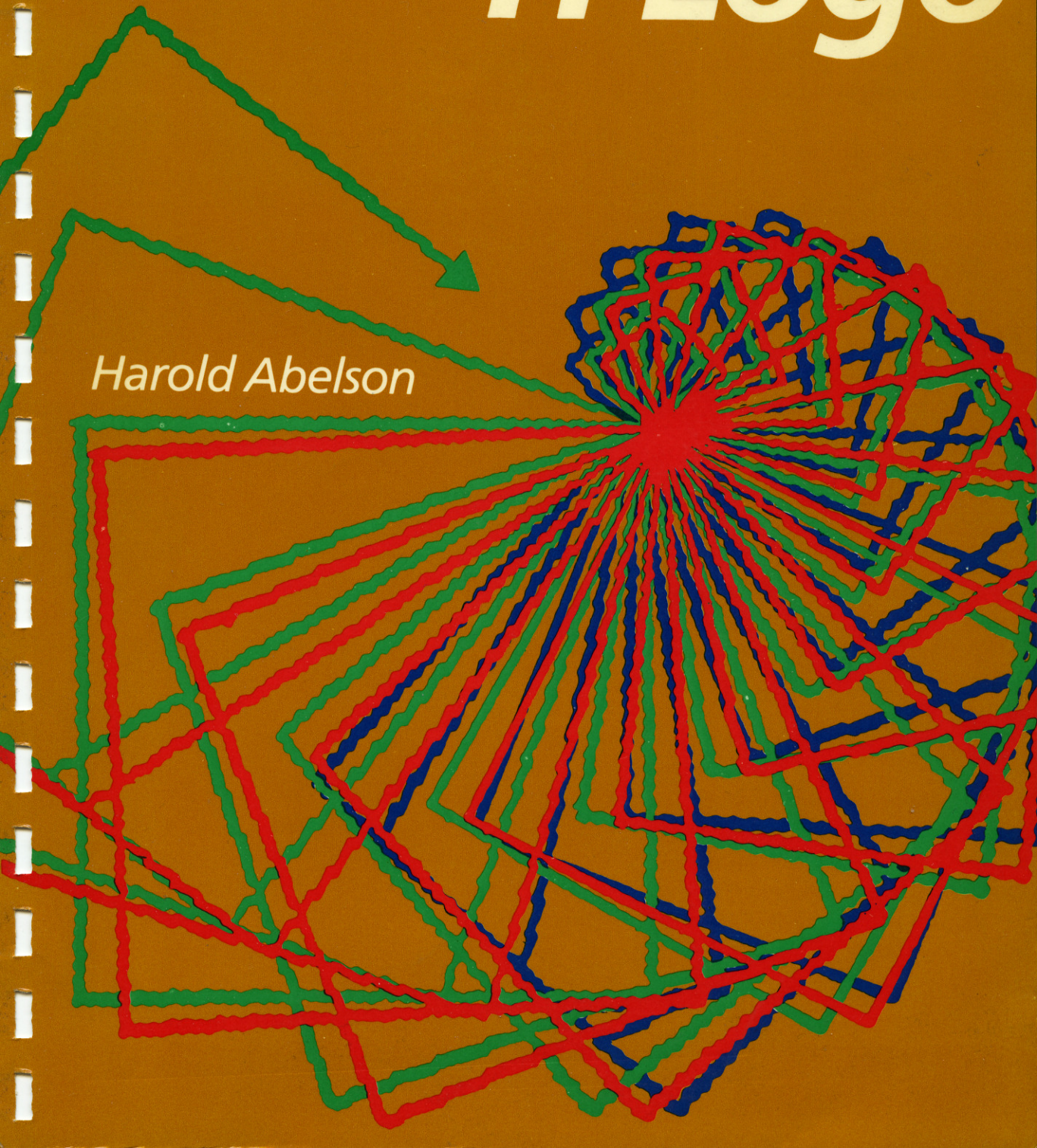
A BYTE BOOK

>$17.95

TI Logo



Harold Abelson

TI Logo

by Harold Abelson

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

To my parents, Anne and Benjamin Abelson

Library of Congress Cataloging in Publication Data

Abelson, Harold.
TI Logo.

Bibliography: p.
Includes index.
1. TI 99/4A (Computer) — Programming. 2. LOGO (Computer program language) I. Title. II. Title: T.I. Logo.
QA76.8.T133A23 1984 001.64'24 83-19608
ISBN 0-07-038459-2

Copyright © 1984 by McGraw-Hill, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher.

1234567890 HAL/HAL 8987654

Printed and bound by Halliday Lithograph

Contents

Introduction

Logo is the name for a philosophy of education and for a continually evolving family of computer languages that aid its realization. Its learning environments articulate the principle that giving people personal control over powerful computational resources can enable them to establish intimate contact with profound ideas from science, from mathematics, and from the art of intellectual model building. Its computer languages are designed to transform computers into flexible tools to aid in learning, in playing, and in exploring.

Logo's designers are guided by the vision of an educational tool with no threshold and no ceiling. We try to make it possible for young children to control the computer in self-directed ways, even at their very first exposure to Logo. At the same time, we believe that Logo should be a general-purpose programming system of considerable power and wealth of expression. In fact, we regard these two goals as complementary rather than conflicting, since it is the very lack of expressive power of primitive languages such as BASIC that makes it difficult for beginners to write simple programs that do interesting things. More than 10 years of experience at MIT and elsewhere have demonstrated that people across the whole range of "mathematical aptitude" enjoy using Logo to create original and sophisticated programs. Logo has been successfully and productively used by preschool, elementary, junior high, high school, and college students, and by their teachers.

Some of the important features of Logo are:

- Logo is a *procedural* language. Logo programs are created by combining commands into groups called procedures and by using these procedures as steps in other procedures, and so on to arbitrary levels of complexity. Each individual step of a procedure may be any primitive Logo command or any user-defined procedure. Procedures can communicate among themselves via *inputs* and *outputs*.

- Logo is an *interactive* programming language. Any Logo command, whether built into the language or defined as a procedure, can be executed by simply typing the command at the keyboard. Logo's integrated editor makes it easy to define, execute, and modify procedures, because there is no necessity to deal with separate compilers, loaders, monitors, and so forth.

- Logo's data objects (those things that can be named by individual variables, passed directly as inputs to procedures, and returned as values) include not only numbers and character strings, but also compound structures called *lists*. Many computer languages force the programmer to manipulate data structures in terms of sequences of operations on individual numbers and

character strings. In contrast, Logo's lists are functional units that can be transformed in single operations, making Logo a convenient and powerful language for applications involving symbol manipulation. Moreover, the fact that Logo procedures can themselves be represented and manipulated as lists means that users can attain considerable direct control over the way commands are interpreted—for example, to provide special interfaces to Logo for the physically handicapped or the very young.[1]

Another important aspect of Logo is its incorporation of a programming area called *turtle geometry.* A turtle is a computer-controlled "cybernetic animal" that lives on the display screen and responds to Logo commands that make it move (FORWARD or BACK) and rotate (LEFT or RIGHT). As the turtle moves, it leaves a trace of its path and in this way can be used to make drawings on the display screen. For example, the following Logo procedure tells Logo how to make the turtle draw a square by repeating four times the commands "go FORWARD 100 units, turn RIGHT 90 degrees":

```
TO SQUARE
REPEAT 4 [FORWARD 100 RIGHT 90]
END
```

Turtle graphics is highly successful, both as an introduction to programming for people of all ages and also as a foundation for a computer-based mathematics curriculum. In this book, we use turtle graphics to introduce the basic ideas of Logo programming, although we also cover other aspects of the language.[2]

[1]Section 11.2.1 gives an example of such an interface. The implementation makes use of the fact that it is possible to write Logo procedures that themselves define procedures. The book by Goldenberg [10] describes work during 1976 and 1977 using Logo with physically and emotionally handicapped children. More recent research in this area is discussed in the article by Weir [17].

[2]In his book *Mindstorms* [5], Papert discusses the turtle as exemplary of the kind of computational "object to think with" through which technology can lead to fundamental educational change. *Mindstorms* also discusses the Logo philosophy of education and the role of computer technology in transforming education. The book by Abelson and diSessa [1] uses turtle geometry as the basis for exploring in mathematics and presents extended treatments of mathematical topics ranging from elementary geometry through General Relativity. Although turtle geometry originated as a part of the Logo language, its use is not restricted to Logo. Other languages that have incorporated turtle graphics are Smalltalk (Kay [13] and Goldberg [9]) and UCSD Pascal (Bowles[5]).

TI Logo

Since its creation in 1968, Logo has been under continual development.[3] As Logo is a complex and sophisticated language, most Logo work during the 1970s was conducted using large research computer systems. It is only recently that computers capable of supporting Logo have become inexpensive enough for widespread use in schools and homes. In 1979 the MIT Logo Group and Texas Instruments began a joint effort to develop an implementation of Logo for the TI home computer.[4] The resulting TI Logo system, which runs on the TI 99/4 and 99/4A computers is a powerful yet easy-to-use programming language, which incorporates all the aspects of Logo mentioned above. In addition, TI Logo includes the following special features:

- TI Logo makes it easy for even very young children to create spectacular animation effects through the use of *sprites*. Sprites, like turtles, are "creatures" that live on the display screen. But unlike ordinary Logo turtles, sprites can change their color and shape, and can move across the screen smoothly and continuously under program control. Even more than turtle graphics, sprites and animation provide an exciting area in which beginners can experiment with the power of computation.
- TI Logo II includes commands for generating *music* with up to three voices plus a "drum." When combined with the power of Logo procedures and lists, this makes it easy to write programs that play tunes and harmonies. Moreover, by synchronizing music and sprite graphics, even beginning programmers can create animated movies with musical accompaniment.

A guide to this book

This book is an introduction to the Logo system and to programming in Logo.[5] You should think about learning Logo in three stages. The first stage, covered in Chapters 1 and 2, includes the basics of defining procedures and using turtle graphics to draw pictures on the display screen. Chapter 3 consists of suggestions for programming projects based on this material. Chapter 4 introduces sprites and animation, and shows how to write simple programs for making movies. Chapter 5 describes the mechanics of keeping

[3]Logo was initially developed in 1968 as part of a National Science Foundation sponsored research project conducted at Bolt, Beranek & Newman, Inc., in Cambridge, Massachusetts (Feurzeig, *et. al.* [7]). The majority of Logo work since then has been conducted at MIT in the Artificial Intelligence Laboratory and the Division for Study and Research in Education, but there has also been significant continuing work at BBN (Feurzeig, *et. al.* [8]), at the University of Edinburgh (Howe, *et. al.* [12]), and at a number of other universities throughout the world.

[4]The TI Logo project implementation project at MIT was carried out under the supervision of Seymour Papert. Principal contributors to this effort were Gary Drescher, Edward Hardebeck, Mark Gross, Leigh Klotz, John Berlow, Ralph Payne, Maxine Bobco, Sid Nolte, Richard Tarrent, John D'Angelo, Al Riccomi, and Wyatt Dodd.

[5]There are currently two releases of TI Logo. The major difference between them is that TI Logo II includes commands for generating music while TI Logo I does not. This manual can be used with either version.

track of procedures and saving them in files. The next stage in learning Logo includes writing procedures that use data—numbers, words, and lists as introduced in Chapter 6—to carry out projects such as the ones presented in Chapters 7 and 8, and also for using the Logo music system, which is described in Chapter 9. The last section of Chapter 6 also discusses some fine points of Logo syntax, which are mostly ignored in the first five chapters. Chapters 10 and 11 cover advanced topics in Logo programming, including using recursion to deal with words and lists and using lists to represent complex data structures. Chapter 12 is a reference that describes the primitive commands included in the TI Logo system.

Acknowledgements

The examples included in this book draw upon research conducted over the past 10 years by members of the Logo Group at the MIT Division for Study and Research in Education and the MIT Artificial Intelligence Laboratory. Section 7.3 reprints a 1970 AI Lab Memo by Seymour Papert and Cynthia Solomon. The projects in Chapter 3 come from material prepared by Dan Watt as part of a teaching experiment conducted in the Brookline, MA, elementary school system which is documented in [6]. I would like to thank Greg Gargarian for help in assembling this chapter. The Dynaturtle project in Chapter 8 is based on work by Andy diSessa and Dan Watt. The music system draws on numerous ideas and experimental systems developed by Jeanne Bamberger. I would also like to thank Dan Watt, Leigh Klotz, Nola Sheffer, and Richard Carter for comments on previous drafts of this book.

NOTICE

P-Code Card

If you are using the Peripheral Expansion Box with the P-Code card inserted, turn the P-Code OFF or remove the card. If you do a SAVE, RECALL, or PRINTOUT with the P-Code card turned on, the computer goes through the console power-up code and into the UCSD p-System.*

SPRITE Wrap-around

Unlike the first version of TI LOGO, where SPRITE 50 would automatically default to SPRITE 18, LOGO II attends only to sprites addressed as 0 through 31. Sprites can be addressed only as SPRITE 0 through SPRITE 31 in LOGO II. Do not call a sprite beyond the number 31.

*UCSD p-System is a trademark of the Regents of the University of California.

CHAPTER 1

A First Look at Logo

This chapter introduces the basic mechanics of using Logo. It describes how to execute simple commands and how to define and edit procedures. The examples are given in terms of using turtle graphics to draw pictures on the screen. Even though we do not, at this point, introduce more than a few commands or attempt a full explanation of the rules for writing programs, the material in this chapter and the next is sufficient to allow you to use Logo for a wide variety of interesting projects such as the ones described in Chapter 3. Try to work through this chapter at the computer keyboard, experimenting with the different features as they are introduced.

1.1. The Computer Keyboard

If you have never used a computer before, you will need to become accustomed to a few idiosyncrasies of computer keyboards as compared with typewriter keyboards. Be careful not to type the numeral 0 in place of the letter O, or the numeral 1 in place of the letter l. These may look alike to a person, but the keys generate different signals for the computer to interpret. TI Logo uses uppercase letters only, so do not worry about using the SHIFT key for typing letters. There are, however, a few symbols that are typed using the SHIFT key. For example, the asterisk symbol * appears as SHIFT-8, just as on an ordinary typewriter keyboard. To type *, hold down the SHIFT key and press the 8 key (rather than trying to press both SHIFT and 8 simultaneously).

Computer keyboards generally include a few keys not ordinarily found on typewriters. The key marked ENTER is used in Logo to signal the computer to process a command line that has been typed.

The FCTN (function) key on the 99/4A is used like an alternate shift key to obtain the symbols that are marked on the front of various keys. For instance, the open bracket character [appears on the front of the R key, so to type a [on the 99/4A, hold down the FCTN key and press R. Throughout this book, we specify function characters by the prefix "FCTN," as in "FCTN-R." Other FCTN symbols that are used in Logo programming are closed bracket], double quote ", and the four arrow keys ←, →, ↑, ↓.

Logo also makes use of special symbols called DEL, ERASE, CLEAR, BEGIN, PROC'D, AID, BACK, and QUIT. On the TI-99/4A, these are typed using FCTN, together with the keys on the top row of the keyboard. The special symbol

names are marked on the plastic strip that is supplied with the TI-99/4A. Figure 1.1 shows a diagram of the keyboard on the 99/4A with indications of the special keys used with Logo.[1]

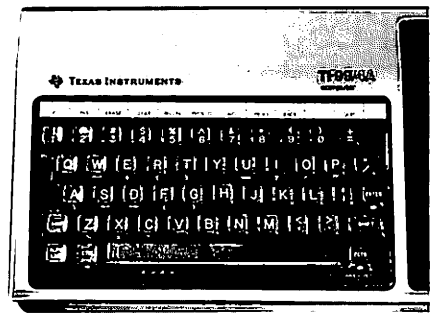
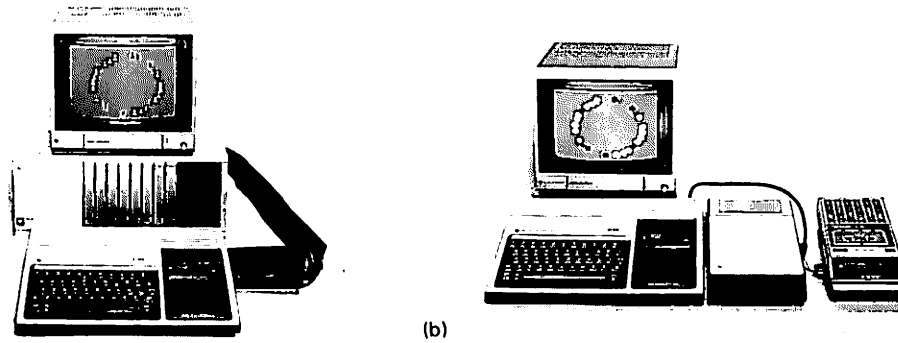


Figure 1.1: The TI-99/4A keyboard with special keys indicated.

On the TI-99/4, which has no FCTN key, the additional FCTN symbols are typed in alternative ways, using SHIFT. [Appendix A gives a complete list of the special symbols used by Logo and the key sequences required to type them on both the 99/4 and the 99/4A.]

1.2. Preparing to Use Logo

The TI Logo system operates on the TI-99/4 and TI-99/4A home computers. In addition to the computer and the Logo cartridge, the system requires a TI Memory Expansion Unit or a Peripheral Expansion Box with a 32K memory expansion card. If you wish to save your work on a diskette, you must also attach a TI Disk Memory System. Alternatively, you can save your work on cassette tape by attaching a cassette recorder. If your TI computer system has an attached printer, you can use this to produce printed copies of your work. Figure 1.2 shows two 99/4A systems configured to run



(a) (b)

Figure 1.2: A TI-99/4A system configured to run Logo.

[1] One important point to keep in mind when using Logo is to *never* press the QUIT key unless you are done using Logo. Pressing QUIT resets the computer and erases all programs and data from memory. On the 99/4A, be especially careful when you type the symbol + (SHIFT – =) to be sure that you do not mistakenly type QUIT(FCTN – =) instead.

TI Logo. The first shows a Peripheral Expansion Box with a disk drive. The second shows a Memory Expansion Unit with a cassette recorder.

Powering Up

If you are using the Disk Memory System, be sure to follow these steps in powering up the system:

1. First turn on the disk controller and disk drive(s).
2. Next turn on the Peripheral Expansion Box or the Memory Expansion unit.
3. Then turn on the computer console and any other devices.
4. Turn on the computer console and TV monitor last.

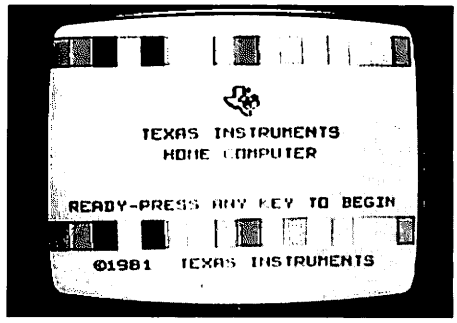
You must follow these steps in order, or the computer will not be able to access the disk system. In this case, you must turn the power off and power up the devices in the correct order.

Starting Logo

With the system powered up and the Logo cartridge inserted into the slot on the computer console, you will see the master title display on the computer screen.[2] Press any key, and you will obtain a menu of available system choices. Press the number next to TI Logo or TI Logo II. Logo will start after a pause of about 5 seconds.

1.3. Using Logo Commands

Figure 1.3 is a photograph of the display screen as it appears when Logo is first started. The system prints a welcome message followed by a line beginning with a question mark. The question mark, called a *prompt*, indicates that Logo is waiting for you to give it a command. Just to the right

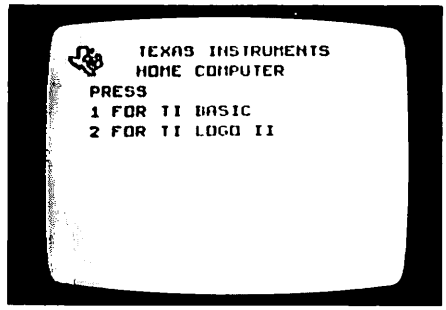


(a)

Figure 1.3: The display screen as it appears when Logo is first started.

[2]You can insert the Logo cartridge either before or after powering up the system. There is an automatic reset feature built into the computer so that the system will return to the master title display whenever a cartridge is inserted into the console. If you want to remove the cartridge from the console, it is best to *first* return the computer to the master title screen by pressing QUIT.

(b)



(c)

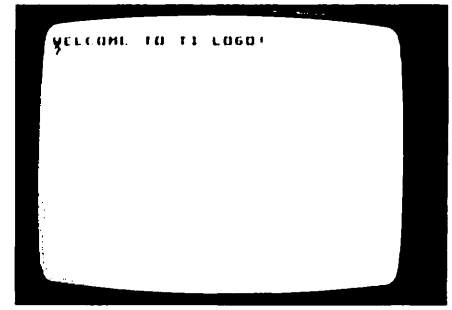


Figure 1.3: (Continued)

of the prompt is a black flashing symbol called a *cursor.* The cursor indicates the position at which the characters you type will appear on the screen.

To give Logo a command, type the command and press the ENTER key. For instance, to tell Logo to print the product of 37 and 67, you type the command line

```
PRINT 37 * 67
```

That is, you type the keys P, R, I, N, T, space, 3, 7, space, *, space, 6, 7, ENTER. The computer then prints 2479, followed by a new line with a question mark prompt, indicating readiness to accept a new command. Bear in mind that when you type a command line, it is not executed until you press the ENTER key. To tell Logo to print the message "Logo is a language," you type the command line

```
PRINT [LOGO IS A LANGUAGE]
```

followed by ENTER. This example illustrates how square brackets are used in Logo to group words into *lists.*[3] You can use lists in this way to print messages on the screen, but there are many other uses for lists in Logo, and we will study these in detail in Chapter 6.

The spaces in these command lines are important, because they indicate to Logo how the line is to be broken into its component parts.[4] If you type the first command line omitting the space between the T and the 3 as follows:

```
PRINT37 * 67
```

---

[3]The open and closed brackets are typed on the 99/4A as FCTN-R and FCTN-T, respectively. On the 99/4, they are typed as SHIFT-4 and SHIFT-5.

[4]Logo has some knowledge about where it is reasonable to divide lines into component parts, even when they are not separated by spaces. For example, it knows enough to interpret the string of 5 characters 37 * 67 as containing three elements: the number 37, the symbol *, and the number 67. However, it is a good habit to always use spaces to separate the elements of command lines, even when this is not strictly necessary. The rules that determine exactly where spaces are necessary are discussed in Section 6.7.

then Logo will think you are telling it to execute a command named PRINT37 and complain that it does not know how to do this, by responding with the error message:

TELL ME HOW TO PRINT37
?PRINT 37 * 67
2479
?PRINT [LOGO IS A LANGUAGE]
LOGO IS A LANGUAGE
?PRINT37 * 67
TELL ME HOW TO PRINT37

Figure 1.4 shows a photograph of the screen as it appears after you have given the three command lines described above, along with the computer's responses to each line. The question mark shown at the beginning of each command line is the prompt typed by Logo, and the rest of the line is the command typed by the user. In this book, when we want to emphasize the difference between the characters that you type and the characters that Logo types, we print the latter characters in italics. For example, the first command interaction in Figure 1.4 would be printed as

?PRINT 37 * 67
2479

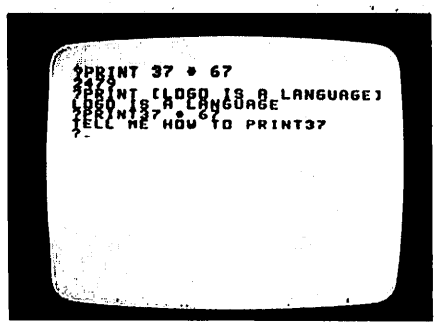


Figure 1.4: Three command lines typed to Logo, and the system's responses.

In later chapters, we will see how to write Logo programs that manipulate numbers and text. But we begin our study of Logo by investigating how to use the computer to produce drawings on the display screen by issuing commands to a "creature" known as a *turtle*. To set up the screen for drawing, type

TELL TURTLE

and press ENTER. The screen should now appear as shown in Figure 1.5, with the entire screen blank, except for a small triangle in the center and a question mark near the bottom. The question mark, as before, is the prompt indicating that Logo is ready to accept a command. When drawing, Logo reserves the six lines at the bottom of the screen for your typed commands and the computer's typed responses. The rest of the screen is for drawings. When Logo is used in this way to draw pictures on the screen, the system is said to be in *turtle mode*. The original screen arrangement with no space reserved for graphics, as shown in Figures 1.3 and 1.4, is called *noturtle mode*.[5] Whenever Logo is in turtle mode, you can make it return to noturtle mode by giving the command NOTURTLE.

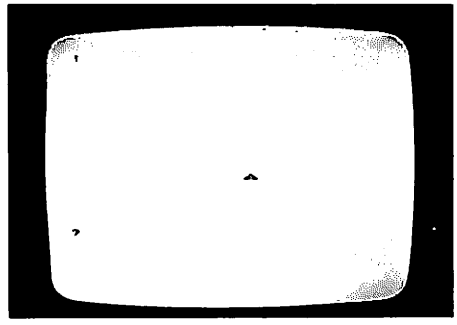


Figure 1.5: Appearance of the display screen when Logo enters turtle mode.

1.3.1. Basic Turtle Commands

The turtle is the triangular pointer that appears at the center of the screen when Logo enters turtle mode. You make drawings by telling the turtle to move and to leave a trace of its trail. There are four basic commands for moving the turtle. The commands FORWARD and BACK make the turtle move along the direction it is pointing. Each time you give a FORWARD or BACK command, you must also specify a number that indicates how far the turtle should move. The commands RIGHT and LEFT cause the turtle to rotate. RIGHT and LEFT each require you to specify the amount of rotation in degrees. Try typing the following sequence of Logo commands:

```
RIGHT 45
FORWARD 100
LEFT 135
FORWARD 150
```

[5] In noturtle mode there are 24 lines for typing. A more complete explanation of the different modes in which the Logo system operates is given in Section 1.6.

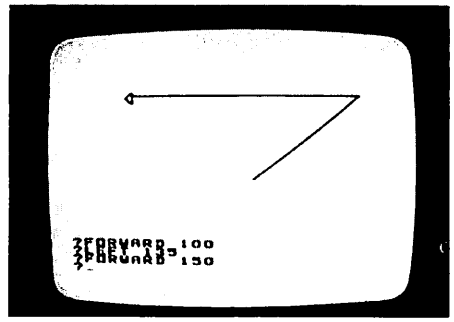


Figure 1.6: Photograph of the display screen showing a simple sequence turtle commands.

This should produce the wedge-shaped drawing shown in Figure 1.6. Remember to terminate each command line with ENTER and to include a space between the command word and the number. If you mistype a character, you can delete the character by pressing ERASE.[6] See Section 1.3.2 for more details on correcting typing errors.

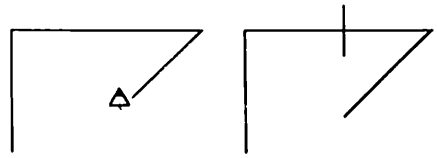
The number following the command is called an *input*. FORWARD, BACK, LEFT, and RIGHT each need one input. Logo commands may or may not require inputs, depending on the command. CLEARSCREEN is an example of a command that takes no input. Later on we will see examples of commands that require more than one input.

If you want to move the turtle without drawing a line, give the PENUP command. Subsequent FORWARD and BACK commands will now make the turtle move without leaving a trail. To resume drawing, give the PENDOWN command. Neither PENUP nor PENDOWN takes an input. The HIDETURTLE command causes the turtle pointer to disappear, although the turtle is still "there" and will draw lines if the pen is down. SHOWTURTLE makes the pointer reappear. Figure 1.7 illustrates the use of these commands to draw a simple picture.

If you want to start over and draw a new picture, you can use the CLEARSCREEN command. This erases the screen and restores the turtle to its initial location at the center of the screen, pointing straight up.[7]

[6] ERASE is typed as FCTN-3 on the 99/4A and as SHIFT-T on the 99/4.

[7] CLEARSCREEN can also be used in noturtle mode to clear the screen and return the cursor to the upper left-hand corner.



BACK 50
LEFT 45

PENUP
FORWARD 25
PENDOWN
FORWARD 25
HIDETURTLE

Figure 1.7: Drawing with the turtle.

1.3.2. Correcting Typing Errors

As you type Logo commands, you will undoubtedly make a few typing errors. Common errors include omitting characters, typing extra or wrong characters, and transposing characters. To correct typing errors, use ERASE. Each time you press ERASE, the character immediately to the left of the cursor is erased, and the cursor moves one space to the left. For example, if you typed

```
FORWXYD 100
```

when you meant to type

```
FORWARD 100
```

you can correct the error by pressing ERASE 7 times to erase back to the W and then retyping the rest of the line.

1.3.3. Error Messages

If Logo cannot execute the input line, it replies with an error message. Logo's error messages attempt to be helpful in describing what went wrong. For example, if you try to execute the command line

```
PRINT 3 +
```

Logo will reply

TELL ME MORE

because it expects to find something more on the line after the + to be added to 3. Another common error message is the result of attempting to use a command that has not been defined. For instance, if you try to execute

```
TURN 100
```

Logo will respond

TELL ME HOW TO TURN

unless you have first defined a procedure named TURN.[8] The *TELL ME HOW TO* error message often occurs as a result of a typing error. For example, if you type an input line like

```
FORWARD100
```

omitting the space between the D and the 1, Logo responds

TELL ME HOW TO FORWARD100

because Logo reads the entire line as a single word, which it assumes is supposed to be the name of a procedure.

When Logo responds to your command with an error message, you should try to determine the reason for the error. Sometimes it is a simple typing error. If so, you can retype the line. Alternatively, the reason for the error may be hidden deep in the design of one of your programs. The activity of rooting out and repairing errors in programs is called *debugging*, and Logo provides debugging aids to make this task easier. These are described in Section 5.3.

1.3.4. Practice With Commands

If this is your first exposure to Logo, it would be a good idea to review the material covered so far by drawing some figures using the turtle commands. Try to understand any error messages that occur. Following are some things to note in your exploring.

Wraparound

The turtle screen is 240 "turtle steps" wide by 144 steps high. If you give a command that moves the turtle outside this range, the turtle wraps around to appear at the opposite edge of the screen. That is to say, driving the turtle off the top of the screen makes it reappear at the bottom of the screen and continue drawing. Driving the turtle off the right edge of the screen makes it reappear at the left of the screen, and so on.

[8] Section 1.4 explains how to define procedures.

Out of Ink

After you have drawn a large number of lines on the screen, Logo may signal the error message

OUT OF INK

This indicates that the turtle's capacity for drawing has been used up, and it cannot draw any additional lines. At this point, you must clear the screen if you want to continue drawing.[9]

Abbreviations

Some of the commonly used Logo commands have abbreviations to help you save typing. Abbreviations for some of the commands we have seen so far are

FORWARD	FD
BACK	BK
RIGHT	RT
LEFT	LT
PENUP	PU
PENDOWN	PD
HIDETURTLE	HT
SHOWTURTLE	ST
CLEARSCREEN	CS

Multiple Commands on a Line

There is no restriction that each line be only a single Logo command. If you like, you can execute lines like

```
FORWARD 10 PENUP FORWARD 10
```

Logo will execute the separate commands in order, from left to right. If some command on the line causes an error, Logo will execute the commands up until the point of the error before typing an error message. However, single lines that contain many separate commands can be confusing, and it is generally better to use only one command per line.

[9]The limited drawing capacity is a consequence of the way that turtle lines are implemented using *tile graphics*. We will discuss this in Section 4.3.3.

The REPEAT Command

One useful addition to your repertoire of Logo commands is REPEAT. REPEAT takes two inputs—a number and a list of commands—and repeats the commands in the list the designated number of times. For example,

```
REPEAT 4 [FORWARD 30 RIGHT 90]
```

makes the turtle draw a square. Notice that the list of commands is enclosed in square brackets.[10] This is a very simple example of how lists are used in Logo to group things. Lists are introduced in Section 6.4.

REPEATs can be nested. For a pretty effect, try

```
REPEAT 10 [REPEAT 4 [FORWARD 30 RIGHT 90] RIGHT 36]
```

which produces the drawing shown in Figure 1.8. Playing with nested REPEATs can be fun, but in terms of program clarity and power, it is much better to combine commands by defining *procedures*, as we describe in Section 1.4.

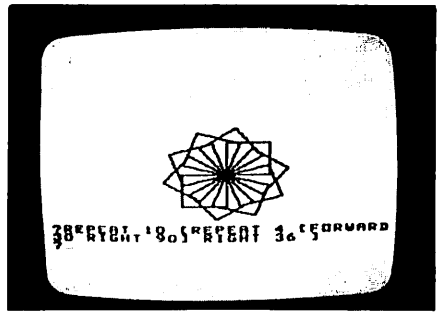


Figure 1.8: Using nested REPEATs to produce a complex drawing.

Long Command Lines

Lines on the display screen can be at most 30 characters long. Figure 1.8 illustrates how Logo treats command lines that are longer than 30 characters. When you type the 31st character of a command line, Logo will move the cursor to the next screen line, at which point you can continue typing. To execute a long line, you type ENTER as usual. Even with this multiple line capability, no input line may be longer than 127 characters. Logo will refuse to insert more than this many characters in a command line.

In this book, long command lines are not shown as they appear on the screen. Instead they are indented to make them easier to read. When you type in the program examples in the book, continue typing the indented portions as part of one long line, as shown in Figure 1.8.

---

[10] Be sure to use square brackets [], not parentheses (), for lists. You type [] by pressing FCTN-R and FCTN-T.

Stopping Execution With the BACK Key

When Logo is executing a command, pressing the BACK key (FCTN-9) causes it to stop whatever it is doing and wait for a new command. Logo types

STOPPED

followed by the question mark prompt. For example, if you should start Logo executing some long process like

```
REPEAT 10000 [PRINT 1]
```

and then think better of it, you can halt it by pressing BACK. *Be sure to use* BACK *rather than* QUIT *to halt a Logo program.*[11]

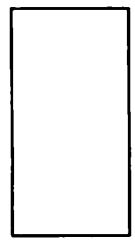
1.4. Introduction to Procedures

You can regard Logo commands like FORWARD, PRINT, and so on, as words that the computer understands when the Logo system is started. These "built-in" words are called *primitives*. One of the most important things about the Logo language is that it makes it easy for you to teach the computer *new* words. Once you define a new word, it becomes part of the computer's working vocabulary and can be used just as if it were a primitive. You teach Logo new words by defining them in terms of words that are already known. These definitions are called *procedures*, and this section describes the simple mechanics of how to define and edit procedures. As in the previous section, the examples are drawn from turtle graphics programs.

1.4.1. Simple Procedures

The following sequence of commands makes the turtle draw a rectangular box as shown in Figure 1.9:

(a)



```
FORWARD 40
RIGHT 90
FORWARD 20
RIGHT 90
FORWARD 40
RIGHT 90
FORWARD 20
```

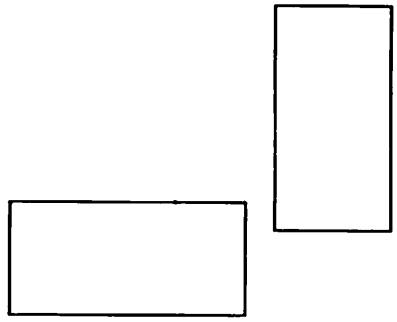
```
TO BOX
FORWARD 40
RIGHT 90
FORWARD 20
RIGHT 90
```

```
FORWARD 40
RIGHT 90
FORWARD 20
END
```

Figure 1.9: Shapes drawn by the BOX, BOXES, and PINWHEEL procedures.

[11] Pressing QUIT resets the computer and destroys all stored data.

You can teach the computer to execute this sequence of commands whenever you give the command BOX by defining BOX as a procedure:



```
TO BOX
FORWARD 40
RIGHT 90
FORWARD 20
RIGHT 90
FORWARD 40
RIGHT 90
FORWARD 20
END
```

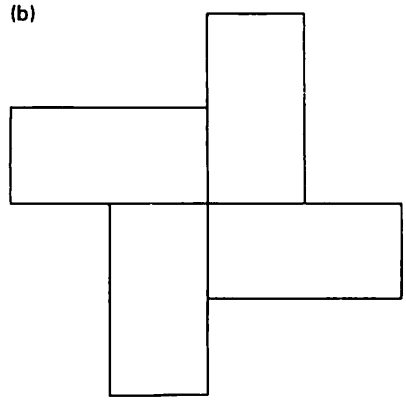
```
TO BOXES
BOX
PENUP
FORWARD 5
LEFT 90
FORWARD 15
RIGHT 90
PENDOWN
BOX
END
```

Before typing this definition to Logo, you will need to know about the Logo *procedure editor* which is described below in Section 1.4.2. Notice first that the format of the procedure definition is

- A *title line*, which consists of the word TO followed by the name you choose for the procedure.
- A *body*, which is the sequence of command lines that make up the definition.
- The word END to indicate that this is the end of the definition.

Once BOX is defined, it can now be used in further definitions, such as

(b)



```
TO PINWHEEL
REPEAT 4 [BOX]
END
```

```
TO BOXES
BOX
PENUP
FORWARD 5
LEFT 90
FORWARD 15
RIGHT 90
PENDOWN
BOX
END
```

or

Figure 1.9: (Continued)

```
TO PINWHEEL
REPEAT 4 [BOX]
END
```

which produce the drawings shown in Figure 1.9. When a procedure is used as part of the definition of a new procedure, it is referred to as a *subprocedure* of the new procedure.

