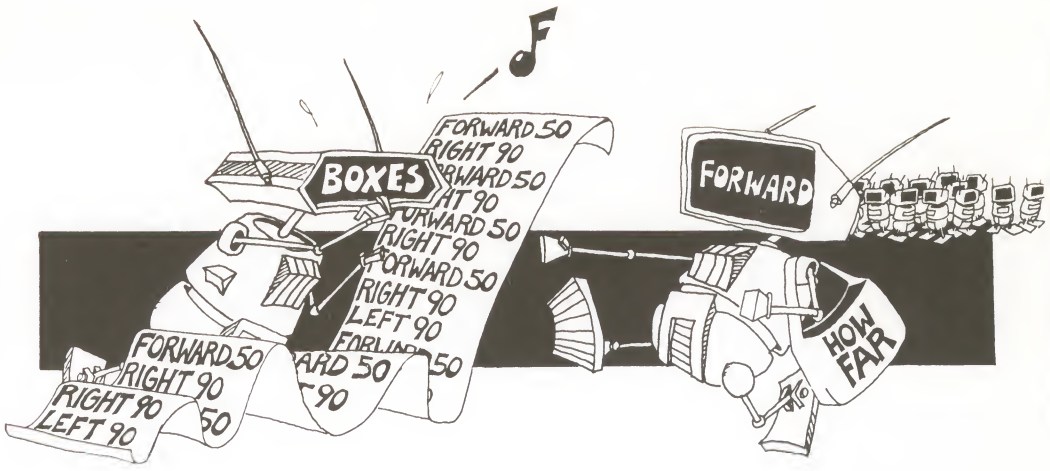
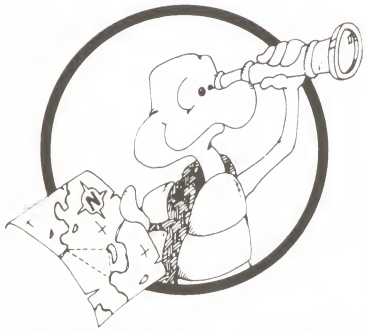


Figure 5.6b: Without subprocedures, the procedure would be more confusing.



Many other shapes that look hard at first can be drawn easily by using subprocedures. See if you can use squares to make some designs like this:

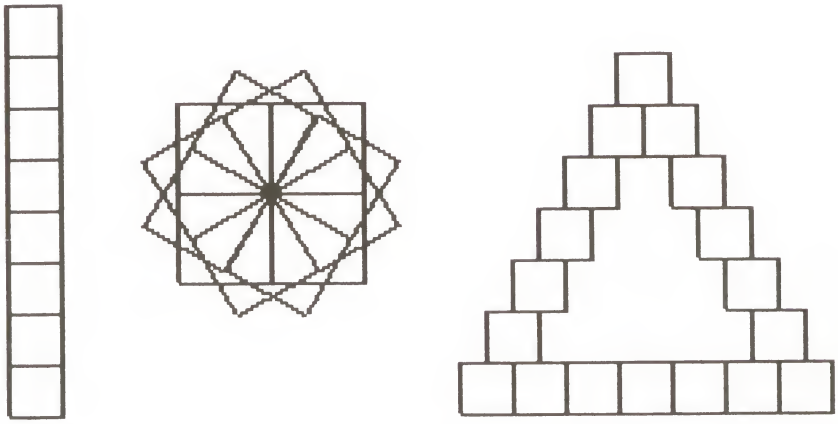


Figure 5.7: Designs made with squares.

Many of these designs can be drawn even more easily using REPEAT or *recursion*, which are introduced in Section 5.3 and Section 5.4. They are shown here without those techniques in order to help build a rationale for them. If learners ask for an easier way to make the computer repeat things, you could refer them directly to those sections.

The concept of a *subprocedure* is one of the most important ideas in Logo and is part of what makes Logo so much easier to learn than many other computer languages. As a programmer you create your own language and use the commands you create to make the computer do new things that would have been much more difficult without your specific language. Since each procedure is an independent entity, you can use it as part of many different projects. You can even use a procedure as a *subprocedure of itself*. This may sound paradoxical, but it turns out to be a very powerful idea that makes lot of programs easier. See the description of *recursion* in Section 5.4.

Section 5.2. Regular Shapes

A *regular* shape is one with all its sides and angles equal. A square, for example, is a regular shape. To make a square with the turtle, you make it repeat FORWARD 50 RIGHT 90 until it is back to its starting place. By using different angles, many other shapes can be made this way. For example, you can begin to build a star by typing

```
FORWARD 50
RIGHT 150
FORWARD 50
RIGHT 150
```

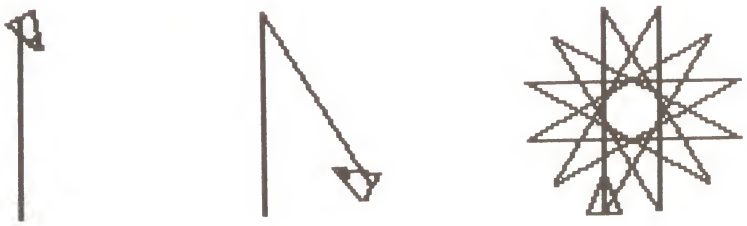


Figure 5.8: Steps in building a star.

Keep repeating FORWARD 50 RIGHT 150 until the shape is complete. Then give it a name and teach it to the computer.

Make a mirror image of the star in Figure 5.8, turning the turtle LEFT 150 every time. Make other shapes by starting with different angles.

You can use the same idea to make a regular triangle—one with three equal sides. The hardest part will be figuring out what angle to use. See if you can find the angle by experimenting. Use what you already know about angles and regular shapes. A turtle turn of 90 degrees would be too small to make a triangle.

```
FORWARD 50 RIGHT 90
FORWARD 50 RIGHT 90
FORWARD 50 RIGHT 90
```

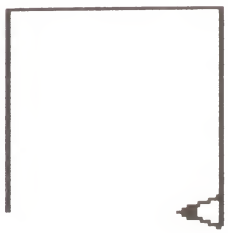


Figure 5.9: Three 90-degree turns do not make a triangle.

A turn of 150 degrees would be too big.

```
FORWARD 50 RIGHT 150
FORWARD 50 RIGHT 150
FORWARD 50 RIGHT 150
```

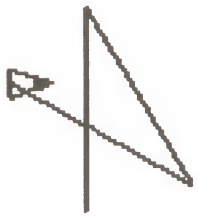


Figure 5.10: Three 150-degree turns don't make a triangle either.

The exact answer will be somewhere between the 90 and 150 degrees that you've already tried. Repeat this process with different angles until you get the turtle back to its starting place in exactly three steps.

When you get very close to the correct angle, it might be very hard to see whether the two ends meet exactly. You will be able to see the ends of the lines more clearly if you hide the turtle.

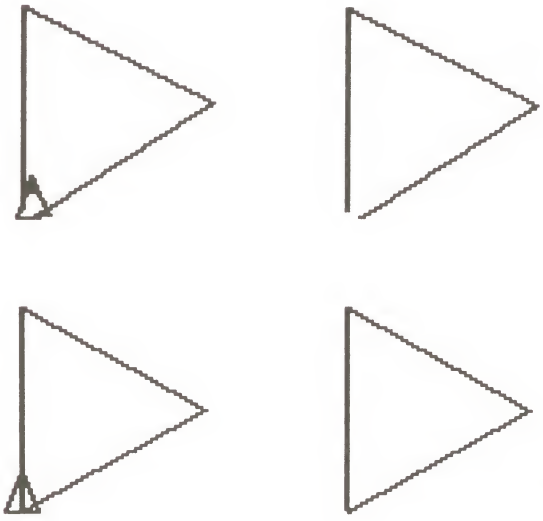


Figure 5.11: Hiding the turtle will make it easier to see if your triangle is complete.

Section 5.3. Using the REPEAT Command

The Logo command REPEAT makes the computer repeat a *list of commands* as many times as you want. REPEAT is very useful when you already know how many times to repeat something:

```
REPEAT 4 [FORWARD 20 LEFT 90]
REPEAT 12 [SQUARE RIGHT 30]
```

REPEAT needs *two* inputs. The first is the *number of repeats*. The second is a *list of commands* to repeat. In Logo, a list is always typed within *square brackets*, **[** and **]**.

[and **]** are not marked on the keyboard of the Apple II plus. To type them, use **SHIFT-N** and **SHIFT-M.** If you have an Apple IIe, be sure *not* to use **SHIFT** when you type **[** and **]**.

REPEAT can be used for procedures like SQUARE and BOXES from Section 5.1.

```
TO SQUARE 1
REPEAT 4 [FORWARD 50 RIGHT 90]
END

TO BOXES1
REPEAT 4 [SQUARE1 LEFT 90]
END

TO STAR1
REPEAT 2 [BOXES1 RIGHT 45]
END
```

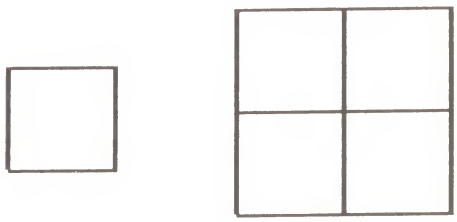


Figure 5.12: Shapes drawn using the REPEAT command.

REPEAT is also very useful for regular shapes like stars or triangles.

```
TO STAR 150
REPEAT 12 [FORWARD 100 RIGHT 150]
END

TO TRIANGLE
REPEAT 3 [FORWARD 80 RIGHT 120]
END
```

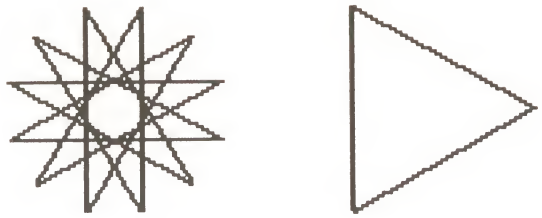


Figure 5.13: A star and triangle drawn using the REPEAT command.

REPEAT can also be used when you want to *find* the angle needed for a particular shape. For example, what angle would you use to make the turtle draw a five-pointed star like the one in Figure 5.14?

Figure 5.14: A five-pointed star.

You can use REPEAT to make the question easier to answer. Keep using REPEAT with the same forward step and different angles.

REPEAT 5 [FORWARD 50 RIGHT *something*]

First make an *estimate* of what the angle might be. Look at the first turn. It is bigger than 90 degrees, which would make a square, but less than 180 degrees, which would turn the turtle all the way around.

Figure 5.15: The correct angle is between 90 and 180 degrees.

Pick one angle to start with, say 120 degrees, and then keep increasing the angle until your shape closes.

REPEAT 5 [FORWARD 80 LEFT 120]

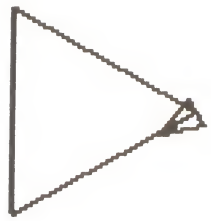


Figure 5.16: A turn of 120 degrees is too small.

```
REPEAT 5 [FORWARD 80 LEFT 130]
```

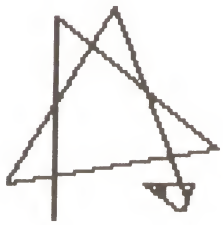


Figure 5.17: A turn of 130 degrees is still too small.

```
REPEAT 5 [FORWARD 80 LEFT 140]
```

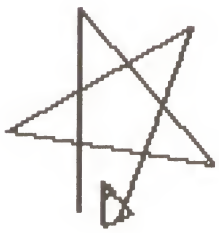


Figure 5.18: A 140-degree turn is getting close.

This looks pretty close. If you make the turn a little larger, the shape might close.

```
REPEAT 5 [FORWARD 80 LEFT 150]
```

Figure 5.19: A 150-degree turn is too much.

This time the last side *crossed* the first one, so a turn of 150 degrees is too big. Now see if you can finish the problem. The angle needed for a five-pointed star is between 140 and 150 degrees. Try to find it yourself.

When your angle gets very close, it might be hard to see whether the two ends meet exactly. Hide the turtle so you will be able to see the ends of the lines more clearly.

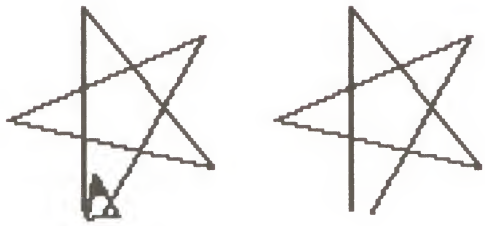


Figure 5.20: Hide the turtle so you can see more clearly whether the ends meet.

Here are some activities that you can try with REPEAT.

- Use REPEAT to make designs that use squares, stars and triangles as subprocedures.

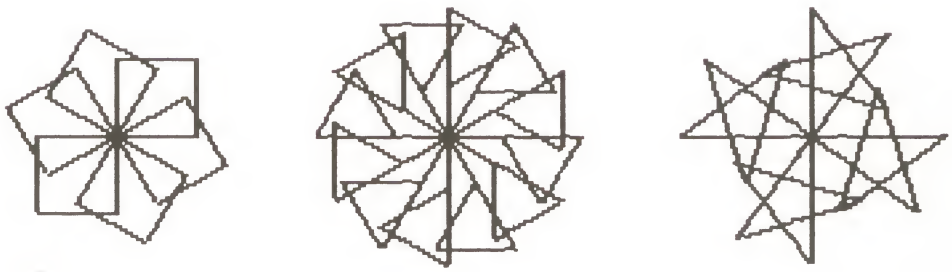


Figure 5.21: Designs made by using REPEAT.

- Make *mirror image* shapes using REPEAT. Make a square with the turtle turning to the right, then another square with the turtle turning to the left. Do the same thing with triangles and stars for which the turtle turns both left and right.

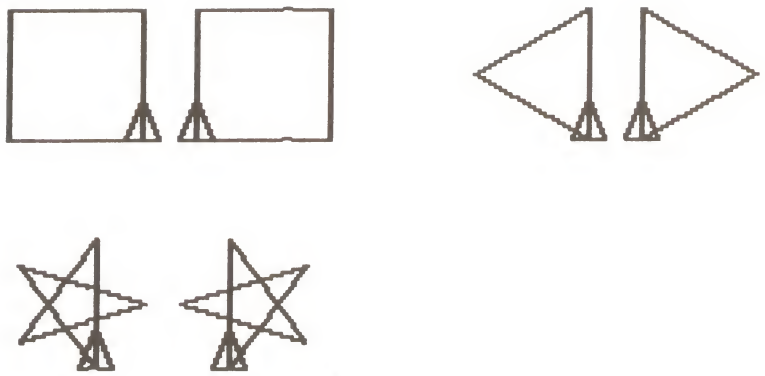


Figure 5.22: Shapes with their mirror images.

- Make other regular shapes, like a six-sided polygon or an eight-pointed star.

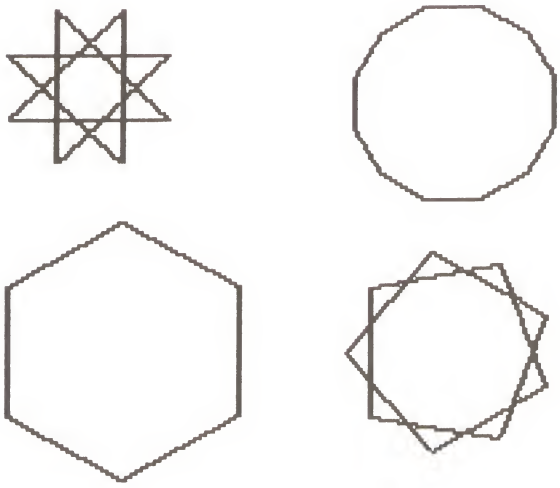


Figure 5.23: Polygons and stars made with REPEAT.

The five-pointed star example given above illustrates one systematic way to solve a type of problem common in turtle geometry. At least three powerful ideas for problem solving are demonstrated there.

1. Simplify the problem. There are so many *possible* five-pointed stars that it could take forever to draw one. By choosing a *regular* shape—one with all sides and angles the same—we make the problem manageable. Simplifying the problem in this way also makes it very much like some other problems we have *already* solved, such as drawing a square or a triangle.
2. Limit the exploration to *one* element at a time, in this case, the *angle* needed for a five-pointed star. Failure to limit the exploration is a very common problem solving bug for adults as well as children. Watch people solve this type of problem sometime. You'll notice that many people include several elements (for example, the size or orientation of the star) in addition to the angle in their explorations. Such an approach makes the problem a much more difficult and frustrating one.
3. Make an initial estimate and then systematically narrow the limits. When you know that the solution is between 140 and 150, you're much closer than when you knew it was

between 90 and 180. If you vary the angle randomly, it could take much longer to approach a solution.

Some people are much better visual estimators than others. Unless they are *also* systematic problem solvers, good estimators sometimes take longer to solve a problem than people with poor estimating ability but good problem solving techniques. I chose 120 degrees as my first estimate to show how to systematically and surely improve even a relatively poor first choice.

There is an entirely different way to solve the problem of drawing a five-pointed star—a mathematical or *analytical approach*. This approach tells us that *in order to draw a closed shape, the turtle must turn through a total angle which is an exact multiple of 360 degrees*. That is, the turtle has to make a total turn of *exactly* 1 × 360 (360 degrees) or 2 × 360 (720 degrees) or 3 × 360 (1080 degrees), etc., before it can get back to its starting point. This fact is sometimes called the "Total Turtle Trip Theorem." Since the turtle has to turn *five* times in drawing a five-pointed star, try 360 / 5 as the angle. If that doesn't work, try (2 × 360) / 5, or (3 × 360) / 5, etc. You can use Logo arithmetic commands to make the computer do the calculations.

```
REPEAT 5 [FORWARD 80 LEFT 360 / 5]
REPEAT 5 [FORWARD 80 LEFT (2 * 360) / 5]
```

and so on.

Section 5.4. Using Recursion

There is another way to make the computer repeat something over and over—make a procedure that calls *a copy of itself* as a subprocedure. This way of repeating something is called *recursion*. It is useful when you want to experiment with shapes but don't know how many times to repeat something. It's a good way to draw something when you know the angle, but not the number of repeats.

```
TO STAR160
FORWARD 100
RIGHT 160
STAR160
END
TO FLOWER
TRIANGLE
RIGHT 60
FLOWER
END
```

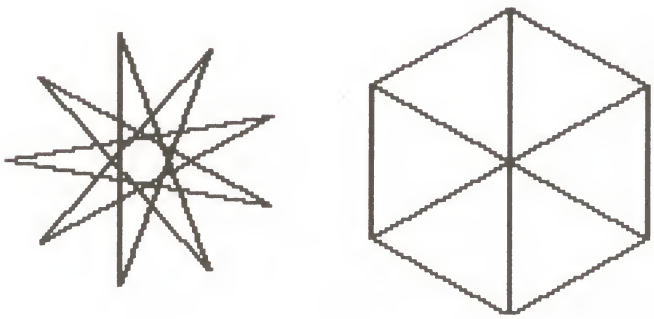


Figure 5.24: A star and a flower drawn by using recursion.

What's happening here? Why do these procedures work? Look at Figure 5.25. When you type the command STAR160, Logo calls a procedure with that name. Think of the procedure as one of Logo's mechanical assistants.

Figure 5.25a

Now STAR160 goes to work. It calls the FORWARD command and tells it to move the turtle 100 steps.

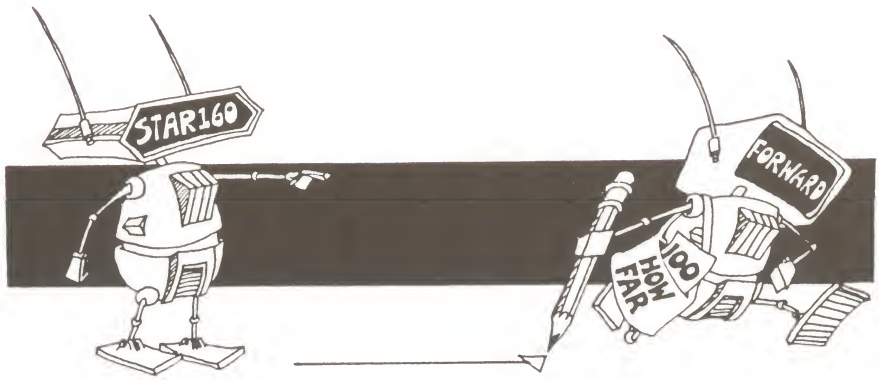


Figure 5.25b

Next, STAR160 calls the RIGHT command and tells it to turn the turtle 160 degrees.

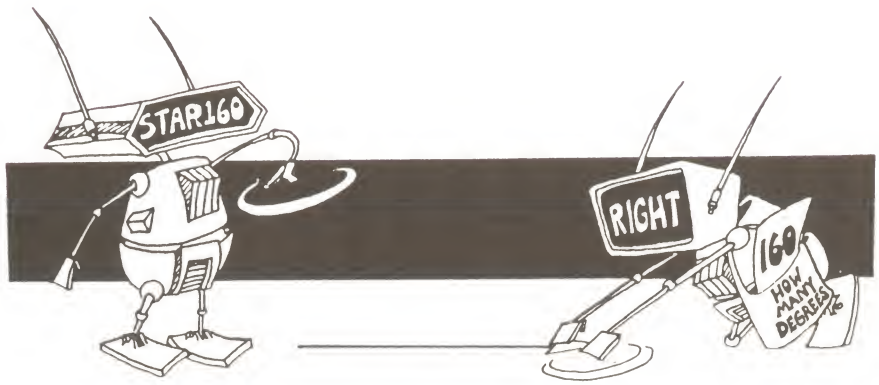


Figure 5.25c

Finally, STAR160 calls *another* assistant, the subprocedure STAR160.

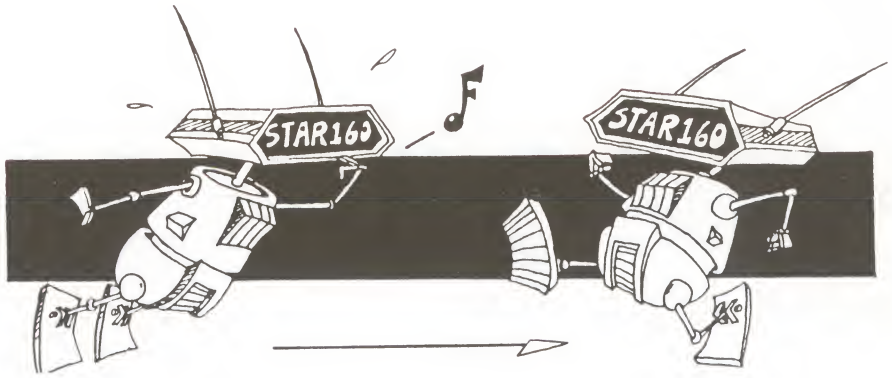


Figure 5.25d

This new assistant, the second STAR160, will now start doing its job. It will call FORWARD, RIGHT, and another new assistant, STAR160. And so on.

You might ask, "How does it ever stop?" Good question! The answer is, "It doesn't!" You have to stop it by typing **CTRL-G**. Try a procedure like one of the examples and see. In Chapter 7 you'll learn how to make a procedure that can stop when it is finished. For now, the only way to stop a recursive procedure is by typing **CTRL-G**.

Here's something else that's fun to try with recursion. Make a silly shape, any old thing, with the turtle. Give it a name and then make a recursive procedure with it.

```
TO SILLY
FORWARD 50
LEFT 90
FORWARD 20
LEFT 120
FORWARD 30
END
```

Figure 5.26: The shape drawn by SILLY.

By itself this isn't much of anything, but put it into a recursive procedure and see what happens.

```
TO SILLYONE
SILLY
SILLYONE
END
```

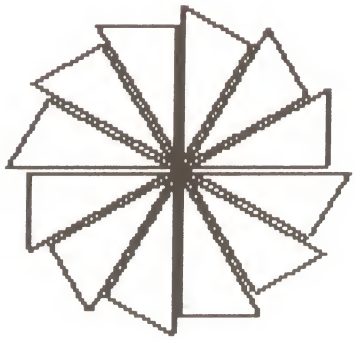


Figure 5.27: The shape drawn by SILLYONE.

Or, add a few steps to SILLY and then add a recursion line.

```
TO SILLYTWO
SILLY
RIGHT 60
BACK 50
SILLYTWO
END
```

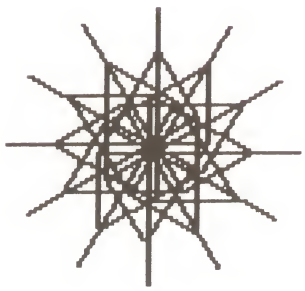


Figure 5.28: The shape drawn by SILLYTWO.

Make your own variations of this idea. You can start with just about *anything* and make an interesting design.

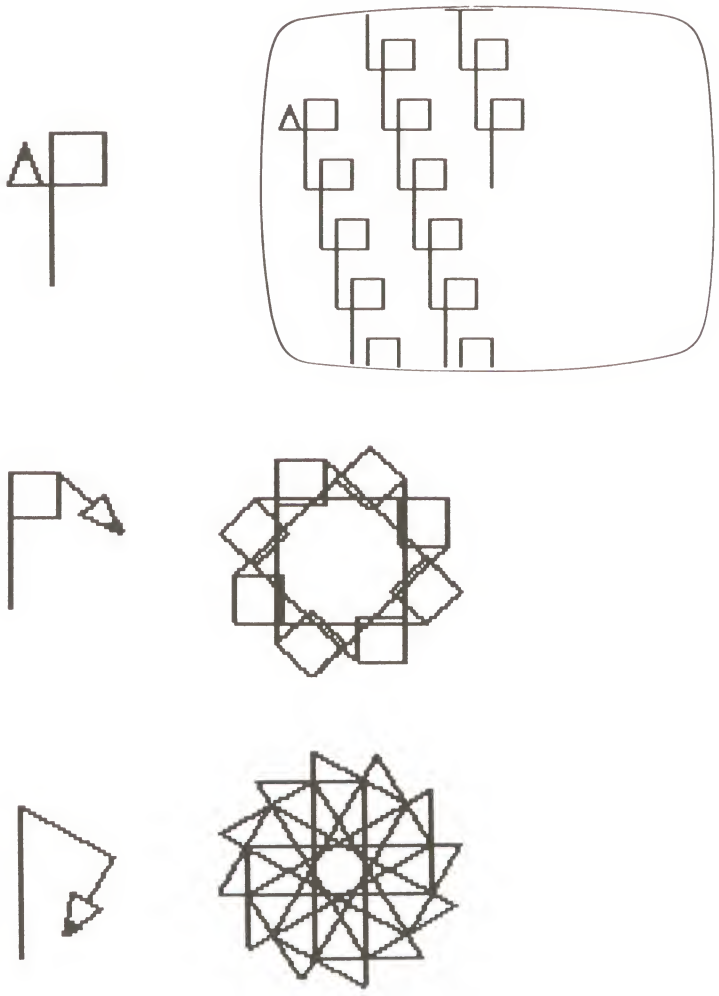


Figure 5.29: Examples of starting shapes and their recursive designs.

Recursion is good for making stars and polygons. Regular shapes with lines that cross each other, like STAR135, are usually called "stars." Regular shapes with lines that don't cross, like STAR45, are usually called "polygons."

```
TO STAR135
FORWARD 60
LEFT 135
STAR135
END

TO STAR45
FORWARD 30
RIGHT 45
STAR45
END
```

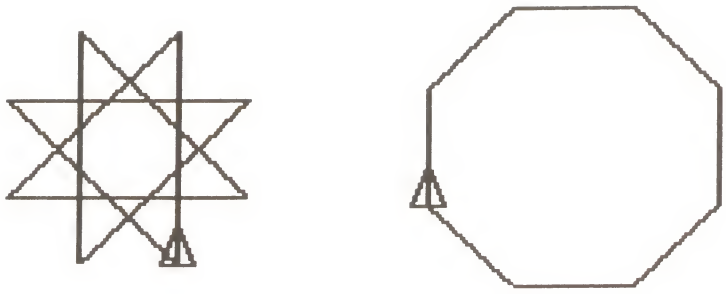


Figure 5.30: Shapes drawn by STAR135 and STAR45.

Can you see how to use recursion to make a circle? Hint: pretend *you're* the turtle. Walk in a circle using combinations of steps and turns, but don't move and turn at the same time—the turtle can't do that even though you can. Look at the illustration showing the turtle walking in a circle.

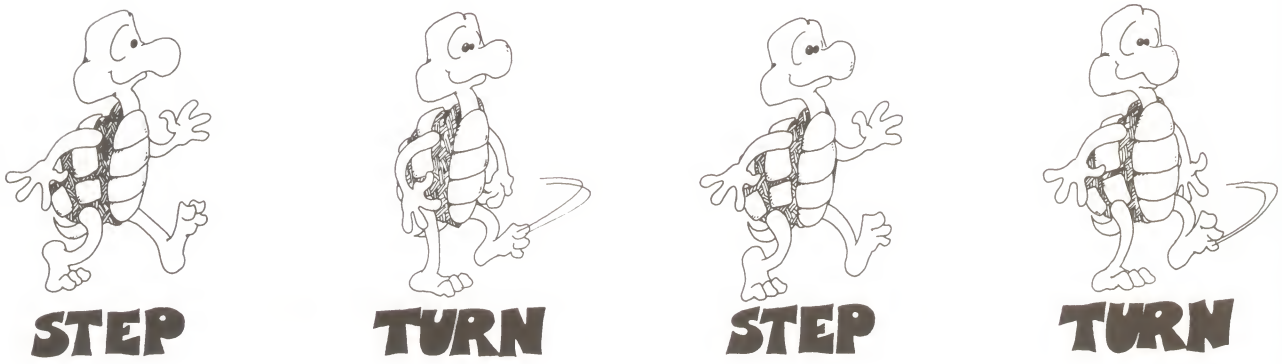


Figure 5.31: Playing turtle to figure out how to draw a circle.

Now can you see how to write a circle procedure? Here's part of a "circle" drawn by the turtle. See if you can make the turtle draw a full "circle" using recursion.

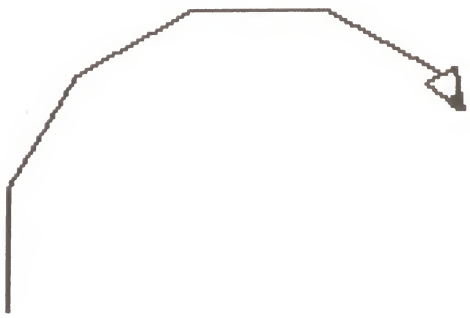


Figure 5.32: A partial circle drawn with 20-degree turns.

Recursion is another of Logo's most powerful ideas—a procedure which calls itself! This turns out to be useful for making some otherwise complicated things very easy. (The most complicated are beyond the scope of this book, but you'll want to learn about them to do "serious" Logo programming. There are some examples in Chapter 14, "How the Special Tool Procedures Work.")

If you're familiar with BASIC, you might mistakenly think that recursion is just another way to do what BASIC does using GOTO or FOR-NEXT loops. The *process* of recursion should not be confused with the *process* of looping. Recursion really means calling a *new* subprocedure with the same name. It's very different from going back to an earlier instruction and continuing on.

There's a lot more about recursion in Chapter 7. That's where variables and conditional stop rules are first used. It is introduced in this chapter as a good way of making designs, which is, in turn, a good way of introducing recursion.

Section 5.5. Designs with Circles and Arcs

In Chapter 2, you started using circle and arc procedures. Now that you know something about teaching the computer new commands, you can use circles and arcs to make more interesting shapes. First you have to LOAD "CIRCLES from the LWAL Procedures Disk. Circle and arc procedures RCIRCLE, LCIRCLE, RARC, and LARC each need one input number, the *radius* of the circle or arc you want to draw.[1]

Here are some designs you can make with circles. If you hide the turtle while drawing circles and arcs, it will draw them a lot faster.

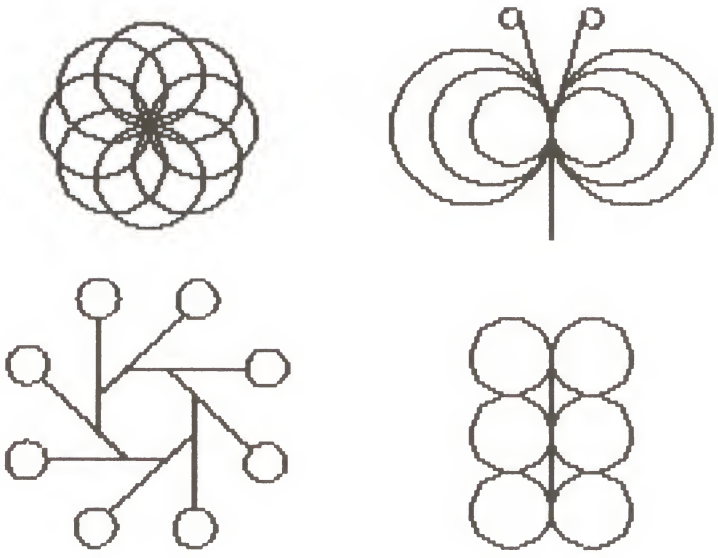


Figure 5.33: Some designs made with circles.

[1] If you prefer, you can use the procedures CIRCLER, CIRCLEL, ARCR, and ARCL, which are part of the startup file on the Apple Logo Language Disk. ARCR and ARCL use two inputs. The *first* is the radius of the arc. Use 90 for the *second* input, to draw the designs shown here. Section II.7 of Appendix II explains how to add my circle-and-arc procedures to the Apple Logo startup file.

Can you see how to make this one?

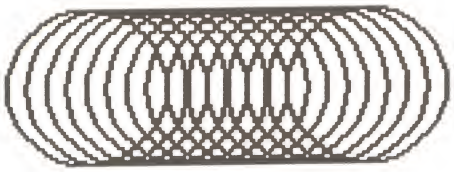


Figure 5.34: A "slinky" design.

You just draw a circle, move a turtle a little (with its pen up), and do that all again. If you add a rotation before each move, you'll get something like the design in Figure 5.35.

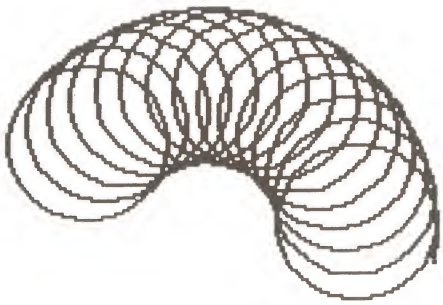


Figure 5.35: A "curved slinky" design.

Figure 5.36 shows a harder design.

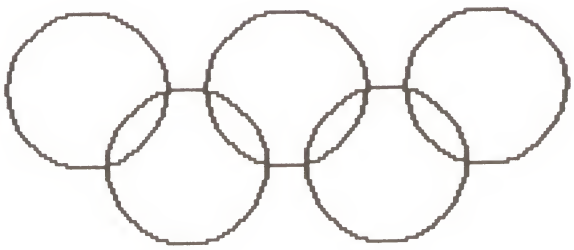


Figure 5.36: Five interlocking circles, the symbol of the Olympic Games.

Hint: Make two separate rows of circles, then figure out where the second row should start so that the design will come out the way you want it.

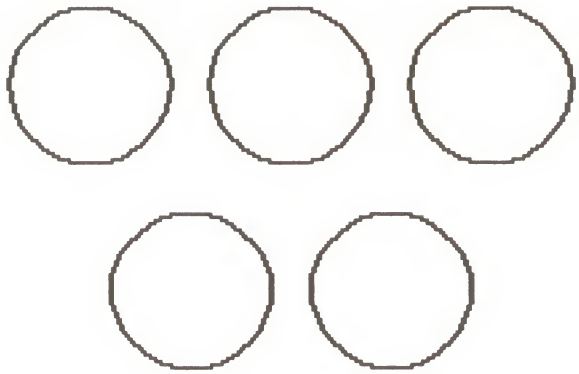


Figure 5.37: Two rows of circles.

Here is a very hard problem: Can you make the turtle draw a circle starting and ending at its *center*? This is very useful for many designs. Figure 5.38 shows some of the steps. See if you can make them into a procedure.

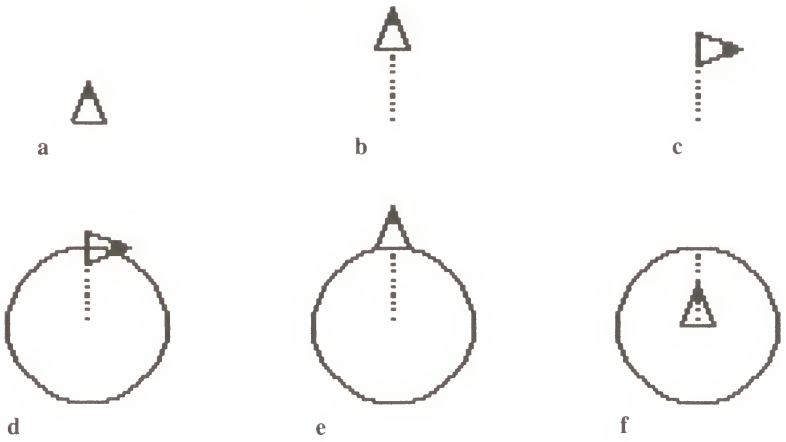


Figure 5.38: Some of the steps for making a centered circle.

Put this circle and some lines together to draw a sort of sun, as shown in Figure 5.39.

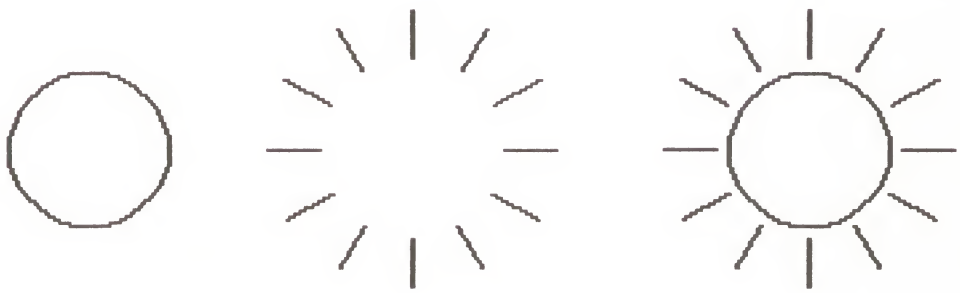


Figure 5.39: Combine a centered circle with centered lines to create a sun.

RARC and LARC draw *quarter circles*. You can make interesting designs by putting them together. Figure 5.40 shows a "snake" made by using right and left arcs one after another.

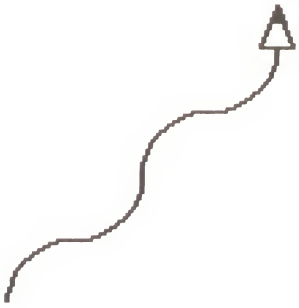


Figure 5.40: A snake made by using arc procedures.

This can be made into a different kind of sun. Draw a snake, turn the turtle almost all the way around, and repeat the whole process.

```
TO SUN
SNAKE
RIGHT 160
SUN
END
```

Figure 5.41: Making a snake into a sun.

You'll get very different designs by changing the angle.

You can also put two arcs together to make a petal, as shown in Figure 5.42.

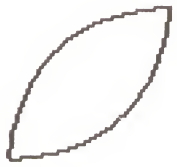


Figure 5.42: A petal made from two arcs.

Can you see how to make the petal? First, pick a size for your arc.

```
RARC 30
```

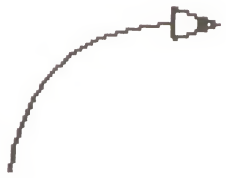


Figure 5.43: An arc drawn by RARC 30.

If you just add another arc, you get a *semicircle*.

```
RARC 30
RARC 30
```

Figure 5.44: Two arcs make a semicircle.

To make the petal, you have to turn the turtle before drawing the second arc. How much should you turn the turtle? Believe it or not, a petal is very much like a square. This is because the turtle turns through a square corner, exactly 90 degrees, while drawing RARC.

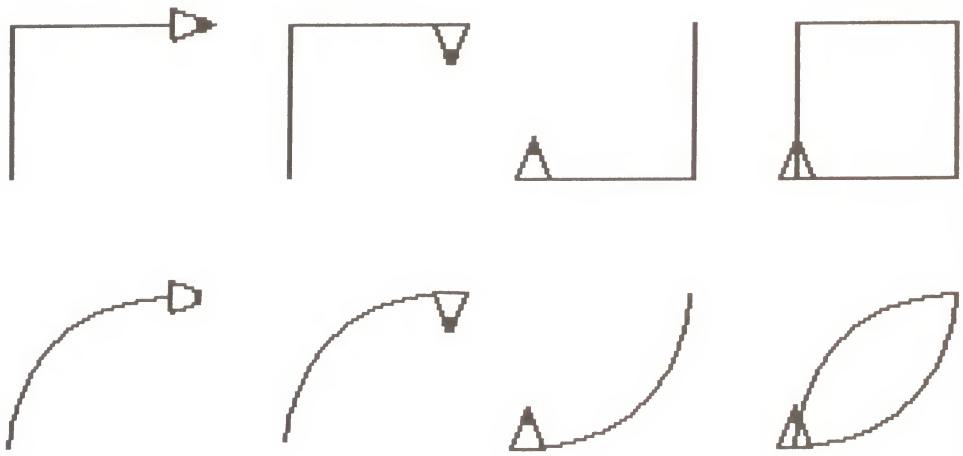


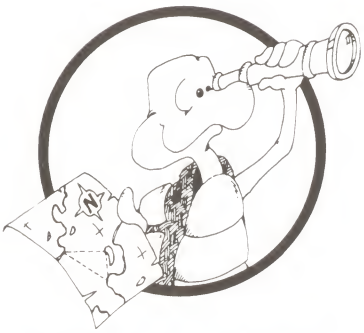
Figure 5.45: Making the petal is a lot like making a square.

To complete the PETAL, you need a 90-degree turn at each end.

```
TO PETAL
RARC 30
RIGHT 90
RARC 30
RIGHT 90
END
```

Like SQUARE, PETAL leaves the turtle back where it started.

Put some petals together to make different kinds of flowers.



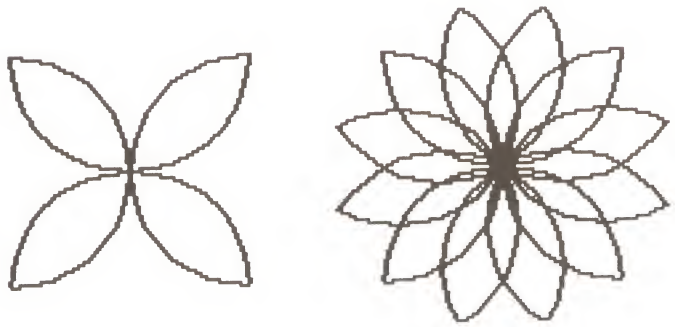


Figure 5.46: Two different kinds of flowers.

Draw a spiral by making arcs get larger and larger. (In Chapter 7 you'll learn how to do this more easily using *variables*.)

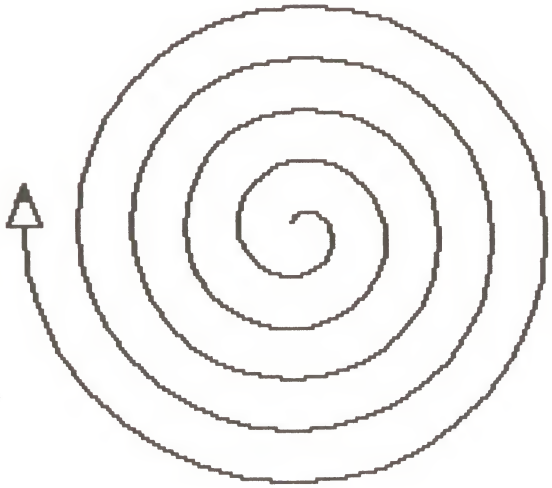


Figure 5.47: A growing spiral.

Here's another neat trick you can do with arcs. Make the turtle retract its steps to get back where it started. Start with a snake, for example. Turn the turtle right or left 180 degrees, you can retrace the turtle's steps using the *same* procedure.

```
TO SNAKE
RARC 30
LARC 30
RARC 30
LARC 30
END

TO RAY
SNAKE
RIGHT 180
SNAKE
RIGHT 180
END
```

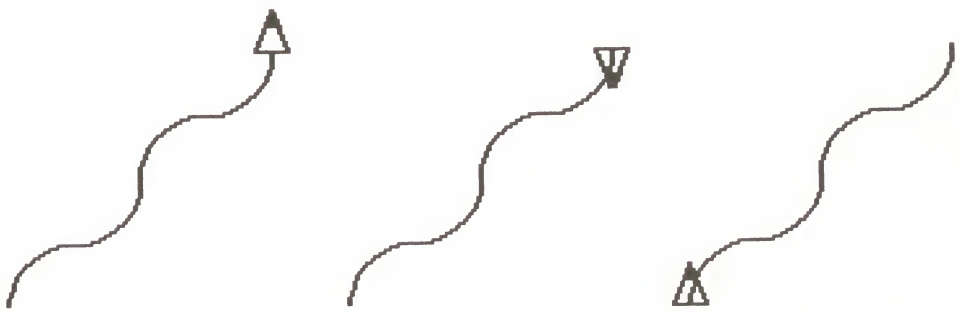


Figure 5.48: Retracing the turtle's steps to draw a ray.

Combine RAY with a rotation to make another kind of star.

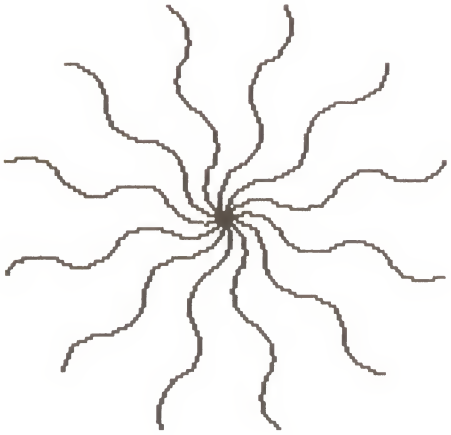


Figure 5.49: A star made from rays.

Make a *mirror image* snake by reversing the order of right and left arcs. Then you can put SNAKE and MIRRORSNAKE together to make this kind of design:

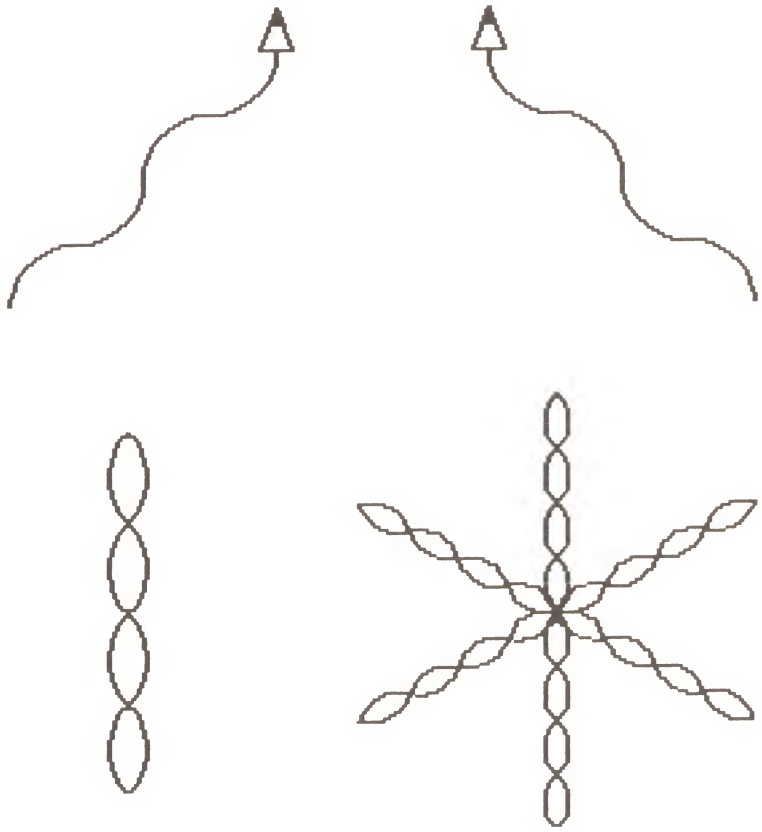


Figure 5.50: Designs using SNAKE and MIRRORSNAKE.

CHAPTER 6

New commands used: none
*LWAL Procedures Disk files used: "*CIRCLES
New tool procedures used: none.

6 Turtle Projects 2: Drawings

You can use the turtle to make all kinds of drawings: people, animals, cars, planes, trucks, or even your initials, for example. In this chapter you'll see see some examples of how this is done, so that you can make the turtle draw whatever you want.

Drawing with the turtle can be very different than other kinds of drawing. This is because the turtle is so dumb! You have to tell it every single step and turn to make. Of course, once it *knows* how to draw something, it can easily redraw the same picture over and over again.

Don't be afraid to try a project that looks difficult at first. Any turtle drawing, no matter how complicated, can be built up from small, simple pieces. If you work slowly and think carefully about what you are doing, you should be able to complete any project in this chapter.

Here are a few tips that can make complicated turtle drawing projects a lot easier:

1. Draw your idea first in your journal or on a piece of paper.
2. Divide the drawing into parts. You can do a very complicated drawing one part at a time.
3. Give each part a name. You'll use these names for procedures.
4. Draw a picture showing how the parts could fit together, and number them in the order you want to put them together.
5. Simplify the project. If some of the parts look too hard to draw, leave them out or substitute something that's easier to draw.
6. Now start writing procedures. Some people like to do all the procedures in order so that they can see the project fit together as they go along. Others like to make all the pieces first, starting with the easiest, and put them together later. Either way is fine, depending on how you like to do things and the particular project you're working on.

HELPER'S HINT

People have very different approaches to developing projects. The approach I am suggesting here is sometimes called *top-down* programming because it starts from the "top," the general ideas, and gradually works "down" to the details. Not everyone is comfortable with this abstract an approach to planning a project. Some learners accept and understand project design principles like these very readily. Others develop their own approaches to project design over a long period of trial and error. These ideas are meant to be *suggestions*, available for anyone who wants to use them. They shouldn't be forced on anyone who isn't ready for them or who prefers to do things differently.

In fact, in teaching Logo to many beginners I have found that it is best for them to find their own ways to do things. For a helper, the most important thing is to have patience, to understand how each particular person thinks about what he or she is doing, and to offer help within that understanding. It can be as counter-productive to force everyone into the same mode of learning as it would be to make everyone do the same project. In time, students come to "own the process" as well as the product. This ownership cannot be forced. The trick of teaching is to know when to make a suggestion and when to leave someone alone. Perhaps even more subtle is to learn to make a suggestion and then be comfortable with a learner who rejects it.

Not everyone can hold two images of the same project in his or her head at once, that is, see the project both as a totality and as the sum of its parts. Some people prefer a more intuitive approach to using subprocedures and simply start out drawing the entire design. When it is clear that part of the design is finished, or when a good stopping point is reached, give that part a name and teach it to the computer. In this way, the idea that a large project can be built up from subparts can be introduced gradually, without imposing it in advance. Later (sometimes much later) someone may begin to see the advantage of planning the parts ahead of time.

Section 6.1. Drawing a Truck

Suppose you want to make the turtle draw a truck like the one drawn by hand in Figure 6.1.

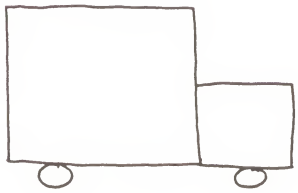


Figure 6.1: Hand-drawn design for a truck.

I would design this truck by dividing it into three parts named BIGBOX, SMALLBOX, and WHEELS.

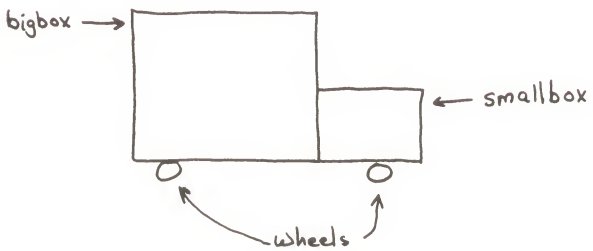


Figure 6.2: Dividing the truck into small parts.

These parts can be put together in any order. First, let's make each part separately. Start each part with the screen cleared and the turtle at home.

```
TO BIGBOX
REPEAT 4 [FORWARD 60 RIGHT 90]
END

TO SMALLBOX
REPEAT 4 [FORWARD 30 RIGHT 90]
END

TO WHEELS
RIGHT 90
RCIRCLE 5
FORWARD 90
RCIRCLE 5
BACK 90
LEFT 90
END
```

Figure 6.3 shows what each of the three parts of the truck look like.

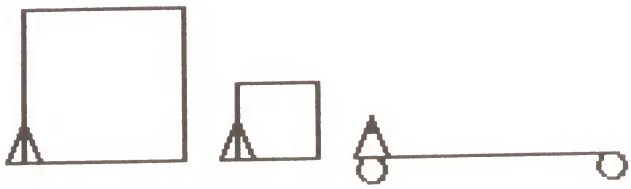
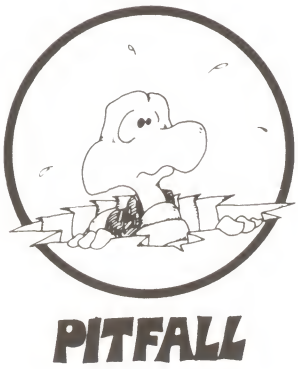


Figure 6.3: The three parts of the truck.

To draw the WHEELS you'll need to LOAD "CIRCLES from the LWAL Procedures Disk. If you don't have a complete LWAL Procedures Disk yet, you can copy the circle procedures from Appendix I.



Be sure to put your own Logo work disk back in the disk drive after loading "CIRCLES.

Now, put them all together.

```
TO TRUCK
BIGBOX
SMALLBOX
WHEELS
END
```

Type the command, TRUCK.

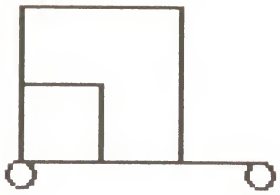


Figure 6.4: The first try for a truck doesn't quite work.

PITFALL

WHOOPS! This isn't exactly what we planned. We've got a bug! Don't be discouraged. This happens to everybody who works with computers.

The bug is easy to fix. We made each part, but we forgot to put them where they belong. We need two more procedures, MOVEOVER and MOVEBACK. One moves the computer over from the BIGBOX to the SMALLBOX. The other moves it back. There are lots of ways to do this. Try to figure out how to do it yourself before looking ahead.

POWERFUL IDEA

Everyone who works with computers meets bugs from time to time. They can't be avoided. Sometimes the best thing to do when something like this happens is to laugh at how dumb the computer is.

Figure 6.5 shows what I'd like the MOVEOVER and MOVEBACK procedures to do.

Figure 6.5: What MOVEOVER and MOVEBACK do.

```
TO MOVEOVER
RIGHT 90
FORWARD 60
LEFT 90
END
```

```
TO MOVEBACK
LEFT 90
FORWARD 60
RIGHT 90
END
```

Now edit TRUCK and add these two new procedures.

```
TO TRUCK
BIGBOX
MOVEOVER
SMALLBOX
MOVEBACK
WHEELS
```

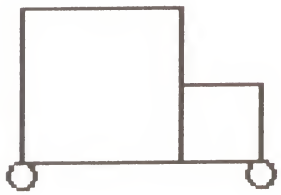


Figure 6.6: The truck looks better on the second try!

At last! A finished truck! Or is it? If I were being fussy I might say, "That's not like the picture I drew. The wheels aren't in the right places yet."

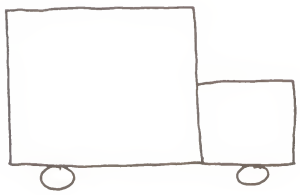


Figure 6.7: Compare the truck with our original design.

Now it's up to you. Lots of people are satisfied with something that's pretty much like what they started out to do. You might even like it *better* this way! On the other hand, you could change the WHEELS procedure so that it looks more like the original plan.

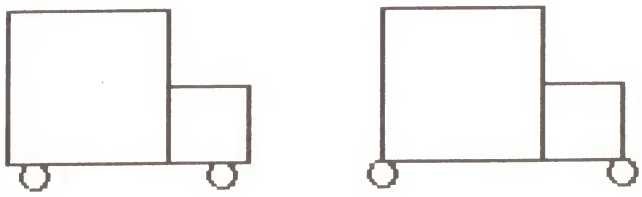
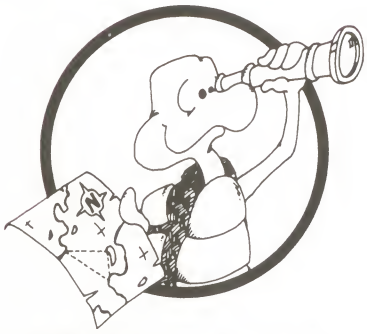


Figure 6.8: A truck with changed wheels.

PITFALL

Don't forget to save your procedures now by typing SAVE "TRUCKS or whatever name you choose.



EXPLORATION

Now that you've built the truck, you might want to add some features or make some different kinds of trucks. Add a cab and bumpers, change the wheels, or change it into a van or a pickup. Figure 6.9 shows a fleet of Logo trucks.

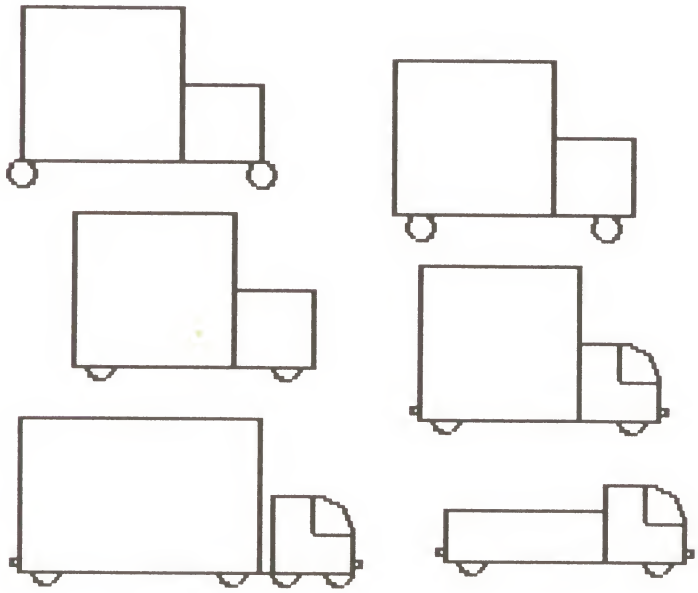


Figure 6.9: A fleet of Logo trucks.

HELPER'S HINT

Bugs were deliberately introduced into this example to illustrate two important ideas. First, bugs do appear all the time. Often they can be quite amusing. Second, some bugs, although they lead to an unintended result, can be quite acceptable and maybe even improve a design or transform it into something else. Exact completion of the original plan is neither necessary nor desirable.

The first bug is an example of a *turtle state bug*. It comes from not having accounted for the *state* of the turtle—its exact position and heading—before and after drawing each part of the figure. That is, we failed to realize that the turtle had to move over after drawing BIGBOX before it could draw SMALLBOX. Such bugs are very common in all kinds of turtle projects.

The second bug is also a turtle state bug, but it's a little different. It was due to our first making a simpler version of the wheels and then deciding whether to elaborate it to make it more like the original plan. It's a good strategy to make the first attempts as simple as possible, get the whole thing together, and then elaborate. I often encourage people to simplify a design before programming it, with the understanding that it can be elaborated or extended later.

A plan is not meant to be set in stone. The best problem-solving processes incorporate flexibility and allow for reevaluation at any point. If something turns out to be harder than you thought, simplify it. You can always come back to it later. Or, if you get a good idea for how something could be more interesting, don't hesitate to revise your plan as you go along.

Section 6.2. Drawing a Person

Here's another easy project—drawing a stick figure of a person. This person has arms, legs, a body, and a head.

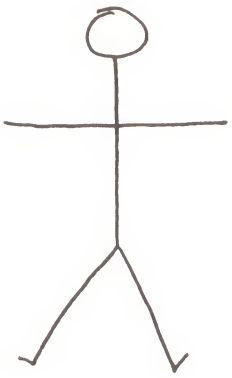


Figure 6.10: Hand-drawn design for a stick-figure person.

This time, let's teach the computer to PERSON.

```
TO PERSON
BODY
LEGS
ARMS
HEAD
HAT
END
```

PERSON is a *superprocedure;* that is, it's the one procedure that makes everything else happen. A *superprocedure* is like the *boss* of the project. You tell PERSON to go to work, and then PERSON calls on all its helpers to get the job done.

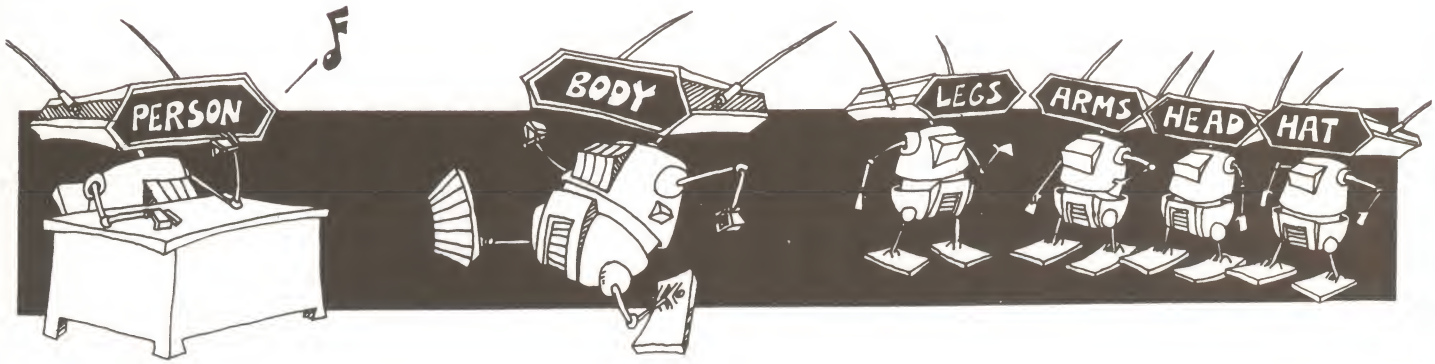


Figure 6.11: The superprocedure PERSON gives orders to its helpers.