Remember that once a procedure is defined, you can consider it to be just another word that the computer "knows." You tell Logo to execute any of these procedures in the same way that you tell it to execute a primitive command—by typing the name of the command followed by ENTER.

1.4.2. Defining Procedures

Procedure definitions like the ones in the previous section are typed into the Logo system using a *procedure editor.* The following paragraphs describe how to define procedures such as the BOX procedure shown above. When Logo gives its question mark prompt, you type

```
TO BOX
```

and press ENTER. The screen should now be clear, except for a procedure title line TO BOX, followed by an END. The screen background also changes color to a light green to indicate that you are now using the procedure editor, or, are in so-called edit mode. This configuration is shown in Figure 1.10.

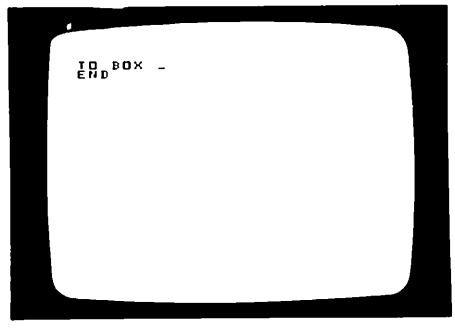


Figure 1.10: The display screen as it appears when you enter edit mode by typing TO BOX.

The Procedure Editor

In edit mode, you type in the procedure definition line by line. The major difference between typing at the procedure editor and typing regular Logo commands is that pressing ENTER merely moves the cursor to the beginning of the next line, rather than telling Logo to execute the current line as a command. Logo is now *storing* your command lines as part of the procedure, rather than executing them.

After you have typed in the procedure definition, you press BACK. The definition will be processed and Logo will be ready to accept a new command.

Editing Commands

When you type your definitions into the procedure editor, you can type characters and use ERASE to correct typing errors as usual. There are also a large number of more powerful *editing commands* to aid you in typing and changing procedure definitions.

ERASE	FCTN-3	Pressing the ERASE key, just as at Logo command level, deletes the character to the left of the cursor and moves the cursor one space to the left. In addition, if the cursor is at the beginning of the line, pressing ERASE combines that line with the previous line.
DEL	FCTN-1	Pressing the DEL key deletes the character at the current cursor position, that is, the character over which the cursor is flashing. In addition, if the cursor is at the right end of the line, pressing DEL combines that line with the next line.
Arrow keys	FCTN-E FCTN-S FCTN-D FCTN-F	Pressing any of the arrow keys (up, down, right, or left) moves the cursor one space in the direction of the arrow *without* rubbing out any character.
BEGIN	FCTN-5	Pressing the BEGIN key moves the cursor to the beginning of the line.
PROC'D	FCTN-6	Pressing the PROC'D key moves the cursor to the right end of the line.
CLEAR	FCTN-4	Pressing the CLEAR deletes all characters on the line from the cursor rightwards.

For example, to change the line

```
FORWXYD 100
```

to

```
FORWARD 100
```

start with the cursor just to the right of the number 100. Then you can position the cursor under the X by pressing left arrow 7 times, then delete the X and the Y by pressing DEL twice, and then type the characters AR. Another

way to make the same change is to position the cursor under the D by pressing the left arrow key 5 times, then delete the X and the Y by pressing ERASE twice, and then type AR.

If you use an editing key in a context where it doesn't make sense (for instance, trying to move to a nonexistent line), Logo will flash the screen briefly.

Changing Procedure Definitions

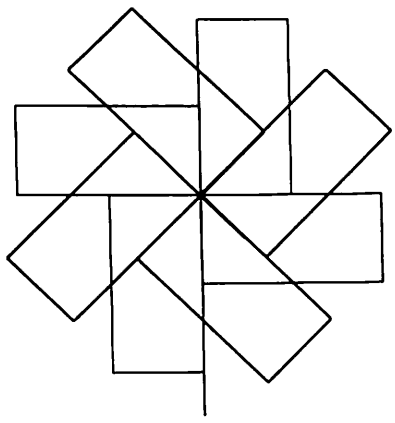
Suppose you want to change the definition of a procedure. For example, you may want to change the definition of PINWHEEL on page 9 from

```
TO PINWHEEL
REPEAT 4 [BOX]
END
```

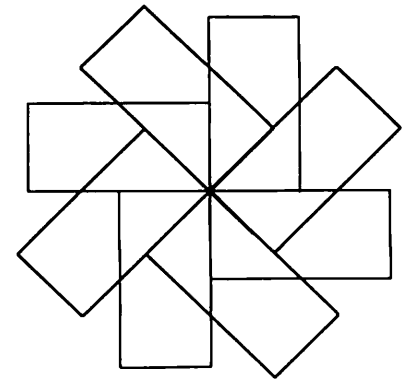
to

```
TO PINWHEEL
FORWARD 50
REPEAT 8 [RIGHT 45 BOX]
BACK 90
END
```

so that it now makes the drawing shown in Figure 1.11. To accomplish this, you give the command



```
TO PINWHEEL
FORWARD 50
REPEAT 8 [RIGHT 45 BOX]
BACK 90
END
```



```
TO FAN
REPEAT 8 [RIGHT 45 BOX]
END
```

Figure 1.11: Shapes drawn by the modified PINWHEEL and FAN procedures.

```
EDIT PINWHEEL
```

Logo now places you in edit mode with the original text of the PINWHEEL procedure shown on the screen. Now edit the definition, inserting and deleting text using any of the editing commands described above. When you have finished editing, press BACK. The definition is now changed and Logo is ready for a new command.

When you change a procedure definition, the computer then uses the new, not the old, definition anytime the procedure is executed.

Changing the procedure's name (by editing the title line) is equivalent to defining a new procedure with the new title. For example, if you edit the PINWHEEL definition to read

```
TO FAN
REPEAT 8 [RIGHT 45 BOX]
END
```

(which draws the shape shown in Figure 1.11), Logo will remember *both* FAN and PINWHEEL.

Printing Procedures and Titles

In order to see the definition of a procedure, you can use the PO (PO stands for "print out") command followed by the name of the procedure. Here is an example:

```
PO PINWHEEL
TO PINWHEEL
FORWARD 50
REPEAT 8 [RIGHT 45 BOX]
BACK 90
END
```

Another useful Logo command is PP (PP stands for "print procedures"), which lists the title lines of all procedures that are currently defined; for example:

```
PP
TO PINWHEEL
TO FAN
TO BOXES
TO BOX
```

If the printout is too long to fit on a single screen, Logo will pause when it fills the screen and type the message

```
PRESS ENTER TO CONTINUE
```

Pressing ENTER will show the next screenful. See Section 5.1.1 for more details on printing procedures.

Defining More Than One Procedure at a Time

If you like, you can use the editor to define more than one procedure definition at a time. You simply type in the definitions in sequence. Be sure to end each separate procedure definition with END.

As with multiple commands on a line, defining more than one procedure at once can cause confusion, because if some procedure definition is badly formed and causes a definition error (see Section 1.4.3 below), the procedure definitions that follow it will not be processed. It is generally better to enter and exit edit mode for each procedure separately.

More Than One Screenful

Sometimes, either because you have a very long procedure definition, or, more commonly, because you are defining many procedures at once, you may want to edit more lines of text than can fit on the screen at once. Logo allows you to do this. If the cursor is at the bottom of the screen and you press RETURN, the lines of text will scroll upwards to produce a new blank line. In general, the screen can be thought of as a window onto a much longer page of text that scrolls as you move the cursor from line to line so that the part you are editing is always within the window.

Long Lines in Procedures

As is the case with command lines, lines in Logo procedures can be more than one screen-line (30 characters) long, up to 127 characters. When you type the 31st character of a long line using the procedure editor, the cursor moves to the left of the next screen line while you continue typing.

1.4.3. Errors in Procedures

If Logo encounters an error while executing a procedure, it prints an error message as described in Section 1.3.3 together with four pieces of information:

- A description of the error.
- The level number at which the error occurred.
- The number of the line that contained the error.
- The name of the procedure in which the error occurred.

The meaning of "level number" follows. A procedure that is called directly by a typed command line is said to be running at level 1; a procedure called by a level 1 procedure is said to be at level 2, and so on. The greater the level, the longer the "chain of procedure calls" from the typed-in command to the procedure in which the error occurred.

For example, suppose you define the procedure

```
TO BLOCK
ELL
RIGHT 90
ELL
END
```

and the definition of the subprocedure ELL contains a typing error (in the third line of the procedure):

```
TO ELL
FORWARD 50
RIGHT 90
FORWAXD 25
END
```

Then if you give the command BLOCK, Logo will run until it tries to execute the third line in ELL for the first time at which point it will type

TELL ME HOW TO FORWAXD
AT LEVEL 2 LINE 3 OF ELL

At this point you should edit ELL and correct the mistyped line.

Errors in Procedure Definitions

When Logo processes a procedure definition, it does not look for errors in the lines that make up the body of the definition. For example, if you make a typing error, as in the second line below:

```
TO PINWHEEL
REPEAT 8 [RIGXT 45 BOX]
END
```

the fact that RIGHT has been mistyped as RIGXT will cause an error when Logo attempts to *execute* PINWHEEL, not when you define the procedure.[12] On the other hand, there are certain things that can cause errors when you press the BACK key and the definition is processed. For example, you may mistakenly use the editing operations to remove the word TO from the title line while you are editing or cause the definition to be badly formed in some other way. Logo will complain, for example, if you try to define a procedure

[12]One very good reason for this is that it is always possible that you *did* mean to type RIGXT, and you will be defining a procedure named RIGXT before using PINWHEEL. One facility that a computer language can provide to encourage sound programming practices is to make it possible to write definitions in terms of procedures that have not yet been defined.

with the same name as some Logo primitive. For instance, if you attempt to define a procedure named FORWARD, Logo will respond to your pressing the BACK key with the error message

TO DOESN'T LIKE FORWARD AS INPUT

1.5. Other Graphics Commands

In addition to the turtle commands FORWARD, BACK, LEFT, and RIGHT, Logo allows you to move the turtle by specifying *x,y* Cartesian coordinates. The SXY command takes two numeric inputs and moves the turtle to the corresponding *x,y* screen location. There are also commands XCOR and YCOR which output the turtle's position.[13] The SETHEADING command rotates the turtle so that it faces in a specified direction, and the HEADING command outputs the turtle's heading. Giving the command HOME moves the turtle back to its initial position at the center of the screen and facing straight up.

Besides drawing with PENUP and PENDOWN, you can also make the turtle *erase* any lines that it passes over. You do this by using the command PENERASE (abbreviated PE). For instance, if you want to erase some lines in a drawing, you can type PENERASE and then drive the turtle over those lines. There is also a command PENREVERSE (abbreviated PR), which is like a combination of PENDOWN and PENERASE. When the pen is reversed, the turtle will "reverse" any points that it passes over. Any dot that is off will be turned on, and any dot that is on will be turned off.

Section 12.1 gives a complete list of the graphics commands that are built into Logo.

1.5.1. Drawing in Color

The SETCOLOR command (abbreviated SC) changes the color in which the turtle draws. SETCOLOR takes as input a number, which specifies the designated color. There are 16 colors available in TI Logo:

CLEAR	0	RUST	8
BLACK	1	ORANGE	9
GREEN	2	YELLOW	10
LIME	3	LEMON	11
BLUE	4	OLIVE	12
SKY	5	PURPLE	13
RED	6	GRAY	14
CYAN	7	WHITE	15

[13]See Section 6.2 on how to use outputs.

For instance, to make the turtle draw in white, you can give the command

```
SETCOLOR 15
```

Alternatively, you can specify the *name* of the color rather than the number. You do this by using the name as input, preceded by a colon (:) as in[14]

```
SETCOLOR :WHITE
```

The COLOR command outputs (as a number) the current color in which the turtle is drawing.

1.5.2. The Background

You can also change the background color of the screen to be any of the above colors. There are two ways to do this. One is to use the command COLORBACKGROUND (abbreviated CB). For instance, to change the background color to yellow, you can type either

```
COLORBACKGROUND 10
```

or

```
COLORBACKGROUND :YELLOW
```

TELL and Graphical Objects

You can also change the background color by typing

```
TELL BACKGROUND
```

followed by a SETCOLOR command, such as

```
SETCOLOR :YELLOW
```

The general idea here is that TELL is a command that "directs the computer's attention" to various kinds of graphical objects. So far we have seen two graphical objects, the TURTLE and the BACKGROUND. When the

---

[14]The use of the colon here is not specifically related to colors or to drawing. Rather, it reflects the general way in which things in Logo can be named. When Logo is started, the symbol WHITE is predefined to be a name for the number 15 (and the other color names likewise). For instance, if you type

```
PRINT :WHITE
```

Logo will print 15. The colon syntax :WHITE directs Logo to find the value associated with WHITE. We will see other uses of : in dealing with inputs to procedures in Section 2.1 and with the MAKE command in Section 6.5.

computer is "talking to" an object (via TELL), all the graphics commands refer to that object. If you type

```
TELL BACKGROUND
```

and then give the COLOR command, the number returned will be the color of the background. You must also be sure that the command is one that makes sense for the object you are TELLing. For instance, if you type

```
TELL BACKGROUND
FORWARD 50
```

Logo will respond with the error message

BACKGROUND CAN'T FORWARD

In this case, you probably meant for the turtle to go forward, so you should redirect the computer's attention to the turtle by typing

```
TELL TURTLE
```

As graphical objects go, the background is a rather limited one, since all it can do is change color. We'll meet more versatile graphical objects in Chapter 4.

1.6. Modes of Using the Screen

This chapter has presented the basics of executing Logo commands and defining simple procedures. As a summary, we note that Logo uses the display screen in three different ways, or *modes*.

1.6.1. Noturtle Mode

Logo starts in noturtle mode. You type in command lines, terminated with ENTER. Logo executes the line and prints a response, if appropriate.

1.6.2. Turtle Mode

Typing TELL TURTLE causes Logo to enter turtle mode as shown in Figure 1.5, with the screen cleared and the turtle at the center. In turtle mode, you use the turtle for drawing on the screen. The NOTURTLE command exits turtle mode and enters noturtle mode.

1.6.3. Edit Mode

Executing the commands TO or EDIT places Logo in edit mode, which allows you to use the procedure editor as described in Section 1.4.2. Pressing the BACK key exits edit mode and processes the definitions.

CHAPTER 2

Programming with Procedures

In the Introduction we stressed that the ability to define procedures is one of the powerful features of the Logo language. In this chapter we explain more about how procedures can be used and, in particular, how they can be used to build up complex programs in simple steps. With the material covered in this chapter, you should have enough information about Logo to undertake many projects in turtle geometry. Be sure to type TELL TURTLE before trying any of the activities in this chapter.

2.1. Procedures with Inputs

The procedures discussed in Section 1.4 do exactly the same thing each time they are executed. Each turtle procedure draws the same drawing each time. Contrast this with a command like FORWARD.

```
FORWARD 50
```

does not draw exactly the same thing as

```
FORWARD 25
```

The fact that the FORWARD command takes an *input* is what enables you to use this one command to draw lines of all different lengths.

In Logo, you can define procedures that take inputs. Consider, for example, the following procedure, which draws a square 50 units in a side:

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

Whenever you give the command SQUARE, the turtle draws a square with side 50. You can change the definition of SQUARE so that it can be used to draw squares of all different sizes:

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

The new SQUARE procedure takes an input that specifies the side of the square to be drawn. The procedure is executed just like any Logo command

that takes an input. That is, to draw a square of side 50, you type

```
SQUARE 50
```

To draw a square of side 25, you type

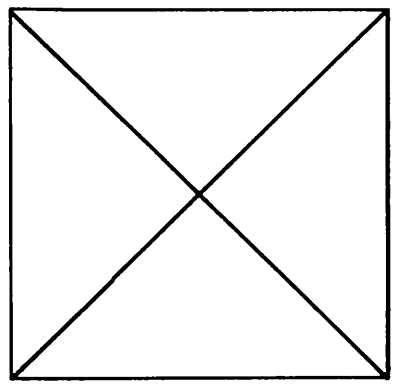
```
SQUARE 25
```

and so on.[1]

The definition of SQUARE illustrates the general rule for defining procedures that take inputs. You choose a name for the input and include it in the procedure title line, preceded by a colon.[2] Now you use the input name (with the colon) wherever you would normally use the value of the input in the procedure body.

To define a procedure with inputs, you use the procedure editor just as in defining any procedure. To enter the editor, type TO followed by as much of the title line as you like, followed by ENTER. For example, if you type TO SQUARE :SIDE (ENTER), you will enter the editor, and the title line of the procedure will be TO SQUARE :SIDE. If you type TO SQUARE (ENTER), you will enter the editor with the title line TO SQUARE, and the :SIDE part of the title line can be added using the normal editing operations.[3]

Here's another example. You can modify the original (side 50) SQUARE procedure to draw a diagonal of the square and return the turtle to its starting point. The procedure uses the fact that the length of the diagonal is the square root of 2 (about 1.4, or 7/5) times the length of the side.



```
TO DIAG
REPEAT 4 [FORWARD 50 RIGHT 90]
RIGHT 45
FORWARD 70
BACK 70
LEFT 45
END
```

Figure 2.1: Shape drawn by the DIAG procedure.

Figure 2.1 shows the shape drawn by this procedure. To draw the shape in all different sizes, you can use

[1] A common beginners' mistake is to type SQUARE :50, based on the (reasonable) misunderstanding that the colon means something like "here is your input." Instead, as we shall see below, the colon as used in :SIDE means "the value associated with the name SIDE."

[2] Logo tradition is to pronounce the colon as "dots." That is :SIDE is pronounced "dots SIDE."

[3] In the first release of TI Logo, you should type only TO and the procedure title and add the inputs with the editing operations. In TI Logo II, you can use either method as described above.

```
TO DIAG :SIZE
REPEAT 4 [FORWARD :SIZE RIGHT 90]
RIGHT 45
FORWARD (:SIZE * 7) / 5
BACK (:SIZE * 7) / 5
LEFT 45
END
```

2.1.1. Multiple Inputs

Logo procedures may be defined to accept more than one input. You simply choose a name for each input and include it in the title line, preceded by a colon. For example, the following two-input procedure can be used to draw rectangles of varying sizes and shapes:

(a)

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

(b)

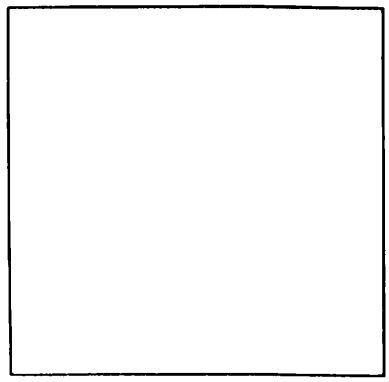


Figure 2.2: Two rectangles drawn by the RECTANGLE procedure.

As shown in Figure 2.2, executing the command

```
RECTANGLE 50 10
```

draws a long, skinny rectangle, whereas

```
RECTANGLE 50 50
```

draws a square.

2.1.2. Inputs as Private Names

Defining a Logo procedure involves grouping together a series of commands under a *name* chosen by the programmer. Using inputs also involves naming, but in a different sense. Although a new procedure is incorporated as part of Logo's working vocabulary, the name of an input is *private* to the procedure that uses the input.

Since input names are private, different procedures may use the same names for inputs without these names interfering with each other. One way to think about this is to imagine that each time a procedure is executed, it sets up a "private library" that associates with its input names the actual input values with which the procedure was called. When the procedure executes a line that contains an input name (signaled by :) it looks up the value in the library and substitutes the values for the name. For example, the previous RECTANGLE procedure, called with

```
RECTANGLE 10 50
```

would set up a private library as shown in Figure 2.3.

RECTANGLE

HEIGHT	10
LENGTH	50

Figure 2.3: Private library set up by executing RECTANGLE 10 50

The input values are associated with the input names in the order in which they appear in the title line. In this case, the first input, 10, is associated with the first input name, HEIGHT, and the second input, 50, is associated with the second name, LENGTH.

We've already seen in Chapter 1 that the individual steps in a procedure can themselves be procedures. Since each procedure maintains its own private library of input values, there is no conflict between the input names used by the different procedures. For example, here is RECTANGLE used as part of a procedure for drawing a flag, as shown in Figure 2.4:

```
TO FLAG :HEIGHT
FORWARD :HEIGHT
RECTANGLE (:HEIGHT / 2) :HEIGHT
BACK :HEIGHT
END
```

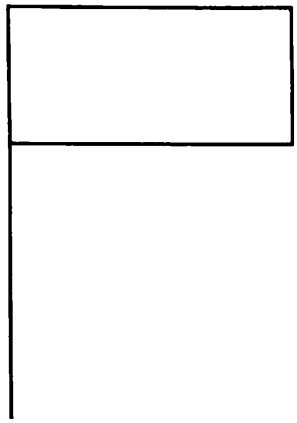


Figure 2.4: Figure drawn by executing FLAG 50.

The FLAG procedure draws a "pole" of a specified HEIGHT, then draws on top of the pole a rectangle of dimensions HEIGHT/2 by HEIGHT, then moves the turtle back to the base of the pole. Note the use of parentheses around (:HEIGHT / 2). These are not actually necessary for Logo to understand what is meant, but they make the program easier to read.[4]

Let's examine in detail what happens when you execute the command

```
FLAG 50
```

This creates a private library for FLAG in which HEIGHT is associated with 50 and begins executing the definition of FLAG, starting with the first line

```
FORWARD :HEIGHT
```

Looking in the private library, Logo finds that 50 is the value associated with HEIGHT, so it makes the turtle go FORWARD 50. Next it must execute the line

```
RECTANGLE (:HEIGHT / 2) :HEIGHT
```

To do this, Logo first determines the values of the two inputs that must be given to RECTANGLE. The first input is half the value of HEIGHT, or 25, and the second input is HEIGHT itself, or 50. Now RECTANGLE is called with inputs 25 and 50. This sets up a private library for RECTANGLE in which the names of RECTANGLE's inputs, HEIGHT and LENGTH, are associated with 25 and 50, respectively. The entire picture is as shown in

[4]Section 6.7.2 discusses the rules for using parentheses in Logo.

Figure 2.5. Even though the name HEIGHT is associated with 50 in FLAG's library and with 25 in RECTANGLE's library, there is no conflict between the two. Each procedure looks up its own values in its own library.

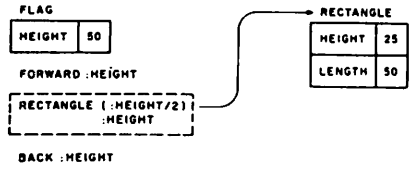


Figure 2.5: Private libraries set up by executing FLAG 50.

The importance of private input names is that you can use a procedure without concern for the details of precisely *how it is coded*, but rather just concentrating on *what it does*. When you write the FLAG procedure, you can regard RECTANGLE as a "black box" that draws a rectangle, without worrying about what names it uses for its inputs. Indeed, as far as FLAG is concerned, RECTANGLE might have been a primitive included in the Logo system.

The technique of regarding a procedure (even a complex procedure) as a black box whose details you needn't worry about at the moment is a crucial idea in programming or, indeed, in any kind of design enterprise. Each time you define a new procedure, you can use it as a building block in more complex procedures, and in this way you can build up very complex processes in what Papert [15] refers to as "mind-size bites."

As a simple illustration, once you have defined FLAG you can use it to easily make a procedure that draws a flag and moves the turtle over a bit:[5]

```
TO FLAG.AND.MOVE :SIZE :SPACING
PENDOWN
FLAG :SIZE
PENUP
RIGHT 90
FORWARD :SPACING
LEFT 90
END
```

You can use this to draw a row of flags as in Figure 2.6:

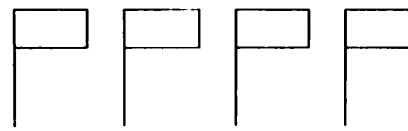


Figure 2.6: Picture drawn by ROW 20 30 4.

[5]The period used in a name like FLAG.AND.MOVE is interpreted as an ordinary character. Logo does not allow spaces to be part of procedure names, so the period is a useful way to make long names more readable.

```
TO ROW :SIZE :SPACING :HOW.MANY
REPEAT :HOW.MANY [FLAG.AND.MOVE :SIZE :SPACING]
END
```

2.1.3. An ARC Procedure

As another example of using procedures with inputs, we'll consider the problem of writing a procedure to draw circular arcs. This is not only a good example of using procedures, but is also a useful building block to have in making drawings.

The ARC procedure is based on making the turtle go FORWARD a small fixed distance, turning a small fixed angle, and repeating this over and over—this draws a good approximation to a circular arc.[6] When the turtle has turned through 360 degrees, a complete circle will have been drawn. This leads to the following CIRCLE procedure:[7]

```
TO CIRCLE1
REPEAT 360 [FORWARD 1 RIGHT 1]
END
```

This draws a circle, but it is very slow, especially if you use it without hiding the turtle. The problem is that there are so many FORWARD 1, LEFT 1 moves. And these are mostly unnecessary, because, within the accuracy of the display screen, a regular polygon with more than 20 sides is indistinguishable from a circle. For example, you can replace the CIRCLE1 procedure above by the following procedure, which draws a regular 36-sided polygon:

```
TO CIRCLE2
REPEAT 36 [FORWARD 10 RIGHT 10]
END
```

(Notice that you multiply the FORWARD step by 10 in order to keep the circle the same size as before.) The CIRCLE2 procedure runs about 10 times as fast as CIRCLE1 and looks almost the same on the display screen.

---

[6] This is a fundamental idea in turtle geometry, based on the fact that a circle is a curve of constant curvature. This observation is the key to many turtle-based approaches to mathematics as described in the book by Abelson and diSessa [1].

[7] The digit 1 included as part of the name CIRCLE1 is interpreted as an ordinary character. It is standard practice to name minor variants of procedures by appending a number to the name.

You can make this procedure more useful by giving it an input that varies the size of the circle:

```
TO CIRCLE :SIZE
REPEAT 36 [FORWARD :SIZE RIGHT 10]
END
```

Note that the turtle still turns 10 degrees at each step, so varying the size of the FORWARD step varies the size of the circle. Figure 2.7 shows some circles drawn by the CIRCLE procedure.

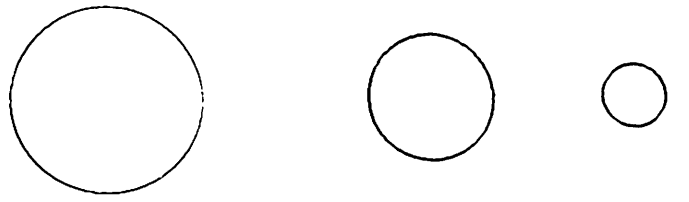


Figure 2.7: Circles drawn by the CIRCLE procedure.

An arc procedure can be implemented in the same way, except the turtle should turn through as many degrees as there are degrees in the arc. The following procedure draws circular arcs turning toward the right:

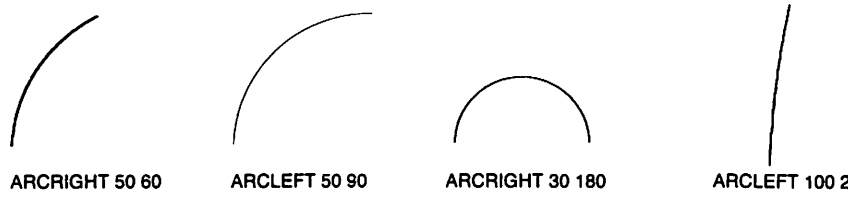


Figure 2.8: Circular arcs drawn by the ARCRIGHT procedure.

```
TO ARCRIGHT :SIZE :DEGREES
REPEAT :DEGREES/10 [FORWARD :SIZE RIGHT 10]
END
```

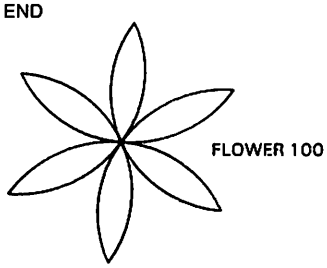
Note that we divide the DEGREES input by 10 to obtain the number of 10-degree steps the turtle should perform to construct an arc of that many degrees.[8] An ARCLEFT procedure can be designed in exactly the same way. Figure 2.8 shows some arcs generated by this procedure.

Once you have defined ARCRIGHT and ARCLEFT, you can use them to develop all sorts of interesting shapes. Figure 2.9 shows two examples.

[8] In TI Logo, division always produces an integer quotient; for instance, 76/10 yields 7. Our arc procedure will give a correct result only when the DEGREES input is a multiple of 10.

2.2. Repetition and Recursion

```
TO PETAL :SIZE
ARCRIGHT :SIZE 60
RIGHT 120
ARCRIGHT :SIZE 60
RIGHT 120
END

TO FLOWER :SIZE
REPEAT 6 [PETAL :SIZE RIGHT 60]
END
```




```
TO RAY :SIZE
ARCLEFT :SIZE 90
ARCRIGHT :SIZE 90
ARCLEFT :SIZE 90
ARCRIGHT :SIZE 90
END

TO SUN :SIZE
REPEAT 9 [RAY :SIZE RIGHT 160]
END
```

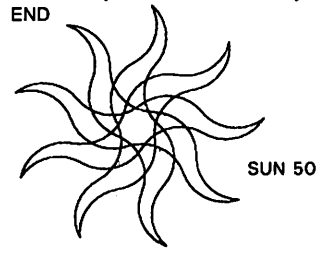


Figure 2.9: Simple procedures that use arcs.

We've already seen the use of the Logo REPEAT command (page 7) to repeat a series of steps a fixed number of times. Another way to make something repeat is to define a procedure that includes a call to itself as the final line. For example,

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

makes the turtle move in a square pattern over and over again until you stop it by pressing BACK. You can think of the way this procedure works as a kind of joke—the steps of a procedure can include calls to any procedure, so why not call the procedure itself? In this case, the definition of SQUARE is "go forward, turn right, and then do SQUARE again." And this last step entails going forward, turning right, and then doing SQUARE again, and so on forever.[9]

One disadvantage of this SQUARE, as opposed to the one we have been previously using,

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

[9]Compare: If a genie appears and offers you three wishes, you should use your third wish to wish for three more wishes.

is that it goes on indefinitely and so is not a good building block to use in making more complex drawings. On the other hand, this kind of indefinite repetition can be useful in situations in which you do not know (or cannot easily figure out) how many times to repeat some sequence of steps. The following program is an excellent example:

```
TO POLY :SIDE :ANGLE
FORWARD :SIDE
RIGHT :ANGLE
POLY :SIDE :ANGLE
END
```

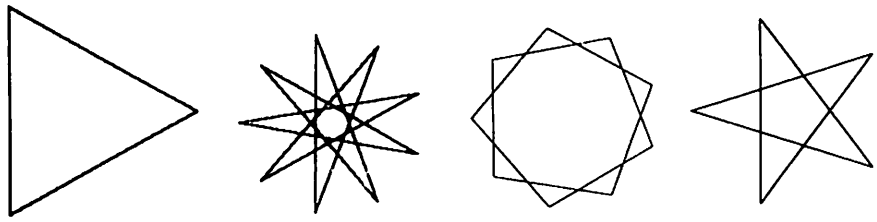
Figure 2.10 shows some of the many figures drawn by POLY as the angle varies. They are all closed figures, but the number of sides that must be drawn before the figure closes depends in a complicated way upon the ANGLE input to the program.[10] Using the indefinite repeat you can draw them all with a single, simple procedure.

Recursion is the programming word for the ability to use the term POLY as part of the definition of POLY or, in general, to write procedures that call themselves.[11]

2.2.1. Thinking About Recursion

The recursive procedures above have a very simple form—they merely repeat an unchangeable cycle over and over again. Recursion is a much more powerful idea and can be used to obtain much more complicated effects. We shall meet many examples. To take just a small step beyond the purely repetitive kind of recursion, consider

```
TO COUNTDOWN :NUMBER
PRINT :NUMBER
COUNTDOWN : NUMBER - 1
END
```



POLY 50 120 POLY 50 160 POLY 60 80 POLY 80 144

Figure 2.10: Shapes drawn by the POLY program.

[10] This phenomenon forms the basis for a number of mathematical investigations involving symmetry and number theory, described in Abelson and diSessa [1].

[11] Languages like Fortran and (most versions of) BASIC do not allow recursion because the implementation of a computer language is simplified if one can assume that there are no recursive functions.

Let's examine what happens if you give the command

```
COUNTDOWN 10
```

To understand the effect of this command, look back at the definition of the COUNTDOWN procedure. You see that it needs an input and that is uses the name NUMBER for this input. In this case, you have given 10 as the input, so the procedure takes NUMBER to be 10.[12] The first line says

```
PRINT :NUMBER
```

so it prints *10* and goes on to the next line, which is

```
COUNTDOWN :NUMBER - 1
```

or, in this case

```
COUNTDOWN 9
```

This order causes the same effect as if you had typed in the command

```
COUNTDOWN 9
```

which would be to print *9* and then give the order

```
COUNTDOWN 8
```

and so on . . . In sum, the effect of

```
COUNTDOWN 10
```

is to print *10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, −1, −2,* . . . until you stop the process by pressing BACK.

Another example of the same programming technique is the following modification of the POLY program on page 23:

```
TO POLYSPI :SIDE :ANGLE
FORWARD :SIDE
RIGHT :ANGLE
POLYSPI (:SIDE + 3) :ANGLE
END
```

Giving the command

```
POLYSPI 0 90
```

[12]Using the terminology introduced in Section 2.1.2, we would say that COUNTDOWN sets up a private library in which the name NUMBER is associated with 10.

leads to the sequence of turtle moves

```
FORWARD 0
RIGHT 90
FORWARD 3
RIGHT 90
FORWARD 6
RIGHT 90
FORWARD 9
RIGHT 90
.
.
.
```

which procedures a square-like spiral.[13] By changing the ANGLE input, you can draw all sorts of spiral shapes, as shown in Figure 2.11. Part of the power of recursion is the fact that such simple programs can lead to such varied, unexpected results.

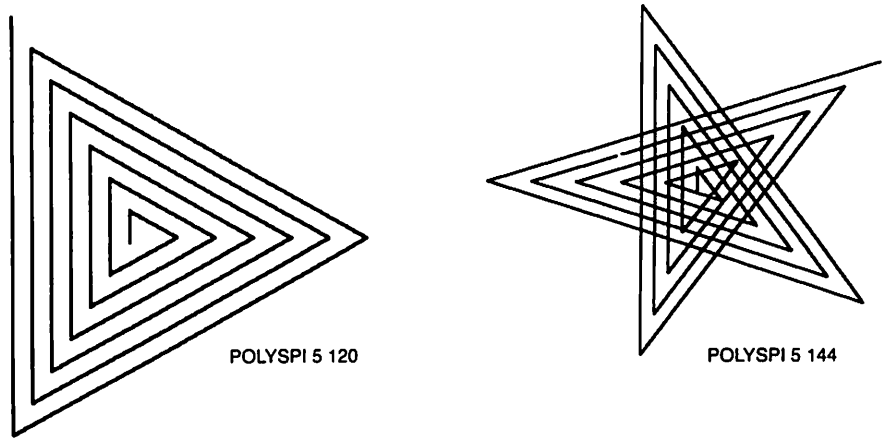


Figure 2.11: Shapes drawn by the POLYSPI program.

2.2.2. Conditional Commands and STOP

Suppose you want COUNTDOWN to stop before printing 0. You can do this as follows:

```
TO COUNTDOWN :NUMBER
IF :NUMBER = 0 STOP
PRINT :NUMBER
COUNTDOWN :NUMBER - 1
END
```

[13] Or "squiral," as it was dubbed by a fifth-grade Logo Programmer who discovered this figure.

The IF statement is used in Logo to perform tests, in this case to test whether the value of NUMBER is zero. If so, the COUNTDOWN procedure STOPs. That is, rather than continuing with the next line in the procedure, it returns control to wherever the procedure was originally called from. So in response to the command

```
COUNTDOWN 5
```

the computer prints 5, 4, 3, 2, 1 and prompts for a new command.

Keep in mind that the idea of STOP is that when a procedure stops, the next command that gets executed is the one after the command that called the procedure. For example,

```
TO BLASTOFF
COUNTDOWN 10
FORWARD 100
END
```

counts down from 10 to 1 and then moves the turtle.[14]

The IF statement is called a *conditional expression*. It has the form

```
IF {some condition is true} {do some action}
```

If the condition is true, then the rest of the line is executed. If not, execution proceeds with the next line. If you like, you can separate the condition and the action with the word THEN as in

```
TO COUNTDOWN :NUMBER
IF :NUMBER = 0 THEN STOP
PRINT :NUMBER
COUNTDOWN : NUMBER - 1
END
```

Either way is acceptable. The THEN is completely optional.

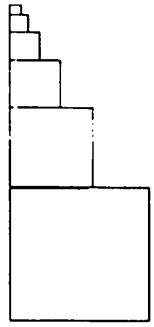
The kinds of conditions that can be tested are generated by Logo operations called *predicates*. Predicates are things whose value is either true or false. COUNTDOWN uses =, which is true if the two things it is comparing are equal. Two other predicates are >, which tests whether the number on its left is greater than the number on its right, and <, which tests for less than. These three predicates deal with numbers.[15] Logo includes other predicates for dealing with other kinds of data. It is also easy to define your own special-purpose predicates (see Section 6.6).

---

[14]This stopping behavior is just what normally happens after a procedure executes its final line. If you like, you can imagine that every procedure includes a STOP command at its end.