If you type the command PERSON before teaching the computer how to BODY, ARMS, LEGS, etc., Logo will complain. Try it and see.

Now let's teach it to BODY. This is just a stick.

```
TO BODY
FORWARD 30
BACK 30
END
```

Figure 6.12: Result of the BODY procedure.

Look at the legs for a minute. The left leg and right leg are *symmetrical*. That is, they are exact mirror images of each other. We'll use two subprocedures, LEFTLEG and RIGHTLEG.

```
TO LEGS
LEFTLEG
RIGHTLEG
END
```

The LEFTLEG procedure makes the turtle start and end at the same place so that RIGHTLEG can be easily added later.

```
TO LEFTLEG
RIGHT 30
BACK 30
LEFT 90
FORWARD 5
BACK 5
RIGHT 90
FORWARD 30
LEFT 30
END
```

Figure 6.13: Result of the LEFTLEG procedure.

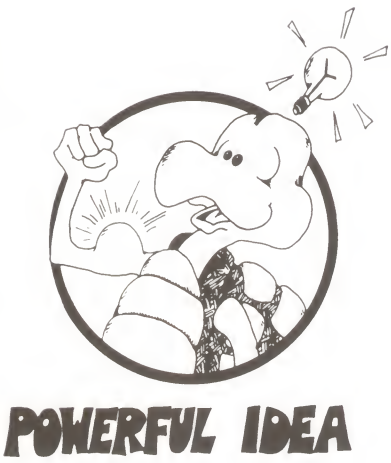
The last four steps *reverse* the first four in order to get the turtle back where it started.

RIGHTLEG uses the same FORWARD and BACK steps as LEFTLEG, but *reverses all the angles*.

```
TO RIGHTLEG
LEFT 30
BACK 30
RIGHT 90
FORWARD 5
BACK 5
LEFT 90
FORWARD 30
RIGHT 30
END
```

Figure 6.14 shows what we get when we type the command LEGS.

Figure 6.14: Result of the LEGS procedure.



When something is *symmetrical*, the right side is the mirror image of the left or the top is the mirror image of the bottom. You can use symmetry to save yourself a lot of work in many projects. With *right/left symmetry*, if you know how to draw one side, you can draw the other by reversing angles.

To use this idea, you also have to use another powerful idea—making the procedure start and end at the same place.

Now, type the command PERSON. The computer should draw part of the figure and print a message telling you that it hasn't been taught how to ARMS yet.

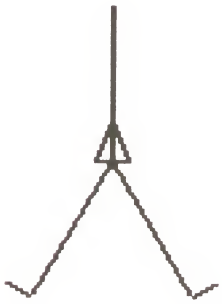


Figure 6.15: Putting LEGS and BODY together.

Now let's teach it to ARMS.

```
TO ARMS
FORWARD 20
RIGHT 90
FORWARD 20
BACK 40
FORWARD 20
LEFT 90
BACK 20
END
```

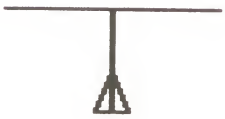


Figure 6.16: Result of the ARMS procedure.

Now if you type PERSON, the computer should draw the design in Figure 6.17.

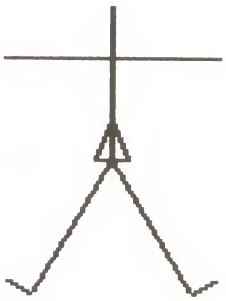


Figure 6.17: PERSON still needs a head.

The head should look something like a balloon.

Figure 6.18: This balloon should make a good head.

See if you can figure out the HEAD procedure by yourself. You'll have to load the "CIRCLES file from your LWAL Procedures Disk or copy the circle procedures from Appendix I. Remember to have it start and end at the same position so that the other parts of the person will connect with it.

The last subprocedure is HAT.

```
TO HAT
PENUP
FORWARD 40
PENDOWN
DRAWHAT
PENUP
BACK 40
END
```

HAT uses PENUP so that the turtle doesn't draw a line across the person's face. HAT has a subprocedure of its own, DRAWHAT. DRAWHAT uses two RARC procedures, one after another, to make a semicircle.

```
TO DRAWHAT
LEFT 90
FORWARD 10
BACK 5
RIGHT 90
RARC 5
RARC 5
LEFT 90
FORWARD 5
BACK 10
LEFT 90
END
```

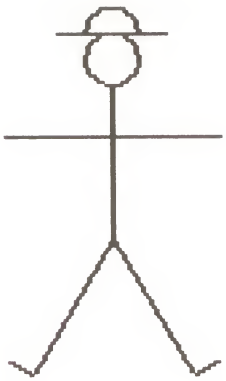


Figure 6.19: Our finished PERSON.

If you used a radius of more or less than 5 for your HEAD procedure, you'll have to adjust the first and last step of HAT so that it fits. Can you change HAT so that it fits on the head as shown in Figure 6.20?

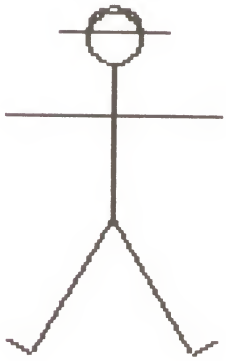


Figure 6.20: A PERSON with a HAT on the middle of its head.

Which way does it look better to you?

Figure 6.21 gives some variations you might want to try.

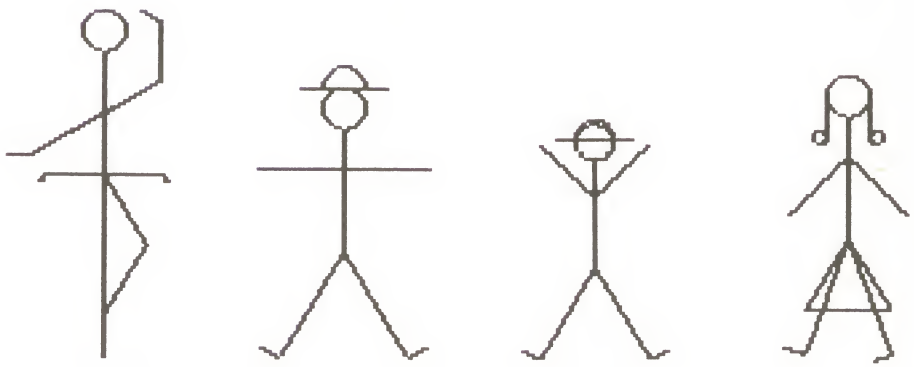


Figure 6.21: PERSON can be modified in a number of ways.

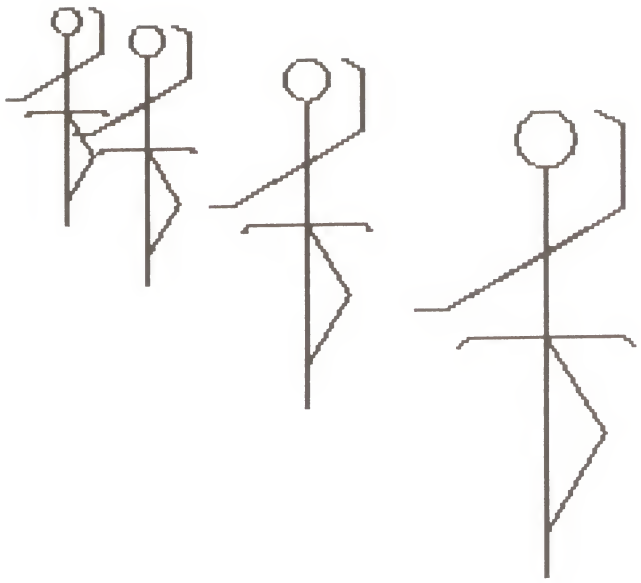


Figure 6.21 (continued).

There are several important ideas illustrated by this project.

1. *Symmetry* is useful in many contexts. Using symmetry to solve a problem is related to a more general idea: When dividing a problem into parts, look for ways of doing it so that the total problem is simplified.
2. A procedure in which the turtle starts and ends in the same *state* is said to be *state transparent*. At first, state-transparent shapes seem unnecessarily complex. They have certain advantages, however. For example, state-transparent subprocedures can be combined in any order. I could have written PERSON in the opposite order and it would still have worked. Another important advantage is that I can substitute any other state-transparent procedure for one of my subprocedures and everything else would still work. To draw the figure with V-shaped arms, I could just substitute a different ARMS procedure without changing anything else.
3. The use of a *superprocedure* as a *plan*: PERSON is both a *superprocedure*—that is, a procedure which makes the turtle draw the entire shape—and a *plan* for the whole project. By writing the superprocedure, I was also writing a clear plan for the whole project. Teaching the computer the superprocedure before defining any of the subprocedures allowed me to check my progress as I went along without having to type in a bunch of subprocedures. Like any plan, I might have had to change the superprocedure as I worked out the details of the project. In this case, though, I was able to complete the project without modifying my original plan.

Section 6.3. Drawing a Flower

There are lots of ways to draw a flower. Here's one that uses arcs to make the petals and leaves. The basic building block is called PETAL. It is made from two RARC or LARC procedures.

Figure 6.22: An individual petal and a finished flower.

Do you remember how to make the petal? We made one just like it in Section 5.5. Look back at that section if you need help. Of course you'll have to load the "CIRCLES file from your LWAL Procedures Disk or copy it from Appendix I. Then make several petals of different sizes so that you can make a lot of different flowers.

Use PETAL to make a lot of different kinds of blossoms. The size of each blossom depends on the size you choose for your petals.

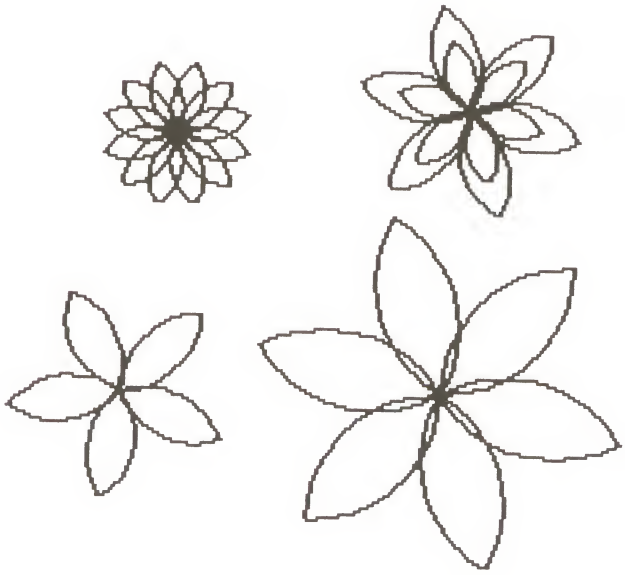


Figure 6.23: Different kinds of blossoms.

Make a flower by putting together a stem, a leaf or two, and a blossom.

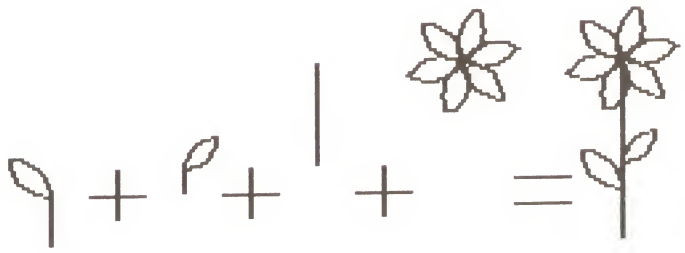


Figure 6.24: Assembling a flower.

Make yourself a whole garden. Use different pen colors if you have a color TV or monitor.

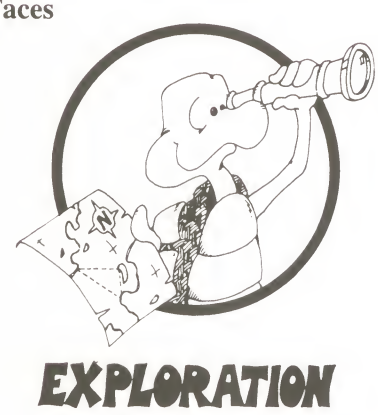
Figure 6.25: A garden of flowers.

In Chapter 7, you'll learn how to teach the computer procedures with *inputs* that make it easy to draw different sized blossoms, leaves, and flowers. You'll be able to make a really fancy garden when you know about inputs.

Section 6.4. More Ideas for Turtle Drawing Projects

The project ideas in this section come from students who have used Logo. They are given as a collection of pictures, sometimes with a hint or two. These ideas are included here to serve as starting points for your own projects. You may never want to do any one of these particular projects, but they will give you ideas of what other people have enjoyed doing with Logo.

Faces



Faces tend to be *symmetrical*, so you can use symmetry when you draw them. Other faces are funnier if they are *not* symmetrical, or if they include parts that are not. Study each design to see what its parts are and how they were put together.

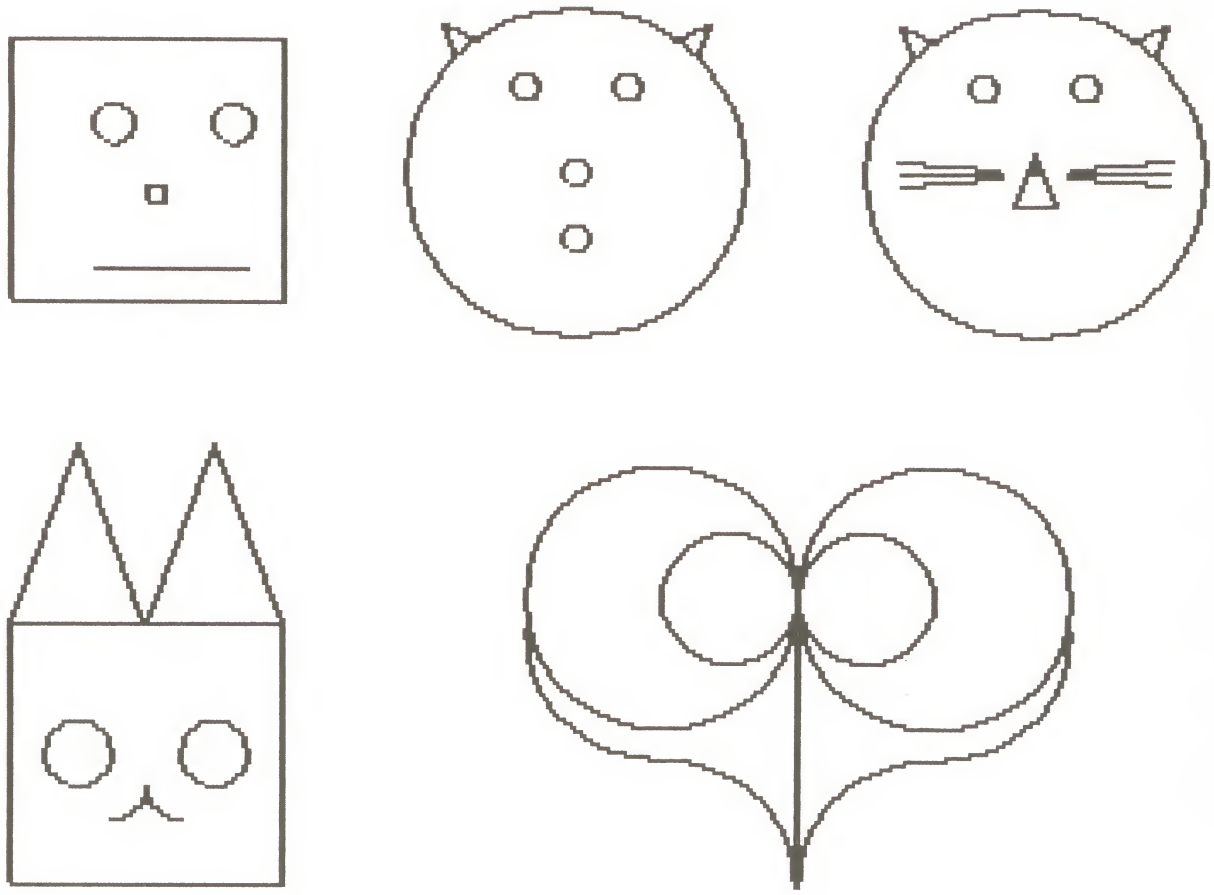


Figure 6.26: A variety of faces.

The head in Figure 6.27 was designed and built by an 11-year-old boy named Donald. He started by teaching the computer a superprocedure called HEAD. Then he carefully made each of the subprocedures in order. Each one took a lot of planning and testing to make it work. Some of the subprocedures have subprocedures of their own.

```
TO HEAD
BOX
EYES
NOSE
MOUTH
BEARD
HAIR
EARS
HAT
FLOWER
END
```

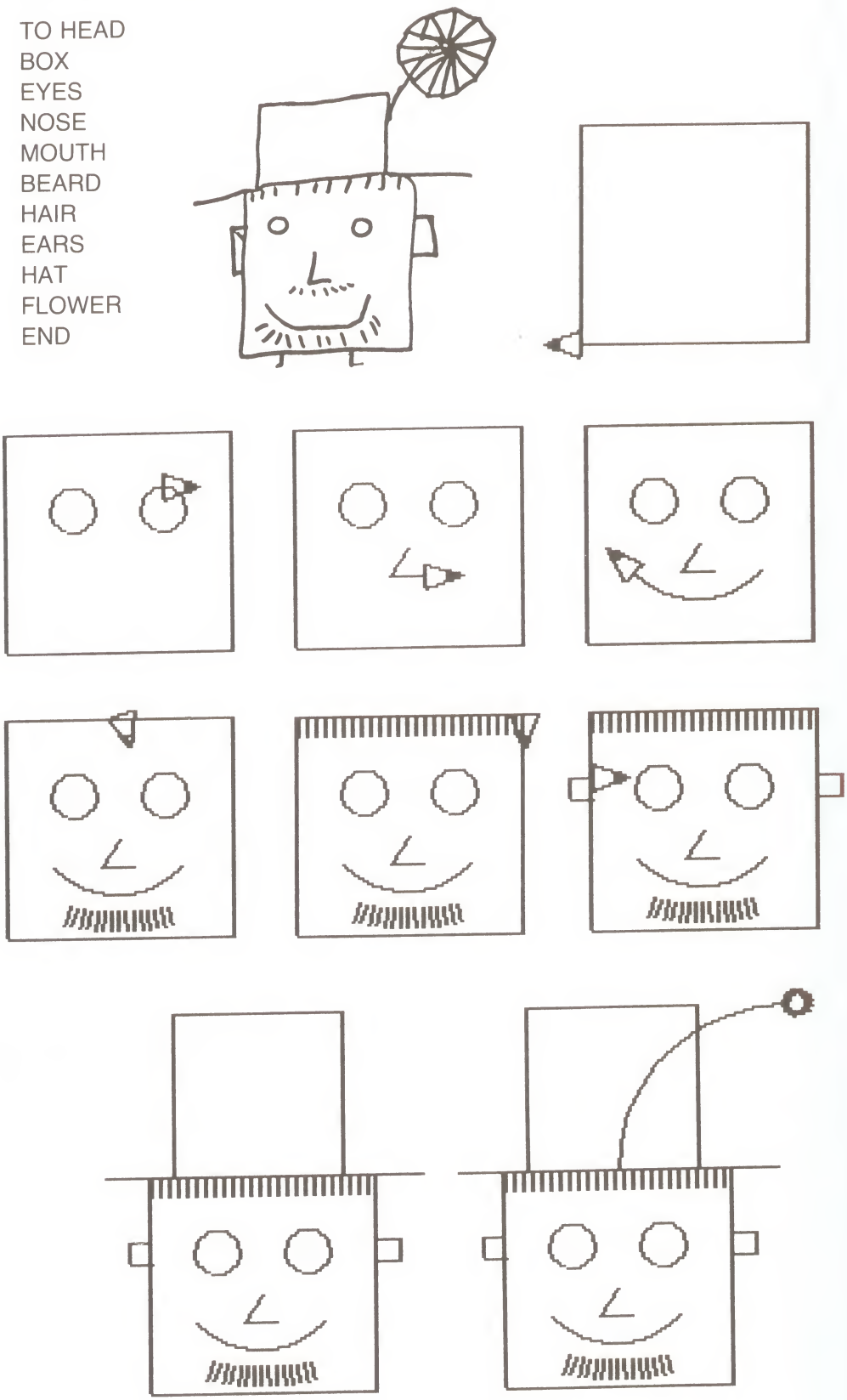


Figure 6.27: Donald's head and how it was assembled.

Bugs and Animals

In this case, "bugs" means drawings of insects. These projects show how a turtle drawing can give the *idea* of an animal using very simple shapes.

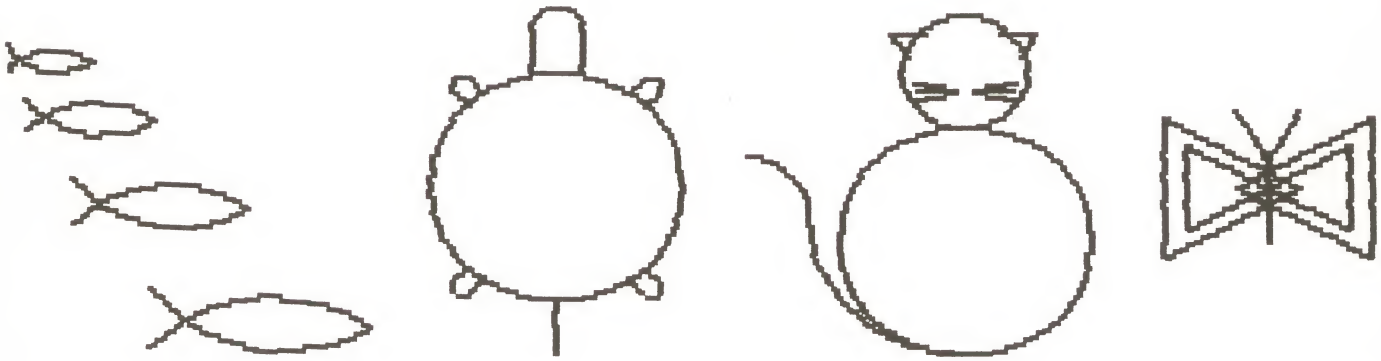


Figure 6.28: A collection of Logo animals.

Trees

This example shows how you can start with a twig and go on to build a whole forest.

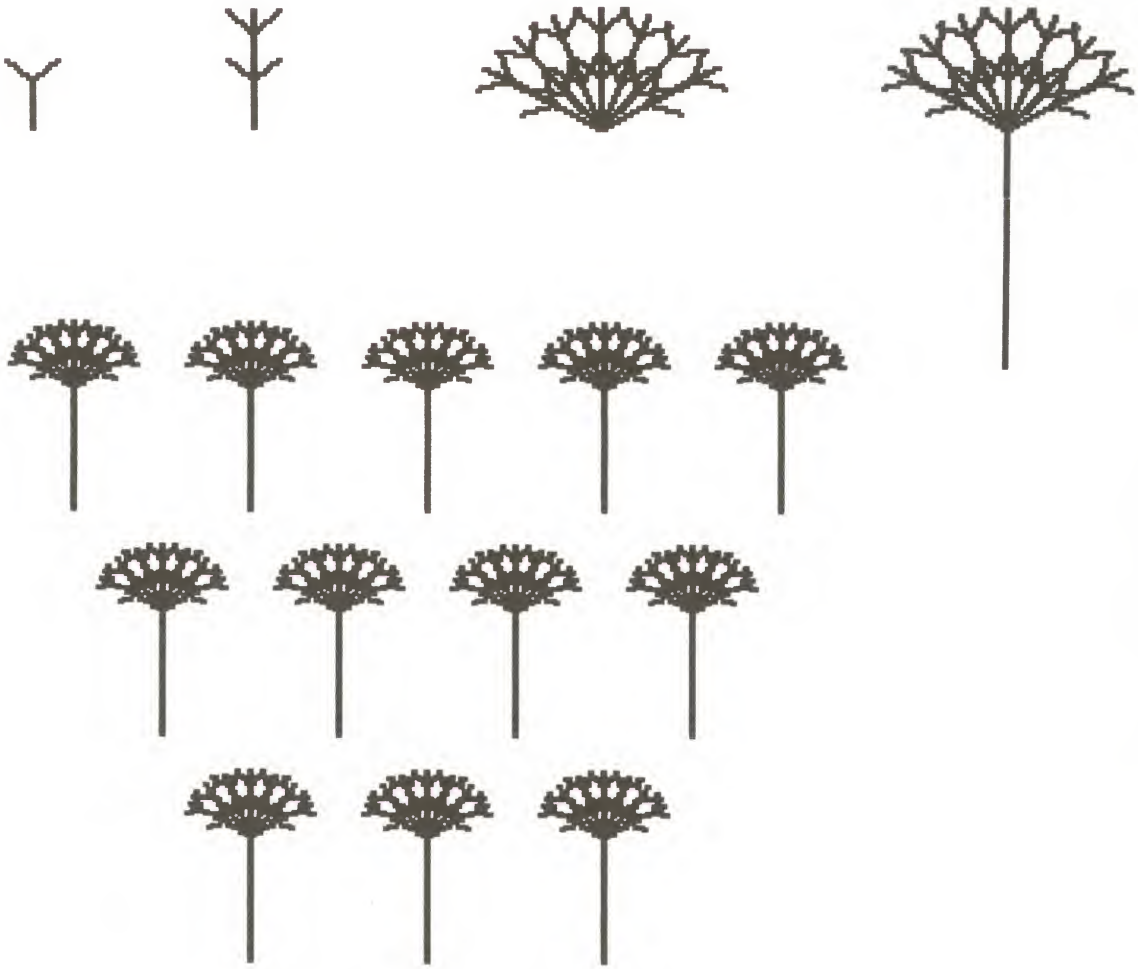


Figure 6.29: A branch, bush, tree, and forest.

Going Places

These all represent vehicles that help people get somewhere.

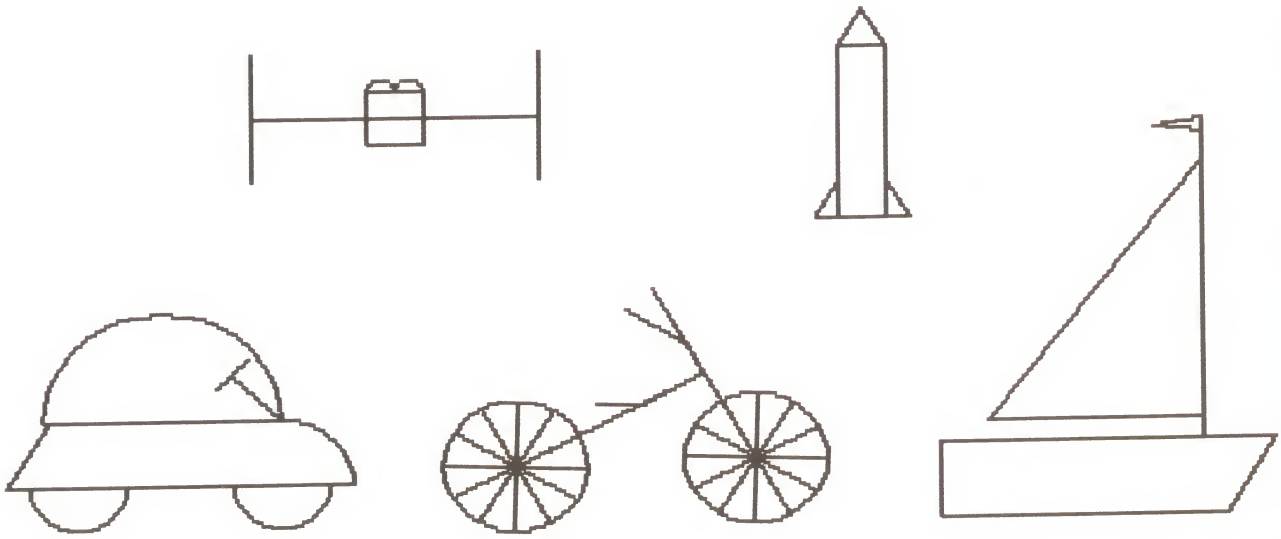


Figure 6.30: Different kinds of transportation.

Baseball

Two boys worked on this project together. They made a general plan, and then each made different parts. It was a big project that was part of an even bigger project—a Logo baseball game. Even though the bigger project was never finished, the boys had a lot of fun working on this part and were quite satisfied with what they had done.

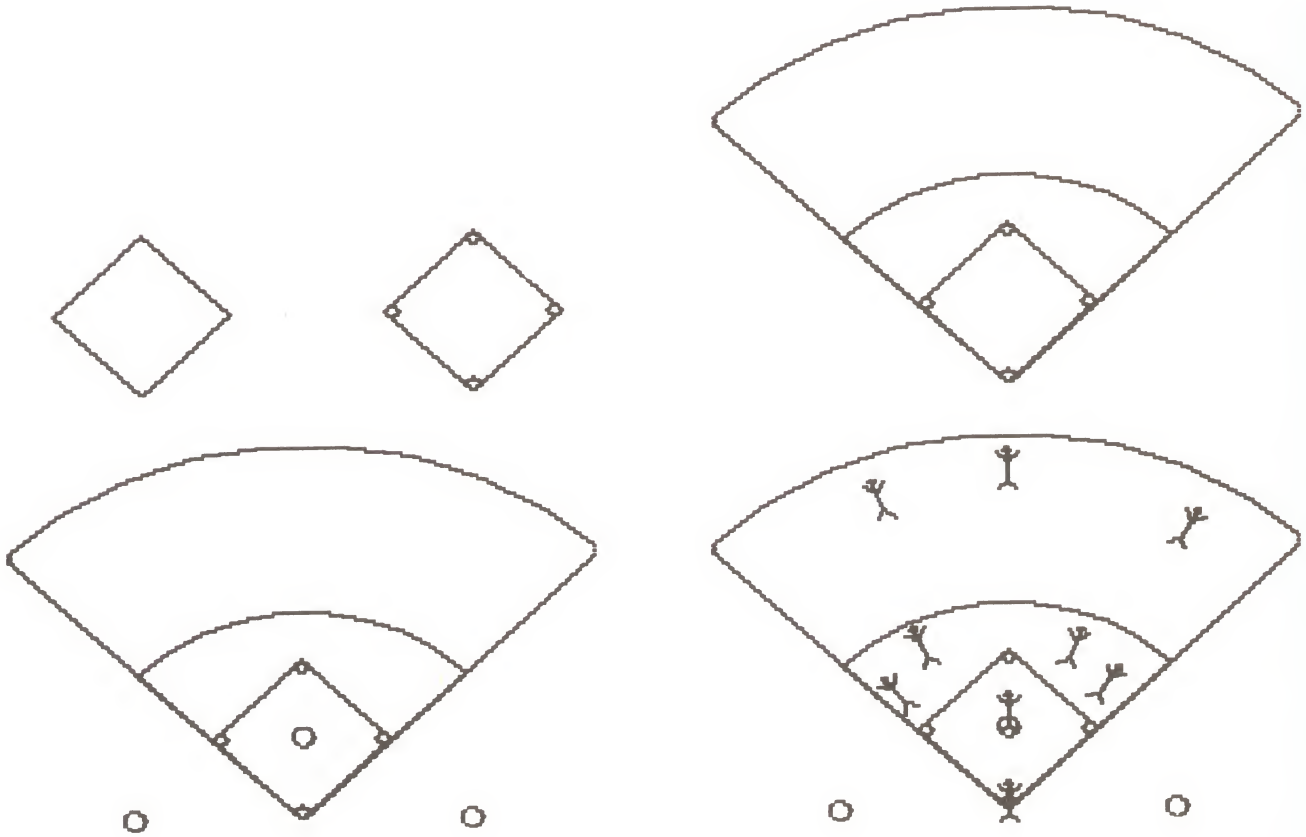


Figure 6.31: Constructing a baseball field.

CHAPTER 7

Command	Short Form	Examples With Inputs
+		FORWARD :SIZE + 10, PRINT 5 + 3
IF		IF :SIZE < 10 [STOP]
STOP		IF :SIZE < 10 [STOP]
<		IF :SIZE < 10 [STOP]
>		IF :ANGLE > 90 [STOP]
=		IF :SIZE = 100 [STOP] PRINT 5 = 3 + 2
HEADING		IF HEADING = 0 [STOP] PRINT HEADING

LWAL Procedures Disk files used: "CIRCLES
New tool procedures used: none.

7 Variables

In this chapter you'll learn how Logo keeps track of information using *variables*. A *variable* is a *piece of information with a name*. The computer stores the name and the information in its working memory. You can easily change or *vary* the information whenever you want. That's one reason it has the name *variable*.

You can store a lot of different kinds of information with variables. In Logo a variable can be a number, a word, or a list. You'll learn more about different kinds of variables in Chapter 9. All the variables in this chapter are going to be numbers.

There are two ways to *create* a Logo variable. In Chapter 9 you'll learn how to use a built-in Logo command, MAKE, to create Logo variables. In this chapter we'll use *procedures with inputs* to create variables. Inputs to *Logo commands* like FORWARD, BACK, RIGHT, and LEFT make it possible for those commands to do many different things. Inputs to *procedures* like SQUARE and TRIANGLE make it possible for *them* to do many different things also.

Section 7.1. Inputs that Change the Size of a Design

Look at this design in Figure 7.1.

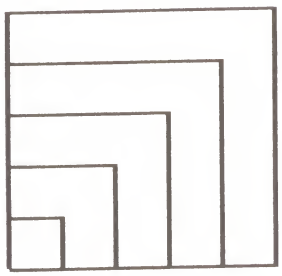


Figure 7.1: A design made from five squares.

This design has five different boxes in it. Without variables we would have to make five different procedures, each one using a different input to FORWARD. Using variables, we can create *one* procedure which uses an *input* to tell it how big the square should be.

Copy this procedure

```
TO SQUARE :SIZE
FORWARD :SIZE
RIGHT 90
FORWARD :SIZE
RIGHT 90
FORWARD :SIZE
RIGHT 90
FORWARD :SIZE
RIGHT 90
END
```

Or, if you like, type it this way

```
TO SQUARE :SIZE
REPEAT 4 [FORWARD :SIZE RIGHT 90]
END
```

You can use any word you like instead of :SIZE, of course. The name :SIZE helps you remember that it determines the *size* of the object, just as the name SQUARE helps you remember which shape the procedure draws.

To make the turtle draw a square, type the command SQUARE *followed by an input*. The input is typed just like the input to FORWARD or RIGHT.

```
SQUARE 30
SQUARE 40
SQUARE 50
etc.
```

Now you can make a procedure to draw the design shown in Figure 7.1.

```
TO BOX5
SQUARE 10
SQUARE 20
SQUARE 30
SQUARE 40
SQUARE 50
END
```

PITFALL

The : symbol used with Logo variables is something new. There are two special rules for its use.

1. Always type :SIZE without a space after the :. If you leave a space and type : SIZE, Logo will complain.
2. Don't type the : at all when you are giving a value for the size. Watch what happens if you do.

```
SQUARE :30
30 HAS NO VALUE
```

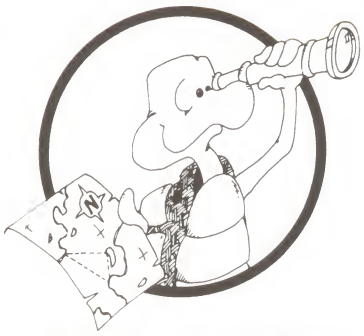
What this means is that there is no *variable* named *30*. When you type :30, Logo looks for something *named* 30, to use its value. When it can't find it, Logo complains.

You can also use variables to draw the STAR and TRIANGLE shapes from Chapter 5.

```
TO STAR5 :SIZE
REPEAT 5 [FORWARD :SIZE RIGHT 144]
END
```

and

```
TO TRIANGLE :SIZE
REPEAT 3 [FORWARD :SIZE RIGHT 120]
END
```



Make some designs using different sized stars and triangles.

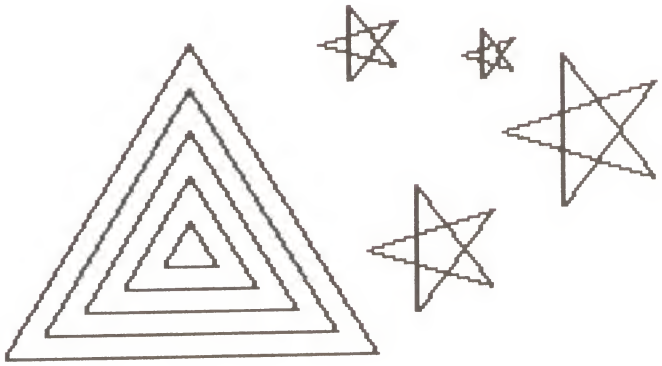
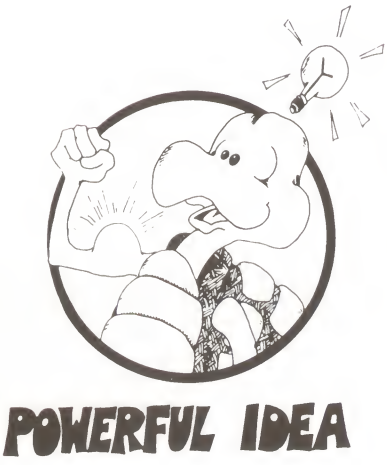


Figure 7.2: Variable-sized stars and triangles.



There are a lot of ways to think about variables. Figure 7.3 shows what happens when you use a procedure with inputs.

1. When you teach the computer a procedure with an input, such as TO SQUARE :SIZE, Logo stores *two* things in its memory—the list of instructions for the procedure, and an empty slot with the *name* of the input, "SIZE on it. The slot is empty because "SIZE has not been given a value yet.

Figure 7.3a: Logo first stores the procedure with an empty slot called "SIZE.

2. When you type a command with an input, like SQUARE 20, Logo puts the value 20 in the box and calls a SQUARE worker to carry out the procedure.

Figure 7.3b: Logo puts the input, 20, in the empty slot.

3. Logo calls SQUARE and puts a value of 20 in its input pouch.

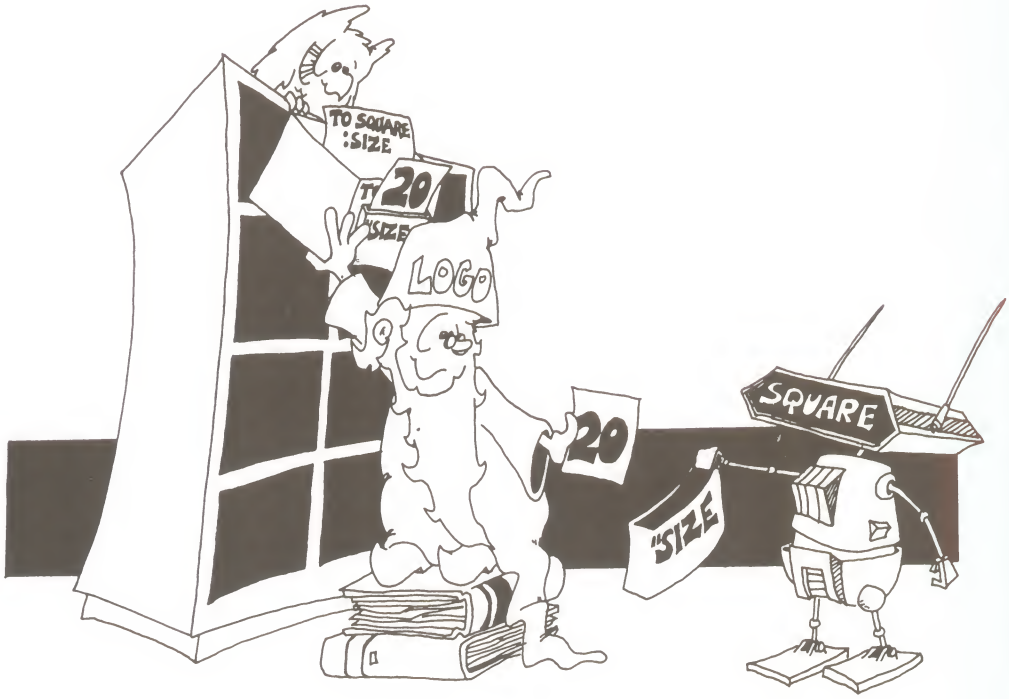


Figure 7.3c: Logo calls the procedure SQUARE.

4. Whenever SQUARE comes to an instruction like FORWARD :SIZE, the : tells SQUARE to find the value in the slot called "SIZE and use that value as an input to FORWARD.

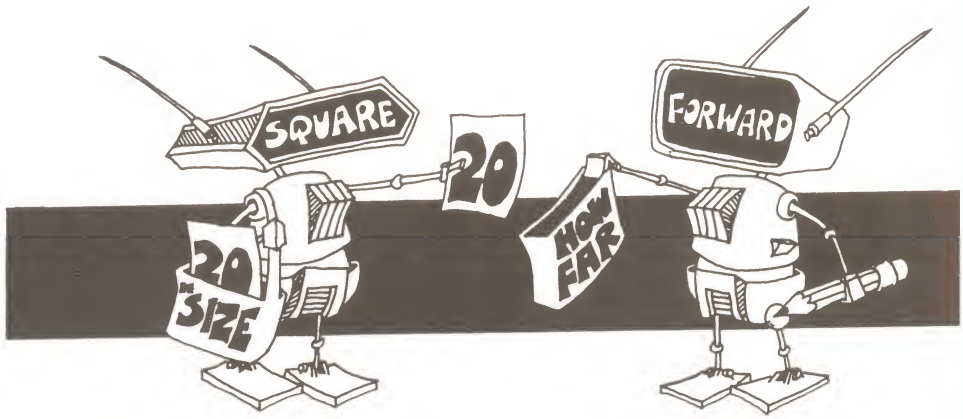


Figure 7.3d: SQUARE gives FORWARD the value, 20, to use as its input.

HELPER'S HINT

The most important thing for a learner to remember right now is how to *use* variables—especially the two rules for avoiding pitfalls given above.

The long explanation about what the computer is doing does not have to be understood the first time variables are used. As the use of variables gets more complex, this explanation may become clearer and more helpful.

There are some other points worth noting here. Logo uses the symbols : and " to make some important distinctions:

:SIZE means *output the value in the slot named* "SIZE.

"SIZE means the name SIZE itself.

SIZE without : or " means a *command or procedure* with the name SIZE.

Without these two symbols, : and ", the meaning of an expression which included the word SIZE would be ambiguous. You can make things even clearer by using the words *dots* and *quotes* to describe these symbols. The expression :SIZE is pronounced *dots-size* and the expression "SIZE is pronounced *quotes-size*. Use these words when you talk about variables.

There's another thing to keep in mind. When you create a procedure with an input, like TO SQUARE :SIZE, the value in the slot "SIZE is kept as *private information* to be used only *within* that procedure. When you give the command SQUARE 20, the value 20 is put into the slot and used until SQUARE finishes doing its job. Once SQUARE is complete, the box that contained 20 is emptied again.

Try this sequence of commands to see what I mean.

```
PRINT :SIZE
SQUARE :SIZE
SQUARE 20
PRINT :SIZE
```

"SIZE has a value only *while* SQUARE 20 is actively functioning. Now try this sequence.

```
MAKE "SIZE 50
PRINT :SIZE
SQUARE :SIZE
SQUARE 20
PRINT :SIZE
```

MAKE is the other way of creating a Logo variable. It creates a *public* slot named "SIZE and puts the value 50 in it. There are now *at least two* slots named "SIZE, the private slot belonging to SQUARE (which is still empty) and the public slot. The command PRINT :SIZE uses the value in the public slot. The command SQUARE :SIZE uses 50, the value in the public slot, and puts it temporarily into the private slot for SQUARE. When you type SQUARE 20, 20 is put into SQUARE's private slot temporarily. When you type PRINT :SIZE, the value from the public slot is used again. If this sounds confusing now, just let it sink in for a while and come back to it later after you've read Chapter 9, where MAKE is introduced "officially."

Section 7.2. Inputs that Change the Shape of a Design

These three designs in Figure 7.4 are all made from the same procedure. It takes an *angle* as an input.

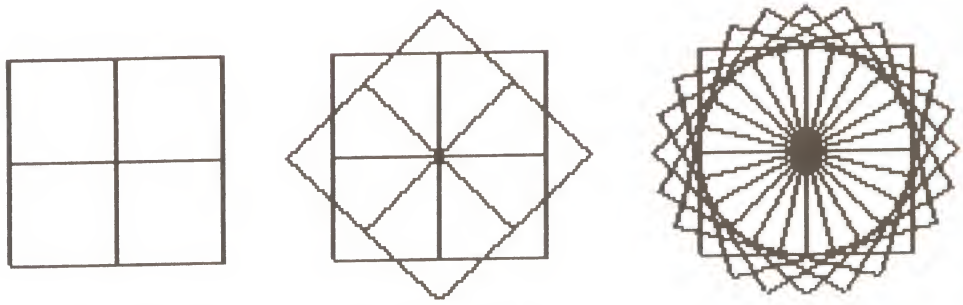


Figure 7.4: Shapes made by SPINSQUARES :ANGLE with inputs of 90, 45, and 15 degrees.

```
TO SPINSQUARES :ANGLE
SQUARE 50
RIGHT :ANGLE
SPINSQUARES :ANGLE
END
```

SPINSQUARES uses *recursion*, the technique you first learned about in Section 5.4. When the SPINSQUARES procedure reaches its last line, it calls *another* SPINSQUARES procedure. As you can see, SPINSQUARES will never stop unless you type **CTRL-G**.

The line in the procedure at which SPINSQUARES calls another SPINSQUARES is called the *recursion line*.

PITFALL

Whenever you write a procedure that has inputs and recursion, be sure to include the variable name in the *recursion line*. Once you include an input in the name of the procedure, the procedure will need a value for that input every time it is called. Just as RIGHT needs a value and uses :ANGLE to know how much to turn the turtle, SPINSQUARES needs a value, :ANGLE, to do its job.

Here's another example. This procedure makes a lot of different stars (if you give it an input larger than 90).

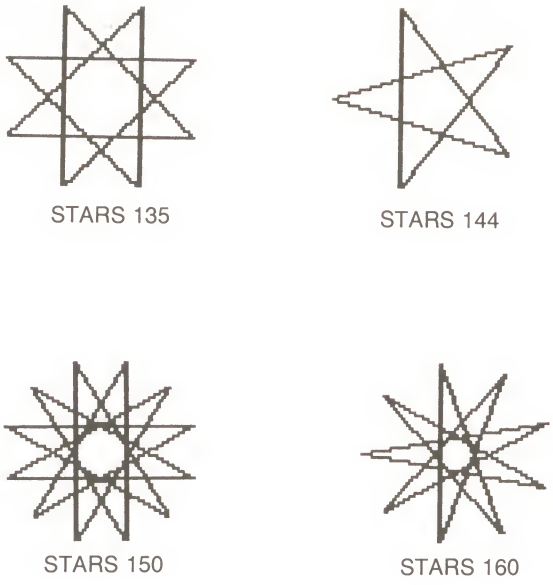
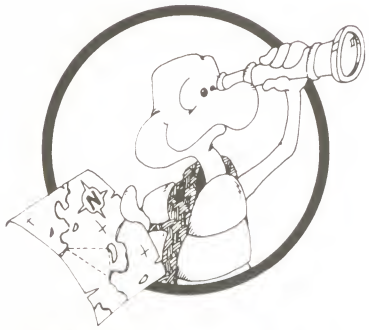


Figure 7.5: Stars created with STARS :ANGLE and inputs of 135, 144, 150, and 160 degrees.

```
TO STARS :ANGLE
FORWARD 50
RIGHT :ANGLE
STARS :ANGLE
END
```



EXPLORATION

- Make a lot of different stars by using STARS :ANGLE with a lot of different input numbers. Can you make a nine-pointed star? A ten-pointed star? A fifteen-pointed star?
- Use the *same input* for SPINSQUARES and STARS. Try a lot of different numbers. The two designs will have a lot in common because they have the same angle input.

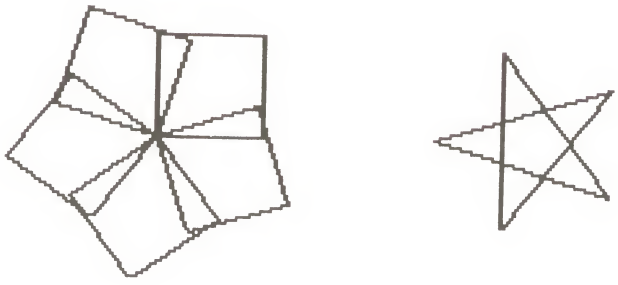


Figure 7.6: Designs created with SPINSQUARES 144 and STARS 144.

- Make some other spinning designs of your own, using any shapes you want.

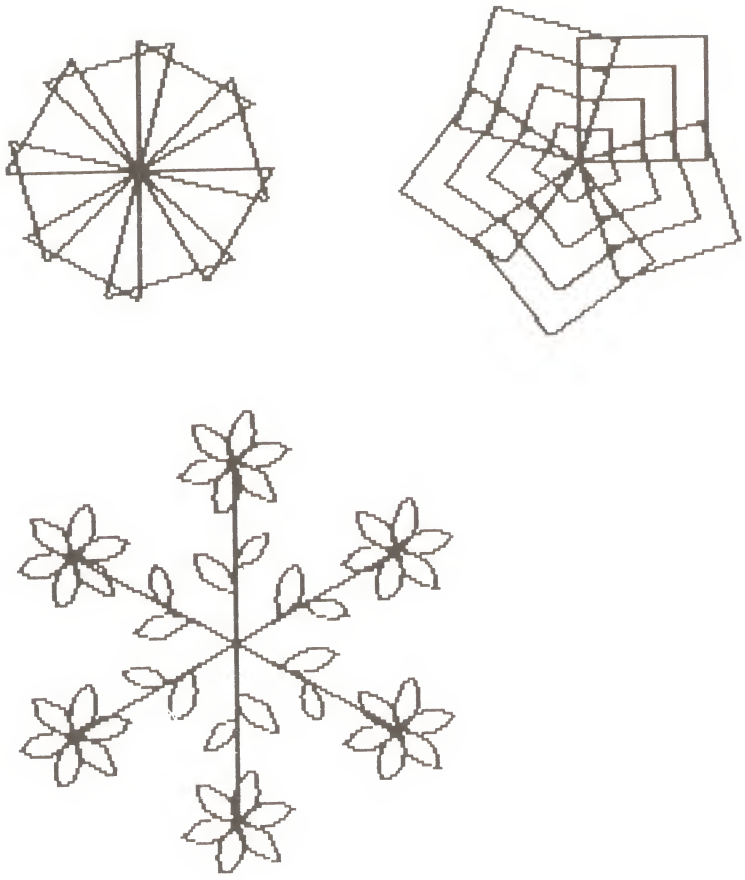


Figure 7.7: Spinning designs made from several different shapes.

Section 7.3. Procedures with Two or More Inputs

This procedure draws stars with different sizes and angles.

```
TO STARS2 :SIZE :ANGLE
FORWARD :SIZE
RIGHT :ANGLE
STARS2 :SIZE :ANGLE
END
```

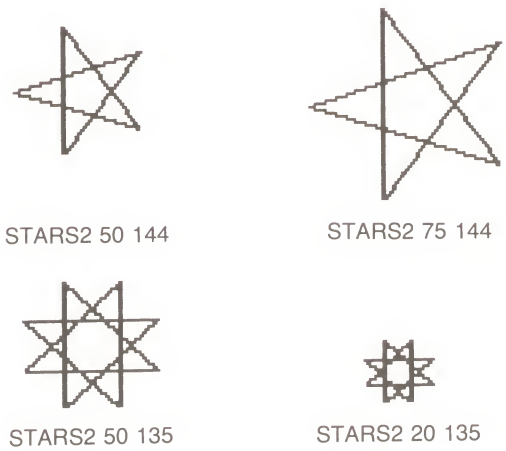


Figure 7.8: Shapes made by STARS2 :SIZE :ANGLE.

In Chapter 8, you'll learn a lot more about this kind of procedure and what you can do with it.

Another shape that needs two inputs is a *rectangle*. It needs two different *sizes* that correspond to its length and width.

```
TO RECTANGLE :LENGTH :WIDTH
FORWARD :LENGTH
RIGHT 90
FORWARD :WIDTH
RIGHT 90
FORWARD :LENGTH
RIGHT 90
FORWARD :WIDTH
RIGHT 90
END
```

Depending on the inputs you use for the length and width, this procedure can make a lot of different shapes.

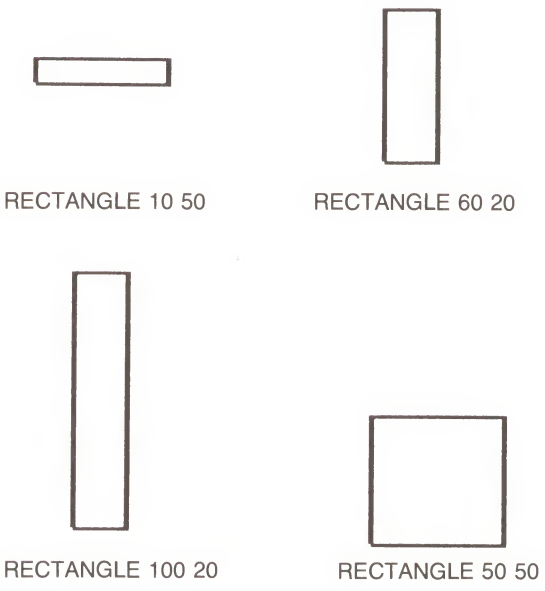


Figure 7.9: Shapes made by RECTANGLE :LENGTH :WIDTH.

A rectangle with *equal* inputs is a square.

Make spinning designs with lots of different shapes and sizes, like those in Figure 7.10.

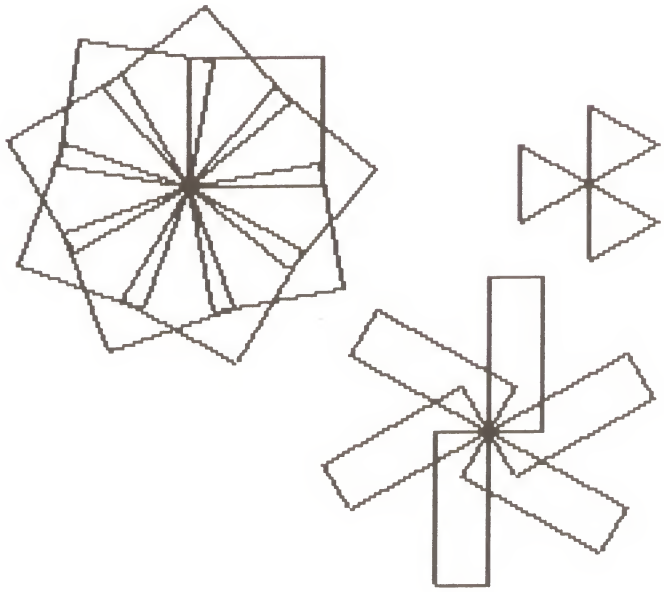


Figure 7.10: Spinning designs made from squares, triangles, and rectangles.

Section 7.4. Subprocedures With Variables

SPINSQUARES in Section 7.2 used SQUARE 50 as a subprocedure. SPINSQUARES2 uses a *variable-sized* square as a subprocedure, and has two inputs, just like STARS2.

```
TO SPINSQUARES2 :SIZE :ANGLE
SQUARE :SIZE
RIGHT :ANGLE
SPINSQUARES2 :SIZE :ANGLE
END
```

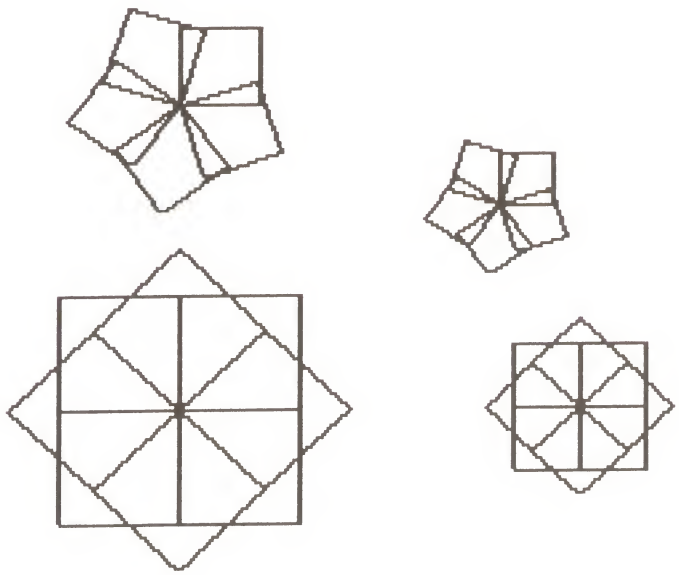


Figure 7.11: Designs created by SPINSQUARES2 with several sets of inputs.

Here's another example using subprocedures with variables. Suppose you made a "house" using fixed-length square and triangle procedures with a length of 50.

```
TO HOUSE
SQUARE
FORWARD 50
RIGHT 30
TRIANGLE
LEFT 30
BACK 50
END
```

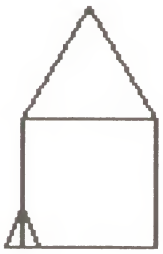


Figure 7.12: Results of the fixed-size HOUSE procedure.

You can make a *variable-sized* house by shifting to variable SQUARE and TRIANGLE procedures and adding a size input to HOUSE.

```
TO HOUSE :SIZE
SQUARE :SIZE
FORWARD :SIZE
RIGHT 30
TRIANGLE :SIZE
LEFT 30
BACK :SIZE
END
```

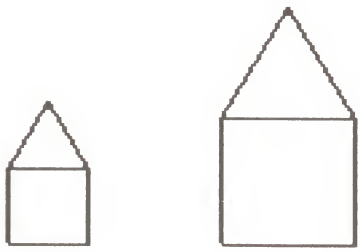


Figure 7.13: Houses drawn by HOUSE 20 and HOUSE 50.

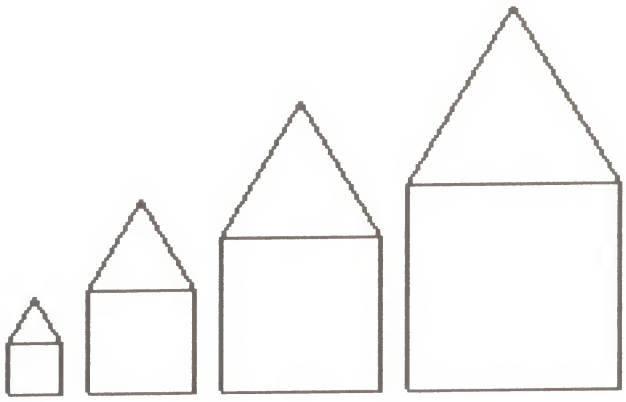
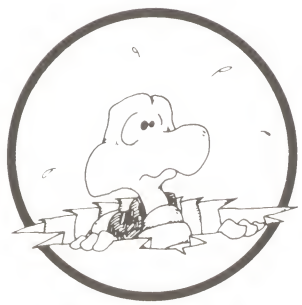


Figure 7.14: A row of houses made with the HOUSES procedure.

```
TO HOUSES
MOVEOVER -100
HOUSE 10
MOVEOVER 10
HOUSE 20
MOVEOVER 20
HOUSE 30
MOVEOVER 30
HOUSE 40
MOVEOVER 40
END
```

HOUSES needs another subprocedure that I call MOVEOVER. MOVEOVER was given a *negative* input in the first line of HOUSES so that the turtle would move over to the *left* instead of *right*.

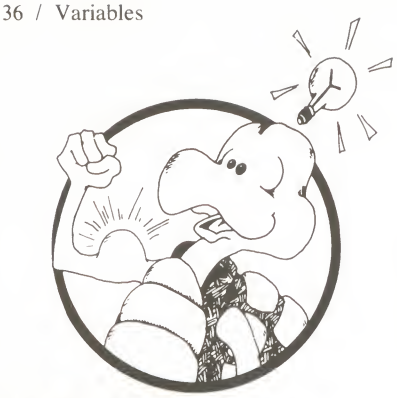


PITFALL

When you type a *negative* number like −100, there's no space between the − and the 100. If you type − 100 with a space, Logo might think you were trying to *subtract* 100 from something.

Here is MOVEOVER. It is a lot like the MOVEOVER procedure used in Section 6.1 except that the distance moved is a variable.

```
TO MOVEOVER :SIZE
PENUP
RIGHT 90
FORWARD :SIZE + 10
LEFT 90
PENDOWN
END
```



Did you notice the line FORWARD :SIZE + 10 in MOVEOVER? One of the best things about using number variables is that you can add, subtract, multiply, and divide them just as you can with any numbers. We'll use arithmetic with variables a lot in the rest of this book. Watch for it!

Remember the FLOWER and BLOSSOM procedures in Chapter 6? They can be made with variables, too. (Remember to load the "CIRCLES file from your LWAL procedures disk before trying these.)

```
TO PETAL :SIZE
RARC :SIZE
RIGHT 90
RARC :SIZE
RIGHT 90
END

TO BLOSSOM :SIZE
REPEAT 6 [ PETAL :SIZE RIGHT 60 ]
END

TO STEM :SIZE
FORWARD :SIZE
PETAL :SIZE
FORWARD :SIZE * 2
END

TO FLOWER :SIZE
STEM :SIZE
BLOSSOM :SIZE
BACK :SIZE * 3
END
```

Notice that *multiplication* was used in STEM and FLOWER. Can you figure out why the turtle went forward :SIZE * 2 in STEM, and back :SIZE * 3 in FLOWER? In case you didn't know, * is Logo's symbol for *multiplication.*

FLOWER can be used to make a garden.

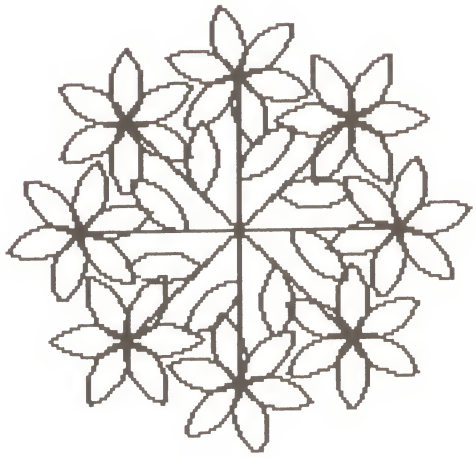
```
TO GARDEN
MOVEOVER -50
FLOWER 30
MOVEOVER 20
FLOWER 40
MOVEOVER 20
FLOWER 50
END
```

Figure 7.15: A GARDEN with flowers of different sizes.

FLOWER can also be used to make a star-like design.

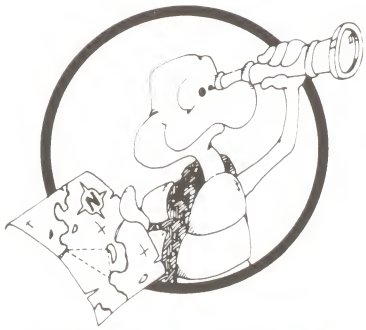
```
TO FLOWERDESIGN :SIZE :ANGLE
FLOWER :SIZE
RIGHT :ANGLE
FLOWERDESIGN :SIZE :ANGLE
END
```

FLOWERDESIGN 30 60



FLOWERDESIGN 60 30

Figure 7.16: Shapes made by FLOWERDESIGN :SIZE :ANGLE.



EXPLORATION

- Use variables with other drawings and designs from Chapters 5 and 6, and then combine them into pictures. The variable sizes let you adjust each part of the drawing so that everything goes together properly. Figure 7.17 shows an example.

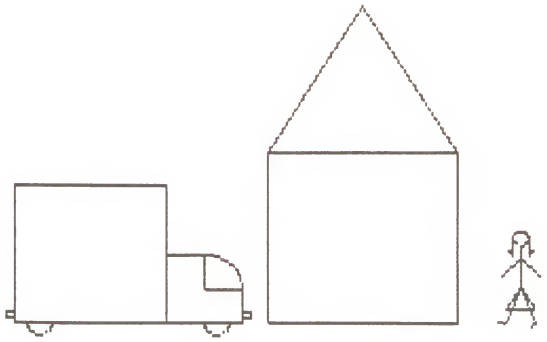


Figure 7.17: This design combines several procedures from Chapters 5 and 6 by using variables to adjust the sizes.

- Make some new procedures with variables.

Section 7.5. Making a Design "Grow" and "Stop"

Several of the earlier examples used designs that grow. For example, BOX5 in Section 7.1 grows.

```
TO BOX5
SQUARE 10
SQUARE 20
SQUARE 30
SQUARE 40
SQUARE 50
END
```

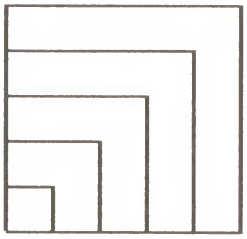


Figure 7.18: A design created by BOX5.

HOUSES, from Section 7.4, makes bigger and bigger houses.

```
TO HOUSES
MOVEOVER -100
HOUSE 10
MOVEOVER 10
HOUSE 20
MOVEOVER 20
HOUSE 30
MOVEOVER 30
HOUSE 40
MOVEOVER 40
END
```

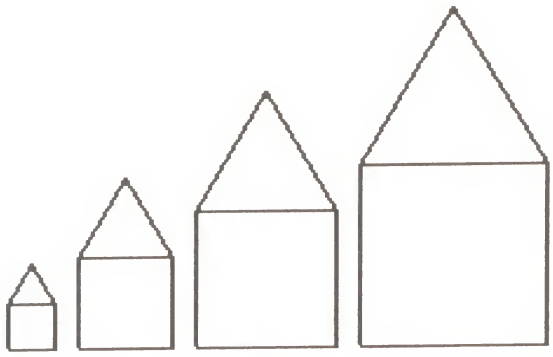


Figure 7.19: A row of growing houses.

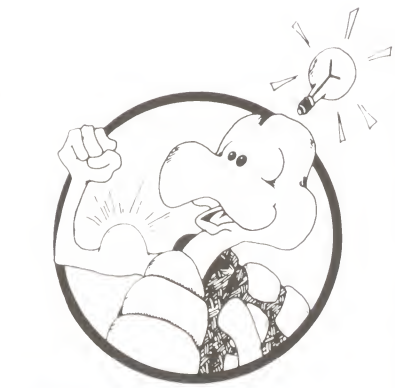
We can replace and simplify these procedures by ones that use *variables* to make them grow.

```
TO GROWSQUARES :SIZE
SQUARE :SIZE
GROWSQUARES :SIZE + 10
END
```

and

```
TO GROWHOUSES :SIZE
HOUSE :SIZE
MOVEOVER :SIZE
GROWHOUSES :SIZE + 10
END
```

The first input to GROWSQUARES or GROWHOUSES gives the *starting* size for the growing shapes: GROWSQUARES 10 or GROWHOUSES 10. Try them and see. The only way to stop them is to type **CTRL-G**.



GROWSQUARES and GROWHOUSES are unusual procedures. They use the subprocedures SQUARE, HOUSE, and MOVEOVER to do all the work of drawing designs. All that GROWSQUARES and GROWHOUSES do is *pass messages that change the values of their inputs!* Look at GROWSQUARES line by line. Suppose you type

```
GROWSQUARES 10
```

GROWSQUARES starts with a value of 10 in its input slot.

```
SQUARE :SIZE
```

GROWSQUARES calls SQUARE and puts the value 10 in the input slot for SQUARE. SQUARE uses this value to draw a square with size equal to 10.

```
GROWSQUARES :SIZE + 10
```

GROWSQUARES calls another GROWSQUARES procedure, but adds 10 to the value of "SIZE in its input slot, and gives the *new*

GROWSQUARES an input of 20 to put in *its* input slot. The entire process starts again.

SQUARE :SIZE
SQUARE is called with an input of 20.

GROWSQUARES :SIZE + 10
A *new* GROWSQUARES is called with an input of 30 (20 + 10), and so on. . . .

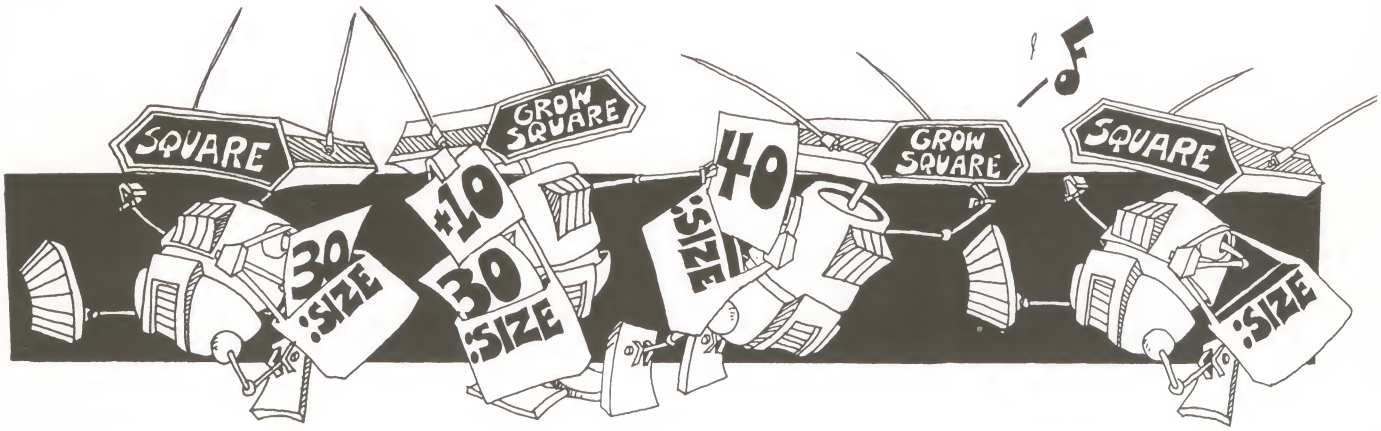
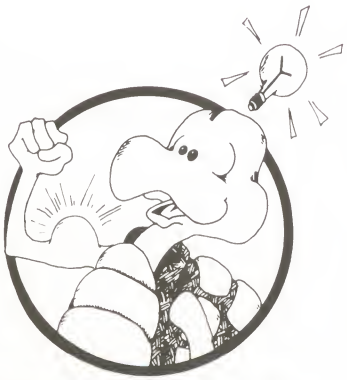


Figure 7.20: Each GROWSQUARES procedure calls another, with its input increased by 10.

As you can see from this explanation, GROWSQUARES will keep on growing squares forever. Try it by typing

GROWSQUARES 10

To stop the process, type **CTRL-G**.



POWERFUL IDEA

There ought to be a way to make the computer stop automatically, and there is! You can tell it to stop the procedure when :SIZE gets bigger than a certain value.

Suppose the largest size you want is 100. Add the following command to GROWSQUARES:

IF :SIZE > 100 [STOP]

IF, STOP and > are all Logo commands. This type of command is sometimes called an *IF statement*, a *conditional* or a STOP *rule*. In this example, IF has two inputs, a *condition*, :SIZE >100, and an action list, [STOP]. When the condition is *true*, the action list is carried out. Otherwise, GROWSQUARES goes on to the next line of the procedure.

The next question is, *where* in the procedure does the STOP rule go? Since GROWSQUARES has only two other commands, it shouldn't be too hard to figure this out by experiments. There are only three possible ways. Try them all and see what happens.

```
TO GROWSQUARES1 :SIZE
SQUARE :SIZE
GROWSQUARES1 :SIZE + 10
IF :SIZE > 100 [STOP]
END
```

or

```
TO GROWSQUARES2 :SIZE
SQUARE :SIZE
IF :SIZE > 100 [STOP]
GROWSQUARES2 :SIZE + 10
END
```

or

```
TO GROWSQUARES3 :SIZE
IF :SIZE > 100 [STOP]
SQUARE :SIZE
GROWSQUARES3 :SIZE + 10
END
```

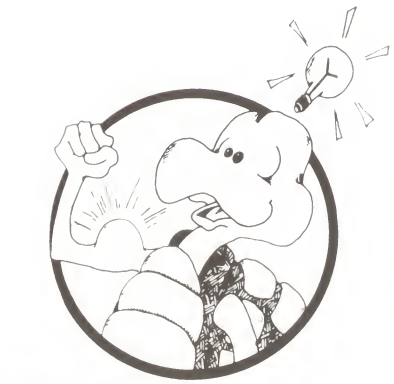
One of these will not stop the procedure at all. One of them will stop it after drawing a square of size 100. One of them will stop it after drawing a square of size 110. Predict which one will do which and then test them all.

PITFALL

Bugs like the ones found in GROWSQUARES1 and GROWSQUARES2 are very common. It seems natural to put the STOP rule *last*, like GROWSQUARES1. If you think a little bit, you'll see why this doesn't work. The computer never gets to the third line in the procedure! Every time it gets to the second line, it calls a new GROWSQUARES1 procedure which starts from the beginning.

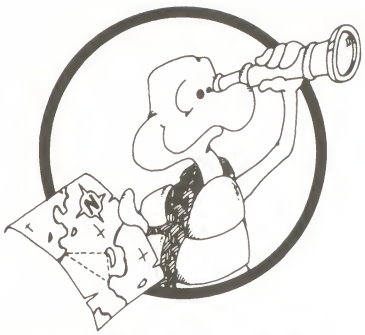
The problem with GROWSQUARES2 is a little different. GROWSQUARES2 does stop, but *after* drawing a square that is bigger than 100. Since the STOP rule comes *after* the SQUARE command, GROWSQUARES2 draws a square first, *then* checks to see if it is too big.

GROWSQUARES3, on the other hand, checks to see if the size is too big *before* drawing the square. So when :SIZE is 110, GROWSQUARES3 will stop *before* drawing a square of that size.



POWERFUL IDEA

GROWSQUARES uses one of the most important ideas in all computer programming. Do something using variables. Check to see if a *condition* is true. If it *isn't* true, change the variables and keep the process going. If it *is* true, *stop* the whole process. You will see this idea again and again in this book and anywhere you do things with computers.



EXPLORATION

- Add a STOP rule to GROWHOUSES. Check to be sure it does *exactly* what you want.
- Make some other growing designs using different shapes. Give them STOP rules too.

HELPER'S HINT

The best way to help people understand this type of process is to step through a procedure line by line, the way I stepped through GROWSQUARES. Sometimes this is called *playing computer*, which is a lot like playing turtle—*pretend you're the computer*, and decide exactly what to do as you come to each command. Then *pretend to be each procedure*. Each procedure can be thought of as a "worker" with a job to do. (You could even do this as a play, with several people taking the role of different procedures.)

It's especially important that each time a GROWSQUARES worker is called upon to do its job, it be thought of as a *new* worker rather than as "GROWSQUARES starting over." In other words, the *old* GROWSQUARES worker calls on a *new* GROWSQUARES worker and gives it a new input (in this case, by adding ten to the old input). The old GROWSQUARES then *waits* until the new GROWSQUARES is finished before it can go on to *its* next line (in this case, END). The idea that each procedure has to wait until the next one is finished is one of the hardest things to understand. Figure 7.21 shows this process at work.

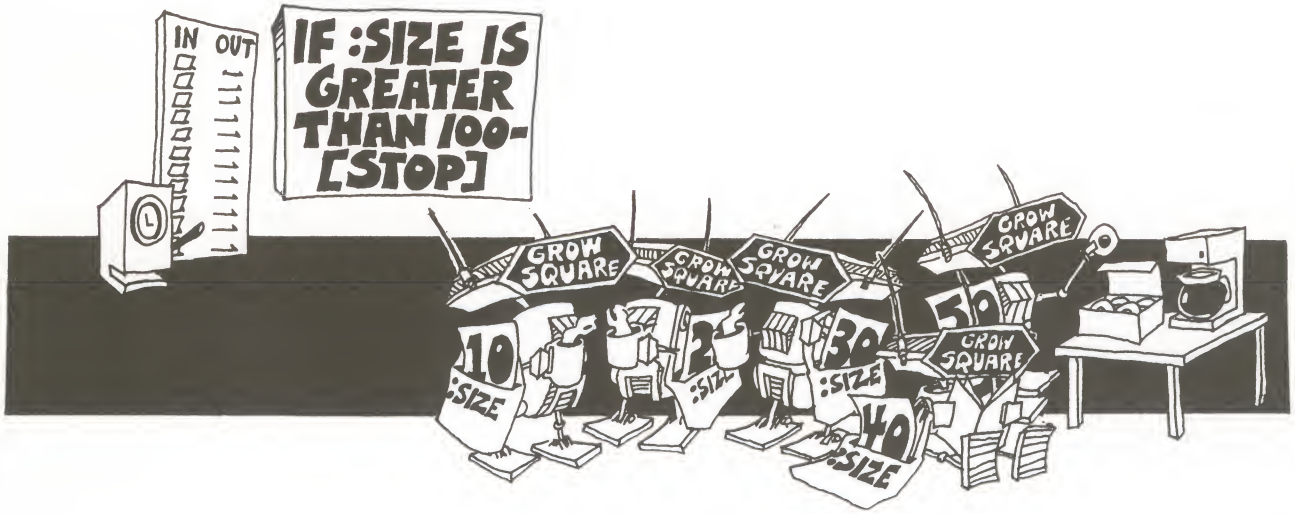


Figure 7.21a: Think of each GROWSQUARES procedure as being different from the last.

Figure 7.21b: Each "old" GROWSQUARES has to wait for the "new" GROWSQUARES to finish its work.

This kind of "explanation" may seem like overkill at this point, since GROWSQUARES is relatively easy to understand. However, talking about procedures as workers who give commands to other workers helps to explain a lot about some of the much more complicated procedures that you will encounter later. This example establishes a *metaphor* for talking about ways that more complicated procedures really work and for *debugging* them when necessary.

Section 7.6. More Procedures that Grow and Stop

SPINSQUARES2 in Section 7.4 had two inputs.

```
TO SPINSQUARES2 :SIZE :ANGLE
SQUARE :SIZE
RIGHT :ANGLE
SPINSQUARES2 :SIZE :ANGLE
END
```

Try it with several different inputs.

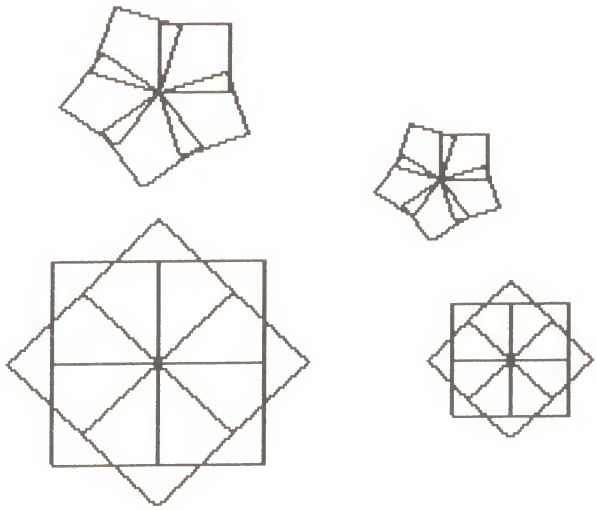
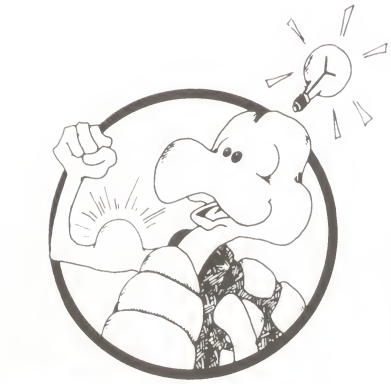


Figure 7.22: Designs created by using different sets of inputs for SPINSQUARES2.

Now let's add a stop rule to SPINSQUARES2 so that it stops after drawing the complete design once.



Since we don't know the *number* of squares in the design (that depends on the angle input), the simplest stop rule is one that tells the computer to stop when the turtle returns to its starting position. In this case, since the turtle started pointing straight up, it should stop when it's pointing straight up again; that is, when its *heading* is equal to 0 again.

```
TO SPINSQUARES3 :SIZE :ANGLE
SQUARE :SIZE
RIGHT :ANGLE
IF HEADING = 0 [STOP]
SPINSQUARES3 :SIZE :ANGLE
END
```

SPINSQUARES3 will stop after drawing its design once.

POWERFUL IDEA

Here's how the STOP rule IF HEADING = 0 STOP works. HEADING is a Logo command that sends a message giving the heading of the turtle. To see this, turn the turtle and tell Logo to print its heading.

```
DRAW
RIGHT 45
PRINT HEADING
45
RIGHT 90
PRINT HEADING
135
```
etc.

= is also a Logo command. It needs *two* inputs. It sends a message with the word "TRUE if its two inputs are the same or the word "FALSE if its two inputs are different. You can see this work when you type

```
PRINT 5 = 4 + 1
TRUE
PRINT 3 = 0
FALSE
```
etc.

IF is a Logo command that needs to be given either the word "TRUE or the word "FALSE as an input. When IF receives "TRUE as an input it carries out its list of actions (in this case, the command STOP). When IF receives an input of "FALSE, it skips the rest of the line, and the computer goes on to the next line.

HELPER'S HINT

It's worth looking at this one example in even more detail. This one line

```
IF HEADING = 0 [STOP]
```

contains *four* Logo *primitives* (built-in commands), IF, HEADING, =, and STOP. Think of each command as a worker with a job to do. The jobs these workers do involve messages that they send and receive. Look at the sequence in which this conditional line is carried out.

First, IF looks for an input. It calls on = to send it a message. IF gives = two inputs. The first input is the number 0. The second tells = to call on the Logo command HEADING to obtain its second input.

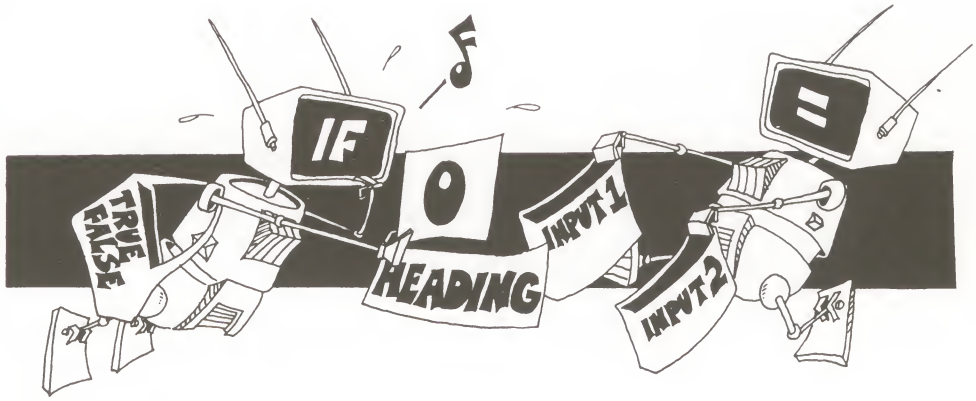


Figure 7.23a

HEADING sends = a message giving the turtle's current direction.

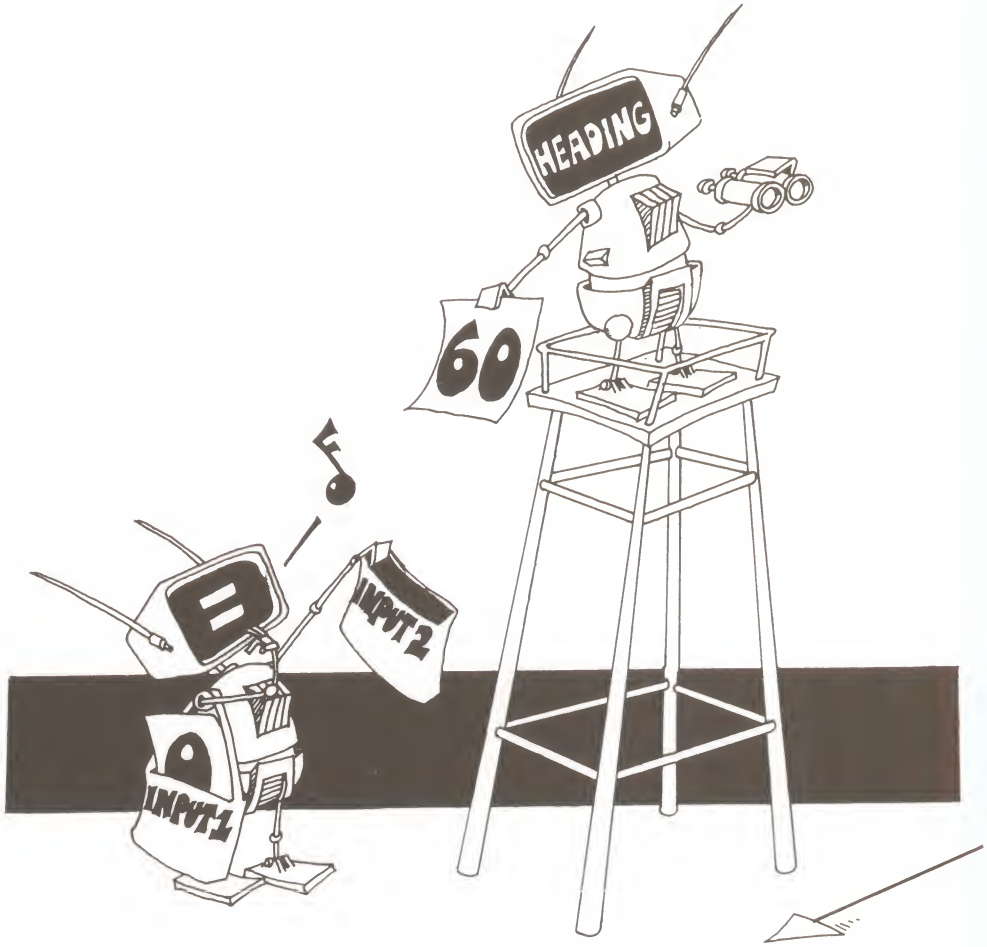


Figure 7.23b

= compares the message sent by HEADING with the number 0. If the two numbers were the same, it would send IF the message, "TRUE. In this case, the turtle's heading is not the same as 0, so = sends the message "FALSE.

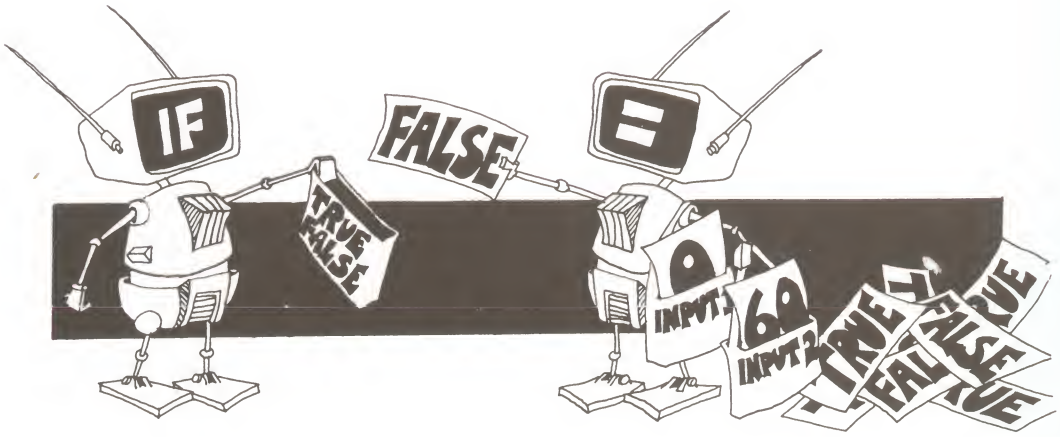


Figure 7.23c

Now, IF has its input. If its input were "TRUE, it would tell the SPINSQUARES3 to carry out its action list, stopping the procedure. In this case, its input is "FALSE. Accordingly, it tells the procedure to ignore the rest of the line, and go directly on to the next line.

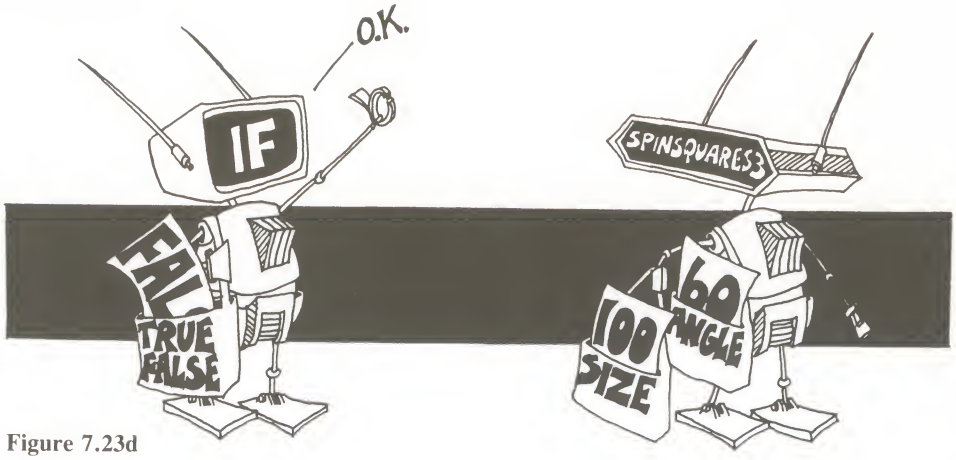


Figure 7.23d

Now the process continues. IF calls =, giving it inputs of 0 and HEADING.

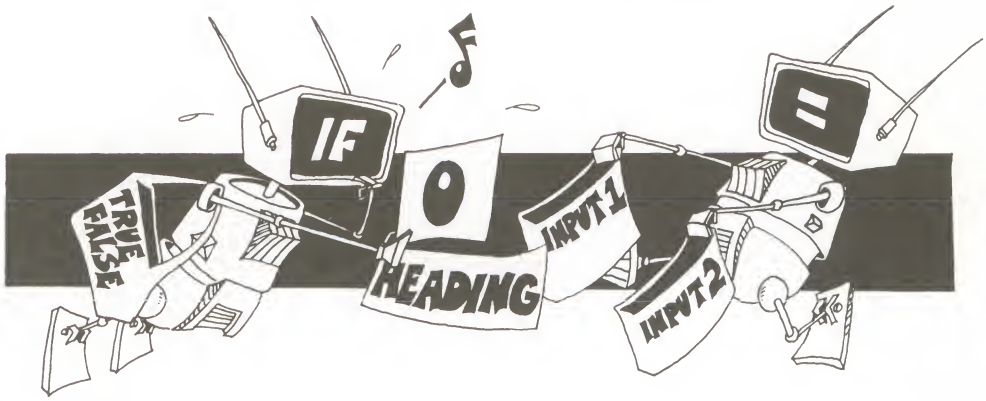


Figure 7.23e

This time the turtle is pointing straight up on the screen. HEADING sends = the number 0 as its second input.

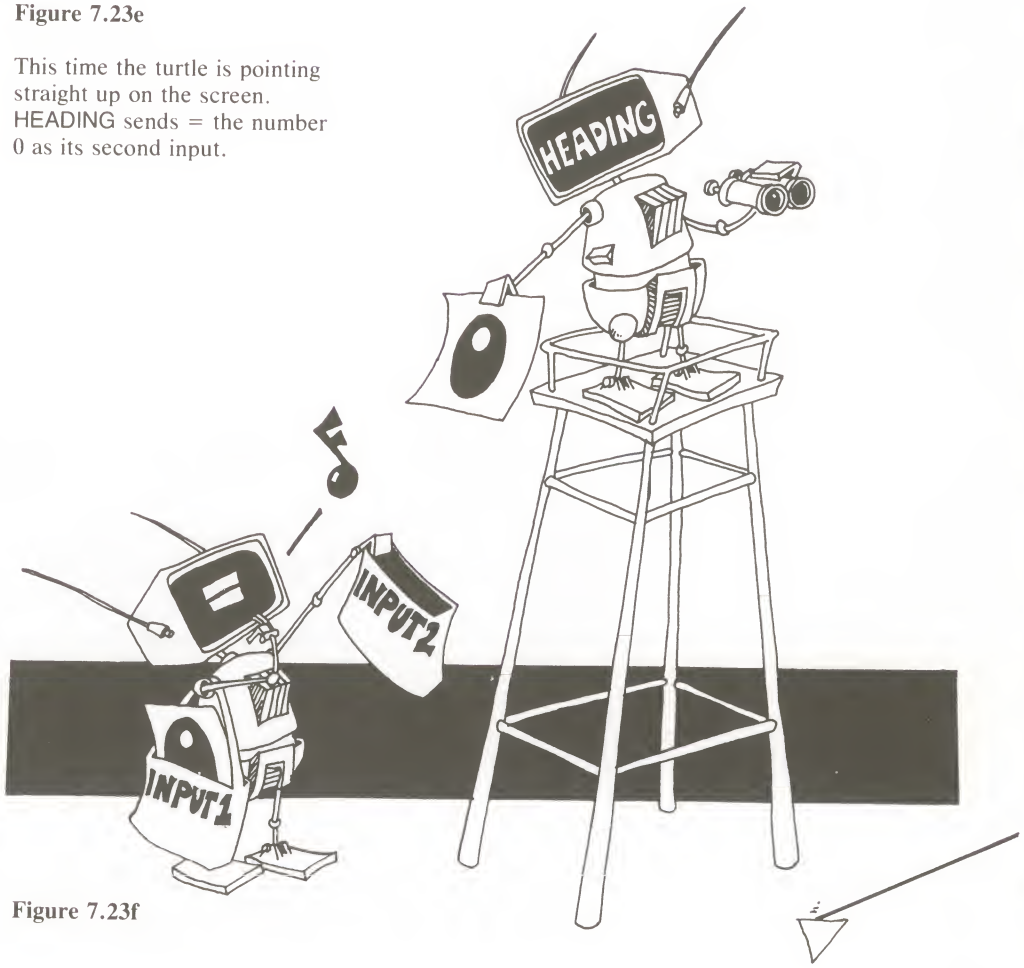


Figure 7.23f

This time, = sends IF the message "TRUE. Since IF's input is "TRUE, the action list containing the STOP command is carried out, making the SPINSQUARES3 procedure stop working, and letting the procedure that called it continue.

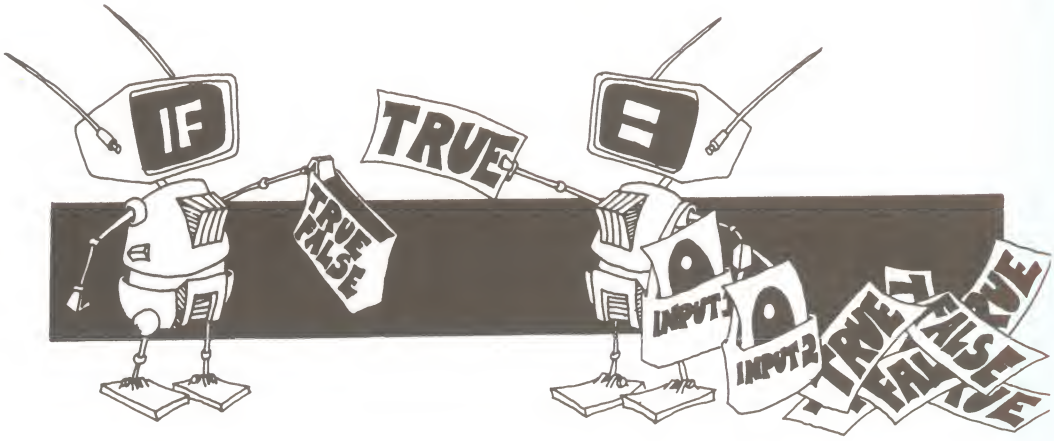


Figure 7.23g

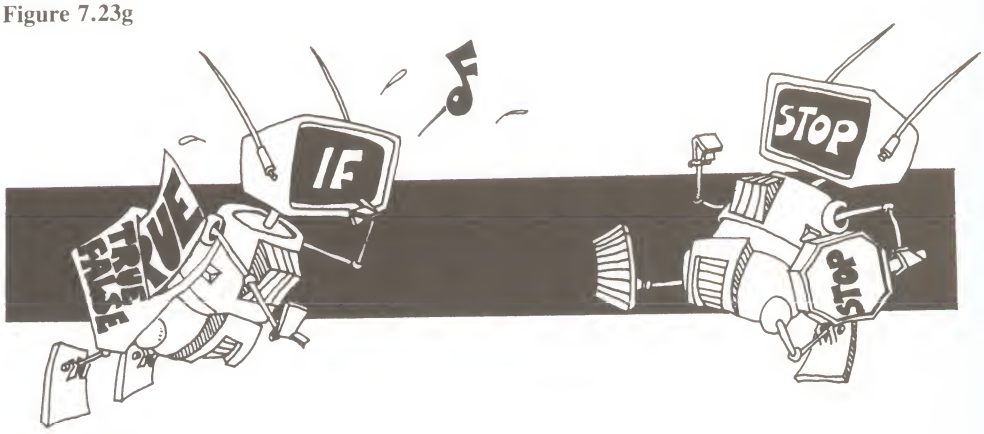


Figure 7.23h

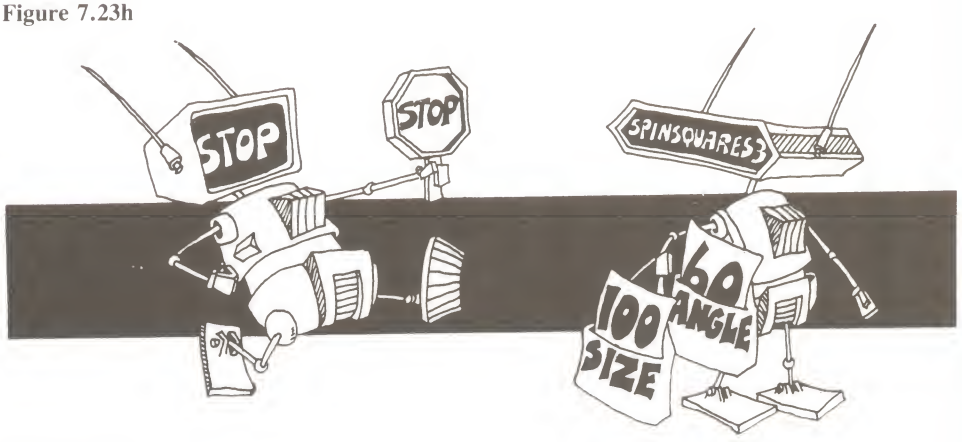


Figure 7.23i

When this procedure stops, the procedure that called it will be able to finish. So will the procedure that called *it*, and so on, up the line. Each procedure has been waiting for the one that is called to stop, so that it can go on to its next line. The next line in each procedure is the command END. So as soon as a SPINSQUARES3 procedure gets word that the one that it called has stopped, it can also stop in turn.

The important idea here is that the procedures stop, one at a time, each one in its turn. The STOP command in the *last* procedure makes only that one procedure stop, not the whole process.

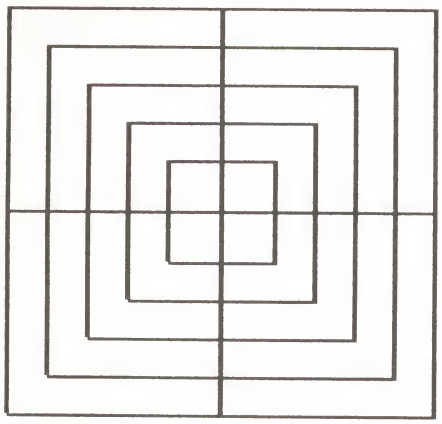
Figure 7.24: After the last procedure stops, each waiting procedure stops in the reverse order from that in which it was called.

Now, let's make a new procedure that makes SPINSQUARES3 *grow*. I'll call this procedure GRSPSQ (short for "grow-spin-squares").

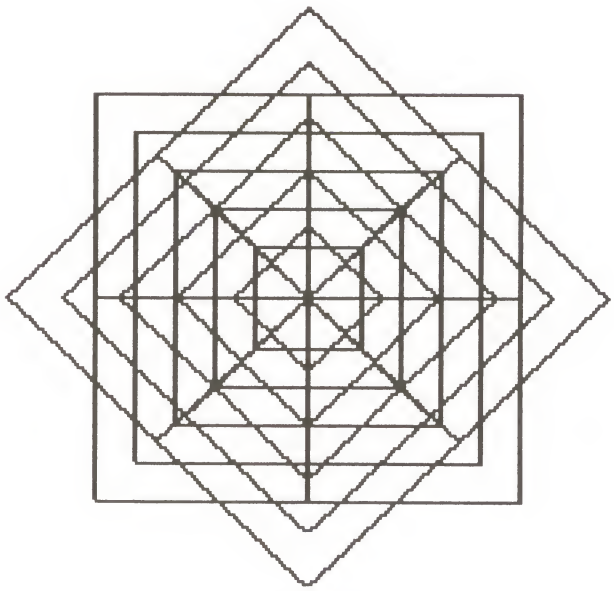
```
TO GRSPSQ :SIZE :ANGLE
IF :SIZE > 100 [STOP]
SPINSQUARES3 :SIZE :ANGLE
GRSPSQ (:SIZE + 20) :ANGLE
END
```

The parentheses around (:SIZE + 20) in the last line are *not* needed by Logo. Logo will do the same thing with or without them. They are there to help *people* who read the procedure understand that (:SIZE + 20) is *one* input to GRSPSQ.

The *angle* input determines *shape* of the design. The *size* input will determine the *starting* size. Figure 7.25 shows two examples.

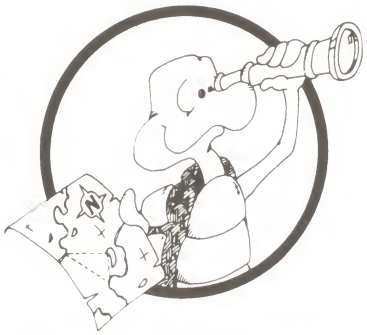


GRSPSQ 20 90



GRSPSQ 20 45

Figure 7.25: Designs created by GRSPSQ :SIZE :ANGLE.



EXPLORATION

- Try GRSPSQ with a lot of different angle inputs. (The size input really doesn't do much in this case.)
- Remove the STOP rule from the procedure and see what happens to your design as the shapes wrap around the screen.
- Make other procedures like GRSPSQ, using shapes other than squares as the starting point. For example, a triangle or even a flower might make a nice design.

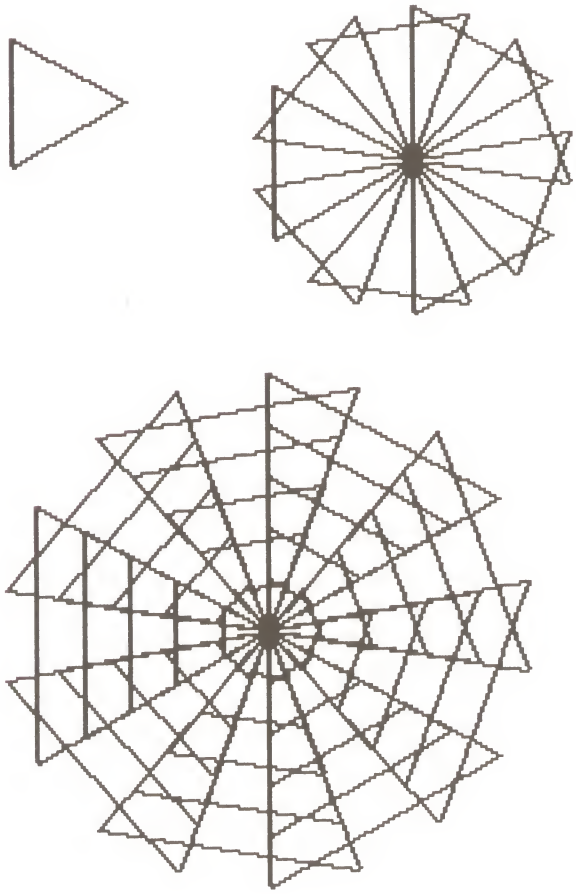


Figure 7.26: Spinning designs made using triangles instead of squares.

CHAPTER 8

Command	Short Form	Examples With Inputs
MAKE		MAKE "START HEADING
		MAKE "SIZE 50
–		PRINT 35 – 10
		FORWARD :SIZE – 10
*		PRINT 3 * 5, FORWARD :SIZE * 3

LWAL Procedures Disk files used: none
New tool procedures used: none

8 POLY and Its Relatives

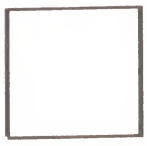
In this chapter you'll learn many more ways of creating and exploring geometric designs. If you enjoy the mathematical part of the designs, you'll learn a lot of mathematics. If you prefer just making designs, this chapter will show you how to create many more designs with the turtle. If you'd rather do something else with Logo besides turtle drawings, skip this chapter for now, and go on to Chapter 9.

Section 8.1. POLY

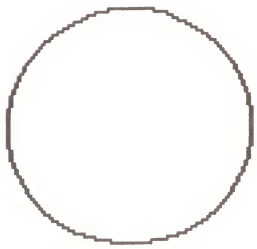
In Chapter 7, Section 7.3, a procedure named STAR2 was used to make *stars* of different sizes and shapes. The same procedure can also be used to make many different *polygons*. In this chapter, we'll give that procedure a different name; we'll call it POLY.

```
TO POLY :SIZE :ANGLE
FORWARD :SIZE
RIGHT :ANGLE
POLY :SIZE :ANGLE
END
```

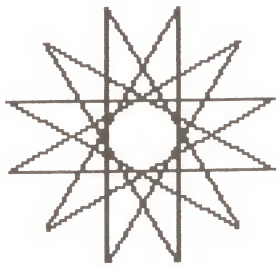
POLY needs two inputs, a size and an angle. The first line of POLY tells the turtle the size to draw each line. The second line tells it the angle to turn each time. The third line tells POLY to call another POLY procedure with the same inputs, and so on. . . . This POLY procedure will never stop unless you type **CTRL-G**. Here are a few designs made with POLY.



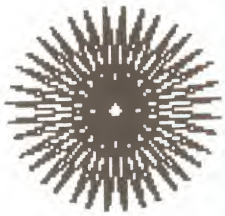
POLY 50 90



POLY 10 10

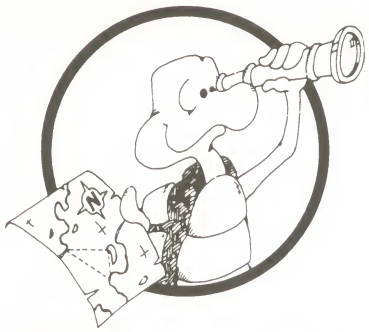


POLY 100 150



POLY 100 170

Figure 8.1: Designs made with POLY :SIZE :ANGLE.



EXPLORATION

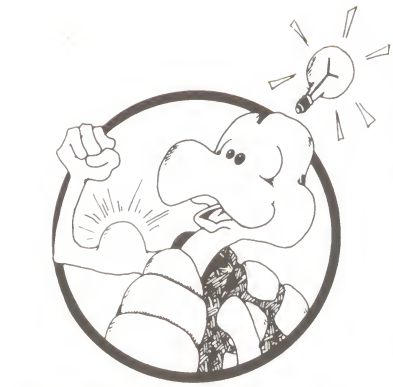
Since POLY can do many different things, it would be a good idea to keep records of everything you do in your Logo journal. Next to each set of inputs, write down what happened. Make a special note for any designs that are interesting enough to try again. A chart like the one shown here is a good way to keep track of explorations with POLY.

Size	Angle	Polygon(P) or Star(S)	Number of sides	Does it Wrap(w) around the screen?
30	30	P	12	n
60	30	P	12	w
30	60	P	6	n
30	80	S	9	n
etc.				

Here are some explorations you can try:

- Experiment with any input numbers you can think of for both size and angle. Use tiny numbers as well as very large ones to see what happens.
- Use a lot of different number combinations. The results might surprise you! Try inputs like POLY 99 99, POLY 12 345, POLY 6 600, and so on.
- Pick one number for the *size* input—50, for example. Then keep that number the same every time, and change the angle input.
  POLY 50 50, POLY 50 75, POLY 50 100, POLY 50 150,
  POLY 50 1000, and so on.
- Keeping the *size* the same, see if can find the angle needed to draw these *polygons*: square, triangle, hexagon (6 sides), octagon (8 sides), and nonagon (9 sides). A pentagon (5 sides) might be a little harder. As a special challenge, try to make a *heptagon* (7 sides).
- Still keeping the *size* constant, try drawing some *stars* with different numbers of points—5 points, 8 points, 9 points, etc. Use your chart to keep track of the angle inputs you use, and the number of points each star has.
- Now keep the *angle* input the same, and see what happens when you change the *size*.
  POLY 10 100, POLY 50 100,
  POLY 100 100, and so on.
- Use very small numbers for both POLY inputs.
  POLY 1 2, POLY 2 5, POLY 5 1, and so on.
- Use a "normal" size input like 50 and 100 with tiny angle inputs like 1 or 2.
  POLY 100 1, POLY 100 2, POLY 100 3, and so on.
- Use a "normal" size input with giant angle inputs like 1000 or 20000.
  POLY 100 1000, POLY 100 5000,
  POLY 100 20000, and so on.

- Use "normal" angle inputs like 60 or 150 with tiny size inputs or giant size inputs.
  POLY 1 150, POLY 3 60, and so on or
  POLY 1000 150, POLY 5000 60, and so on.
- Keep both inputs *equal*. Use all kinds of numbers—small, medium and large.
  POLY 1 1, POLY 2 2, POLY 20 20,
  POLY 90 90, POLY 150 150, POLY 1000 1000, and so on.
- Try some very special angles, and see what happens.
  POLY 20 0, POLY 20 180, POLY 20 360
- Write down your own ideas for POLY explorations, and try them.



There is a mathematical rule connecting the number of sides (or points) of a POLY shape with the angle input that is used to make it. Have you discovered it yet? If you know the rule, you can predict how many sides a POLY design will have for any angle. Here's a little quiz. I'll tell you an angle, and you guess how many sides there will be. Write down your guess, and then check to see if you were right. You'll have to try it out on the computer to find out. (The first group of angles all draw polygons. The second group all draw stars.)

- How many sides for each of these angles: 30, 60, 90, 120, 180?
- Can you find a rule connecting the number of sides with the angle?
- How about these: 80, 150, 160, 200?
- Can you find a rule for these angles?

I'll give you one hint—the rule involes the number 360. Every time the turtle comes all the way back to where it started, it turns exactly 360 degrees at least once. This is sometimes called the *Total Turtle Trip Theorem*. *A Total Turtle Trip* is a series of steps that brings the turtle back to exactly the same position and heading from which it started.

Figure 8.2: A "Total Turtle Trip." The turtle returns to where it started.

Here's another way to say the Total Turtle Trip Theorem: Whenever it makes a total turtle trip, the turtle turns exactly 360 degrees one or more times. It might turn 360 degrees once, twice (720 degrees total), three times (1080 degrees total), or more.

That's as much as I will tell you about the rule connecting the angle and the number of sides of a POLY shape. When you know the rule, you'll be able to predict the number of sides correctly for every polygon and star. You won't need me (or anyone else) to tell you if you're right.

Section 8.2. Making POLY Stop

It would be nice if POLY made the turtle stop drawing, once its design was complete. All we have to do is add a *conditional* command to POLY, telling the computer to check and see if the turtle is back where it started:

```
TO POLY :SIZE :ANGLE
FORWARD :SIZE
RIGHT :ANGLE
IF HEADING = 0 [STOP]
POLY :SIZE :ANGLE
END
```

HEADING is a Logo command that tells which way turtle is pointing at any time. When the turtle starts after a CLEARSCREEN command, its HEADING is always 0. So we tell the computer to check whether the turtle's heading is 0 again. If it is, the turtle is back where it started, and POLY should stop. Try the new POLY now and see what happens. If it works properly, you should be able to make a design like those in Figure 8.3.

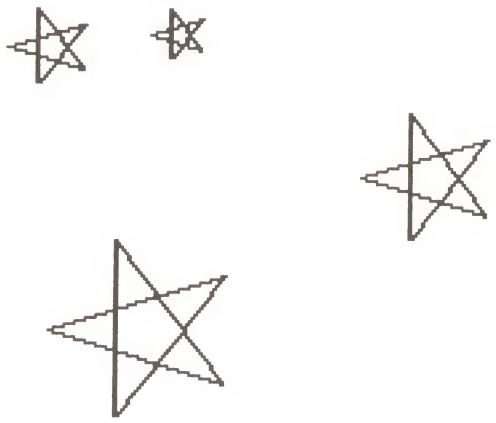


Figure 8.3: Designs made with the new POLY.

Improving the Stop Rule

You might have a problem with POLY if it *starts* drawing with the turtle heading somewhere other than 0. If that's the case, POLY may stop in the wrong position or even never stop because the turtle's original heading

wasn't 0. To fix this, we must make the procedure a little more complicated. Edit POLY to read like this:

```
TO POLY :SIZE :ANGLE
FORWARD :SIZE
RIGHT :ANGLE
IF HEADING = :START [STOP]
POLY :SIZE :ANGLE
END
```

This uses the new variable :START, which stands for the turtle's original heading. But we haven't told the computer what value to use for start. If you type

```
POLY 50 100
```

Logo will complain that "START HAS NO VALUE. . . ." One way to give :START a value is to use the command MAKE.

```
MAKE "START HEADING
```

MAKE needs two inputs. The first is the *name* of a variable, and the second is its *value*. The command MAKE "START HEADING gives the variable named "START a value equal to whatever HEADING the turtle has right now. Now type

```
POLY 50 100
```

Since :START *does* have a value now, POLY should do the right thing. Clear the screen. Turn the turtle to any angle you like, and repeat this process.

```
CLEARSCREEN
RIGHT 99
MAKE "START HEADING
POLY 45 45
```

(You can use *any* angle as an input for RIGHT.)

PITFALL

When you use MAKE, the first input is always a *name*. In Logo, a name is indicated by using a " symbol in front of a word like "JOHN, "SALLY, or "START. The : symbol is used for the *value* of the variable. "START is *not* the same as :START. Try this to demonstrate the difference.

```
PRINT "START
PRINT :START
```

Of course it would be boring to have to type MAKE "START HEADING every time you want the turtle to draw a POLY. So let's put it into a new superprocedure called POLY1 that uses POLY as a subprocedure.

```
TO POLY1 :SIZE :ANGLE
MAKE "START HEADING
POLY :SIZE :ANGLE
END
```

Now try

```
POLY1 50 90
RIGHT 45
POLY1 50 90
```

No matter where the turtle is on the screen, POLY1 should draw a design and then stop. Here are some designs that can be made with POLY1. Try some of them. If you have a color TV, add some color variations.

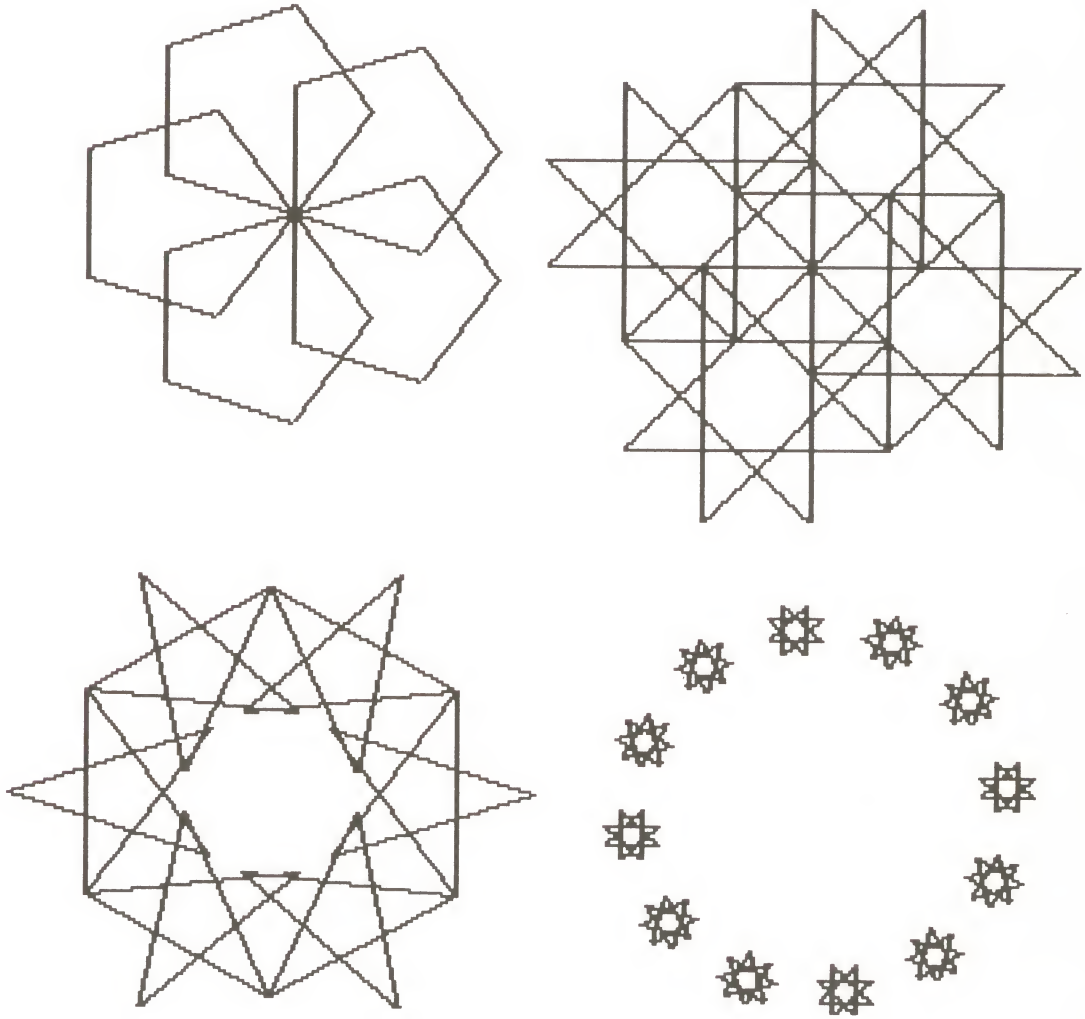


Figure 8.4: Designs made with POLY1.

POLY1 is a very special kind of superprocedure. Its *only* job is to set up the start of another procedure. In this case, it creates a value for the variable named "START and then calls POLY, which uses :START. This type of idea is very important in more complicated programs. We'll use it often from now on.

Section 8.3. Thinking More About Stop Rules

You can read this section now if you want to learn more about Stop rules. If you prefer to make more designs, go on to Section 8.4, "Polyspirals;" then come back and read this section later.

The correct position of a STOP rule in a procedure is very important. All four of the procedures below are identical except for their names and the *locations* of their STOP rules. Try each variation and see what happens.

```
TO POLYA :SIZE :ANGLE
IF HEADING = 0 [STOP]
FORWARD :SIZE
RIGHT :ANGLE
POLYA :SIZE :ANGLE
END
TO POLYB :SIZE :ANGLE
FORWARD :SIZE
IF HEADING = 0 [STOP]
RIGHT :ANGLE
POLYB :SIZE :ANGLE
END
TO POLYC :SIZE :ANGLE
FORWARD :SIZE
RIGHT :ANGLE
IF HEADING = 0 [STOP]
POLYC :SIZE :ANGLE
END
TO POLYD :SIZE :ANGLE
FORWARD :SIZE
RIGHT :ANGLE
POLYD :SIZE :ANGLE
IF HEADING = 0 [STOP]
END
```

One of the most common *bugs* in any computer program is having a conditional command in the wrong position. Understanding these four procedures will help you *debug* this kind of problem. Think through each of the procedures step by step and try to explain why *only* POLYC does the right thing and exactly why each of the other variations doesn't work.

HELPER'S HINT

This exploration is very useful for understanding the kinds of things that can go wrong with conditionals. In helping someone understand this, there are two important things to stress.

First, think through the procedure step by step. Sometimes this is called *playing computer*. Like *playing turtle*, this involves putting yourself in the computer's place and deciding what it would do for each step. Then try the procedure and see what *really* happens.

Second, it's very important to force yourself to *explain* what you expect to happen, and then explain what you think really happened after you've tried it. A major source of confusion with computers is the gap between what someone *expects* to happen and what *really* happens. Since this happens to everyone, it's critical to be able to do this kind of exercise.

There are many ways to explain these things, and no single explanation is best. Since the real learning comes when you confront the difference between what you expected and what really happened, it's important to establish an atmosphere in which people can talk about these things without embarrassment, without feeling judged. This can be especially hard in a classroom, where students are usually expected to know the "correct" answer to something before they speak. The most important objective here would be to break down that expectation. In its place, it would be great to establish an expectation of free discussion of the ways people are thinking about what the computer is doing. An exchange of ideas about what the computer is doing can be a wonderful way to help everyone in the discussion debug misconceptions about what is really happening.

If you are a teacher or parent, the best way to foster this type of atmosphere is to expose bugs in your own thinking. More than anything else, this will help the learners you're working with develop the confidence to risk explaining their thinking in public.

Section 8.4. Polyspirals

In Chapter 7 variables were used to make shapes grow in procedures like GROWSQUARES and GROWSPINSQUARES.

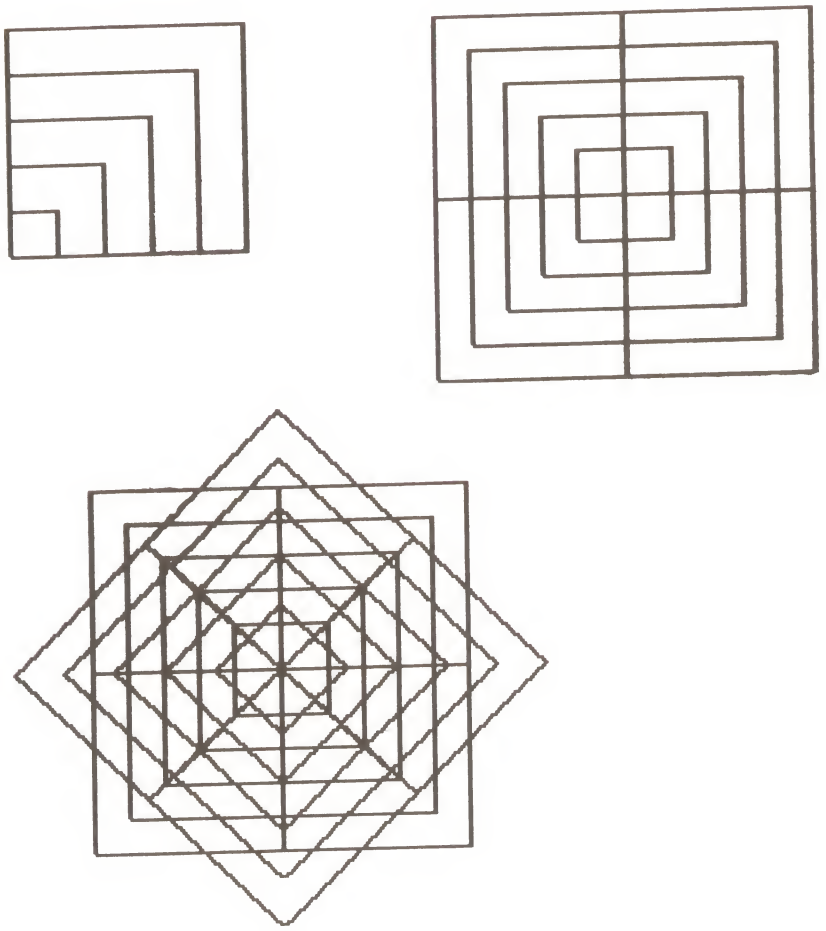
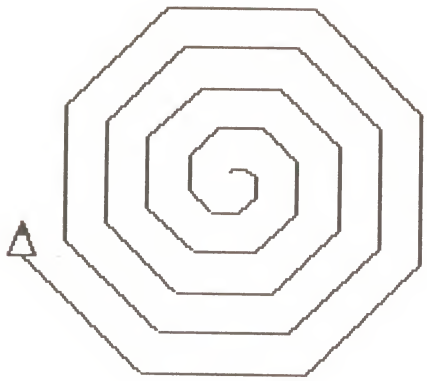


Figure 8.5: GROWSQUARES and GROWSPINSQUARES.