[15]Actually, = can be used for testing equality of any two pieces of Logo data. A precise description of the behavior of = is given in Section 12.6.

Here is a turtle program based on the COUNTDOWN model. It draws a tower of squares that get smaller and smaller and stops when the squares get very tiny, as shown in Figure 2.12.



```
TO TOWER :SIZE
IF :SIZE < 1 THEN STOP
SQUARE :SIZE
FORWARD :SIZE
TOWER :SIZE / 2
END
```

Figure 2.12: Picture drawn by TOWER 50.

2.2.3. Thinking Harder About Recursion

The recursion examples we have seen so far, in which the recursive call is the final step in the procedure, can be readily viewed as a kind of generalized repetition.[16] Other uses of recursion can be much more powerful but, unfortunately, much harder to understand. Let's compare the COUNTDOWN procedure from Section 2.2.2:

```
TO COUNTDOWN :NUMBER
IF :NUMBER = 0 STOP
PRINT :NUMBER
COUNTDOWN :NUMBER - 1
END
```

with the following similar-looking procedure:

```
TO MYSTERY :NUMBER
IF :NUMBER = 0 STOP
MYSTERY :NUMBER - 1
PRINT :NUMBER
END
```

As we saw,

```
COUNTDOWN 3
```

[16] The special case of recursion in which the recursive call is the final step is sometimes called *tail recursion*. Logo includes techniques for implementing tail recursion efficiently, so that a tail recursive procedure can effectively run "forever" without running out of storage.

prints 3, 2, 1. In contrast

```
MYSTERY 3
```

prints 1, 2, 3. Most people find this very hard to understand.

Let's trace through the process carefully. You first call MYSTERY with the input 3, so, as explained on page 9, MYSTERY sets up a private library in which NUMBER is associated with 3. It checks whether the value of NUMBER is 0, which it is not, so MYSTERY proceeds to the next line which produces the command

```
MYSTERY 2
```

Now let's stop and think. Eventually this second MYSTERY call will stop, and the original

```
MYSTERY 3
```

procedure will have to continue with the next command after the call. But this means that there will have to be, in some sense, *two* MYSTERY procedures existing at once—the one called by the command

```
MYSTERY 2
```

and the original one called by the command

```
MYSTERY 3
```

which is waiting for the other MYSTERY to stop, so it can continue. Moreover, each MYSTERY has its *own* value for NUMBER—NUMBER is 2 for one and 3 for the other. Each MYSTERY must maintain a *separate* private library.[17] The situation is shown in Figure 2.13.

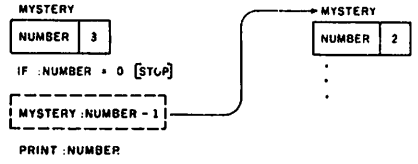


Figure 2.13: Beginning execution of MYSTERY 3.

[17] In other words, the private library is associated, not with a procedure, but with a given call to a procedure (or what is technically called an *activation* of a procedure).

Let's go on. The first thing that the

MYSTERY 2

procedure does is check whether the value for NUMBER is equal to 0. Since this is not the case, MYSTERY gives the command

MYSTERY 1

and so now there are *three* MYSTERY procedures! And

MYSTERY 1

does a test and calls up yet another

MYSTERY 0

which makes four MYSTERY calls all existing at once as shown in Figure 2.14. Note that so far nothing has been printed. All that has happened is that MYSTERY procedures have called up more MYSTERY procedures.

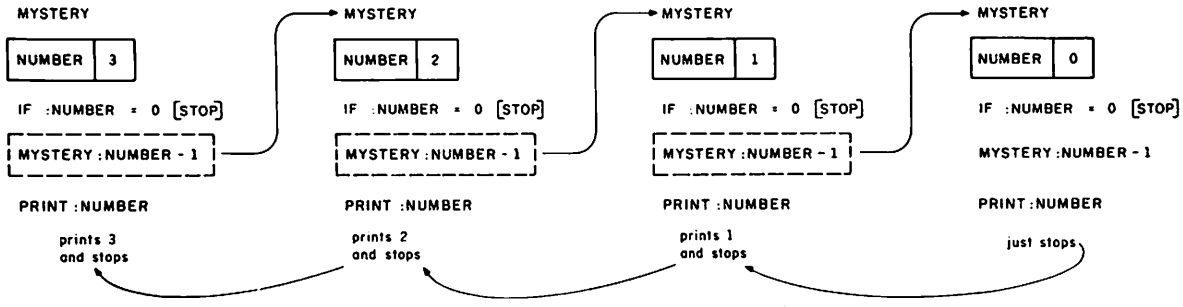


Figure 2.14: Complete execution of MYSTERY 3.

Now

MYSTERY 0

performs its test and finds that the value of NUMBER is indeed 0. So it STOPs and the process continues with the procedure that called it, namely,

MYSTERY 1

This MYSTERY now proceeds with the next line after the call, which says to print the value of NUMBER. Since NUMBER is 1 (in *this* MYSTERY's private library), it prints 1. Now it is done and so returns to the procedure that called it, namely

MYSTERY 2

This MYSTERY now,continues with the line after the call, which says to print NUMBER. Since NUMBER is 2 (in *this* private library), it prints 2 and returns to its caller, namely,

```
MYSTERY 3
```

which prints 3 and returns to its caller, which is the main Logo command level.

Whew! Try going through this example again step by step, referring to Figure 2.14. In essence, this complex process is doing nothing more than unwinding the following rule:

- When a procedure is called, the calling procedure waits until the second procedure stops and then continues with the next instruction after the call.

Recursion, however, forces us to appreciate all the ramifications of this simple sounding rule. In particular:

- There may be several instances (or "activations") of the "same" procedure all coexisting at once.
- Each procedure activation has a *separate* private library, so the "same" name may be associated with different values in different procedure activations.
- The order in which things happen can be very confusing.[18]

2.2.4. Drawing Trees

As another example of complex use of recursion, let's look at a program that draws a *binary tree*, as in Figure 2.15.

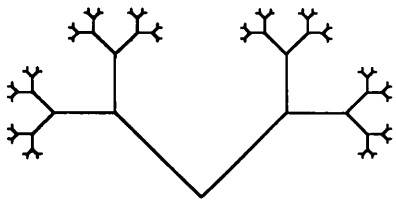


Figure 2.15: A binary tree.

Think about how you would describe this figure. One way to do it would be to say something like "the tree is a vee-shape with a smaller tree at each tip. And each smaller tree is a vee-shape with a still smaller tree at each of its

[18]More specifically, things happen in the reverse order from the way one might expect. This is a consequence of the fact that the last procedure called is the first one to stop.

tips, and so on." This is a *recursive description* of the tree. You can translate this description into a *recursive procedure* that draws the figure. You start with the following commands that make the turtle draw a vee-shape of a certain length and return to its initial position and heading:

```
LEFT 45
FORWARD :LENGTH
BACK :LENGTH
RIGHT 90
FORWARD :LENGTH
BACK :LENGTH
LEFT 45
```

This is the basic vee-shape of the tree. Now, according to the recursive description, the entire tree consists of this vee with smaller vees (say, half as big) drawn at each tip. So the TREE procedure should be something like

```
TO TREE :LENGTH
LEFT 45
FORWARD :LENGTH
TREE :LENGTH / 2
BACK :LENGTH
RIGHT 90
FORWARD :LENGTH
TREE :LENGTH / 2
BACK :LENGTH
LEFT 45
END
```

But this doesn't quite work. Consider—if you call TREE with an input of 20, this will make the turtle go LEFT 45, FORWARD 20 and call

```
TREE 10
```

which will make the turtle go LEFT 45, FORWARD 10 and call

```
TREE 5
```

and so on forever.[19] This is something like the forever-running COUNTDOWN procedure on page 23, or even more like the chain of MYSTERY procedures on page 26, in that no procedure finishes until the last one to be called has stopped. What you need is a *stop rule* to keep the

[19]That is, until Logo runs out of storage.

process from going on forever. You can make the process stop by having the procedure just stop without drawing anything if LENGTH is very small:

```
TO TREE :LENGTH
IF :LENGTH < 2 THEN STOP
LEFT 45
FORWARD :LENGTH
TREE :LENGTH / 2
BACK :LENGTH
RIGHT 90
FORWARD :LENGTH
TREE :LENGTH / 2
BACK :LENGTH
LEFT 45
END
```

You can modify the TREE procedure to produce a procedure TREE1, in which the subtree branches have the same length as the original branches, rather than half the length. If you do this, however, then the branches of successive subtrees will not get smaller and smaller, which means that you cannot use the same stop rule as in TREE. A different strategy for providing a stop rule is to include for TREE1 an extra input, DEPTH, which determines the "depth" to which the tree is drawn. Each tree of a given depth spawns two subtrees of depth one less. When the TREE procedure is called with DEPTH equal to 0, it just stops without drawing:

```
TO TREE1 :LENGTH :DEPTH
IF :DEPTH = 0 THEN STOP
LEFT 45
FORWARD :LENGTH
TREE1 :LENGTH :DEPTH - 1
BACK :LENGTH
RIGHT 90
FORWARD :LENGTH
TREE1 :LENGTH :DEPTH - 1
BACK :LENGTH
LEFT 45
END
```

Thinking in terms of recursive descriptions can take a lot of getting used to, and the programs can be subtle. One especially subtle point about the TREE program is the final BACK and LEFT moves, which are needed to restore the turtle to its initial heading so that the different calls to TREE will fit together correctly. On the other hand, many seemingly complex designs

have simple recursive descriptions and can be drawn by remarkably brief programs. The design of recursive turtle programs for drawing complex patterns is discussed extensively in Abelson and diSessa [1].

To illustrate the flavor of recursive designs, here is a modification to TREE1, in which the left branch of each vee is twice as long as the right branch. We'll also allow the angle of the vee to be varied as an input. Figure 2.16 shows some of the patterns that result.

```
TO NEW.TREE :LENGTH :ANGLE :DEPTH
IF :DEPTH = 0 THEN STOP
LEFT :ANGLE
 FORWARD 2 * :LENGTH
 NEW.TREE :LENGTH :ANGLE :DEPTH - 1
 BACK 2 * :LENGTH
 RIGHT 2 * :ANGLE
FORWARD :LENGTH
NEW.TREE :LENGTH :ANGLE :DEPTH - 1
BACK :LENGTH
LEFT :ANGLE
END
```

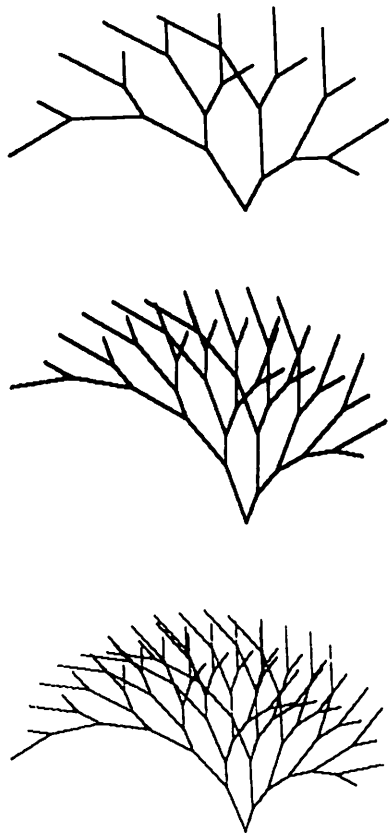


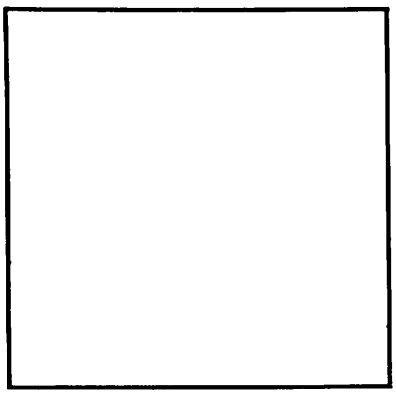
Figure 2.16: Some figures drawn by the NEW.TREE procedure.

CHAPTER 3

Projects in Turtle Geometry

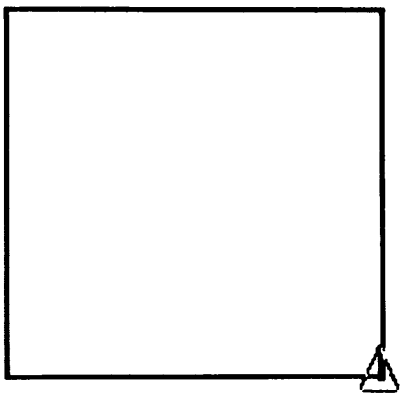
Here are some projects that use Turtle Geometry. Refer to other portions of this text for help in defining or editing programs. Feel free to change programs that are offered and to design new programs. Be sure to type TELL TURTLE before trying any of these projects.

Here is a square procedure.

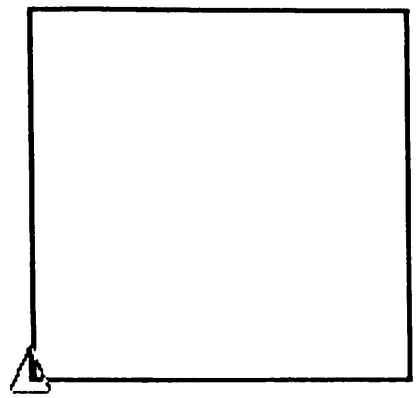


```
TO SQUARE
REPEAT 4 [FORWARD 60 RIGHT 90]
END
```

Here are two square procedures designed to allow variable size. The triangles show the turtle's initial position.



```
TO LSQUARE :SIZE
FORWARD :SIZE
LEFT 90
FORWARD :SIZE
LEFT 90
FORWARD :SIZE
LEFT 90
FORWARD :SIZE
LEFT 90
END
```



or

```
TO LSQUARE :SIZE
REPEAT 4 [FORWARD :SIZE LEFT 90]
END
```

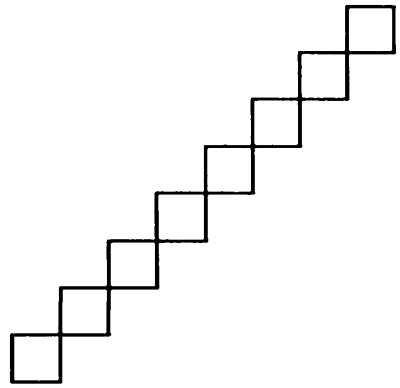
```
TO RSQUARE :SIZE
FORWARD :SIZE
RIGHT 90
FORWARD :SIZE
RIGHT 90
FORWARD :SIZE
RIGHT 90
FORWARD :SIZE
RIGHT 90
END
```

or

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

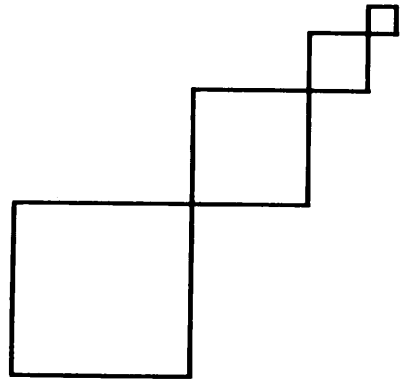
Some procedures using RSQUARE and recursion.

Some ideas for using square procedures.

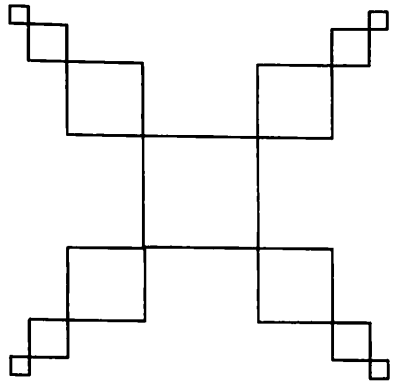


```
TO MOVE :SIZE
FORWARD :SIZE
RIGHT 90
FORWARD :SIZE
LEFT 90
END
```

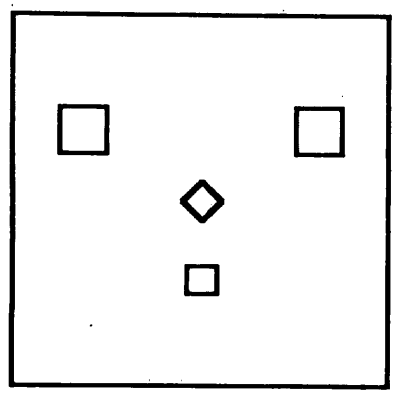
```
TO STAIRS :SIZE
RSQUARE :SIZE
MOVE :SIZE
STAIRS :SIZE
END
```

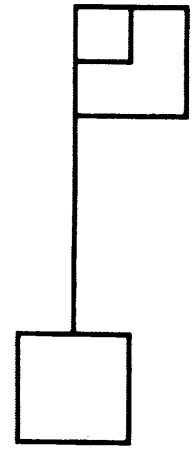


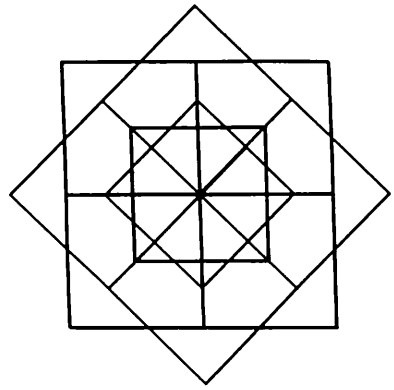
```
TO BOXES
RSQUARE 30
MOVE 30
RSQUARE 20
MOVE 20
RSQUARE 10
MOVE 10
RSQUARE 5
RIGHT 180
PENUP
MOVE 60
RIGHT 180
PENDOWN
END
```

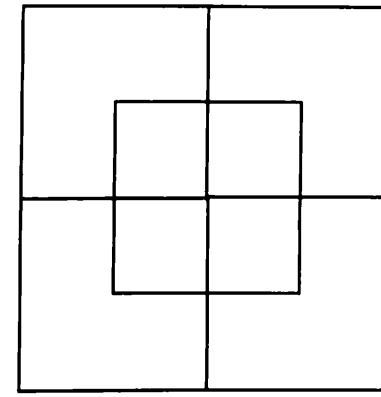


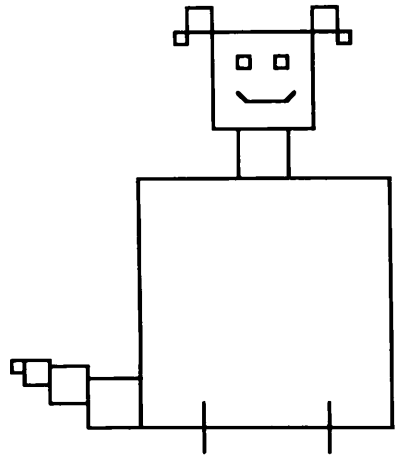
```
TO MANYBOXES
BOXES
FORWARD 30
RIGHT 90
MANYBOXES
END
```

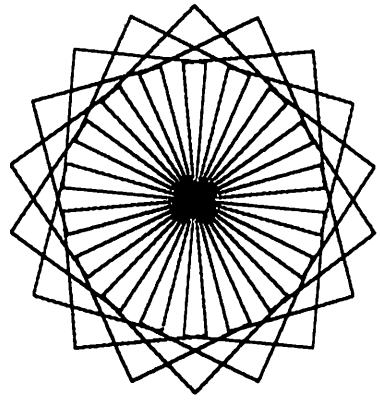




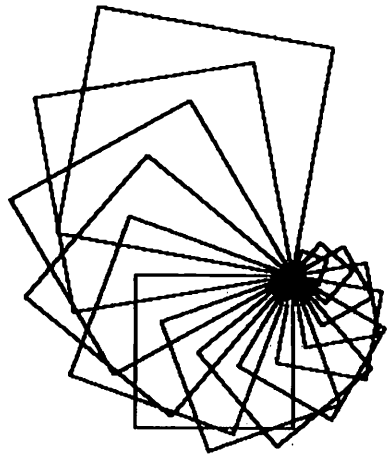






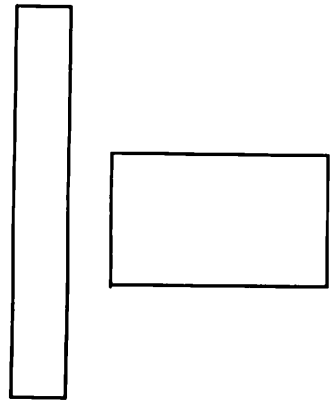


```
TO SPINSQUARES :SIZE
RSQUARE :SIZE
RIGHT 20
SPINSQUARES :SIZE
END
```

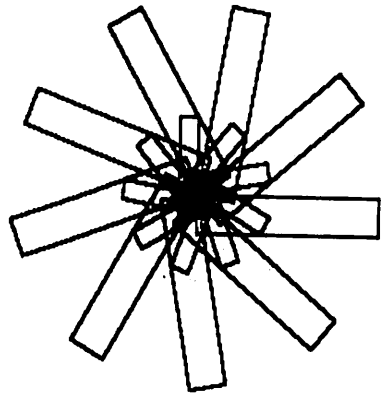


```
SPINSQUARES 40

TO GROWSQUARES :SIZE
RSQUARE :SIZE
RIGHT 20
GROWSQUARES :SIZE + 5
END
```

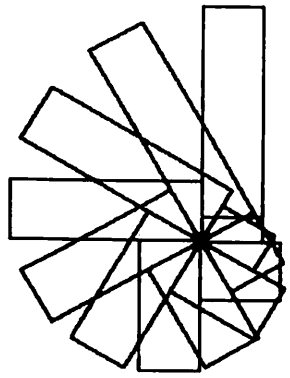
A rectangle procedure designed to allow variable size and some examples that use it.



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

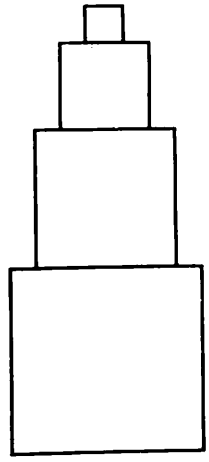


```
TO FLOWER
RECTANGLE 50 10
RIGHT 20
RECTANGLE 5 20
RIGHT 20
FLOWER
END
```

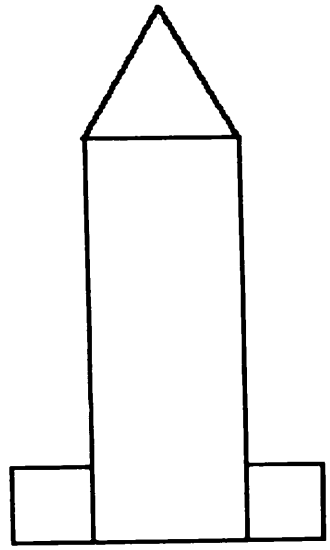


```
TO SPINRECS :SIZE
IF :SIZE < 10 STOP
RECTANGLE :SIZE 20
LEFT 30
SPINRECS :SIZE -5
END
```

Examples using RSQUARE and RECTANGLE.



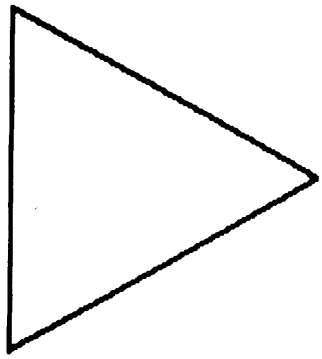
```
TO HOP :SIZE
FORWARD :SIZE
RIGHT 90
FORWARD 3
LEFT 90
END

TO TELESCOPE :SIZE
IF :SIZE < 6 STOP
RSQUARE :SIZE
HOP :SIZE
TELESCOPE :SIZE - 6
END
```

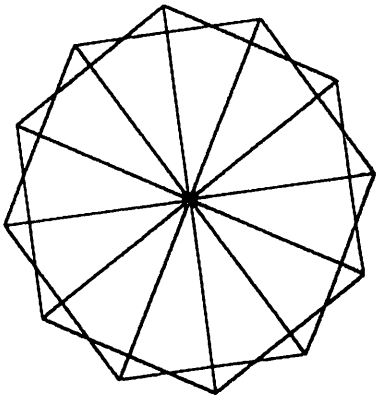


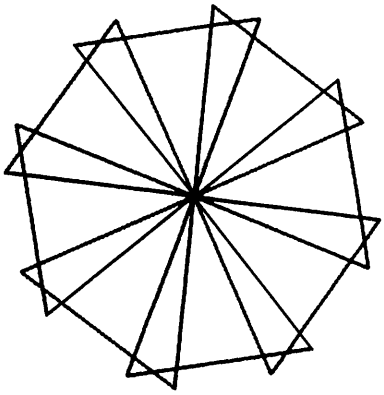
```
TO ROCKTOP
LEFT 30
FORWARD 30
LEFT 120
FORWARD 30
END

TO ROCKET
RECTANGLE 80 30
LEFT 90
RECTANGLE 15 15
BACK 30
RIGHT 90
RECTANGLE 15 15
FORWARD 80
ROCKTOP
END
```

Here are some examples that use a triangle procedure.

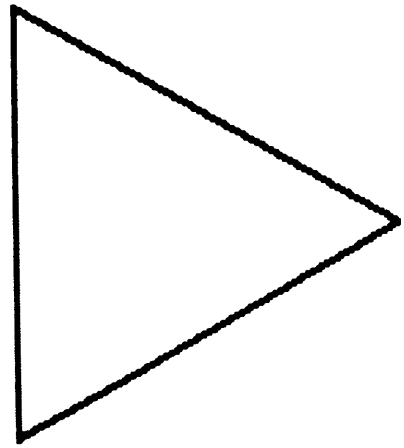
```
TO TRI
REPEAT 3 [FORWARD 70 RIGHT 120]
END
```





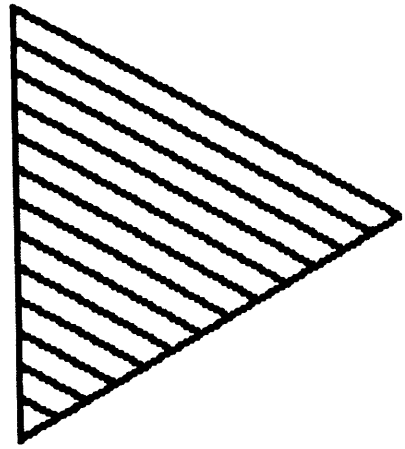


A triangle procedure designed to allow variable size and an example that uses it.



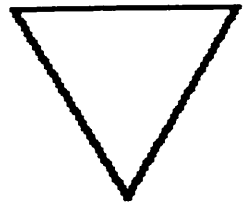
```
TO TRIANGLE :SIZE
FORWARD :SIZE
RIGHT 120
FORWARD :SIZE
RIGHT 120
FORWARD :SIZE
RIGHT 120
END
```

This procedure is different in design but has a similar result.



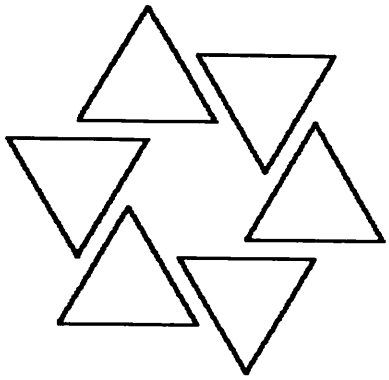
```
TO FLUFF :SIZE
IF :SIZE < 10 STOP
TRIANGLE :SIZE
FLUFF :SIZE - 10
```

Some more triangle examples.

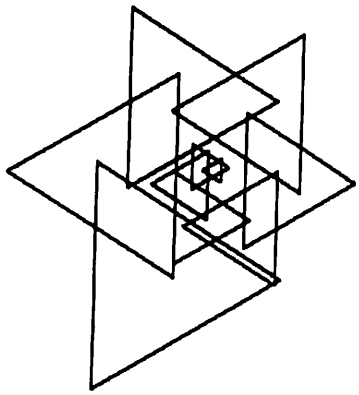


```
TO NEWTRIANGLE :SIZE
LEFT 30
TRIANGLE :SIZE
RIGHT 30
END
```

```
TO CREEP :SIZE
PENUP
FORWARD :SIZE
PENDOWN
END
```



```
TO LOOPS :SIZE
NEWTRIANGLE :SIZE
CREEP :SIZE
RIGHT 60
LOOPS :SIZE
END
```



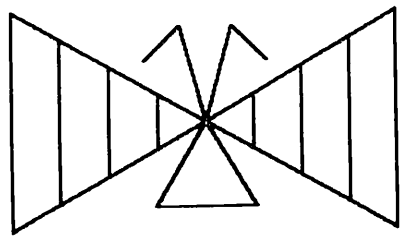
```
TO NEWLOOP :SIZE
IF :SIZE < 20 STOP
NEWTRIANGLE :SIZE
CREEP :SIZE/2
RIGHT 60
NEWLOOP :SIZE – 5
END
```

And one more.

```
TO LEFTANT
LEFT 15
FORWARD 30
LEFT 120
FORWARD 15
BACK 15
RIGHT 120
BACK 30
RIGHT 15
END
```

```
TO RIGHTANT
RIGHT 15
FORWARD 30
RIGHT 120
FORWARD 15
BACK 15
LEFT 120
BACK 30
LEFT 15
END
```

```
TO ANTS
RIGHTANT
LEFTANT
END
```



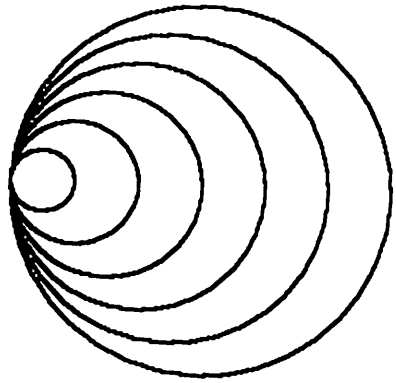
```
TO BUTTERFLY
RIGHT 60
WING
RIGHT 180
WING
RIGHT 120
ANTS
RIGHT 150
TRIANGLE 30
END
```

```
TO WING
TRIANGLE 80
TRIANGLE 60
TRIANGLE 40
TRIANGLE 20
END
```

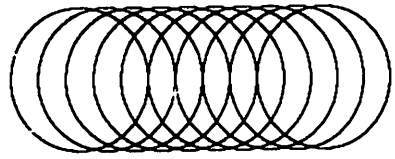
RCP and LCP are abbreviations for "Right Circle Piece" and "Left Circle Piece." RARC and LARC stand for "right arc" and "left arc." A circle can be made from pieces of either left or right arcs, leaving the turtle at the left-most or right-most point of the circle.

```
TO RCP :R
RIGHT 15
FORWARD :R/2
RIGHT 15
END

TO LCP :R
LEFT 15
FORWARD :R/2
LEFT 15
END

TO RARC :R
REPEAT 3 [RCP :R]
END

TO LARC :R
REPEAT 3 [LCP :R]
END

TO RCIRCLE :R
REPEAT 12 [RCP :R]
END

TO LCIRCLE :R
REPEAT 12 [LCP :R]
END
```

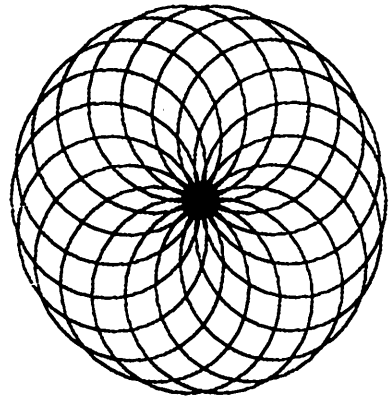
Examples Using Circle Procedures.



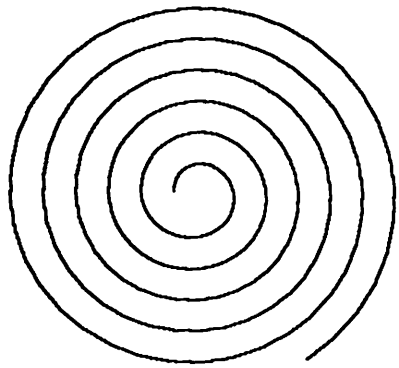
```
TO SHRINKRCIRCLE :SIZE
IF :SIZE < 4 STOP
RCIRCLE :SIZE
SHRINKRCIRCLE :SIZE - 2
END
```



```
TO RSLINKY :SIZE
RCIRCLE :SIZE
PU RT 90 FD 10 LT 90 PD
RSLINKY :SIZE
END
```

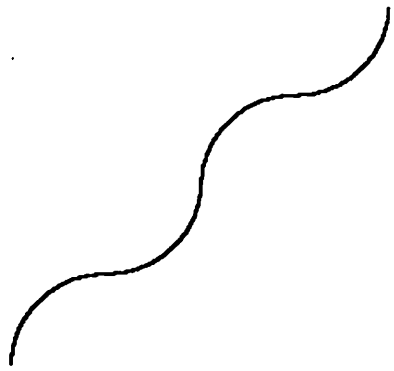


```
TO SPINSLINK :SIZE
RCIRCLE :SIZE
RIGHT 20
SPINSLINK :SIZE
END
```

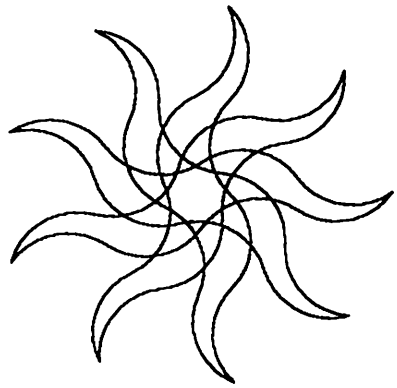


```
TO GROWCIRCLE :SIZE
REPEAT 4 [RCP :SIZE]
GROWCIRCLE :SIZE + 1
END
```

Examples using RARC and LARC.

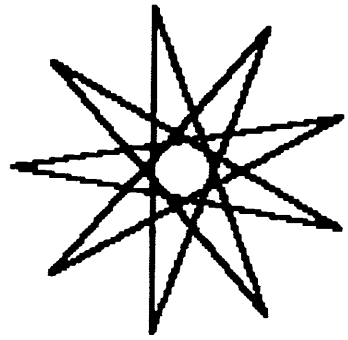


```
TO RAY :SIZE
RARC :SIZE
LARC :SIZE
RARC :SIZE
LARC :SIZE
END
```



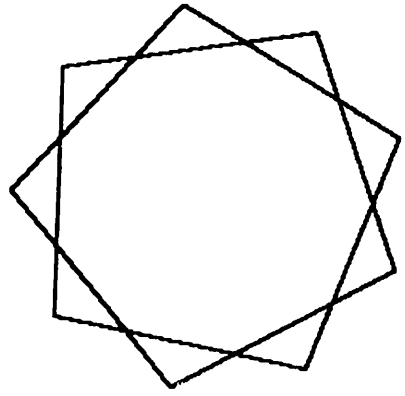
```
TO SUN :SIZE
RAY :SIZE
RIGHT 160
SUN :SIZE
END
```

POLY procedures have variable size and angle. Here are some examples.