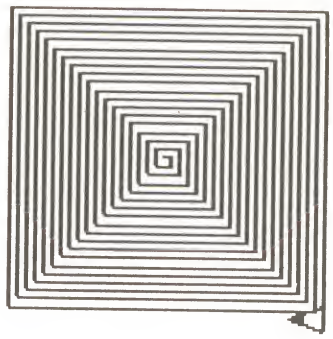
Polyspirals are first cousins to polys. They are polys that grow. In a polyspiral, the size variable is increased or decreased each time the procedure is repeated.

```
TO POLYSPI :SIZE :ANGLE
FORWARD :SIZE
RIGHT :ANGLE
POLYSPI (:SIZE + 1) :ANGLE
END
```

Figure 8.6 shows two examples.



POLYSPI 1 45



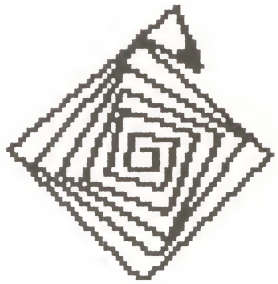
POLYSPI 1 90

Figure 8.6: Polyspiral designs.

- Try POLYSPI with a lot of different *angle* inputs. Some of the most interesting designs are made with angles that are *close* to (but not the same as) the angles used for a particular star or polygon.

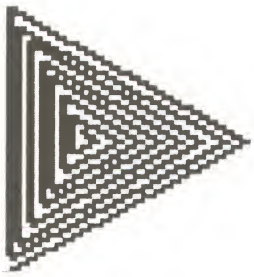
POLYSPI 1 88

POLYSPI 1 90

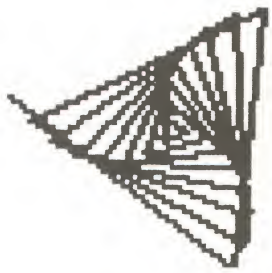


POLYSPI 1 92

POLYSPI 1 118



POLYSPI 1 120



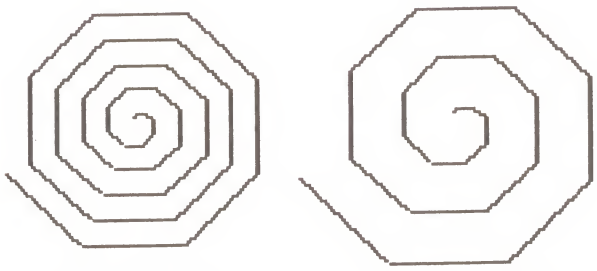
POLYSPI 1 122

Figure 8.7: More polyspiral designs.

- Let some polyspis grow very large before stopping them with **CTRL-G**. See what happens as the designs wrap around the screen.

One way to vary the procedure is to change the *amount* of the increase in size by adding another variable, :INC. (INC is short for "increase." I use INC rather than INCREASE because it's easier to type.)

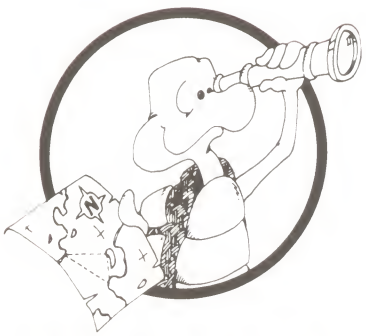
```
TO POLYSPI2 :SIZE :ANGLE :INC
FORWARD :SIZE
RIGHT :ANGLE
POLYSPI2 (:SIZE + :INC) :ANGLE :INC
END
```



POLYSPI2 1 45 1 POLYSPI2 1 45 3

Figure 8.8: Varying the increase in a polyspiral design.

The third input controls the amount of increase in size each time.



EXPLORATION

Try POLYSPI2 with a lot of different values for :INC. Another variation is to include a STOP rule in the procedure.

```
TO POLYSPI3 :SIZE :ANGLE :INC
IF :SIZE > 100 [STOP]
FORWARD :SIZE
RIGHT :ANGLE
POLYSPI3 (:SIZE + :INC) :ANGLE :INC
END
```

Still another variation has the forward step start *large* and grow *smaller* each time through the procedure. (DEC is short for DECREASE.)

```
TO POLYSPI4 :SIZE :ANGLE :DEC
IF :SIZE < 1 [STOP]
FORWARD :SIZE
RIGHT :ANGLE
POLYSPI4 (:SIZE - :DEC) :ANGLE :DEC
END
```

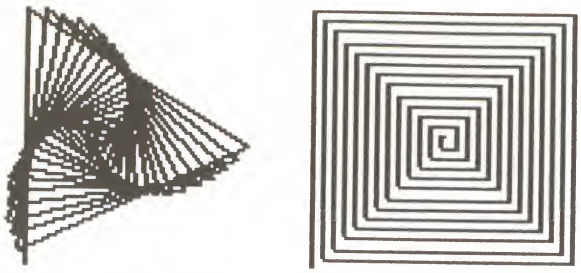


Figure 8.9: Polyspirals with decreasing sizes.

Here are some designs that put several polyspis together. The trick in making interesting designs like this is to choose POLYSPI angles and rotations that work well together. Figure 8.10 shows some combinations that I like.

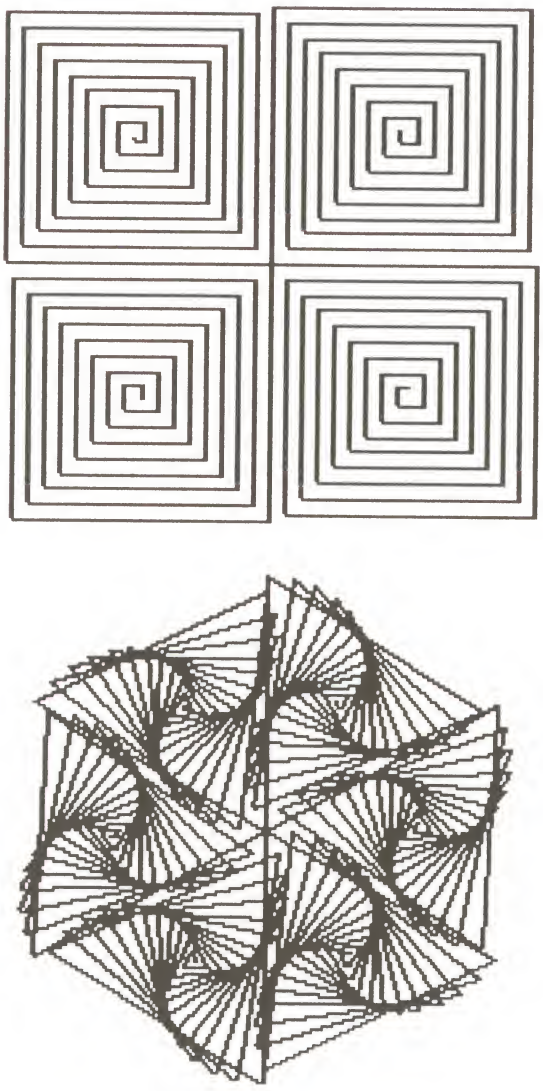
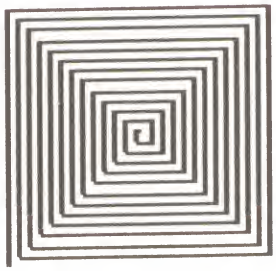


Figure 8.10: Combining *decreasing* polyspirals.

Here's one last variation. You might *see* a POLYSPI design differently if you could see only the corners instead of the sides. POLYSPI5 uses PENUP and PENDOWN in such a way that you only see the *corners* of the design.

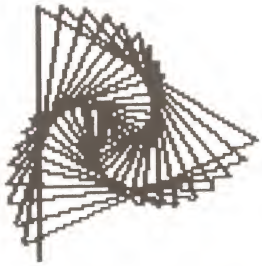
```
TO POLYSPI5 :SIZE :ANGLE :DEC
IF :SIZE < 1 [STOP]
PENUP FORWARD :SIZE
PENDOWN FORWARD 1 BACK 1
RIGHT :ANGLE
POLYSPI5 (:SIZE - :DEC) :ANGLE :DEC
END
```

POLYSPI5 100 90 2



POLYSPI4 100 90 2

POLYSPI5 100 122 2



POLYSPI4 100 122 2

Figure 8.11: Polyspirals with only the "points" showing and those with full lines showing.

Watch what happens as the angle changes by 5 degrees.

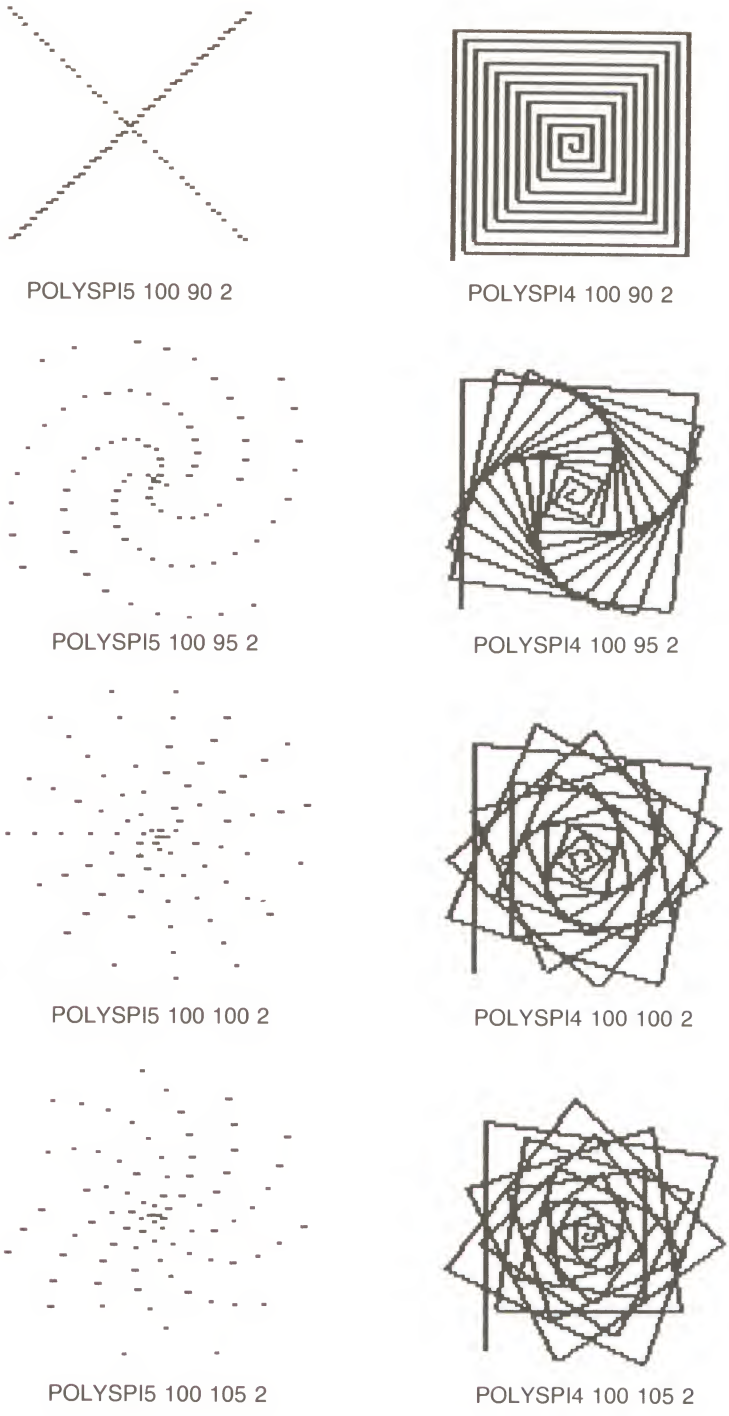
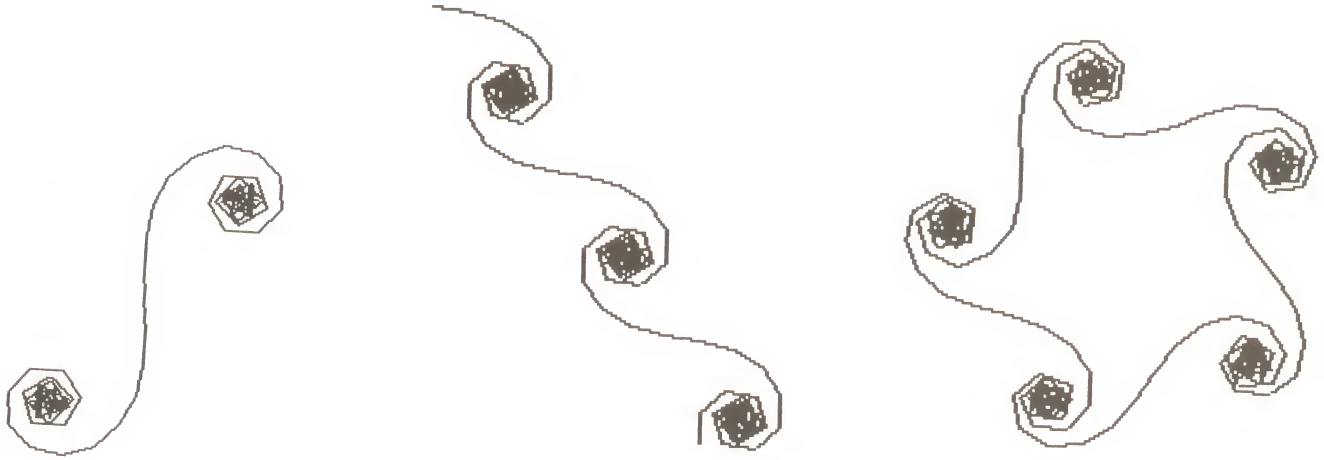


Figure 8.12: Slight changes in the angle of polyspiral designs.

Section 8.5. Inspirals

Inspirals are also cousins of POLY and POLYSPI. Instead of changing the *size* after every turn, the *angle* is increased or decreased.

```
TO INSPI :SIZE :ANGLE
FORWARD :SIZE
RIGHT :ANGLE
INSPI :SIZE (:ANGLE + 10)
END
```



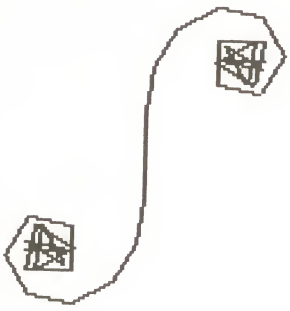
INSPI 20 10 INSPI 20 45 INSPI 20 33

Figure 8.13: Inspiral designs.

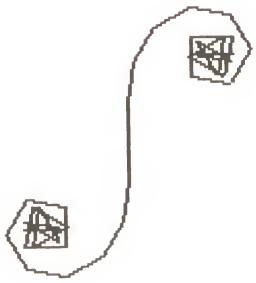
The amount of the increase can also have an effect on the overall shape. Adding another variable makes it easier to experiment with this.

```
TO INSPI2 :SIZE :ANGLE :INC
FORWARD :SIZE
RIGHT :ANGLE
INSPI2 :SIZE (:ANGLE + :INC) :INC
END
```

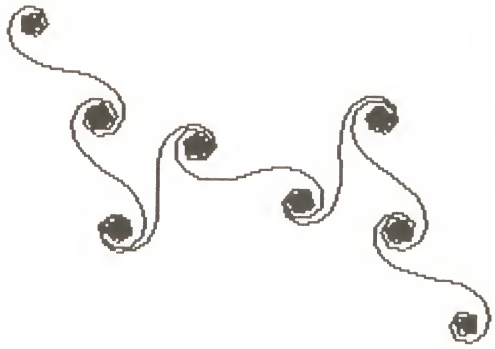
Changing the starting angle and the amount of increase both affect the overall design in surprising ways.



INSPI2 20 10 5

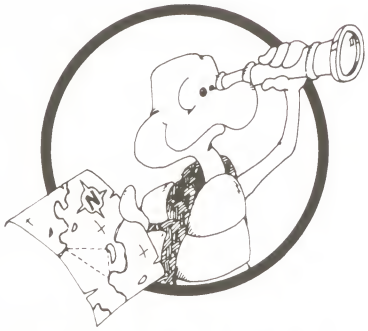


INSPI2 20 10 10



INSPI2 20 10 7

Figure 8.14: Varying the increase in an inspiral design.



- There are many interesting designs that can be made by changing the second and third inputs of INSPI2. The complete mathematical rule for INSPI designs is very complicated. Here are some interesting questions to investigate.
- Which combinations of inputs to INSPI2 make designs with only *two* clusters? Which ones produce *three* clusters? *Four* clusters? Etc.

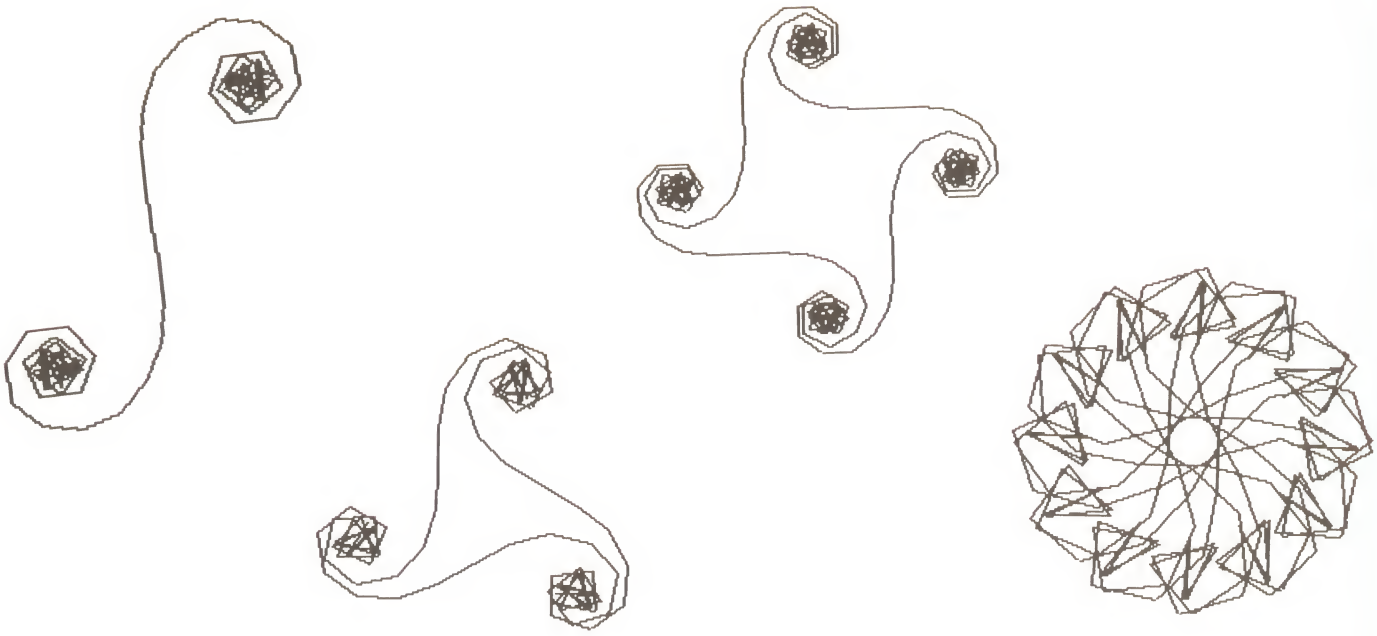


Figure 8.15: INSPI2 designs with two, three, four, and twelve clusters.

- Which combination make designs that never come back to where they started?

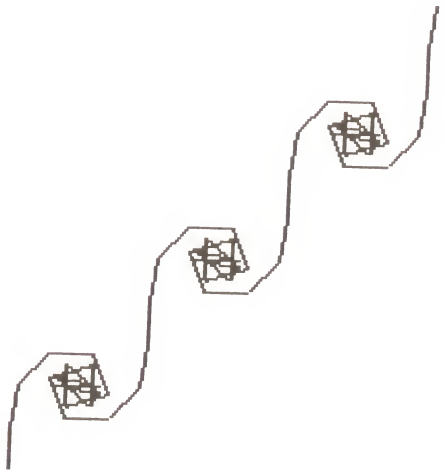


Figure 8.16: An INSPI2 design that doesn't come back to its starting point.

Write down in your journal any discoveries about inspiral designs and see how much you can find out about them.

Section 8.6. More POLY Relatives

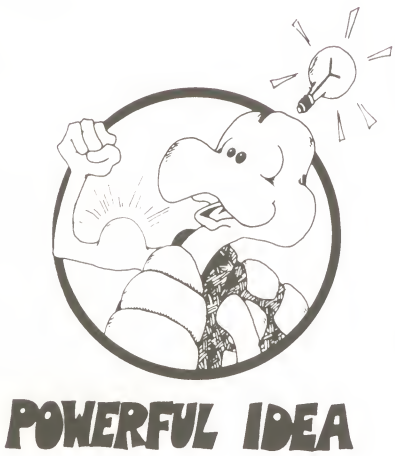
If you've enjoyed the explorations and designs in this chapter, you might also enjoy a book called *Turtle Geometry*, by Harold Abelson and Andrea diSessa. It has many more ideas for explorations with the turtle. I've borrowed a few ideas from that book to end this chapter. There are many possible variations for POLY, POLYSPI, and INSPI. I'll show you some of them and hope you invent lots more of your own.

First, let's look at 2POLY.

```
TO 2POLY :S1 :A1 :S2 :A2
FORWARD :S1 RIGHT :A1
FORWARD :S2 RIGHT :A2
2POLY :S1 :A1 :S2 :A2
END
```

2POLY just uses two different *poly steps* instead of one, and it needs *four* inputs. :S1 and :S2 are short for :SIZE1 and :SIZE2, and :A1 and :A2 for :ANGLE1 and :ANGLE2. It's nice to try positive and negative inputs for both sizes and angles. Put negative inputs in parentheses so the computer won't try to *subtract*. Try this one:

```
2POLY 50 45 (-25) 90
```



Telling the turtle to go FORWARD (−25) is the same as telling it to go BACK 25. Telling it to turn RIGHT (−30) is the same as telling it to turn LEFT 30. Try this out for yourself and see.

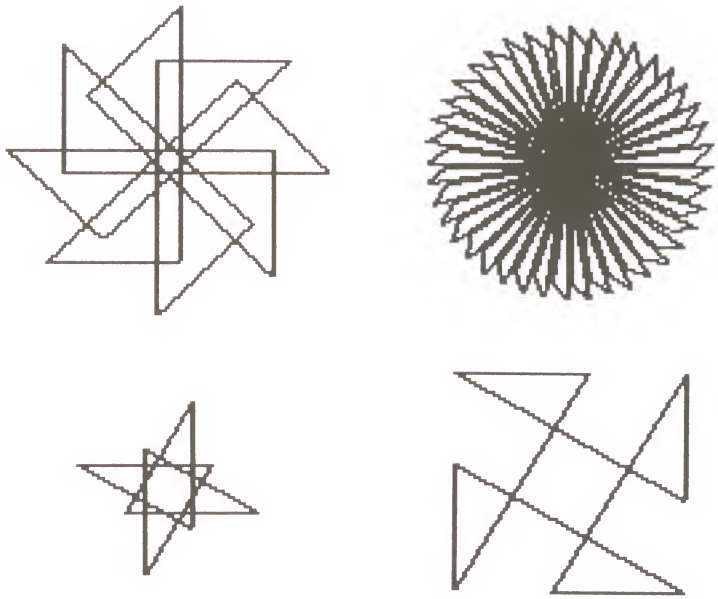


Figure 8.17: Designs made by 2POLY.

Another really simple variation for POLY is to *switch* the angle and size inputs every time.

```
TO SWITCHPOLY :SIZE :ANGLE
FORWARD :SIZE
RIGHT :ANGLE
IF HEADING = 0 [STOP]
SWITCHPOLY :ANGLE :SIZE
END
```

Notice that the two variable names are *reversed* in the last line. You can have a lot of fun with this one.

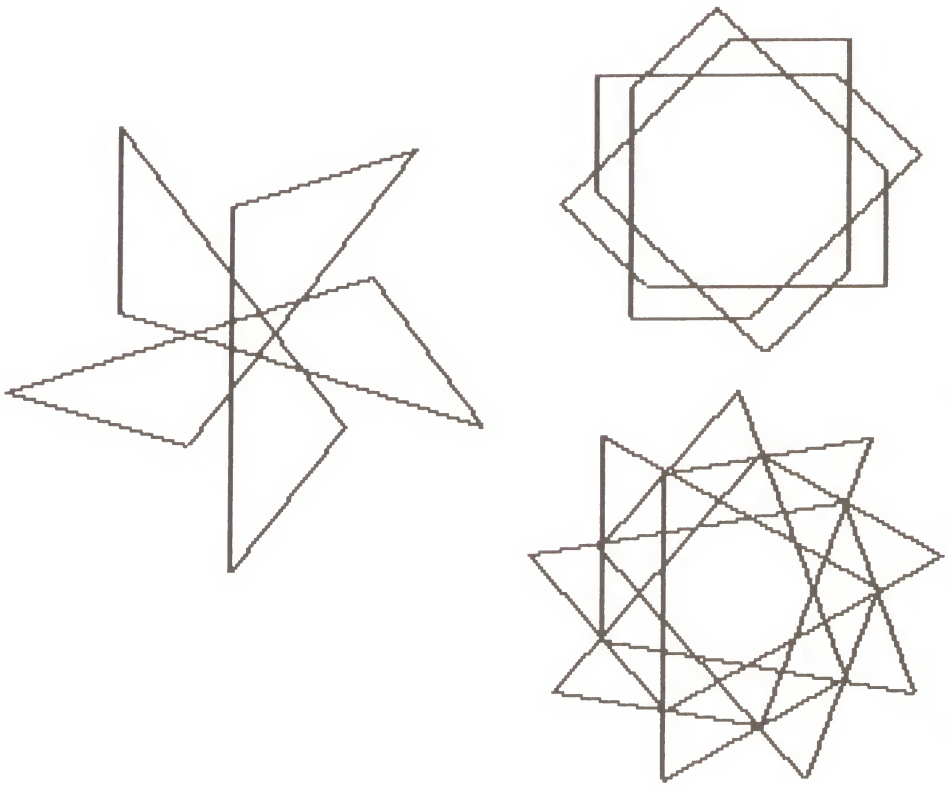


Figure 8.18: SWITCHPOLY designs.

Here's another variation. Instead of making a "poly step" consist of just forward and turn, make it draw a figure as well. First you need to teach the computer how to TRIANGLE :SIZE, of course.

```
TO POLYTRI :SIZE :ANGLE
TRIANGLE :SIZE
FORWARD :SIZE
RIGHT :ANGLE
IF HEADING = 0 [STOP]
POLYTRI :SIZE :ANGLE
END
```

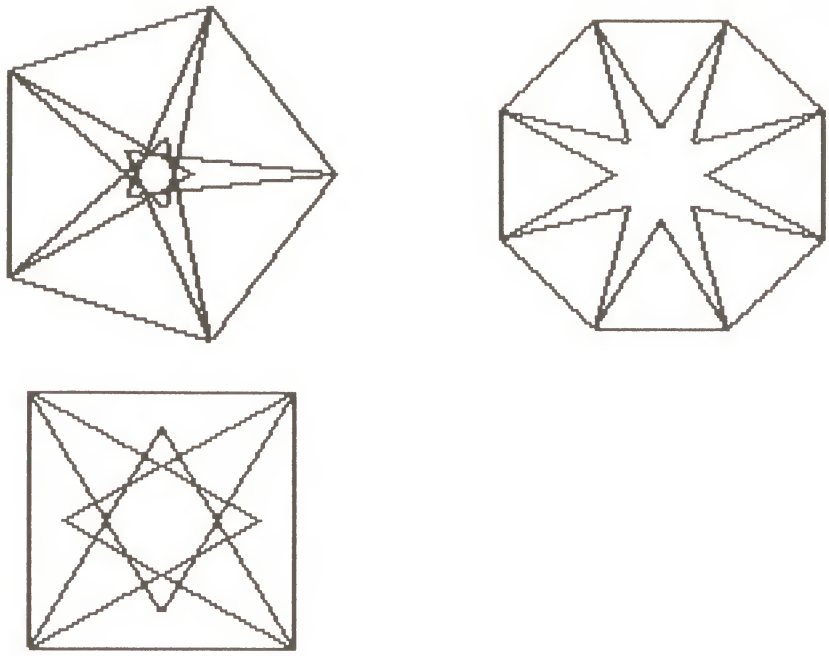
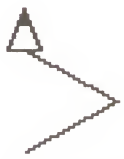


Figure 8.19: Designs made by POLYTRI.

Of course, you could use any shape you wanted in place of the triangle. Here's an interesting one called a "scissors."

```
TO SCISSORS :SIZE :ANGLE
RIGHT :ANGLE
FORWARD :SIZE
LEFT 2 * :ANGLE
FORWARD :SIZE
RIGHT :ANGLE
END
```



```
SCISSORS 40 60
```

Figure 8.20: A SCISSORS design.

When this is used in a POLY, I call it a POLYSCI.

```
TO POLYSCI :SIZE :ANGLE
SCISSORS :SIZE :ANGLE
RIGHT :ANGLE
POLYSCI :SIZE :ANGLE
END
```

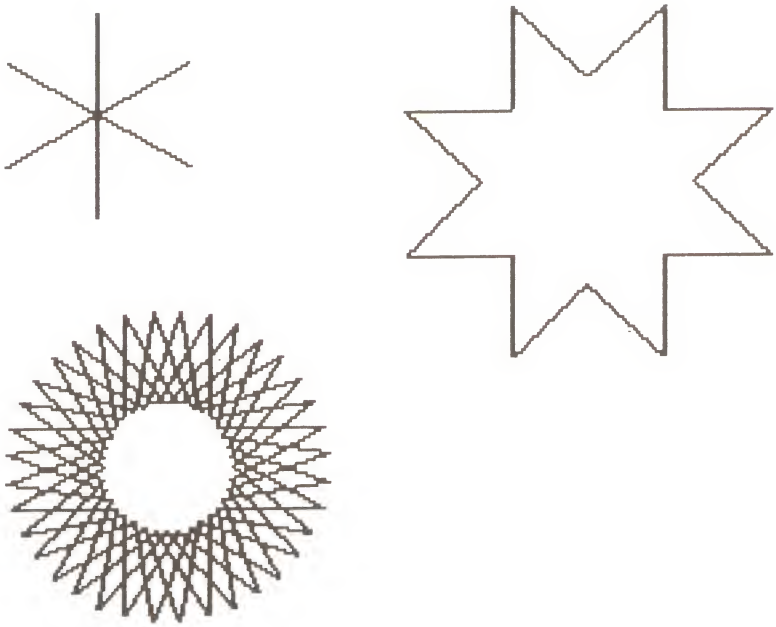


Figure 8.21: Designs made by POLYSCI.

These can be made even more interesting if the angle for the scissors is not the same as the angle for the "poly step."

```
TO POLYSCI2 :SIZE :A1 :A2
SCISSORS :SIZE :A1
RIGHT :A2
POLYSCI2 :SIZE :A1 :A2
END
```

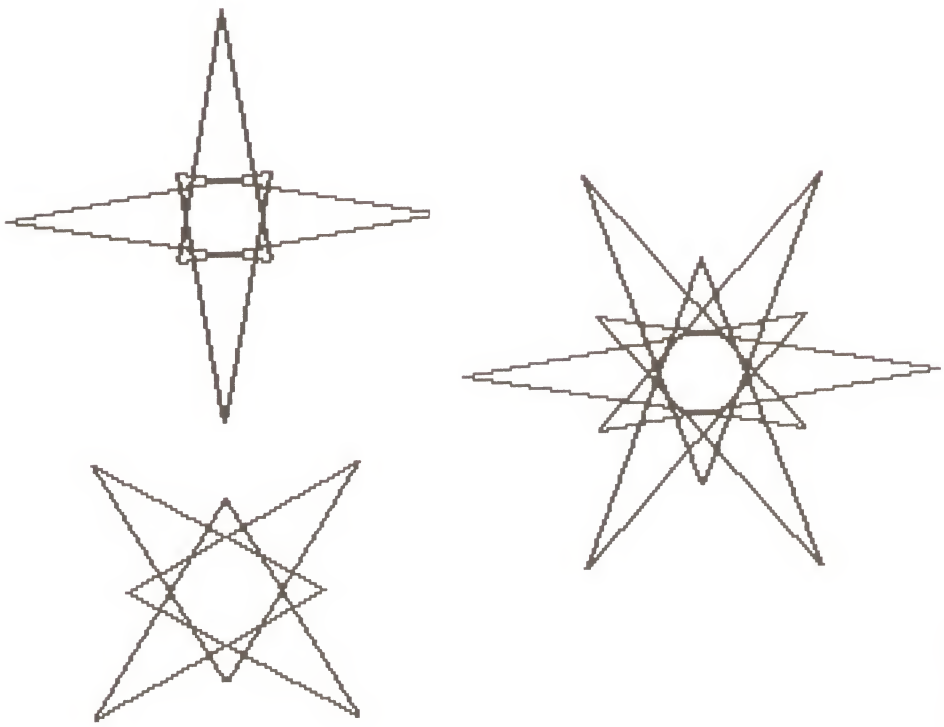
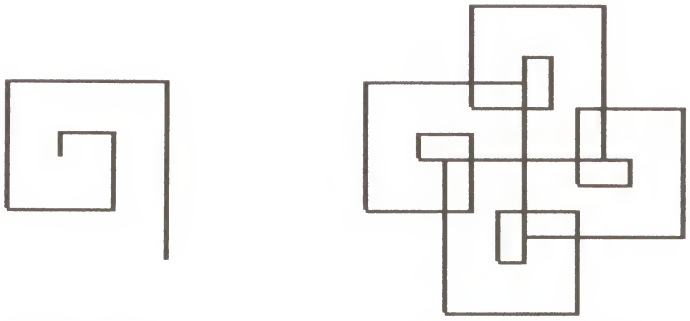


Figure 8.22: POLYSCI2 designs.

Any of these shapes can be made into polyspirals or inspirals by increasing the size or angle each time.

For one final example, let's look at a *spirolateral*, a geometric shape that is based on the idea of a polyspiral. The difference is that the length increases for a fixed number of times. Then the whole shape is repeated until the figure is complete. For example, Figure 8.23 shows a 90 degree polyspiral repeated 7 times. Then that whole shape is repeated.



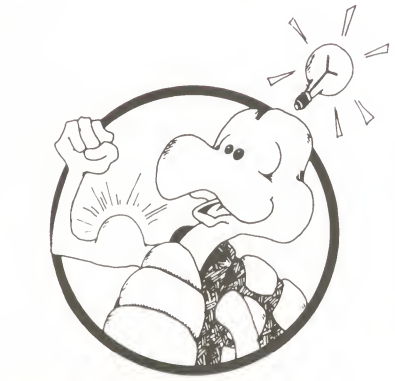
SPIRO 10 90 7

SPIROLATERAL 90 7

Figure 8.23: Spirolateral designs.

Here are the procedures:

```
TO SPIRO :SIZE :ANGLE :NUMBER
IF :NUMBER = 0 [STOP]
FORWARD :SIZE
RIGHT :ANGLE
SPIRO (:SIZE + 10) :ANGLE (:NUMBER - 1)
END
TO SPIROLATERAL :ANGLE :NUMBER
SPIRO 10 :ANGLE :NUMBER
SPIROLATERAL :ANGLE :NUMBER
END
```



SPIRO is a procedure that *counts*. It is almost the same as POLYSPI (Section 8.4) except that it has one more variable, :NUMBER, that tells it exactly how many poly steps to take. Every time SPIRO is called, :NUMBER is decreased by one. When it counts down to zero, the procedure stops. All that SPIROLATERAL does is keep calling the procedure SPIRO with a fixed size input of 10.

We can vary the *size* of a spirolateral if we make the *increase* in size a variable in SPIRO and let the *starting size* be a variable in SPIROLATERAL. SPIROLATERAL and SPIROLATERAL2 draw the same shapes, but SPIROLATERAL2 can have different sizes.

```
TO SPIRO2 :SIZE :ANGLE :INC :NUMBER
IF :NUMBER = 0 [STOP]
FORWARD :SIZE
RIGHT :ANGLE
SPIRO2 (:SIZE + :INC) :ANGLE :INC (:NUMBER - 1)
END
TO SPRIOLATERAL2 :SIZE :ANGLE :NUMBER
SPIRO2 :SIZE :ANGLE :SIZE :NUMBER
SPIROLATERAL2 :SIZE :ANGLE :NUMBER
END
```

When SPIROLATERAL2 calls SPIRO2, it uses the *same value* for *both* the :SIZE and :INC variables of SPIRO2. This little trick keeps the shapes proportional to each other.

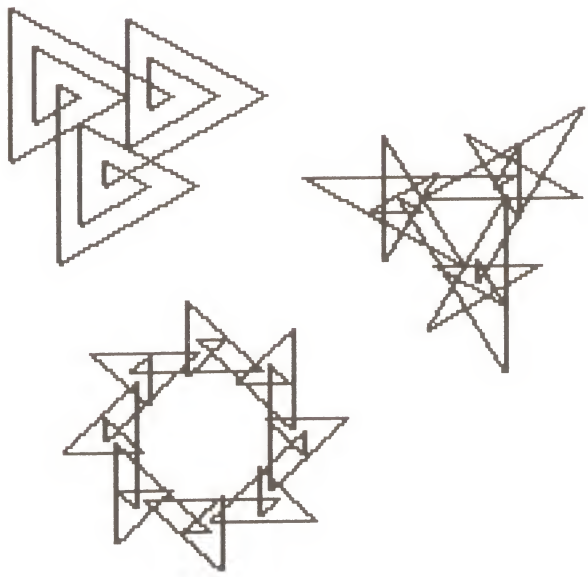


Figure 8.24: More spirolateral designs.

The most interesting mathematical question about spirolaterals have to do with the fact that some spirolaterals make *closed* shapes that keep repeating themselves while others go off wrapping around the screen forever.

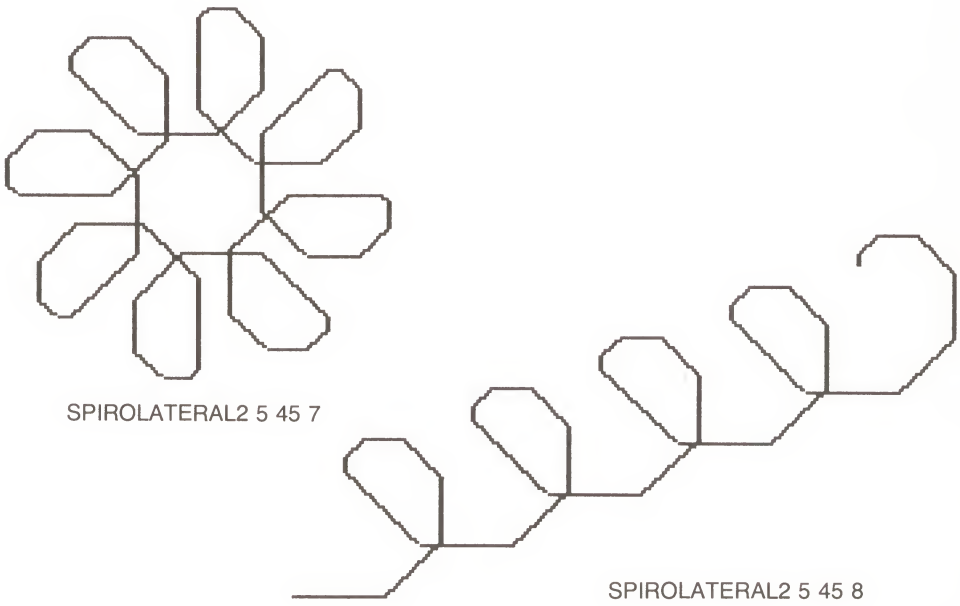


Figure 8.25: Closed and open spirolaterals.

Which combinations of angles and numbers will make closed shapes, and which ones will make shapes that don't return to their starting points? The answer to this question is very similar to the rule connecting the number of sides to the angle of a POLY (Section 8.1).

HELPER'S HINT

The emphasis in this chapter is on making designs and exploring questions about them. Someone might want these shapes used more explicitly to *teach* mathematics. I'm quite sure that someday all these lovely shapes will be part of the mathematics curriculum for everyone. I can't help feeling that some of the loveliness will be lost when that happens. Abelson and diSessa's *Turtle Geometry* is a college-level textbook that goes a long way towards capturing both the mathematics *and* the fun of explorations with the turtle. I recommend it both for the spirit in which the learning of mathematics has been combined with the aesthetics of turtle explorations and for the hundreds of fascinating turtle projects described in it.

If you are interested in enhancing the mathematical learning of someone using *this* book, I have a few suggestions. First, emphasize the finding of patterns made by families of shapes, patterns that relate numbers to shapes. The angle numbers are most critical in determining patterns.

People often get confused dealing with a lot of different variables. Isolating the effect of one variable at a time is another important idea that can be learned through these activities. I have discovered, however, that this idea is not obvious or even interesting to many people at first. Some people have to look at many facets of a situation before they are willing to settle down and isolate one for exploration. Too many teachers force their students into a mold of "rational exploration" before they are ready to focus on one aspect of a situation. It is my experience that when a student is ready for a structured exploration, he or she can appreciate the value (even the beauty) of isolating variables. On the other hand, students forced into a rational mode while they are still exploring the whole of a phenomenon can become convinced that mathematics and science are boring. In this way I believe that we discourage many students who might otherwise become creative scientists or mathematicians.

The moral is that it's a good idea for a teacher to have a large "bag of tricks" containing many suggestions to help students rationalize their thinking processes *and* to help them be more creative in their explorations. Above all, it is imporant to be sensitive to just which suggestions will most further a student's real learning goals at a particular moment. This is a subtle skill that makes teaching more of an art than a science. In preparing this chapter I tried to plant seeds of both rational exploration of mathematical patterns and creative explorations of artistic ones. Let each learner, each teacher, take from it what is best for him or her at the moment.

End of sermon!

CHAPTER 9

Command	Short Form	Examples With Inputs
/		PRINT 360 / 3, RIGHT 360 / 3
WORD		PRINT WORD "HEL "LO
		PRINT (WORD "A "B "C)
SENTENCE	SE	PRINT SENTENCE [HELLO] [THERE]
		PRINT SENTENCE "HELLO [THERE]
		PRINT SENTENCE "HI "FRIEND
		PRINT (SE [HELLO] [MY] [FRIEND])
FIRST		PRINT FIRST "HELLO
		PRINT FIRST [HELLO THERE FRIEND]
BUTFIRST	BF	PRINT BUTFIRST "HELLO
		PRINT BF [HELLO THERE FRIEND]
LAST		PRINT LAST "HELLO
		PRINT LAST [HELLO MY FRIEND]
BUTLAST	BL	PRINT BL "HELLO
		PRINT BUTLAST [HELLO MY FRIEND]
READLIST	RL	MAKE "ANSWER READLIST
CLEARTEXT		
TYPE		TYPE [GUESS A NUMBER]
RANDOM		PRINT RANDOM 20
TEST		TEST :ANSWER = 7
IFTRUE	IFT	IFTRUE [PRINT [HOORAY!]]
IFFALSE	IFF	IFFALSE [PRINT [SORRY]]
AND		IF AND :NUM1 = 1 :NUM2 = 0 [PRINT [OKAY]]
OR		IF OR :ANS = [TWO] :ANS = [ ] [PRINT [RIGHT!]]

LWAL Procedures Disk files used: "READNUMBER, "GUESSNUMBER, "MATHQUIZ

New tool procedures used:

Tool Procedures	Examples
READNUMBER	MAKE "ANSWER READNUMBER

9 Conversations with the Computer: Activities with Numbers, Words, and Lists

This chapter explains more about how Logo keeps track of information and tells you how to carry on conversations with the computer. It shows how Logo uses three different kinds of information—numbers, words, and lists. The ideas in this chapter will also help you understand something about how computers do what is called *data processing*. Data processing involves making the computer take in, store, change, and print information. Often it involves some kind of *interaction* between the computer and the person who is using it. This chapter is an introduction to interactive Logo procedures.

Section 9.1. Numbers, Words, and Lists

Information used by a computer is called *data*. Logo can handle three different kinds of data: numbers, words, and lists.

Numbers are just what you think they are—good old everyday numbers, large and small, positive and negative. Logo can add, subtract, multiply, and divide. It can also compare two numbers to see if they are equal or to see which one is larger or smaller. Try these examples:

```
PRINT 4 + 5
PRINT 36 - 6
PRINT 36 / 7
PRINT (3 + 5) * 7
```

The parentheses around 3 + 5 tells the computer to add 3 and 5 *first* and then multiply their sum by 7. Compare what happens when you type the line *without* parentheses.

```
PRINT 3 + 5 * 7
```

In this case, Logo multiplies 5 and 7 *first* and then adds 3. When there are no parentheses, Logo will always multiply or divide before it adds or subtracts.

These examples show how Logo compares numbers:

```
PRINT 5 = 7
PRINT 5 = (7 - 2)
PRINT 5 > 7
PRINT 5 < 7
```

The expression 5 > 7 is a *question*: "Is 5 *greater than* 7?" (Naturally, Logo should answer **FALSE**.) The expression 5 < 7 asks Logo, "Is 5 *less than* 7?" (Logo should answer **TRUE**.) Logo does not need parentheses in the command PRINT 5 = (7 − 2) since it always does arithmetic before comparing, but I often use parentheses to make some lines clearer for a *person* who might be reading them.

Words in Logo are a lot like normal everyday English words, even though they may not always make sense to you or me. Almost any combination of symbols can be part of a Logo word. A " symbol *before* a word shows that whatever follows is a Logo word. A " symbol is *never* used *after* a word in Logo. This is one way that Logo is different from ordinary English and from many other computer languages. Almost any combination of symbols can be part of a Logo word. Logo words can be made from almost any keyboard symbols. Try these examples:

```
PRINT "HELLO
PRINT "ABCXYZ
PRINT "R2D2
PRINT "AB.$ −) * *
PRINT "3 + 4
```

One very important use of words in Logo is as *names* for variables or files. You have been using Logo words for file names ever since Chapter 4.

Numbers are also Logo words. They can be used with or without " symbols.

```
PRINT "25
PRINT "25 + "25
```

A " symbol with nothing after it is called an *empty word*. You can use an empty word to make Logo print a blank line.

```
PRINT "
```

A Logo word ends when you type a *space*. Nothing after the space is part of the word. Try this:

```
PRINT "HELLO THERE
```

The computer prints the word "HELLO, but since there is no quote in front of THERE, Logo thinks that THERE is meant to be a procedure. Try the same command with " in front of THERE:

```
PRINT "HELLO "THERE
```

This time Logo knows that "THERE is a word, but it doesn't know what to do with it.

Lists are Logo's way of combining words into groups. A list is contained within square brackets, **[** and **]**, and can include words, numbers, and even other lists. Try these examples:

```
PRINT [HELLO THERE]
PRINT [1 2 3 4 5 6]
PRINT [MY NAME IS DAN]
PRINT [THIS IS A LIST: [THIS IS A LIST: ]]
PRINT [[A B] [C D] [E F]]
```

A list with nothing in it, **[ ]**, is called an *empty list*. Like an empty word, an empty list can be used to print a blank line.

```
PRINT [ ]
```

Section 9.2. Commands for Using Words and Lists

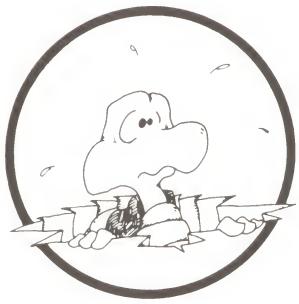
Logo has a number of commands that can be used to combine words into bigger words, to combine words into lists, or to combine words and lists into bigger lists. It also has commands that take words and lists apart.

WORD is a Logo command that takes two words as inputs and *outputs* the combination as one bigger word.

```
PRINT WORD "BIG "WORD
PRINT WORD "WO "RD
```

WORD can have more than two inputs if the command and its inputs are placed inside **( )**.

```
PRINT (WORD "BIG "GER "WORD )
```



PITFALL

Leave a space after the last word in the group or Logo will think that the last **)** is part of the last word. Watch what happens when you type

```
PRINT (WORD "BIG "GER "WORD)
```

Another common bug is to type the **(** *after* the WORD command. This will also confuse Logo and make it complain.

```
PRINT WORD ("BIG "GER "WORD )
```

SENTENCE is used to combine words and lists into one bigger *list*. Ordinarily it has two inputs. Each one can be either a word or a list.

```
PRINT SENTENCE "A [WORD PLUS A LIST]
PRINT SENTENCE [A LIST PLUS A] "WORD
PRINT SENTENCE "TWO "WORDS
```

SE is the short form of SENTENCE.

```
PRINT SE [TWO LISTS] [MAKE A LIST, TOO]
```

With parentheses, **(** and **)**, around the command and its inputs, SENTENCE can have more than two inputs.

```
PRINT (SENTENCE "THIS [WILL BECOME] [ONE LIST] "TOO )
```

Logo commands FIRST, LAST, BUTFIRST (or BF) and BUTLAST (or BL) are used to take apart words and lists. See what they do with these examples, then make up some more of your own.

```
PRINT FIRST "HELLO
PRINT LAST "HELLO
PRINT BUTFIRST "HELLO
PRINT BUTLAST "HELLO
```

FIRST and LAST output the first or last character of a word. BUTFIRST and BUTLAST output *everything but* the first or last character of a word.

The same commands can be used for lists.

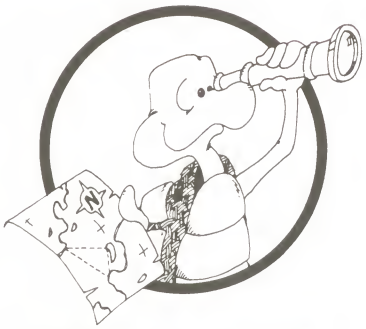
```
PRINT FIRST [HELLO MY FRIEND]
PRINT LAST [HELLO MY FRIEND]
PRINT BUTFIRST [HELLO MY FRIEND]
PRINT BUTLAST [HELLO MY FRIEND]
```

FIRST and LAST output the first and last items of a list (usually a word). BUTFIRST and BUTLAST output a list with everything but the first or last item of the input list. FIRST, LAST, BUTFIRST, and BUTLAST can be combined in different ways. See if you can predict what these commands will do.

```
PRINT FIRST BUTFIRST [HELLO MY FRIEND]
PRINT BUTFIRST FIRST [HELLO MY FRIEND]
```

BUTFIRST of the list [HELLO MY FRIEND] is the *list* [MY FRIEND]. FIRST of that list is the *word* "MY. Therefore, FIRST BUTFIRST [HELLO MY FRIEND] outputs the word "MY.

FIRST of the list [HELLO MY FRIEND] is the *word* "HELLO. BUTFIRST of "HELLO is the *word* "ELLO. Therefore, BUTFIRST FIRST [HELLO MY FRIEND] outputs the word "ELLO.



EXPLORATION

Experiment with FIRST, LAST, BUTFIRST, and BUTLAST until you have a good idea of what they will do. The combination FIRST BUTFIRST [HELLO MY FRIEND] outputs the *second* word of that list. Can you find a combination that will always output the *third* element of a list? The *fourth*? How about the second and third element from the *end* of a list?

READLIST, abbreviation RL, is a command that waits for the *user* to type a line from the keyboard and outputs that line as a list. Here's a funny little procedure that uses SENTENCE and READLIST:

```
TO TALK
PRINT [PLEASE TYPE SOMETHING FOR ME TO SAY]
PRINT SENTENCE [YOU JUST MADE ME SAY] READLIST
END
```

When Logo carries out the READLIST command, it waits for you to type something. Then it prints a sentence that includes what you just typed. Try TALK a few times and see what happens.

Here's another variation:

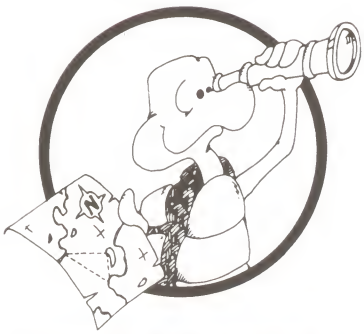
```
TO BACKTALK
PRINT [PLEASE TYPE SOMETHING FOR ME TO SAY]
PRINT SENTENCE [BUT I HATE TO SAY] READLIST
BACKTALK
END
```

Try BACKTALK. You'll need to press **CTRL-G** to stop it.

And here's another one:

```
TO AGREE
PRINT [TELL ME SOMETHING YOU LIKE]
PRINT (SENTENCE [I LIKE] READLIST [TOO])
PRINT [TELL ME SOMETHING YOU HATE]
PRINT (SENTENCE [I HATE] READLIST [EVEN MORE THAN YOU DO!])
AGREE
END
```

Notice the parentheses around the SENTENCE commands. They are needed because SENTENCE has *three* inputs.



EXPLORATION

Try to make up some "talk" programs of your own. Can you make one that *disagrees* with everything you type?

HELPER'S HINT

The second line of the TALK procedure is an interesting one to study (see Figure 9.1). It includes three Logo commands and two lists of information in one command line:

PRINT SENTENCE [YOU JUST MADE ME SAY] READLIST

PRINT needs an input in order to know what to print. It gets that input from the output of the Logo command SENTENCE.

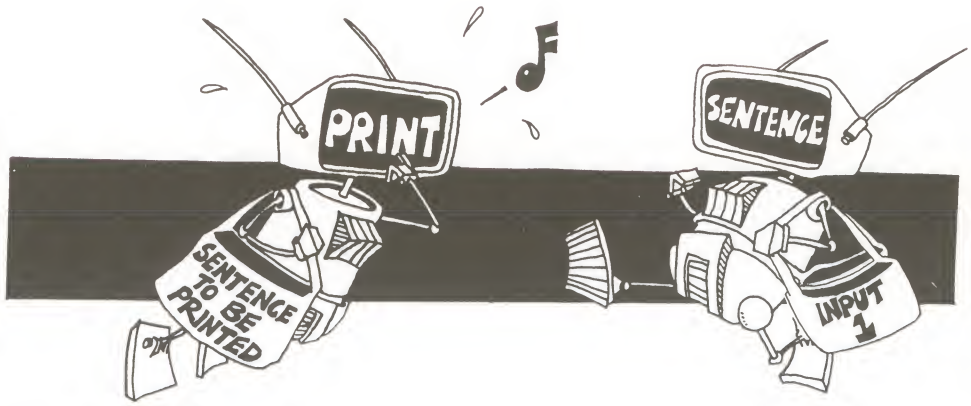


Figure 9.1a

SENTENCE needs *two* inputs. They could be two Logo words, two lists, or a word and a list. In this case, PRINT give SENTENCE its first input, the list [YOU JUST MADE ME SAY], and tells SENTENCE to call on another Logo command, READLIST, for its second input.

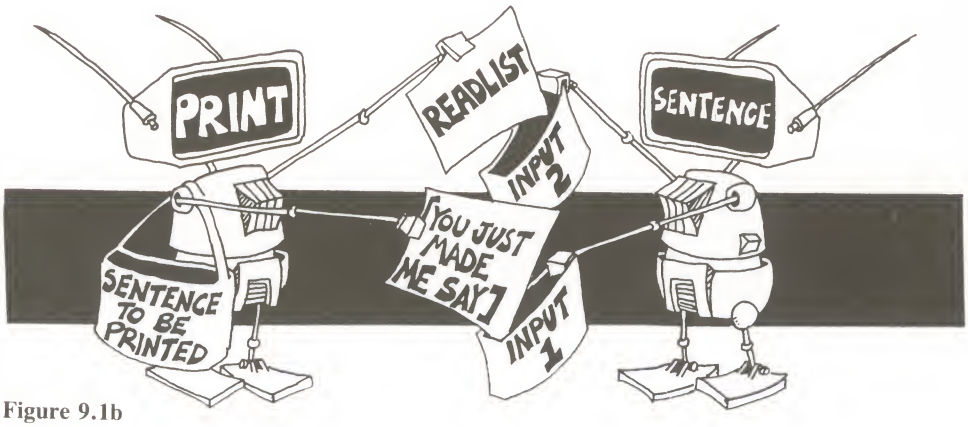


Figure 9.1b

SENTENCE calls READLIST to get its second input.

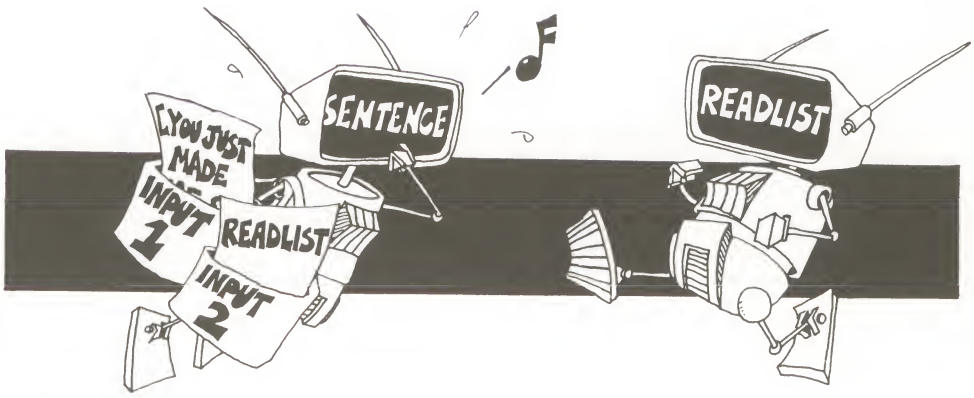


Figure 9.1c

READLIST gets its input from whatever the user types at the keyboard, and then *outputs* that information as a list to the procedure that called it. In this case, suppose you type HELLO.

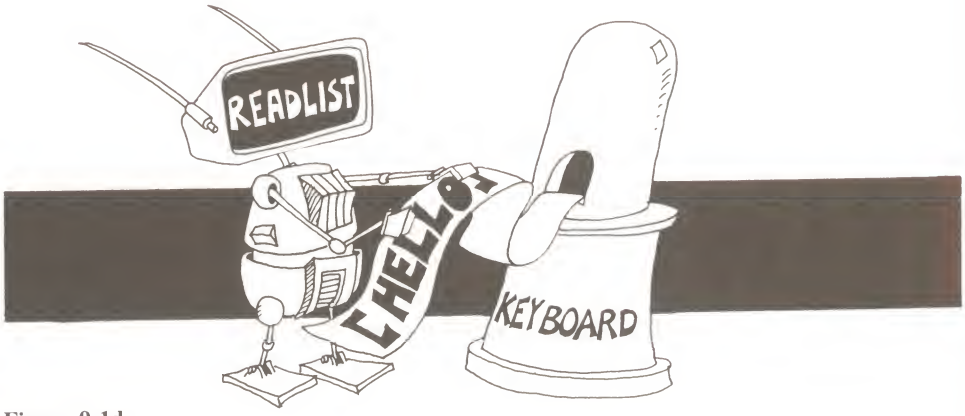


Figure 9.1d

READLIST outputs [HELLO] back to SENTENCE.

Figure 9.1e

SENTENCE combines [HELLO] with its first input, and outputs [YOU JUST MADE ME SAY HELLO] back to PRINT.

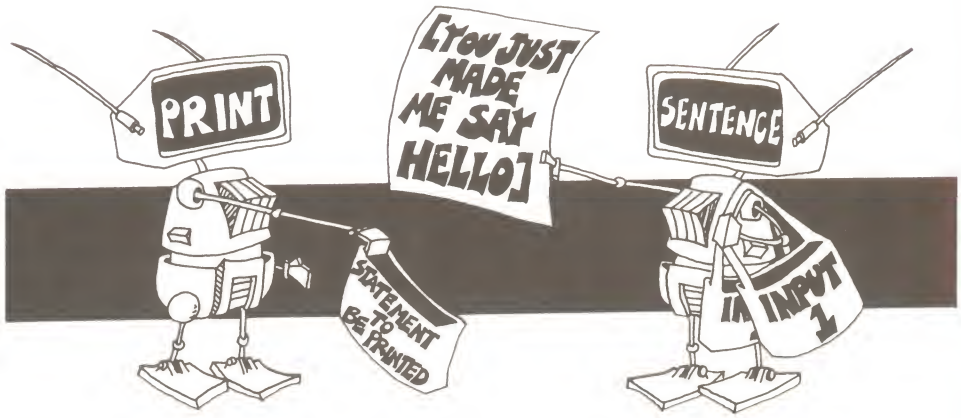


Figure 9.1f

PRINT prints the list YOU JUST MADE ME SAY HELLO (without the brackets) on the TV screen.

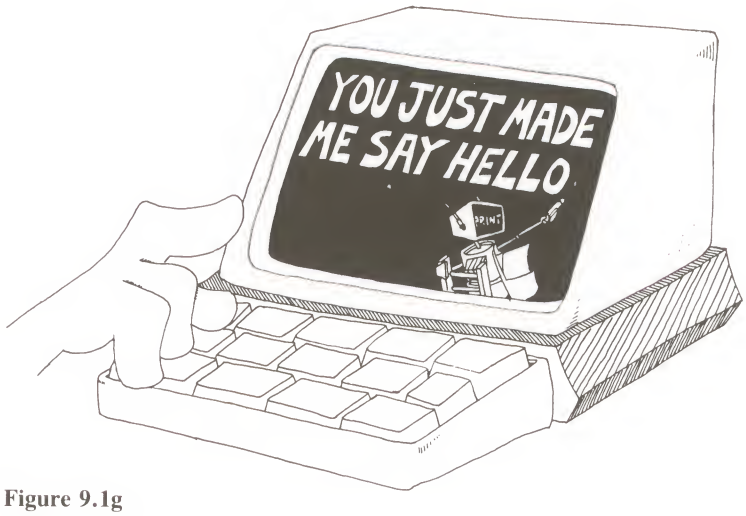


Figure 9.1g

HELPER'S HINT

I have deliberately introduced the term "output" in this chapter. In Logo jargon, "to output" means to send a message with information. The information must be a number, word, or list. The message is "sent" to whichever procedure, or command, called the command that outputs. In most of the examples, the message was sent to PRINT, so that it was printed on the screen.

A command or procedure that outputs information is sometimes called an "operation". Arithmetic commands are operations because they output numerical results. Comparison commands like =, >, or < are operations because they output a word, "TRUE or "FALSE. FIRST, LAST, BUTFIRST, and BUTLAST are operations because they output words or lists.

Section 9.3. Numbers, Words, and Lists as Variables

In Chapters 7 and 8, numbers were used as variables to control or change the size or shape of a turtle procedure. Words and lists can also be given names, stored in the computer's memory, and used as variables. One way to do this is by using them as inputs to a procedure.

```
TO SPEAK :MESSAGE
PRINT [THE MESSAGE I AM GOING TO PRINT IS]
PRINT :MESSAGE
END
```

The input to SPEAK can be a word, a list, or even just a number.

```
SPEAK "HELLO
SPEAK [NOW IS THE TIME]
SPEAK 103
```

Another way to make a number, word, or list into a variable is to use the Logo command MAKE. MAKE needs two inputs. The first input is the *name* of the variable, which must be a Logo word. The second is the *value* of the variable, which can be a number, word, or list.

Watch what happens when you use MAKE several times in a row, to create a new Logo variable called "MESSAGE (see Figure 9.2). You won't be able to see the effect of MAKE each time until you tell Logo to PRINT :MESSAGE. Suppose you type:

```
MAKE "MESSAGE [THIS IS GETTING SILLY]
```

MAKE creates a new variable with the name "MESSAGE and the value [THIS IS GETTING SILLY].

Logo stores this value in the computer's working memory. To see what the new value is, type PRINT :MESSAGE

Figure 9.2a

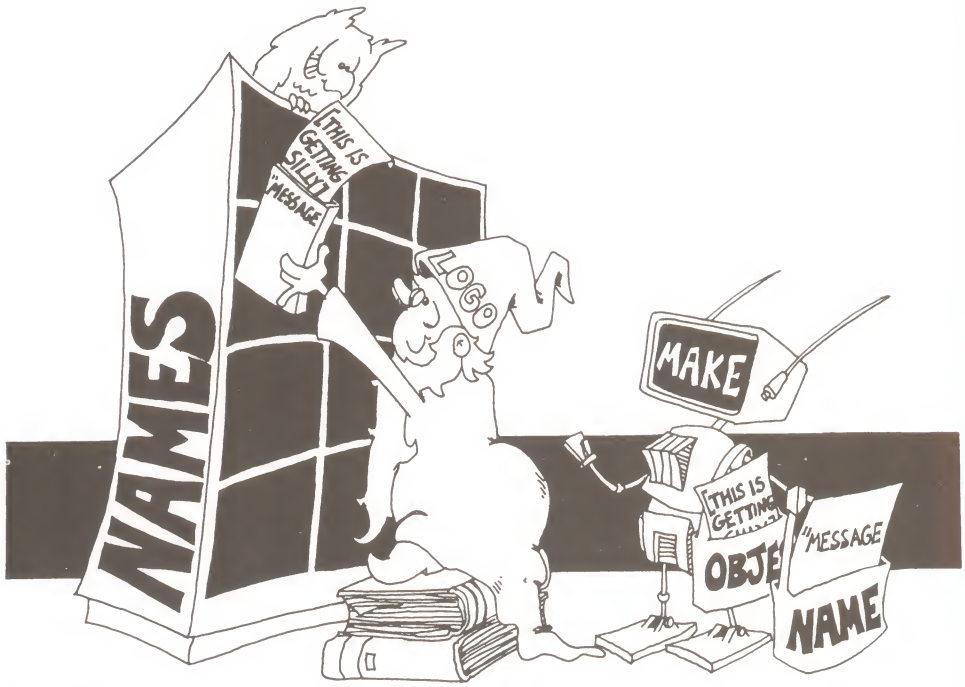


Figure 9.2b

When you change the value of the variable, MAKE creates a new variable with the same name, "MESSAGE, but a new value, "HELLO.

```
MAKE "MESSAGE "HELLO
```

Logo stores *this* value in the computer's working memory, and throws away the old one. To see what the new value is now, type: PRINT :MESSAGE

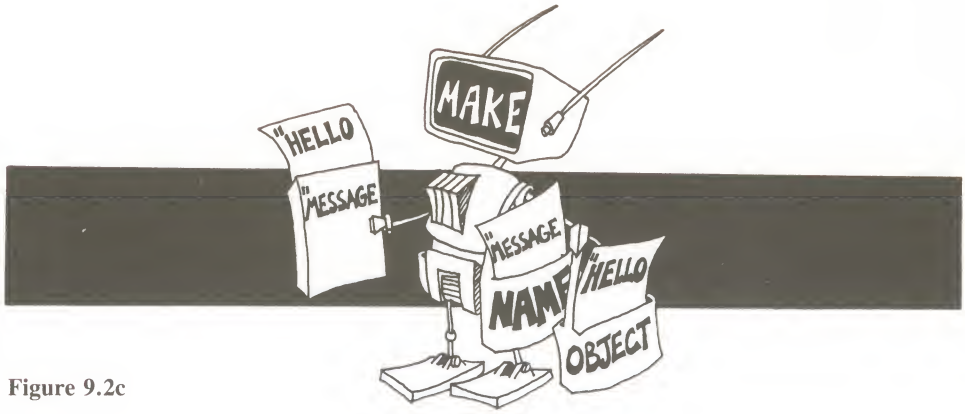


Figure 9.2c

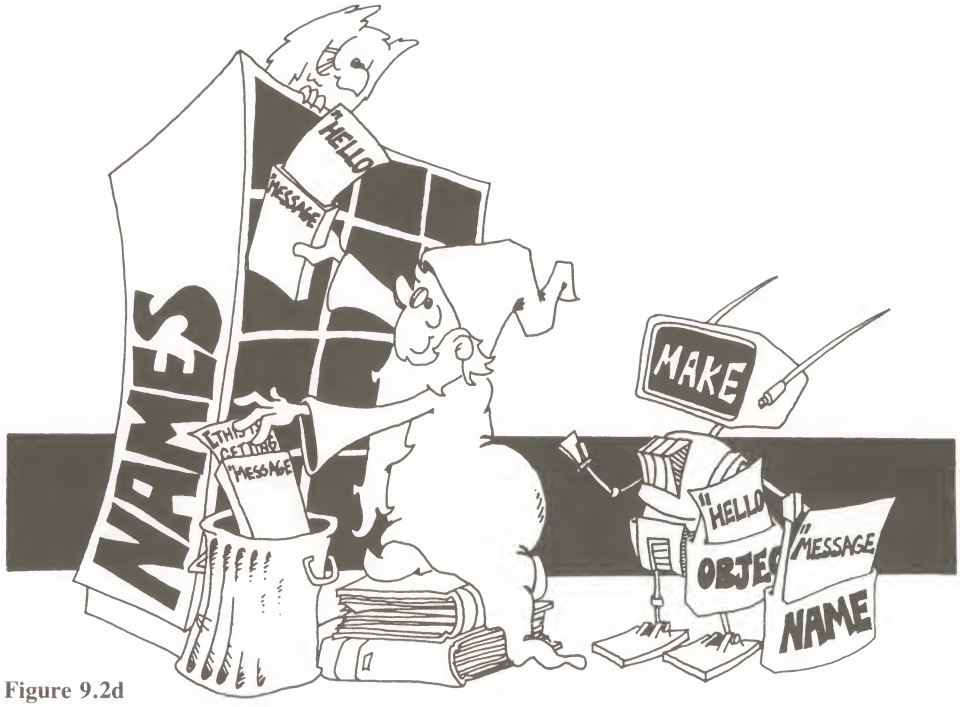


Figure 9.2d

When you use MAKE again, the process is repeated.

```
MAKE "MESSAGE 1000
```

To see the latest value of the variable with the name "MESSAGE, type

```
PRINT :MESSAGE
```

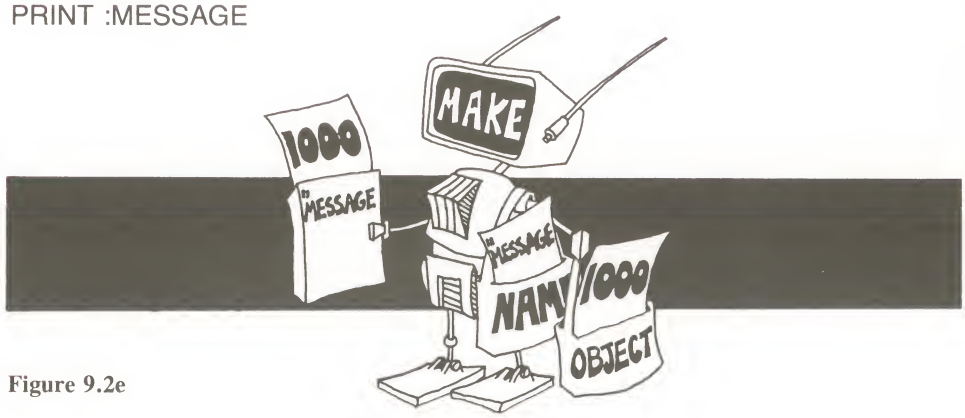


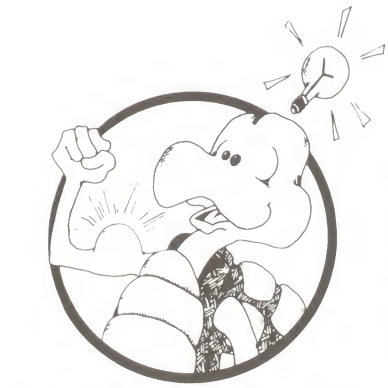
Figure 9.2e

Figure 9.2f

Now try this. Before you do it, try to predict what will happen.

```
PRINT "MESSAGE
PRINT :MESSAGE
```

PRINT "MESSAGE tells the computer to print the Logo word "MESSAGE. PRINT :MESSAGE tells the computer to print the value of the variable *named* "MESSAGE. When you talk about these two different things it's a good idea to say *quotes message* when you refer to "MESSAGE and *dots message* when you refer to :MESSAGE. This way of talking may seem a little silly at first, but it will help you (and people you are talking to) understand what you are talking and thinking about.



The idea of giving a name to a piece of information and then using that name to do things with the information is one of the most important ideas in all mathematics and computer programming. It is very important to understand the difference between the name and the thing it stands for. This is why the expressions *quotes* and *dots* are so important. They are an easy way of saying the name, or the thing it stands for, and being absolutely clear which is which.

MAKE can also be used to *change* the value of a variable. Try this:

```
MAKE "NUMBER 5
PRINT :NUMBER
MAKE "NUMBER :NUMBER + 5
PRINT :NUMBER
```

The command MAKE "NUMBER :NUMBER + 5 tells the computer to add 5 to the value :NUMBER and give this *new* value the name "NUMBER.

MAKE can also be used to change the value of a list or word variable. Type these commands:

```
MAKE "MESSAGE [HELLO THERE]
PRINT :MESSAGE
MAKE "MESSAGE SENTENCE :MESSAGE "FRIEND
PRINT :MESSAGE
```

HELPER'S HINT

Let's examine the workings of the command

MAKE "MESSAGE SENTENCE :MESSAGE "FRIEND

First of all, remember that if you give the command MAKE "MESSAGE [HELLO THERE], MAKE will create a variable named "MESSAGE with a value, [HELLO THERE] and Logo will store that variable in the computer's working memory.

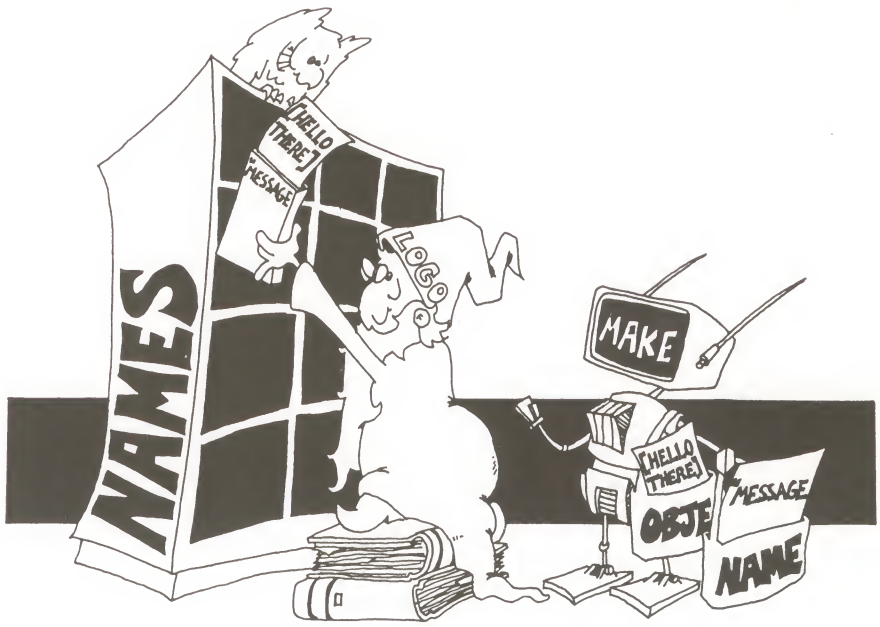


Figure 9.3a

If you tell Logo to PRINT :MESSAGE, Logo finds the value in the slot marked "MESSAGE, and passes that value, [HELLO THERE], to PRINT.

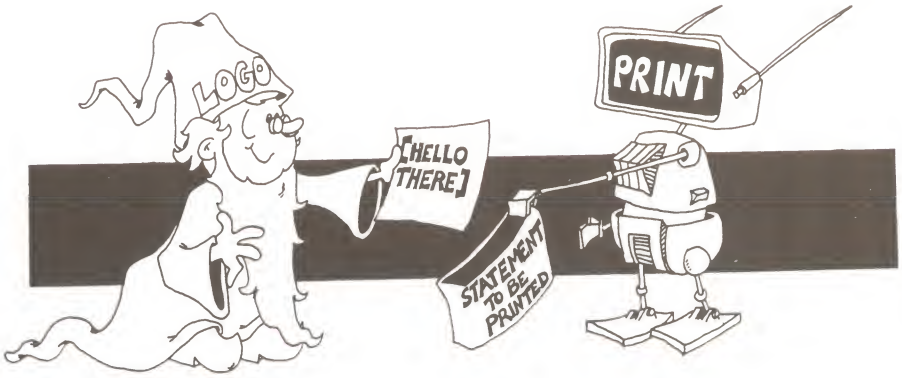


Figure 9.3b

Now look at the more complex command,

MAKE "MESSAGE SENTENCE :MESSAGE "FRIEND

MAKE still needs two inputs, a name and a value. The first input, the name is "MESSAGE. For its second input, Logo tells MAKE to call SENTENCE, and to give SENTENCE the inputs, :MESSAGE and "FRIEND.

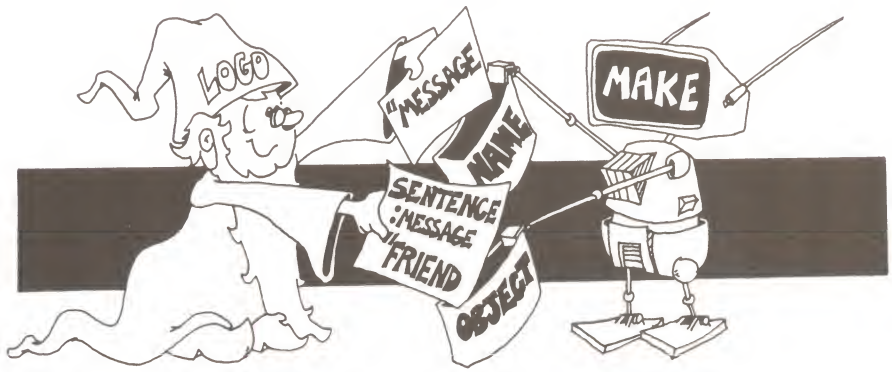


Figure 9.3c

MAKE calls SENTENCE and gives it two inputs. The first is the *present value* stored in the slot called "MESSAGE, which is still the list, [HELLO THERE]. The second is the word "FRIEND.

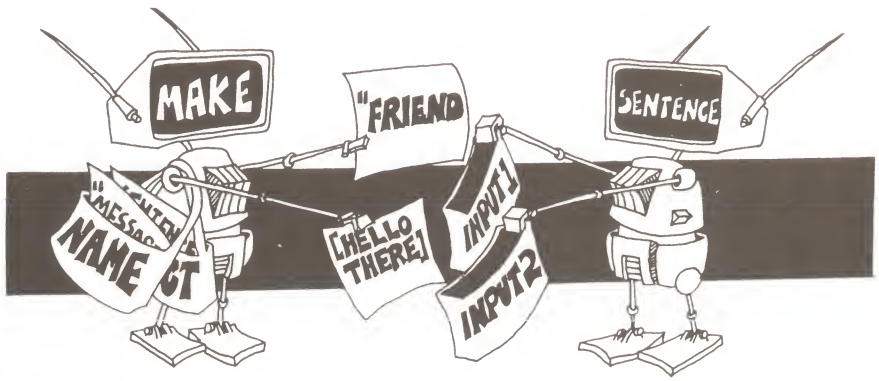


Figure 9.3d

SENTENCE combines its two inputs and outputs **[HELLO THERE FRIEND]** back to MAKE, which uses that list as *its* second input.

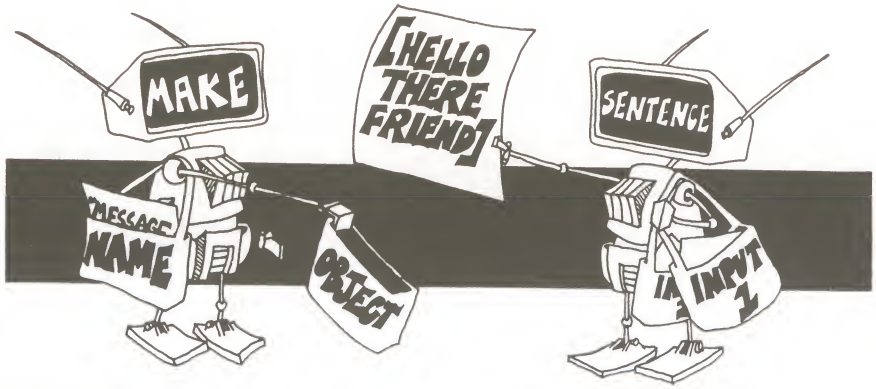


Figure 9.3e

Now MAKE creates a new variable with the name "MESSAGE and the value **[HELLO THERE FRIEND]**. Logo stores this value in the computer's working memory, replacing the old value, which is eliminated.

Figure 9.3f

Now if you tell the computer to PRINT :MESSAGE, Logo gives PRINT the new value.

Figure 9.3g

Here's an example in a procedure:

```
TO GROW :MESSAGE
PRINT [TYPE SOMETHING NEW]
MAKE "NEWPART READLIST
MAKE "MESSAGE SENTENCE :MESSAGE :NEWPART
PRINT [THE MESSAGE IS NOW]
PRINT :MESSAGE
GROW :MESSAGE
END
```

See what happens when you type GROW "HELLO and follow the computer's directions.

```
GROW "HELLO
```

Here's how GROW works—the first line of GROW just prints **TYPE SOMETHING NEW**. The second line tells the computer to give the name "NEWPART to whatever the user types.

The third line creates a sentence of :MESSAGE and :NEWPART and gives the whole thing the name "MESSAGE.

The fourth line prints **THE MESSAGE IS NOW**. The fifth line prints the new value, :MESSAGE. The sixth line calls another GROW procedure, using the *new* message as input, and the whole process starts again.

Section 9.4. Questions and Answers

The ideas in this chapter can be used to make simple quiz programs. Here's what a simple Logo quiz might look like:

```
QUIZ1
WHAT IS THE NAME OF THIS BOOK?
LOGO
NO, NOT QUITE. PLEASE TRY AGAIN. . .
WHAT IS THE NAME OF THIS BOOK?
LOGO LEARNING
NO, NOT QUITE. PLEASE TRY AGAIN. . .
WHAT IS THE NAME OF THIS BOOK?
LEARNING WITH APPLE LOGO
YOU GOT IT!
```

If the user types "Learning With Apple Logo," the procedure prints "You got it!" and stops. Any other answer makes the computer print "No, not quite. Please try again. . ." and start the whole process over again. This is the procedure, QUIZ1:

```
TO QUIZ1
PRINT [WHAT IS THE NAME OF THIS BOOK?]
MAKE "ANSWER1 READLIST
IF :ANSWER1 = [LEARNING WITH APPLE LOGO] [PRINT [YOU GO IT!]
    STOP]
PRINT [NO, NOT QUITE. PLEASE TRY AGAIN. . .]
QUIZ1
END
```

Sometimes a Logo command line is too long for a single line in this book. The { symbol is used to remind you to *type* it as a single line, without pressing **RETURN** until the end.

In QUIZ1, the *conditional* command is more complicated than the ones we had earlier. IF has two inputs, a *condition* and an action list. The condition is :ANSWER1 = [LEARNING WITH APPLE LOGO]. The action list is [PRINT [YOU GOT IT!] STOP].

The only way to stop QUIZ1 *without* typing a correct answer is to press **CTRL-G**.

I'll show you two more examples, then you can make up some quizzes of your own.

```
TO QUIZ2
PRINT [WHAT IS THE SHORT FORM OF FORWARD?]
MAKE "ANSWER2 READLIST
IF :ANSWER2 = [FD] [PRINT [CORRECT!] STOP]
PRINT [NO, THAT'S NOT IT.]
PRINT [WOULD YOU LIKE TO TRY AGAIN?]
MAKE "TRY READLIST
IF :TRY = [NO] [PRINT [THE ANSWER IS: FD] STOP]
IF :TRY = [YES] [QUIZ2 STOP]
PRINT [I QUIT! YOU DIDN'T ANSWER YES OR NO.]
END
```

QUIZ2 has an improvement over QUIZ1. It lets the user decide whether to keep trying. If the user types anything other than YES or NO, the procedure complains and quits, without giving the answer.

QUIZ3 has one more improvement. Sometimes a question has more than one correct answer (or an answer that's almost correct). Every possible correct answer should be included in a quiz procedure. One way to do this is with the Logo command OR. OR outputs "TRUE if *any* of its inputs are true. (Another Logo command, AND, outputs "TRUE only if *all* of its inputs are true.)

```
TO QUIZ3
PRINT [HOW MANY INPUTS DOES THE]
PRINT [MAKE COMMAND NEED?]
MAKE "ANS3 READLIST
IF OR :ANS3 = [TWO] :ANS3 = [2] [PRINT [RIGHT.] STOP]
PRINT [NO, MAKE NEEDS TWO INPUTS,]
PRINT [A NAME AND A VALUE]
END
```

QUIZ3 allows two possible correct answers but it does not make the user try again. So you can see there are many possible variations.

Make up several quiz procedures of your own. You can combine them into one big quiz program if you like.

```
TO LOGOQUIZ
QUIZ1
QUIZ2
QUIZ3
QUIZ4
QUIZ5
. . .
. . .
. . .
END
```

There are a lot of things to think about when you're making up a quiz. How should the information be printed on the screen? Should the questions and answers be serious or funny? Should the responses to wrong answers be helpful? Should the computer give clues? How many wrong answers should be allowed before the correct answer is given? Should the computer keep track of how many answers were right and wrong? If you want to make an "intelligent" quiz procedure that accounts for a lot of different choices and possibilities, you may wind up with a very complicated program.

Section 9.5. GUESSNUMBER

GUESSNUMBER is a number guessing game. This example shows how it is played.

```
GUESSNUMBER
I AM THINKING OF A NUMBER BETWEEN 0
AND 100. SEE IF YOU CAN GUESS IT.
> 50
TOO LOW
> 75
TOO LOW
> 85
TOO HIGH
> 80
TOO LOW
> 82
TOO HIGH
> 81
GOT IT!
```

You can play GUESSNUMBER by reading a file called "GUESSNUMBER from your LWAL Procedures Disk or by typing in the procedures given here. If you are going to type in the procedures, you will need a tool procedure called READNUMBER, found on the LWAL Procedures Disk. Just insert the LWAL Procedures Disk into the disk drive and type LOAD "READNUMBER. If you don't yet have a complete LWAL Procedures Disk, you can copy READNUMBER from the listing in Appendix I.

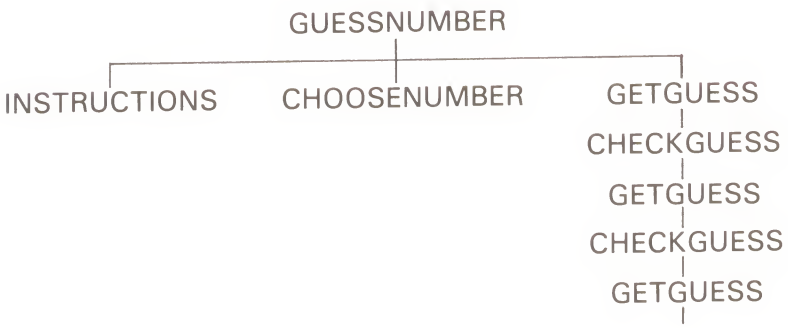
GUESSNUMBER is something like the quiz procedures of the last section except that the computer keeps giving clues until the player guesses correctly. The clues only say whether the guess was too high or too low. Let's examine what the computer has to do to play this game.

1. Print some instructions.
2. Choose a secret number between 0 and 100.
3. Wait for the user to type a guess.
4. Check the guess to see if it is too high, too low, or just right. If it is just right, end the game. If it is too high or too low, print a message and ask for another guess.

Each of these actions is carried out by a separate procedure.

```
TO GUESSNUMBER
INSTRUCTIONS
CHOOSENUMBER
GETGUESS
END
TO INSTRUCTIONS
CLEARTEXT
PRINT [I AM THINKING OR A NUMBER BETWEEN 0]
PRINT [AND 100. SEE IF YOU CAN GUESS IT.]
END
TO CHOOSENUMBER
MAKE "NUMBER 1 + RANDOM 99
END
TO GETGUESS
TYPE ">
MAKE "GUESS READNUMBER
CHECKGUESS :GUESS :NUMBER
END
TO CHECKGUESS :GUESS :NUMBER
IF :GUESS = :NUMBER [PRINT [GOT IT] STOP]
IF :GUESS > :NUMBER [PRINT [TOO HIGH] GETGUESS STOP]
IF :GUESS < :NUMBER [PRINT [TOO LOW] GETGUESS STOP]
END
```

A diagram that shows the order in which each procedure calls its subprocedures is called a *procedure tree*. GUESSNUMBER calls INSTRUCTIONS, CHOOSENUMBER, and GETGUESS. GETGUESS calls CHECKGUESS. CHECKGUESS calls another GETGUESS. Then CHECKGUESS and GETGUESS keep calling copies of each other until the number is guessed.



The lower branches of the tree keep repeating until the number is guessed by the user. A procedure tree is an important tool for understanding how a set of procedures interrelates.

GUESSNUMBER and its subprocedures use three Logo commands that we've never seen before, CLEARTEXT, RANDOM and TYPE, and the tool procedure READNUMBER.

CLEARTEXT clears the text screen and puts the cursor at the top.

RANDOM takes a number as an input and outputs a number between zero and one less than the input number. It outputs a *different* number each time it is called. RANDOM 5 outputs a number between 0 and 4. RANDOM 99 outputs a number between 0 and 98, etc.

TYPE prints a word or list *without* moving the cursor down to the next line.

READNUMBER waits for a user to type a number and then outputs the number. READNUMBER is just like READLIST except that it outputs a *number* and READLIST outputs a *list*. If you want an explanation of how it works, see Chapter 14, "How the Special Tool Procedures Work." Meanwhile, you can use it just as though it were a Logo command without understanding how it works. It can be used in many different projects.

Now let's look at how the procedures work.

```
TO GUESSNUMBER
INSTRUCTIONS
CHOOSENUMBER
GETGUESS
END
```

GUESSNUMBER is a *superprocedure*. All it does is call three subprocedures, one after another.

```
TO INSTRUCTIONS
CLEARTEXT
PRINT [I AM THINKING OF A NUMBER BETWEEN 0]
PRINT [AND 100. SEE IF YOU CAN GUESS IT.]
END
```

The first line of INSTRUCTIONS clears the screen. The next two lines print the instructions for the game.

```
TO CHOOSENUMBER
MAKE "NUMBER 1 + RANDOM 99
END
```

CHOOSENUMBER creates a new variable called "NUMBER with a random value between 1 and 99. It does this by adding 1 to the output of RANDOM 99 (which has a value between 0 and 98).

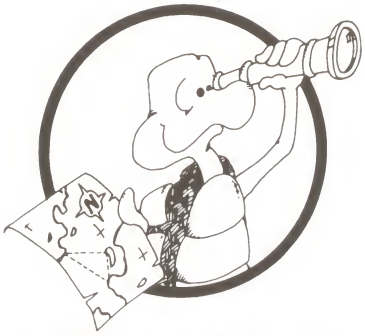
```
TO GETGUESS
TYPE ">
MAKE "GUESS READNUMBER
CHECKGUESS :GUESS :NUMBER
END
```

GETGUESS calls two subprocedures, READNUMBER and CHECKGUESS. The first line of GETGUESS prints the > symbol. The second line prints the symbol > on the screen. The third gives the name "GUESS to whatever number the user types. READNUMBER gets that number from the user. The third line calls CHECKGUESS, which takes two inputs. The first, :GUESS, is the number just typed in by the user. The second, :NUMBER, is the value of the computer's secret number.

```
TO CHECKGUESS :GUESS :NUMBER
IF :GUESS = :NUMBER [PRINT [GOT IT] STOP]
IF :GUESS > :NUMBER [PRINT [TOO HIGH] GETGUESS STOP]
IF :GUESS < :NUMBER [PRINT [TOO LOW] GETGUESS STOP]
END
```

The first line of CHECKGUESS checks to see if the guess is correct. If it is, GETGUESS stops and the game is over. The second line checks to see if the guess is bigger than the secret number. If so, it prints **TOO HIGH** and calls GETGUESS to get another guess from the user. If the guess is smaller than the secret number, the third line prints **TOO LOW** and calls GETGUESS to get another guess.

GETGUESS gets another guess and calls CHECKGUESS again with a *new* value for :GUESS. CHECKGUESS compares this *new* guess with the secret number. CHECKGUESS and GETGUESS keep calling copies of each other until the secret number is guessed. If the user is a lucky or clever guesser, the game may be over quickly. If not, it could go on for a long time.



EXPLORATION

See if you can improve GUESSNUMBER so that it counts the number of guesses you need. To do this, you would need another variable called "COUNT. GETGUESS should increase the value of "COUNT by one every time it is called and print the value every turn, telling you how many turns you've used. Here's what a sample game might look like.

```
GUESSNUMBER2
I AM THINKING OF A NUMBER BETWEEN 0
AND 100. SEE IF YOU CAN GUESS IT.
1 > 50
TOO HIGH
2 > 25
TOO LOW
3 > 35
TOO HIGH
4 > 30
TOO HIGH
5 > 28
GOT IT!
```

You can do this by adding three lines to your procedures: a line in GUESSNUMBER to start counting at zero,

```
MAKE "COUNT 0
```

and two lines in GETGUESS,

```
MAKE "COUNT :COUNT + 1
TYPE WORD :COUNT. ">
```

WORD :COUNT "> combines the value of the count with the > symbol.

Try to make a set of procedures that works like the example.

Section 9.6.
MATHQUIZ

Logo's arithmetical capabilities may also be used to create a simple math quiz. This one lets you practice addition of two-digit numbers. You can figure out yourself how to extend it to any other kind of arithmetic you want to practice.

To try the quiz, read the program from a file called "MATHQUIZ on the LWAL Procedures Disk, or copy the procedures listed in this section. If you copy the procedures you will also need to copy a tool procedure READNUMBER from Appendix I.

Here's an example of how the program works.

```
MATHQUIZ
HOW MANY PROBLEMS WOULD YOU LIKE?
2
PROBLEM 1
35 + 41 = 76
CORRECT
PLEASE TYPE RETURN
PROBLEM 2
78 + 43 = 111
SORRY, THE ANSWER IS 121
```

PLEASE TYPE RETURN
YOUR SCORE IS 1
OUT OF 2 PROBLEMS

Let's see what the procedure has to do.

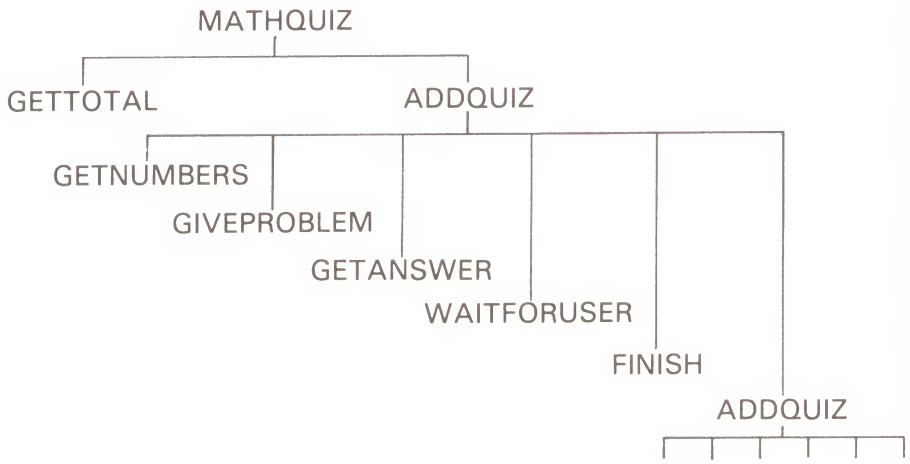
1. Find out how many problems you want to do.
2. Choose two numbers and present the first problem.
3. Check the answer you type—if it is correct, print "correct" and increase your score by one.
4. Check to see if you have completed the number of problems you wanted. If so, print your score and stop. If not, increase the *count* by one, choose two more numbers, and go on.

Here is a set of procedures that perform these tasks:

```
TO MATHQUIZ
CLEARTEXT
GETTOTAL
MAKE "COUNT1
MAKE "SCORE 0
ADDQUIZ :COUNT :TOTAL :SCORE
END
TO GETTOTAL
PRINT [HOW MANY PROBLEMS DO YOU WANT?]
MAKE "TOTAL READNUMBER
END
TO ADDQUIZ :COUNT :TOTAL :SCORE
CLEARTEXT
GETNUMBERS
GIVEPROBLEM
GETANSWER
WAITFORUSER
IF :COUNT = :TOTAL [FINISH STOP]
ADDQUIZ ( :COUNT + 1 ) :TOTAL :SCORE
END
TO GETNUMBERS
MAKE "NUMBER1 RANDOM 100
MAKE "NUMBER2 RANDOM 100
MAKE "RIGHTANSWER :NUMBER1 + :NUMBER2
END
TO GIVEPROBLEM
PRINT SENTENCE [PROBLEM] :COUNT
PRINT [ ]
TYPE ( SENTENCE :NUMBER1 [+] :NUMBER2 [=] )
END
TO GETANSWER
MAKE "RESPONSE READNUMBER
TEST :RESPONSE = :RIGHTANSWER
IFTRUE [PRINT [CORRECT]]
IFTRUE [MAKE "SCORE :SCORE + 1]
```

```
IFFALSE [PRINT SENTENCE [SORRY, THE ANSWER IS]
  :RIGHTANSWER]
END
TO WAITFORUSER
PRINT [PLEASE TYPE RETURN]
PRINT READLIST
END
TO FINISH
CLEARTEXT
PRINT SENTENCE [YOUR SCORE IS] :SCORE
PRINT ( SENTENCE [OUT OF] :TOTAL [PROBLEMS] )
END
```

This procedure tree shows the order in which each procedure and subprocedure is called.



Now let's examine each procedure

```
TO MATHQUIZ
CLEARTEXT
GETTOTAL
MAKE "COUNT 1
MAKE "SCORE 0
ADDQUIZ :COUNT :TOTAL :SCORE
END
TO GETTOTAL
PRINT [HOW MANY PROBLEMS DO YOU WANT?]
MAKE "TOTAL READNUMBER
END
```

MATHQUIZ clears the text screen, calls GETTOTAL to ask the user how many problems to give, and sets the value of "TOTAL. MATHQUIZ then sets the starting value of "COUNT to 1 and "SCORE to 0 and calls ADDQUIZ.

```
TO ADDQUIZ :COUNT :TOTAL :SCORE
CLEARTEXT
GETNUMBERS
GIVEPROBLEM
GETANSWER
WAITFORUSER
IF :COUNT = :TOTAL [FINISH STOP]
ADDQUIZ ( :COUNT + 1 ) :TOTAL :SCORE
END
```

ADDQUIZ starts with three inputs. The value of :COUNT is 1 for the first problem, the value of :TOTAL is the number of problems the user asked for, and the value of :SCORE is 0. ADDQUIZ clears the text screen and calls GETNUMBERS.

```
TO GETNUMBERS
MAKE "NUMBER1 RANDOM 100
MAKE "NUMBER2 RANDOM 100
MAKE "RIGHTANSWER :NUMBER1 + :NUMBER2
END
```

GETNUMBERS chooses the numbers for the problem, finds the correct answer, and gives it the name "RIGHTANSWER. ADDQUIZ then calls GIVEPROBLEM.

```
TO GIVEPROBLEM
PRINT SENTENCE [PROBLEM] :COUNT
PRINT [ ]
TYPE (SENTENCE :NUMBER1 [+] :NUMBER2 [=])
END
```

GIVEPROBLEM just prints the problem. ADDQUIZ then calls GETANSWER.

```
TO GETANSWER
MAKE "RESPONSE READNUMBER
TEST :RESPONSE = :RIGHTANSWER
IFTRUE [PRINT [CORRECT]]
IFTRUE [MAKE "SCORE :SCORE +1]
IFFALSE [PRINT SENTENCE [SORRY, THE ANSWER IS]
  :RIGHTANSWER]
END
```

GETANSWER gets an answer from the user and uses the Logo commands TEST, IFTRUE, and IFFALSE to see whether that answer is correct. If it is correct, the score is increased by one.

TEST, IFTRUE, and IFFALSE are similar to the Logo command IF. TEST has *one* input, a condition which must be either true or false. The action list that follows IFTRUE will be carried out if the condition tested was *true*; the action list following IFFALSE will be carried out if the condition was *false*.

ADDQUIZ then calls WAITFORUSER:

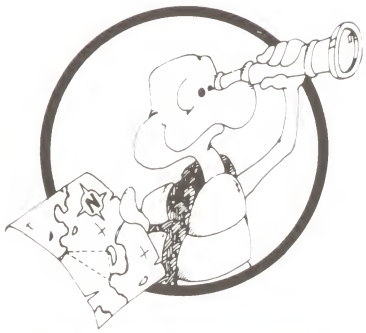
```
TO WAITFORUSER
PRINT [PLEASE PRESS RETURN]
PRINT READLIST
END
```

WAITFORUSER is a clever trick that allows the *user* to decide how long to wait before going on to the next problem. READLIST waits for the user to type something, *anything*. The only purpose of the PRINT command is to give Logo something to do with what the user types. (If the user follows instructions, he or she will just press the **RETURN** key and READLIST will then output an *empty list*, **[ ]**.) WAITFORUSER can be used as a tool in other conversational programs to allow the user to decide when to go on.

The next to last line of ADDQUIZ checks to see if the *count* is equal to the *total* number of problems asked for. If it is, the subprocedure FINISH is called to print the score and ADDQUIZ stops.

```
TO FINISH
CLEARTEXT
PRINT SENTENCE [YOUR SCORE IS] :SCORE
PRINT ( SENTENCE [OUT OF] :TOTAL [PROBLEMS] )
END
```

If the *count* is *not* equal to the *total*, the last line of ADDQUIZ calls *another* ADDQUIZ with :COUNT increased by 1, with the same value of :TOTAL, and a new value for :SCORE (if the user gave the correct answer to the problem).



EXPLORATION

Listed below are some of the many ways you might want to modify this type of program.

- Allow the user to decide how *large* the numbers in the problem will be. This is done by using a number typed by the user (in place of 100) as an input to RANDOM in GETNUMBERS.
- Print the numbers in a different format. For example:

  PROBLEM 1
  17
  +
  28
  – –

  One way to do this is to make blank space using \ (**CTRL-Q**). Try this:

  ```
  PRINT SENTENCE "\ \     25
  ```

  (The spaces after each \ mark are printed just as they appear.)

- If you want to get really fancy, make a set of numbers and symbols for the turtle to draw and "print" the problems and answers on the turtle's screen. This, in itself, would be quite an elaborate turtle geometry project.

- Make problems get *harder* when the user gets a correct answer. This can be done by increasing the value of the largest possible number whenever an answer is correct.
- Change the messages that are printed as the quiz goes along. Make the computer print different messages each time.
- Offer *helpful* suggestions if wrong answers are typed. This is one of the hardest things to do because first you have to decide what kind of suggestions would be helpful.
- Make new quizzes for subtraction, multiplication, and division. Then make a superprocedure which lets the user choose which kind of match to practice.

HELPER'S HINT

Writing computer programs with which a user may interact in a way that is both friendly and helpful is one of the most difficult challenges for any programmer. Most so-called computer-assisted instruction (CAI) is very poorly thought out and often a waste of time for anyone who has to use it. On the other hand, attempting to *write* a CAI program can be a powerful learning experience for two reasons. First, it forces a programmer to think carefully about how a computer and a person should interact to help someone learn. Second, many of the programming ideas involved in such a project will be useful for many other programs, as we will see as we go through the book.

An interesting project for a group of students might be to have each one write a CAI drill program of some kind. Then let people try out each others' programs and evaluate them as a group, both as programs and as educational experiences.

CHAPTER 10

Command	Short Form	Examples With Inputs
SETPOS		SETPOS [100 30], SETPOS [10 −20]
SETX		SETX 50, SETX −100
SETY		SETY 35, SETY −20
SETHEADING	SETH	SETHEADING 90, SETH 30
WAIT		WAIT 100
PENREVERSE	PX	
TOWARDS		SETHEADING TOWARDS [0 0]

LWAL Procedures Disk files used: "SHOOT, "DISTANCE, "CCIRCLE, "READNUMBER, "WAIT

New tool procedures used:

Tool Procedure	Examples
DISTANCE	PRINT DISTANCE [0 100] IF (DISTANCE :PSTART < 10) [ENDGAME]
CCIRCLE	CCIRCLE 50

10 SHOOT: An Interactive Turtle Game

The SHOOT game was introduced as an activity in Chapter 3. The object of the game is to aim the turtle at a target on the screen and then "shoot" the turtle at the target. It is designed to help people learn how to control the turtle—particularly how to estimate angles and distances with the turtle. Before reading this chapter you should look back at Chapter 3 to remind yourself how the game is played.

In this chapter you'll learn how the SHOOT procedures work and how to improve them to make the game more interesting. This is the first of four chapters that deal with *interactive* programming projects. An *interactive* program is one in which there is *communication* between the computer and a person using the computer. What the computer does depends on what the user does, and vice versa. You had a taste of interactive programming in Chapter 9 with the quiz and number-guessing games. This chapter uses ideas from Chapter 9 along with ideas from earlier chapters to make an interactive turtle game.

This chapter and the next three chapters start with a partially completed project that really works. I'll explain how the procedures work and show you how to change them to make the projects more interesting. I'll also use the projects in these four chapters to explain more about how Logo works and how to think about designing your own projects. By the time you finish Chapter 13, you should have a lot of project ideas of your own and know enough about Logo programming to make them happen.

The procedures for each project, as well as the tools needed to make them work, are stored on the LWAL Procedures Disk. If you don't have a complete LWAL Procedures Disk you can copy the procedures from Appendix I.

HELPER'S HINT

My experience in teaching Logo has shown me that people learn as much from adapting and modifying someone else's work as they do from creating their own projects entirely from scratch. In fact, a great deal of "real world" programming activity involves starting with someone else's idea for a program and adapting it for a different use or improving it in some way.

SHOOT is an example of a typical computer programming project—a computer game. Programming a computer so that it interacts with a user and takes on a kind of personality seems to be one of the chief attractions of computer programming for many learners. Most people who continue programming beyond the beginning stages want to make the computer play a game, ask questions and check answers, or engage in some kind of conversation with a user.

This kind of activity involves *data processing*—creating, keeping track of, and manipulating information with a computer. In addition to learning how to design and modify Logo projects, this chapter and the ones that follow will help a reader understand something about what data processing is, how it happens, and why it's important.

As I suggested in a helper's hint in Chapter 3, SHOOT provides an excellent opportunity for a collaboration between older and younger learners. An older, more experienced programmer anywhere from age ten to adult could work with a younger learner, from age five or six on up. They could work together as a team to modify the game so that it is more interesting to both of them.

In a school or computer club, this could be an exciting joint project for a class of sixth or seventh graders working with a class of second or third graders. With a lot of people working together on the same type of project, there would be many opportunities for sharing and borrowing ideas. I wouldn't think of this as a competition to see who could design the *best* SHOOT game, but rather as a collaboration to see how many different and exciting variations could be developed by a group all starting from the same point and sharing ideas and techniques with each other.

Section 10.1. New Logo Commands and Tool Procedures Used in the SHOOT Game

Everything we've done with the turtle so far in this book has used turtle commands like FORWARD, BACK, RIGHT, and LEFT. These commands move the turtle from wherever it is to a new position. Logo can also use turtle commands that move the turtle to an exact position on the screen using *X and Y coordinates*. X and Y coordinates are measured from the center of the screen, as shown in Figure 10.1.

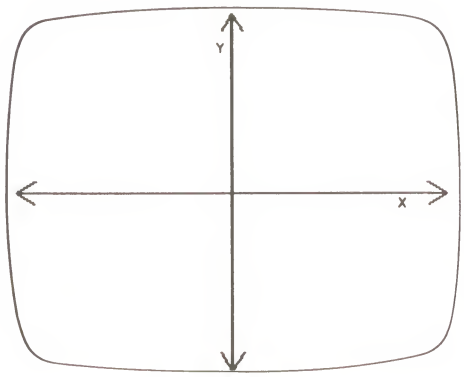


Figure 10.1: The X and Y coordinates of a point tell its horizontal and vertical distance from the center of the screen.

X and Y coordinates are used in the SHOOT game to set the locations of the target and of the turtle's starting position. The turtle can be moved directly to any point on the screen using the command SETPOS with a list of two numbers as input. The first number gives the X (horizontal) position, and the second input gives the turtle a Y (vertical) position. *Positive* values of X move the turtle to the *right* half of the screen; *negative* values move it to the *left* half. *Positive* values of Y move the turtle to the *upper* half of the screen, *negative* values to the *lower* half. Try these examples:

```
CLEARSCREEN
SETPOS [20 30]
SETPOS [-50 75]
SETPOS [-100 -20]
SETPOS [45 -80]
```

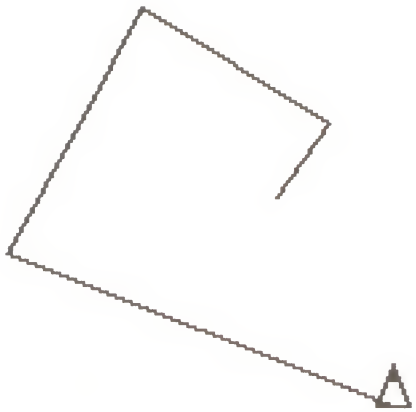


Figure 10.2: SETPOS moves the turtle directly to a fixed point without changing its heading.

When you give a command a *negative* input like −50, be sure there's no space between the − and the 50. Otherwise Logo may think you are trying to subtract.

As you can see from Figure 10.2, SETPOS moves the turtle without changing its heading. You can change the turtle's heading with the command SETHEADING (abbreviated SETH). Try this:

```
SETHEADING 45
SETHEADING 300
SETHEADING 400
SETHEADING 1000
```

Figure 10.3: Changing the turtle's heading with SETHEADING.

If you give SETHEADING an input larger than 360, it will subtract 360 from that value. Thus, SETHEADING 400 has the same effect as SETHEADING 40 (400 − 360) and SETHEADING 1000 has the same effect as SETHEADING 280 (1000 − 360 − 360).

Two other commands, SETX and SETY, each take only one input and change the turtle's X or Y position without affecting the other.

WAIT is a Logo command that makes the computer pause for a while. It needs one input, a number that tells the computer how long to wait. An input of 100 will make the computer wait for about two seconds. Try these examples to see what happens:

```
WAIT 100
WAIT 200
WAIT 1000
```

The SHOOT procedures also use the tool procedures DISTANCE, CCIRCLE, and READNUMBER; these can be loaded from files on your LWAL Procedures Disk or copied from Appendix I.

DISTANCE needs a list as its input, the X- and Y-coordinates of a point. It outputs a message giving the distance between the turtle and the point. It is used in the SHOOT game to tell how far the turtle is from the center of the target. To try these examples, first load the file "DISTANCE from the LWAL Procedures Disk or copy it from Appendix I. Then type

```
CLEARSCREEN
PRINT DISTANCE [100 0]
IF (DISTANCE [100 0] < 10) [PRINT [YOU GOT IT!]]
IF (DISTANCE [5 5] < 10) [PRINT [YOU GOT IT!]]
```

In the first example, the computer prints the distance between the turtle's current position and the point X = 100, Y = 0. In the second and third examples, it checks to see if the turtle is less than 10 units from a point (which might be the center of a target), and if that's true, it prints a message saying you've hit the target.

CCIRCLE takes one input, the radius of a circle, and draws a circle whose *center* is the turtle's starting point. To try CCIRCLE, load "CCIRCLE from the LWAL Procedures Disk or copy it from Appendix I.

```
CCIRCLE 50
```

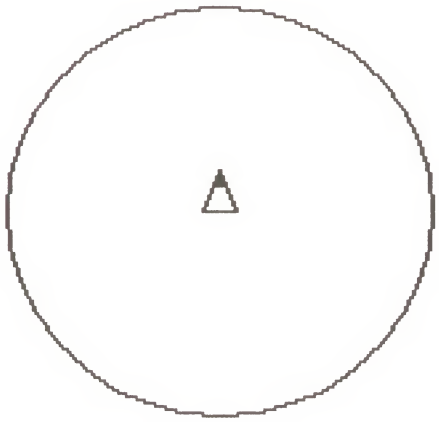


Figure 10.4: CCIRCLE draws a circle with the turtle at its center.

CCIRCLE is used in the SHOOT game with SETPOS to move the turtle to a particular point and draw a circle whose center is at that point. The same point is later used as input to DISTANCE to check whether the turtle has hit the target.

READNUMBER, which was also used in Chapter 9, is like the Logo command READLIST. It waits for the user to type a *number* at the keyboard and outputs that number as a message. (READLIST also waits for the user to type something, but it outputs a *list* as a message rather than a number.)

To try READNUMBER, load the "READNUMBER file from the LWAL Procedures Disk or copy it from Appendix I. Then try this example:

```
PRINT READNUMBER + 55
```

When you type this command, the computer will wait for you to type a number followed by **RETURN**. It will then add 55 to the number you typed and print the result. If you type something that is *not* a number, Logo will complain.

Now that you understand SETPOS, SETHEADING, WAIT, DISTANCE, CCIRCLE, and READNUMBER, you are ready to understand how the SHOOT game procedures work. A few additional commands and tool procedures that are less important than these will be explained as you go along.

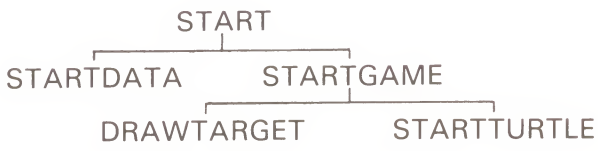
Section 10.2. How the SHOOT Game Works

The SHOOT game program is made up of seven different procedures. Two of them, START and SHOOT, are *superprocedures*, that is, commands that you type to make things happen. The subprocedures are called STARTDATA, STARTGAME, DRAWTARGET, STARTTURTLE, HIT, and MISS. The project also uses three tool procedures: DISTANCE, CCIRCLE, and READNUMBER, which were described in the last section.

At this point you can either load the SHOOT procedures from the file called "SHOOT on your LWAL Procedures Disk or type them in as you read this chapter. If you are typing the procedures yourself you will have to load the tool procedures from their files on your LWAL Procedures Disk or copy them from Appendix I.

The START Superprocedure

We'll begin by looking at the first superprocedure, START, and its subprocedures, STARTDATA, STARTGAME, DRAWTARGET, and STARTTURTLE. This *procedure tree* shows how the procedures are organized:



START commands two subprocedures, STARTDATA and STARTGAME. STARTGAME also commands two subprocedures, DRAWTARGET and STARTTURTLE. Here's how they all fit together:

```
TO START
STARTDATA
STARTGAME
END
```

START calls STARTDATA to set up the starting conditions for the game and then calls STARTGAME to draw the target and place the turtle in its starting position.

```
TO STARTDATA
MAKE "SHOTNUMBER 0
MAKE "XTARGET (90 - 10 * RANDOM 19)
MAKE "YTARGET (80 - 10 * RANDOM 6)
MAKE "XSTART (90 - 10 * RANDOM 19)
MAKE "YSTART (-10 * RANDOM 3)
MAKE "HSTART (10 * RANDOM 36)
MAKE "PTARGET SENTENCE :XTARGET :YTARGET
MAKE "PSTART SENTENCE :XSTART :YSTART
END
```

STARTDATA gives names to all the variables that store the *information* (data) that will be needed by the computer during the game. It uses the MAKE command to assign the proper value to each name. For example,

```
MAKE "SHOTNUMBER 0
```

creates a variable named "SHOTNUMBER with starting value of 0.

The next five lines use the Logo command RANDOM to create values for the target location and the starting location and heading of the turtle. The use of RANDOM assures that each time the game is played, the locations will be different. "XTARGET and "YTARGET are the names of the X and Y coordinates of the center of the target. Every time the game is played, the X coordinate of the target will be somewhere between −90 and 90 and the Y coordinate will be between 30 and 80.

"XSTART, "YSTART, and "HSTART are the names of the X and Y coordinates of the turtle's starting position and its starting heading. That position will always be somewhere between −90 and 90 in the X direction and between −20 and 0 in the Y direction. The starting heading will be between 0 and 350 degrees. The last two commands make the target and starting positions into lists.

```
TO STARTGAME
CLEARSCREEN
SETBG 6
HIDETURTLE
DRAWTARGET :PTARGET
STARTTURTLE :PSTART :HSTART
SHOWTURTLE
END
```

STARTGAME uses DRAWTARGET and STARTTURTLE to draw the target and start the turtle in the position chosen by STARTDATA. The SETBG 6 command in the second line makes the turtle draw the thinnest possible lines so that its actions are as clear as possible for playing the game.

```
TO DRAWTARGET :PTARGET
PENUP
SETPOS :PTARGET
CCIRCLE 10
END
```

DRAWTARGET uses SETPOS and the tool procedure CCIRCLE to draw a circular target at the position chosen by STARTDATA.

```
TO STARTTURTLE :PSTART :HSTART
PENUP
SETPOS :PSTART
SETHEADING :HSTART
END
```

STARTTURTLE uses SETPOS and SETHEADING to set the turtle at the starting position chosen by STARTDATA.

The SHOOT Superprocedure

The second superprocedure, SHOOT, is used after a player has aimed the turtle toward the target and is ready to try a shot. This procedure tree shows the organization of SHOOT and its three subprocedures. The tool procedures (DISTANCE and READNUMBER) are not shown in the procedure tree.

The superprocedure SHOOT is the most important one used in the entire game. Let's study it step-by-step.

```
TO SHOOT
MAKE "SHOTNUMBER :SHOTNUMBER + 1
PRINT [HOW FAR?]
MAKE "SHOT READNUMBER
PENDOWN FORWARD :SHOT
TEST DISTANCE :PTARGET < 10
IFTRUE [HIT]
IFFALSE [MISS]
END
```

In the line

```
MAKE "SHOTNUMBER :SHOTNUMBER + 1
```

the variable "SHOTNUMBER, is used to keep track of how many shots you have taken. Every time you type SHOOT, the value of "SHOTNUMBER is increased by one. The Logo command MAKE has two inputs. The first input, "SHOTNUMBER, creates a name. The second, :SHOTNUMBER + 1, gives that name a value. In this case, the computer makes the name "SHOTNUMBER stand for the *old* value, :SHOTNUMBER, plus one. This idea is used in many computer programs. It is called *incrementing the value of* "SHOTNUMBER or *updating* "SHOTNUMBER.

Next, the line

```
PRINT [HOW FAR?]
```

asks the player to type a number. Then,

```
MAKE "SHOT READNUMBER
```

gives the name "SHOT to whatever number the player types.

The fourth line

```
PENDOWN FORWARD :SHOT
```

moves the turtle forward the amount of the shot and draws a line on the TV screen.

```
TEST (DISTANCE :PTARGET) < 10
```

tests whether the distance to the center of the target is less than 10 turtle steps. The TEST command makes the computer temporarily store the result, *true* or *false*. The parentheses around DISTANCE and its input make the line easier to read. They are not needed by Logo.

The sixth line

```
IFTRUE [HIT]
```

calls the HIT procedure to declare a "hit" if the result of the test was *true*.

Finally,

```
IFFALSE [MISS]
```

calls the MISS procedure to declare a "miss" if the test was *false*, wait a little while, and then send the turtle back to where it started so that you can try again.

The HIT and MISS procedures are quite simple.

```
TO HIT
PRINT [CONGRATULATIONS! YOU HIT THE TARGET!]
PRINT (SENTENCE [IT TOOK YOU ONLY] :SHOTNUMBER [SHOTS])
END
```

The parentheses in the second line of HIT *are* needed because SENTENCE has more than two inputs.

```
TO MISS
PRINT SENTENCE [MISSED! SHOT NUMBER] :SHOTNUMBER
WAIT 200
STARTTURTLE :PSTART :HSTART
END
```

After telling you that you've missed your shot and how many shots you've had, the computer waits a little while before using STARTTURTLE—the same STARTTURTLE procedure that was used to place the turtle in position at the beginning of the game—to return the turtle to its starting point.

HELPER'S HINT

The SHOOT game procedures make use of *global or public* variables whose values can be known and used by all procedures. This is in contrast to procedures that use *local or private* variables, that is, variables that are defined in the title of a particular procedure. Global variables are useful where the same variables are used by several procedures and when changes in any of the variables need to be maintained for all the procedures that use them.

When using global variable names it is critical to make sure that there is no duplication of names anywhere throughout your entire system of procedures. With local variables this is not necessary because the names are private for each procedure. Two procedures can use the same names for *different* local variables without any problem. This is why many programmers prefer to use local variables whenever possible.

Section 10.3. Ways to Improve the SHOOT Game

Almost everyone who plays SHOOT has ideas for improving the game. Unlike many other computer games you may have played, you can change SHOOT very easily. Some of the ideas that people have suggested for improving the game are:

1. Make the game more *interesting* by having something wonderful happen when the turtle hits the target.
2. Make the game *harder* to play. One way to do this would be to make the target smaller. Another way would be to make the turtle move with its pen up, so that you can't see where your last shot went.
3. Make the game *easier* by making the target larger or by having the computer give helpful hints after you miss a shot.

4. Make the printed messages more interesting or amusing or make the computer print a set of instructions for new players.
5. Make a SUPERSHOOT game that combines several of these ideas and allows a player to choose whether the game will be hard, easy, or medium.

In the rest of this chapter, you'll learn how to make these changes. Learning how to change the SHOOT game will make it easier for you to change any Logo game or activity.

Section 10.4. Making the Game More Interesting

One of the best ways to add interest to a game like SHOOT is to make something interesting happen when the turtle hits the target. It could be some kind of explosion, for example. Here's one possibility:

```
TO EXPLODE :SIZE
HIDETURTLE
REPEAT 18 [FORWARD :SIZE BACK :SIZE RIGHT 20]
END
```

EXPLODE has a variable size input so that you can decide later how big the explosion should be.

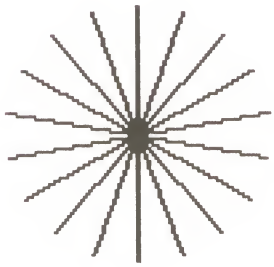


Figure 10.5: Adding an explosion to the SHOOT game might make it more fun.

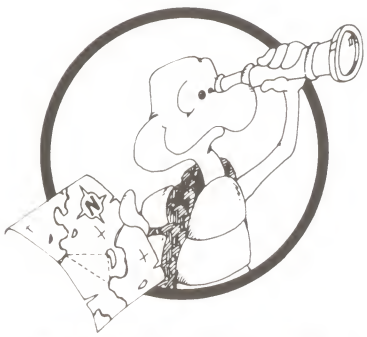
The next thing to do is to decide where to put the EXPLODE procedure into the SHOOT game. If you look at the SHOOT procedure in Section 10.2, you will see that one of its subprocedures is called HIT. HIT has the job of printing the message telling that the turtle hit the target. You have a choice about where to put EXPLODE. You can either add it to SHOOT in the same line as HIT or add it directly to HIT. If you add it to the sixth line of SHOOT, that line will become

```
IFTRUE [HIT EXPLODE 20]
```

If you add it to HIT, it can be placed before or after the print statements.

```
TO HIT
PRINT [CONGRATULATIONS! YOU HIT THE TARGET!]
PRINT (SENTENCE [IT TOOK YOU ONLY] :SHOTNUMBER [SHOTS])
EXPLODE 20
END
```

Both ways of adding EXPLODE to SHOOT will have the same *effect* as far as a player of the game is concerned. It's up to you to decide which way will make the procedures clearer to you (in case you ever want to change them again or explain them to someone).