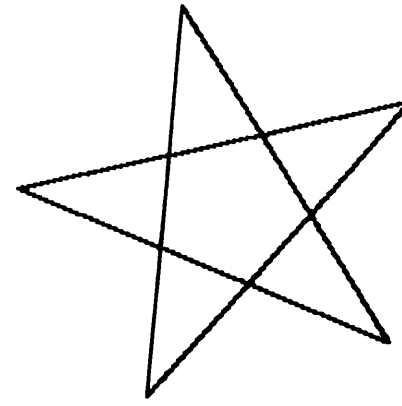


```
TO POLY :SIDE :ANGLE
FORWARD :SIDE
RIGHT :ANGLE
POLY :SIDE :ANGLE
END
```

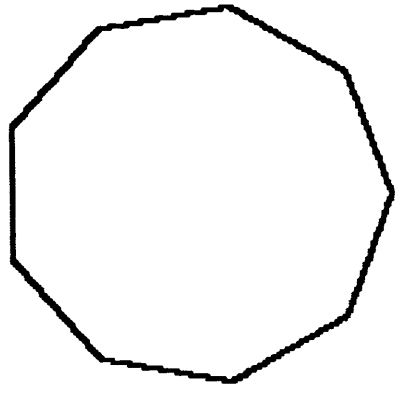
SIDE = 50 ANGLE = 160



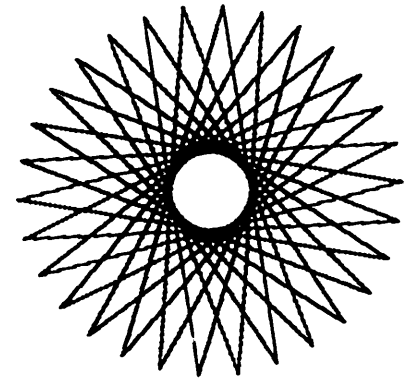
SIDE = 60 ANGLE = 80



SIDE = 80 ANGLE = 144



SIDE = 20 ANGLE = 40

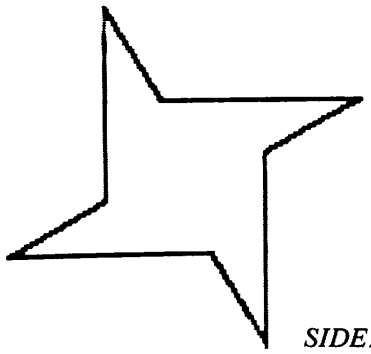


SIDE = 100 ANGLE = 156

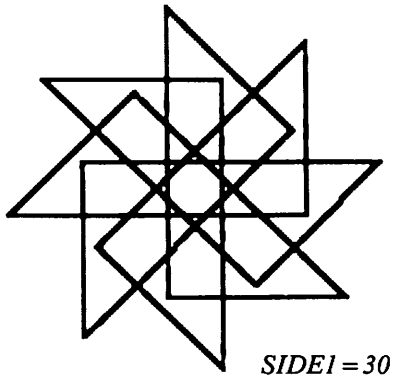
POLYSTEP is a piece of a POLY procedure. Here are some examples using it.

```
TO POLYSTEP :SIDE :ANGLE
FORWARD :SIDE
RIGHT :ANGLE
END
```

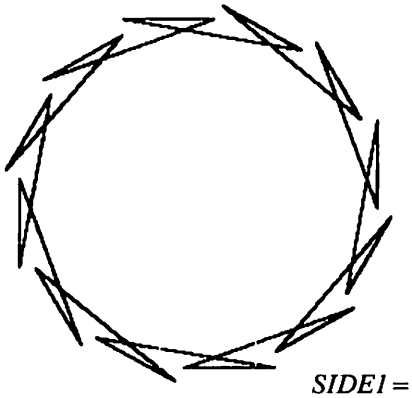
```
TO TWOPOLY :SIDE1 :ANGLE1 :SIDE2 :ANGLE2
POLYSTEP :SIDE1 :ANGLE1
POLYSTEP :SIDE2 :ANGLE2
TWOPOLY :SIDE1 :ANGLE1 :SIDE2 :ANGLE2
END
```



SIDE1 = 30 ANGLE1 = 60 SIDE2 = 60 ANGLE2 = 210

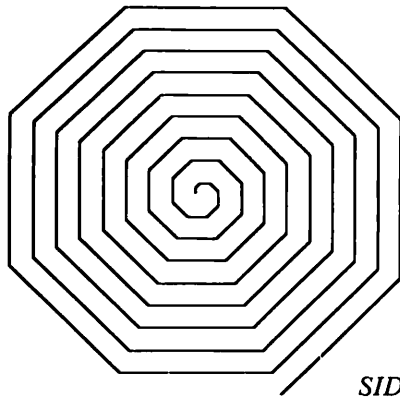


SIDE1 = 30 ANGLE1 = 90 SIDE2 = 50 ANGLE2 = 135



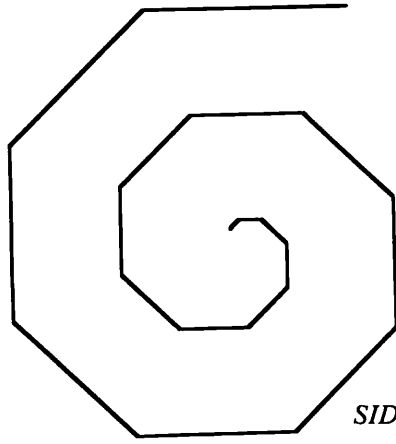
SIDE1 = 25 ANGLE1 = 190 SIDE2 = 50 ANGLE2 = 200

```
TO POLYSTEP :SIDE :ANGLE
FORWARD :SIDE
RIGHT :ANGLE
END

TO POLYSPIRAL :SIDE :ANGLE :INC
POLYSTEP :SIDE :ANGLE
POLYSPIRAL (:SIDE + :INC) :ANGLE :INC
END
```

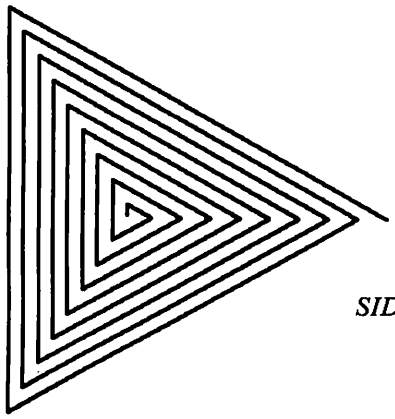


SIDE = 1 ANGLE = 45 INCREMENT = 1

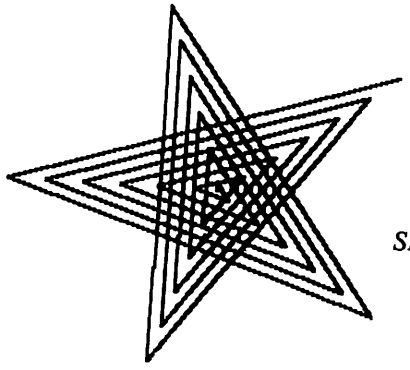
More programs using POLYSTEP. You may need to change the incrementing value inside of the procedure, that is, the value being added to the side each time the program recurses.



SIDE = 1 ANGLE = 45 INCREMENT = 3

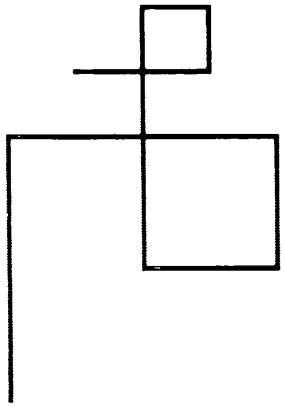


SIDE = 5 ANGLE = 120 INCREMENT = 3

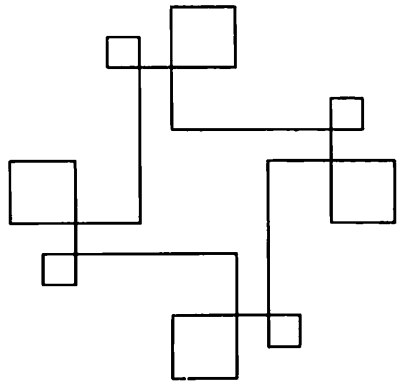


SIDE = 5 ANGLE = 144 INCREMENT = 3

Here's an example that begins by defining a shape and uses it to make a more interesting shape.



```
TO DESIGN
FORWARD 20
RIGHT 90
FORWARD 20
RIGHT 90
FORWARD 10
RIGHT 90
FORWARD 10
RIGHT 90
FORWARD 20
RIGHT 90
FORWARD 5
RIGHT 90
FORWARD 5
RIGHT 90
FORWARD 10
END
```

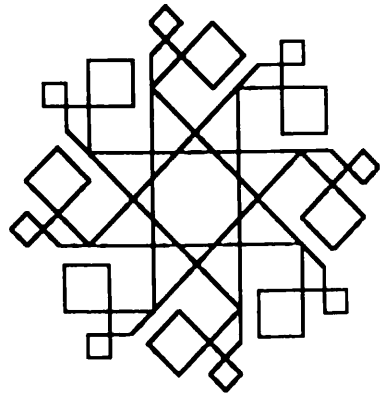


```
TO DESIGN4
DESIGN
DESIGN
DESIGN
DESIGN
END
```

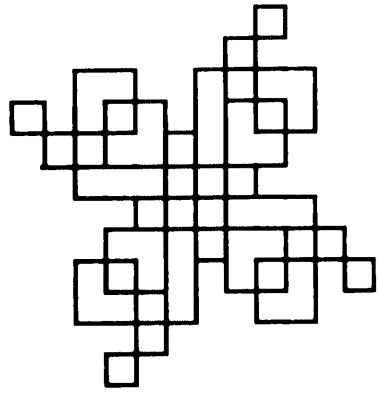
or

```
TO DESIGN4
REPEAT 4[DESIGN]
END
```

And two more shapes.

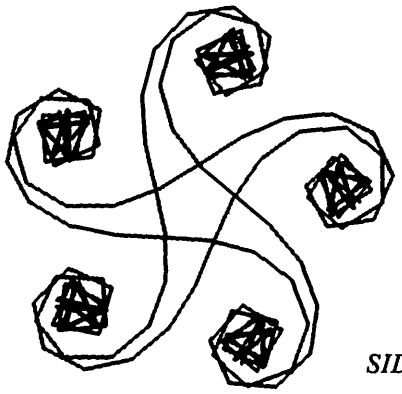


```
TO CRYSTAL
DESIGN
LEFT 45
FORWARD 35
CRYSTAL
END
```



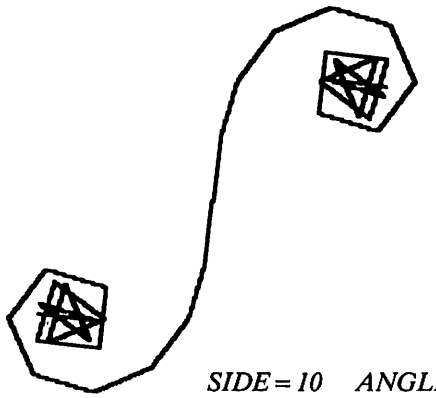
```
TO JENGU
DESIGN
DESIGN
LEFT 90
JENGU
END
```

Here are some programs using INSPI. Try various inputs.

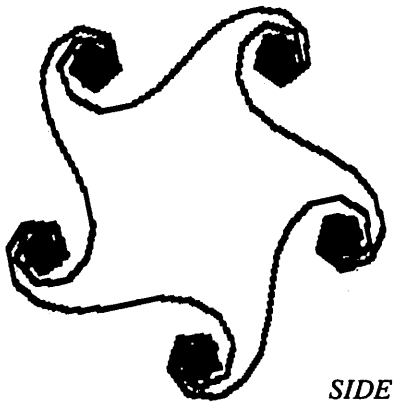


```
TO INSPI :SIDE :ANGLE
POLYSTEP :SIDE :ANGLE
INSPI :SIDE (:ANGLE + 10)
END
```

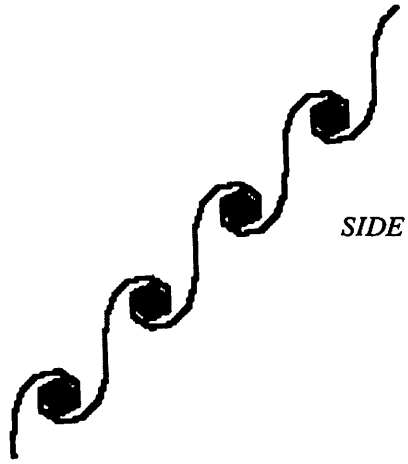
SIDE = 10 ANGLE = 1



SIDE = 10 ANGLE = 10



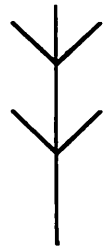
SIDE = 7 ANGLE = 3



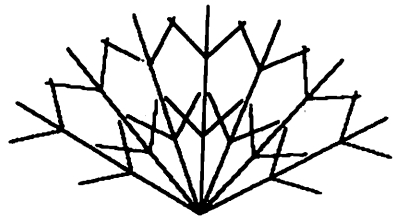
SIDE = 7 ANGLE = 5

Here is a sequence of procedures that start with a leaf (VEE) and end with a forest (TREES).

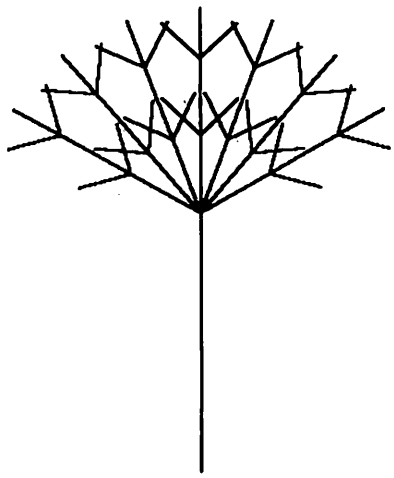
```
TO VEE
LEFT 45
FORWARD 10
BACK 10
RIGHT 90
FORWARD 10
BACK 10
LEFT 45
END
```



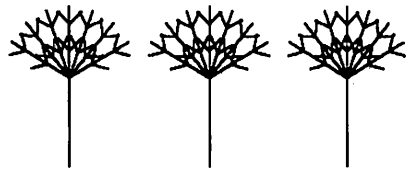
```
TO BRANCH
FORWARD 15
VEE
FORWARD 15
VEE
FORWARD 10
BACK 40
END
```



```
TO BUSH
LEFT 60
REPEAT 6 [BRANCH RIGHT 20]
BRANCH
LEFT 60
END
```

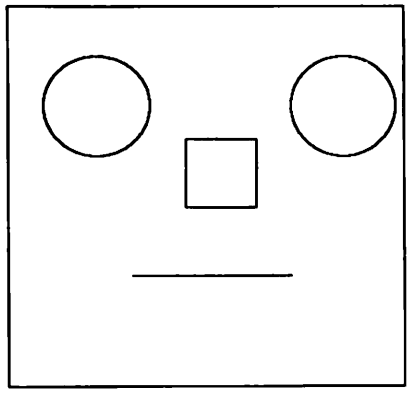


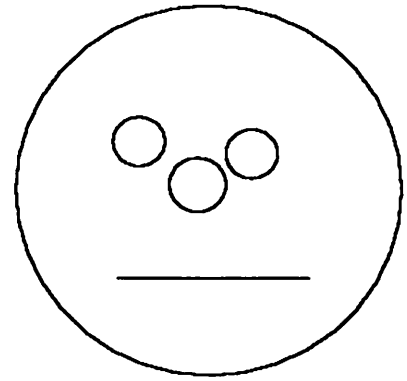
```
TO GREENTREE
FORWARD 50
BUSH
BACK 50
END
```

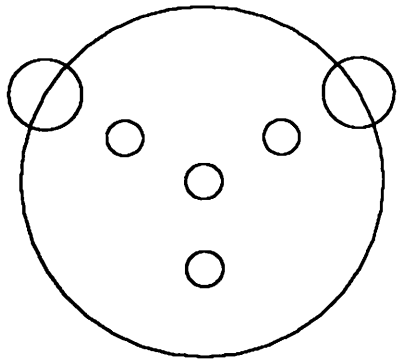


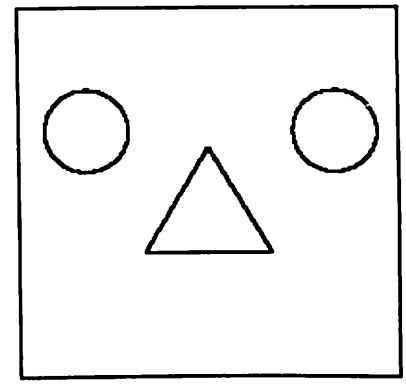
```
TO MOVE
PENUP
RIGHT 90
FORWARD 80
LEFT 90
PENDOWN
END

TO TREES
REPEAT 3 [GREENTREE MOVE]

MOVE
GREENTREE
MOVE
GREENTREE
END
```

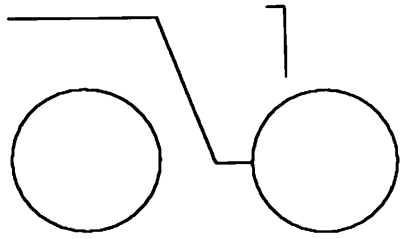
Here are some project ideas using the procedures you have already seen. Feel free to make up your own projects.

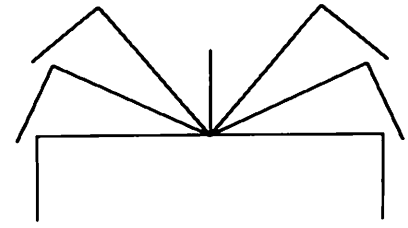


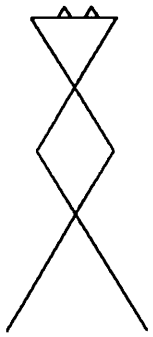


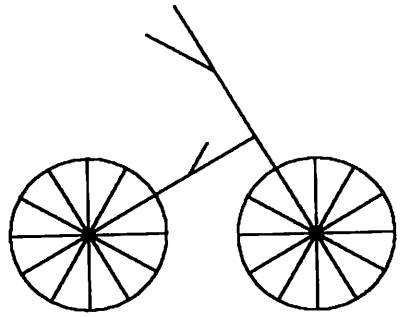


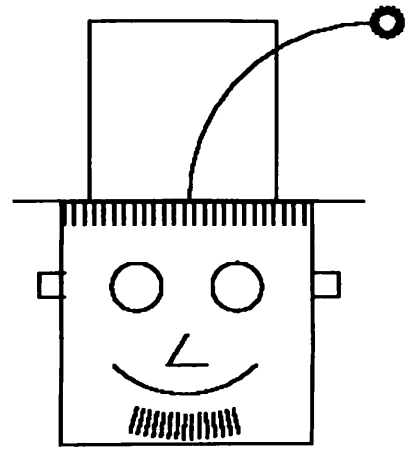


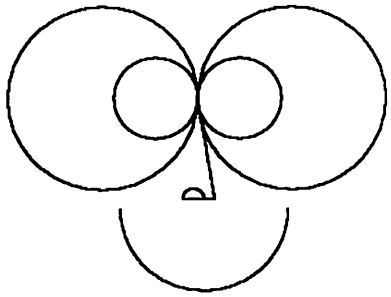


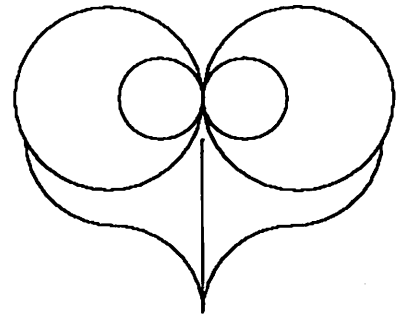












CHAPTER 4

Animation

We've seen how to use Logo to draw with the turtle. In this chapter, we show how to use *sprites* to make pictures that move. Sprites, like turtles, are graphical objects. Like turtles, they respond to commands FORWARD and RIGHT. But unlike turtles, sprites can change their color and their shape, and, most importantly, sprites can be *set in motion*. We begin by introducing the Logo commands for dealing with sprites, both one at a time and in groups, and give some simple procedures that control sprites. In Section 4.2 we show how you can define your own shapes for sprites to supplement the shapes that are built into Logo. Section 4.3 introduces *tiles*. Tiles, like sprites, can be given various shapes and colors, but they cannot move. They are useful for making elaborate backgrounds for screen graphics. In Section 4.4 we combine sprites, tiles, and Logo programming in the design of a simple movie.

4.1. Sprites

A sprite, like a turtle, is an object that lives on the computer display screen. Like a turtle, a sprite has a position and a heading, and responds to the commands FORWARD, BACK, RIGHT, and LEFT. Unlike turtles, sprites can be given various colors and shapes. You control a sprite's color by means of the SETCOLOR command (abbreviated SC), using any of the 16 color names (or color numbers) given in the chart on page 14. You can also tell a sprite to CARRY a given shape. Logo comes with five predefined shapes for sprites, given in the following chart below. You can also define your own shapes using the MAKESHAPE command as described in Section 4.2.

1	PLANE
2	TRUCK
3	ROCKET
4	BALL
5	BOX

4.1.1. Exploring with Sprites

To begin exploring with sprites, type

```
TELL SPRITE 1
```

(followed, as are all Logo command lines, by pressing ENTER). This indicates to Logo that subsequent commands will be addressed to sprite number 1. If

you have been using the turtle, clear it from the screen using the NOTURTLE command. Next type

```
HOME
```

As with the turtle, HOME places a sprite at the center of the screen.[1] At this point you won't see anything on the screen, because you have not given the sprite a color or a shape. Type

```
SETCOLOR :RED
CARRY :ROCKET
```

and you should see a red rocket at the center of the screen.[2]

The rocket responds to FORWARD and BACK commands. Typing

```
FORWARD 50
```

makes the rocket move up. If you type

```
RIGHT 90
```

you will not see any change on the screen. However, a subsequent

```
FORWARD 50
```

will make the rocket move sideways. As with the turtle, RIGHT and LEFT change the direction of motion. But unlike turtles, the *image* of the sprite on the screen does not rotate: a sprite shape always appears in the same orientation.

As with the turtle, you can change a sprite's heading by using the SETHEADING command. For example, typing

```
SETHEADING 90
```

will set the sprite's heading towards the right, and the next move it makes will be in that direction. TI Logo also includes the built-in names NORTH, EAST, SOUTH, and WEST that you can use together with SETHEADING. For instance,

```
SETHEADING :EAST
```

---

[1] HOME does not reset a sprite's heading as it does with the turtle.

---

[2] You can also obtain the same effect using the numbers of the color and shape:

```
SETCOLOR 6
CARRY 3
```

As described in the note on page 14, the use of the name and the colon reflects the general way in which names are used in Logo.

will give the same result as SETHEADING 90.

The most important difference between sprites and turtles is that sprites can be set in motion. This is done using the command SETSPEED (abbreviated SS). Type

```
SETSPEED 10
```

and the rocket will move slowly across the screen. Giving the command

```
SETSPEED 100
```

will make the rocket go much faster. In general, the SETSPEED command takes as input a number between −127 and 127.[3] Positive speeds make the sprite move in the direction of its heading. Negative speeds make it move in the opposite direction.

Now type

```
LEFT 90
```

This makes the rocket move vertically and illustrates that you can change a sprite's direction while it is moving. Giving a SETCOLOR command will change the rocket's color.

More Sprites

Now let's add another sprite to the picture:

```
TELL SPRITE 2
HOME
SETCOLOR :GRAY
CARRY:TRUCK
RIGHT 90
SETSPEED 5
```

makes a gray truck move slowly across the screen. Add another sprite:

```
TELL SPRITE 3
HOME
SETCOLOR :YELLOW
CARRY:BALL
RIGHT 45
SETSPEED 10
```

[3] A number outside this range will result in the error message *SETSPEED DOESN'T LIKE* {number} *AS INPUT*

Now you have a yellow ball moving diagonally. You can have more than one sprite carry the same shape. For example, you can add another truck:

```
TELL SPRITE 4
HOME
SETCOLOR :BLACK
CARRY :TRUCK
RIGHT 90
SETSPEED 8
```

4.1.2. Practice with Sprites

At this point you should take some time to play with sprites. There are 32 sprites in all, numbered from 0 through 31. Use TELL to pick a sprite, followed by CARRY, SETCOLOR, and HOME to give it a shape, a color, and an initial position. Then move it using FORWARD, BACK, LEFT, RIGHT, and SETSPEED. At any point, the sprite that responds to your command is the one that you designated by the previous TELL instruction.

Here are some things to note in your exploring:

Overlapping Sprites

When two sprites overlap, the one with the smaller number will appear to be on top.

Four Sprites on a Line

When you have many sprites on the screen, you will notice that some of them will flicker or partly disappear. This reflects a restriction built into TI Logo that *at most 4 sprites may appear on a horizontal screen line.* When there are more than 4 sprites on a horizontal line, the portions of the fifth, sixth, . . . , sprites on the same lines will be masked out. (The sprites that are masked are the higher numbered sprites.)

FREEZE and THAW

At any point, you can stop all motion by typing the FREEZE command. The THAW command restores the motion.

Making Sprites Disappear

The CLEARSCREEN will erase the text on the screen, but will not erase the sprites. To make a sprite vanish, tell it to SETCOLOR 0.

No Pen

Unlike the turtle, sprites cannot carry a pen and cannot leave a trail on the screen. Pen commands such as PENUP and PENDOWN are always ignored by sprites.

BIG and SMALL

Typing the command BIG will make all sprites double in size. The SMALL command restores sprites to their original size. (These commands are not included in TI Logo I.)

Coordinates for Sprites

As with the turtle, you can position a sprite on the screen with *x,y* coordinates using the SXY command. (See Figure 4.1) You can also obtain a sprite's position and heading using XCOR, YCOR, HEADING, SHAPE, and COLOR. In addition, XVEL and YVEL output the *x* and *y* components of the sprite's velocity. The command SV takes two numbers as inputs and changes the sprite's velocity by setting the *x* and *y* components of the velocity to these inputs. In all cases, the coordinates in question are those of the sprite specified by the most recent TELL command.

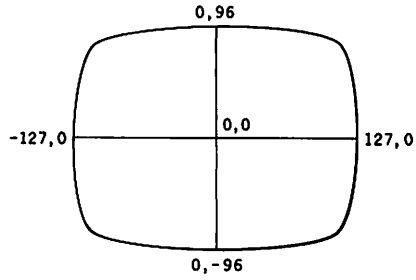


Figure 4.1: The x,y coordinate system for sprites.

Wraparound

When sprites move beyond an edge of the screen, they wrap around to reappear at the opposite edge. This wraparound behavior extends to other sprite attributes besides position. For example, color numbers for sprites "wrap around" after the maximum value of 15; thus SETCOLOR 16 is equivalent to SETCOLOR 0, SETCOLOR 17 to SETCOLOR 1, and so on. Shape numbers and sprite numbers behave similarly.

4.1.3. Talking to More Than One Sprite at a Time

So far we've seen how to control sprites using TELL, but only one sprite at a time. You can also use TELL to talk to a group of sprites all at once.

Typing

```
TELL :ALL
```

directs subsequent commands to all 32 sprites. For instance, you can use the following procedure to clear all sprites from the screen by setting their color to 0:

```
TO CLEARSPRITES
TELL :ALL
SETCOLOR 0
END
```

You can also give TELL a list of sprite numbers, and subsequent commands will be directed to those sprites. For instance,

```
TELL [1 2 3]
CARRY :BALL
TELL [4 5 6]
CARRY :TRUCK
```

will give a ball shape to sprites 1, 2, and 3, and a truck shape to sprites 4, 5, and 6. In fact, TELL :ALL is a special case of this, because ALL in TI Logo is just a name for the list of numbers 0 through 31.

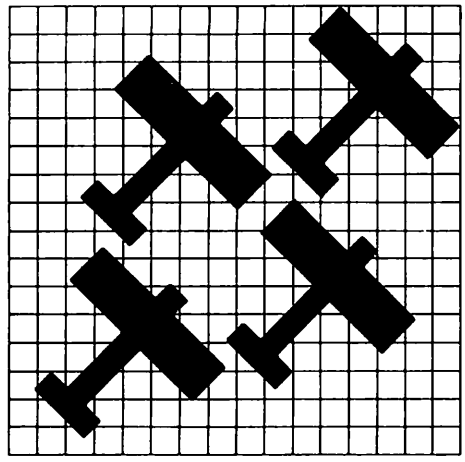


Figure 4.2: A squadron of four planes flying in formation.

Naming Groups of Sprites

It is often convenient to be able to refer to a group of sprites by name, rather than by typing the list of numbers. For example, let's make a squadron of four planes flying in formation, as shown in Figure 4.2.

We'll use sprites 1, 2, 3, and 4 to carry the planes, so we'll give the name SQUADRON to the list [1 2 3 4]:

```
MAKE "SQUADRON [1 2 3 4]
```

This illustrates the general way in which the Logo command MAKE is used to name things. Notice that the name SQUADRON is preceded by a quotation mark. Once we've given the MAKE command, we can refer to the list [1 2 3 4] as :SQUADRON. We'll discuss the use of MAKE more fully in Section 6.5.

Here is a procedure that initializes the squadron of planes. It first stops all four sprites (in case they had been moving) and sets them at the center of the screen, pointing to the right (heading 90). Then it spreads sprites 1 and 2 a bit horizontally, sprites 3 and 4 a bit vertically, and puts the squadron in motion.

```
TO SQUAD
MAKE "SQUADRON [1 2 3 4]
TELL :SQUADRON
SETSPEED 0
HOME
SETHEADING 90
CARRY :PLANE
SETCOLOR :BLACK
TELL 1 FORWARD 20
TELL 2 BACK 20
TELL 3 LEFT 90 FORWARD 10 RIGHT 90
TELL 4 LEFT 90 BACK 10 RIGHT 90
TELL :SQUADRON SETSPEED 20
END
```

Notice how TELL is used to direct commands either to the individual sprites or to the entire squadron. Once you have given the SQUAD command, you can fly the squadron around using SETSPEED, RIGHT, and LEFT. (When SQUAD terminates, the effect of its final command, TELL :SQUADRON, will cause subsequent commands to still be directed to the entire SQUADRON.)

EACH and YOURNUMBER

It is often useful to be able to talk to a group of sprites all at once, but to have each sprite do a slightly different thing. The Logo command EACH takes a list of commands as input and processes these commands for each of the sprites you are currently talking to (as specified by the previous TELL). EACH is most useful in conjunction with the Logo command YOURNUMBER (abbreviated YN), which outputs the number of the current sprite.

For example, set your squadron flying across the screen as above, and type

```
TELL :SQUADRON
EACH [SETSPEED 10 * YOURNUMBER]
```

The planes will break formation, because they are now going at different speeds: sprite 1 at 10, sprite 2 at 20, sprite 3 at 30, and sprite 4 at 40. Notice that the commands for EACH are enclosed in brackets as a list, just as with REPEAT (Section 1.3.4).

```
EACH [SETCOLOR YOURNUMBER]
```

will set the planes to different colors: sprite 1 to color 1 (black), sprite 2 to color 2 (green), sprite 3 to color 3 (lime), and sprite 4 to color 4 (blue).

Combining EACH and REPEAT yields a clever little procedure to help you explore with sprites:[4]

```
TO SPREAD :COMMANDS
EACH [REPEAT YOURNUMBER :COMMANDS]
END
```

Try the following

```
TELL :ALL
SETSPEED 0
HOME
SETHEADING 0
CARRY :BALL
SETCOLOR :RED
SPREAD [RIGHT 10]
SETSPEED 20
```

The SPREAD points the sprites at 10-degree increments all along a circle. When you start them all going, the effect is that of a circle of sprites exploding outward from the center of the screen.[5] Notice that the input to SPREAD is a *list* of commands. The use of YOURNUMBER as the first input to REPEAT in SPREAD means that SPRITE 0 will repeat the list of commands zero times, sprite 1 will repeat it one time, sprite 2, two times, etc. Remember to type the input to SPREAD as a list, as in SPREAD [RIGHT 10].

Here's another nice thing to do with SPREAD:

```
TELL [1 2 3 4 5 6]
SETSPEED 0
SETHEADING 0
HOME
CARRY :BALL
SETCOLOR :RED
SPREAD [FORWARD 30 RIGHT 60]
```

---

[4]This example is courtesy of A. diSessa.

[5]There's an interesting phenomenon lurking here: as soon as the sprites begin to wrap around the borders of the screen, the pattern starts to look random. But if you say SETSPEED −20, causing the sprites to reverse direction, they will eventually all converge at the center of the screen.

This positions the 6 sprites at the vertices of a regular hexagon. To see why, think about the relation between this and the POLY procedure (page 32). By choosing different numbers of sprites and different angles, you can make similar patterns based on other regular polygons.

To WHO(m) Are You Talking?

With all these possibilities for TELL, it is useful to be able to check which sprite or group of sprites you are currently talking to. The Logo command WHO indicates the sprite or list of sprites to which commands are currently directed. For example:

```
TELL SPRITE 1
FORWARD 100
PRINT WHO
SPRITE 1

TELL :SQUADRON
SETSPEED 20
PRINT WHO
1 2 3 4
```

If you are talking to the turtle or the background, then WHO will output TURTLE or BACKGROUND.

4.2. Defining Shapes

The shape carried by a sprite can be any of the five shapes built into Logo. The Logo MAKESHAPE command (abbreviated MS) allows you to create your own shapes or to modify any of the built-in shapes.

To define a new shape, first decide what number shape you are defining. The built-in shapes have the numbers given in the table on page 51. In general, you can have 26 different shapes, numbered 0 through 25.

Suppose you want to modify the PLANE shape (shape 1). Type

```
MAKESHAPE 1
```

You will see on the screen a 16 × 16 grid of small squares with the plane design blacked in on the grid, as shown in Figure 4.3. The background of the screen has also changed color to indicate that you are now using the *shape editor.*

To change the shape, you move the cursor around on the grid, blacking in new squares and whiting out others. The keys that move the cursor are the keys marked with arrows: S, D, E, and X. Pressing any of these keys moves the cursor in the direction of the corresponding arrow. (The cursor wraps around if you move it past the edge of the grid.) When you move the cursor

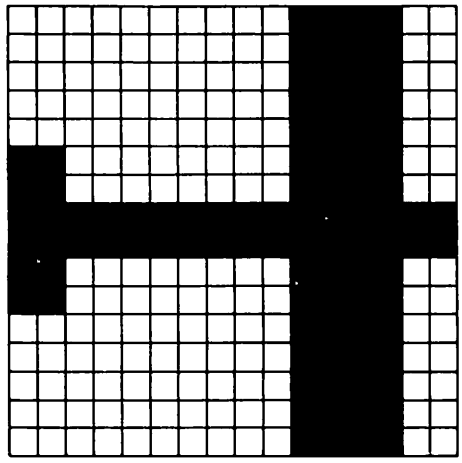


Figure 4.3: Appearance of the screen in response to MAKESHAPE 1.

out of a square, it leaves that square blank. If you hold down the FCTN key and move the cursor, then the square that the cursor leaves will be blacked in.[6] Try changing the shape of the plane's wings, as shown in Figure 4.4.

When you are finished changing the shape, press the BACK key. Logo exits the shape editor, and the new shape will be installed as shape number 1 (PLANE). Any sprite you tell to CARRY :PLANE or CARRY 1 will now have the new appearance.

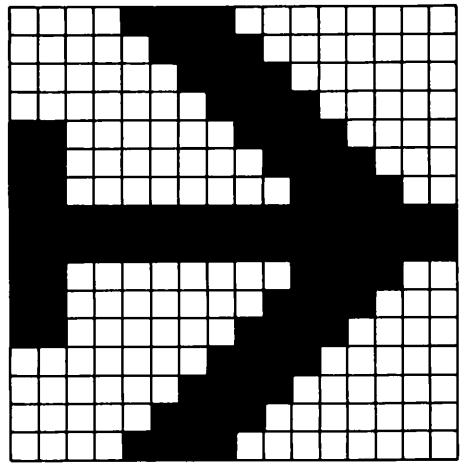


Figure 4.4: A new PLANE shape.

4.2.1. Example: Birds Flying

As an example of defining new shapes, let's make some flying birds. Begin by using the shape editor to define two shapes corresponding to birds with wings up and wings down. For these, use two new shape numbers 6 and 7. It is good practice to assign mnemonic names to the shapes you use and to work with the names rather than with the numbers directly:

[6]Use FCTN on the TI 99/4A, SHIFT on the 99/4.

```
MAKE "UPWING 6
MAKE "DOWNWING 7
```

Now use the shape editor:

```
MAKESHAPE :UPWING
```

and

```
MAKESHAPE :DOWNWING
```

to define the shapes shown in Figure 4.5.

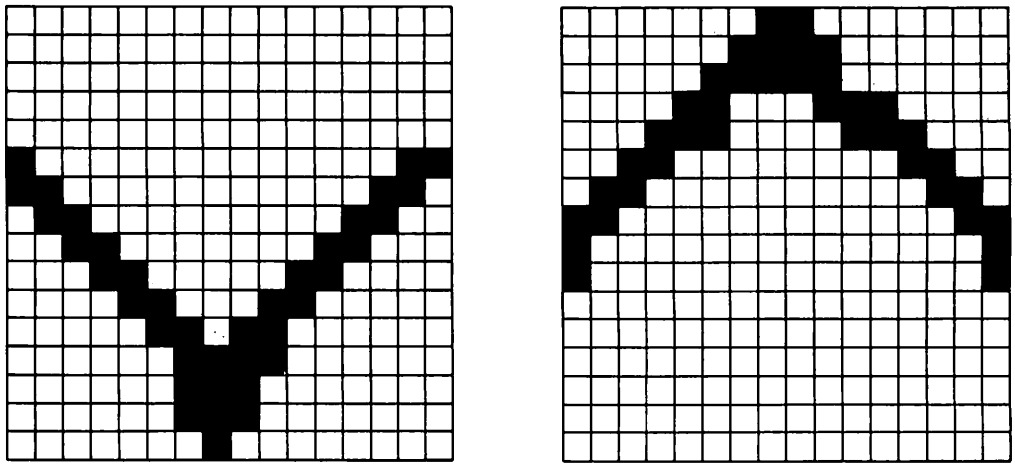


Figure 4.5: UPWING and DOWNWING shapes for flying birds.

Now set up a group of 6 sprites, called BIRDS, to carry these shapes:

```
TO SETBIRDS
MAKE "BIRDS [1 2 3 4 5 6]
TELL :BIRDS
SETSPEED 0
HOME
SETCOLOR :WHITE
CARRY :UPWING
SETHEADING 45
SPREAD [FORWARD 20]
SETHEADING 90
END
```

The SPREAD procedure (See Section 4.1.3) gives a useful way to spread the sprites out along a diagonal. Now you can set the birds in motion:

```
TELL :BIRDS
SETSPEED 20
```

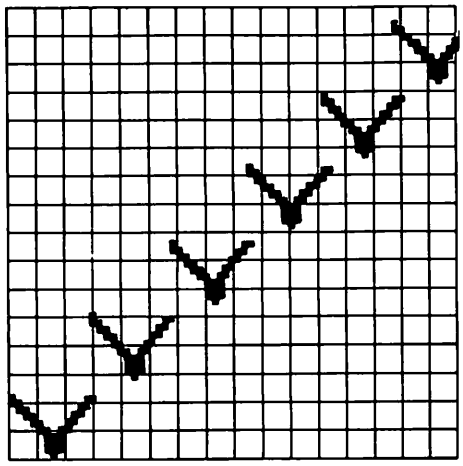


Figure 4.6: A line of birds created by SETBIRDS.

The result, shown in Figure 4.6, is a line of 6 birds drifting across the screen, all with their wings up.

Typing

```
CARRY :DOWNWING
```

makes the birds put their wings down. Alternating this with

```
CARRY :UPWING
```

makes the wings flap. You can define a procedure that continues the flapping:

```
TO FLAP
CARRY :UPWING
WAIT 30
CARRY :DOWNWING
WAIT 30
FLAP
END
```

The Logo command WAIT makes the computer wait for a given number of sixtieths of a second. In this case the procedure is waiting 1/2 second between wing beats.