You can probably think of a lot of ways to make more interesting explosions. Make them different colors or sizes. Make them change a little every time the game is played. Make them larger or smaller depending on how many shots it took to hit the target. Or, you might have an entirely different kind of effect. Make a "flag" appear on the target after a hit or have the target itself disappear in some dramatic way.

EXPLORATION

Section 10.5. Making the SHOOT Game Harder

Many people find the SHOOT game quite easy after a little practice. Another way to make the game more interesting is to make it more challenging.

One simple way to do this is to make the turtle move *without* drawing lines. The lines make it easier for a player to tell whether to turn the turtle more or less than the time before and whether to shoot it longer or shorter. If the lines are not visible, a player will be forced to *remember* more about what has already happened. The simplest way to do this is to remove the PENDOWN command from the fourth line of the SHOOT procedure as given in Section 10.2.

If you want to leave some kind of clue without showing the whole line, you might have the turtle draw an "X" or make some other kind of mark on the screen before it returns to its starting point.

Another way to give a more limited clue would be to have the turtle draw the line as usual when the shot is made, but *erase* it when it returns to the starting point. You can do this by having the turtle draw its lines using PENREVERSE.

When you use PENREVERSE, the turtle *will* draw a line if there is not a line already on the screen. If there is a line on the screen when the turtle is set to PENREVERSE, the line will be erased as the turtle moves over it. Try this:

```
DRAW
PENREVERSE
FORWARD 50
BACK 50
```

You should see the turtle draw a line as it goes forward and erase it as it goes backward. To use this effect, the command PENREVERSE should be added to the fourth line of SHOOT, instead of PENDOWN.

```
PENREVERSE FORWARD :SHOT
```

There's one other problem. The turtle's pen cannot be up when it draws with PENREVERSE. We have to change the MISS procedure to remove the PENUP command as the turtle returns to its starting point. MISS uses STARTTURTLE to move the turtle back to its original position.

```
TO MISS
PRINT SENTENCE [MISSED! SHOT NUMBER] :SHOTNUMBER
WAIT 200
STARTTURTLE :PSTART :HSTART
END
```

But STARTTURTLE includes a PENUP command.

```
TO STARTTURTLE :PSTART :HSTART
PENUP
SETPOS :PSTART
SETHEADING :HSTART
END
```

To eliminate the PENUP, replace STARTTURTLE in MISS by a new procedure called RESTART, which is the same as STARTTURTLE but without PENUP.

```
TO RESTART :PSTART :HSTART
SETPOS :PSTART
SETHEADING :HSTART
END
```

and

```
TO MISS
PRINT SENTENCE [MISSED! SHOT NUMBER] :SHOTNUMBER
WAIT 200
RESTART :PSTART :HSTART
END
```

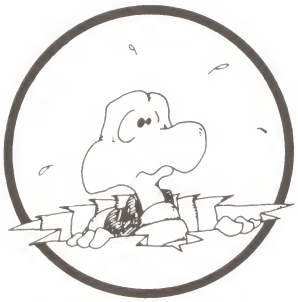
To try out these changes, play the game to make sure it works properly.

Another way to make the game harder is to make the target smaller. To do this, change DRAWTARGET so that it draws a smaller circle.

```
TO DRAWTARGET :PTARGET
PENUP
SETPOS :PTARGET
CCIRCLE 5
END
```

Originally, CCIRCLE had an input of 10. Remember that you also have to change SHOOT so that when the computer *checks* the distance from the turtle to the target, it will use the *new* target size. To do this you have to change the fifth line of SHOOT.

```
TEST (DISTANCE :TARGET) < 5
```



PITFALL

One of the most common sources of bugs in a computer program is when someone makes a change in one part of the program and forgets to change other parts of the program that are related to the part that was changed. If you changed the size of the target without changing the TEST line in SHOOT, the target would *look* smaller, but the computer would still print a "hit" message if the turtle was within *10* units of the target—the *original* distance.

POWERFUL IDEA

If you are going to vary the size of the target a lot, you can solve this problem in advance by making *both* the target size in DRAWTARGET *and* the distance checked in SHOOT depend on a *variable* that could be set at the beginning of the game by STARTDATA. If you did this, you would have to change STARTDATA and STARTGAME as well as DRAWTARGET and SHOOT.

Suppose you call the new variable "RTARGET (meaning "radius of the target"). Then DRAWTARGET would need a new input.

```
TO DRAWTARGET :PTARGET :RTARGET
PENUP
SETPOS :PTARGET
CCIRCLE :RTARGET
END
```

and the DRAWTARGET line in STARTGAME would be changed to

```
DRAWTARGET :PTARGET :RTARGET
```

The fifth line of SHOOT would become

```
TEST DISTANCE :PTARGET < :RTARGET
```

To make this all work properly, a new line needs to be added to STARTDATA.

```
MAKE "RTARGET 5
```

The big *advantage* of changing all these procedures is that now you can easily change the target size any time you want to just by changing the number assigned to "RTARGET in STARTDATA. All the other changes will then be made automatically.

Section 10.6. Making the Game Easier

Some new players may find even the original SHOOT game very difficult. For such players you will want to make an *easier* version of the game. The simplest way to do this would be to make the target larger by *reversing* the changes suggested in the last section—making the target have a radius of 15 or 20, for example, If you already made the target size a variable by following the suggestions in Section 10.5, you just have to change the line in STARTDATA that sets the value of "RTARGET and everything else will be reset automatically.

It might be even more interesting to make the game program "intelligent" enough to give a player some help after missing a shot. To do this, the program needs to keep track of some more information. It needs to know the distance and the angle from the starting point to the target and to keep track of a player's moves so that it can compare a missed shot with a correct shot and give the player some advice. Here's how that might work.

You will need to use the DISTANCE tool procedure and the Logo command TOWARDS to calculate the distance and direction of the target from the turtle's starting position. Add two more variables to STARTGAME, "DTARGET and "HTARGET, to keep track of the distance and the heading of the target from the place where the turtle starts. You can do this by adding two more lines to STARTGAME.

```
MAKE "DTARGET DISTANCE :PTARGET
MAKE "HTARGET TOWARDS :PTARGET
```

The first new line gives the name "DTARGET to the distance from the turtle's starting position to the target. The second uses TOWARDS to give the name "HTARGET to the heading of the target from the starting position. Later the computer will compare these values with the distance and heading of the player's actual shot. STARTGAME now becomes

```
TO STARTGAME
HIDETURTLE
DRAWTARGET :PTARGET :RTARGET
STARTTURTLE :PSTART :HSTART
MAKE "DTARGET DISTANCE :PTARGET
MAKE "HTARGET TOWARDS :PTARGET
SHOWTURTLE
END
```

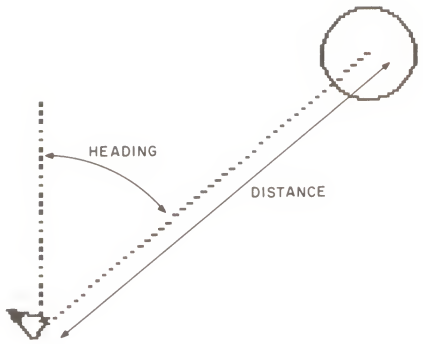


Figure 10.6: Two new variables, "DTARGET and "HTARGET, keep track of the starting distance and the heading of the target.

Now you need to add a new variable called "SHOTANGLE to SHOOT in order to keep track of the *angle* of the player's shot. SHOOT already has the variable "SHOT that keeps track of the *distance* of the shot.

```
TO SHOOT
MAKE "SHOTNUMBER :SHOTNUMBER + 1
MAKE "SHOTANGLE HEADING
PRINT [HOW FAR?] MAKE "SHOT READNUMBER
PENDOWN FORWARD :SHOT
TEST (DISTANCE :PTARGET) < :RTARGET
IFTRUE [HIT]
IFFALSE [MISS]
END
```

The last thing to do is add a new command called HELP to MISS.

```
TO MISS
PRINT SENTENCE [MISSED SHOT NUMBER] :SHOTNUMBER
WAIT 200
STARTTURTLE :XSTART :YSTART :HSTART
HELP
END
```

HELP *asks* if the player wants help, and if so, it gives some adivce about the next shot.

```
TO HELP
PRINT [WOULD YOU LIKE SOME HELP WITH YOUR NEXT SHOT?]
PRINT [PLEASE ANSWER YES OR NO.]
MAKE "ANSWER READLIST
IF :ANSWER = [YES] [GIVEADVICE STOP]
IF :ANSWER = [NO] [STOP]
HELP
END
```

Putting the command HELP in the last line forces a player to answer either yes or no. Any other answer will cause the computer to ask the question again.

Now you need the most important new subprocedure, GIVEADVICE. This is the one that does all the "intelligent" work of checking the difference between the player's actual shot and a winning shot. See if you can figure out how GIVEADVICE works.

```
TO GIVEADVICE
TEST :SHOT > (:DTARGET + :RTARGET)
IFTRUE [PRINT [TRY MAKING YOUR NEXT SHOT SHORTER.]]
TEST :SHOT < (:DTARGET - :RTARGET)
IFTRUE [PRINT [TRY MAKING YOUR NEXT SHOT LONGER.]]
TEST :SHOTANGLE > (:HTARGET + 5)
IFTRUE [PRINT [TRY TURNING THE TURTLE MORE TO THE LEFT,]]
IFTRUE [PRINT [OR LESS TO THE RIGHT.]]
TEST :SHOTANGLE < (:HTARGET - 5)
IFTRUE [PRINT [TRY TURNING THE TURTLE MORE TO THE RIGHT,]]
IF TRUE [PRINT [OR LESS TO THE LEFT.]]
END
```

Notice that GIVEADVICE allows for a *range* of possible distances and angles. The procedure won't give any advice about the distance if the last shot landed within the correct range of distances, nor will it give any advice about the angle if the shot was aimed within five degrees of the correct angle.

Does this advice really help anyone? You'll have to try these changes with a beginning player to see. Even if this particular set of procedures isn't really helpful, you might find it interesting to think about what a procedure should "know" in order to give intelligent help to a learner.

Section 10.7. Adding Instructions and Changing Messages

Some people like a game to provide its own instructions. To do this, just add to START a procedure called INSTRUCTIONS. Sometimes it's hard to decide exactly how much to tell someone about playing a game. It's up to you to decide what the instructions should be and how to print them on the screen so they they are easy to read. You need to plan each line so that it has no more than forty characters (including letters, numbers, and spaces). It's also probably a good idea to leave blank lines between different instructions by using PRINT [ ] to print an empty list.

Another good idea is to let the player choose whether he or she wants instructions. Nothing is more boring than having to read instructions before playing a game you already know how to play.

Remember that different people read at different speeds. It's nice to allow each reader to decide when to go on to the next screen. You can use the procedure WAITFORUSER (from Chapter 9) at the end of each page so that the reader can control what happens.

```
TO WAITFORUSER
PRINT [PLEASE PRESS RETURN]
PRINT READLIST
END
```

As explained in Section 9.6, the *trick* is that READLIST makes the computer wait until the user types a message. Because of the READLIST command, the computer will *wait* for the reader to press **RETURN** before doing anything else.

To change the messages printed throughout the game, you need to change the procedures that print messages: SHOOT, HIT, MISS, and, if you have them, HELP and GIVEADVICE. It is entirely up to you to decide what these messages should say. Personally, I object to messages that insult me when I'm playing a game, like

YOU DUMMY, YOU MISSED THE TARGET AGAIN!

Other people may find insulting messages funny. You have to decide what you like. Remember that what may seem like a very funny message to you when you're inventing a game may get rather boring when you're seeing it for the ninety-ninth time.

Section 10.8. Putting All the Options Together

By now you may have made several different versions of the SHOOT game—the original one with a few special effects, a harder version, an easier one that gives advice, one that offers instructions, etc. Now let's put them all together into one "supergame" that lets each player choose different versions. To do this, we will have to add a procedure called CHOICES to the beginning of the game. It should be the first command in the START procedure.

```
TO START
CHOICES
CLEARSCREEN
STARTDATA
STARTGAME
END
TO CHOICES
TEXTSCREEN
CLEARTEXT
PRINT [WELCOME TO THE GAME OF SUPERSHOOT]
PRINT [DO YOU WANT INSTRUCTIONS?]
MAKE "ANSWER READLIST
IF :ANSWER = [YES] [INSTRUCTIONS]
CHOOSELEVEL
END
```

```
TO CHOOSELEVEL
CLEARTEXT
PRINT [WHAT LEVEL GAME WOULD YOU LIKE TO PLAY?]
PRINT [ ]
PRINT [1—EASY GAME: THE COMPUTER WILL HELP]
PRINT [2—MEDIUM GAME]
PRINT [3—HARD GAME]
PRINT [ ]
PRINT [PLEASE TYPE 1, 2, OR 3.]
PRINT [THEN TYPE RETURN.]
MAKE "CHOICE READNUMBER
IF :CHOICE = 1 [MAKE "LEVEL "EASY STOP]
IF :CHOICE = 2 [MAKE "LEVEL "MEDIUM STOP]
IF :CHOICE = 3 [MAKE "LEVEL "HARD STOP]
CHOOSELEVEL
END
```

Sometimes people get confused when a variable happens to have a word as its *value*. In this case, "EASY, "MEDIUM, and "HARD are *not* variable names. The value of the variable named by the word "LEVEL just happens to *also* be a word.

The computer now "knows" whether the variable named "LEVEL has the value "EASY, "MEDIUM, or "HARD. Now you can change the game procedures to do the right thing, depending on what choice the player has made. For example, we need *three* new lines in STARTDATA to choose the target size for an easy, medium, or hard game.

```
IF :LEVEL = "EASY [MAKE "RTARGET 20]
IF :LEVEL = "MEDIUM [MAKE "RTARGET 10]
IF :LEVEL = "HARD [MAKE "RTARGET 5]
```

If a player chooses an *easy* game, we want the computer to give advice whenever a shot is missed, so we add a line to MISS.

```
TO MISS
PRINT SENTENCE [MISSED SHOT NUMBER] :SHOTNUMBER
WAIT 200
RESTART :PSTART :HSTART
IF :LEVEL = "EASY [HELP]
END
```

HELP is the procedure we used in Section 10.6 with GIVEADVICE as a subprocedure.

STARTGAME also has to include the lines from Section 10.6 that give names to the values "DTARGET and "HTARGET.

If the player chooses to play a *medium* game, nothing else has to happen. For a *hard* game, we want the computer to erase the line drawn by the turtle after each shot. This happens by using PENREVERSE in the SHOOT procedure and making sure there is no PENUP command in the RESTART procedure, as described in Section 10.5.

```
TO SHOOT
MAKE "SHOTNUMBER :SHOTNUMBER + 1
MAKE "SHOTANGLE HEADING
PRINT [HOW FAR?]
MAKE "SHOT READNUMBER
TEST :LEVEL = "HARD
IFTRUE [PENREVERSE]
IFFALSE [PENDOWN]
FORWARD :SHOT
TEST (DISTANCE :PTARGET) < :RTARGET
IFTRUE [HIT EXPLODE 20]
IFFALSE [MISS]
END
```

and

```
TO RESTART :PSTART :HSTART
SETPOS :PSTART
SETHEADING :HSTART
END
```

This example clearly shows how complicated things can get when you make changes to a system of procedures like the SHOOT game. You're almost sure to have forgotten something somewhere. There's no substitute for trying the game several times after making changes to make sure that everything works properly.

When you test a game, make sure you try every possible choice and play the game all the way through each time. Better still, find a friend who is willing to help you by playing the game several times, looking for bugs. Be careful though—your friend may have several new suggestions for improving the game and you may find that your project isn't finished yet!

CHAPTER 11

Command	Short Form	Examples With Inputs
THROW		IF :COM = "E [THROW "TOPLEVEL]
THING		PRINT THING :PICT

LWAL Procedures Disk files used: "QUICKDRAW, "READKEY, "READNUMBER

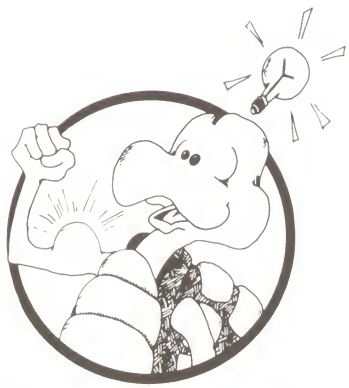
New tool procedures used:

Tool Procedure	Examples
READKEY	MAKE "COM READKEY

11 QUICKDRAW: A Turtle Drawing Activity for Young Children

In Chapter 3 QUICKDRAW was used as an introductory activity. Programs like QUICKDRAW were originally invented for very young children so that they could draw with the turtle and begin to explore Logo without the difficulty of typing out full Logo commands and input numbers. Similar programs have been used with people who have physical disabilities that make it hard for them to type.

QUICKDRAW is an *interactive* project like the quizzes of Chapter 9 or the SHOOT game of Chapter 10. It differs from those games by using single-key inputs from the keyboard. Each key stands for an action. As soon as you press a key, something happens. You don't even have to type **RETURN** first. This makes it really easy for a beginner to use.



POWERFUL IDEA

There is a very important idea here. You can use Logo to change the way Logo itself works! People often ask, "What is the earliest age at which children can begin to learn Logo?" My answer is usually something like, "You can *adapt* Logo for children of just about any age. All you have to do is decide how you'd like it to be, and then create a Logo procedure to make it that way." What you're really doing is using Logo to create a special learning environment for someone special.

I suggest that you try this as a project: find someone a lot younger than you are and use QUICKDRAW as a starting point for making a Logo learning environment. Watch carefully as it is used and redesign it to make it better. I'll explain more about how to do this at the end of this chapter.

Section 11.1. How the QUICKDRAW Procedures Work

Before reading this chapter, reread Section 3.2 of Chapter 3, "Drawing with an 'Instant' Turtle." This will remind you of how QUICKDRAW works. If you already have the "QUICKDRAW file on your LWAL Procedures Disk, you might also want to play with QUICKDRAW for a while before reading any further.

QUICKDRAW uses the tool procedure READKEY, which is also on the LWAL Procedures Disk. If you are typing in the procedures now, load the "READKEY file from the LWAL Procedures Disk before going on. If you don't have READKEY on your LWAL Procedures Disk, you can copy it from Appendix I.

First, let's look at a very simple quick-drawing program, then we can improve it to be the same as the one in Chapter 3. Finally, I'll show you some other extensions and improvements you can make.

A quick drawing program has to be able to "read" the keys that a user

types. If the user types **F** the turtle should move forward a little bit. If **R** or **L** are typed the turtle should turn right or left. The drawing should end when **E** is typed. (The keys **F**, **R**, **L**, and **E** are chosen because they correspond to the Logo commands FORWARD, RIGHT, LEFT, and END. You can choose any keys you like for this. Some people like to group the keys in one place on the keyboard to make them easier to find. I like to use these particular keys because it helps someone to use standard Logo commands. Type in DRAW, then type QUICKDRAW and try it.

```
TO QUICKDRAW
COMMAND
QUICKDRAW
END
TO COMMAND
MAKE "COM READKEY
IF :COM = "F [FORWARD 20]
IF :COM = "R [RIGHT 30]
IF :COM = "L [LEFT 30]
IF :COM = "E [THROW "TOPLEVEL]
END
```

QUICKDRAW is about as simple as it could be. It calls COMMAND to get a new command from the user and then calls another QUICKDRAW procedure. COMMAND is also a very simple procedure. The first line gives the name "COM (short for "command") to whatever key the user types. The next four lines check to see if this is one of the *active* keys, **F**, **R**, **L**, or **E**. If not, nothing happens. If :COM *is* one of the active keys, the rest of the line tells the computer what to do—move the turtle, turn it, or stop the procedure. When the command THROW "TOPLEVEL is used in *any* procedure, it makes *all* procedures stop. The Logo ? *prompt* appears and the user must type commands on the keyboard to make something happen.

PITFALL

You might think that the Logo command STOP would do this, but STOP only stops the procedure it is in. In this case, using the STOP command in the line

```
IF :COM = "E [STOP]
```

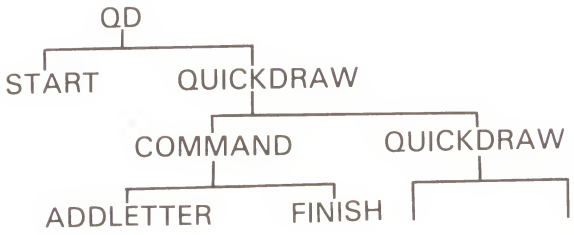
would *not* stop QUICKDRAW; it would stop only COMMAND. QUICKDRAW would then go on to *its* next line and call another QUICKDRAW, which calls COMMAND, and so on. Try using STOP in place of THROW "TOPLEVEL and see what happens. This is a very common bug—one that can be very difficult to detect.

Section 11.2. Making QUICKDRAW Remember Its Moves

The QUICKDRAW procedure we have right now can draw pictures only once. When the screen is cleared, the picture is forgotten forever. We need a way to give a picture a name and keep track of all the steps in it so that it can be redrawn on command. There are many ways to accomplish this. In

this version, QUICKDRAW keeps a *list* of all the commands that have been typed. When you finish a drawing, that list is given a name. Another procedure, REDRAW, uses that list to redraw the original picture whenever you want it.

To make QUICKDRAW work the way it did in Chapter 3, we need a superprocedure that we'll call QD and several other new procedures. Here is a procedure tree showing QD and its subprocedures:



The superprocedure QD and its subprocedures START and QUICKDRAW are still quite simple:

```
TO QD
START
QUICKDRAW
END
```

QD calls START to get things going and then calls QUICKDRAW.

```
TO START
MAKE "DRAWLIST [ ]
CLEARSCREEN
END
```

"DRAWLIST is the list used to keep track of all the commands typed. START makes it an empty list, [ ], and clears the turtle's screen.

```
TO QUICKDRAW
COMMAND
QUICKDRAW
END
```

QUICKDRAW remains the same, although its subprocedure COMMAND, is changed to include two new procedures, ADDLETTER, which adds letters to :DRAWLIST, and FINISH, which makes the user give the drawing a name. Another command has also been added to allow the turtle to move backward as well as forward.

```
TO COMMAND
MAKE "COM READKEY
IF :COM = "F [FORWARD 20 ADDLETTER :COM]
IF :COM = "B [BACK 20 ADDLETTER :COM]
IF :COM = "R [RIGHT 30 ADDLETTER :COM]
IF :COM = "L [LEFT 30 ADDLETTER :COM]
IF :COM = "E [FINISH THROW "TOPLEVEL]
END
TO ADDLETTER :LETTER
MAKE "DRAWLIST SENTENCE :DRAWLIST :LETTER
END
```

ADDLETTER uses SENTENCE to add a new letter to :DRAWLIST every time any command is carried out.

```
TO FINISH
SPLITSCREEN
PRINT [PLEASE CHOOSE ONE WORD AS A NAME]
PRINT [FOR THIS DRAWING.]
PRINT [TO FORGET IT, JUST PRESS RETURN]
MAKE "REPLY READLIST
IF :REPLY = [ ] [STOP]
MAKE (FIRST :REPLY) :DRAWLIST
END
```

Here's the last line of FINISH again, with its input more clearly separated:

```
MAKE   (FIRST :REPLY)   :DRAWLIST
```

This last line uses MAKE in a new way. Every time we've used MAKE before, it has been something like

```
MAKE "START HEADING
```

where MAKE's first input is a one word name and its second input is a piece of Logo data. This time, MAKE gets its first input from :REPLY, a one word *list* that the user just typed. (The Logo command READLIST *always outputs a list*.) Since :REPLY is a *list* and MAKE needs a *word* as its first input, we use the Logo command FIRST to get the first word of :REPLY and use it as MAKE's first input. MAKE's second input is :DRAWLIST, the list of letters that has stored all the commands used so far by QUICKDRAW.

HELPER'S HINT

The tricky point here is the difference between a *word* and a *list with only one word in it*, and the way you can turn a one-word list into a word using FIRST. Try this:

```
PRINT [HELLO]
HELLO
PRINT "HELLO
HELLO
PRINT "HELLO = [HELLO]
FALSE
```

Even though [HELLO] and "HELLO look the same when you print them, they are different objects to Logo. We can make them identical by using FIRST.

```
PRINT FIRST [HELLO]
HELLO
PRINT "HELLO
HELLO
PRINT "HELLO = FIRST [HELLO]
TRUE
```

So FIRST [HELLO] is the same as the word "HELLO. The first item in a one-word list is a word. That is why FIRST :REPLY was used as the first input for MAKE in the command

```
MAKE (FIRST :REPLY) :DRAWLIST
```

Redrawing Pictures

To redraw any picture that has a name, we need the procedure RD, that uses :DRAWLIST as its input.

```
TO RD :DRAWLIST
IF :DRAWLIST = [ ] [STOP]
RECOMMAND FIRST :DRAWLIST
RD BUTFIRST :DRAWLIST
END
```

POWERFUL IDEA

RD is a classic "list-processing" procedure. Its pattern is repeated over and over again in more advanced Logo programming. Similar patterns are used in some of the tool procedures that are described in Chapter 14. This is the pattern:

RD takes a list as its input. First, it checks to see if the list is empty. If it is, RD stops. If the list is not empty, RECOMMAND does something with the *first* element of the list.

Finally, another RD is called, with BUTFIRST *(everything but the first element)* of its original list as input. As each RD calls the next one with BUTFIRST :DRAWLIST as input, the input lists keep getting shorter and shorter. Eventually RD will be given an *empty list*, [ ], as input and it will stop.

RD's subprocedure RECOMMAND carries out each of the commands in the list. RECOMMAND is a lot like COMMAND, except that it uses an input—the first letter on the draw list—rather than getting a command directly from the user.

```
TO RECOMMAND :COM
IF :COM = "F [FORWARD 20]
IF :COM = "B [BACK 20]
IF :COM = "R [RIGHT 30]
IF :COM = "L [LEFT 30]
END
```

When you use RD, you have to give it a list as input. If your drawing was given the name "HOUSE, you would type

```
RD :HOUSE
```

to redraw it, as described in Chapter 3. The reason you type :HOUSE as input for RD rather than "HOUSE or just plain HOUSE, is that RD needs a *list of letters* as input. "HOUSE is the *name* of the particular list that was used to draw the house. :HOUSE, the value of the variable "HOUSE, is the list you want.

HELPER'S HINT

QUICKDRAW offers another example of why it is critical to distinguish between the name of an object and the object itself. One way to help someone understand the relation between the list of drawing commands and the name of that list is to keep track of the process while it is happening. Make the computer print out the value of :DRAWLIST as each new command is added and have it print out the final list whenever a picture is finished. Add this line to QUICKDRAW just after COMMAND:

```
PRINT SENTENCE [:DRAWLIST IS NOW] :DRAWLIST
```

and a similar line to the end of FINISH

```
PRINT (SENTENCE WORD ": FIRST :REPLY [IS NOW] :DRAWLIST)
```

These new command lines don't change the functions of the procedures at all. They are for learning purposes only and can be removed after they have served their purpose.

Another way to implement a procedure like QUICKDRAW is given in Chapter 9 of Harold Abelson's books *Logo for the Apple II* and *Apple Logo*, and in Chapter 10 of Abelson's *TI Logo*. Abelson's INSTANT procedure uses the Logo command DEFINE to create a new *procedure* when the user is finished drawing and has named the picture. This makes it even simpler to use because there is no need for an RD procedure. Each picture can be redrawn just by typing its name. Since I do not discuss the use of the DEFINE command in this book, I have chosen a different method. All the ideas for extending QUICKDRAW given in the next sections of this chapter can be used to extend Abelson's INSTANT procedure as well.

Section 11.3. Improving QUICKDRAW

In this section, I'll show you how to make some improvements to QUICKDRAW. You'll probably think of many more yourself if you work with a younger person who is actually using QUICKDRAW. Let the younger person decide what the program should do. Then see if you can make it happen. Here are a few suggestions.

1. Add new commands to COMMAND and RECOMMAND. For example, if you want to make it possible to draw circles, load the file called "CIRCLES from the LWAL Procedures Disk and add this line to COMMAND:

```
IF :COM = "C [RCIRCLE 10 ADDLETTER :COM]
```

Then add a corresponding line to RECOMMAND:

```
IF :COM = "C [RCIRCLE 10]
```

If you have a color TV or monitor, you'll probably want to add color commands. You might want to use numbers for these.

```
IF :COM = "0 [SETPC 0 ADDLETTER :COM]
IF :COM = "1 [SETPC 1 ADDLETTER :COM]
IF :COM = "2 [SETPC 2 ADDLETTER :COM]
```

Also add corresponding lines to RECOMMAND.

Or add commands to COMMAND and RECOMMAND for PENUP and PENDOWN.

```
IF :COM = "U [PENUP ADDLETTER :COM]
IF :COM = "D [PENDOWN ADDLETTER :COM]
```

2. Make a command to clear the screen and make :DRAWLIST empty again, if you don't like the picture at all. Add this line to COMMAND:

```
IF :COM = "Q [START]
```

START is the procedure in QD that starts everything off.

In this case you don't use ADDLETTER because you're not adding to the list—you're emptying it. You also don't need a line for this in the RECOMMAND procedure. Once you type **Q**, you'll have to start drawing all over again.

3. Make it possible to add an existing drawing (one that already has a name) to the one you're drawing. To do this, you need a new line in the COMMAND procedure:

```
IF :COM = "A [ADDPICTURE]
```

and a new procedure called ADDPICTURE:

```
TO ADDPICTURE
PRINT [WHICH PICTURE DO YOU WANT TO ADD?]
MAKE "PICT FIRST REQUEST
RD THING :PICT
ADDLETTER THING :PICT
END
```

ADDPICTURE needs a little bit of explanation. THING is a Logo command that outputs the *value* associated with a name. When you type in a name, the second line of ADDPICTURE,

```
MAKE "PICT FIRST REQUEST
```

gives the name you type a *new* name—"PICT. RD and ADDLETTER

both need *lists* as inputs. :PICT is a *word* that is the *name* for a list. THING :PICT is the *list* that goes with the name :PICT.

To understand it better, try it out with direct commands. First make a list of commands

```
MAKE "SILLYPICTURE [F F R R F]
MAKE "PICT "SILLYPICTURE
```

This is what happens in the second line of ADDPICTURE. Now type

```
PRINT :PICT
SILLYPICTURE
PRINT THING "SILLYPICTURE
F F R R F
PRINT :SILLYPICTURE
F F R R F
PRINT THING :PICT
F F R R F
```

The last three of these commands should all print the same list of letters since :PICT *is* "SILLYPICTURE and THING "SILLYPICTURE, :SILLYPICTURE, and THING :PICT are all different ways of getting the same Logo object—the list **[F F R R F].**

If you don't completely understand this now, don't worry. Using a name to stand for another name can be quite confusing. Just make sure that ADDPICTURE works properly.

4. The final change I'll suggest is to vary the turtle step and turning angle used in COMMAND and RECOMMAND. This can be done by changing all the command lines that move or turn the turtle and adding some lines to START that set the values of these variables.

```
TO START
MAKE "DISTANCE 20
MAKE "ANGLE 30
MAKE "DRAWLIST [ ]
CLEARSCREEN
END
TO COMMAND
MAKE "COM READKEY
IF :COM = "F [FORWARD :DISTANCE ADDLETTER :COM]
IF :COM = "B [BACK :DISTANCE ADDLETTER :COM]
IF :COM = "R [RIGHT :ANGLE ADDLETTER :COM]
IF :COM = "L [LEFT :ANGLE ADDLETTER :COM]
IF :COM = "C [RCIRCLE :DISTANCE ADDLETTER :COM]
IF :COM = "E [FINISH THROW "TOPLEVEL]
END
```

```
TO RECOMMAND :COM
IF :COM = "F [FORWARD :DISTANCE]
IF :COM = "B [BACK :DISTANCE]
IF :COM = "R [RIGHT :ANGLE]
IF :COM = "L [LEFT :ANGLE]
IF :COM = "C [RCIRCLE :DISTANCE]
END
```

To change these commands, just change the values given in the first two lines of START.

If you want to get just a little fancier, you can add procedures that will allow the user to change these values. To do this, you'll need three new procedures: CHANGE, GETSIZE, and GETANGLE. First, change START to include CHANGE.

```
TO START
MAKE "DISTANCE 20
MAKE "ANGLE 30
MAKE "DRAWLIST [ ]
CLEARSCREEN
CHANGE
CLEARTEXT
END
TO CHANGE
PRINT SENTENCE [THE TURTLE NOW MOVES FORWARD]
    :DISTANCE
PRINT [IF YOU WANT TO CHANGE IT TYPE Y]
IF READLIST = [Y] [GETSIZE]
PRINT [ ]
PRINT SENTENCE [THE TURTLE'S TURNING ANGLE IS NOW]
    :ANGLE
PRINT [IF YOU WANT TO CHANGE IT TYPE Y]
IF READLIST = [Y] [GETANGLE]
END
TO GETSIZE
PRINT [HOW BIG DO YOU WANT THE TURTLE'S]
PRINT [FORWARD STEP SIZE TO BE?]
MAKE "DISTANCE READNUMBER
PRINT SENTENCE [THE FORWARD STEP SIZE IS NOW] :DISTANCE
END
TO GETANGLE
PRINT [HOW BIG DO YOU WANT THE TURTLE'S]
PRINT [TURNING ANGLE TO BE?]
MAKE "ANGLE READNUMBER
PRINT SENTENCE [THE TURNING ANGLE IS NOW] :ANGLE
END
```

READNUMBER is a tool procedure that can be read from a file on the LWAL Procedures Disk or copied from Chapter 14.

Every time you start a drawing, the computer will give you a chance to

change the values of the size and the angle. If you don't want to change them, just type **RETURN**. If you have added a "quit" (**Q**) command as part of COMMAND,

```
IF :COM = "Q [START]
```

you should change it to

```
IF :COM = "Q [CLEARSCREEN MAKE "DRAWLIST [ ] ]
```

so that your distance and angle values are *not* changed when you erase a drawing and start over.

CHAPTER 12

Command	Short Form	Examples With Inputs
PADDLE		FORWARD PADDLE 0 PRINT PADDLE 1
BUTTONP		IF BUTTONP 1 [CLEARSCREEN]
REMAINDER		PRINT REMAINDER 17 3
AND		IF AND (YCOR > 0) (XCOR > 0) [STOP]
XCOR		PRINT XCOR, SETX XCOR + 20
YCOR		PRINT YCOR, SETY YCOR − 20
OUTPUT	OP	OUTPUT "FALSE, OP :LENGTH + 10

LWAL Procedures Disk files used: "READKEY, "CCIRCLE, "DISTANCE, "BOXES, "RACE

New tool procedures used:

Tool Procedure	Examples
DRAWBOX	DRAWBOX 0 0 50 30
INBOX?	IF (INBOX? 0 0 50 30) [STOP]
OUTBOX?	IF (OUTBOX? −20 −20 70 50) [STOP]

12 Animating the Turtle: Building a Racetrack Game

Playing an animated game is one of the most exciting things you can do with a computer. This chapter tells how to make a simple game using an animated turtle.

Animating an object means giving it *motion* so that it seems to come to life, seems to be *like an animal*. To animate the turtle, you make a procedure that keeps it moving until you change its motion by pressing command keys on the keyboard. It's easy to make a procedure that keeps the turtle moving. A lot of the turtle design procedures from earlier chapters will do that. The QUICKDRAW procedure in Chapter 11 shows how to control the turtle from the keyboard. In this chapter, these ideas are combined to create an action game with the turtle.

Drawing with an animated turtle can be a lot of fun all by itself. I'll show you how to control it using game paddles instead of keys on the keyboard. Then we will make the activity into a game by drawing a racetrack and driving the turtle around it. Finally, you can improve the game by adding procedures that check to see if the turtle is on or off the racetrack and whether the race is finished.

Some versions of Logo include objects called sprites which are designed to make animation very easy, in the same way that the turtle is designed to make drawing easy. Sprites can take on many shapes and colors and can move freely around the TV screen. I don't have space in this book to talk about sprites, but I will say that they are very colorful, fast, and exciting. If your ambition as a programmer is to invent videogame programs, versions of Logo that have sprites would help you do that easily.

Section 12.1. Animating the Turtle

Here is the simplest procedure that makes the turtle keep moving. I call it DRIVE because I think of this kind of activity as *driving the turtle*.

```
TO DRIVE :DISTANCE
FORWARD :DISTANCE
DRIVE :DISTANCE
END
```

DRIVE 1 will move the turtle forward one step at a time. DRIVE 10 makes it move ten times as far each time. Since it's taking bigger steps, it also appears to be moving faster.

A turtle that just moves forward isn't very interesting. You need to be able to change its motion as it moves. You can do this by adding a COMMAND procedure that reads the keyboard.

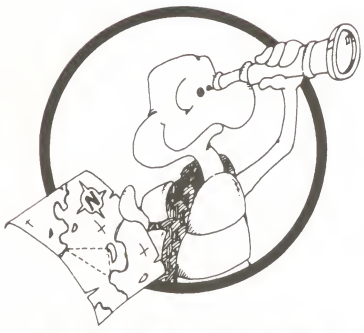
```
TO DRIVE :DISTANCE
FORWARD :DISTANCE
COMMAND
DRIVE :DISTANCE
END
```

COMMAND will check to see if you type a special command key. If so, it carries out that command. Then DRIVE makes the turtle move forward again. If you don't type a command, COMMAND stops without doing anything, and DRIVE keeps moving the turtle forward. COMMAND uses the tool procedure READKEY. Load the "READKEY file from the LWAL Procedures Disk, or copy the procedure from Appendix I:

```
TO COMMAND
MAKE "COM READKEY
IF :COM = "  [STOP]
IF :COM = "R [RIGHT 30 STOP]
IF :COM = "L [LEFT 30 STOP]
END
```

The first line of COMMAND gives the name "COM (short for "command") to the letter typed by the user. The second line of COMMAND is not a mistake. A " symbol with nothing after it is an *empty word*, that is, a word with nothing in it. An empty word is the message sent by READKEY if you don't type anything. The second line of COMMAND stops the procedure immediately if you don't type anything. The STOP command on the end of each of the other lines stops the COMMAND procedure as soon as it finds the letter you typed and carries out the action you have commanded. Without a STOP command on each line, the computer would check more things than it had to and slow the whole process down considerably. This might not matter much when you've only got two commands, but it will make a big difference as you start improving your drive program by adding more commands.

Now you can drive the turtle around the screen without any trouble. If you give DRIVE a large input the turtle will move faster. With a smaller input, it will move slower and give you more control.



EXPLORATION

Try drawing designs using DRIVE. Can you make the turtle draw a circle? How about making it write your name or initials? Draw some other shapes on the screen and see if you can make the turtle move around without touching them. Figure 12.1 shows some shapes that you might be able to draw with DRIVE.

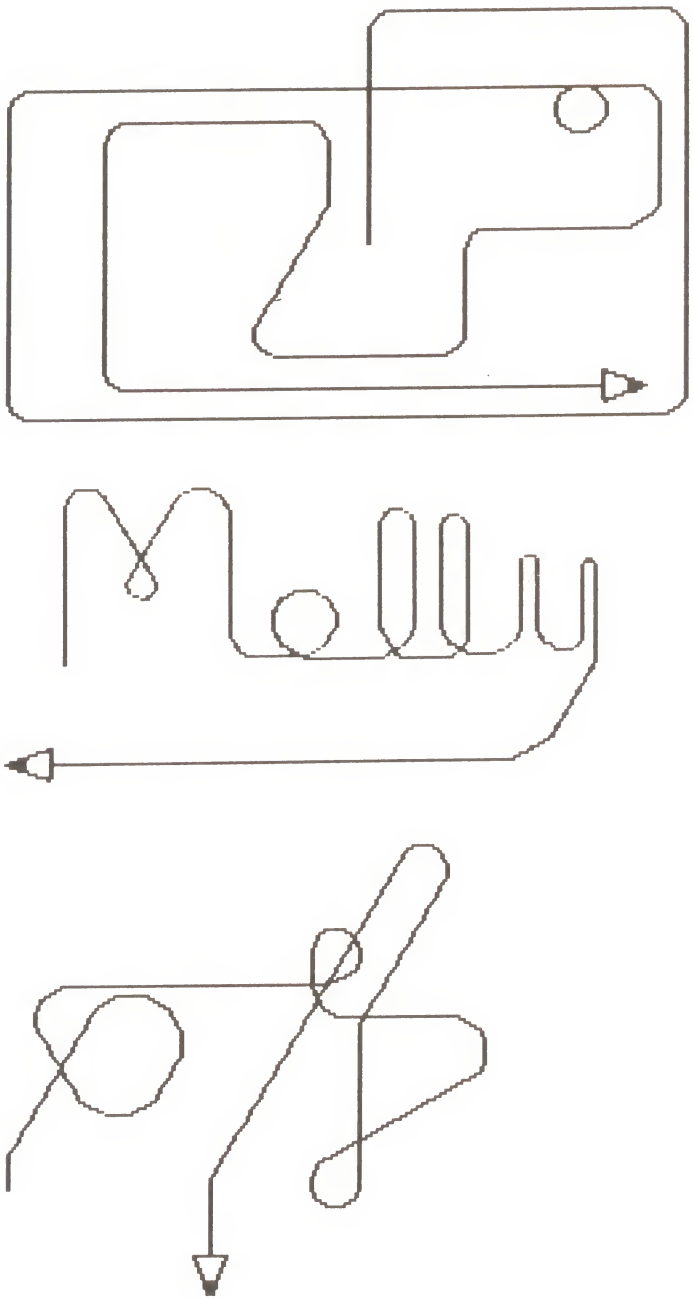


Figure 12.1: Designs made using DRIVE.

I want to mention some subtleties of the COMMAND procedure. First of all, let's think about the concept of an *empty word*. The concept of an *empty object* is very dear to mathematicians but seems terribly abstract to ordinary people. This procedure, however, shows how an empty word can be really important in Logo programming. Empty words or empty lists are often used in conditional commands to tell Logo when a process should be stopped. (In DRIVE, this happens when you don't type anything.)

COMMAND could very well have been introduced without the empty word and without the STOP command on every line.

```
TO COMMAND
MAKE "COM READKEY
IF :COM = "R [RIGHT 30]
IF :COM = "L [LEFT 30]
END
```

In this limited version of COMMAND, with only two letters to be checked, adding one extra line and STOP to the other lines wouldn't speed up the computer very much. But as we expand the project, there will be more and more things to check, and then the speed with which the COMMAND procedure does its job will be critical to the enjoyment of the activity.

This kind of situation always poses a problem for me as a teacher. Do I introduce the simpler idea and wait for the more complex version to be *needed* by someone before I show it, or do I anticipate a future problem and show someone a more complex idea at the beginning to save having to modify a procedure later? There is no obvious answer to this question. When I teach person-to-person, I tend to prefer the first method—introduce a more complex idea only when it is needed. This usually gives someone a better understanding of what is being learned. For example, if you *experience* the problem of the process slowing down as more and more conditional commands are added, you have a better idea of what the computer is really doing as well as a better understanding of the particular control techniques being introduced.

Writing a book forces me to make a decision without knowing what the learner is thinking. In this case I've decided that fast response is so important to the animation process that I should introduce it right from the start. When *you* teach this kind of technique to someone, you might make a different decision and leave out the STOP commands and the empty word entirely, or not introduce them until there is a clear need.

Section 12.2. Improving the Animation

Now that you can move the turtle around freely, you may want to enhance the COMMAND procedure to give you more control over the turtle's motion. It's really easy to add new commands—just decide what letter to use for a particular action and add a new line to COMMAND. Here are some examples that other people have enjoyed.

```
IF :COM = "D [PENDOWN STOP]
IF :COM = "U [PENUP STOP]
IF :COM = "F [MAKE "DISTANCE   :DISTANCE + 1 STOP]
IF :COM = "S [MAKE "DISTANCE   :DISTANCE - 1 STOP]
IF :COM = "C [RCIRCLE :DISTANCE * 5 STOP]
```

You can also add commands for pencolors. One way to do this would be to use numbers.

```
IF :COM = 0 [SETPC 0 STOP]
IF :COM = 1 [SETPC 1 STOP]
```

and so on.

Some people like to add commands like this one to make the turtle do something unexpected.

```
IF :COM = "Z [ZAP STOP]
TO ZAP
PENUP
RIGHT 90
FORWARD 100
LEFT 90
PENDOWN
END
```

This command makes the turtle move instantly to another part of the screen and then keep going in its original direction. Another favorite is to make the turtle stop in its tracks (without stopping the procedure).

```
IF :COM = "H [HALT STOP]
TO HALT
MAKE "DISTANCE 0
END
```

To start the turtle moving again, just press the **F** or the **S** key.

Some people prefer to have all the command keys near each other on the keyboard rather than using the letters to stand for the action. In this case you might want to make a little chart showing the location of your command keys and the function of each one.

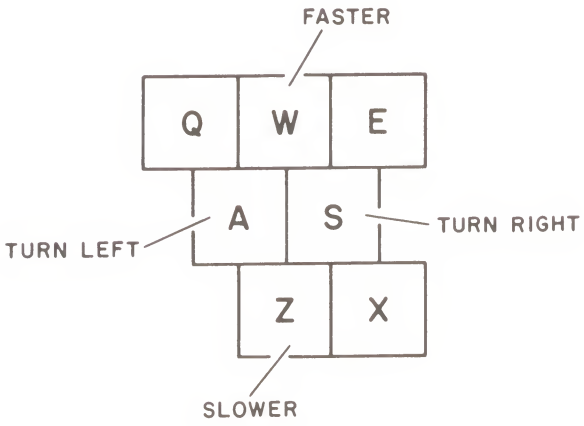


Figure 12.2: A chart of the command keys helps you remember what each one does.

Section 12.3. Animating the Turtle Using Game Paddles

If you have a set of game paddles for your Apple, you can make a DRIVE procedure using game paddle inputs. The primitive Logo commands that let you do this are PADDLE 0, PADDLE 1, BUTTONP 0, and BUTTONP 1. PADDLE 0 and PADDLE 1 both output numbers from 0 to 255 depending on the position of the paddle. BUTTONP 0 and BUTTONP 1 output "TRUE or "FALSE messages depending on whether the button is pressed down. Here is a simple procedure for paddle control:

```
TO PADDLECONTROL
FORWARD ((PADDLE 0) - 128) / 10
RIGHT ((PADDLE 1) - 128) / 5
IF BUTTONP 0 [CHANGECOLOR]
IF BUTTONP 1 [CLEARSCREEN]
END

TO CHANGECOLOR
MAKE "COLOR REMAINDER (:COLOR + 1) 6
SETPC :COLOR
END
```

REMAINDER is a Logo command that outputs the *remainder* left when its first input is divided by its second. Try this:

```
REMAINDER 12 7
5
REMAINDER 3 3
0
REMAINDER 0 7
0
```

You can start the activity with a SETUP procedure.

```
TO SETUP
CLEARSCREEN
MAKE "COLOR 1
SETPC :COLOR
END
```

Instead of DRIVE, use this procedure:

```
TO PDRIVE
PADDLECONTROL
PDRIVE
END
```

If you want more control, you can add a COMMAND procedure with keyboard commands as well.

PADDLECONTROL and CHANGECOLOR use some fancy mathematics. Let's look at some of their commands.

```
FORWARD ((PADDLE 0) - 128) / 10
```

((PADDLE 0) − 128) can have a value of 127 (255 − 128), if paddle 0 is turned all the way to right, and −128 (0 − 128), if paddle 0 is turned all the way to the left. By dividing this number by 10, the forward distance is limited to a range of 12.7 to −12.8. To stop the turtle, adjust the paddle so that it is in the middle.

```
RIGHT ((PADDLE 1) - 128) / 5
```

Dividing ((PADDLE 1) − 128) by 5 limits the right turn angles to between 22.4 degrees and −22.25 degrees. To make the turtle move straight ahead, adjust the paddle so that it is in the middle.

Experiment with different numbers for the divisors in these lines to see what effects they have.

CHANGECOLOR uses some interesting arithmetic too.

```
REMAINDER (:COLOR + 1) 6
```

outputs the remainder of :COLOR + 1 divided by 6. Setting the new pencolor to this value ensures that it will always be between 0 and 5. Every time you change the color by holding down the button on paddle 0, the pencolor number will increase by 1. When :COLOR is 5, the next new value will be the remainder of (5 + 1) divided by 6. The remainder of 6 divided by 6 is 0. The pencolor will now be set to 0 and the numbers will start increasing again.

PDRIVE shows how paddle input commands work. Paddle inputs can also be used in a lot of other projects. For example, you can make a version of SHOOT that uses paddle inputs instead of inputs typed at the keyboard. Or use paddle inputs for a polyspi procedure like the ones in Chapter 8.

```
TO PADDLESPI :SIZE
FORWARD :SIZE
RIGHT PADDLE 0
PADDLESPI :SIZE + ((PADDLE 1) / 20)
END
```

PADDLESPI uses the value output by PADDLE 0 to control the angle and the value output by PADDLE 1 to control the increase in size. This is another way to animate the turtle and keep changing its motion.

Section 12.4. Racing with the Turtle, Part I

The easiest way to use the DRIVE procedure for a racing game is to draw a racetrack and drive the turtle around it. To do this, load the file called "CIRCLES from the LWAL Procedures Disk and draw a circular racetrack. Put the turtle on the track and give the DRIVE command.

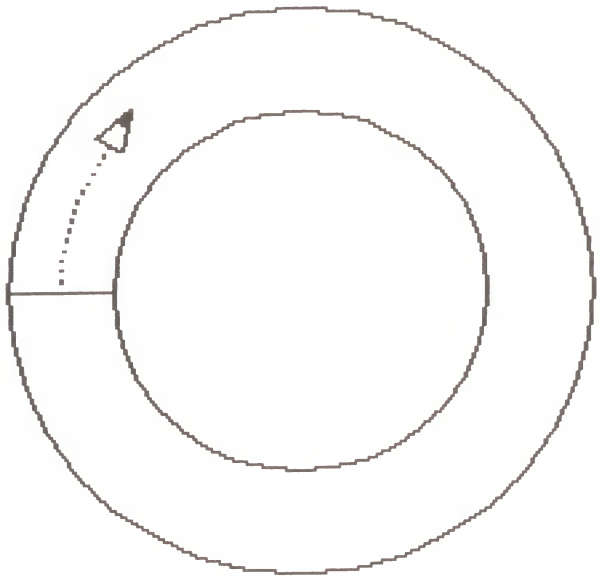


Figure 12.3: The turtle on a circular racetrack.

In this simple game, the computer does not keep track of the turtle's motion. You can drive it anywhere you want. To make more of a game out of it, you can make the turtle stop if it crashes through the wall or goes all the way around the track without crashing.

Since the computer is not checking anything, you can make the track any shape you want, add obstacles to be avoided, design dead ends, etc.

Use your imagination to build an exciting track and then see how well you can do at keeping the turtle from crashing at fast speeds. Or design a maze and see if you can drive the turtle around it without hitting a wall.

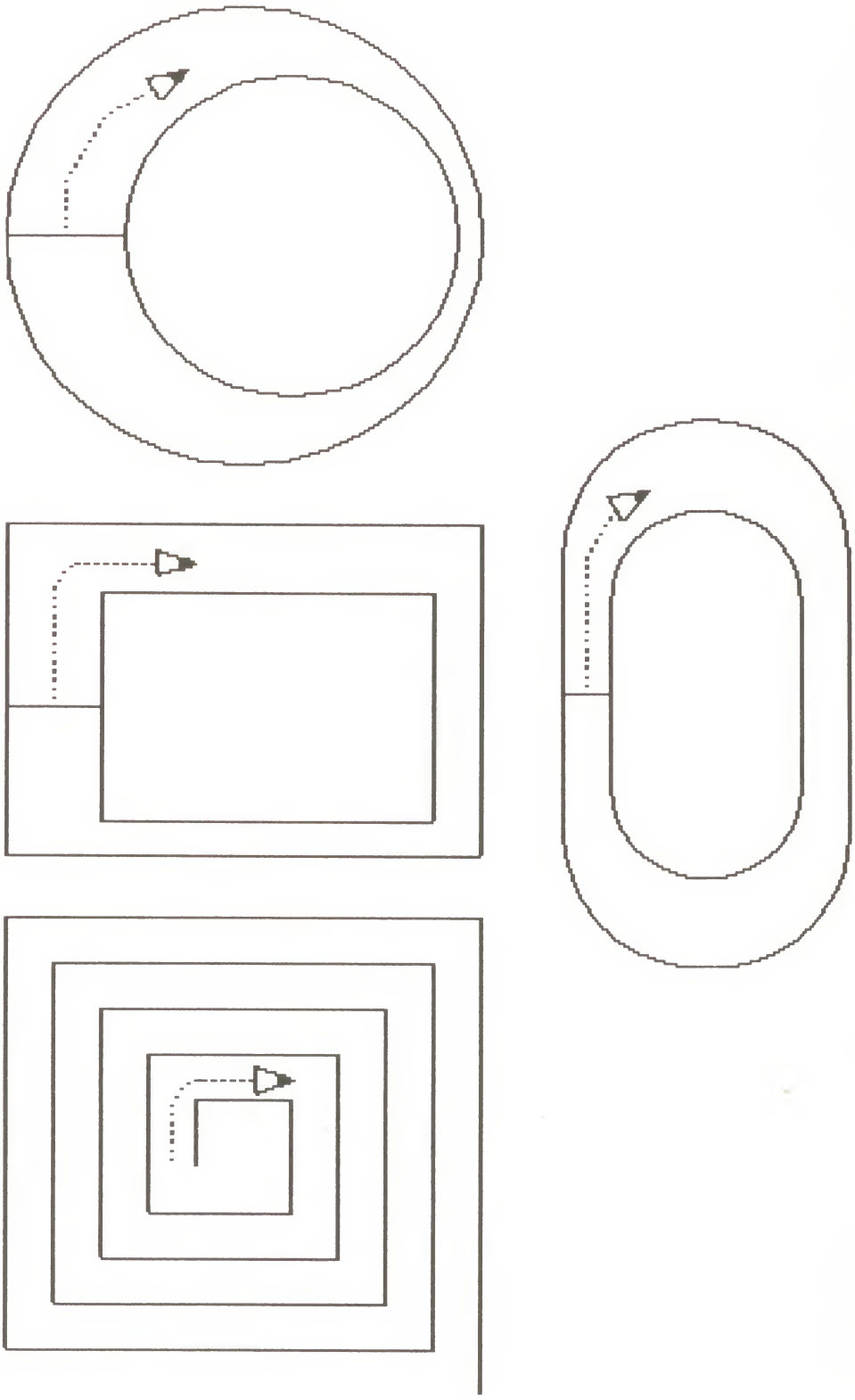


Figure 12.4: A turtle racetrack can be simple or complex.

HELPER'S HINT

This extremely simple-minded racetrack game is amazingly satisfying for many people. The freedom to design a number of different racetracks and the kinetic experience of driving the turtle around them provide enjoyable opportunities for exploring the behavior of the turtle and the creation of geometric shapes.

I have used this kind of DRIVE procedure to create a domain for introductory drawing projects like those in Chapter 6. I provide the DRIVE procedures as tools and let the students have the fun of designing visual environments in which the turtle can play. Eventually, of course, the DRIVE program itself becomes an object of interest, providing a learner with a different kind of introduction to recursion and conditional statements.

The next step—designing a game in which the computer keeps track of the turtle's position—involves a fair amount of data processing and can be a large leap for many people. Don't dismiss the value of the kind of simple activity described in this section.

Section 12.5. Racing with the Turtle, Part II

Now let's make the computer do the work of keeping track of the turtle's progress. A sample RACE game can be read from a file on the LWAL Procedures Disk. Load the "RACE file or copy the RACE procedures from Appendix I. To play the game, type RACE and use the **F**, **S**, **R**, and **L** keys to move the turtle around. I'll explain how the game works in this section and give some ideas for changing or improving it in the next.

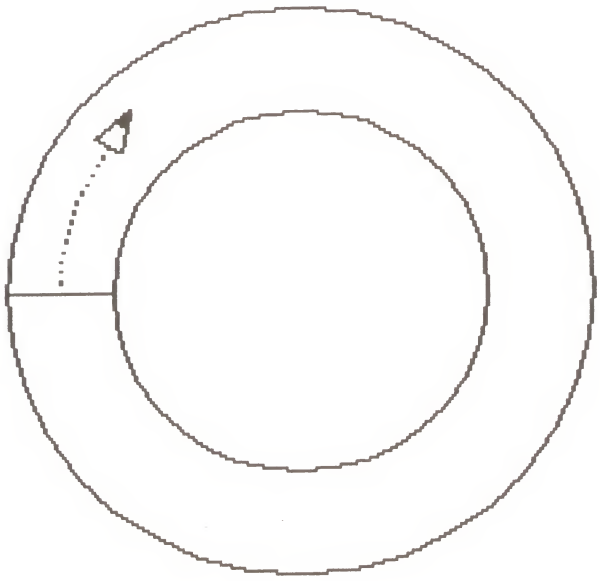
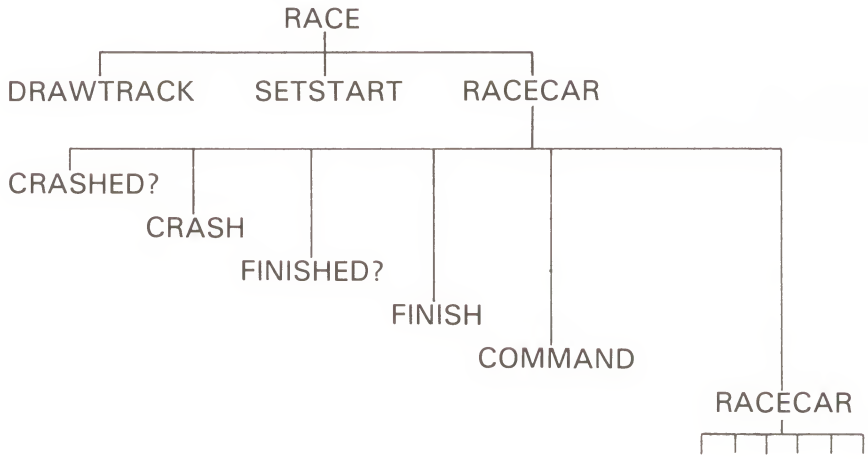


Figure 12.5: The turtle's position after the start of the race.

As the turtle races around its circular track, the computer decides if it has crashed into the wall or crossed the finish line. When the race is over, it tells you your score—the time it took you to complete one lap. The object of the game is to make your score as low as possible while keeping the turtle safely on the track. A procedure called RESTART allows you to start again without redrawing the track.

The RACE procedures make use of three tool procedures that were

also used in the SHOOT game of Chapter 10: CCIRCLE, DISTANCE, and READKEY. You can load them from files on your LWAL Procedures Disk or copy them from Appendix I. Here is the procedure tree for the RACE game:



The superprocedure RACE is very simple:

```
TO RACE
DRAWTRACK
SETSTART
RACECAR 0
END
```

DRAWTRACK and SETSTART create the starting conditions. I'll talk about them later. RACECAR is the procedure that does all the real work of the game.

```
TO RACECAR :TIME
IF CRASHED? [CRASH STOP]
IF FINISHED? [FINISH STOP]
FORWARD :DISTANCE
COMMAND
RACECAR :TIME + 1
END
```

:TIME is a variable that keeps track of the elapsed time. Since the Apple computer doesn't really measure time *directly*, we measure it *indirectly* by increasing RACECAR's input by 1, every time it calls another version of itself.

CRASHED? and FINISHED? are the most interesting new procedures. They are *question procedures* that have the job of checking the turtle's position and deciding whether the game is over. CRASHED? will output the answer "TRUE if the turtle has crossed an edge of the track. If the turtle has *not* hit an edge, CRASHED? outputs "FALSE. Since the outer circle has a radius of 70 and the inner circle a radius of 50, CRASHED? has to output "TRUE if the distance to the center is larger than 70 or smaller than 50.

```
TO CRASHED?
IF (DISTANCE [0 0]) > 70 [OUTPUT "TRUE]
IF (DISTANCE [0 0]) < 50 [OUTPUT "TRUE]
OUTPUT "FALSE
END
```

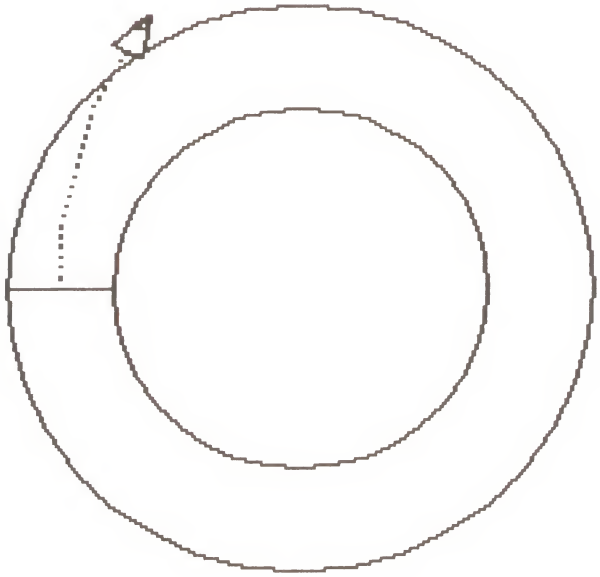
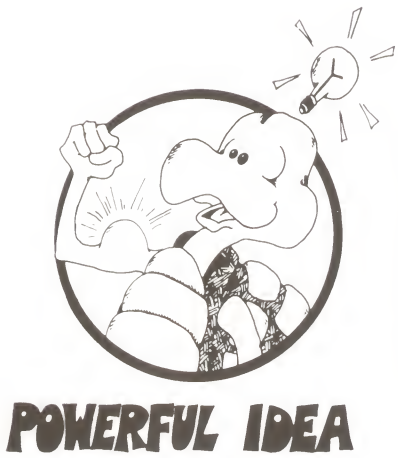


Figure 12.6: The turtle has "crashed" on the edge of the racetrack.