You now have all the ingredients for a simple movie:

```
SETBIRDS
SETSPEED 20
FLAP
```

The birds will continue flapping until you stop the procedure by pressing the BACK key.

4.2.2. Two Notes on the Shape Editor

When you enter the shape editor using MAKESHAPE, the sprites will still be visible on the screen. You can take advantage of this fact. Before typing MAKESHAPE, tell one or more sprites to CARRY the shape you are defining. Then, as you mark in the grid using the shape editor, you will see the shape in its actual size carried by the sprite.

When the shape editor is in use, the only keys that have any effect are the arrow keys, BACK (to exit to Logo) and CLEAR, which clears all the squares on the grid.

4.3. Tiles

The final kind of graphical object that you can address with TELL in Logo is called a *tile*. A tile, like a sprite, can have a color and a shape. Unlike a sprite, a tile cannot move. Tiles are useful in designing backgrounds for graphics.

You can have 256 different tiles, numbered 0 through 255. In making screen backgrounds, you should normally avoid using tiles 0 through 10 and 32 through 95, since these are used to create screen characters as described in Section 4.3.3. In particular, tiles 0 and 1 are used for the regular cursor and the shapes editing cursor, and cannot be changed.

To design a tile, you use the Logo command MAKECHAR (abbreviated MC). This works almost identically to MAKESHAPE (Section 4.2) except that the grid for a tile is only 8 squares on a side—one quarter the size of the grid for a sprite. For example, you can use MAKECHAR to design a tile that looks like a small spiral. Let this be, say, tile 96:

```
MAKECHAR 96
```

You now obtain a grid on which you can draw the spiral, as shown in Figure 4.7.

Once you have defined a tile, you can give it a color using TELL and SETCOLOR:

```
TELL TILE 96
SETCOLOR :RED
```

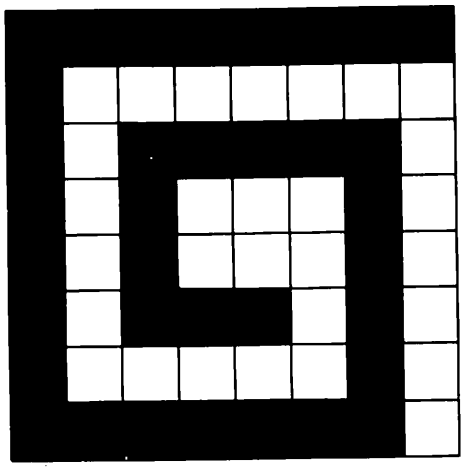


Figure 4.7: Using MAKECHAR to make a tile in the shape of a spiral.

4.3.1 Positioning Tiles on the Screen

To place a tile on the screen you use the PUTTILE command (abbreviated PT). This takes as input the number of the tile you wish to put on the screen, followed by the screen position, specified as a column number and a row number. Figure 4.8 shows how the screen can be viewed as a grid numbered by columns and rows. There are 32 columns, numbered 0 through 31, and 24 rows, numbered 0 through 23.

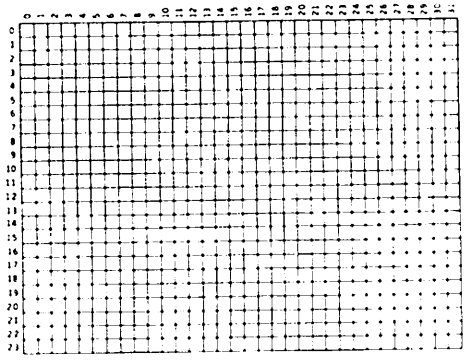


Figure 4.8: The display screen as a grid of columns and rows.

The center of the screen is column 16, row 12. Hence, with tile 96 defined as above, you can place a small red spiral at the center of the screen with

```
PUTTILE 96 16 12
```

Figure 4.9 shows more spirals placed on the screen, via

```
PUTTILE 96 16 12
PUTTILE 96 10 10
PUTTILE 96 20 20
PUTTILE 96 10 20
PUTTILE 96 5 5
```

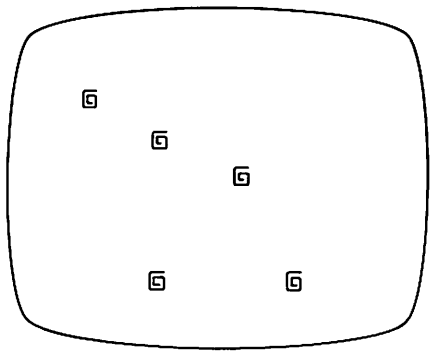


Figure 4.9: Using PUTTILE to place spirals on the screen.

Warning

Observe that the column and row numbers used with PUTTILE to place a tile on the screen are *not* the same as the *x,y* coordinates used with SXY to position sprites and turtles on the screen. You cannot use SXY (or FORWARD, LEFT, and so on) when talking to tiles. Section 7.4.1 includes some useful procedures that will aid you in dealing with this problem.

4.3.2. Foreground and Background Colors

We saw just above that tiles can be given a color by using the SETCOLOR command. Actually, each tile has *two* colors associated with it: a *foreground color* and a *background color.* When you show a tile, the blacked-in squares (on the MAKECHAR grid) will be shown in the foreground color, and the other squares on the grid will be shown in the background color. If you do not specify a background color, as in

```
SETCOLOR :RED
```

then only the foreground color will be changed.[7] To set a background color, use SETCOLOR with a list of the two color numbers. For instance, to give the spiral defined above (tile 96) a foreground color of RED (color 6) and a background color of white (color 15), you type

```
TELL TILE 96
SETCOLOR [6 15]
```

If you give this command with the spirals on the screen, as in Figure 4.9, each will appear as a red spiral within a white square.[8]

---

[7]The background color of a tile is initially defined to be CLEAR (color 0) so that you will not see the background.

[8]You can accomplish the same color change using color names instead of numbers by typing

SETCOLOR SENTENCE :RED :WHITE

The Logo command SENTENCE, as we shall see in Section 6.4 is used to construct lists.

Color Groups

One restriction on tile colors is that every group of 8 tiles must have the same color. That is, tiles 0 through 7 must have the same color, tiles 8 through 15 must have the same color, and so on. Changing the color of a tile (either foreground or background color) changes the color of every tile in the group.

4.3.3. Characters as Tiles

The characters that Logo prints on the screen are, in fact, defined as tiles, using tile numbers 32 through 95. The following chart shows the correspondence between tile numbers and characters, and how these tiles are arranged in groups:

Group 1		Group 2		Group 3		Group 4	
Code Number	*Character*	*Code Number*	*Character*	*Code Number*	*Character*	*Code Number*	*Character*
32	(space)	40	(	48	0	56	8
33	!	41	)	49	1	57	9
34	"	42	*	50	2	58	:
35	#	43	+	51	3	59	;
36	$	44	,	52	4	60	<
37	%	45	-	53	5	61	=
38	&	46	.	54	6	62	>
39	'	47	/	55	7	63	?

Group 5		Group 6		Group 7		Group 8	
Code Number	*Character*	*Code Number*	*Character*	*Code Number*	*Character*	*Code Number*	*Character*
64	@	72	H	80	P	88	X
65	A	73	I	81	Q	89	Y
66	B	74	J	82	R	90	Z
67	C	75	K	83	S	91	[
68	D	76	L	84	T	92	\
69	E	77	M	85	U	93	]
70	F	78	N	86	V	94	^
71	G	79	O	87	W	95	_

You can take advantage of this to modify the way in which Logo prints characters. For instance, typing

```
MAKECHAR 52
```

allows you to edit tile 52, which is the numeral 4. If you change it as shown in Figure 4.10, then all the 4s printed by Logo from then on will have this new shape. Tiles 2 through 10 are initially used for miscellaneous portions of the TI master title screen.[9]

[9]For instance, tile number 4 is in the shape of West Texas.

Another thing you can do is use SETCOLOR to change the color in which characters are printed on the screen.

```
TELL TILE 48
SETCOLOR :WHITE
TELL TILE 56
SETCOLOR :WHITE
```

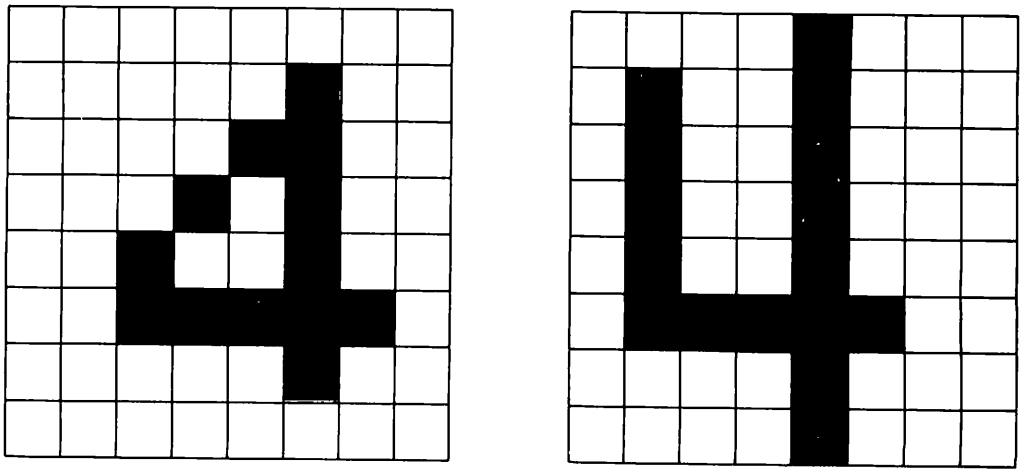


Figure 4.10: Using MAKECHAR 52 to define a new style for the numeral 4.

will cause all tiles in the same group as 48 and 56 (that is tiles 48 through 55 and 56 through 63, which includes all the numerals and :, ;, <, =, >, ?) to be printed in white, while the other characters will be printed in black.

Turtle Lines as Tiles

Another way that tiles are used in Logo is to create the lines drawn by the turtle. Each time the turtle draws inside a small screen square, a tile is created whose "shape" is the turtle line. As the turtle draws in more and more squares, it uses more and more tiles, using first tiles 0 through 32 and then tiles 96 through 255. For this reason, if you have some tiles defined and then use the turtle, the tile shapes may be destroyed. This also explains the meaning of the OUT OF INK error message (page 6). The turtle is "out of ink" when all available tiles have been used.

You can also set a background color when drawing with the turtle. Typing

```
TELL TURTLE
SETCOLOR [{color1} {color2}]
```

and drawing with the pen down will cause each square the turtle passes over to be filled in with {color2}, with a thin line in {color1} drawn through the square.

4.4. Project: A Simple Movie

In this section, we'll combine what we've learned about sprites and tiles to create a simple movie. The movie, shown in Color Plate I; shows an ocean with waves and whitecaps. A boat sails along the ocean and birds fly overhead. This project also illustrates how a substantial Logo program is designed and implemented as a cluster of simple procedures.

If you look at Plate I, you'll see that there are five parts to the picture: the sky, the ocean, the sun, the boat, and the birds.

The sky is simple. We'll simply use the screen background after changing its color to SKY (color 5). The ocean and the sun will be assembled from tiles, and the boat and the birds will be sprites.

The Ocean

Let's begin by drawing the ocean. There are two parts: the top row, consisting of the waves, and the rest of the water. Start with the rest of the water, using a tile that will give a square of blue water with a small whitecap. Call this tile WATER and use tile 108:

```
MAKE "WATER 108
MAKECHAR :WATER
```

Figure 4.11 shows the water tile that you make with the MAKECHAR command. The color should be a foreground color of WHITE for the whitecap (color 15) and a background color of BLUE (color 4) for the water.

For the top row of waves, use three tiles: WAVE1, WAVE2, and WAVE3, whose shapes are also shown in Figure 4.11. The color for these tiles is foreground BLUE, background CLEAR. Since the color is different from the color for the WATER tile, you must create the waves using tiles in a different color group. (See Section 4.3.2.) You can use tiles 100, 101, and 102 for the waves:

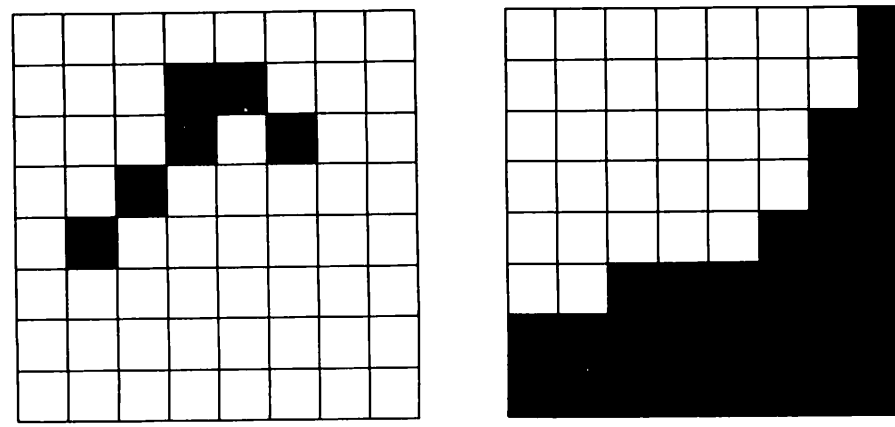


Figure 4.11: Shapes for water and wave tiles.

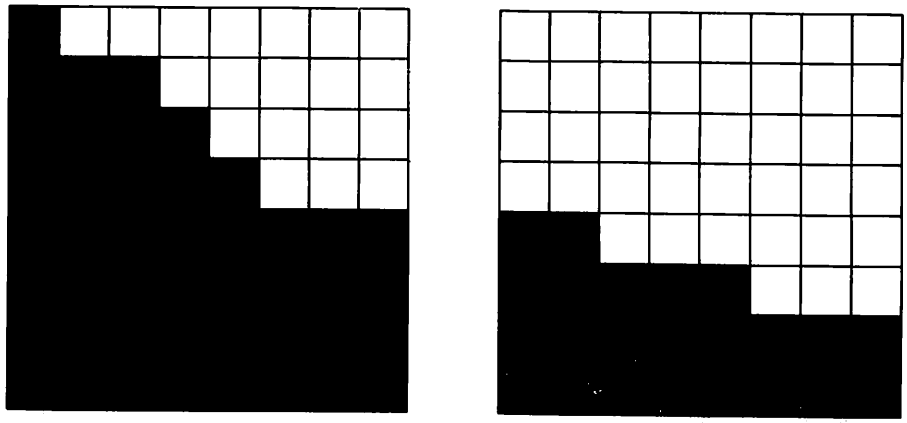


Figure 4.11: (Continued)

```
MAKE "WAVE1 100
MAKE "WAVE2 101
MAKE "WAVE3 102
```

Now we'll draw the top row of waves on the screen. The idea is to start at the leftmost column of the screen, placing a sequence of three tiles, WAVE1, WAVE2, WAVE3, and repeat this over and over for as many columns as there is room. A good position for the top row of the ocean is row 15, which we'll call OCEANTOP:

```
MAKE "OCEANTOP 15
```

The following procedure draws the top row of the ocean. It is called initially with the starting column as input:

```
TO WAVETOPS :COL
IF:COL > 31 STOP
PUTTILE :WAVE1 :COL :OCEANTOP
PUTTILE :WAVE2 :COL + 1 :OCEANTOP
PUTTILE :WAVE3 :COL + 2 :OCEANTOP
WAVETOPS :COL + 3
END
```

You should compare this recursive "test and stop" procedure form with the COUNTDOWN or TOWER procedures discussed in Section 2.2. To actually draw the waves, you can now give the command:

```
WAVETOPS 0
```

The rest of the ocean is drawn by using PUTTILE with the WATER tile. Here are two useful procedures that fill the screen with a given tile, starting from a specified top row:

```
TO MAKEROWS :TILE :TOPROW
IF :TOPROW > 22 STOP
MAKEROW :TILE 0 :TOPROW
MAKEROWS :TILE :TOPROW + 1
END

TO MAKEROW :TILE :COL :ROW
IF:COL > 31 STOP
PUTTILE :TILE :COL :ROW
MAKEROW :TILE :COL + 1 :ROW
END
```

These also use a recursive scheme similar to COUNTDOWN (Section 2.2). MAKEROW fills in a single row by placing tiles in successive columns until it reaches column 31. MAKEROWS calls MAKEROW on successive rows until it reaches the bottom of the screen at row 22. With these procedures, the body of the ocean can now be drawn as

```
MAKEROWS :WATER :OCEANTOP + 1
```

(The top ocean row is one below the wave tops.)

Here, then, is a procedure that draws the complete ocean:[10]

```
TO MAKEOCEAN
TELL TILE :WATER
SETCOLOR [15 4]
TELL TILE :WAVE1
SETCOLOR [4 0]
WAVETOPS 0
MAKEROWS :WATER :OCEANTOP + 1
END
```

The Sun

The sun in the picture is formed from 4 tiles, each a quarter circle:

```
MAKE "SUN1 113
MAKE "SUN2 114
MAKE "SUN3 115
MAKE "SUN4 116
```

[10] All *three* tiles names WAVE1, WAVE2, and WAVE3 have their colors set to foreground BLUE, background CLEAR, by telling any *one* of them to do so—this is because they are all part of the same color group.

Since these are to be yellow, you must pick tiles in a different color group than either the waves or the water. Figure 4.12 shows the four sun tiles constructed using MAKECHAR.

The following procedure positions the tiles on the screen:

```
TO MAKESUN
TELL TILE :SUN1
SETCOLOR :YELLOW
PUTTILE :SUN1 25 7
PUTTILE :SUN2 26 7
PUTTILE :SUN3 25 8
PUTTILE :SUN4 26 8
END
```

Notice that setting the color of SUN1 sets all 4 sun tiles since they are in the same color group.

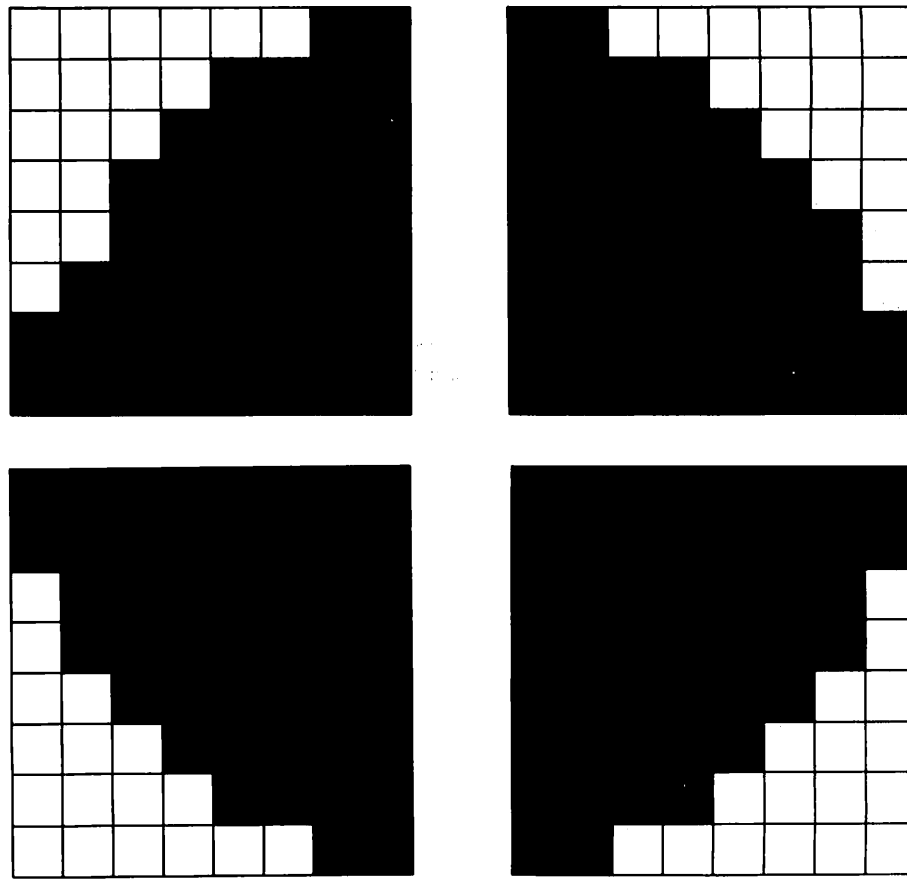


Figure 4.12: The sun constructed from four tiles.

The Boat

The boat is a single sprite that moves horizontally across the screen. You can define a BOATSHAPE as shape number 9:

```
MAKE "BOATSHAPE 9
MAKESHAPE :BOATSHAPE
```

and construct the shape shown in Figure 4.13.

Use sprite 0 to carry the boat:

```
MAKE "BOAT 0
```

Here is the procedure that sets up the boat:

```
TO MAKEBOAT
TELL :BOAT
SETHEADING 90
SETCOLOR :BLACK
SXY (-50) (-50)
CARRY :BOATSHAPE
SETSPEED 5
END
```

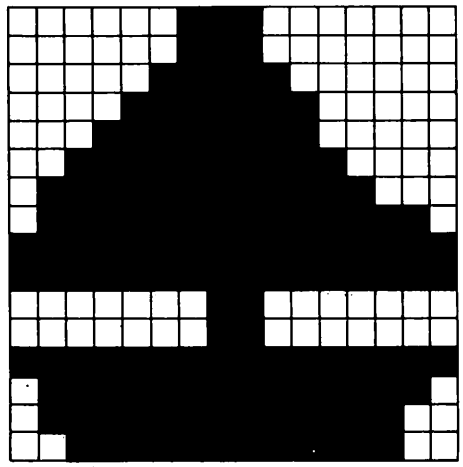


Figure 4.13: The BOATSHAPE for the movie.

The Birds

Finally, we'll add birds flying across the screen. You do this by incorporating the bird shapes and the SETBIRDS and FLAP procedures from Section 4.2.1, with a few small changes.

As before we'll have 6 birds, using sprites 1 through 6:

```
MAKE "BIRDS [1 2 3 4 5 6]
```

The bird shapes will be UPWING and DOWNWING as shown in Figure 4.5:

```
MAKE "UPWING 6
MAKE "DOWNWING 7
```

The FLAP procedure is the same as before:

```
TO FLAP
CARRY :UPWING
WAIT 30
CARRY :DOWNWING
WAIT 30
FLAP
END
```

Setting up the birds will be similar to the SETBIRDS procedure on page 58, with a few changes to make the movie more interesting. First of all, you can make the birds be different colors. Colors 8 through 15 give a good set of colors, so we can set each sprite 1 through 8 to the color 7 + YOURNUMBER. You can also have the birds travel at slightly different speeds and at slightly different headings. This latter is conveniently done using the SPREAD procedure (page 56).

```
TO MAKEBIRDS
TELL :BIRDS
CARRY :UPWING
EACH [SETCOLOR 7 + YOURNUMBER]
SETSPEED 0
HOME
SETHEADING 90
SPREAD [LEFT 1]
SPREAD [FORWARD 10]
EACH [SETSPEED 5 + YOURNUMBER]
END
```

Putting It All Together

The only thing that remains is to write a procedure that puts all the parts together. This clears the screen, sets up the sky by changing the BACKGROUND color to SKY (color 5), draws all the parts of the picture, and sets the birds flapping:

```
TO MOVIE
CLEARSCREENANDSPRITES
MAKESKY
MAKEOCEAN
MAKESUN
MAKEBOAT
MAKEBIRDS
TELL :BIRDS FLAP
END

TO CLEARSCREENANDSPRITES
CLEARSCREEN
TELL :ALL CARRY 0
END

TO MAKESKY
TELL BACKGROUND
SETCOLOR :SKY
END
```

This completes the movie, as well as a substantial Logo project. Notice how we were able to isolate the different parts of the project by using separate procedures. Even though the program as a whole is long, the individual procedures are rather short and could be designed and tested separately. This is one of the important advantages of Logo's procedural organization.

To save these procedures and shapes on a cassette tape or floppy disk, see Section 5.2, "Saving and Retrieving Information." To save *both* your procedures *and* your tile and sprite shapes, be sure to choose option 3.

CHAPTER 5

Workspace, Filing, and Debugging

When you use the Logo system, you can think of the computer as having two memories. The first memory, called *workspace*, is where Logo keeps track of the procedures and variables you have defined. Each time you define a procedure or assign a value to a name, that information becomes part of the workspace. When you leave Logo by pressing QUIT or by turning off the computer, the information in the workspace is destroyed. The second and more permanent memory consists of files that you save on diskette or on cassette tape. Each file contains a complete workspace.[1] The normal way to use the Logo file system is to work for a while on a project and then, when you are finished for the day, to save your workspace in a file. The next time you use Logo, you can read in your saved workspace and continue where you left off. You can also maintain a number of different files containing the workspaces for different projects you are working on.

5.1. Managing Workspace

The Logo system includes commands for examining and deleting various parts of the workspace. These are useful for keeping track of what procedures are currently defined and for getting rid of unwanted definitions.

5.1.1. PO

The basic command for examining workspace is PO. If you type PO followed by the name of a procedure, Logo prints the definition of the procedure. PO also has a few variants:

PP Prints out the titles of all *procedures* in workspace.
PN Prints out the *names* and associated values in the global library.
PA Prints out *all* the procedures and names in the workspace.

5.1.2. ERASE

The ERASE command is used to get rid of parts of the workspace. ERASE followed by a procedure title name removes the definition of the procedure. ERASE followed by a variable name, as in

```
ERASE "X
```

[1] You can also elect to save only the procedures or only the sprite and tile shapes contained in the workspace.

will erase a variable name from the global library. (In this example, the variable name X.)

5.2. Saving and Retrieving Information

The Logo system allows you to save procedure definitions on diskette or on cassette tape, or to print them on a printer. The Logo commands SAVE and RECALL are used for storing and retrieving.

When you give the SAVE command, Logo first displays a menu, as shown in Figure 5.1, asking you to indicate whether you wish to save (1) only your procedures and names, (2) only your shapes, or (3) both of these. Press 1, 2, or 3 to indicate your choice. (You ordinarily choose 3 when saving new shapes—otherwise choose 1.) Next, Logo asks you to indicate the device used to save information, as shown in Figure 5.2.

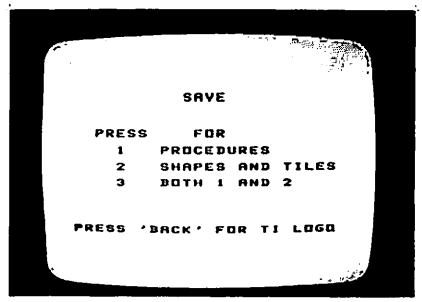


Figure 5.1: Screen display menu offering SAVE options.

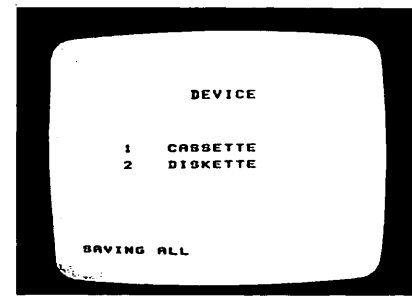


Figure 5.2: Screen display menu offering SAVE device options.

5.2.1. Using Cassette Tape

Pages 1-8 through 1-12 of the User's Reference Guide that comes with your TI 99/4 or 99/4A contain detailed information about setting up and using a cassette recorder to save and recall information. It is particularly important that you adjust the volume and tone control settings properly the first time you save or recall procedures, and then mark those settings so that they can be used again the next time you want to save or recall. If you are using your cassette recorder only for saving and recalling with your computer, you might want to tape the volume and tone adjustment wheels ir position so that they do not change accidentally between uses.

To save or recall information using a cassette recorder, first give the SAVE or RECALL command. Then choose whether to save or recall (1) procedures only, (2) shapes only, or (3) both shapes and procedures. Then choose (1) from the device menu (see Figure 5.2) to choose the cassette recorder as the device for saving or recalling.

Once you choose to use a cassette, the computer will give exact instructions for using the cassette, telling you which cassette buttons and computer keys to press, and when. The process is quite straightforward. You use the cassette recorder's *playback* mode to recall information, and its *record* mode to save information. The computer will tell you whether an error has occurred in saving or recalling. In case of error, you may have to repeat the entire process of saving or recalling. The User's Reference Guide that comes with your computer offers suggestions about what to check in case of errors in saving or recalling.

5.2.2. Using Diskette

In contrast to cassette tape, a single diskette may contain many different files, and the files are distinguished by the fact that they are named.[2]

When you type SAVE and indicate "(2) diskette" in response to Logo's query, Logo will next ask you for a *file name* under which the information should be stored. This is shown in Figure 5.3. You may use file names up to 8 characters long. If you use the same names as a file that is already on the disk, the information in the old file will be replaced by the new information being written.

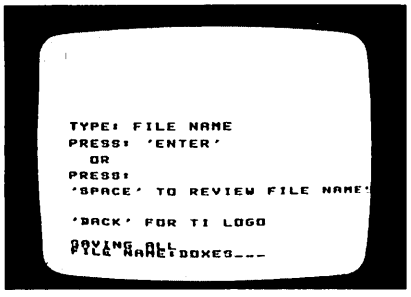


Figure 5.3: Screen display asking for file name.

If you press the space bar in response to Logo's request for a file name, then Logo will review the file names on the disk in alphabetical order, printing another name each time the space bar is pressed. To replace a file, press ENTER when the file name appears on the screen.[3]

[2]Before using a blank diskette for the first time to save files, the diskette must be initialized with the TI Disk Manager cartridge, which is packaged with the disk controller. Place the blank diskette in the drive and follow the instructions provided with the Disk Manager.

[3]When reviewing file names like this, you cannot type a new file name. To type a new file name, press BACK to return to Logo and begin again with SAVE.

Keep in mind that the name you give your file has *no relation* to the names of the procedures that will be saved in the file. Each time you save a file, *all* procedures in the workspace are included.[4]

When working on large projects, it is a good idea to save your workspace periodically in a file. Also, as you continue to make modifications, you should keep on disk the last two or three versions of your project. A technique for doing this is to include a version number as part of the file name. For example, if you are working on a project called CIRCLES, save the information the first time as CIRCLES1. After you have made modifications, save the updated version as CIRCLES2. You can use the utilities provided with the Disk Manager to get rid of unneeded versions. As you work on the project, it's a good idea to always keep around the previous two or three versions, just in case you mistakenly save a bad version on disk.

Recalling Files From Diskette

If you give the RECALL command and specify "(2) diskette," Logo will ask you for the name of the file to recall. You may either type in a file name, or press the space bar to ask Logo to review the names of the files currently on the diskette. Each time you press the space bar, a new file name appears. When you reach the file you want, press ENTER. The information will be read, and Logo will return to command level.

5.2.3. Saving and Recalling Using Other

The third option, (3) other on the device menu, allows you to use a second or third disk drive.[5] If you choose option (3), you will be prompted for device and file names (Figure 5.4):

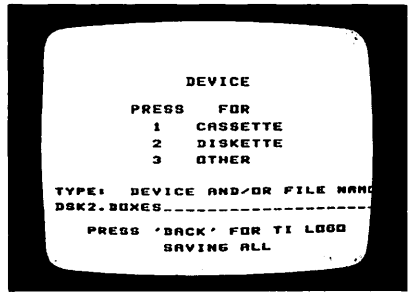


Figure 5.4: After choosing option (3), other on the device menu, you will be prompted for device and file name.

[4]This sometimes causes confusion with beginners. For example, if someone has procedures BOX and HOUSE, they write both files BOX and HOUSE, thinking that a separate file is needed for each procedure. The result is that they end up with two files, each containing both procedures. In general, you should name your files with a name that describes the *group* of procedures being saved.

[5]This option will allow access to other types of memory devices as they become available.

Your response must include the letters DSK (meaning "disk"), a number (2 or 3), a period (.), and a file name. It must be typed on one line with no spaces, as shown in the example.

Repeat the same process when you recall information saved using option (3). You can recall it from a different disk drive if you wish. For example, you can recall a file saved on disk 2, from disk 1, but you must recall it using option (3).

5.2.4. Other Uses of the File System

Although SAVE and RECALL are almost always used to save and restore complete workspaces, it is also convenient to be able to manipulate files in other ways. For example, suppose you want to merge the procedure definitions in two files to create a larger file. You can do this as follows:

1. Starting with an empty workspace, RECALL the first file.
2. RECALL the second file. Now your workspace contains the definitions in both files.
3. SAVE your workspace as the new, combined file.

As another example, suppose you want to delete a few procedure definitions from a file. One way to do this is to RECALL the file, ERASE the unwanted definitions, and SAVE the new workspace using the same file name.

5.2.5. Obtaining Hard Copy: the PRINTOUT Command

The PRINTOUT command allows you to print all the procedures in your workspace using a TI Thermal Printer or an RS232 printer. To use an RS232 printer you must have an RS232 card in your Peripheral Expansion Box or an RS232 Expansion unit attached to your computer. When you type PRINTOUT, you will first be prompted for the name of the device you are using.

If you are using a TI Thermal Printer, just type TP, and the printer will begin printing the contents of your workspace. If you have an RS232 printer, the device name must include the symbols "RS232.BA=" and a *baud rate*, all typed without any spaces. For example,

```
RS232.BA=9600
```

9600 is the baud rate in this example — the rate at which information can be sent to the printer. This will depend on the capabilities of your particular printer, so you will have to consult your printer manual for this information.

Next, you will be asked for a line length, which must be less than the longest line length your printer can handle. Again, this will vary from printer

to printer. After you specify the device name and line length, your printer will begin printing.

5.3. Aids for Debugging

One of the main features of Logo as a computer language for education is that students *design* and *write* programs as well as use them. *Debugging* a program is a crucial part of the programming process. This section describes features included in the Logo system to aid in debugging programs.

5.3.1. Pausing Execution with the AID Key

When Logo is running a procedure, pressing the AID key works somewhat like pressing the BACK key—it stops procedure execution. The difference is that AID stops the procedure "in the context where it is executing" and allows you to examine the values of local names.

As a simple example, consider the FLAG and RECTANGLE procedures that we introduced in Section 2.1.2. Suppose that RECTANGLE had a bug—an extra last line that calls RECTANGLE recursively, so that the procedure keeps running forever:

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

Now suppose you inadvertently use this procedure as part of FLAG:

```
TO FLAG :HEIGHT
FORWARD :HEIGHT
RECTANGLE (:HEIGHT / 2) :HEIGHT
BACK :HEIGHT
END
```

If you run FLAG 50 you will see the turtle draw part of the flag and then get "stuck" tracing the same rectangle over and over. If you now press AID, you will see a message like this:

```
PAUSE AT LEVEL 10 LINE 8 OF RECTANGLE
L10?
```

This message tells you that Logo was executing the RECTANGLE procedure, line 8, when you pressed AID. The meaning of *level* here is the same as that typed by error messages and described on page 23: you are 10 levels away from the typed-in command—your typed command called FLAG which called RECTANGLE, which called RECTANGLE again, which called RECTANGLE again, and so on.

The prompt L10? indicates that you are now typing commands within the context of RECTANGLE at level 10. You are free to type and execute *any* Logo command just as if you were at top level. The big difference is that now the *variable names* you use will refer to names in the *private library* of the procedure in which you paused. In the current example, the private libraries for FLAG and RECTANGLE are as shown in Figure 2.5 on page 20, so that we could examine RECTANGLE's private variables:

```
L10? PRINT :HEIGHT
25
L10? PRINT :LENGTH
50
```

This ability to examine local variables can be useful when you are trying to track down bugs.

Pressing BACK from within such a "pause break" causes Logo to return to top level and wait for a new command. Also, executing a command that causes an error will return Logo to top level.[6]

5.3.2. TRACEBACK

When you are within a pause, you can use the TRACEBACK command to find out "where you are." For instance, typing TRACEBACK in the example above yields:

```
L10? TRACEBACK
WE'RE NOW INSIDE RECTANGLE, FLAG
```

In general, TRACEBACK indicates the chain of procedures from where you are currently back to the top level. In this case, the pause happens inside RECTANGLE, which was called by FLAG, which was called at command level.

---

[6]Unless the DEBUG option has been set. See Section 5.3.3 below.

5.3.3. The DEBUG Option

Normally, when Logo encounters an error, it halts execution, types an error message, and returns to command level. Alternatively, you can direct Logo to enter a pause when an error is encountered, so that you can examine the values of local variables, as with AID. The DEBUG command acts as an "on-off" switch that controls this option. Turn on the option by typing DEBUG:

```
DEBUG
ON
```

Logo's response indicates that the debug option is now on.

With the option turned on, suppose the RECTANGLE had another bug—a misspelling in the second line:

```
TO RECTANGLE :HEIGHT :LENGTH
FORWARD :HEIGHT
RIGXT 90
FORWARD :LENGTH
RIGHT 90
FORWARD :HEIGHT
RIGHT 90
FORWARD :LENGTH
RIGHT 90
RECTANGLE :HEIGHT :LENGTH
END
```

Executing FLAG 20 now results in the following error:

```
TELL ME HOW TO RIGXT
  AT LEVEL 2 LINE 2 OF RECTANGLE
L2?
```

As with using AID, we can now type commands at level 2, for example, to print the values of local variables (although examining local variables isn't much help in dealing with this particular bug).

Typing DEBUG when the debug option is on, turns the option off:

```
DEBUG
OFF
```

CHAPTER 6

Numbers, Words, and Lists

In the previous chapters, we used turtle geometry to introduce the basic techniques for writing Logo procedures. We now move away from graphics to discuss Logo programs that work with "data." Like most computer languages, Logo provides operations for manipulating numbers and character strings, which in Logo are called *words*. One significant difference between Logo and other simple programming languages is that Logo also provides the ability to combine data into structures called *lists*. This chapter introduces these three kinds of data objects—numbers, words, and lists—together with simple programs that manipulate them. The most important concept in working with Logo data is the notion of a procedure that *outputs a value*. This is introduced in Section 6.2 below. We also discuss the use of Logo variables for naming data and give a more complete explanation of testing and conditionals than the one provided in Section 2.2.2. The material presented here provides enough background to complete many programming projects such as the ones described in Chapter 7.

6.1. Numbers and Arithmetic

We have already seen examples of using numbers in turtle programs. Logo provides the basic arithmetic operations of addition, subtraction, multiplication, and division, denoted by +, −, *, and /, respectively. In combined arithmetic operations, multiplications and divisions are performed before additions and subtractions, unless you use parentheses to make the grouping explicit:

```
PRINT 3 + 2 * 5
13

PRINT (3 + 2) * 5
25
```

TI Logo deals with integers only. The division operation / truncates its quotient to be an integer:

```
PRINT 5 / 2
2
```

Warning

TI Logo can only handle integers in the range ±32767 (that is, in the range between -2^{15} and 2^{15}). If you do a computation that exceeds this range (e.g., adding 32767 plus 1), the answer returned will be *incorrect* but there will be no error message:

```
PRINT 32767 + 1
-32767
```

6.2. Outputs

Using the arithmetic operations presented above, you can write procedures that manipulate numbers. For example,

```
TO PSQUARE :X
PRINT :X * :X
END
```

prints the square of its input, and

```
TO PAVERAGE :X :Y
PRINT (:X + :Y) / 2
END
```

prints the average of its two inputs:

```
PSQUARE 100
10000
PAVERAGE 1 3
2
```

These procedures may be instructive, but they are not very useful. PSQUARE, for example, just prints the square of its input. Having computed the square, there is nothing more you can do with it. Yet the whole power of the procedure concept is that you should be able to use procedures as *building blocks* in defining more complex procedures. You can make complex turtle programs by combining the designs drawn by simple procedures. But there is no way to combine PSQUARE and PAVERAGE to obtain, for instance, the square of the average of two numbers.

What is needed is some way for a procedure not only to compute some result, but also to make that result accessible to other procedures. In Logo, this is accomplished by the OUTPUT command. To see how it works, compare the PSQUARE procedure above with the following:

```
TO SQUARE :X
OUTPUT :X * :X
END
```

When SQUARE runs, it returns its result as an *output* that is to be used as an input to whatever command called SQUARE. For example, you can type

```
PRINT SQUARE 3
9
```

in which case the output of SQUARE is passed to PRINT to be printed. More significantly, you could type

```
PRINT (SQUARE 3) + (SQUARE 4)
25
```

Here SQUARE is called twice, and the results are combined by + before being passed to PRINT. You can do the same thing with computing averages by defining a procedure:

```
TO AVERAGE :X :Y
OUTPUT (:X + :Y) / 2
END
```

6.2.1. Combining Operations

The OUTPUT command is just what is needed to combine operations. For instance, you can find the square of the average of two numbers:

```
PRINT SQUARE (AVERAGE 4 6)
25
```

or the average of the squares:

```
PRINT AVERAGE (SQUARE 4) (SQUARE 6)
26
```

Alternatively, you can define a procedure to return this value to be used in further processing:

```
TO AVERAGE.OF.SQUARES :X :Y
OUTPUT AVERAGE (SQUARE :X) (SQUARE :Y)
END
```

As with any procedure, once you have defined a procedure that outputs some result, that procedure becomes part of Logo's working vocabulary and can be used just as if it were a primitive command. For instance, Logo has no primitive absolute value function. But if you define one:

```
TO ABS :X
IF :X < 0 THEN OUTPUT (- :X)
OUTPUT :X
END
```

then you can use this ABS operation in performing further computations.

When a procedure executes an OUTPUT instruction, it returns the indicated output to the procedure that called it, and no further commands within the procedure are executed. Thus, for example, only one of the two OUTPUT instructions in ABS will be executed each time ABS is called.

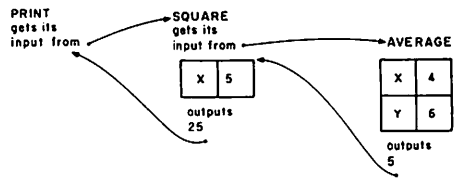


Figure 6.1: Procedure calls in executing PRINT SQUARE (AVERAGE 4 6).

To help you visualize outputs, Figure 6.1 shows a diagram, similar to the diagrams in Section 2.1.2, for the procedure calls involved in executing the command line

```
PRINT SQUARE (AVERAGE 4 6)
```

SQUARE and AVERAGE each have a private variable X, but since these are in different private libraries, there is no conflict.

As shown in the diagram, you can regard inputs and outputs as communication channels between procedures. If procedure A calls procedure B then A can use inputs to communicate values to B. B's output enables it to communicate values back to A.

A very common error in Logo programming is to attempt to make a procedure output without using the OUTPUT instruction. For example, you might attempt to define AVERAGE as

```
TO AVERAGE :X :Y
(:X + :Y) / 2
END
```

Calling this procedure, say

```
AVERAGE 4 6
```

would result in the error message

```
TELL ME WHAT TO DO WITH 5
  AT LEVEL 1 LINE 1 OF AVERAGE
```

when the procedure was executed. In general, you should say what Logo is supposed to do with generated values—print them, output them, or whatever.

6.2.2. Example: Remainders and Random Numbers

One useful operation you can create with OUTPUT is a REMAINDER procedure, which outputs the remainder of its two arguments:

```
TO REMAINDER :NUM :DIV
OUTPUT :NUM - (:NUM / :DIV) * :DIV
END
```

The procedure works by taking advantage of the fact that division in Logo truncates the quotient. Therefore, when you take :NUM / :DIV and multiply the result by :DIV, you obtain the largest multiple of :DIV that is less than :NUM. Subtracting this from :NUM yields the remainder.

You can use REMAINDER to implement another useful procedure called RAND, which takes a positive number *n* as input and outputs a random number between 0 and *n* - 1. RAND uses the Logo built-in operation RANDOM, which returns a single digit (0–9) selected at random:

```
PRINT RANDOM
5
PRINT RANDOM
2
```

To implement RAND, you can begin by writing a procedure RANDOM4, which outputs a four-digit random number. (We'll only worry about using RAND with inputs less than 10,000.)

```
TO RANDOM4
OUTPUT RANDOM + 10 * RANDOM + 100 * RANDOM
               + 1000 * RANDOM
END
```

Now, to obtain a random number less than some number *n*, you simply need to take the remainder by *n* of the number returned by RANDOM4:

```
TO RAND :N
OUTPUT REMAINDER RANDOM4 :N
END
```

6.3. Words

In Logo, strings of characters are called *words*. Logo provides operations for manipulating words: combining words into longer words and breaking words into parts. As with numbers, words may be passed among procedures as inputs and outputs.

To indicate a word in Logo, you type the character string prefixed by a quotation mark, as in:

```
PRINT "WHOOPIE
WHOOPIE
```

Notice that (unlike the rule in English) the quotation mark goes only at the beginning of the word. The word itself is taken to be all of the characters between the quotation mark and the following space or the end of the line. Beware that if you put a quotation mark at the end of a word, that quotation mark will be taken to be part of the word:

```
PRINT "A"
A"
```

Logo provides the following operations for extracting parts of words:

FIRST	Outputs the first character of its word input. Abbreviated F.
LAST	Outputs the last character.
BUTFIRST	Outputs a word containing all but the first character. Abbreviated BF.
BUTLAST	Outputs a word containing all but the last character. Abbreviated BL.

Here are some examples:

```
PRINT FIRST "ABCD
A
PRINT BUTFIRST "ABCD
BCD
PRINT LAST BUTLAST "ABCD
C
```

In the third example, the thing that is printed is the LAST of the BUTLAST of ABCD which is the LAST of ABC which is C.

For constructing larger words from smaller ones, Logo provides the WORD operation. This takes two words as inputs and combines them to form a single word:

```
PRINT WORD "NOW "HERE
NOWHERE
```

Sample Procedures That Use Words

The following recursive procedure is a word analogue of the COUNTDOWN procedure on page 24:

```
TO TRIANGLE :WORD
PRINT :WORD
IF :WORD = FIRST :WORD THEN STOP
TRIANGLE BUTFIRST :WORD
END

TRIANGLE "LOLLIPOP
LOLLIPOP
OLLIPOP
LLIPOP
LIPOP
IPOP
POP
OP
P
```

Whereas COUNTDOWN reduced a number to smaller numbers by successively subtracting 1, TRIANGLE reduces a word to smaller words by successively removing the first character. The process stops when the word has been reduced to a single character, that is, to a word that is equal to its own first character.

TRIANGLE illustrates the use of words as inputs to procedures. As an example of words as outputs, consider the simple procedure DOUBLE, which takes a word as input and outputs the word concatenated with itself:

```
TO DOUBLE :X
OUTPUT WORD :X :X
END

PRINT DOUBLE "BOOM
BOOMBOOM
```

Observe the importance of using OUTPUT: you can operate on a word using DOUBLE and use the result as an input to other operations:

```
PRINT DOUBLE DOUBLE "BOOM
BOOMBOOMBOOMBOOM

TRIANGLE DOUBLE "ABC
ABCABC
BCABC
CABC
ABC
BC
C
```

Warning: Words and Numbers

In TI Logo, you can form words whose characters are all digits. But these are *not* treated as numbers, even though they look like numbers. For example, the arithmetic operations will not accept something like "25 as an input. Conversely, the word-manipulating operations do not work on numbers. This distinction between words and numbers can be confusing, because error messages that result from inappropriate inputs do not distinguish between numbers and words:[1]

```
PRINT FIRST "25
2

PRINT FIRST 25
FIRST DOESN'T LIKE 25 AS INPUT

PRINT "25 + 5
+ DOESN'T LIKE 25 AS INPUT

PRINT 25 + 5
30
```

6.4. Lists

Many languages force the programmer to work with text in terms of character strings. A long text must be viewed as a long character string that is manipulated on a character-by-character basis. One of the advantages of Logo is that it allows you to manipulate sequences of words on a *word-by-word* basis. In Logo, a sequence of words is called a *list*. A list may

[1] To add to the confusion, the built-in operation WORD?, which tests whether its input is a word, returns TRUE for numbers.

be indicated by separating the words in the list by spaces and enclosing them in square brackets:[2]

```
PRINT [THIS IS A LIST]
THIS IS A LIST
```

Notice that the words in the list are not quoted and that the surrounding brackets are not printed. The spaces between the words serve only to delimit the words. Extra spaces are ignored:

```
PRINT [EXTRA        SPACES]
EXTRA SPACES
```

The Logo operations FIRST, LAST, BUTFIRST, and BUTLAST that we introduced for use with words also operate on lists. When used with lists, these operations pick out the first or last word of the list, rather than the first or last character, as they do with words.

```
PRINT FIRST [THIS IS A LIST]
THIS

PRINT FIRST BUTFIRST [THIS IS A LIST]
IS

PRINT BUTLAST [THIS IS STILL A LIST]
THIS IS STILL A

PRINT BUTFIRST [THIS]
(blank line)
```

Note that in the last example, taking all but the first word of a list that has only one word produces a list containing *no* words, called the *empty list*. It can be typed into Logo as [ ].[3]

In Logo, a list is never considered to be equal to a word. For example, a word is not considered equal to a list that contains that single word, even though Logo prints these in the same way:

```
PRINT "BUBBLE
BUBBLE
```

---

[2]Logo lists are used not only for making sequences of words, but also for creating data structures in general. See Section 11.1. Remember to delimit lists with square brackets [ ] and not parentheses ( ).

[3]In the current implementation of TI Logo, taking the BUTFIRST or BUTLAST of a single character word (i.e., removing all characters from a word) results in the empty list.

```
PRINT [BUBBLE]
BUBBLE

PRINT "BUBBLE = [BUBBLE]
FALSE
```

SENTENCE is the operation for putting lists together, analogous to WORD for words. SENTENCE (abbreviated SE) takes words or lists as inputs and assembles these into one list:

```
PRINT SENTENCE [THIS IS] [HOW SENTENCE WORKS]
THIS IS HOW SENTENCE WORKS

PRINT SENTENCE "THIS [IS TOO]
THIS IS TOO

PRINT SENTENCE "THIS "ALSO
THIS ALSO
```

Sample Procedures That Use Lists

Here are the list procedures analogous to the word procedures TRIANGLE and DOUBLE of Section 6.3. Notice that we have changed the stop rule for TRIANGLE to test for an empty input.

```
TO TRIANGLE.LIST :X
IF :X = [] STOP
PRINT :X
TRIANGLE.LIST BUTFIRST :X
END

TO DOUBLE.LIST :X
OUTPUT SENTENCE :X :X
END

TRIANGLE.LIST [THIS IS A LIST]
THIS IS A LIST
IS A LIST
A LIST
LIST

PRINT DOUBLE.LIST [HUP 2 3 4]
HUP 2 3 4 HUP 2 3 4
```

```
TRIANGLE.LIST DOUBLE.LIST [DING DONG]
DING DONG DING DONG
DONG DING DONG
DING DONG
DONG
```

The main thing to observe in these examples is that lists, like numbers and words, can be passed between procedures as inputs and outputs.

The following list procedures make use of the Logo command READLINE (abbreviated RL), which makes it easy to write interactive programs using lists. READLINE waits for you to type in a line (terminated by ENTER) and outputs the typed-in line as a list.

```
TO BOAST
PRINT [WHO'S THE GREATEST?]
IF READLINE = [ME] THEN PRINT [OF COURSE!] STOP
PRINT [NO, TRY AGAIN]
BOAST
END
```

```
BOAST
WHO'S THE GREATEST?
>MIGHTY MOUSE
NO, TRY AGAIN
WHO'S THE GREATEST?
>ME
OF COURSE!
```

Note the prompt > printed in the example above. Logo prints this prompt to remind you that it is waiting for you to respond to a READLINE. Bear in mind that READLINE always outputs a list. If you type a single word, the output of READLINE will be a list containing that one word.

Here's another example:

```
TO CHAT
PRINT [WHAT'S YOUR NAME?]
PRINT SENTENCE [HELLO] READLINE
PRINT [TYPE SOMETHING YOU LIKE]
PRINT SENTENCE [I'M GLAD YOU LIKE] READLINE
END
```

Notice how the second line of the procedure is constructed: the list being PRINTed is a SENTENCE of two things—the list [HELLO] and the list output by READLINE.

```
CHAT
WHAT'S YOUR NAME?
>LUCY
HELLO LUCY
TYPE SOMETHING YOU LIKE
>PICKLE JELLO
I'M GLAD YOU LIKE PICKLE JELLO
```

6.5. Naming

We have seen different kinds of *naming* in Logo programs: the use of names to refer to inputs to procedures and the idea of naming procedures themselves. In Chapter 4, we also saw that the Logo command MAKE can be used to give names to things.

Consider the following example:

```
MAKE "NUMBER 5
PRINT :NUMBER
5
```

In the first line you tell Logo that you are going to call the number 5 by the name NUMBER. The first input to MAKE is the *name* and the second input is the *thing* you are naming. The effect of the command is to establish a relationship between the word NUMBER and the number 5. We express this by saying that "5 is the thing associated with NUMBER." In the line

```
PRINT :NUMBER
```

you can see how : recovers the thing associated with the name, just as it recovers the value associated with an input to a procedure. Here are more examples:

```
MAKE "COLR "YELLOW
PRINT "COLR
COLR

PRINT :COLR
YELLOW

MAKE "SLOGAN [I LOVE BANANAS]
PRINT :SLOGAN
I LOVE BANANAS

PRINT SENTENCE (BUTLAST :SLOGAN) :COLR
I LOVE YELLOW
```

In these examples, and in most programs, the name is specified as a literal, quoted word. This is not the only possibility:

```
MAKE (WORD "PART "1) [DO MI SOL]
PRINT :PART1
DO MI SOL
```

Here is a tricky example:

```
MAKE "FLOWER "ROSE
PRINT :FLOWER
ROSE

MAKE :FLOWER [IS A ROSE IS A ROSE]
PRINT :FLOWER
ROSE

PRINT :ROSE
IS A ROSE IS A ROSE
```

In the third command line, the name associated with [IS A ROSE IS A ROSE] is not the literal word FLOWER, but rather the *thing* associated with FLOWER, that is, the *word* ROSE. Therefore,

MAKE :FLOWER <something>

has the same effect as

MAKE "ROSE <something>

The Logo function THING returns the thing associated with its input. The use of : is actually an abbreviation for THING in the case where the input to THING is a quoted literal word. But THING can be used in more general circumstances.

```
MAKE "NAME1 [JOHN Q. CITIZEN]
PRINT :NAME1
JOHN Q. CITIZEN

PRINT THING "NAME1
JOHN Q. CITIZEN

PRINT THING (WORD "NA "ME1 )
JOHN Q. CITIZEN

PRINT THING (FIRST [NAME1 PLACE1])
JOHN Q. CITIZEN
```

There is also the Logo predicate THING?, which takes a word as input and outputs TRUE if the word has something associated with it.

```
PRINT THING? "NAME1
TRUE

PRINT THING? "NAME2
FALSE
```

6.5.1. Local and Global Names

In Section 2.1.2 we saw that the names of inputs are *private* to the procedures using them. Different procedures reference names in different private libraries, and two procedures may use the same names for different purposes without any conflict. The same holds true if the procedure uses the MAKE command to change the value associated with some input name. This is illustrated in the following example.

```
TO DEMO :X
PRINT :X
CHANGE :X
PRINT :X
END

TO CHANGE :X
MAKE "X :X + 1
PRINT :X
END

DEMO 1
1       (printed in DEMO)
2       (printed in CHANGE)
1       (printed in DEMO)
```

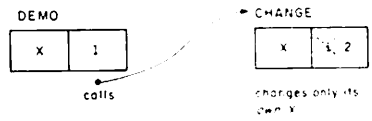


Figure 6.2: Private libraries for DEMO and CHANGE.

The important point to notice is that when the value of X is printed in DEMO the second time, it is still 1, even though CHANGE "changed" X to 2. The reason is that DEMO and CHANGE each have their own meaning for X in different private libraries, as shown in Figure 6.2. When CHANGE uses the MAKE statement it changes *its* X, but not DEMO's.

When you use a MAKE statement at command level, you are also associating a value with a name in some library. But this is not a library associated with any procedure. Rather it is a library associated with the command level. Definitions in this library are sometimes called *global variables*. Just as the private libraries of two procedures are distinct, names in procedure libraries will not conflict with names in the global library. Compare the following example to the one above.

```
MAKE "X 1
CHANGE :X
2
PRINT :X
1
```

6.5.2. Free Variables

One of the reasons that procedures are so important is that they provide a way to design complex programs in small pieces. But whenever you design something by breaking it into pieces, you eventually have to deal with the issue of how these pieces can interact. The importance of the private library mechanism is that it guarantees that the names used by different procedures will refer to different things and hence that the *only* way procedures can interact is through inputs and outputs. This guarantee provides a good handle on controlling the complexity of the entire program.

Sometimes, however, it is convenient for procedures to interact other than through inputs and outputs. For example, if the computation performed by a procedure depends on a large number of parameters, it may be cumbersome to specify them all as inputs each time the procedure is called. Again, using only inputs and outputs to pass information may require passing "superfluous" inputs through many levels of nested procedures until they reach the procedure that actually needs them. For these reasons it is useful to be able to have the computation performed by a procedure depend not only on the information provided *explicitly* by the inputs, but also on information that is *implicit* in the *context* in which the procedure is used.

Consider the following procedure:

```
TO NEW.PRICE :P
OUTPUT :P + :OVERHEAD
END
```

Suppose you would like to be able to use this procedure in such a way that the price computed depends on some OVERHEAD that is obtained from the context in which the procedure is used. For example:

```
MAKE "OVERHEAD 50
PRINT NEW.PRICE 100
150
MAKE "OVERHEAD 25
PRINT NEW.PRICE 100
125
```

You might also want to have the context determined by a procedure, as in

```
TO TRY :OVERHEAD
PRINT NEW.PRICE 100
PRINT NEW.PRICE 200
END

TRY 100
200
300

TRY 50
150
250
```

The name OVERHEAD in the NEW.PRICE procedure is what is technically known as a *free variable.* A free variable is a name that is used in a procedure, but not as a name for an input. As the NEW.PRICE example shows, Logo procedures can have free variables. The presence of free variables leads to the following rule for finding the value associated with a name in a Logo procedure:

- If the name is one of the names of the inputs to the procedure, the value can be found in the procedure's private library.
- Otherwise, see if the name is in the library of the procedure that *called* the current procedure.
- Otherwise, see if the name is one of the names in the procedure that called *that* procedure, and so on, all the way through to the global library.

Free variables provide a powerful mechanism for passing information between procedures. But their indiscriminate use leads to obscure programs and may result in intractable program bugs. This is especially true if you use MAKE to change the value of a free variable, since the actual variable affected may appear arbitrarily far back in the nest of procedure calls.

6.6. Conditional Expressions and Predicates

We saw in Section 2.2.2 the use of conditional expressions IF . . . THEN . . . in Logo programs. This section provides more information about conditional expressions.

IF and THEN can be augmented by the Logo primitive ELSE. For example, the following procedure tells whether a number is positive or negative:

```
TO SIGN :N
IF :N < 0 THEN OUTPUT "NEGATIVE ELSE OUTPUT "POSITIVE
END

PRINT SIGN 57
POSITIVE
PRINT SIGN (10 - 20)
NEGATIVE
```

IF. . . THEN . . . ELSE expressions are often confusing for beginning programmers, due to the need to work with a single statement that specifies both a test and the actions to be taken depending on the outcome of the test. Logo therefore includes another form of conditional that separates the testing from the actions. This form is TEST. . . IFT. . . IFF. The TEST used in a procedure checks some condition. Subsequent procedure lines that begin with IFT and IFF are executed or not, depending on the result of the TEST. Here's another way to write the SIGN procedure using TEST:

```
TO SIGN :N
TEST :N < 0
IFT OUTPUT "NEGATIVE
IFF OUTPUT "POSITIVE
END
```

A procedure can include more than one TEST, and any IFT or IFF statements always refer to the most recent TEST. Also, the result of a TEST is kept private within a procedure, so the use of IFT and IFF within a procedure is not affected by any TESTs performed in a subprocedure.

Predicates; TRUE and FALSE

The conditions checked by IF and TEST are known as *predicates*. We already introduced the Logo predicates >, <, and = for working with numbers. Section 12.6 gives a complete list of the predicates built into Logo. It is also easy to define new predicates, because a predicate in Logo is nothing more than a procedure that outputs either the word TRUE or the

word FALSE. For instance, you can transform the SIGN procedure given above into a predicate that outputs FALSE if the input is less than 0 and TRUE otherwise:[4]

```
TO POSITIVE? :X
IF :X < 0 OUTPUT "FALSE ELSE OUTPUT "TRUE
END
```

As another example, the following predicate takes a word as input and tests whether it begins with a vowel:

```
TO BEGINS.WITH.VOWEL? :X
IF (FIRST :X) = "A OUTPUT "TRUE
IF (FIRST :X) = "E OUTPUT "TRUE
IF (FIRST :X) = "I  OUTPUT "TRUE
IF (FIRST :X) = "O OUTPUT "TRUE
IF (FIRST :X) = "U OUTPUT "TRUE
OUTPUT "FALSE
END
```

Once a predicate has been defined, it can be used with IF or TEST just as if it were one of the predicates built into Logo. Here is a procedure that adds "a" or "an" to a word, as appropriate:[5]

```
TO ADD.A.OR.AN :X

TEST BEGINS.WITH.VOWEL? :X
IFT OUTPUT SENTENCE "AN :X
IFF OUTPUT SENTENCE "A :X
END

PRINT ADD.A.OR.AN "COMPUTER
A COMPUTER
PRINT ADD.A.OR.AN "ELEPHANT
AN ELEPHANT
```

[4]It is a good programming habit to name predicates with names that end with a question mark. As far as the Logo system is concerned, though, the question mark has no special significance. It is treated as an ordinary character.

[5]Later on, when we see how to take advantage of Logo lists as data structures, we will learn more flexible ways of computing functions like BEGINS.WITH.VOWEL. Compare the alternative version of the ADD.A.OR.AN procedure that is used in the ANIMAL program of Section 11.3.2.

You can regard IF and TEST as operations that take an input that must be either TRUE or FALSE. In fact, the primitive predicates built into Logo are themselves operations that output TRUE or FALSE:[6]

```
PRINT 3 > 5
FALSE
PRINT "XYZ = "XYZ
TRUE
```

For combining predicates, Logo includes the operation BOTH, which takes two inputs that must be either TRUE or FALSE and outputs TRUE if both inputs are TRUE and FALSE otherwise. There is also EITHER, which outputs TRUE if at least one of its inputs is TRUE, and NOT, which outputs TRUE if its input is FALSE, and FALSE if its input is TRUE.

```
PRINT BOTH (1 < 2) (2 > 3)
FALSE
PRINT EITHER (1 < 2) (2 > 3)
TRUE
PRINT NOT (2 + 2 = 4)
FALSE
```

For example, here are three equivalent ways to write a predicate BETWEEN?, which tests whether a specified number is in a given range:

```
TO BETWEEN? :X :LOW :HIGH
IF :X < :LOW OUTPUT "FALSE
IF :X > :HIGH OUTPUT "FALSE
OUTPUT "TRUE
END

TO BETWEEN? :X :LOW :HIGH
IF EITHER (:X < :LOW) (:X > :HIGH) OUTPUT "FALSE
OUTPUT "TRUE
END
```

---

[6]TI Logo II includes special TRUE and FALSE which output the words TRUE and FALSE, respectively. Thus, in TI Logo II you can use TRUE and FALSE with or without quotes, as you prefer; for example,

```
IF :X = 5 OUTPUT "TRUE
```

and

```
IF :X = 5 OUTPUT TRUE
```

are both valid. In the first release of TI Logo, only the first form will work.

```
TO BETWEEN? :X :LOW :HIGH
IF BOTH (NOT (:X < :LOW)) (NOT (:X > :HIGH)) OUTPUT "TRUE
OUTPUT "FALSE
END
```

BOTH and EITHER are themselves predicates that output TRUE or FALSE. This means that the second two versions of BETWEEN? can also be written in another way, in which the TRUE or FALSE output by BOTH and EITHER is output directly to the procedure that calls BETWEEN?:

```
TO BETWEEN? :X :LOW :HIGH
OUTPUT BOTH (NOT (:X < :LOW)) (NOT (:X > :HIGH))
END

TO BETWEEN? :X :LOW :HIGH
OUTPUT NOT EITHER (:X < :LOW) (:X > :HIGH)
END
```

6.7. Details on Logo Syntax

This section collects some information about how Logo interprets the command lines that you type to it. This includes such information as where to include spaces and parentheses in command lines and how Logo groups sequences of commands.

6.7.1. How Logo Separates Lines into Words

Any Logo line is interpreted as a sequence of words. In general, you must separate words by spaces. For example, if you mean to type

```
FORWARD 100
```

and instead type

```
FORWARD100
```

Logo will respond with the error message

TELL ME HOW TO FORWARD100

because it will interpret FORWARD100 as a single word and look for a procedure with that name.[7] As a general rule, it is a good idea to type each line with spaces between the different elements. For example:

[7]Of course, you may have actually defined a procedure whose name was FORWARD100, in which case Logo would run that procedure.

```
PRINT ( 3 + 4 ) * 5
35
```

Logo does, however, understand that parentheses and arithmetic operators are normally meant to break words, so

```
PRINT (3+4)*5
35
```

works, too. In fact, if you define a procedure that includes a line such as the previous one, and later print out the procedure, you will find that Logo has inserted spaces into the line.

6.7.2. Using Parentheses

We have already seen some complex Logo expressions; for example, the following line is from the AVERAGE.OF.SQUARES procedure in Section 6.2.1.:

```
OUTPUT AVERAGE (SQUARE :X) (SQUARE :Y)
```

In this line, the OUTPUT command takes one input, which is the result of AVERAGE. AVERAGE in turn takes two inputs:

```
(SQUARE :X)
```

and

```
(SQUARE :Y)
```

Notice that parentheses perform grouping by enclosing the operation *together with its inputs*. That is, you should write

```
(SQUARE :X)
```

and not

```
SQUARE (:X)
```

to indicate that :X is the input to SQUARE.[8]

In fact, this expression would work perfectly well if you wrote it without any parentheses at all,

```
OUTPUT AVERAGE SQUARE :X SQUARE :Y
```

[8]This rule can be confusing, since the latter expression is more like SQUARE (X), which is what is used in mathematics or in languages like BASIC or Pascal. Keep in mind that parentheses in Logo are used to indicate *grouping*, not as special symbols for delimiting the list of inputs to functions.

because when Logo interprets the line, it breaks things up according to the following method. The first word it sees is OUTPUT, and this requires one input. So Logo scans the line trying to find that input. The next thing it runs into, though, is AVERAGE, which requires two inputs of its own. So Logo now scans to find two inputs for AVERAGE and runs into SQUARE, which requires one input which Logo finds as :X. This completes the input to SQUARE and also completes the first input to AVERAGE. Logo now looks for the second input to AVERAGE, and the next thing it sees is another SQUARE, which requires one input. Logo finds this as :Y. Now SQUARE has its input. This completes both inputs to AVERAGE, which completes the entire input to OUTPUT.

Generalizing this method, you can see that as long as Logo knows how many inputs each procedure needs, and as long as each procedure name is a prefix operator (i.e., it is written to the left of its inputs), then you don't need parentheses at all in writing Logo commands. On the other hand, parentheses help considerably in enabling the human eye to see the pattern. So unless you are very practiced, you should not write a complex expression without parentheses for fear of not being able to read it the next day.

The above rule for parsing expressions is modified for infix operators (i.e., operators that are written between their inputs, rather than to the left of them). In a line such as

```
AVERAGE 3 + 2  7
```

the 3 is combined with the 2 by + before any unit is assigned as an input to AVERAGE, so the line gets broken up as:

```
AVERAGE (3 + 2) 7
```

which gives 6. The general rule is that the infix arithmetic operators +, −, *, and / have higher priority than prefix operators.

The infix predicates >, <, and = have lower priority than prefix operators. So

```
AVERAGE 3 5 > AVERAGE 2 4
```

must be combined as

```
(AVERAGE 3 5) > (AVERAGE 2 4)
```

or you will get an error message.

Logo's rules for parentheses are designed to enable you to write simple expressions without worrying about parentheses. For complex expressions, it is better to use parentheses to avoid confusion.

SENTENCE with a Variable Number of Inputs

Although we haven't mentioned it yet, the SENTENCE operation can take a variable number of inputs, as in the following example,

```
PRINT (SENTENCE [THE BIG] [BAD] [WOLF])
THE BIG BAD WOLF
```

which uses one SENTENCE operation to combine three things into a list. The fact that SENTENCE is combining three things rather than its usual two is indicated by the parentheses grouping SENTENCE together with its inputs. In this way, SENTENCE can take any number of inputs.

Examples

Here are a few examples illustrating the rules discussed above, including some common errors and their explanations:

```
PRINT SENTENCE "A "B "C
A B
TELL ME WHAT TO DO WITH C
```

The default number of inputs to SENTENCE is 2, so SENTENCE combines A and B, and the result is printed by PRINT. Now Logo is faced with the rest of the line, and it runs across the symbol "C. Since there are no outstanding operations that need inputs, Logo complains that there is nothing to do with the "C.

```
PRINT (SENTENCE "A "B "C )
A B C
```

Here parentheses are correctly used to group the three inputs to SENTENCE.

```
PRINT (SENTENCE "A "B "C)
TELL ME MORE
```

The problem here is that there is no space separating the "C from the closing parentheses. Logo therefore interprets "C) as a two-character word which is the third input to SENTENCE and goes on to search for more inputs. Remember that when you indicate a word with a quotation mark, you must use a space to separate it from the rest of the line.

```
PRINT SENTENCE ("A "B "C )
TELL ME WHAT TO DO WITH "B
```

The problem here is that when you use parentheses to indicate grouping, you should group an operation together with its inputs. The parentheses are being used in this example as they would be used in BASIC, to surround the inputs

alone. But Logo always tries to interpret a parenthesized expression as a complete unit, which does not make sense in this case.

6.7.3. The Minus Sign

If the minus sign is preceded by a space and followed by a number, then TI Logo tries to interpret it as a negation sign. Otherwise, minus is interpreted as subtraction, in the context where that makes sense. Here are some examples:

```
PRINT 1-2
-1      (infix subtraction)
PRINT 1 - 2
-1      (infix subtraction)
PRINT 1 -2
1
TELL ME WHAT TO DO WITH -2
```
(Logo interpreted the - as negation, and got stuck.)
```
PRINT -2
-2      (negation)
PRINT - 2
-2      (negation)
```

In the last example, even though there is a space after the -, no value was pending that could be regarded as an input to - on the left, so Logo interpreted the - as negation.

When used in lists, the minus sign written before a number is regarded as signaling a negative number. Otherwise - is regarded as a separate word in the list:[9]

```
PRINT FIRST [-2  3]
-2
PRINT FIRST [- 2  3]
-
PRINT FIRST [2-3]
2
PRINT FIRST [-X  3]
-
```

In the third example, the list has three words: 2, -, and 3. In the fourth example, the list also has three words: -, X, and 3.

[9] In the first release of TI Logo, the minus sign is always interpreted as a separate word in a list. This makes it impossible to directly type in lists containing negative numbers. Such lists must be constructed using SENTENCE.

CHAPTER 7

More Logo Projects

This chapter presents four open-ended projects, suitable for beginning students. The first project is a simple arithmetic quiz program similar to the drill and practice computer systems used in some schools. The next project shows how to use lists to write programs that generate "random" sentences. We then reprint a paper by Papert and Solomon [11] that describes a simple game-playing program and discusses ideas about how to involve students in planning and carrying out complex projects. Finally, we use sprites and tiles to design a movie more elaborate than the one discussed in Section 4.4.

7.1. Arithmetic Quiz Program

Here's a simple arithmetic drill and practice program:

```
QUIZ
HOW MUCH IS 37 + 64
>101
GOOD
HOW MUCH IS 29 + 46
>87
THE ANSWER IS 75
HOW MUCH IS 21 + 11
>32
GOOD
```

and so on.

Designing a quiz program that works like this is a good programming project for elementary school students.[1] Here is one of many possible versions. It uses the RAND procedure discussed on page 80.

[1] And designing such a program is probably a better educational experience than *using* such a program, which is unfortunately much more typical of how computers are currently used in schools.

```
TO QUIZ
MAKE "NUM1 RAND 100
MAKE "NUM2 RAND 100
MAKE "ANSWER :NUM1 + :NUM2
PRINT (SENTENCE [HOW MUCH IS] :NUM1 [+] :NUM2)
MAKE "REPLY READNUMBER
TEST :REPLY = :ANSWER
IFT PRINT [GOOD]
IFF PRINT SENTENCE [THE ANSWER IS] :ANSWER
QUIZ
END

TO READNUMBER
OUTPUT FIRST READLINE
END
```

You use READNUMBER rather than READLINE directly because READLINE outputs a list. If the user types in a single number, READLINE outputs a list containing that number as its only item.[2] To obtain the number itself, you extract the first item from the list returned by READLINE.

You can also modify READNUMBER to check that the response is actually a number:

```
TO READNUMBER
MAKE "IN FIRST READLINE
TEST NUMBER? :IN
IFT OUTPUT :IN
IFF PRINT [PLEASE ANSWER WITH A NUMBER]
IFF OUTPUT READNUMBER
END
```

The behavior of QUIZ is now:

```
QUIZ
HOW MUCH IS 6 + 14
>B0
PLEASE ANSWER WITH A NUMBER
>20
GOOD
etc.
```

[2]Thus, if you set ANSWER to be the *list* returned by READLINE, ANSWER would never be equal to the sum of NUM1 and NUM2, which is a *number*. For example, if the user types 7 followed by ENTER, the value returned by READLINE will be the *list [7]*, not the *number* 7.

Observe that the recursive call in the final line of the procedure makes the procedure keep asking until the user responds with a number.

QUIZ is a good programming project because it has a simple core, yet there are so many extensions and variations. Some of these are as follows:

- Allowing the user to keep trying a question until getting the correct answer
- Keeping score
- Progressing to harder and harder problems when the score is good
- Giving advice

7.2. Random-Sentence Generators

You can have lots of fun with programs that print random sentences. In designing such programs, it is very useful to have as a building block a procedure PICKRANDOM that takes a list as input and outputs an item selected at random from a list; for example:

```
PRINT PICKRANDOM [EENEY MEENEY MINEY MO]
MEENEY
PRINT PICKRANDOM [EENEY MEENEY MINEY MO]
MO
```

PICKRANDOM is not built into Logo as a primitive, but it can be implemented by using lists and recursion, an aspect of Logo programming that we have not yet discussed. PICKRANDOM is implemented in terms of a procedure PICK, which outputs the *n*th item in a given list:[3]

```
TO PICKRANDOM :X
OUTPUT PICK (1 + RAND (LENGTH :X)) :X
END
```

PICK is defined as follows:

```
TO PICK :N :X
IF :N = 1 OUTPUT FIRST :X
OUTPUT PICK (:N - 1) (BUTFIRST :X)
END
```

---

[3] In the first release of TI Logo, LENGTH must also be implemented as a procedure:

```
TO LENGTH :X
IF :X = [ ] OUTPUT 0
OUTPUT 1 + LENGTH BUTFIRST :X
END
```

We will study programs such as these in Chapter 10, and we will explain how PICK and PICKRANDOM work in Section 10.2.1. In the meantime you can regard PICKRANDOM (and give it to beginning students) as a black box.