In a game like this it's a good idea to choose the geometry so that the checks are easy for you to figure out and for the computer to calculate. Circles were deliberately chosen for the RACE game because it's so easy to check whether an object is inside or outside of a circle—just check whether the distance from the object to the center of the circle is more or less than the circle's radius. Choosing a shape that's easy to *check* is half the battle in making a game procedure like this work without a lot of blood, sweat, and tears.

If the answer to CRASHED? is "TRUE, the rest of the IF statement is carried out. CRASH prints a message and RACECAR stops.

```
TO CRASH
PRINT [YOU CRASHED INTO THE TRACK WALL!]
END
```

If the answer to CRASHED? is "FALSE, the computer goes on to the next line to find the answer to the FINISHED? question. The way that the computer answers this is a little bit tricky. The finish line is placed along a horizontal line with its Y-coordinate equal to zero. When the turtle crosses the finish line, it moves from a position at which its Y-coordinate was *negative*, as shown in Figure 12.7a, to a *positive* Y-coordinate as shown in Figure 12.7b. The FINISHED? procedure uses this idea to decide whether the race is over.

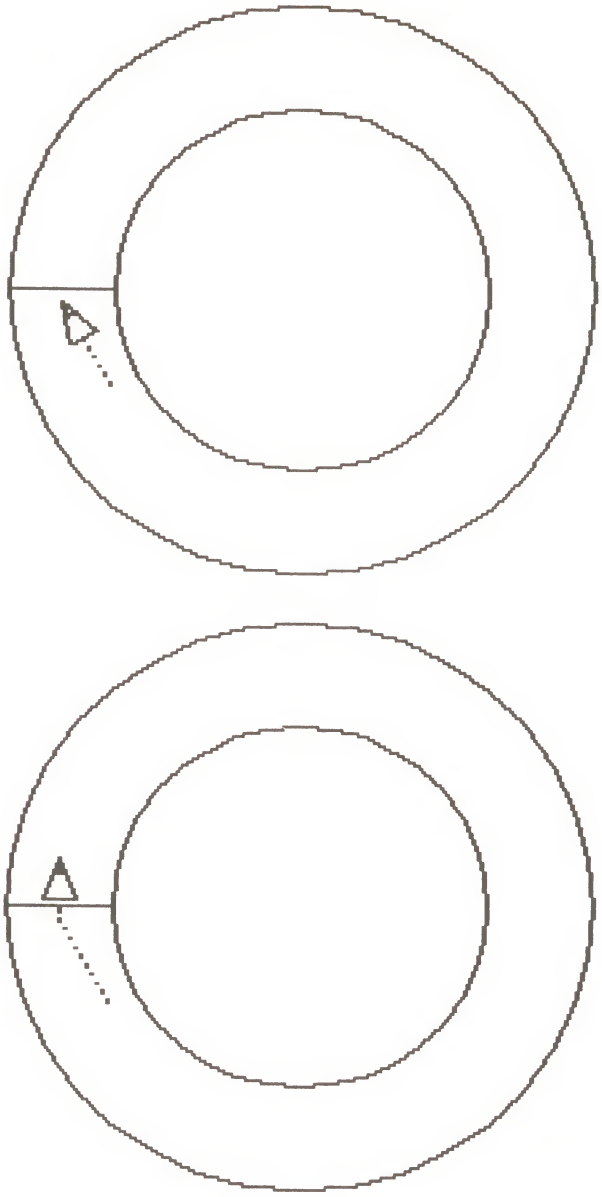


Figure 12.7: The turtle crosses the finish line.

When the turtle crosses the line, *two* things will be true. Its *present* Y-coordinate will be bigger than zero, and its *previous* Y-coordinate will have been less than zero. The computer has to compare where the turtle *is* with where it *was* in order to answer the FINISHED? question. The Logo command YCOR is used to find where the turtle *is*. A variable called "OLDY is used to keep track of where it *was*.

```
TO FINISHED?
IF AND (YCOR > 0) (:OLDY < 0) [OUTPUT "TRUE]
MAKE "OLDY YCOR
OUTPUT "FALSE
END
```

The first line checks to see if the finishing condition is true. The Logo command AND checks to see if *both* of its inputs, (YCOR > 0) and (:OLDY < 0), are true. If they are, FINISHED? outputs "TRUE. If not. Logo makes "OLDY equal to the present YCOR and outputs "FALSE. If the answer to FINISHED? is "TRUE, the FINISH message is printed and the game stops.

```
TO FINISH
PRINT [YOU CROSSED THE FINISH LINE]
PRINT SENTENCE [WITH A TIME OF] :TIME
END
```

PITFALL

People often get confused about the STOP commands in both conditional lines of RACECAR. The STOP commands are there to stop the RACECAR procedure if either FINISHED? or CRASHED? has a true answer. If you leave them out or put STOP commands into the FINISH or CRASH procedures, those procedures will stop but RACECAR won't, even though all of the conditions for ending the race are satisfied.

Another way to make RACECAR stop is to put the Logo command THROW "TOPLEVEL at the end of the FINISH and CRASH procedures. THROW "TOPLEVEL tells the computer to make *everything* stop. It's like applying an emergency brake. It's tempting to use in a case like this, but it's not necessarily a good programming idea, because it could make it harder to improve the game later.

COMMAND, the next-to-last subprocedure of RACECAR, is just about the same as it was before.

```
TO COMMAND
MAKE "COM READKEY
IF :COM = "   [STOP]
IF :COM = "F [MAKE "DISTANCE   :DISTANCE + 5 STOP]
IF :COM = "S [MAKE "DISTANCE   :DISTANCE - 5 STOP]
IF :COM = "R [RIGHT 30 STOP]
IF :COM = "L [LEFT 30 STOP]
END
```

RACECAR's last subprocedure is another RACECAR procedure which keeps the whole process going with an increased time input.

Now let's look back at the first two subprocedures of RACE, DRAWTRACK and SETSTART:

```
TO DRAWTRACK
CLEARSCREEN
HIDETURTLE
CCIRCLE 50
CCIRCLE 70
LEFT 90
PENUP FORWARD 50
```

```
PENDOWN FORWARD 20
PENUP BACK 70
RIGHT 90
END
TO SETSTART
PENUP SETPOS [-60 0]
SETHEADING 0
FORWARD 1 SHOWTURTLE
MAKE "OLDY 1
MAKE "DISTANCE 0
END
```

You should be able to figure out what these procedures do without much difficulty. The two CCIRCLE procedures in DRAWTRACK draw circles that have their centers at the center of the screen. SETSTART moves the turtle to its starting place, makes the starting value of "OLDY bigger than zero, and gives the turtle a starting speed of zero by making "DISTANCE 0.

HELPER'S HINT

This project is deceptively simple. The whole process of tracking the turtle's position can get very complex if any but the simplest shapes are used. That's why this game uses a circular track with the center of both circles at the origin and a horizontal finish line with a Y-coordinate of zero.

In principle, it's possible to have any kind of track and have the computer calculate whether the turtle is on or off it at any time, it would be a tricky problem for a high school or college student to determine the correct formulas for anything more than the simplest kind of track. (I'll give some challenging examples that are just a little harder than this one in the next section.)

Then there's another problem. If the calculations get complicated and the computer has to check a lot of possible positions each time, the process can slow down to the point where it isn't much fun anymore. Those fancy arcade games in which everything happens so fast are programmed in unreadable machine language by professional programmers. In making programs so much easier to understand, Logo must sacrifice speed. Therefore it's necessary to choose the parameters very carefully in a project like this.

The critical role for a teacher is helping the learner focus on a problem he or she can solve. I've seen a lot of people get frustrated to the point of wanting to quit after setting themselves a problem that's just a little too complex (like racing a turtle around a geometrically complex track). If you can help someone learn to simplify a complex problem into a doable one, you've taught them one of life's most important skills! Once you can solve a problem in its simplest form, it may be possible to add some interesting complexities later. I'll suggest some of these in the next section.

Meanwhile, if someone is determined to drive the turtle around complex racetracks and mazes, it's best to encourage them *not* to try to make the computer keep track of things. After enjoying the pleasures that come with exploring complex shapes and motions for a while, he or she may be ready to tackle a "simpler" problem.

Section 12.6. Turtle RACE Variations

Now that you've got the basic idea of how the RACE procedure works, I'll suggest a few frills. You'll probably have many more ideas yourself. I won't tell you everything about how to make them work, just make some suggestions the way I did for changing the SHOOT game in Chapter 10.

1. Provide instructions. Just decide what the printed instructions should be and add an INSTRUCTIONS procedure to RACE. Remember that only a first-time user will need instructions, so it would be nice to let the user choose whether to read them.

2. Change the messages printed by CRASH and FINISH to make them more interesting. This will involve changing those two procedures.

3. Keep track of the fastest time as the game is played. The faster your time the lower your score.

   It would be nice if the computer remembered the lowest score you made every time you played the game. This might make the game more challenging for some players. To do this you need a new variable called "LOWSCORE. Then add a procedure to FINISH after it prints the score.

```
TO COMPARESCORES
IF (:TIME < :LOWSCORE) [MAKE "LOWSCORE :TIME]
PRINT SENTENCE [THE FASTEST TIME FOR THE GAME IS]
    :LOWSCORE
END
```

   There's one more thing you have to do—initialize "LOWSCORE the very first time you play the game. One good way to do this is with a SETSCORE procedure.

```
TO SETSCORE
MAKE "LOWSCORE 500
SAVE "RACE
ERASE "SETSCORE
END
```

   Choose a large number so that the first winning score is bound to be lower. Notice that the procedure saves the "RACE file with the value of "LOWSCORE in it, then erases itself. You only need to use it *once*. If you want to save the *new* low score each time, save "RACE after you are all finished playing the game. If not, every time you read the file, it will start with a "low score" of 500. Remember to use SETSCORE by itself, the first time you play.

4. Make something "interesting" happen when the turtle crashes or finishes. Add an EXPLODE procedure to CRASH or a FLAGWAVING procedure to FINISH. Watch out though. If you make the explosion *too* interesting the player may never want to finish the game properly.

5. Make the track width variable. By making the track narrower you make the game harder. By making it wider, you make the game easier. You can give the user a choice of a hard, medium, or easy game, just as suggested for the SHOOT game. Or you can let the user choose the width by asking a question and using the command MAKE "WIDTH READNUMBER. (READNUMBER is a tool procedure. Load the "READNUMBER file from the LWAL Procedures Disk or copy it from Appendix I.)

   If you vary the width, you have to change the command in CRASHED? that checks to see if the turtle has crossed the outside of the track. You will also have to change the turtle's starting position in SETSTART if you want the turtle to start in the center of the track.

6. Change the shape of the track. This is the hardest variation because it can get very tricky to check whether the turtle is on or off the track.

There are a few fairly simple changes you can make that will add variety to the game. Remember that if you change the procedures that draw the track, you may have to change the point in SETSTART where the turtle starts.

- Track variation #1: Make the two circles have different centers.

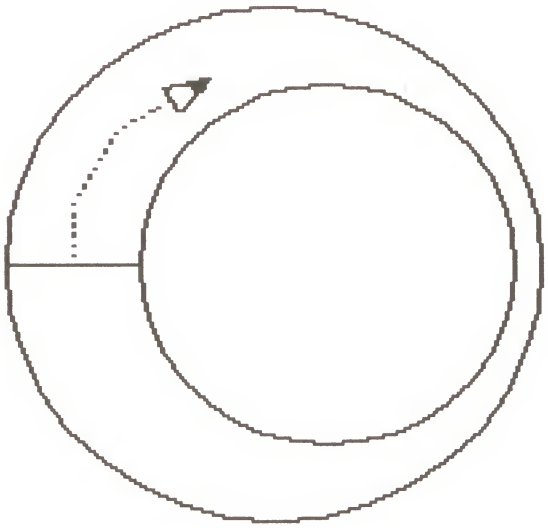


Figure 12.8: A racetrack with uneven width.

You can do this by changing DRAWTRACK so that the two circles start in different places. Then be sure to change CRASHED? so that Logo checks the distance to the center of each circle. Instead of DISTANCE [0 0], use the actual X and Y coordinates of the center of each circle as inputs to DISTANCE.

- Track variation #2: Make the track a rectangular box.

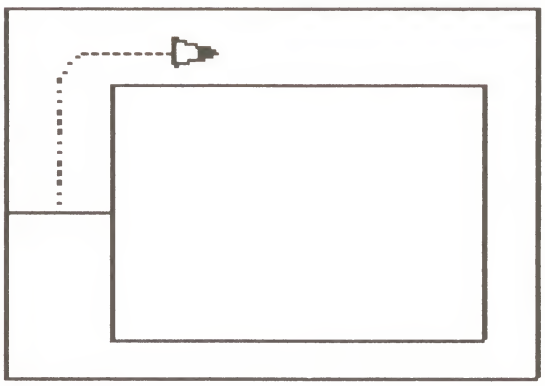


Figure 12.9: A rectangular racetrack.

A set of tool procedures called DRAWBOX, INBOX?, and OUTBOX? can be used to draw rectangular boxes and check whether the turtle is in-

side or outside of a rectangular racetrack. Load them from a file called "BOXES on the LWAL Procedures Disk or copy them from Appendix I. DRAWBOX, INBOX?, and OUTBOX? need four inputs giving the X- and Y-coordinates of diagonally opposite corners of a box. If the opposite corners of the inside box are labeled A and B and those of the outside box are labeled C and D, then the coordinates of those points could be called XA,YA, XB,YB, XC,YC and XD,YD. A new version of DRAWTRACK is needed to create names for all eight variables and use those names to draw two boxes. INBOX? and OUTBOX? use those same values as inputs to the CRASHED? procedure.

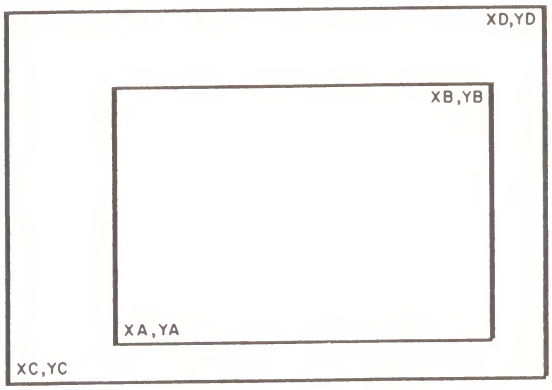


Figure 12.10: A rectangular racetrack is defined by the coordinates of the corners of the track.

```
TO DRAWTRACK
MAKE "XA -40 MAKE "YA -40
MAKE "XB 60 MAKE "YB 80
MAKE "XC -100 MAKE "YC -80
MAKE "XD 90 MAKE "YD 90
DRAWBOX :XA :YA :XB :YB
DRAWBOX :XC :YC :XD :YD
SETPOS SENTENCE :XA 0
PENDOWN SETX :XC
END
TO CRASHED?
IF INBOX? :XA :YA :XB :YB [OUTPUT "TRUE]
IF OUTBOX? :XC :YC :XD :YD [OUTPUT "TRUE]
OUTPUT "FALSE
END
```

The first line of SETSTART will also have to be changed to:

```
PENUP SETX (:XA + :XC) / 2
```

All the other procedures would remain the same.

- Track variation #3. Design an oval track.

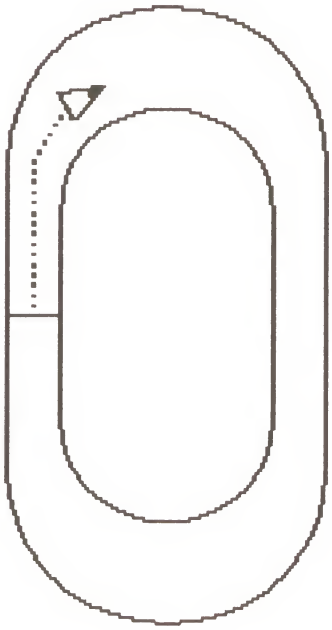


Figure 12.11: An oval track.

When you really understand how rectangular and circular tracks work, you should be able to work out procedures for this one too. This oval shape combines a rectangle with a semicircle, but I won't say any more about it.

HELPER'S HINT

These last few projects provide a really good opportunity to help someone understand how coordinate geometry works. There are few shortcuts. To help someone do these projects you have to be willing to patiently explain X- and Y-coordinates, positive and negative numbers, and the meaning of the concepts "greater than" and "less than" for negative numbers.

I don't pretend that I've given enough information here to explain these things. If you need to know more, a good junior high or high school algebra textbook will probably be helpful, or a math teacher would be delighted to help! Don't be afraid to let somebody explore these concepts for a while and then give up. You could always come back to it in a year or two. Some of these ideas may become much clearer as a learner gains more mathematical sophistication in other ways.

If this is a "cop out," so be it. I never promised that everybody would be able to do everything—even everything in this book.

CHAPTER 13

New commands used: none
LWAL Procedures Disk files used: "POET, "PICKRANDOM, "READNUMBER

New tool procedures used:

Tool Procedure	Examples
PICKRANDOM	PRINT PICKRANDOM [HOUSE BOY CAT MILK]

13 Meet the Poet

Here's a poem I just wrote:

One misty evening
A firefly glitters over the dark meadow
Soft summer twilight

Now here's a poem my computer just wrote:

The empty river
A sunset floats near each dark rain
Delicate still butterfly

Who do you think wrote this one?

Every swirling forest
A firefly murmurs over the wild brook
Misty frosty night

What does it mean to "write a poem"? When I wrote the first poem, I had a certain pattern in mind, and certain words that I thought would fit. I also wanted to capture a special mood and feeling. I was thinking of the sense of beauty and feeling of contentment I had while taking a walk near my home in New Hampshire on a summer evening.

What do you think the computer "had in mind" when it "wrote" the second and third poems? Does a computer have a "mind"? Can it "have something in mind"? When I see that a computer can produce a poem, it makes me stop and think just a little.

You and I know that the computer was just following a procedure. The procedure tells it to select certain types of words according to a fixed pattern. It selects the words from several long lists of different types of words: nouns, verbs, adjectives, etc. I guess there's no need to think of the computer as having been "creative." If anyone was creative, it was probably the programmer who told the computer what pattern to choose, or the person who supplied it with its lists of words.

But wasn't I doing the same thing when I wrote my poem? I was following a procedure, too. The only difference was that I had a much larger choice of patterns and a bigger list of words in my head from which to choose. In fact, every time I write something I'm following some kind of pattern—English grammar is a definite pattern—and trying to choose the best words. Some writers use a thesaurus to help them choose words. How is that different from what the computer was doing? Suppose the computer had been given a much more complex set of patterns, a greater variety of words and types of words, and some rules for choosing among those words and patterns. Then would it have been doing the same thing I was when I wrote my poem?

I would still say, "Of course not!" My poem had *meaning*, it was

based on *feelings* and *experiences* that I remember. No matter how complex a pattern the computer is given, it can't have a meaning or feelings, or remember an experience. Or can it? I do believe that a very clever programmer could make a computer program complicated enough to write poems that would seem so "human" that a poetry expert might have difficulty telling the difference.

This chapter won't go that far, but I will show you some ways that you can use Logo to explore word patterns just as you used it to explore geometric patterns. By trying to make Logo imitate some simple language patterns, you might learn something about *human language patterns*.

Think about what makes a human being human as you work through this chapter. Thinking about a computer as being "almost human" seems like science fiction. But if people start using computers to write poems and stories, we may have trouble telling the difference between the "science" and the "fiction."

Section 13.1. Sentences

In this section we'll explore some sentence patterns that can be created using Logo procedures. First you'll need to load the file "PICKRANDOM from the LWAL Procedures Disk or copy PICKRANDOM and its subprocedure PICK from Appendix I. Now let's look at some very simple sentence patterns and see what kinds of words we're going to need. Here are some simple English sentences that all follow the same pattern:

The rabbit runs.
A girl hops.
A computer computes.

The pattern can be described this way: *article*, *noun*, *verb*. *Articles* are words that help identify nouns, like *a*, *an*, *the*, *this*, or *that*. *Nouns* are words that identify persons, places, or things, like *house*, *book*, *baseball player*, *boy*, and *rabbit*. *Verbs* are action words—things something does—like *runs*, *jumps*, *thinks*, or *computes*.

This Logo program creates simple sentences at random:

```
TO SENTENCE1
OUTPUT (SENTENCE ARTICLE NOUN VERB)
END
```

PITFALL

When you use SENTENCE with *more* than two inputs (as in the procedure SENTENCE1), always put parentheses around SENTENCE and *all* its inputs—not around the inputs by themselves. SENTENCE without parentheses can have only two inputs.

SENTENCE is a Logo command that combines words and lists to make a bigger list. (See Chapter 9 for more information about words and lists.)

ARTICLE, NOUN, and VERB are Logo procedures that choose words randomly from different word lists.

```
TO ARTICLE
OUTPUT PICKRANDOM :ARTICLELIST
END
TO NOUN
OUTPUT PICKRANDOM :NOUNLIST
END
TO VERB
OUTPUT PICKRANDOM :VERBLIST
END
```

PICKRANDOM is a tool procedure that *outputs* a random element from its input list. Now all we need are some lists.

```
MAKE "ARTICLELIST [A THE ONE EACH]
MAKE "NOUNLIST [BOY GIRL RABBIT KANGAROO HOUSE
    COMPUTER BICYCLE REFRIGERATOR]
MAKE "VERBLIST [GOES RUNS HOPS FLIES JUMPS PLAYS
    CATCHES SWIMS SLEEPS KISSES]
```

Let's try it.

```
PRINT SENTENCE1
A COMPUTER SWIMS
```

Again:

```
PRINT SENTENCE1
THE REFRIGERATOR FLIES
```

Well, you can see already that there's more to human language than patterns. Both of those are "proper" English sentences in structure, but they sure don't make sense. Let's forget about making sense for a while and try a few more.

```
REPEAT 5 [PRINT SENTENCE1]
EACH BOY FLIES
THE BICYCLE RUNS
A KANGAROO KISSES
ONE REFRIGERATOR KISSES
THE BICYCLE HOPS
```

Let's mix things up a little more by adding some more kinds of words and trying some different sentence patterns. Remember, we're still not worried about making sense.

```
TO ADJECTIVE
OUTPUT PICKRANDOM :ADJECTIVELIST
END
TO ADVERB
OUTPUT PICKRANDOM :ADVERBLIST
END
```

Now we need some new lists. *Adjectives* are words that describe nouns—that is, they tell what things are like. *Adverbs* describe verbs—they tell how things are done.

```
MAKE "ADJECTIVELIST [NICE QUIET RICH POOR HAPPY SAD UGLY
    BEAUTIFUL FAST SLOW]
MAKE "ADVERBLIST [QUICKLY SLOWLY HAPPILY SADLY
    PRECISELY QUIETLY LOUDLY SOFTLY]
```

Of course you can add your own words to these lists. I'm sure yours will be much more interesting than mine. Choosing *interesting* words for something like this is very hard for me.

Now let's invent some sentence patterns. Try some that you think will be proper English and others that you're not sure about. Even improper sentence patterns might be fun to try. With five types of words, there are dozens of possible sentence patterns if you use some words more than once and sometimes connect two phrases together. Here are a few ideas for patterns. I'm sure you can think of a lot more. Try to make at least a dozen.

```
TO SENTENCE2
OUTPUT (SENTENCE ARTICLE ADJECTIVE NOUN VERB ADVERB)
END
PRINT SENTENCE2
ONE QUIET RABBIT CATCHES SLOWLY
```

Is it a proper sentence if we reverse it?

```
TO SENTENCE3
OUTPUT (SENTENCE ADVERB VERB NOUN ADJECTIVE ARTICLE)
END
PRINT SENTENCE3
PRECISELY SLEEPS RABBIT SAD ONE
```

How about putting the procedures in alphabetical order?

```
TO SENTENCE4
OUTPUT (SENTENCE ADJECTIVE ADVERB ARTICLE NOUN VERB)
END
PRINT SENTENCE4
NICE PRECISELY EACH REFRIGERATOR KISSES
```

Or in a random order?

```
TO SENTENCE5
OUTUT (SENTENCE ARTICLE ADVERB NOUN ADJECTIVE VERB)
END
PRINT SENTENCE5
A SLOWLY GIRL BEAUTIFUL PLAYS
```

How about using more than one of the same kind of word?

```
TO SENTENCE6
OUTPUT (SENTENCE ADVERB ARTICLE ADJECTIVE ADJECTIVE
    ADJECTIVE NOUN VERB)
END
```

```
PRINT SENTENCE6
SADLY ONE QUIET POOR SLOW GIRL PLAYS
```

Here's a neat trick. SENTENCE2 and SENTENCE3 are the reverse of each other. What would it be like to *combine* them?

```
TO SENTENCE7
OUTPUT (SENTENCE SENTENCE2 [AND] SENTENCE3 )
END
PRINT SENTENCE7
THE RICH GIRL KISSES PRECISELY AND SOFTLY CATCHES BOY
    NICE EACH
```

Now that almost sounds like a *poem*!

Suppose you make twelve different sentence patterns. Here's a procedure that will choose *randomly* from among the twelve patterns. (The first line of the procedure adds 1 to RANDOM 12 because RANDOM 12 outputs a number between 0 and 11 and the procedure needs a number between 1 and 12.)

```
TO SENTENCES
MAKE "NUMBER (1 + RANDOM 12)
IF :NUMBER = 1 [PRINT SENTENCE1]
IF :NUMBER = 2 [PRINT SENTENCE2]
.
.
.
IF :NUMBER = 12 [PRINT SENTENCE12]
SENTENCES
END
```

```
SENTENCES
SLOWLY JUMPS GIRL SLOW THE
EACH QUIET COMPUTER PLAYS LOUDLY
ONE RICH RABBIT KISSES PRECISELY
HAPPILY THE FAST SLOW QUIET BOY SLEEPS
EACH SLOW KANGAROO FLIES HAPPILY
A HOUSE SWIMS
ONE BEAUTIFUL RABBIT HOPS SOFTLY
THE SLOWLY GIRL HAPPY SWIMS
THE RICH REFRIGERATOR GOES HAPPILY AND PRECISELY FLIES
    GIRL QUIET A
SAD QUIETLY A RABBIT FLIES
.
.
.
```

If any of these sentence patterns really offend you, take them out of the SENTENCES procedure and replace them by ones you like. See if you can decide which sentences are proper English and which ones are improper. Which ones are proper but boring? Which are improper but interesting? Make lists of proper and improper patterns, interesting and uninteresting

patterns. Compare your lists with someone else's to see if you have different ideas.

Section 13.2. Making Sentences Make Sense

Making *everything* random in a group of sentences may be fun for a while, but I get bored rather quickly. How can we create sentences that make more sense? One way is to choose *sets* of words that go together. Select a whole bunch of nouns, verbs, adjectives, and other types of words that all fit together in some way. For example, make them all about sports, animals, nature, science fiction. If you choose interesting sets of words, your sentences will start to get more interesting.

I decided to make a collection of nature words for the POET program. I picked some of them to make a new wordlist for sentences.

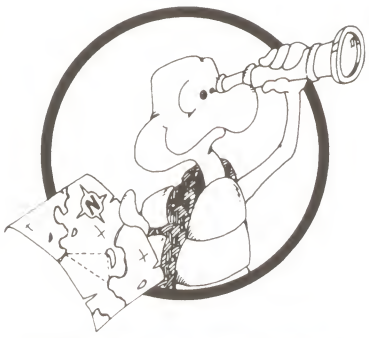
```
MAKE "NOUNLIST [WATERFALL RIVER BREEZE MOON RAIN WIND
    SEA MORNING SNOW LAKE SUNSET SHADOW PINE LEAF
  GLITTER DAWN FOREST]
MAKE "VERBLIST [SHAKES DRIFTS [IS HIDDEN] SLEEPS CREEPS
    MURMURS FLIES FLUTTERS [HAS FALLEN] ]
MAKE "ADJECTIVELIST [AUTUMN HIDDEN BUBBLING BOILING
    SWIRLING GREEN BITTER MISTY SILENT EMPTY]
MAKE "ADVERBLIST [QUICKLY SLOWLY HAPPILY SADLY QUIETLY
    LOUDLY SOFTLY]
MAKE "ARTICLELIST [A THE EACH EVERY]
```

Notice that some of these lists have short lists—phrases like [IS HIDDEN]—inside them.

Now let's try SENTENCES.

```
SENTENCES
LOUDLY THE SWIRLING MISTY SWIRLING SEA SLEEPS
SOFTLY THE BUBBLING EMPTY AUTUMN RAIN MURMURS
SLOWLY DRIFTS RIVER AUTUMN EVERY
ONE QUIETLY PINE GREEN CREEPS
HAPPILY DRIFTS MORNING BUBBLING EACH
EACH BITTER LAKE SHAKES QUICKLY
EVERY BITTER SUNSET DRIFTS QUIETLY AND SLOWLY HAS
    FALLEN WIND BITTER ONE
QUIETLY THE HIDDEN HIDDEN HIDDEN RAIN FLUTTERS
SWIRLING QUICKLY THE FOREST DRIFTS
EVERY BUBBLING SHADOW SLEEPS HAPPILY
```

These sentences are made from the same set of sentence patterns we had before. But because the words go well together, it almost seems like poetry. And if we choose our sentence patterns carefully, maybe it will *be* poetry.



EXPLORATION

Before going on to look at the poetry procedures, try to choose some interesting sets of words of your own. It might even be more fun to do this with someone else. When brainstorming, two or three heads usually *are* better than one. Use words that relate to one of your special interests—animals, sports, computers, horror movies, people, politics, astronomy, . . . whatever.

Section 13.3. POET

POET is created by choosing interesting words *and* interesting sentence patterns. The complete set of POET procedures and wordlists can be copied from Appendix I or loaded from a file called "POET on the LWAL Procedures Disk.

```
TO POET
PRINT LINE1
PRINT LINE2
PRINT LINE3
END
```

Now, let's try POET.

```
POET
EACH LIMPID POND
ONE BIRD RACES OVER THE FROSTY FIR
WILD BLUE MOON
```

Can you detect the pattern?

```
TO LINE1
OUTPUT (SENTENCE ARTICLE ADJECTIVE NOUN)
END
TO LINE2
OUTPUT (SENTENCE ARTICLE NOUN VERB PREPOSITION ARTICLE
    ADJECTIVE NOUN)
END
TO LINE3
OUTPUT (SENTENCE ADJECTIVE ADJECTIVE NOUN)
END
```

The only new word procedure is PREPOSITION, which needs a list of prepositions to go with it. *Prepositions* are words that indicate relationship, location, or direction.

```
TO PREPOSITION
OUTPUT PICKRANDOM :PREPOSITIONLIST
END
MAKE "PREPOSITIONLIST [ON IN OFF [OUT OF] UNDER OVER NEAR
    BENEATH OVER AROUND BELOW ABOVE]
```

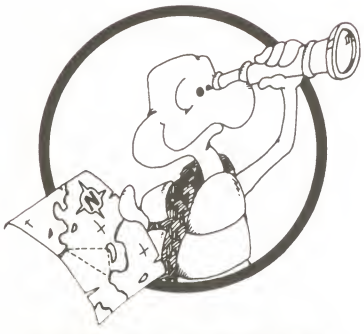
The noun, verb, and adjective lists used by POET are much longer than the ones I used in Section 13.2. If you want to, you can copy the entire set of lists from Appendix I.

The POET file also contains a superprocedure called POEMS that asks how many poems you want and prints that number of poems for you.

```
TO POEMS
CLEARTEXT
PRINT [HOW MANY POEMS DO YOU WANT?]
MAKE "N READNUMBER
CLEARTEXT PRINT [ ] PRINT [ ]
PRINT SENTENCE :N [POEMS BY THE LOGO POET]
PRINT [ ]
REPEAT :N [POET PRINT [ ]]
END
```

POEMS uses READNUMBER, a tool procedure. Load the file called "READNUMBER from the LWAL Procedures Disk or copy it from Appendix I.

```
POEMS
HOW MANY POEMS DO YOU WANT?
3
3 POEMS BY THE LOGO POET
EVERY BOILING DUST
EVERY RIVER HAS STOPPED ON THE BLUE NIGHT
MISTY SPARKLING CROW

ONE EMPTY MORNING
EVERY CROW MURMURS OVER ONE BLUE SNOW
BOILING COLD SHADOW

THE FROSTY RAVEN
THE LEAF HAS PASSED ON THE AZURE NIGHT
BUBBLING LIMPID MORNING
```



EXPLORATION

Now see if you can make up a poetry procedure of your own. Try different variations on the lines, choose different words, or use different numbers of lines. One interesting way to make up POET procedures is to study some real poems that you enjoy reading and try to figure out their patterns.

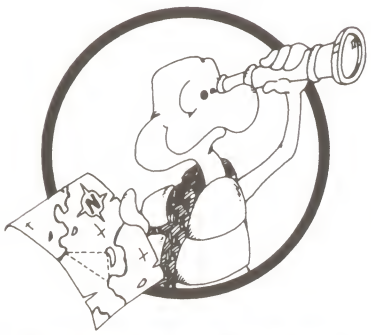
By the way, who would you say is the *author* of the poems written by POET? Is it the computer, or is it the person who programmed the computer? Or is it the real poet whose idea was used as a model for the POET procedure? Perhaps these poems have three authors: the poet, the programmer, and the computer. What do you think?

Section 13.4. More Explorations with Language

In this section I'll suggest a few more things you might try using the ideas I've shown you in this chapter.

Take a famous quotation and use that as a pattern. Keep most of the line so that your new lines sound enough like the original to be funny. Here's a sample from Shakespeare's *Hamlet*:

"To be or not to be, that is the question."



EXPLORATION

Here's a Logo Procedure to make silly sentences out of this:

```
TO HAMLET
MAKE "VERB1 VERB
OUTPUT (SENTENCE [TO] :VERB1 [OR NOT TO] :VERB1 [THAT IS
    THE] NOUN)
END
```

Now try it.

```
PRINT HAMLET
TO SWIM OR NOT TO SWIM THAT IS THE BUTTERFLY
```

The first line of HAMLET gave the name "VERB1 to the output from the VERB procedure so that we could use the same word twice. If we had used the VERB procedure directly in the sentence, as in this version,

```
TO HAMLET1
OUTPUT (SENTENCE [TO] VERB [OR NOT TO] VERB [THAT IS THE]
    NOUN)
END
```

we would get this kind of result:

```
PRINT HAMLET1
TO MURMUR OR NOT TO JUMP THAT IS THE REFRIGERATOR
```

Use a short poem in the same way. Here's a famous poem by Carl Sandburg:

> The fog comes
> on little cat feet.
> It sits looking
> over harbor and city
> on silent haunches
> and then moves on.

First you have to decide which words to keep the same and which to let the computer choose. This procedure might work:

```
TO FOG
PRINT (SENTENCE [THE] NOUN [COMES])
PRINT (SENTENCE [ON] ADJECTIVE ANIMAL [FEET.])
PRINT [ ]
PRINT (SENTENCE [IT SITS] VERBING)
PRINT (SENTENCE [OVER] NOUN [AND] NOUN)
PRINT (SENTENCE [ON] ADJECTIVE [HAUNCHES])
PRINT (SENTENCE [AND THEN] VERB [ON.])
END
```

VERBING ouputs a verb with an *ing* ending, and ANIMAL outputs an animal name. Here's one result:

```
FOG
THE BASEBALL COMES
ON BEAUTIFUL ROOSTER FEET.

IT SITS SINGING
OVER BICYCLE AND HOSPITAL
ON BOILING HAUNCHES
AND THEN FLUTTERS ON.
```

You can probably do better with more interesting choices of words.

To carry this silliness to the limit, take an entire short story or a folk tale like *Little Red Riding Hood*. Select certain key words for the computer to choose and leave all the others the same. Part of the story might come out like this:

```
WHY GRANDMOTHER, WHAT JUICY FRISBEES YOU HAVE!
THE BETTER TO DRIVE YOU WITH MY DEAR.
```

If this kind of exploration with language seems silly to you, make it as serious as you like by choosing serious patterns and selecting the best words you can to fit them. The limit of this kind of activity is really your own patience and creativity—and the memory limits of your computer.

Some computer scientists with big computers are programming them to turn out poems, mystery stories, and other kinds of "literary works." Will there ever be a time when you won't be able to tell the difference between something written by a computer and something written by a person? Someday you may have a chance to try to answer that question for yourself.

CHAPTER 14

Command	Short Form	Examples With Inputs
PEN		PRINT PEN
SHOWNP		PRINT SHOWNP
SETPEN		SETPEN "TRUE
OR		IF OR (XCOR > 0) (YCOR < 0) [STOP]
SQRT		PRINT SQRT 100
KEYP		IF KEYP OUTPUT READCHAR
READCHAR	RC	MAKE "KEY READCHAR
COUNT		PRINT COUNT [A B C D]
NOT		IF NOT (XCOR > 0) [STOP]
NUMBERP		IF (NOT NUMBERP :ANSWER) [OUTPUT READNUMBER]
.DEPOSIT		.DEPOSIT 53008 7
CHAR		PRINT CHAR 17

LWAL Procedures Disk files used: "CIRCLES, "CCIRCLE, "BOXES, "DISTANCE, "READKEY, "PICKRANDOM, "READNUMBER, "PRINTSCREEN.S, "PRINTSCREEN.G

New tool procedures used:

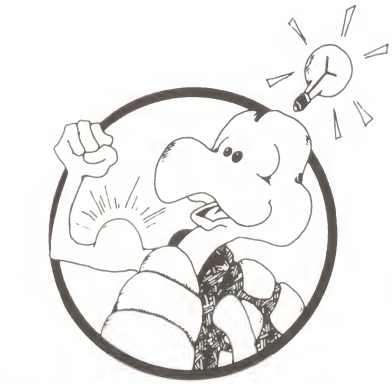
Tool Procedure	Examples
PICK	PRINT PICK 3 [A B C D]

14 How the Special Tool Procedures Work

This entire chapter should be labeled "helper's hint." However, anyone who has read the entire book this far is more than ready to be a "helper" and should be able to understand most of this chapter fairly easily. Some of the tool procedures are quite simple compared to some of the complex projects of the last few chapters.

If you *haven't* read the whole book and just want to know how one particular tool procedure works, you can find out by reading the particular section you want very carefully. But watch out! Not everything in this chapter is going to be easy.

Why *tool procedures*? Why single out these special few Logo procedures, call them tools, and not explain how they work until the end of the book? Why not explain them when they are first introduced? Well, certain procedures seem to make *other things* much easier. These are the ones I think of as tools. People should be able to use tools to build and learn with without having to construct those tools themselves. You wouldn't want to have to know how to *make* a hammer before you learned to *use* one, would you? I feel the same way about Logo tools.



Tool procedures are one of the most important ways that Logo facilitates learning. As a teacher, I can use Logo to create different kinds of learning environments for different students. A procedure that I write can be used by a learner just as if it were another command in the language. A teacher writing Logo procedures for students is creating a language for learning, just as an artist creating turtle drawing procedures might be creating a language for artistic expression. And the language that a teacher creates merges into a learner's own, personally created language as the learner becomes able to understand procedures that were originally presented as "primitives."

Let me give an example. The first tools used were circle procedures, way back in Chapter 2. They were used to create designs before procedures were even introduced. Some teachers prefer to show students how to make the turtle draw circles and how to write procedures *before* they are able to make designs with circles. I prefer to do things the other way around. I give people circle procedures at the beginning. Long before they know how to write procedures with inputs or understand the geometry of circles, people can make exciting circle designs and gain a sense of control over the computer. Later, I can explain some fine points of Logo programming and teach a little geometry that probably would have been too complicated to understand at first.

There's also another reason for having tool procedures. When the same little procedure is used over and over again—while one is trying to accomplish a number of different projects—it's nice to save that procedure separately so that it's handy when you need it again. I hope you have been (or will be) making and saving some of your own tools as you go along.

So you might ask, why bother to explain these little procedures at all? Why not let people figure them out for themselves or just use them without understanding them? The main reason is that it gives me an opportunity for one last set of explanations, one more chance to show you some of the fine points of Logo. These fine points will help you go on to create a lot more projects and keep learning with Logo on your own. So here goes. . . .

Section 14.1. Circles and Arcs

The circle and arc procedures RCIRCLE, LCIRCLE, RARC, and LARC were introduced in Chapter 2. They are built from the subprocedures RCP and LCP. Each of these procedures requires one input, the *radius* of the circle being drawn. Right and left circle procedures are the same except for the direction of turn.

Any Logo "circle" is really a polygon. In this case I decided that a 36-sided polygon was a close visual approximation of a circle. The basic building block, RCP or LCP, turns the turtle 10 degrees so that the total turn for a full circle is 360 degrees.

```
TO RCP :R
RIGHT 5
FORWARD :R * 3.14159 / 18
RIGHT 5
END
```

There are two interesting questions about this procedure. Where did the formula :R * 3.14159 / 18 come from? And why did I make the turtle turn RIGHT 5 *twice* instead of RIGHT 10 *once*?

The number 3.14159 is an approximation of the geometric constant π (pi), the ratio between the circumference of a circle and its diameter. In Figure 14.1, the circumference is labeled C, the diameter D, and the radius, R. The length of 1/36 of the circumference is labeled S; this is the turtle step size needed for a 36-sided polygon.

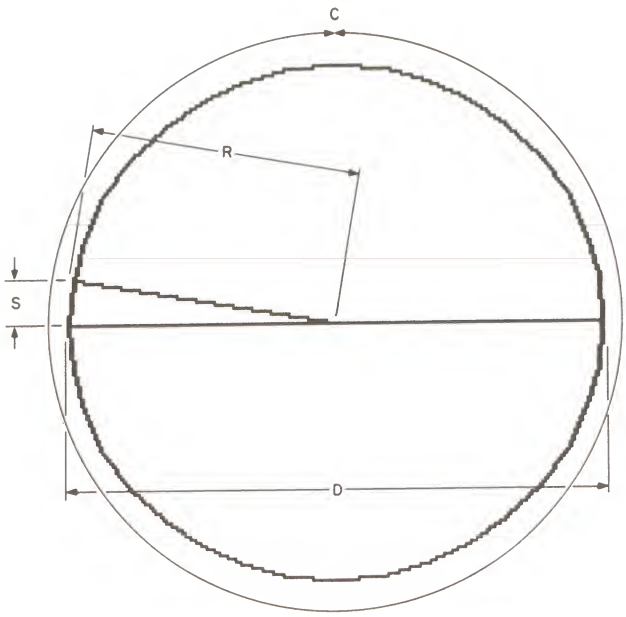


Figure 14.1: Each "right-circle-piece" is 10/360 or 1/36 of the entire circumference.

The mathematical equation relating the circumference to the diameter is:

$C = 3.14159 \times D$

If you use twice the radius in place of the diameter, the equation is

$C = 3.14159 \times R \times 2$

To find the length of 1/36 of the circumference, divide both sides of the equation by 36.

$S = C / 36 = 3.14159 \times R \times 2 / 36$

If you replace 2/36 by 1/18, you will have the formula I used for the forward step in RCP and LCP.

$S = 3.14159 \times R / 18$

Written as a Logo command, this becomes

```
FORWARD :R * (3.14159) / 18
```

I left the formula expressed as :R * 3.14159 / 18 instead of :R * 0.17453 so that people who know about π can see where the expression came from.

RCP uses RIGHT 5 twice rather than RIGHT 10 once so that the center of the "circle" is located at a 90 degree angle from the direction in which the turtle started. If I had used RIGHT 10, the center of the circle would be slightly *above* the point at which the turtle started. Since the difference is

very slight for a 36-sided polygon, I will illustrate what happens with a 6-sided polygon. Compare the results of these two turtle commands:

```
REPEAT 6 [FORWARD 50 RIGHT 60]
```

and

```
REPEAT 6 [RIGHT 30 FORWARD 50 RIGHT 30]
```

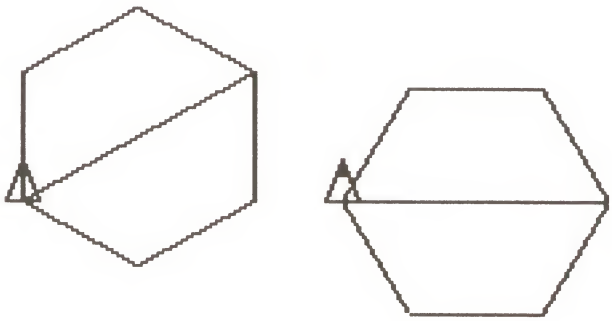


Figure 14.2: Turning before and after each forward step keeps a 6-sided polygon level.

The actual difference for a 36-sided polygon is much harder to detect, but even such a slight difference would make it difficult to complete many of the circle and arc designs suggested in this book. Compare:

```
REPEAT 36 [FORWARD 10 RIGHT 10]
```

with

```
REPEAT 36 [RIGHT 5 FORWARD 10 RIGHT 5]
```

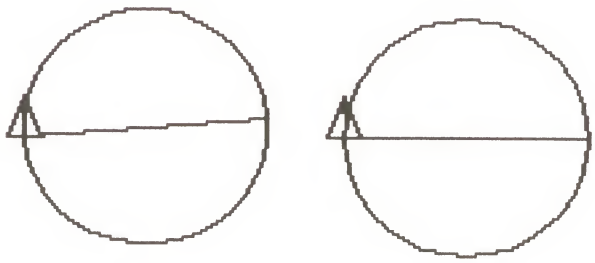


Figure 14.3: With 36-sided polygons, the effect is similar but harder to notice.

RARC and LARC each make quarter-circle arcs by repeating RCP 9 times. Quarter circles are useful for many designs, including rounding the corner of a box and building semicircles. You can make other arcs by repeating RCP or LCP more or less than 9 times.

Section 14.2. CCIRCLE

CCIRCLE draws a circle which starts at its own *center*. It is used for activities where the center is determined in advance and it is important to know the distance from that center to the edge of the circle—for example, in the target game of Chapter 10 or the racetrack game of Chapter 12.

CCIRCLE also uses the subprocedure RCP repeated 36 times. Here is the main part of the procedure:

```
HIDETURTLE
PENUP FORWARD :R
RIGHT 90
PENDOWN REPEAT 36 [RCP :R]
LEFT 90
PENUP BACK 90
```

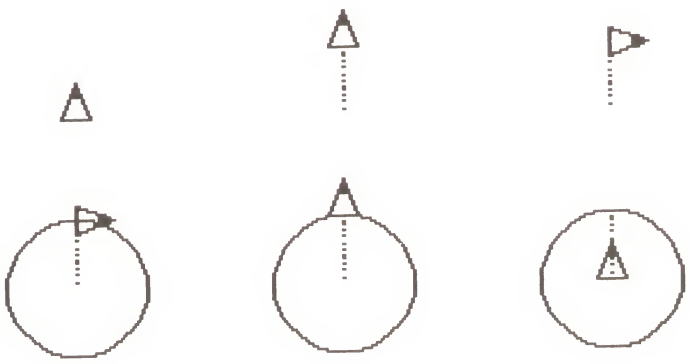
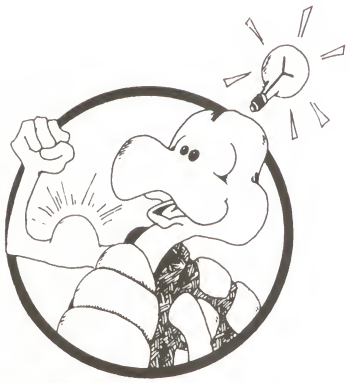


Figure 14.4: CCIRCLE creates a centered circle.

There is one fine point in the CCIRCLE procedure. If the turtle's pen was down at the start, CCIRCLE puts it back down when it finishes. If the turtle was visible originally, it is shown again at the end. This is accomplished with the commands PEN and SHOWNP, which I have not yet mentioned. PEN outputs "TRUE when the turtle's pen is down and "FALSE when the pen is up. SHOWNP outputs "TRUE when the turtle is visible and "FALSE if it is hidden. Now you should be able to follow the entire CCIRCLE procedure.

```
TO CCIRCLE :R
MAKE "PEN? PEN
MAKE "SHOWN? SHOWNP
HIDETURTLE
PENUP FORWARD :R
RIGHT 90
PENDOWN REPEAT 36 [RCP :R]
LEFT 90
PENUP BACK :R
SETPEN :PEN?
IF :SHOWN? [SHOWTURTLE]
END
```

The first two lines of CCIRCLE gives the names "PEN? and "SHOWN? to the values output by PEN and, SHOWNP. The last two lines use these values. If :PEN? is "TRUE, the pen will be put back down. If :SHOWN? is "TRUE, the turtle will be shown again. Otherwise the pen would be left up and the turtle would remain hidden.



This is an example of a procedure that returns the turtle to its original state—not just to its original position and heading, but to its "pen state" and its "shown state" as well. It illustrates an important idea in computer programming: keep track of all the starting conditions of a system so that you can restore those conditions once a particular process is complete.

Section 14.3. Boxes

The box tool procedures are designed to be used when it is important to know whether the turtle is inside or outside of a box—as in the rectangular racetrack variation of RACE in Chapter 12. In this case we use commands like SETPOS, SETX, and SETY to draw the box in terms of its *coordinates* and use the same coordinates to check whether the turtle is in or out of the box. These procedures are less important than circle procedures, but since I haven't used X and Y coordinates in the rest of the book very much, I thought they would be worth discussing here.

To use the procedure DRAWBOX, INBOX?, and OUTBOX?, you have to know the X and Y coordinates of two points on the box. The coordinates of the lower left-hand corner of the box are called X1,Y1. The coordinates of the upper right-hand corner are called X2,Y2. The box can be anywhere on the screen, and the coordinates can be positive or negative. Logo commands SETPOS, SETX, and SETY are used to draw the box. With these commands, the turtle moves directly from where it is to the point named without changing anything else. Its heading is totally irrelevant and stays just as it was. DRAWBOX ends by returning the turtle to its original position with its pen up.

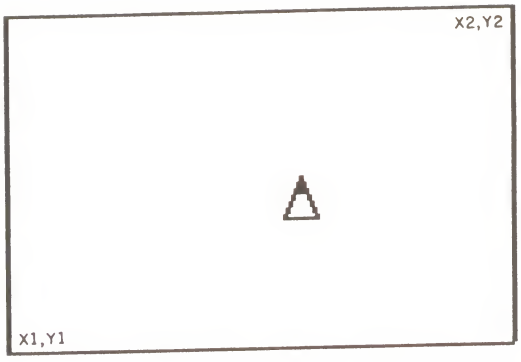


Figure 14.5: A rectangular box is defined by the coordinates of its opposite corners.

```
TO DRAWBOX :X1 :Y1 :X2 :Y2
MAKE "P0 POS
PENUP SETPOS SENTENCE :X1 :Y1 PENDOWN
SETX :X2
SETY :Y2
SETX :X1
SETY :Y1
PENUP SETPOS :P0
END
```

The same coordinates, X1, Y1, X2, and Y2, are used to test whether the turtle is in or out of the box. If the turtle is in the box, its X coordinate and its Y coordinate will both be larger than X1 and Y1 and smaller than X2 and Y2. That is, *all four* of those conditions must be true. INBOX? consists of one long command line.

```
TO INBOX? :X1 :Y1 :X2 :Y2
OUTPUT (AND (XCOR > :X1) (YCOR > :Y1)
    (XCOR < :X2) (YCOR < :Y2) )
END
```

AND outputs "TRUE only if *all* of its inputs are "TRUE. Otherwise it outputs "FALSE.

The turtle is out of the box if its X or Y coordinate is less than X1 or Y1, or larger than X2 or Y2, that is, if any one of those conditions is true. OUTBOX? also consists of one long command line:

```
TO OUTBOX? :X1 :Y1 :X2 :Y2
OUTPUT (OR (XCOR < :X1) (YCOR < :Y1)
    (XCOR > :X2) (YCOR > :Y2) )
END
```

OR outputs "TRUE if *any one* of its inputs is "TRUE. If all of them are "FALSE it outputs "FALSE.

These procedures make a nice example of inclusion and exclusion. For those who know mathematical logic, AND is equivalent to the logical operator *AND*, and OR is equivalent to the logical operator, *OR*.

You can use INBOX? and OUTBOX? to tell whether the turtle is in or

out of a rectangular region of the screen. There does not need to be a box *drawn* on the screen in order to use these procedures. The coordinates of any two points on the screen can be used to define a rectangular region. You will probably find it helpful to draw diagrams on paper when you use these procedures.

PITFALL

DRAWBOX, INBOX?, and OUTBOX? can have positive or negative inputs. If any one of the inputs is negative, be sure there is no space between the – and the input. Otherwise Logo will try to subtract.

The first two inputs to these procedures must be the X and Y coordinates of the *lower left corner* of a box as shown in Figure 14.5. The last two inputs must be the X and Y coordinates of the *upper right corner*. If the inputs are incorrect, the procedures may appear to work, but will give incorrect results.

Section 14.4. DISTANCE

The DISTANCE procedure outputs the distance from the turtle to a particular point. It can be used to tell whether the turtle is inside or outside of a particular circle, as in the target or racetrack game. If you know the Pythagorean Theorem you will understand where this procedure comes from: "The distance between two points is the square root of the sum of the squares of the sides of a right triangle joining the two points."

In Figure 14.6, XCOR and YCOR are the turtle's coordinates and :X1 and :Y1 are the coordinates of the point from which distance is being measured.

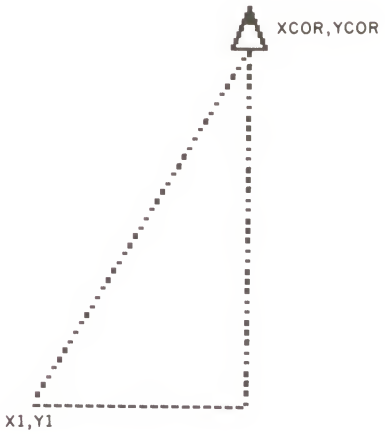


Figure 14.6 DISTANCE uses the Pythagorean Theorem to determine the distance between the turtle and any point on the screen.

The horizontal distance between the two points (XCOR − :X1). The vertical distance is (YCOR − :Y1). The direct distance is the square root (SQRT) of the sum of the squares of those values.

```
TO DISTANCE :P1
MAKE "X1 FIRST :P1
MAKE "Y1 LAST :P1
OUTPUT SQRT ( (XCOR − :X1) * (XCOR − :X1)
    + (YCOR − :Y1) * (YCOR − :Y1) )
END
```

That's all there is to it. (The parentheses *are* necessary!)

Section 14.5. READKEY

READKEY is used to read a single key from the keyboard in the midst of an ongoing procedure. It uses two Logo primitive commands, READCHAR and KEY. READCHAR waits for you to type a single character and then ouputs that character—just as READLIST waits for you to type a list. KEYP asks if you have typed a character. If you have, it outputs "TRUE; if not, it outputs "FALSE. READKEY uses these in a tricky combination.

```
TO READKEY
IF KEYP [OUTPUT READCHAR]
OUTPUT "
END
```

The "English translation" of READKEY is: "If a character has been typed, output that character as a Logo word. If not, output an empty word."

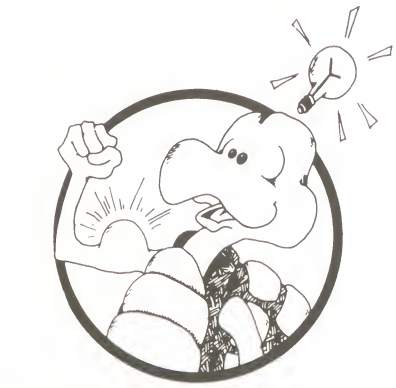
Here's a funny thing about READKEY. It won't do anything by itself. It has to be in some kind of repetitive procedure. First try READKEY by itself. See if you can make it do anything. Then try it in a procedure like this one:

```
TO PRINTKEYS
MAKE "LETTER READKEY
IF :LETTER = " [PRINTKEYS]
PRINTLETTER :LETTER
PRINTKEYS
END
TO PRINTLETTER :LETTER
IF :LETTER = " [STOP]
IF :LETTER = "A [PRINT [YOU TYPED A. NOW I CAN STOP]
    THROW "TOPLEVEL]
PRINT SENTENCE [YOU TYPED] :LETTER
END
```

READKEY is usually used as part of a COMMAND procedure where each letter makes the procedure do something different.

Section 14.6. PICKRANDOM and PICK

PICK is a classic example of a recursive Logo procedure with outputs. This is among the hardest aspects of Logo for many people to understand, which is why I haven't dealt with it until the last chapter.



There are two basic principles to remember in trying to understand this type of procedure.

1. The procedure must end by outputting something. It outputs back to the procedure or command that called it. If it calls another procedure to output something to it, it has to wait for that procedure to output before it can go on with its own job.
2. One basic strategy for designing recursive procedures that manipulate information is to assume you *already* know how to carry out the procedure with the *butfirst* of the object being manipulated. Use what you *assume* you have already done to complete something you don't yet know how to do. Sound peculiar? It is! But in some very interesting and important cases, it works very well.

Here are the procedures we are looking at:

```
TO PICKRANDOM :OBJECT
MAKE "NUMBER RANDOM COUNT :OBJECT
OUTPUT PICK (:NUMBER + 1) :OBJECT
END
TO PICK :NUMBER :OBJECT
IF :NUMBER = 1 [OUTPUT FIRST :OBJECT]
OUTPUT PICK (:NUMBER - 1) (BUTFIRST :OBJECT)
END
```

First I'll describe in words what each of these procedures does, then I'll try to explain how it does it.

PICK has two inputs, a number and an object. It outputs the *n*th element of the word or list (where *n* is the input number). PICK has the same effect as the Apple Logo command ITEM.

PICKRANDOM has one input, a word or list object. It outputs a random element from the object.

First, let's look at the PICK procedure.

```
TO PICK :NUMBER :OBJECT
IF :NUMBER = 1 [OUTPUT FIRST :OBJECT]
OUTPUT PICK (:NUMBER - 1) (BUTFIRST :OBJECT)
END
```

PICK is a recursive procedure which cannot stop unless its first input is 1, or it gets a value to OUTPUT from another PICK procedure. As you follow the example, pretend that all the PICK procedures are numbered. That way you can keep track of exactly what each one of them is doing.

Let's see what happens when you give the command

```
PICK 3 [A B C D]
```

Logo gives PICK two inputs, the number 3 and the list [A B C D].

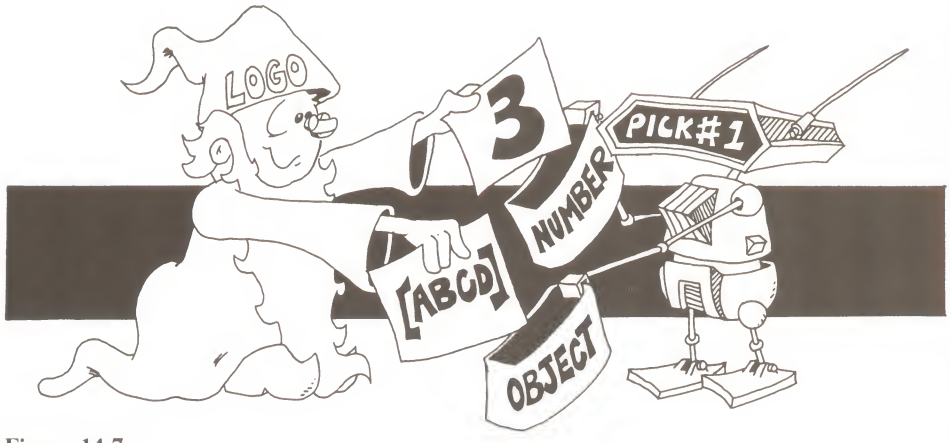


Figure 14.7a

PICK#1's number input is *not* 1, so it goes on to carry out its second line. This tells it to OUTPUT a value that it gets by calling another PICK procedure. PICK#1 gives this new procedure, PICK#2, a number input of 2, one less than PICK#1's original number input, 3. It gives PICK#2 an object input of [B C D], the BUTFIRST of PICK#1's original object input, [A B C D].

Then PICK#1 waits for a value to output back to Logo.

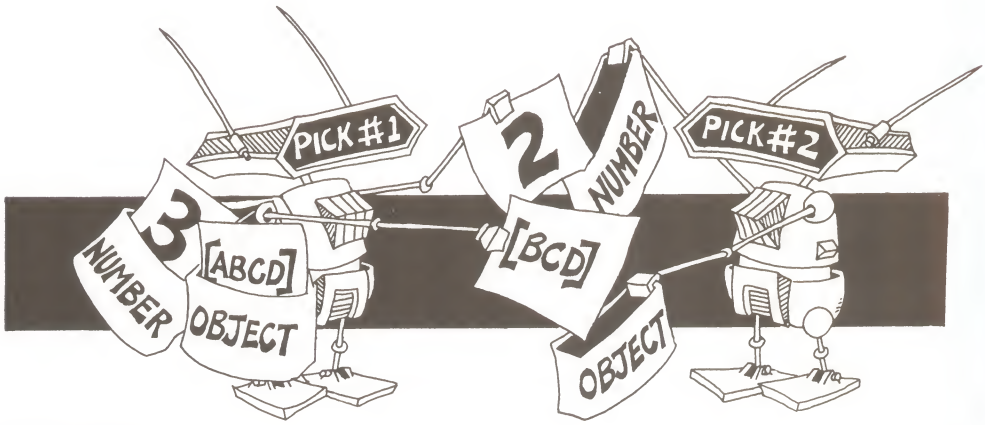


Figure 14.7b

PICK#2's inputs are 2 and [B C D]. Its number input still isn't 1, so it goes on to its next line, and calls PICK#3 with inputs of 1 and [C D]. Then PICK#2 waits for a value to output back to PICK#1.