Once you have PICKRANDOM, it is easy to generate simple random sentences of the form {noun} {verb} by picking words at random from lists of nouns and verbs:

```
TO CHATTER
MAKE "NOUNS [DOGS CATS CHILDREN TIGERS]
MAKE "VERBS [RUN BITE TALK LAUGH]
BABBLE
END

TO BABBLE
PRINT SENTENCE (PICKRANDOM :NOUNS)
                (PICKRANDOM :VERBS)
BABBLE
END
```

```
CHATTER
CATS LAUGH
TIGERS TALK
CHILDREN BITE
TIGERS BITE
DOGS TALK
.
.
.
```

You can make the sentence generator more interesting by occasionally telling the computer to ask for a new noun or verb to be typed in and added to the corresponding list. For nouns this can be done with

```
TO LEARN.NOUN
PRINT [TEACH ME A NEW NOUN]
MAKE "NOUNS SENTENCE :NOUNS READLINE
END
```

Observe that this uses the SENTENCE operation to combine the typed-in word with the list of current nouns. There is a similar LEARN.VERB procedure for verbs.

Now you can modify BABBLE to ask for a new noun or verb every so often (1 chance in 10):

```
TO BABBLE
IF (RAND 10) = 0 LEARN.NOUN
IF (RAND 10) = 0 LEARN.VERB
PRINT SENTENCE (PICKRANDOM :NOUNS)
                (PICKRANDOM :VERBS)
BABBLE
END
```

The behavior of the program is now

```
CHATTER
CHILDREN TALK
TIGERS RUN
TEACH ME A NEW VERB
>WALK
DOGS RUN
CATS BITE
TEACH ME A NEW NOUN
>BANANAS
DOGS WALK
BANANAS BITE
.
.
.
```

There are many extensions to this project, including making more complex sentences by adding other parts of speech such as adjectives and adverbs, matching singular verbs with singular nouns and plural with plural, and generating random "poetry." Papert [15] describes the experience of one 13-year-old while engaged in such a project:

> One day Jenny came in very excited. She had made a discovery. "Now I know why we have nouns and verbs," she said. For many years in school Jenny had been drilled in grammatical categories. She had never understood the differences between nouns and verbs and adverbs. But now it was apparent that her difficulty with grammar was not due to an inability to work with logical categories. It was something else. She had not been able to make any sense of what grammar was about in the sense of what it might be *for*. . . But now, as she tried to get the computer to generate poetry, something remarkable happened. She found herself classifying words into categories, not because she had been told she had to but because she needed to. In order to "teach" her computer to make strings of words that would look like English, she had to "teach" it to choose words of an appropriate class. What she learned about grammar from this experience with a machine was anything but mechanical or routine. Her learning was deep and meaningful. Jenny did more than learn

definitions for particular grammatical classes. She understood the general idea that words (like things) can be placed in different groups or sets, and that doing so could work for her. She not only "understood" grammar, she changed her relationship to it.

7.3. Nim: A Game-Playing Program

This section is a slightly modified version of a paper written by Seymour Papert and Cynthia Solomon, which was originally published as an MIT Artificial Intelligence Laboratory Memo [14]. It illustrates some ideas about how to initiate beginning students into the art of planning and writing a program complex enough to be considered a project rather than an exercise on using the language or simple programming ideas.

The project is to write a program to play a simple game ("one-pile Nim" or "21") as invincibly as possible. The project was developed by Papert and Solomon for a class of seventh-grade students taught during 1968–69 at the Muzzey Junior High School in Lexington, Mass. This was the longest programming project these students had encountered, and the intention was to give them a model of how to go about working under these conditions. To achieve this, the teachers worked very hard to develop a clear organization of sub-goals, which they explained to the class at the beginning of the three-week period devoted to this particular program. You would not expect beginners to find as clear a sub-goal structure as this one; but once they have seen a good example, they are more likely to find clear sub-goals in the future for other problems. Thus the primary teaching purpose was to develop the idea of splitting a task into sub-goals. The intent was to provide the students with good models of various ways in which this can be done and to have them experience the heuristic power of this kind of planning (as opposed to jumping straight into writing programs).

A sub-goal structure can be imposed on a problem in several ways. One way is by "chopping," that is, by recognizing that the final program has distinct functions that can be performed by separate subprocedures. But this is not the only way. Many heuristic programs can be simplified rather than chopped. We illustrate this by first writing a procedure to play the entire game of Nim, but in a "dumb way." Once we have done so, we can study its performance, decide why it plays badly and strengthen its play. Thus the successive partial solutions to the problem appear as making a procedure progressively "smarter."

Describing the evolution of the program in this way has the additional benefit of allowing one to make an analogy valuable in two senses: by using themselves as models, students acquire a fertile source of ideas about programming; on the other hand, the experience of debugging programs can have a therapeutic effect in leading them to see their own mistakes as emotionally neutral *bugs* rather than as emotionally charged *errors*.

7.3.1. The Sub-Goal Plan

The key idea for subdivision of the problem is to write a series of programs, each of which is "smarter" than the previous one. The first program knows nothing about the strategy of play. It does not generate moves, but asks each of two human players in turn what move to make. For example, it may act as a scorekeeper, just keeping track of the number of sticks without bothering about whether the move is legal. From scorekeeper the machine can advance to referee. This means that it checks the move for legality and eventually declares the game over and announces the winner. After we have a working mechanical referee, we start making a mechanical player. The first version of a player chooses legal, but not necessarily good moves. Indeed, it generates a move randomly, uses its ability as a referee to decide if it is legal, and then either accepts it or generates another random move.

When this works, the child may make the program smarter and smarter by adding features or by writing a completely new version until finally—if all goes well—a player with an infallible strategy is evolved.

A natural form for programs of intermediate smartness is the following: the program has a list of simple situations in which it knows how to play; in other situations it plays randomly. In other words, it plays by the form of strategy used by most children in most strategic games.

In working with a class, a good moment should be seized to prod the students into noting and discussing the analogy between this very simple heuristic program and themselves—particularly, how the program gets to be smarter through more or better knowledge. Seeing the program as a cognitive model is a valuable and exciting experience for the students. They can easily be drawn into discussion about how meaningful such models are. To keep the discussion alive, the teacher should be equipped with arguments and examples to counteract extremist, and so sterile, positions. For example, if the students feel that the program is too simple to be a model of human thinking, the teacher might discuss whether a toy airplane is a useful model of a jet-airplane. Does it work by the same principles? Can you learn about airliners by studying toy models? On the other hand, if a class swings over to the position that there really is no difference, the teacher can ask questions about whether the program could learn by itself without a programmer. If this is too enthusiastically accepted it is well for the teacher to ask: "How much do you learn without being told?" Ideally, the teacher should merely guide the discussion without having to say any of this. But awareness of such argument will permit more sensitive guiding. An interesting exercise and base for discussion is to have the students study various programs of intermediate smartness, classify their bad moves by degrees of stupidity, and give the program grades (or say why they think doing so is silly!).

The stratification of the project has the good feature of allowing students to find their own levels. A slower child who gets only as far as the random player, nevertheless has the taste of success if his program does what it does well. Tendencies to feel inferior should be counteracted by the teacher's attitude and by the teacher's encouraging individual variations so that no child's final program is a mere subset of a more advanced one. The teacher's computer culture can be very relevant in this delicate kind of situation. Although the richness of programming permits students to generate many fertile ideas, sensitive filtering by the teacher can enormously improve the achievement-to-frustration ratio.

First Steps with the Students

A move in Nim consists of taking one, two, or three matchsticks from a given pile. Two players move alternately. The player who takes the last stick wins.

The first step is to see that everyone knows the rules and understands what the first program does, for example, by imitating its function or by writing imaginary scripts. In the course of discussing this the teacher introduces some names so the class can talk about what the program is doing.

Here is an example of a script:

THE NUMBER OF STICKS IS 8
JOAN TO PLAY. WHAT'S YOUR MOVE?
2
THE NUMBER OF STICKS IS 6
BILL TO PLAY. WHAT'S YOUR MOVE?
3
THE NUMBER OF STICKS IS 3
JOAN TO PLAY. WHAT'S YOUR MOVE?
3
JOAN IS THE WINNER!

Later in the project the teacher can insist that the students consider what happens when a player replies to WHAT'S YOUR MOVE? by 5 or COW. In the beginnning the teacher should discourage all but the most competent students from worrying about "funny" answers before getting the program to work with normal answers.

Examining the script you see that there must be names for:

- The current number of sticks—say STICKS
- The move—say MOVE
- The next player—say PLAYER
- And, a little more subtle, the other player—say OPPONENT

To be sure that everyone understands, they are asked to fill in these "Logo things" for successive rounds, following the previous script.

Round No.	:STICKS	:PLAYER	:OPPONENT	:MOVE
1	8	JOAN	BILL	2
2	?	?	JOAN	3
3	3	?	?	?

7.3.2. A Simple Scorekeeper

If this is the first game-playing program, the teacher builds up to it by asking some standard questions:

- What shall we call the procedure? (Let's say NIMPLAY)
- What must NIMPLAY do?
- What must NIMPLAY know?

Possible answers are

- Announce the remaining number of sticks.
- Announce the player to move.
- Get the move and make all the modifications.
- Recurse.

To do this, NIMPLAY must remember :STICKS, :PLAYERS, and :OPPONENT from the previous round and get :MOVE by asking for it. The first three things must be passed from one call to NIMPLAY to the next, so they should be inputs. On the other hand, :MOVE comes from the human player, so it does not need to be an input. If you look ahead, you may notice that later on :MOVE will sometimes come from a procedure—that is, when the machine gets to be smart enough to make its own moves. So to keep the door open for changes, the problems of getting :MOVE and using it are separated. The standard way to do this is to plan on a subprocedure—say, called GETMOVE.

Now students can write NIMPLAY:

```
TO NIMPLAY :STICKS :PLAYER :OPPONENT
          When in doubt, have lots of inputs.
PRINT SENTENCE [THE NUMBER OF STICKS IS] :STICKS
          Announce the number of sticks.
```

```
PRINT SENTENCE :PLAYER [TO PLAY. WHAT'S YOUR MOVE?]
MAKE "NEWSTICKS :STICKS - GETMOVE
        Pretend GETMOVE has already been written.
NIMPLAY :NEWSTICKS :OPPONENT :PLAYER
        Recursion line. Notice how :PLAYER
        and :OPPONENT are reversed.
END

TO GETMOVE
MAKE "MOVE READNUMBER
        See READNUMBER  procedure on page 97.
OUTPUT :MOVE
END
```

From Scorekeeper to Referee

As referee the program has some new tasks:

- To decide whether the game is over
- To declare the winner if it is over
- To make sure that :PLAYER takes 1, 2, or 3 sticks each time

The first two tasks are achieved by adding a test and a stop line to NIMPLAY. For example,

```
TEST :NEWSTICKS = 0
IFT PRINT SENTENCE :PLAYER [IS THE WINNER!]
IFT STOP
```

The third task can be accomplished by giving GETMOVE a "try-again" form, using the MEMBER? predicate which takes an item and a list as inputs and checks whether the item is in the list. MEMBER? can be given to students as a black box. The implementation of MEMBER? is explained on page 143.

```
TO GETMOVE
PRINT [YOU MAY TAKE 1, 2, OR 3 STICKS]
MAKE "MOVE READNUMBER
TEST MEMBER? :MOVE [1 2 3]
IFF OUTPUT GETMOVE        If the TEST is FALSE, try again.
OUTPUT :MOVE
END
```

With these changes, NIMPLAY is certainly a referee—but still has some rough edges. For example, when :STICKS is 2, GETMOVE gives permission to take 1, 2, or 3 sticks! And if :PLAYER takes 3 sticks, :STICKS becomes negative, and the game will go on forever, because of a "slip-by" bug. However, we shall leave it as an exercise to remedy these minor failings.

In presenting this section to students, the teacher may want to work through one of the two major modifications with the class and let the students struggle with the other. The slip-by bug we would leave to the class to discover and cure. Those who miss it at this stage will find its presence more obtrusive later. If so, a profitable discussion may develop on the question of why the bug was not found—perhaps because the human player always makes reasonable moves so that :STICKS never becomes negative even though the machine allows it. Later we shall see that when the machine makes its own moves, it is not to be so cooperative unless it is told to be.

Examples of individual frills to a referee program are: timing moves, declaring the winner a move or two ahead (!), allowing a player to take a move back, printing a score sheet, giving advice (!), establishing and imposing handicaps (!), and changing the rules.

7.3.3. A Mechanical Player

How can the machine choose a move? The simplest way is by using PICKRANDOM.[4] For example, you could allow GETMOVE to make the choice:[5] if a person is to play, use READNUMBER; if the machine is to play, use PICKRANDOM. But it has to be told whether the player is human or the computer. So it must have an input.

```
TO GETMOVE :PLAYER
TEST:PLAYER = [COMPUTER]
IFT MAKE "MOVE PICKRANDOM [1 2 3]
IFF PRINT [YOU MAY TAKE 1, 2, OR 3 STICKS]
IFF MAKE "MOVE READNUMBER
.
.
.    as before
```

At this stage the slip-by bug may become serious. One way to kill it is to tell GETMOVE about :STICKS and have it try again if :MOVE comes up greater than :STICKS. To do this you change the title line to:

---

[4]The PICKRANDOM procedure (page 142) can be written by the teacher and given to students as a "primitive."

[5]Notice this anthropomorphism. We find it useful to talk of procedures as agents, of their "state of knowledge," of "telling them," of having them "talk to" one another. And we present this to students as a deliberate metaphor that they may find useful.

```
TO GETMOVE :PLAYER :STICKS
```

and add a pair of lines after the two MAKEs.

```
TEST :MOVE > :STICKS
IFT OUTPUT GETMOVE :PLAYER :STICKS
```

Strategic Play

The plan for writing the Nim-playing program in many strata now calls for it to recognize a few special numbers and know what to do in those cases, but continue to play stupidly in other cases. However, by this time it is likely that the class has already discovered the full strategy. It may still be worthwhile to encourage at least some member to follow the original plan as an instructive joke. In this section we illustrate a general question-answer technique for classroom discussion to encourage habits of heuristic neatness in the students' own thinking.

A good exercise is to observe NIMPLAY in its present condition and to collect and classify its mistakes. An example of a classification made by a student is:

- DUMB MISTAKES
  - ★ There were 5 sticks and the machine took 2. (If the machine had any sense, it would leave the opponent with 4.)
  - ★ There were 6 or 7 sticks and the machine did not leave 4.
- SUPER DUMB MISTAKES
  - ★ There were 2 or 3 sticks and the machine did not take all!

We shall write a procedure to avoid first "super dumb mistakes" and then "dumb mistakes".

- Question: What program form? Answer: TEST.
- Question: What do we test for? English answer: Whether there are 1, 2, or 3 sticks. Logo answer: TEST MEMBER? :STICKS [1 2 3].
- Question: What is the action if the test is passed? English answer: Take all the sticks. Logo answer: OUTPUT :STICKS.
- Question: What if it is not passed? English answer: Move just like before. Logo answer: MAKE "MOVE PICKRANDOM [1 2 3].

Now put this together to make a procedure to make the move:

- Question: What must the procedure know? Answer: :STICKS—so it needs an input.

```
TO MAKEMOVE :STICKS
TEST MEMBER? :STICKS [1 2 3]
IFT OUTPUT :STICKS
IFF OUTPUT PICKRANDOM [1 2 3]
END
```

The procedure is used in place of PICKRANDOM in GETMOVE. So don't forget to change GETMOVE!

Now extra lines can be added. For example:

```
TEST :STICKS = 5
IFT OUTPUT 1
```

The Smart Player

By this time everyone should be very close to understanding the strategy, for example, in the following form:

- Question: How does the game end? Answer: When a player *leaves* 0 sticks.

So let's try making the main actor be the number of sticks we leave. If we can leave 0 that's great. But if we have more than 3 we can't. So we must think ahead.

- Question: What can we leave to help us leave 0 next time? Answer: 4. Because the opponent will leave 1, 2, or 3.
- Question: What can we leave so as to be able to leave 4 next time? Answer: 8.
- Question: So 0, 4, 8 are good numbers to shoot at for leaving. What others? Answer: 12, 16, . . .
- Question: How could you describe the numbers 0, 4, 8, 12, 16, . . . Answer: They are all divisible by 4. REMAINDER :NUMBER 4 is 0.
- $64 Question: If I give you :NUMBER, how can you use it to find the next number down divisible by 4? Answer: Subtract REMAINDER :NUMBER 4.

So there we are! The smart invincible Nim player is made by replacing MAKEMOVE by SMARTMOVE:[6]

```
TO SMARTMOVE :STICKS
MAKE "REM REMAINDER :STICKS 4
IF :REM = 0 OUTPUT 1      It really doesn't matter in this case.
OUTPUT :REM
END
```

[6]Use the REMAINDER procedure from page 80.

7.3.4. Frills and Modifications

Write superprocedures or make additions to the present procedure to produce transcripts such as the one following.

NIM
DO YOU KNOW HOW TO PLAY NIM?
NO
HERE ARE THE RULES: YOU WILL BE SHOWN A COLLECTION OF X'S. YOU MAY REMOVE 1, 2, OR 3. THE PLAYER WHO TAKES THE LAST X WINS. THIS IS PROBABLY TOO VAGUE FOR YOU TO UNDERSTAND, BUT TRY PLAYING AND I'LL CORRECT YOUR MISTAKES.
ARE YOU READY?
I AM
PLEASE SAY "YES" OR "NO"
YES
OK. NOW TELL ME THE NAME OF THE FIRST PLAYER.
JOAN
NOW THE NAME OF THE OTHER PLAYER.
COMPUTER
HOW MANY STICKS DO YOU WANT TO START WITH?
THIRTY-ONE
I'M A DUMB COMPUTER. TYPE A PROPER NUMERAL.
31
JOAN TO PLAY.
THERE ARE 31 STICKS.
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
JOAN, TAKE 1, 2, OR 3.
3
COMPUTER TO PLAY.
THERE ARE 28 STICKS.
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
I TAKE 1
JOAN TO PLAY.
THERE ARE 27 STICKS.
XXXXXXXXXXXXXXXXXXXXXXXXXXX
TAKE 1, 2, OR 3.
3
.
.
.

In addition to such frills, there are unlimited possibilities to play with the ideas in the procedure after it has been made to work. Here are three examples to illustrate the idea that the project has not necessarily run out when the procedure is debugged:

- An interesting simple modification to the rule of the game is to change the 1-2-3 rule to a 1-2 rule or a 1-2-3-4-5 rule. Write a procedure that asks what rule is to be used and then plays by that rule.
- Our stop rule was: the player who takes the last stick wins. Change this to: whoever takes the last stick loses. (The latter is the traditional form.)
- The game can be embedded in a more complex one, such as moving counters along marked paths on a board. If there is just one linear path, the problem is identical, but if branches are allowed, interesting complexities arise.

7.3.5. A Listing of the NIMPLAY Procedures

Here is a listing of the final form of the NIMPLAY procedures. Besides the three procedures listed below, the project also makes use of the REMAINDER procedure on page 80, the READNUMBER procedure on page 98, and the MEMBER? procedure on page 143.

```
TO NIMPLAY :STICKS :PLAYER :OPPONENT
PRINT SENTENCE [THE NUMBER OF STICKS IS] :STICKS
PRINT SENTENCE :PLAYER [TO PLAY. WHAT'S YOUR MOVE?]
MAKE "NEWSTICKS :STICKS - (GETMOVE :PLAYER :STICKS)
TEST :NEWSTICKS = 0
IFT PRINT SENTENCE :PLAYER [IS THE WINNER!]
IFT STOP
NIMPLAY :NEWSTICKS :OPPONENT :PLAYER
END

TO GETMOVE :PLAYER :STICKS
TEST :PLAYER = [COMPUTER]
IFT MAKE "MOVE SMARTMOVE
IFF PRINT [YOU MAY TAKE 1, 2, OR 3 STICKS]
IFF MAKE "MOVE READNUMBER
TEST MEMBER? :MOVE [1 2 3]
IFF OUTPUT GETMOVE :PLAYER :STICKS
TEST :MOVE > :STICKS
IFT OUTPUT GETMOVE :PLAYER :STICKS
OUTPUT :MOVE
END
```

```
TO SMARTMOVE
MAKE "REM REMAINDER :STICKS 4
IF :REM = 0 OUTPUT 1
OUTPUT :REM
END
```

7.4. Growing Flowers

In this section we'll design a movie, shown in Color Plate II. The scene starts at night, with some bulbs planted in a lawn. The sun rises and the sky grows light. Then the flowers begin to grow. As the flowers grow, they sprout leaves and buds. Finally they burst into color and bloom.

This movie is more complex than the one in Section 4.4, because it requires tighter coordination between sprites and tiles. The grass is made up of tiles, and the sun is a sprite. The flowers are made up of both tiles and sprites, which will require some subtlety in the implementation.

7.4.1. Coordinates for Sprites and Tiles

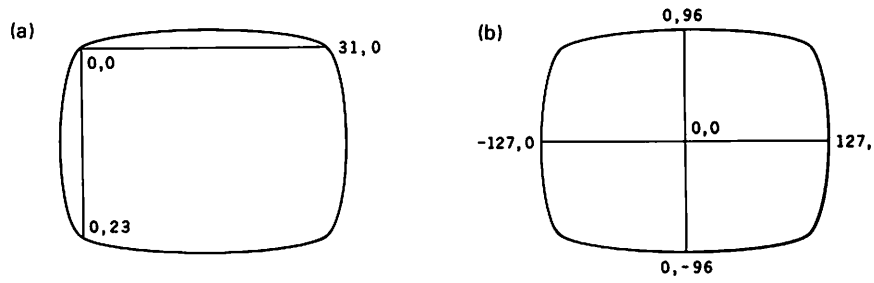


Figure 7.1: Comparison of tile coordinates (a) with sprite coordinates (b).

The problem with making shapes by combining tiles and sprites is that the *x, y* coordinates used to position sprites on the screen are not the same as the row and column numbers used to position tiles. Row and column numbers are interpreted as character positions, shown in Figure 7.1a. Columns are numbered from left to right, 0 through 31, and rows are numbered from top to bottom, 0 through 23. On the other hand, sprite and turtle coordinates are specified in terms of an *x* coordinate between −127 and 127, and a *y* coordinate between −96 and 96, as shown in Figure 7.1b.

It will be useful, therefore, to have some procedures that convert from one coordinate system to the other. Here is a procedure that returns the *x* coordinate corresponding to a given column:

```
TO COLX :COL
OUTPUT (8 * :COL) - 128
END
```

Similarly, we can convert a row number to a *y* coordinate:

```
TO ROWY :ROW
OUTPUT (-8 * :ROW) - 96
END
```

Using these two procedures, we can write a procedure PUTSPRITE, similar to PUTTILE, which positions a sprite at a given column and row:

```
TO PUTSPRITE :SPRITE :COLUMN :ROW
TELL :SPRITE
SXY (COLX :COLUMN) (ROWY :ROW)
END
```

Converting coordinates the other way, it is also useful to be able to find the row and column position of a sprite. The following procedures return the column number corresponding to the XCOR of the current object and the row number corresponding to YCOR:

```
TO XCOLUMN
OUTPUT (XCOR + 128) / 8
END

TO YROW
OUTPUT (-YCOR + 96) / 8
END
```

7.4.2. Defining the Shapes

To make the movie, you will need some shapes, both for tiles and sprites.

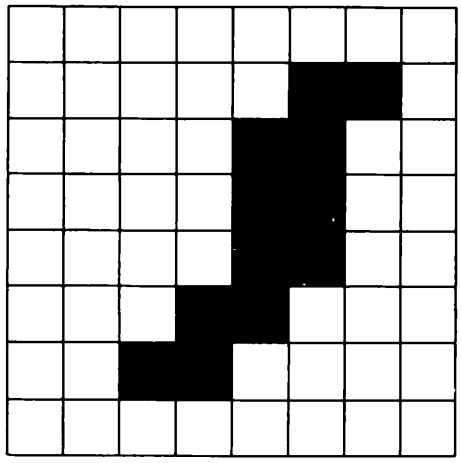
The grass will be a regular pattern, consisting of a background with a small "blade" in it, as shown in Figure 7.2. We'll use tile number 8 for this pattern

```
MAKE "GRASS 8
```

and give it a foreground color OLIVE and a background color GREEN, which will color the blade slightly darker than the rest.

We'll also use the tiles for the stems and leaves of the flowers. There are four different tiles: ordinary left and right halves of a stem, and left and right halves with leaves on them:

```
MAKE "LEFTSTEM 100
MAKE "RIGHTSTEM 101
MAKE "LEFTSTEM1 98
MAKE "RIGHTSTEM1 99
```



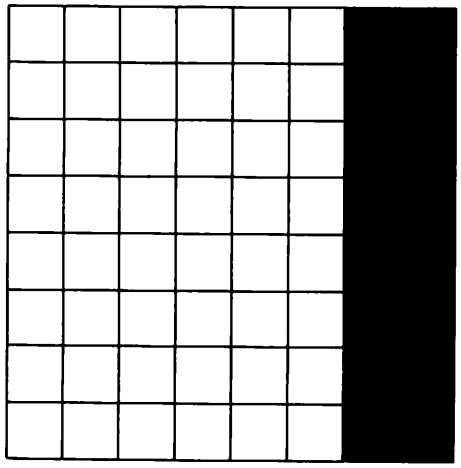
```
LEFTSTEM
RIGHTSTEM
LEFTSTEM1
RIGHTSTEM1
```

Figure 7.2: Shape for GRASS tile.

These will all be colored OLIVE with a CLEAR background color, so we can put them all in the same color group. (See page 61.) Figure 7.3 shows the shapes for these tiles:

As the flowers grow, they evolve through three different shapes—bulbs, buds, and blooms:

```
MAKE "BULB 6
MAKE "BUD 7
MAKE "BLOOM 8
```



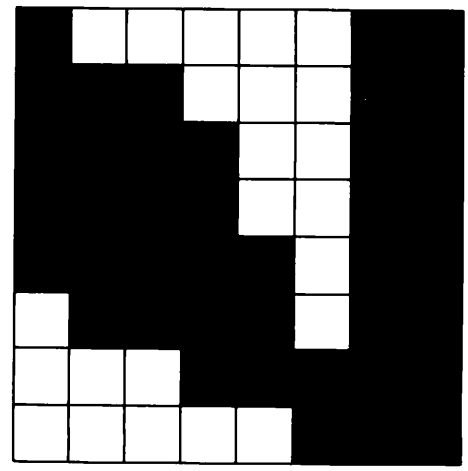
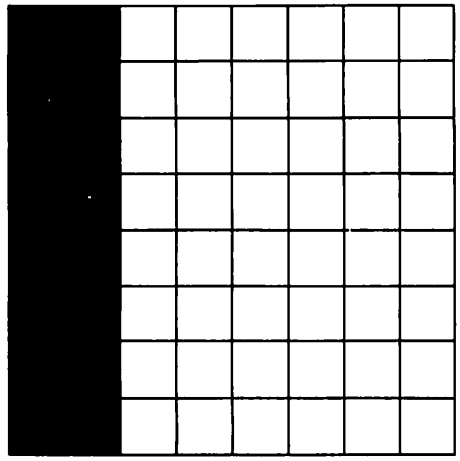
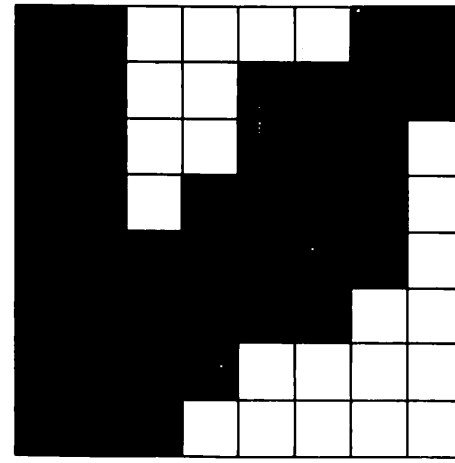


Figure 7.3: Tile shapes flower stems. (a) (b)



(c)



(d)

Figure 7.3: (Continued)

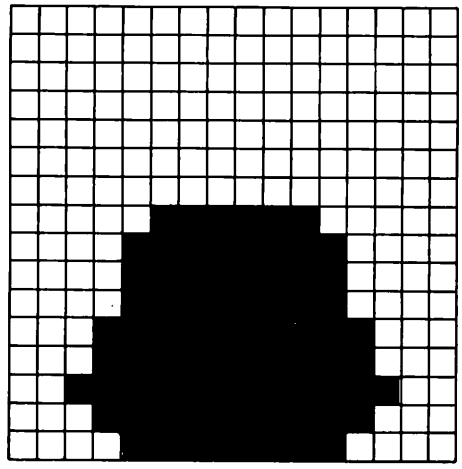
BULB

BUD

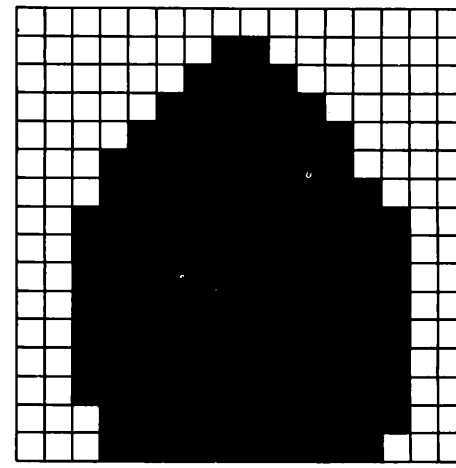
BLOOM

Use MAKESHAPE to define these three shapes as shown in Figure 7.4.

Finally, you need a shape for the sun. The simplest thing to do here is to use the BALL shape built into Logo.

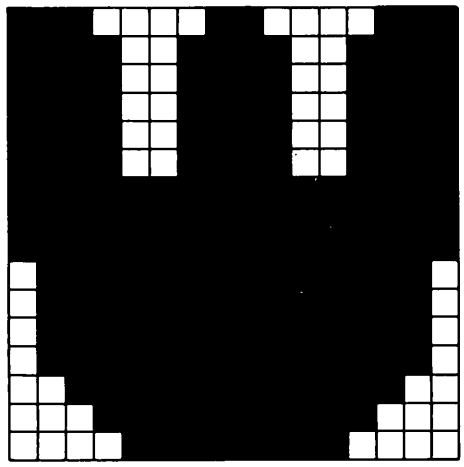


(a)



(b)

Figure 7.4: Sprite shapes for the flowers.



(c)

Figure 7.4: (Continued)

7.4.3. The Grass

To make the grass, you need to lay down a solid block of tiles, starting from some row on the screen and working downwards. This is exactly the way that the WATER was handled in the movie in Section 4.4, and you can use the same MAKEROWS procedure that we used there. For MAKEROWS, you must specify the tile number and the top of the block. Row 14 is a good position at which to start:

```
MAKE "GRASSTOPROW 14

TO MAKEGRASS
TELL TILE :GRASS
SETCOLOR SENTENCE :OLIVE :GREEN
MAKEROWS :GRASS :GRASSTOPROW
END
```

7.4.4. Planting the Bulbs

Planting a bulb is simply a matter of positioning a sprite that is carrying the BULB shape. The following procedure positions a given sprite at a given column and row. Notice how the PUTSPRITE procedure (page 109) comes in handy.

```
TO PLANTBULB :SPRITE :COLUMN :ROW
PUTSPRITE :SPRITE :COLUMN :ROW
TELL SPRITE :SPRITE
SETCOLOR :GREEN
CARRY :BULB
END
```

Now you can plant all the bulbs. Use sprites 0 through 5 to make six flowers. Choose column numbers to spread the flowers out on the screen, and position the bulbs two rows above the top grass row.[7]

```
TO PLANTBULBS
MAKE "BULBROW :GRASSTOPROW - 2
PLANTBULB 0 5 :BULBROW
PLANTBULB 1 8 :BULBROW
PLANTBULB 2 12 :BULBROW
PLANTBULB 3 20 :BULBROW
PLANTBULB 4 16 :BULBROW
PLANTBULB 5 24 :BULBROW
END
```

7.4.5. Sunrise

To make the sunrise, you need only have a sprite move upwards carrying the BALL shape, while the sky changes color: from BLACK to BLUE to SKY to CYAN. You can use sprite number 10 for the sun, starting it near the top row of grass towards the right of the screen.

```
MAKE "SUN 10

TO SUNRISE
COLORBACKGROUND :BLUE
TELL SPRITE :SUN
SXY 75 20
CARRY :BALL
SETCOLOR :YELLOW
REPEAT 30 [FORWARD 1 WAIT 10]
COLORBACKGROUND :SKY
REPEAT 30 [FORWARD 1 WAIT 10]
COLORBACKGROUND :CYAN
END
```

7.4.6. Growing the Flowers

Now you must grow the bulbs into flowers. The first step is to set the color of the tiles for the stem:

```
TO MAKESTEM
TELL TILE :LEFTSTEM
SETCOLOR (SENTENCE :OLIVE :CLEAR)
END
```

[7]This is because a sprite is 2 character positions high, and the inputs supplied to PUTSPRITE specify the position of the top-left corner of the shape.

The color of the stems is OLIVE, slightly darker than the GREEN of the grass and the bulbs. Note that since all the stem pieces are in the same color group, you need only set the color of one of the pieces. (See Section 4.3.2.)

To make the flower grow, you want to make it appear that a stem is growing, pushing up the flower shape carried by a sprite. In order to accomplish this, you can use a trick: lay down a section of the stem in the *same* position as the sprite shape. Since sprites cover tiles, you will not see the stem at this point. Now move the sprite slowly upwards 8 units (the height of a tile). As the sprite moves, more and more of the stem will be uncovered. Every time the STEM procedure is repeated, the tiles will be placed in a higher position and the flower will grow by an amount equal to one tile unit. Here is the procedure that accomplishes this:

```
TO STEM
PUTTILE :LEFTSTEM XCOLUMN YROW + 1
PUTTILE :RIGHTSTEM XCOLUMN + 1 YROW + 1
REPEAT 8 [FORWARD 1]
END
```

The procedures for finding the row and column of a sprite (Section 7.4.1) come in handy here in positioning the tiles. The extra unit added to the row coordinate is because sprites are 2 tile spaces high, and you should place the tile under the lower half of the sprite. The right half of the stem also needs an extra unit added to its column coordinate.

To make the flower sprout leaves, you simply need to replace an ordinary stem shape by one of the stem shapes with leaves on it. Pick a height on the stem a little below the top. Here are procedures that sprout a leaf to the left:

```
TO LEFTLEAF
PUTTILE :LEFTSTEM1 XCOLUMN YROW + 2
END
```

and a leaf to the right:

```
TO RIGHTLEAF
PUTTILE :RIGHTSTEM1 XCOLUMN + 1 YROW + 3
END
```

The + 2 and + 3 added to the row determine how far below the current sprite position the leaf will sprout.

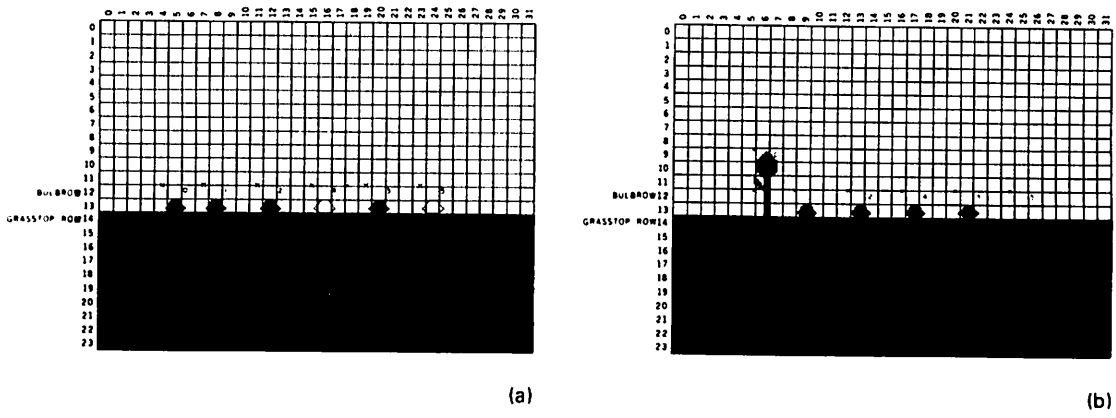
Now you can grow a complete flower. Start with a sprite shaped like a BULB, grow some stem, change the sprite shape to a BUD, sprout a leaf, grow more stem, sprout another leaf, change the bud's color, and change the shape to a BLOOM. The procedure that does this takes as inputs the sprite number, the number of stem segments to grow, and the flower's color:

```
TO GROW :S :LENGTH :COLOR
TELL SPRITE :S
REPEAT :LENGTH [STEM]
CARRY :BUD
LEFTLEAF
REPEAT :LENGTH [STEM]
RIGHTLEAF
WAIT 30
SETCOLOR :COLOR
WAIT 30
CARRY :BLOOM
END
```

By calling this procedure repeatedly, you can grow all the flowers:

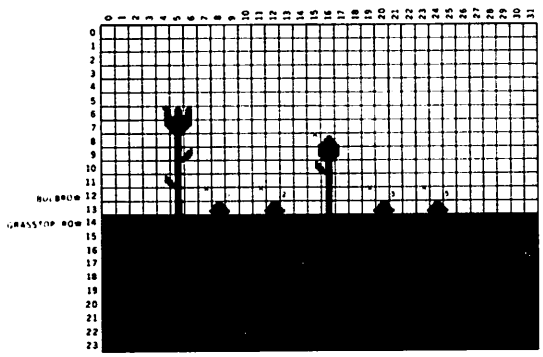
```
TO GROWFLOWERS
MAKESTEM
GROW 0 3 :WHITE
GROW 4 4 :YELLOW
GROW 5 3 :RUST
GROW 1 4 :RED
GROW 3 2 :ORANGE
GROW 2 3 :PURPLE
END
```

The action of GROWFLOWERS is illustrated in Figure 7.5.