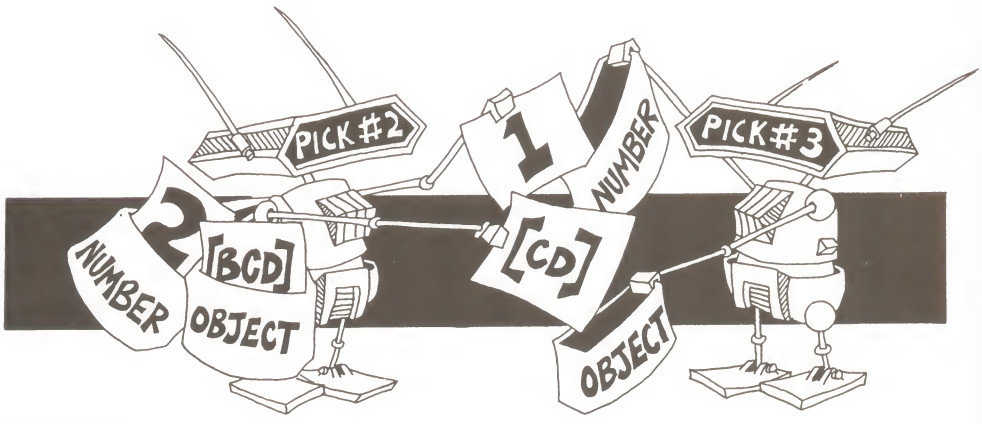


Figure 14.7c

PICK#3 has 1 and [C D] as inputs. Its first input *is* 1, so it outputs the FIRST elements of its second input, the word "C, back to PICK#2. Then PICK#3 stops.

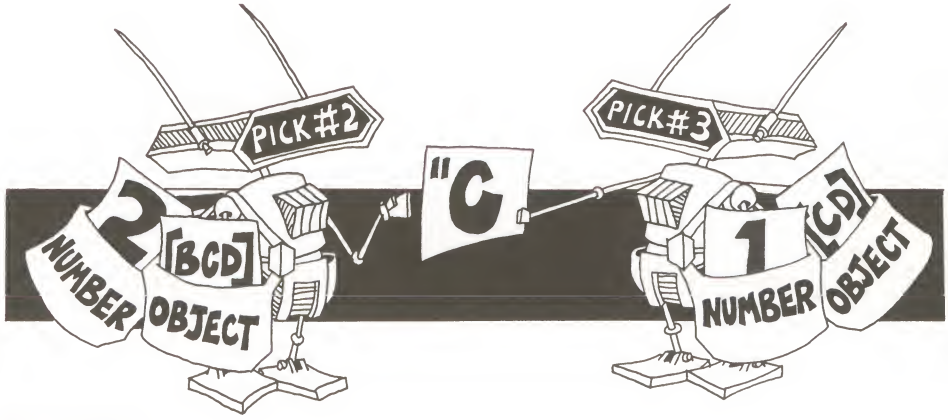


Figure 14.7d

PICK#2 gets "C as the value to be output in its second line. It outputs this back to PICK#1. Then PICK#2 stops.

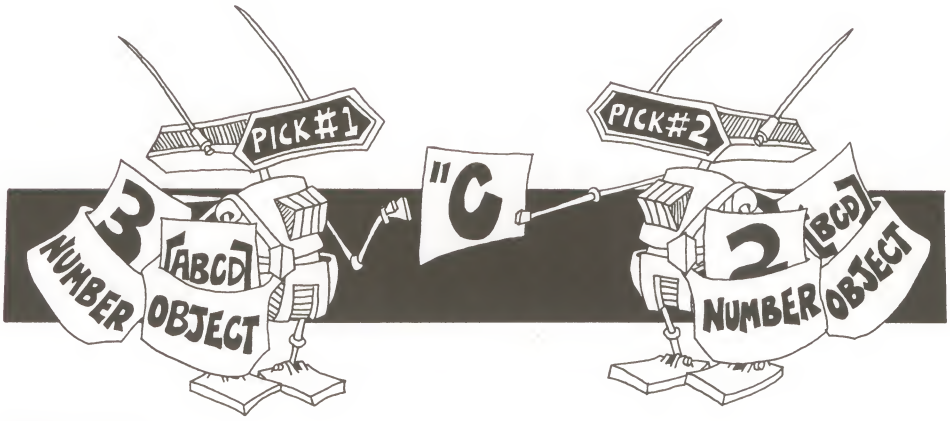


Figure 14.7e

PICK#1 now gets the value "C, and outputs it back to Logo. Then PICK#1 stops.

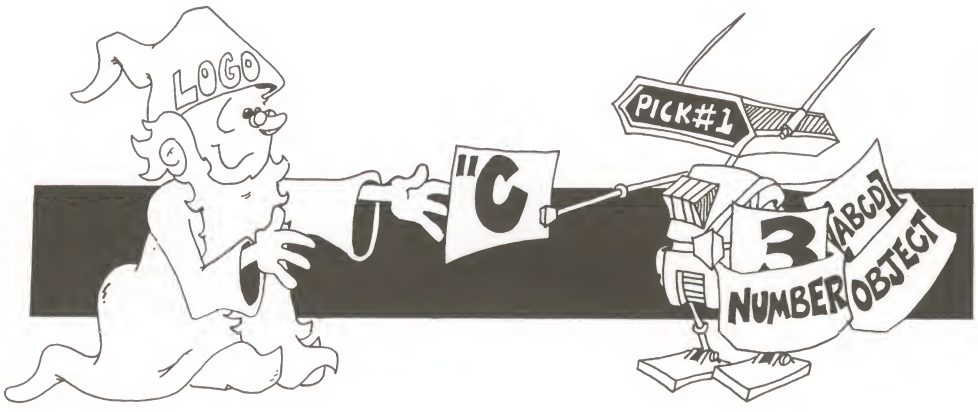


Figure 14.7f

Usually, a command like PICK is itself used as input to another command, for example, PRINT. Suppose you type:

```
PRINT PICK 3 [A B C D]
C
```

Notice that in this case PICK's final output, the letter C, was the third element of the original list, [A B C D]. If PICK's first input had been 2 or 4, it would have finally output B or D. If PICK's first input was larger than the number of elements in its second input, Logo would eventually complain, after the second input got chopped down to nothing.

Do you begin to get the idea? If you want to understand even more about this kind of process, I suggest you read Harold Abelson's *Logo for the Apple II* or *Apple Logo*, or *TI Logo*, published by Byte Books. I won't say any more about it here.

PICKRANDOM uses the Logo primitive command RANDOM and the subprocedure PICK to do its job. We're lucky this time—PICKRANDOM is *not* recursive.

```
TO PICKRANDOM :OBJECT
MAKE "NUMBER RANDOM COUNT :OBJECT
OUTPUT PICK (:NUMBER + 1) :OBJECT
END
```

Watch it work with an example.

```
PICKRANDOM [A B C D]
```

In the first line, COUNT [A B C D] outputs 4, which is used as the input to RANDOM. RANDOM 4 outputs a number from 0 to 3 (suppose this time it output 2).

In its second line, PICKRANDOM outputs PICK (2 + 1) [A B C D]. We already know that PICK 3 [A B C D] outputs C, so in this case we might see the following result:

```
PRINT PICKRANDOM [A B C D]
C
```

If we do it again we have three chances out of four of getting something different. To see what happens with several tries, try this:

```
REPEAT 10 [PRINT PICKRANDOM [A B C D]]
```

Section 14.7. READNUMBER

READNUMBER is another example of a recursive procedure which outputs. It supplies a command that really could have been part of Logo. The only Logo command that reads more than one character from the keyboard is READLIST, which reads and outputs a *list* typed at the keyboard. A number is a special kind of *word*. Since a word can never be equal to a list, we have to do something to get a number from the keyboard. Try this sequence:

```
MAKE "NUMBER READLIST
55
PRINT :NUMBER
55
```

Looks okay so far, but . . .

```
PRINT (:NUMBER = 55)
FALSE
```

Actually, :NUMBER is the *list* **[55]**, and in Logo, the list **[55]** is not equal to the number 55. To get a number from **[55]**, we take the first element of the list.

```
PRINT (FIRST :NUMBER) = 55
TRUE
```

So the main thing that READNUMBER has to do is output the first element of READLIST.

```
TO READNUMBER
OUTPUT FIRST READLIST
END
```

If we knew that every user really would type a number, READNUMBER could be just this simple. Suppose, however, that the user makes a mistake and types **RETURN**. READLIST will output an empty list, **[ ]**, and the result of READNUMBER will be the error message

```
FIRST DOESN'T LIKE [ ] AS INPUT
```

To avoid this and to make sure that the user really does type a number, I've used the Logo command NUMBERP to make a fancier version of READNUMBER

```
TO READNUMBER
MAKE "NUM1 READLIST
TEST :NUM1 = [ ]
IFTRUE [PRINT [PLEASE TYPE A NUMBER] OUTPUT READNUMBER]
TEST NOT NUMBERP FIRST :NUM1
IFTRUE [PRINT [PLEASE TYPE A NUMBER] OUTPUT READNUMBER]
IFFALSE [OUTPUT FIRST :NUM1]
END
```

Let's go through this step by step. Remember the first basic principle of writing procedures that give outputs: *Every time it is called*, READNUMBER *must end by outputting something*. Now let's see what this more complicated version of READNUMBER does.

The first line of READNUMBER gives the name "NUM1 to the list output by READLIST. READLIST, of course, just outputs whatever the user types.

Next READNUMBER performs two tests to see if :NUM1 is empty or if FIRST :NUM1 is *not* a number. If both tests are *false*, that is, if :NUM1 is *not* empty and FIRST :NUM1 *is* a number, the procedure goes on to the last line and outputs FIRST :NUM1. This will happen in most cases that READNUMBER is used.

If :NUM1 *is* empty or FIRST :NUM1 *not* a number, one of the tests is *true*. In this case the computer prints "PLEASE TYPE A NUMBER" and outputs a new READNUMBER. That is, it calls another READNUMBER procedure and waits to output whatever it gets from this *new* READNUMBER. This process will keep happening until either a number is typed or the com-

puter runs out of space in its working memory. When the user does type a number, that number is output back to the procedure that called it. That procedure outputs the number back to whatever procedure called it, and so on, back to the original procedure that called READNUMBER in the first place. It works much like the PICK procedure we talked about in Section 14.6.

Try just typing **RETURN** several times:

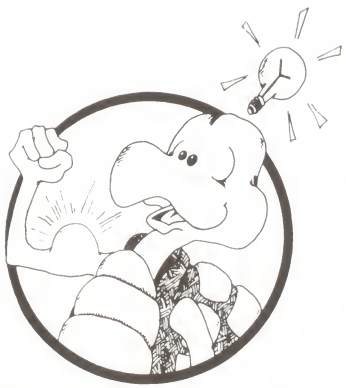
```
PRINT READNUMBER

PLEASE TYPE A NUMBER

PLEASE TYPE A NUMBER

PLEASE TYPE A NUMBER
```

etc.

The same lines will be repeated over and over until you *do* type a number (or press **CTRL-G** to stop everything). If you type anything else that is not a number, READNUMBER does the same thing.

```
PRINT READNUMBER
SEVEN
PLEASE TYPE A NUMBER
7
7
```

In one way, READNUMBER is simpler than PICK. If the user types a number as expected, READNUMBER outputs the number right away. Only if the user makes a mistake is READNUMBER recursive.



I've included this fancy version of READNUMBER as an example of a "user-proof" procedure—that is, one which offers some protection against user errors without spoiling the whole program. If I use the simple version of READNUMBER given above, any program with READNUMBER in it will bomb out if a user types **RETURN** by accident.

It's a good idea to make game and quiz programs as user-proof as you can so that they don't bomb out too often. If you want to learn more about how to make programs more "friendly" to users, I suggest you read *Apple Backpack: Humanized Programming in BASIC*, by Scot Kamins and Mitchell Waite, published by Byte Books. Even though the book is oriented toward BASIC, a lot of its suggestions for what a "humanized" program should be are valid for Logo or any other programming language.

Section 14.8. PRINTSCREEN

PRINTSCREEN commands are actually specialized for each printer that has the capability to print graphics. In this section I will explain the PRINTSCREEN commands for two different printers: the Silentype Printer made by Apple Computer Inc. and the Epson MX80 printer with GRAFTRAX chips and a Grappler interface card made by Orange Micro Incorporated. This last system was used to print all the turtle drawings in this

book, with the generous support of Orange Micro, who loaned me the equipment.

If you have a different kind of printer with graphics capability, you may be able to use these procedures as models and use the specific commands in your printer manual to make your printer print turtle drawings.

The Silentype Printer

The first command, PS, is just an abbreviation for PRINTSCREEN, which actually does all the work.

```
TO PS
PRINTSCREEN
END
TO PRINTSCREEN
.PRINTER 1
.DEPOSIT 53008 7 ;[DARKEST PRINT]
.DEPOSIT 53007 128 ;[UNIDIRECTIONAL PRINTING]
.DEPOSIT 53012 0 ;[REVERSE PRINT]
PRINT CHAR 17
.DEPOSIT 53007 0 ;[BI-DIRECTIONAL PRINTING]
.PRINTER 0
END
```

.PRINTER 1
turns on the printer (if it is in slot #1). If your printer is in any other slot, change the number in the first line to the slot number of the printer.

.DEPOSIT
is a Logo command that "deposits" a number into a memory location in the computer. The first input of .DEPOSIT is the memory location. The second input is the number deposited there. For example,

.DEPOSIT 53008 7
deposits the value 7 in memory location 53008. This makes the Silentype Printer print as darkly as possible. If the second input were smaller, the print would be less dark.

.DEPOSIT 53007 128
makes the printer print in only one direction. This makes copies of screen pictures sharper.

.DEPOSIT 53012 0
makes the printer reverse the screen image and print black on white. If you want white drawings on a black background change this line to .DEPOSIT 53012 255.

PRINT CHAR 17
is actually the command that makes the printing happen. CHAR 17 is the keyboard character **CTRL-Q**, which is the command that makes the Silentype Printer print the high-resolution graphics screen.

.DEPOSIT 53007 0
restores the printer to bi-directional printing, which is faster for printing text.

.PRINTER 0
turns the printer off.

The ; symbol on some of the lines is a *comment* procedure. Everything to the right of the ; can be read by someone looking at the program but will

be ignored by the computer. Comments are not used very much in Logo because the procedure names and variable names are usually as much as you need for a comment. When the commands are as obscure as these are, however, comments can be very helpful. Here is the procedure:

```
TO ; :COMMENT
END
```

The input list :COMMENT is not used for anything. In fact, the ; procedure doesn't do anything at all.

The Epson MX80 Printer with a Grappler Interface

The MX80/Grappler combination has four print options: regular size print, regular size with enhanced (darker) print, large size (rotated), and large size with enhanced print. These are obtained by four procedures, PS, PSE, PSB, and PSBE, which are abbreviations for PRINTSCREEN, PRINTSCREEN.E, PRINTSCREEN.BIG, and PRINTSCREEN.BIG.E.

All four procedures are very similar.

```
TO PRINTSCREEN
.PRINTER 1
REPEAT 2 [PRINT [ ] ]
PRINT (WORD CHAR 9 "G CHAR 13 )
.PRINTER 0
END
TO PRINTSCREEN.E
.PRINTER 1
REPEAT 2 [PRINT [ ] ]
PRINT (WORD CHAR 9 "GE CHAR 13 )
.PRINTER 0
END
TO PRINTSCREEN.BIG
.PRINTER 1
REPEAT 5 [PRINT [ ] ]
PRINT (WORD CHAR 9 "GDR CHAR 13 )
.PRINTER 0
END
TO PRINTSCREEN.BIG.E
.PRINTER 1
REPEAT 5 [PRINT [ ] ]
PRINT (WORD CHAR 9 "GDRE CHAR 13 )
.PRINTER 0
END
```

The graphics printing command for the MX80/Grappler system is a string of characters starting with **CTRL-I** (CHAR 9) and including G and any of the characters D, R, or E. It ends with **RETURN** (CHAR 13). G stands for graphics, D stands for double size, R stands for rotated, and E stands for enhanced. The WORD command in the third line of each procedure puts all the characters together into a complete string.

Appendix I Creating Your Own LWAL Procedures Disk

This appendix contains detailed instructions for creating your own LWAL Procedures Disk, if you choose not to purchase one. It assumes that you are already familiar with the use of the editing and filing systems for whatever version of Logo you are using. In other words, you should be familiar with either the information contained in Chapter 4 of this book or equivalent information for other versions of Logo. You should also have some familiarity with the rules of Logo syntax, especially with regard to the use of variables. This information is contained primarily in Chapter 9 and Chapter 7 of this book.

The LWAL Procedures Disk contains fifteen separate files, containing anywhere from one to nearly twenty procedures. Procedures in one file should be kept separate from those in another. You do this by clearing the working memory before starting to create a file and then clearing it again after saving one file and before beginning work on another.

All of the procedures in the appendix have been carefully tested and debugged. Since you may make some typing errors while typing them in, you should carefully test each set of procedures before saving them in a final version on your LWAL Procedures Disk. In each section of the appendix, references are given to the chapter in the book in which those particular procedures are used. This will allow you to test them properly.

Apple Logo Procedures Disk

Section I.1.
CIRCLES

First clear the working memory by typing

```
ERALL
```

Then copy these procedures exactly as written:

```
TO RCIRCLE :R
REPEAT 36 [RCP :R]
END
TO RARC :R
REPEAT 9 [RCP :R]
END
TO RCP :R
RIGHT 5
FORWARD :R * ( 3.14159 ) / 18
RIGHT 5
END
TO LCIRCLE :R
REPEAT 36 [LCP :R]
END
TO LARC :R
REPEAT 9 [LCP :R]
END
TO LCP :R
LEFT 5
FORWARD :R * ( 3.14159 ) / 18
LEFT 5
END
```

When all these procedures have been typed into the editor, type **CTRL-C** to leave the editor and return to Logo command mode. Test the circle procedures by using them as described in Chapter 2. Then save them on your LWAL Procedures Disk by typing

```
SAVE "CIRCLES
```

Section I.2.
CCIRCLE

Clear the working memory by typing

```
ERALL
```

Type these procedures exactly as written:

```
TO CCIRCLE :R
MAKE "PEN? PEN
MAKE "SHOWN? SHOWNP
HIDETURTLE
PENUP FORWARD :R
RIGHT 90
PENDOWN REPEAT 36 [RCP :R]
LEFT 90
PENUP
BACK :R
SETPEN :PEN?
IF :SHOWN? [SHOWTURTLE]
END
TO RCP :R
RIGHT 5
FORWARD :R * 3.14159 / 18
RIGHT 5
END
```

Now type **CTRL-C** to leave the editor and return to Logo command mode. Test CCIRCLE by using it as described in Chapter 10. Then save it on your disk by typing

```
SAVE "CCIRCLE
```

Section I.3. BOXES

First clear the working memory by typing

```
ERALL
```

Copy these procedures exactly as written:

```
TO DRAWBOX :X1 :Y1 :X2 :Y2
MAKE "P0 POS
PENUP SETPOS SENTENCE :X1 :Y1 PENDOWN
SETY :Y2
SETX :X2
SETY :Y1
SETX :X1
PENUP SETPOS :P0
END
```

OUTBOX? and INBOX? each contain one long line. In typing them, type each long line as a continuous line, pressing **RETURN** only where indicated. Do not type the word "RETURN."

```
TO OUTBOX? :X1 :Y1 :X2 :Y2 RETURN
OUTPUT (OR ( XCOR < :X1 ) ( YCOR < :Y1 ) )
    ( XCOR > :X2 ) ( YCOR > :Y2 ) ) RETURN
END RETURN
TO INBOX? :X1 :Y1 :X2 :Y2 RETURN
OUTPUT ( AND ( XCOR > :X1 ) ( YCOR > :Y1 )
    ( XCOR < :X2 ) ( YCOR < :Y2 ) ) RETURN
END RETURN
```

Type **CTRL-C** to leave the editor and return to Logo command mode. Test the procedures by using them as described in Chapter 12. Then save them on your disk by typing

```
SAVE "BOXES
```

Section I.4.
DISTANCE

Clear the working memory by typing

```
ERALL
```

Type the following procedure as shown. Type **RETURN** only where indicated. Do not type the word "RETURN."

```
TO DISTANCE :P1 RETURN
MAKE "X1 FIRST :P1 RETURN
MAKE "Y1 LAST :P1 RETURN
OUTPUT SQRT ( ( XCOR - :X1 ) * ( XCOR - :X1 )
    + ( YCOR - :Y1 ) * ( YCOR - :Y1 ) ) RETURN
END RETURN
```

Type **CTRL-C** to leave the editor and return to Logo command mode. Test DISTANCE by using it as described in Chapter 10. Then save it on your disk by typing

```
SAVE "DISTANCE
```

Section I.5.
READKEY

Clear the working memory by typing

```
ERALL
```

Then copy the following procedure exactly as written:

```
TO READKEY
IF KEYP [OUTPUT READCHAR]
OUTPUT "
END
```

Type **CTRL-C** to leave the editor and return to Logo command mode. Test READKEY by using it as described in Section 14.5. Then save it on your disk by typing

```
SAVE "READKEY
```

Section I.6.
PICKRANDOM

Clear the working memory by typing

```
ERALL
```

Type these procedures exactly as written:

```
TO PICKRANDOM :OBJECT
MAKE "NUMBER RANDOM COUNT :OBJECT
OUTPUT PICK (:NUMBER + 1) :OBJECT
END
```

```
TO PICK :NUMBER :OBJECT
IF :NUMBER = 1 [OUTPUT FIRST :OBJECT]
OUTPUT PICK (:NUMBER - 1) (BUTFIRST :OBJECT)
END
```

Type **CTRL-C** to leave the editor and return to Logo command mode. Test the procedures by using them as described in Chapter 13. Then save them on your disk by typing

```
SAVE "PICKRANDOM
```

Section I.7. READNUMBER

Clear the working memory by typing

```
ERALL
```

Then copy this procedure exactly as written:

```
TO READNUMBER
MAKE "NUM1 READLIST
TEST :NUM1 = [ ]
IFTRUE [PRINT [PLEASE TYPE A NUMBER] OUTPUT READNUMBER]
TEST NOT NUMBERP FIRST :NUM1
IFTRUE [PRINT [PLEASE TYPE A NUMBER] OUTPUT READNUMBER]
IFFALSE [OUTPUT FIRST :NUM1]
END
```

Type **CTRL-C** to leave the editor and return to Logo command mode. Test READNUMBER by using it as described in Chapter 9. Then save it on your disk by typing

```
SAVE "READNUMBER
```

Section I.8. PRINTSCREEN

Clear the working memory by typing

```
ERALL
```

The first set of procedures are to be used with a *Silentype* (TM) printer, sold by Apple Computer Inc. Copy them exactly as written:

```
TO PRINTSCREEN
.PRINTER 1
.DEPOSIT 53008 7 ;      [DARKEST PRINT]
.DEPOSIT 53007 128 ;    [UNIDIRECTIONAL]
.DEPOSIT 53012 0 ;      [REVERSE PRINT]
PRINT CHAR 17
.DEPOSIT 53007 0 ;      [BIDIRECTIONAL]
.PRINTER 0
END
TO PS
PRINTSCREEN
END
TO ; :COMMENT
END
```

Type **CTRL-C** to leave the editor and return to Logo command mode. Test these procedures by using them as described in Chapter 4. Then save them on your disk by typing

```
SAVE "PRINTSCREEN.S
```

The next set of procedures are for use with the Epson MX80 or MX100 printers, and the *Grappler* graphics interface manufactured by Orange Micro. Clear the working memory by typing

```
ERALL
```

Then copy the following procedures exactly as written:

```
TO PRINTSCREEN
.PRINTER 1
REPEAT 2 [PRINT [ ] ]
PRINT ( WORD CHAR 9 "G CHAR 13 )
.PRINTER 0
END
TO PRINTSCREEN.E
.PRINTER 1
REPEAT 2 [PRINT [ ] ]
PRINT ( WORD CHAR 9 "GE CHAR 13 )
.PRINTER 0
END
TO PRINTSCREEN.BIG
.PRINTER 1
REPEAT 5 [PRINT [ ] ]
PRINT ( WORD CHAR 9 "GDR CHAR 13 )
.PRINTER 0
END
TO PRINTSCREEN.BIG.E
.PRINTER 1
REPEAT 5 [PRINT [ ] ]
PRINT ( WORD CHAR 9 "GDRE CHAR 13 )
.PRINTER 0
END
TO PS
PRINTSCREEN
END
TO PSE
PRINTSCREEN.E
END
TO PSB
PRINTSCREEN.BIG
END
TO PSBE
PRINTSCREEN.BIG.E
END
```

Type **CTRL-C** to leave the editor and return to Logo command mode. Test these procedures with an Epson printer and a Grappler interface as described in Chapter 4. Save them on your disk by typing

```
SAVE "PRINTSCREEN.G
```

Section I.9.
GUESSNUMBER

Clear the working memory by typing

```
ERALL
```

Before typing the GUESSNUMBER procedures, load in the tool procedure READNUMBER by typing LOAD "READNUMBER

Then type these procedures exactly as written:

```
TO GUESSNUMBER
INSTRUCTIONS
CHOOSENUMBER
GETGUESS
END

TO INSTRUCTIONS
CLEARTEXT
PRINT [I AM THINKING OF A NUMBER BETWEEN 0]
PRINT [AND 100. SEE IF YOU CAN GUESS IT.]
END

TO CHOOSENUMBER
MAKE "NUMBER 1 + RANDOM 99
END

TO GETGUESS
TYPE ">
MAKE "GUESS READNUMBER
CHECKGUESS :GUESS :NUMBER
END

TO CHECKGUESS :GUESS :NUMBER
IF :GUESS = :NUMBER [PRINT [GOT IT!] STOP]
IF :GUESS > :NUMBER [PRINT [TOO HIGH] GETGUESS STOP]
IF :GUESS < :NUMBER [PRINT [TOO LOW] GETGUESS STOP]
END
```

Type **CTRL-C** to leave the editor and return to Logo command mode. Test GUESSNUMBER by using it as described in Chapter 9. Then save it on your disk by typing SAVE "GUESSNUMBER

Section I.10.
MATHQUIZ

Clear the working memory by typing ERALL

Then load in the READNUMBER procedure by typing

```
LOAD "READNUMBER
```

Then copy all these procedures exactly as written:

```
TO MATHQUIZ
CLEARTEXT
GETTOTAL
MAKE "COUNT 1
MAKE "SCORE 0
ADDQUIZ :COUNT :TOTAL :SCORE
END
TO GETTOTAL
PRINT [HOW MANY PROBLEMS DO YOU WANT?]
MAKE "TOTAL READNUMBER
END
TO ADDQUIZ :COUNT :TOTAL :SCORE
CLEARTEXT
GETNUMBERS
GIVEPROBLEM
GETANSWER
WAITFORUSER
IF :COUNT = :TOTAL [FINISH STOP]
ADDQUIZ ( :COUNT + 1 ) :TOTAL :SCORE
END
TO GETNUMBERS
MAKE "NUMBER1 RANDOM 100
MAKE "NUMBER2 RANDOM 100
MAKE "RIGHTANSWER   :NUMBER1 + :NUMBER2
END
TO GIVEPROBLEM
PRINT SENTENCE [PROBLEM] :COUNT
PRINT [ ]
TYPE ( SENTENCE :NUMBER1 [+] :NUMBER2 [=] )
END
TO GETANSWER
MAKE "RESPONSE READNUMBER
TEST :RESPONSE = :RIGHT ANSWER
IFTRUE [PRINT [CORRECT]]
IFTRUE [MAKE "SCORE :SCORE + 1]
IFFALSE [PRINT SENTENCE [SORRY, THE ANSWER IS]
    :RIGHTANSWER]
END
TO WAITFORUSER
PRINT [PLEASE PRESS RETURN]
PRINT READLIST
END
TO FINISH
CLEARTEXT
PRINT SENTENCE [YOUR SCORE IS] :SCORE
PRINT ( SENTENCE [OUT OF] :TOTAL [PROBLEMS] )
END
```

Type **CTRL-C** to leave the editor and return to Logo command mode. Test MATHQUIZ by using it as described in Chapter 9. Then save it on your disk by typing SAVE "MATHQUIZ

Section I.11. SHOOT

Clear the working memory by typing

```
ERALL
```

Next, load in the tool procedures CCIRCLE, DISTANCE, and READNUMBER that should already have been saved on your disk. If you have not yet saved these procedures on your disk, type them in and save them individually before typing in the SHOOT procedures. Load in the procedures by typing

```
LOAD "CCIRCLE
LOAD "DISTANCE
LOAD "READNUMBER
```

Then type the following procedures exactly as written:

```
TO START
STARTDATA
STARTGAME
END
TO STARTDATA
MAKE "SHOTNUMBER 0
MAKE "XTARGET ( 90 - 10 * RANDOM 19 )
MAKE "YTARGET ( 80 - 10 * RANDOM 6 )
MAKE "XSTART ( 90 - 10 * RANDOM 19 )
MAKE "YSTART ( -10 * RANDOM 3 )
MAKE "HSTART ( 10 * RANDOM 36 )
MAKE "PTARGET SENTENCE :XTARGET :YTARGET
MAKE "PSTART SENTENCE :XSTART :YSTART
END
TO STARTGAME
CLEARSCREEN
SETBG 6
HIDETURTLE
DRAWTARGET :PTARGET
STARTTURTLE :PSTART :HSTART
SHOWTURTLE
END
TO DRAWTARGET :PTARGET
PENUP
SETPOS :PTARGET
CCIRCLE 10
END
TO STARTTURTLE :PSTART :HSTART
PENUP
SETPOS :PSTART
SETHEADING :HSTART
END
TO SHOOT
MAKE "SHOTNUMBER :SHOTNUMBER + 1
PRINT [HOW FAR?]
MAKE "SHOT READNUMBER
PENDOWN FORWARD :SHOT
```

```
TEST ( DISTANCE :PTARGET ) < 10
IFTRUE [HIT]
IFFALSE [MISS]
END
TO HIT
PRINT [CONGRATULATIONS! YOU HIT THE TARGET!]
PRINT ( SENTENCE [IT TOOK YOU ONLY] :SHOTNUMBER [SHOTS.] )
END
TO MISS
PRINT SENTENCE [MISSED SHOT NUMBER] :SHOTNUMBER
WAIT 200
STARTTURTLE :PSTART :HSTART
END
```

Type **CTRL-C** to leave the editor and return to Logo command mode. Test the SHOOT procedures by using them as described in Chapter 3. Then save them on your disk by typing

```
SAVE "SHOOT
```

Section I.12. QUICKDRAW

Clear the working memory by typing

```
ERALL
```

First you will have to load in the tool procedure READKEY from your disk. If you have not yet saved READKEY on your disk, type in and save it before typing in the QUICKDRAW procedures. Load it in by typing

```
LOAD "READKEY
```

Then type these procedures exactly as written:

```
TO QD
START
QUICKDRAW
END
TO START
MAKE "DRAWLIST [ ]
CLEARSCREEN
END
TO QUICKDRAW
COMMAND
QUICKDRAW
END
TO COMMAND
MAKE "COM READKEY
IF :COM = "F [FORWARD 20 ADDLETTER :COM]
IF :COM = "B [BACK 20 ADDLETTER :COM]
IF :COM = "R [RIGHT 30 ADDLETTER :COM]
IF :COM = "L [LEFT 30 ADDLETTER :COM]
IF :COM = "E [FINISH THROW "TOPLEVEL]
END
```

```
TO ADDLETTER :LETTER
MAKE "DRAWLIST SENTENCE :DRAWLIST :LETTER
END
TO FINISH
SPLITSCREEN
PRINT [PLEASE CHOOSE ONE WORD AS A NAME]
PRINT [FOR THIS DRAWING.]
PRINT [TO FORGET IT, JUST PRESS RETURN]
MAKE "REPLY READLIST
IF :REPLY = [ ] [STOP]
MAKE FIRST :REPLY :DRAWLIST
END
TO RD :DRAWLIST
IF :DRAWLIST = [ ] [STOP]
RECOMMAND FIRST :DRAWLIST
RD BUTFIRST :DRAWLIST
END
TO RECOMMAND :COM
IF :COM = "F [FORWARD 20]
IF :COM = "B [BACK 20]
IF :COM = "R [RIGHT 30]
IF :COM = "L [LEFT 30]
END
```

Type **CTRL-C** to leave the editor and return to Logo command mode. Test the QUICKDRAW procedures by using them as described in Chapter 3. Then save them on your disk by typing

```
SAVE "QUICKDRAW
```

Section I.13. RACE

Clear the working memory by typing ERALL

Before typing in the RACE procedures, you will need to load in the tool procedures CCIRCLE, DISTANCE, and READKEY. If you have not yet saved these procedures on your disk, type them in and save them before typing in the RACE procedures. Then load them in by typing

```
LOAD "CCIRCLE
LOAD "DISTANCE
LOAD "READKEY
```

Then type the following procedures exactly as written:

```
TO RACE
DRAWTRACK
SETSTART
RACECAR 0
END
TO DRAWTRACK
CLEARSCREEN
HIDETURTLE
CCIRCLE 50
CCIRCLE 70
LEFT 90
PENUP FORWARD 50
```

```
PENDOWN FORWARD 20
PENUP BACK 70
RIGHT 90
END
TO SETSTART
PENUP SETPOS [-60 0]
SETHEADING 0
FORWARD 1 SHOWTURTLE
MAKE "OLDY 1
MAKE "DISTANCE 0
END
TO RACECAR :TIME
IF FINISHED? [FINISH STOP]
IF CRASHED? [CRASH STOP]
FORWARD :DISTANCE
COMMAND
RACECAR :TIME + 1
END
TO CRASHED?
IF ( DISTANCE [0 0] ) > 70 [OUTPUT "TRUE]
IF ( DISTANCE [0 0] ) < 50 [OUTPUT "TRUE]
OUTPUT "FALSE
END
TO CRASH
PRINT [YOU CRASHED INTO THE TRACK WALL]
END
TO FINISHED?
IF AND ( YCOR > 0 ) ( :OLDY < 0 ) [OUTPUT "TRUE]
MAKE "OLDY YCOR
OUTPUT "FALSE
END
TO FINISH
PRINT [YOU CROSSED THE FINISH LINE]
PRINT SENTENCE [WITH A TIME OF] :TIME
END
TO COMMAND
MAKE "COM READKEY
IF :COM = " [STOP]
IF :COM = "F [MAKE "DISTANCE :DISTANCE + 5 STOP]
IF :COM = "S [MAKE "DISTANCE :DISTANCE - 5 STOP]
IF :COM = "R [RIGHT 30 STOP]
IF :COM = "L [LEFT 30 STOP]
END
TO RESTART
SETSTART
RACECAR 0
END
```

Type **CTRL-C** to leave the editor and return to Logo command mode. Test the RACE procedures by using them as described in Chapter 12. Then save them on your disk by typing

```
SAVE "RACE
```

Section I.14. POET

Clear the working memory by typing ERALL

Before typing in the POET procedures, you will need to load in the two tool procedures READNUMBER and PICKRANDOM. If you haven't yet saved these procedures on your disk, type them in and save them now. Then load them in by typing

```
LOAD "PICKRANDOM
LOAD "READNUMBER
```

Then type in the following procedures and variable names exactly as written. Notice that the variables "ARTICLELIST, "ADJECTIVELIST, "NOUNLIST, "VERBLIST, and "PREPOSITIONLIST are *not* procedures. They are typed in using the Logo command MAKE before leaving the editor.

```
TO POEMS
CLEARTEXT
PRINT [HOW MANY POEMS DO YOU WANT?]
MAKE "N READNUMBER
CLEARTEXT PRINT [ ] PRINT [ ]
PRINT SENTENCE :N [POEMS BY THE LOGO POET]
PRINT [ ]
REPEAT :N [POET PRINT [ ] ]
END
TO POET
PRINT LINE1
PRINT LINE2
PRINT LINE3
END
TO LINE1
OUTPUT ( SENTENCE ARTICLE ADJECTIVE NOUN )
END
TO LINE2
OUTPUT ( SENTENCE ARTICLE NOUN VERB PREPOSITION ARTICLE
    ADJECTIVE NOUN )
END
TO LINE3
OUTPUT ( SENTENCE ADJECTIVE ADJECTIVE NOUN )
END
TO ARTICLE
OUTPUT PICKRANDOM :ARTICLELIST
END
TO ADJECTIVE
OUTPUT PICKRANDOM :ADJECTIVELIST
END
TO NOUN
OUTPUT PICKRANDOM :NOUNLIST
END
TO VERB
OUTPUT PICKRANDOM :VERBLIST
END
```

```
TO PREPOSITION
OUTPUT PICKRANDOM :PREPOSITIONLIST
END
TO WORDLISTS
CLEARTEXT
PRINT [ARTICLES:]
PRINT [ ] PRINT :ARTICLELIST
PRINT [ ]
PRINT [NOUNS:]
PRINT [ ] PRINT :NOUNLIST
PRINT [ ]
PRINT [VERBS:]
PRINT [ ] PRINT :VERBLIST
PRINT [ ]
PRINT [PLEASE PRESS RETURN TO CONTINUE]
PRINT READLIST
CLEARTEXT
PRINT [ADJECTIVES:]
PRINT [ ] PRINT :ADJECTIVELIST
PRINT [ ]
PRINT [PREPOSITIONS:]
PRINT [ ] PRINT :PREPOSITIONLIST
END
MAKE " ARTICLELIST [A THE ONE EACH EVERY]
MAKE "ADJECTIVELIST [AUTUMN HIDDEN BUBBLING BOILING
  SWIRLING GREEN BITTER MISTY SILENT EMPTY DRY DARK
  SUMMER ICY DELICATE QUIET WHITE COOL SPRING WINTER
  TWILIGHT DAWN CRIMSON AZURE BLUE BILLOWING COLD DAMP
  LIMPID FROSTY WILD SPARKLING MELLOW SCENTED STILL]
MAKE "NOUNLIST [WATERFALL RIVER BREEZE MOON RAIN WIND
  SEA MORNING SNOW LAKE SUNSET SHADOW PINE LEAF GLITTER
  DAWN FOREST HILL CLOUD MEADOW BROOK BIRD BUTTERFLY
  DEW DUST FIR NIGHT POND SNOWFLAKE VIOLET FLOWER
  FIREFLY FOX FISH OTTER CROW RAVEN OWL]
MAKE "VERBLIST [SHAKES DRIFTS [HAS STOPPED] SLEEPS CREEPS
  MURMURS FLIES FLUTTERS [HAS FALLEN] [IS TRICKLING] [HAS
  PASSED] [HAS RISEN] FLOATS LEAPS RACES HIDES [IS HIDDEN]
  CRIES [CRIES OUT] AWAKES RISES]
MAKE "PREPOSITIONLIST [ON IN OFF [OUT OF] UNDER OVER NEAR
  BENEATH ERE OVER AROUND BELOW ABOVE]
```

Type **CTRL-C** to leave the editor and return to Logo command mode. Test the POET procedures by using them as described in Chapter 13. Then save them on your disk by typing

```
SAVE "POET
```

Your LWAL Procedures Disk is now complete!

Appendix II Care and Management of Disks and Files

In this appendix I will explain how to initialize, maintain, and copy Logo work disks. I will also give some advice about caring for disks and the information stored on them.

The most important part of your Logo system is the information and procedures that you have created and saved. And this is just the part that is most easily damaged by accident or carelessness. Because information stored on disks is so easy to change, it's important to have a clear understanding of how to protect and maintain it. It's also a good idea to keep an extra copy of any important work on a backup disk.

In Sections II.5, II.6, and II.7, I will explain a bit about using *packages* of procedures, how to *bury* them, and how to create or modify a *startup file*. Section II.7 also explains how to add the circle procedures used in this book to the startup file.

Section II.1. Initializing Logo Work Disks

Logo work disks are initialized using BASIC commands and using the same process as disks used for other Apple programs. To initialize a Logo work disk, start up your computer using Applesoft BASIC. This can be done with the DOS 3.3 System Master.

1. Insert the DOS 3.3 System Master disk in the disk drive and start the computer.
2. When you see the Applesoft prompt,], insert a blank file disk to be initialized. You can also reinitialize an old disk, but all the information that was on it will now be destroyed.
3. Type

   INIT HELLO **RETURN**

   The disk drive will whir and click for about a minute. When it stops you've got a disk that's ready to accept Logo files.

4. If you want to add the Apple Logo startup file to your new Logo work disk, just start Apple Logo in the usual way. Press **RETURN** without removing the Language Disk. When the Welcome to Logo message appears, remove the language disk and insert your new work disk. Then type

   SAVE "STARTUP "AIDS

   This will save a *package* of procedures called "AIDS in a file called "STARTUP on your Logo work disk. I will explain more about *packages* in Section II.5 and II.6 of this appendix. In Section II.7, I will show how you can add new procedures to the startup file or create your own startup file.

Section II.2. Copying Logo Work Disks

It is sometimes necessary to make a back-up copy of an entire disk. You might want to share a disk full of Logo procedures with a friend. Or you might want a back-up copy of one of your disks as a protection against accidentally losing some important programs. A copy program is available on your DOS 3.3 System Master Disk. It works a little differently depending on whether you have one disk drive or two. Complete directions are given in your DOS 3.3 Manual. Here is a short version of what to do:

1. Insert the DOS 3.3 System Master into drive 1. Turn on the power.
2. When the Applesoft BASIC prompt,], appears, type

   RUN COPYA **RETURN**

3. The computer will then ask which slot number and how many drives you are using. If you have only one disk drive, you will type

   RETURN
   RETURN
   RETURN
   1 (no **RETURN**)

   If you have two disk drives, just press **RETURN** four times.
4. If you have *one* disk drive, insert the disk you are copying from and press **RETURN**. Follow all the directions printed on the screen about switching disks. You will alternate between the original and the duplicate disk until the computer tells you it is finished copying.

If you have *two* disk drives, your job will be a lot easier and quicker. Insert the disk you are copying from, the original, in drive 1. Insert the new disk in drive 2. *Be sure not to mix these up! If you do, you'll be very sorry.* Then press **RETURN**, and the computer will do the rest.

Protect yourself against actually copying a blank disk onto a good one and losing all your information. Just put one of those small, sticky tabs that come with a box of disks over the little notch on the disk *from* which you are copying. This will stop the computer from writing any information onto that disk accidentally.

Section II.3. Copying Files from One Disk to Another

Another way to make back-up copies of your work is to save your procedures on two different initialized Logo work disks as you go along. After saving a file once in the normal way, insert a new disk and save the file again, using the same file name. Then remove the back-up disk and insert your regular disk as you continue your work.

If you want to copy an entire file onto another disk so that you can share a project with another person, the process is very similar. Suppose you want to transfer a file called "TRUCKS. First, clear the working memory by typing

ERALL

Then insert your work disk and type

LOAD "TRUCKS

Now insert the second disk and type

```
SAVE "TRUCKS
```

Section II.4. Updating Files

A good way to protect yourself against accidentally losing information is to *update* your files as you work. This is especially useful when you are working on a large project. When you're working on a large project, it's a good idea to save your file every time you finish teaching the computer an important new procedure. Instead of always using the same old file name, "TRUCKS, for example, you can *number* your files, increasing the number by 1 every time you save a new file, First, type

```
SAVE "TRUCKS1
```

A little later, save something new by typing

```
SAVE "TRUCKS2
```

Still later,

```
SAVE "TRUCKS3
```

and so on. If you type CATALOG, you will see a large number of files on your disk. When you are all finished working, you can erase all the unneeded files, using the ERASEFILE command. It is important to remember to erase the extra files eventually so that you don't waste disk space.

Section II.5. Saving Some of the Procedures in a File

A slightly trickier thing to do is to save *some* procedures from one file in another file. You could use this process to clean up a file that has too many unneeded procedures in it or to transfer some useful tool procedures from one file to another. The process includes a few extra steps. There are several ways to do this. One involves *erasing* all the unneeded procedures and saving the rest. Another involves making *packages* of procedures that you want to keep or erase and saving the package separately.

The first method I'll describe involves erasing unneeded procedures from the work space (working memory). First, clear the work space by typing

```
ERALL
```

Then, load the file that has the procedures you want to save separately by typing

```
LOAD "TRUCKS
```

(Use whatever file name you want instead of "TRUCKS, of course.) Now type

```
POTS
```

to see a list of *all* the procedures in this file. One at a time, erase all the procedures that you *don't* want in the new file. You do this with the ERASE command.

```
ERASE "BOX
ERASE "WHEEL1
.
.
.
```

When you are finished erasing procedures, type

```
POTS
```

to check that only the procedures you want are in the working memory. Then, save the procedures in a new file.

If you want to add these new procedures to a *second* file, load that file now. Suppose the second file is called "CARS. Type

```
LOAD "CARS
```

Then type

```
POTS
```

You should see that your working memory contains all of the procedures in "CARS plus the procedures you wanted from "TRUCKS. Now you can save them all in the file called "CARS. First, erase the old file by typing

```
ERASEFILE "CARS
```

Then, save all the procedures by typing

```
SAVE "CARS
```

Another way to save some procedures from one file in another file is to put all the procedures you want to save in a *package*. Apple Logo has a command called PACKAGE that lets you put a list of procedures into a *package* that can be saved or erased separately. Here's how it works. First, load the file called "TRUCKS and decide which procedures you want to put into a package. Then, pick a name for the package and type

```
PACKAGE "NEWTRUCKS [TRUCK BIGBOX SMALLBOX WHEELS . . .]
```

Make sure that all the procedures you want to keep in the new file are in the list. Now, save the package in a file. It's a good idea to use different names for the file and the package. Type

```
SAVE "NEWT "NEWTRUCKS
```

This saves a *package* of procedures called "NEWTRUCKS in a *file* called "NEWT. If you had used SAVE with only one input in the usual way, *all* the

procedures in the work space would have been saved in the new file. Now, clear the work space by typing

```
ERALL
```

Load the procedures in the "CAR file and the procedures in the "NEWT file by typing

```
LOAD "CARS
LOAD "NEWT
```

Now, erase the old "CARS file and save all the procedures by typing

```
ERASEFILE "CARS
SAVE "CARS
```

There are many other ways to do this sort of thing.

Section II.6. Burying Packages of Procedures

Apple Logo allows you to *bury* a package of procedures so that buried procedures will not be accidentally erased from the work space. Buried procedures are also less visible to a user because their names are not listed when the POTS command is typed.

There are two main reasons to bury procedures. The first is to make tool procedures look like primitive Logo commands. Since buried procedures are not printed out unless they are specially called for, they seem to the user to be part of the "system." Procedures in the startup file have been buried for this reason.

The second reason is to keep tool procedures from being saved in the same file as the other procedures you may be working on. If all the procedures in the startup file were saved every time you saved your work space, you would fill up your work disk a lot faster. Since the startup procedures are available whenever you need them, it's not necessary to save them over and over again in every file.

Here's how it works. Let's use circle-and-arc procedures as an example. Circles and arcs are made with six procedures, RCIRCLE, LCIRCLE, RARC, LARC, RCP and LCP. These can be found in the "CIRCLES file on the LWAL procedures disk. The first thing to do is to put them into a package. There are two ways to do this.

1. Load them directly into a package. Let's call the package "CIRCTOOLS. Type

   ```
   LOAD "CIRCLES "CIRCTOOLS
   ```

   This loads the file called "CIRCLES into the working memory, or work space, and puts all the procedures in that file into a package called "CIRCTOOLS.

2. The other way is to load them into the work space normally and package them with the PACKAGE command.

   ```
   LOAD "CIRCLES
   PACKAGE "CIRCTOOLS [RCIRCLE LCIRCLE RARC LARC RCP LCP]
   ```

You can also use the PACKAGE command with any group of procedures in the working memory.

Now, let's bury them. First type POTS. You should see RCIRCLE and the rest among the procedures listed. Now type

```
BURY "CIRCTOOLS
POTS
```

Now the circle procedures will *not* be listed. You can erase all the rest of your procedures using ERALL and the circle procedures will not be affected. You can save all your other procedures without saving the circle procedures.

Burying can be reversed with the UNBURY command.

```
UNBURY "CIRCTOOLS
```

You can print out the names or the procedures in a buried package by using the package name as input to POTS or POPS.

```
POTS "CIRCTOOLS
```

prints the titles of all procedures in the "CIRCTOOLS package.

```
POPS "CIRCTOOLS
```

prints out all the procedures in the package.

```
PO "RCIRCLE
```

prints out the procedure, RCIRCLE, even if it's part of a buried package.

If you save a *package* of buried procedures in a file, they will be buried whenever they are loaded back in. Just use the package name after the file name when saving them.

```
SAVE "NEWCIRCLES "CIRCTOOLS
```

Whenever you load "NEWCIRCLES, you will load in a package of buried circle procedures.

In the next section, I'll show you how to use a buried package to change the Apple Logo startup file or to create a new one of your own. There's really a lot more that can be done with packaging and burying, but I don't have room to explain it all here. A lot of information is given in the *Apple Logo Reference Manual*. Unfortunately, it's not explained very clearly, but you can figure it out if you work slowly and try things. Exploring with packages and files is the same as exploring with the turtle. You can't hurt the computer. Just make sure you've saved all the information you really need and put a backup disk away before experimenting. There's nothing worse than losing vital information while exploring with files. I know. I've done it.

Section II.7. Modifying the Startup File

The startup file contains a buried package called "AIDS.

If you load Logo normally and then type

```
POTS "AIDS
```

you'll see the names of all the procedures and subprocedures in the "AIDS package. These are the procedures that are saved in the startup file, which Logo is programmed to load automatically when starting.

Now suppose you want to create your own startup file on the Logo Language Disk or on your own Logo work disk. Let's use circle procedures as an example. You start by making a package of buried circle procedures as shown in the last section.

```
LOAD "CIRCLES "CIRCTOOLS
BURY "CIRCTOOLS
```

Now erase the old startup file and save the new one with the "CIRCTOOLS package instead.

```
ERASEFILE "STARTUP
SAVE "STARTUP "CIRCTOOLS
```

Now, you'll have a set of buried circle procedures in your work space everytime you start Logo with that particular work disk.

Suppose you just want to *add* these circle procedures to "STARTUP without erasing the old procedures. Just save *two* buried packages in the startup file. Remember that the original package in the startup file was called "AIDS. You can save both "AIDS and "CIRCTOOLS in the startup file this way.

```
ERASEFILE "STARTUP
SAVE "STARTUP [AIDS CIRCTOOLS]
```

Now all of those procedures will be in the startup file. I recommend this approach as the easiest way to do projects with circles without thinking too much about them.

I've used circle procedures as examples in these two sections. But of course you can bury any package of procedures and put any number of packages of procedures into your startup files. Some people like to make different startup files for different work disks because different projects need different kinds of tools. Just be careful when you experiment with these commands. Make sure you have at least one disk with the original startup file on it, and put that disk away in a safe place before you start erasing files.

Section II.8. Some Commonsense Tips for Caring for Disks

This appendix ends with a few "do's and don'ts" that can help you take care of disks.

DO

- Put each disk away in its own envelope.
- Store disks in a box where they will be free from dust and out of the way when they are not being used.
- Carefully label each disk with a felt-tip pen so that you know what information is on it. Nothing is more confusing than having a whole pile of disks with no way to remember what was stored on each one.

DON'T

- Leave disks lying around out of their envelopes.
- Put disks on a radiator or in direct sunlight.
- Put disks near a magnet or electric motor.
- Write on a disk with a pencil or a ball-point pen.
- Throw, drop, or mangle a disk.

Appendix III
Reference List of Logo Commands Used in This Book

This appendix lists all the Logo commands used in this book. The listing for each command includes its name, short form, if any, examples showing how the command is used, and the page on which its use is described.

Section III.1. Turtle Commands

Page	Command	Short Form	Examples with Inputs
18	CLEARSCREEN	CS	
18	FORWARD	FD	FORWARD 20, FD 20
18	ACK	BK	BACK 10, BK 10
18	RIGHT	RT	RIGHT 90, RT 90
18	LEFT	LT	LEFT 30, LT 30
24	PENUP	PU	
24	PENDOWN	PD	
35	SETPC		SETPC 3
35	SETBG		SETBG 5
39	HIDETURTLE	HT	
39	SHOWTURTLE	ST	
40	WRAP		
40	FENCE		
41	WINDOW		
42	CLEAN		
42	HOME		
42	FULLSCREEN	**CTRL-L**	
42	SPLITSCREEN	**CTRL-S**	
43	TEXTSCREEN	**CTRL-T**	
154	HEADING		IF HEADING = 0 [STOP] PRINT HEADING
206	SETPOS		SETPOS [100 30], SETPOS [10 −20]
207	SETHEADING	SETH	SETHEADING 90, SETH 30
208	SETX		SETX 50, SETX −100
208	SETY		SETY 35, SETY −20
273	PEN		
273	SETPEN		
273	SHOWNP		
275	XCOR		PRINT XCOR, SETX XCOR + 20
275	YCOR		PRINT YCOR, SETY YCOR − 20

Section III.2. Editing and Filing Commands

Page	Command	Short Form	Examples with Inputs
58	TO		TO BOX
58	END		
59	EDIT	ED	EDIT "BOX, ED "BOX
59	ERASE	ER	ERASE "BOX, ER "BOX
65	PO		PO "BOX
65	POALL		
65	ERALL		
70	POTS		
70	SAVE		SAVE "CIRCLES
70	LOAD		LOAD "CIRCLES
70	CATALOG		
70	ERASEFILE		ERASEFILE "OLDSTUFF

Section III.3. Input, Output, and Printing Commands

Page	Command	Short Form	Examples with Inputs
16	PRINT	PR	PRINT [CATHY PERINI]
74	.PRINTER		.PRINTER 1, .PRINTER 0
180	READLIST	RL	MAKE "ANSWER READLIST
195	TYPE		TYPE [GUESS A NUMBER]
195	CLEARTEXT		
241	PADDLE		FORWARD PADDLE 0 PRINT PADDLE 1
241	BUTTONP		IF BUTTONP 1 [CLEARSCREEN]
277	KEYP		IF KEYP OUTPUT READCHAR
277	READCHAR	RC	MAKE "KEY READCHAR

Section III.4. Arithmetic and Number Commands

Page	Command	Short Form	Examples with Inputs
177	+		FORWARD :SIZE + 10, PRINT 5 + 3
177	−		PRINT 35 − 10 FORWARD :SIZE − 10
177	*		PRINT 3 * 5, FORWARD :SIZE * 3
177	/		PRINT 360 / 3, RIGHT 360 / 3
195	RANDOM		PRINT RANDOM 20
242	REMAINDER		PRINT REMAINDER 17 3
277	SQRT		PRINT SQRT 100
282	NUMBERP		IF (NOT NUMBERP :ANSWER) [OUTPUT READNUMBER]

Section III.5. Word, List, and Variable Commands

Page	Command	Short Form	Examples with Inputs
155	MAKE		MAKE "START HEADING MAKE "SIZE 50
179	WORD		PRINT WORD "HEL "LO PRINT (WORD "A "B "C)
179	SENTENCE	SE	PRINT SENTENCE [HELLO] [THERE] PRINT SENTENCE "HELLO [THERE] PRINT SENTENCE "HI "FRIEND PRINT (SE [HELLO] [MY] [FRIEND])
180	FIRST		PRINT FIRST "HELLO PRINT FIRST [HELLO THERE FRIEND]
180	BUTFIRST	BF	PRINT BUTFIRST "HELLO PRINT BF [HELLO THERE FRIEND]
180	LAST		PRINT LAST "HELLO PRINT LAST [HELLO MY FRIEND]
180	BUTLAST	BL	PRINT BUTLAST [HELLO MY FRIEND] PRINT BL "HELLO
231	THING		PRINT THING :PICT

Section III.6. Procedure Control and Conditional Commands

Page	Command	Short Form	Examples with Inputs
82	REPEAT		REPEAT 4 [FORWARD 20 RIGHT 90]
140	IF		IF :SIZE < 10 [STOP]
140	STOP		IF :SIZE < 10 [STOP]
140	>		IF :ANGLE > 90 [STOP]
144	=		IF :SIZE = 100 [STOP] PRINT 5 = 3 + 2
178	<		IF :SIZE < 10 [STOP]
192	AND		IF AND (YCOR > 0) (XCOR > 0) [STOP]
192	OR		IF OR (XCOR > 0) (YCOR < 0) [STOP]
200	TEST		TEST :ANSWER = 7
200	IFTRUE	IFT	IFTRUE [PRINT [HOORAY!]]
200	IFFALSE	IFF	IFFALSE [PRINT [SORRY]]
208	WAIT		WAIT 100
226	THROW		IF :COM = "E [THROW "TOPLEVEL]
247	OUTPUT	OP	OUTPUT "FALSE, OP LENGTH + 10
282	NOT		IF NOT (XCOR > 0) [STOP] IF (NOT :ANSWER) [OUTPUT READNUMBER]

Section III.7. Miscellaneous Commands

Page	Command	Short Form	Examples with Inputs
284	.DEPOSIT		.DEPOSIT 53007 7
284	CHAR		PRINT CHAR 17

Section III.8. Special Keys Used in Logo Command

Page	Key	Name	What it Does
13	←	Rubout	Erases a character to the left of the cursor
14	CTRL-B	Backspace	Moves the cursor back one space
14	→	Right-arrow	Moves the cursor ahead one space
14	REPT	Repeat	Repeats the previous key as long as it is held down
14	CTRL-G	Stop	Stops whatever Logo is doing
14	RESET	Reset	Crashes the system!
16	RETURN	Do-it	Sends Logo a typed command
42	CTRL-L	Fullscreen	Shows the complete turtle screen
42	CTRL-S	Splitscreen	Splits the screen between turtle and text
43	CTRL-T	Textscreen	Shows the complete text screen

Section III.9. Special Keys in Edit Mode

Keys to Move The Cursor

Page	Key	Name	What it Does
66	CTRL-B	Backspace	Moves the cursor *left* one space
66	→	Right-arrow	Moves the cursor *right* one space
66	CTRL-P	Up-arrow	Moves the cursor up to the *previous* line
66	CTRL-N	Down-arrow	Moves the cursor down to the *next* line
66	REPT	Repeat	*Repeats* the previous key, as long as you hold it down
68	CTRL-E	End-line	Moves the cursor to the *end* of a line
68	CTRL-A	Start-line	Moves the cursor to the *beginning* of a line
68	CTRL-V	Next-page	Moves the cursor *forward* one screenful of text
68	ESC V	Previous-page	Moves the cursor *back* one screenful of text

Keys to Change the Text

	Key	Name	What it Does
66	←	Rubout	Erases the character to the left of the cursor
66	RETURN	Return	Moves the cursor down to a new line. Any text to the right of the cursor is moved down with it
68	CTRL-D	Delete	Erases the character at the cursor
68	CTRL-K	Kill	Erases (kills) an entire line to the right of the cursor
68	CTRL-O	Open-line	*Opens* a new line at the cursor. Any text to the right of the cursor is moved down one line

Keys to Finish Editing

	Key	Name	What it Does
14	CTRL-G	Stop	Returns to Logo command mode without defining any procedures
60	CTRL-C	Define	Returns to Logo command mode and *defines* all procedures being edited

Index

Order your

Learning With Logo

and

Learning With Apple Logo

Procedures Disks

Be sure to order the disk that goes with your version of Logo. The *Learning With Logo Procedures Disk* can be used with Terrapin and Krell versions of Logo. The *Learning With Apple Logo Procedures Disk* can be used with Apple Logo, distributed by Apple Computer Company, Inc.

Be sure to specify catalog numbers: _____

90313–LEARNING WITH LOGO PROCEDURES DISK

90314–LEARNING WITH APPLE LOGO PROCEDURES DISK

To order a procedures disk for Terrapin/Krell Logo or Apple Logo, send your check for $15.95 per disk to Creative Publications, P.O. Box 10328, Palo Alto, CA 94303. If you prefer to order by Visa or MasterCard® include your signature, card number, and expiration date.

Computers
General Interest

Learning With Apple Logo®

by Daniel Watt

Learning With Apple Logo offers a thorough introduction to the fascinating uses of Logo, the educational language for children as well as adults. As rewarding for beginners as it is for more experienced programmers, this book teaches the fundamentals of Logo programming through projects that reveal the powerful ideas underlying the language developed at MIT.

The early chapters, written with 10- to 13-year-old readers in mind, start from the very beginning with an easy-to-follow guide to the Logo system, commands for controlling the Logo turtle, and instructions for writing new Logo commands. Dozens of suggested activities lead young learners to create their own unique projects, and special sections highlight powerful ideas and warn against common pitfalls.

Building on these basics, more involved projects including interactive games, quiz programs, and language activities with words and sentences are introduced as the book goes on. The detailed information on poem-generating programs, animation, and special tool procedures will continue to challenge more advanced learners as their knowledge increases.

While most of *Learning With Apple Logo* can be read and used by children, specific "helper's hints" throughout the book are especially designed for teachers and parents who want to help children learn Logo. In these sections, the author offers more detailed information and many helpful teaching suggestions drawn from his broad experience as an educator and Logo researcher.

Comprehensive appendices tell how to use the book with Terrapin and Krell Logo and TI Logo. A disk of procedures used in the book is also available. (Ordering information is found inside.)

Learning With Apple Logo is designed to be used with Logo for the Apple Logo as marketed by Apple Computer Inc. Users of Logo for the Apple II sold by Terrapin Inc. and Krell Software should ask for the *Learning With Logo* edition.

Photo by Jock Gill.

Daniel H. Watt, an editor with *BYTE* and *Popular Computing* magazines, has been an elementary school teacher, curriculum developer, and teacher trainer. For five years he was a researcher with the MIT Logo Group, teaching Logo to children and teachers and conducting research about what students learn when they program computers.

McGraw-Hill Book Company
Serving the Need for Knowledge
1221 Avenue of the Americas
New York, NY 10020

ISBN 0-07-068571-1