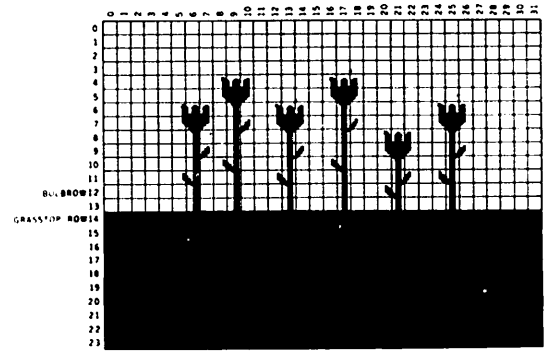


(a) (b)

Figure 7.5: Illustrations of the GROWFLOWERS procedure. The "X" marks in each figure show the coordinate locations of each sprite as the movie develops.



(c)



(d)

Figure 7.5: (Continued)

7.4.7. Combining All the Pieces

Now it only remains to put everything together. Begin by clearing the screen and setting up a BLACK background.

```
TO FLOWERMOVIE
CLEARSCREENANDSPRITES
COLORBACKGROUND :BLACK
MAKEGRASS
PLANTBULBS
SUNRISE
WAIT 30
GROWFLOWERS
```

The CLEARSCREENANDSPRITES procedure, which clears the screen and makes the sprites invisible, is given on page 68.

7.4.8. Elaborations

With this as a beginning, you can extend the movie in all sorts of ways. Make a better sunrise, in which the sky changes through all sorts of beautiful colors at dawn. Add a nightfall, in which the sun sets, the sky darkens, and the flowers close. Add some clouds that drift by overhead or a bee that flits from flower to flower. Movies like this are good projects because you can add parts little by little until you end up with something quite elaborate.

CHAPTER 8

Writing Interactive Programs

We've already seen examples of Logo programs that use PRINT to print information on the display screen and programs that use READLINE to input information from the keyboard. This chapter reviews these commands and describes more elaborate ways of handling input and output. As an example, we show how to create "instant response" Logo systems for very young children. We also show how a Logo-based "dynaturtle" can be used to introduce elementary school children to computer projects involving motion and simple physics.

8.1. Controlling Screen Output

The Logo PRINT command, as used throughout the preceding chapters, is the main command for showing information on the display screen. PRINT takes a word or a list as an input, types it on the screen, and moves the cursor to the next line. Remember that lists are printed without the outer brackets.

The command TYPE is just like PRINT, except that it does not move the cursor to a new line after printing. Compare

```
TO COUNT :X
PRINT :X
COUNT :X + 1
END

COUNT 1
1
2
3
.
.
.
```

```
TO COUNT1 :X
TYPE :X
TYPE ", "
COUNT1 :X + 1
END

COUNT1 1
1, 2, 3, . . .
```

The PRINTCHAR (abbreviated PC) takes a number 0 through 255 as input and prints the character corresponding to that number. Recall (from Section 4.3.3) that these 255 characters include Logo's printing characters plus any tiles you have defined. Here's a way to use PRINTCHAR to see all of the characters that are currently defined:

```
TO SHOWCHARS :N
IF :N > 255 STOP
PRINTCHAR :N
SHOWCHARS :N + 1
END

SHOWCHARS 0
```

8.2. Keyboard Input

The READLINE command is used to read input from the keyboard, as shown in Section 6.4. READLINE causes the computer to wait for you to type in a line (terminated by ENTER) and then outputs that line as a list. Remember that what you type in will *always be interpreted as a list*. For example, if you type in a single word, READLINE returns a list containing that word:

```
MAKE "ANS READLINE
>100
IF :ANS = 100 PRINT "YES ELSE PRINT "NO
NO
IF :ANS = [100] PRINT "YES ELSE PRINT "NO
YES
```

Using READLINE, you can implement a useful procedure that returns a word typed at the keyboard, obtained as the first element of the list returned by READLINE:

```
TO READWORD
OUTPUT FIRST READLINE
END
```

Compare the use of READLINE in the READNUMBER procedure on page 98.

In addition to the "line at a time" input from READLINE, Logo also provides "character at a time" input through the command READCHAR (abbreviated RC). READCHAR causes the computer to pause and wait for you to type in a single character (without ENTER) and then outputs the character that was typed:

```
MAKE "ANS READCHAR
X
IF :ANS = "X PRINT "YES ELSE PRINT "NO
YES
```

8.2.1. Example: Instant Response for Very Young Children

The following program uses READCHAR to provide "instant response" control of the turtle for drawing:

```
TO INSTANT
COMMAND
INSTANT
END

TO COMMAND
MAKE "COM READCHAR
IF :COM = "F FORWARD 10
IF :COM = "R RIGHT 30
IF :COM = "L LEFT 30
IF :COM = "C CLEARSCREEN
END
```

This program causes the turtle to move in response to individual keystrokes: F for go forward, L for left, R for right, and C for clearing the screen and starting over. It can form a good tool for using computer graphics with very young children. This same instant-response mechanism is also useful in designing languages for use by the physically handicapped, for which it is important to minimize the number of keystrokes required.

It is easy to increase the repertoire of this INSTANT language by adding additional lines to the COMMAND procedure. For example, if you want the S key to make the turtle draw a small square, you define a procedure called SQUARE (say, that draws a square of side 20) and add to COMMAND the line

```
IF :COM = "S SQUARE
```

Section 11.2.1 discusses more elaborate extensions to INSTANT.

8.2.2. Keyboard Control of an Ongoing Process

Notice that READLINE and READCHAR both make the computer *stop and wait* for something to be typed. Logo also allows you to write programs in which the keyboard is used to control an ongoing process. That is, if a character is typed at the keyboard, the program is able to respond to the character; but if nothing is typed, the program is able to keep running

anyway. Such programs are implemented in Logo using the RC? command. RC? outputs TRUE or FALSE depending on whether a character has been typed at the keyboard. When RC? is TRUE, the next READCHAR command returns the character that was typed, otherwise READCHAR has to wait until a character is typed.[1] For example, you can modify the INSTANT program so that it makes the turtle move forward *continually*, turning right or left in response to the letters R and L typed at the keyboard. We'll call the resulting program DRIVE:

```
TO DRIVE
FORWARD 1
COMMAND
DRIVE
END

TO COMMAND
MAKE "COM READKEY
IF :COM = "R RIGHT 30
IF :COM = "L LEFT 30
END
```

The difference between this program and INSTANT is that the turtle goes forward each time, rather than when an F is typed. Whereas the COMMAND program used by INSTANT calls READCHAR, the COMMAND program used in DRIVE calls READKEY. READKEY is a procedure that, if a character has been typed, outputs that character, and otherwise outputs the empty list. READKEY is implemented using RC?:

```
TO READKEY
IF RC? OUTPUT READCHAR
OUTPUT [ ]
END
```

READKEY is a good example of a useful "primitive" that can be supplied by the teacher to students working on interactive programming projects.

[1] More specifically, characters typed at the keyboard are saved in a input buffer. READCHAR reads characters from the buffer one by one. RC? outputs TRUE if the buffer is not empty. If Logo is doing a lot of processing in between characters, and if one types characters very fast, a sequence of characters may build up in the buffer, and the program may seem to "fall behind" in its responses to the typed characters.

8.2.3. Instant Response with Sprites

You can make all sorts of computer programs for very young children by installing an INSTANT-style controller for the sprites. The possibilities here are virtually unlimited. One simple example is to begin with a cluster of balls that "explodes" from the center of the screen. To do this, put all the sprites at home, with their headings set at 10-degree intervals, and start them moving:

```
TO BEGIN
TELL :ALL
SETSPEED 0
HOME
CARRY :BALL
SETCOLOR :RED
EACH [SETHEADING 10 * YOURNUMBER]
SETSPEED 10
END
```

Now put the balls under keyboard control, using commands that make them move slowly or quickly, reverse direction, change color, or change shape. The following COMMAND selection is only a sample:

```
TO ACTION
BEGIN
LOOP
END

TO LOOP
COMMAND
LOOP
END

TO COMMAND
MAKE "COM READCHAR
IF :COM = "S SETSPEED 5
IF :COM = "Q SETSPEED 100
IF :COM = "R SETSPEED -SPEED
IF :COM = "M EACH [SETCOLOR RANDOM]
IF :COM = "4 CARRY 4
IF :COM = "5 CARRY 5
IF :COM = "B BEGIN
END
```

8.3. Example: The Dynaturtle Program

DYNATURTLE is an extension of the Logo turtle, developed by Andy deSessa as a computer-based physics environment for elementary school students. It has also been used as an experimental setting for investigating the role of intuition in learning physics. See A. diSessa [6] for details. This section, based on a paper by Dan Watt, is a description of DYNATURTLE that can be used by students and teachers.

8.3.1. What is a Dynamic Turtle?

A *dynamic* turtle or *dynaturtle* behaves as though it were a rocket ship in outer space. To make it move you have to give it a *kick* by "firing a rocket." It then *keeps moving* in the *same* direction until you give it another kick. When you change its direction, it does not move in the new direction until you give it a new kick. Its new motion is a combination of the old motion and the motion caused by the new kick. You may need to experiment with dynamic commands for a while before you understand how the dynaturtle works.

To use the dynaturtle, you will need the procedures in this section. Here is the main procedure:

```
TO DT
MOVETURTLE
COMMAND
DT
END
```

The procedure DT moves the turtle (if you have given it a kick), checks to see if you've typed a command, and then starts doing DT all over again. It keeps running until you stop the procedure by pressing BACK.

In addition to the SIN and COS procedures discussed in Section 8.3.4, and the READKEY procedure on page 119, here are three other procedures you will need.[2]

```
TO MOVETURTLE
SXY ( XCOR + :VX ) ( YCOR + :VY )
END
```

[2]The KICK procedure uses the trigonometric functions SIN and COS in order to change the turtle's velocity, which can be thought of as a vector *(VX, VY)*. KICK together with the SIN and COS procedures would normally be used by elementary school children as a black box.

```
TO COMMAND
MAKE "COM READKEY
IF :COM = "R RIGHT 30
IF :COM = "L LEFT 30
IF :COM = "K KICK
END

TO KICK
MAKE "VX :VX + SIN HEADING
MAKE "VY :VY + COS HEADING
END
```

To start the dynaturtle, you need a procedure to initialize the dynaturtle's position and velocity:

```
TO STARTUP
TELL TURTLE
CLEARSCREEN
MAKE "VX 0
MAKE "VY 0
END
```

8.3.2. Activities with a Dynaturtle

To try out the dynaturtle, type

```
STARTUP
DT
```

At first the turtle will stay at the center of the screen. The COMMAND procedure allows three different commands at present. Later you can change them in any way you like.

- If you type R the turtle will turn right 30 degrees.
- If you type L the turtle will turn left 30 degrees.
- If you type K you will give the turtle a *kick* in the direction it is heading.

The turtle will now keep moving in the direction it started until you give it another kick in some direction.

Start the dynaturtle moving by typing

```
STARTUP
DT
```

and then typing the K key for "kick."

- Make the dynaturtle move in a different direction by typing the R or L key.
- Make the dynaturtle move horizontally across the screen.
- Make the dynaturtle go faster.
- Make the dynaturtle go slower without changing direction.
- Before you start the dynaturtle, place a marker somewhere on the screen. (You can use a tile to form the marker.) Then start the dynaturtle and see if you can move the turtle to the marker. If the marker is easy for the dynaturtle to get to, move it over a little and try again.
- Start the dynaturtle from the center of the screen. Can you make it stop?
- Draw a circular "racetrack" on the screen and see if you can "drive" the dynaturtle around a track.
- Move the dynaturtle to the marker, and make it stop there.

When you try these activities you may find that some of them are harder than you thought. The problems you have making the dynaturtle do what you want it to do are similar to the problems astronauts would have moving around in outer space or maneuvering a rocket to connect up with a space platform or land on the moon.

8.3.3. Changing the Dynaturtle's Behavior

After some experimentation with the dynaturtle, you may want to make changes in the dynaturtle procedures. Since changes in the dynaturtle's behavior are controlled by the COMMAND procedure, you can start by changing that procedure as follows:

```
TO COMMAND
MAKE "COM READKEY
IF :COM = "R RIGHT 30
IF :COM = "L LEFT 30
IF :COM = "K KICK
END
```

If you like, you can change the *angle* the dynaturtle rotates when you type R or L by changing the 30 in COMMAND to another number.

Your COMMAND procedure would now look like this:

```
TO COMMAND
MAKE "COM READKEY
IF :COM = "R RIGHT 30
IF :COM = "L LEFT 30
IF :COM = "K KICK
IF :COM = "U PENUP
IF :COM = "D PENDOWN
END
```

Of course, you can change the key names for carrying out the commands by changing the letters on the keyboard. Some people like to have the right and left keys next to each other on the keyboard. If you choose S for "left" and D for "right," then the COMMAND procedure becomes:

```
TO COMMAND
MAKE "COM READKEY
IF :COM = "D RIGHT 30
IF :COM = "S LEFT 30
IF :COM = "K KICK
IF :COM = "U PENUP
IF :COM = "D PENDOWN
END
```

Another possible change is to make the *force* of the kick a variable. If you did this, you would have to change the KICK procedure and the STARTUP procedure as well as COMMAND.

```
TO KICK :FORCE
MAKE "VX :VX + :FORCE * ( SIN HEADING )
MAKE "VY :VY + :FORCE * ( COS HEADING )
END
```

You would also have to add a line to STARTUP to set the starting value for the force:

```
TO STARTUP
TELL TURTLE
CLEARSCREEN
MAKE "VX 0
MAKE "VY 0
MAKE "FORCE 1
END
```

You can choose any value you want for the starting force.

You would now have to change the KICK line in the COMMAND procedure to read

```
IF :COM = "K KICK :FORCE
```

Also, you could add two more commands (say, H and S for "harder" and "softer") that increased and decreased the force. The COMMAND procedure would now be

```
TO COMMAND
MAKE "COM READKEY
IF :COM = "R RT 30
IF :COM = "L LT 30
IF :COM = "K KICK :FORCE
IF :COM = "U PENUP
IF :COM = "D PENDOWN
IF :COM = "H MAKE "FORCE :FORCE + 1
IF :COM = "S MAKE "FORCE :FORCE - 1
END
```

Now try out the dynaturtle with some of these changes, and see what can happen.

Some other possible changes:

- Add a "reverse kick" command that makes the dynaturtle move more slowly.
- Add commands that make the turtle *print* its speed, heading, and kick force.

8.3.4. Sines and Cosines

The dynaturtle program makes use of procedures SIN and COS that output the sine and cosine of a given angle. Since numbers in TI Logo must be integers, sines and cosines cannot be computed in any straightforward way.

The SIN and COS procedures used by the dynaturtle program take an angle as input and return integer approximations to 3 times the sine of the angle and 3 times the cosine of the angle. The scale factor 3 was chosen to allow for enough different values for kicks: -3, -2, -1, 0, 1, 2, 3, and at the same time be a good scale for working with the dynaturtle.

The implementation of these procedures is based on a clever trick.[3] Choose a sprite, say, sprite 0, and move it invisibly on the screen. If the sprite is moving with speed S at heading H, then the *x* component of its velocity will be S times the sine of H, and the *y* component of its velocity will be S times the cosine of H. Since Logo includes built-in operations for retrieving the *x* and *y* components of a sprite's velocity, you can use this "invisible sprite" technique to compute sines and cosines:

```
TO SIN :H
MAKE "ACTIVE WHO
TELL SPRITE 0
SETSPEED 3
SETHEADING :H
MAKE "ANS XVEL
TELL :ACTIVE
OUTPUT :ANS
END

TO COS :H
MAKE "ACTIVE WHO
TELL SPRITE 0
SETSPEED 3
SETHEADING :H
MAKE "ANS YVEL
TELL :ACTIVE
OUTPUT :ANS
END
```

The use of ACTIVE here is to allow the procedures to be used without fouling up any TELLs that are used outside the procedure. We use WHO to find the current active graphics object and then restore this with TELL before leaving the SIN and COS procedures. This is really more general than you need for dynaturtle, because ACTIVE will always be the turtle. It is also not necessary to set sprite 0's speed to 3 each time. The procedures are shown here in general form so that you can use them in other applications as well.

[3]Tricks such as these are sometimes referred to as computer "hacks" because they take advantage of special properties of a particular implementation and usually do not generalize in any meaningful way. This hack is due to Roger Kirchner of Carleton College.

CHAPTER 9

Logo Music

In addition to working with numbers, words, and lists and creating animated graphics, you can use TI Logo II to generate music from notes over a range of three octaves. You can play one, two, or three voices simultaneously and also make sound effects with a noise generator and a drum. You can execute other Logo commands while music is playing, thus providing musical accompaniment for programs. You can also synchronize music playing with other Logo commands.[1]

9.1. Playing Melodies

The Logo music system has the following basic organization: you first set up the entire piece to be played, using commands that store information in an area of computer memory called the *music buffer.* Afterwards, you play the music that has been stored in the buffer. Thus, there are two kinds of commands in the music system: commands that enter music information in the buffer and commands that play the music. When you are entering music (using the first kind of command) you will not hear any notes being played.

The MUSIC command is used to enter notes in the music buffer. MUSIC takes as inputs two lists: a list of pitches and a list of durations. The pitches are numbered chromatically, with 0 as middle C. For example, in the command

```
MUSIC [0 2 4 0] [4 4 4 4]
```

the first list of four numbers enters pitches for the four notes C, D, E, C. The second lists gives each note a duration of 4 time units. This is the first measure of the tune *Frere Jacques.*

To actually hear the music played, give the command PLAYMUSIC (abbreviated PM). PLAYMUSIC plays the notes that have been entered in the buffer. Every time you execute PLAYMUSIC, the notes in the music buffer will be played again.

Each time you use the MUSIC command, additional notes are appended to the end of the buffer. So if you again execute

```
MUSIC [0 2 4 0] [4 4 4 4]
```

[1]Music commands are not a part of the first release of TI Logo.

and then PM, you will hear the set of four notes played twice—the first two measures of *Frere Jacques.*

When you enter notes using the MUSIC command, the notes are designated to be played in one of four voices. The voice designation is controlled by the SETVOICE command. If you do not specify any voice, music will go to voice 1. SETVOICE 0 clears the music buffer. Section 9.2 below shows how to use SETVOICE to play notes in harmony. You may also want to make use of the following CM procedure (abbreviation for "clearmusic") that clears out the music and sets the voice to voice 1:

```
TO CM
SETVOICE 0
SETVOICE 1
END
```

Logo has a range of slightly over 3 octaves. Pitch 0 is middle C. The highest defined pitch is 24 (C two octaves above middle C). The lowest defined pitch is −15 (the pitch A one octave plus one third below middle C).[2] Figure 9.1 shows the correspondence between Logo pitch numbers and conventional music notation:

								A	A#	B	C
								−15	−14	−13	−12
C#	D	D#	E	F	F#	G	G#	A	A#	B	C
−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1	0
C#	D	D#	E	F	F#	G	G#	A	A#	B	C
1	2	3	4	5	6	7	8	9	10	11	12
C#	D	D#	E	F	F#	G	G#	A	A#	B	C
13	14	15	16	17	18	19	20	21	22	23	24

Figure 9.1: Correspondence between Logo chromatic pitch numbers and conventional music notation. Middle C is assigned to pitch number 0.

[2]The valid pitch range is different when using the MAJOR mode. See Section 9.1.3 below.

9.1.1. A Simple Tune

Now you can complete *Frere Jacques*, writing separate procedures for the different phrases:

```
TO F1
MUSIC [0 2 4 0] [4 4 4 4]
END

TO F2
MUSIC [4 5 7] [4 4 8]
END

TO F3
MUSIC [7 9 7 5 4 0] [3 1 2 2 4 4]

END

TO F4
MUSIC [0 -5 0] [4 4 8]
END
```

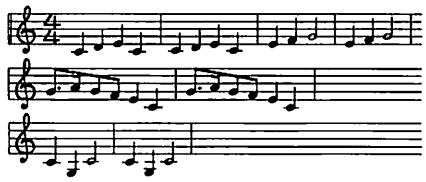


Figure 9.2: The music for *Frere Jacques*.

Observe how the duration numbers specify time units, so that in this piece a duration of 4 corresponds to a quarter note, 8 to a half note, 2 to an eighth note, 1 to a sixteenth note, and 3 to a dotted eighth note.

The complete tune is formed by playing each phrase twice:

```
TO FRERE
F1
F1
F2
F2
F3
F3
F4
F4
END
```

To play the piece, you use the CM procedure and type:

```
CM
FRERE
PM
```

9.1.2. Tuneblocks

The technique of writing each phrase as a separate procedure leads to *tuneblocks*, a musical game invented at MIT by Jeanne Bamberger.

One way to play tuneblocks is to create new tunes by rearranging the parts of a given tune. Here is a new tune assembled from the same blocks used in *Frere Jacques*:

```
F1 F4 F1 F4 F3 F2 F3 F2 F4 F4
```

To experiment with forming new tunes from a given set of blocks, you clear the buffer with CM, execute the sequence of procedures for the blocks you want to play, then follow with PM. Since each tuneblock is a separate Logo procedure, you can break the sequence of procedures into as many Logo command lines as you wish. For instance, the commands

```
CM F1 F4 F1 F4 F3 F2 F3 F2 F4 F4 PM
```

and

```
CM
F1 F4 F1 F4
F3 F2 F3 F2
F4 F4
PM
```

both play the same tune.

Another way to use tuneblocks is as a musical jigsaw puzzle—you supply someone with the blocks for a tune, presented in some arbitrary order, and ask him to reconstruct the tune. Here, for instance, is a set of blocks:

```
TO B1
MUSIC [2 7] [4 4]
END

TO B2
MUSIC [2 4 5] [2 2 4]
END
```

```
TO B3
MUSIC [7 9 7 5] [3 1 2 2]
END

TO B4
MUSIC [4 5 7] [2 2 4]
END

TO B5
MUSIC [4 0] [2 6]
END
```

See if you can guess (and reconstruct) the tune from which these blocks were taken. Remember that a given block may be used in the tune more than once.

Here is a set of blocks for a much more difficult tuneblocks puzzle. See if you can construct a tune using these:

```
TO C1
MUSIC [5 4 2] [2 2 4]
END

TO C2
MUSIC [0 -2] [4 4]
END

TO C3
MUSIC [-2 -3 -5] [2 2 4]
END

TO C4
MUSIC [5 2] [4 4]
END

TO C5
MUSIC [-2 0 2] [2 2 4]
END

TO C6
MUSIC [-3 -2 0] [2 2 4]
END

TO C7
MUSIC [2 5 3 2] [2 2 2 2]
END
```

```
TO C8
MUSIC [7 6 7 9] [2 2 2 2]
END
```

Answers to the two tuneblocks puzzles appear at the end of this chapter.

In her work at MIT, Bamberger has used tuneblocks and other Logo-based music programs to teach music as well as to study people's intuitive notions about music and tonality and to track the development of musical intelligence. See her papers [2, 3, 4] for details.

9.1.3. Specifying Notes

The MUSIC command allows you to create simple melodies by specifying lists of pitches and durations. You can also control the loudness, articulation, and tempo of the notes with additional commands included in the Logo system.

REST

The REST command is used to insert rests (silences) into the music. REST takes a single number as input and inserts a rest of that duration into the music buffer.

STACCATO vs. LEGATO

STACCATO and LEGATO are used to control the amount of "dead time" that Logo inserts between successive notes. Logo normally plays notes with a legato articulation, that is, with only a small separation between successive notes. To change this, give the STACCATO command. This will cause all subsequent notes to be played detached. The LEGATO command restores the slurred articulation.[3]

Controlling Volume

The SETVOLUME command takes a numeric input that controls the volume of subsequent notes entered with the MUSIC command. Volume 0 is the softest and 15 is the loudest. Each unit from 0 to 15 represents a 2-decibel increase in volume.

Controlling Tempo

The SETTEMPO command controls the tempo at which music is played. SETTEMPO takes as input a number that determines the actual durations of subsequent notes as specified in the MUSIC command. When the tempo is set to T, a note of duration D will last for (60 / T) × D seconds. When you execute the command

[3]More precisely, when notes are played staccato, they sound for 5/60 of a second, with the remainder of the note's duration as dead time. When notes are played legato, they sound for all but the final 5/60 of a second.

```
SETTEMPO 100
```

each subsequent duration lasts 60/100 or 6/10 second.

```
SETTEMPO 300
```

reduces the duration to 60/300 or 2/10 second. When Logo is started, the default tempo setting is 300.

Here is an example of tempo change using SETTEMPO to produce an accelerating trill:

```
TO TRILL :TEMPO
IF :TEMPO > 3000 STOP
SETTEMPO :TEMPO
MUSIC [0 2] [1 1]
TRILL :TEMPO + 20
END

CM
TRILL 10
PM
```

MAJOR vs. CHROMATIC

These two commands (which take no inputs) control the meaning of the pitch numbers used with the MUSIC command. When you specify CHROMATIC, pitch numbers designate half steps: 0 is C, 1 is C-sharp, 2 is D, and so on. With MAJOR, successive numbers designate notes on the C major scale: 0 is C, 1 is D, 2 is E, and so on. For example, here are the MUSIC commands for the first phrase of *Frere Jacques* using CHROMATIC:

```
MUSIC [0 2 4 0] [4 4 4 4]
```

and using MAJOR:

```
MUSIC [0 1 2 0] [4 4 4 4]
```

With CHROMATIC, the range of defined pitch numbers is −15 through 24. With MAJOR, the range is −9 through 14. When Logo is started,

CHROMATIC is the default. Figure 9.3 shows the correspondence between Logo pitch numbers and conventional music notation for the MAJOR mode:

				A	B	C
				−9	−8	−7
D	E	F	G	A	B	C
−6	−5	−4	−3	−2	−1	0
D	E	F	G	A	B	C
1	2	3	4	5	6	7
D	E	F	G	A	B	C
8	9	10	11	12	13	14

Figure 9.3: Correspondence between Logo pitch numbers and conventional music notation for the MAJOR mode. Middle C is again assigned to pitch number 0.

MAJOR is useful for playing simple tunes in the key of C that require no sharps or flats. CHROMATIC is required for more complicated tunes or for tunes in other keys.

The NOTE Command

You can use NOTE as an alternative to MUSIC to enter notes in the music buffer. NOTE takes three numbers as inputs: a duration, a pitch, and a volume. For example,

```
NOTE 5 4 10
```

enters into the music buffer a note with duration 5, pitch 4, and volume 10. Unlike MUSIC, NOTE is used to enter a single note at a time rather than a list of notes. Also, the volume is specified explicitly for each note rather than taken from the default volume (as determined by SETVOLUME).

Comments on Specifying Notes

The commands listed in this section, such as SETTEMPO, SETVOLUME, and so on, have no effect on notes that are already in the music buffer. For instance, suppose you create some notes using MUSIC or NOTE, play the music with PM, then change the tempo using SETTEMPO and do PM again. The second PM will sound the same as the first, because SETTEMPO changes the tempo only for notes that will be added to the music buffer *after* the SETTEMPO command.

The music buffer can hold only a fixed number of notes. If you try to add notes when the buffer is full, Logo signals the error

OUT OF NOTES

After issuing the PM command, you can execute other commands while the music is playing. Note that pressing the BACK key does *not* stop the music. To stop music while it is playing you can reset the buffer with SETVOICE 0 or use the CM (clear music) procedure on page 128. Entering the editor by defining or editing a procedure (or a shape or tile) will also stop playing and clear the music buffer.

9.2. Multiple Voices

So far, we have been generating music for a single voice only. Logo allows you to play music using up to three voices plus a noise generator. To do this, simply use SETVOICE to designate the voice for subsequent notes specified with MUSIC or NOTE.

For example, you can make a three-part round of *Frere Jacques* using the basic procedure from page 129. A good tempo setting for this is 400.

```
TO FRERE
F1 F1
F2 F2
F3 F3
F4 F4
END
```

Voice 1 should do FRERE:

```
SETTEMPO 400
SETVOICE 1
FRERE
```

Voice 2 should rest for two repeats of F1 (a total of 32 duration units) and then do FRERE:

```
SETVOICE 2
REST 32

FRERE
```

Voice 3 should rest for 64 units and then do FRERE:

```
SETVOICE 3
REST 64
FRERE
```

Now when you type PM you will hear all three voices playing together.

Rhythm Accompaniment

In the Logo music system, voices 1, 2, and 3 play tones. Voice 4 is a *noise generator* that can be used to supply rhythm accompaniment. One easy way to do this is with the DRUM command. DRUM takes a list of durations (similar to MUSIC) and plays a corresponding "drumbeat." For example,

```
TO BOOMCHACHA
DRUM [4 2 2]
END
```

will play a "quarter-eighth-eighth" drum beat.

You can add this to the *Frere Jacques* round by using voice 4 for the drum. Then the entire round is

```
TO FRERE.JACQUES.ROUND
SETTEMPO 400
CM
SETVOICE 1
FRERE
SETVOICE 2
REST 32
FRERE
SETVOICE 3
REST 64
FRERE
SETVOICE 4
REPEAT 24 [BOOMCHACHA]
END

PM
```

You need not use DRUM only with voice 4. When set to one of the other voices, DRUM uses a short, low tone to make the beat. Conversely, you can specify "pitches" for voice 4 using MUSIC or NOTE. Depending on which "pitch" you choose, voice 4 will generate one of four different sounds.

9.3. Musical Accompaniment to Logo Procedures

We already mentioned that you can continue to execute Logo commands while music is playing. In this way you can provide musical accompaniment to other Logo procedures. Simply generate some music, start it playing with PM, and then start up your other procedures. You can also use the LOOPMUSIC command in place of PM. LOOPMUSIC is almost like PM except that it plays the music in the music buffer over and over. If you start music playing with LOOPMUSIC, the only way to stop it is by resetting the buffer with SETVOICE 0 or by entering edit mode.

Synchronizing Music to Logo Procedures

In addition to playing music and executing procedures at the same time, you can also *synchronize* music to Logo procedures; for example, you can synchronize music to the motion of sprites on the screen. This is done using the PLAYNOTE command. When you issue a PLAYNOTE command, Logo will play the next note from the music buffer, and then wait for the duration of the note.

To illustrate how to use PLAYNOTE for synchronizing music and graphics, let's return to the birds movie developed in Section 4.4. Recall that this was a movie in which a flock of birds moved across the screen, flapping their wings. The flapping was accomplished by changing the shapes of the sprites that represented the birds:

```
TO FLAP
CARRY:UPWING
WAIT 30
CARRY:DOWNWING
WAIT 30
FLAP
END
```

To synchronize the flapping to music, the only change you need make is to the FLAP procedure, replacing WAIT by PLAYNOTE:

```
TO FLAP
CARRY:UPWING
PLAYNOTE
CARRY:DOWNWING
PLAYNOTE
FLAP
END
```

Now enter some music in the buffer and run the entire movie as before. The result is that the birds beat their wings with each new note.

PLAYNOTE works with only one voice at a time. If there is more than one voice entered in the music buffer, PLAYNOTE will use the current voice specified by SETVOICE.

Answers to the Tuneblocks Puzzles

The first set of blocks can be assembled to form *London Bridge is Falling Down*. To play the tune, type

```
CM
B3 B4 B2 B4
B3 B4 B1 B5
PM
```

The second set of blocks is taken from a piece by Bela Bartok, based on a Slovakian folk tune (Bartok's *For Children*, Sz. 42):

```
CM
C3 C5 C8 C8
C3 C5 C8 C8
C2 C7 C6 C4
C2 C7 C6 C8
PM
```

This puzzle is more difficult than the preceding one not only because there are more blocks and the tune is less familiar, but also because the harmonic structure is not typical of the modalities found in western music. For this reason it is correspondingly more difficult for people accustomed to western music to assemble the blocks into patterns that "make sense." In solving either puzzle, you may be able to create a tune that is as interesting to you as the original. Either solving the puzzle or inventing your own tune opens the door to exciting musical explorations.

CHAPTER 10

Inputs, Outputs, and Recursion

One important difference between Logo and other common programming languages is that, in Logo, words and lists can be used as inputs and outputs to procedures. Therefore, when you program in Logo, you can work in terms of operations that act on entire words and lists, rather than only on individual numbers and characters. Consider the DOUBLE.LIST procedure that was introduced on page 84:

```
TO DOUBLE.LIST :X
OUTPUT SENTENCE :X :X
END
```

```
PRINT DOUBLE.LIST [DO RE MI]
DO RE MI DO RE MI
```

The importance of making this procedure OUTPUT its result is not merely so that you can PRINT the result, but so that you can use the result as an input to another procedure that can perform further operations. For instance, if you have an operation REVERSE that reverses a list (as we shall discuss in Section 10.1 below), then you can produce the reverse of the double of a list X by

```
REVERSE DOUBLE.LIST :X
```

More generally, you can construct complex operations on words and lists as successions of procedures, each of which performs a simple operation and passes the result to the next procedure. To obtain an operation that removes the last word from a list, reverses what is left, and doubles the result, you can write:

```
DOUBLE.LIST (REVERSE (BUTLAST :X))
```

as in the command

```
PRINT DOUBLE.LIST (REVERSE (BUTLAST [A B C D]))
C B A C B A
```

which produces the chain of operations shown in Figure 10.1. Building up complex operations by combining simpler operations is common programming practice in working with numbers. For example, it is natural to

think of computing $(x - 1)^2 + 1$ in terms of the simpler operations of subtracting 1 from a number, squaring the result, and adding 1. Logo enables you to use the same kind of strategy in dealing with words and lists.

The ability to construct complex operations as combinations of simpler ones is particularly powerful when combined with another problem-solving strategy: One can often solve a problem by first solving a simpler problem of the same sort and then making a simple modification to the answer. For example, suppose you want to write a procedure that counts the number of words in a list. Imagine that you already know how many words are in the BUTFIRST of the list. Then you could solve your original problem by simply taking the number of words in the BUTFIRST and adding 1.

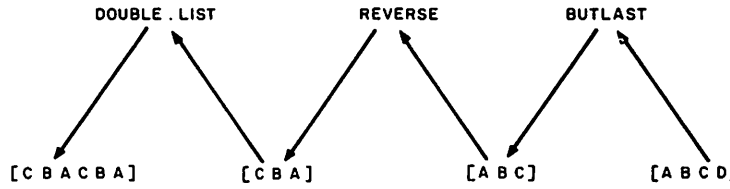


Figure 10.1: Chain of inputs and outputs in a sequence of list operations.

Recursive procedures, in general, are the computational analogues of strategies that attack problems by reducing them to simpler problems of the same sort. Given the ability of Logo procedures to manipulate words and lists, this implies that many useful word and list operations can be implemented as surprisingly simple recursive procedures. This chapter examines a few of them. We consider first a number of procedures involved with reversing words and lists. Then we discuss operations that select words from lists and test whether a word is a member of a list. Finally, we show how the problem of converting numbers from one base to another can be solved by a simple recursive strategy.

10.1. REVERSE

TI Logo II includes a built-in operation called REVERSE that reverses words or lists.[1] If the input to REVERSE is a word, then REVERSE returns the word with the characters reversed:

```
PRINT REVERSE "STRESSED
DESSERTS
```

[1] For the first release of TI Logo, use one of the procedures, REVLIST or REVWORD, given in Sections 10.1.1 and 10.1.2, instead of REVERSE.

```
PRINT REVERSE "RUMPLESTILTSKIN
NIKSTLITSELPMUR

PRINT REVERSE REVERSE "RUMPLESTILTSKIN
RUMPLESTILTSKIN
```

If the input to REVERSE is a list, then REVERSE returns a list of the elements in reverse order:

```
PRINT REVERSE [I AM WHAT I AM]
AM I WHAT AM I

PRINT REVERSE [HELLO]
HELLO
```

Even though REVERSE is included as a primitive operation in TI Logo II, we'll show how you can write such a procedure, since reversing is a good illustration of recursive programming. In order not to conflict with the built-in REVERSE, we'll write separate procedures for reversing words and lists, called REVWORD and REVLIST.

10.1.1. Reversing Words

Consider the problem of writing a procedure REVWORD that reverses a word:

```
PRINT REVWORD "HELLO
OLLEH
```

Logo's LAST and BUTLAST operations, which encourage thinking about a word in terms of its last character and the rest of the word, suggest a recursive strategy for implementing the REVWORD procedure. It is based on the following idea: suppose you are given a word, say BIRD, and you are asked to reverse it. Now imagine that you have somehow managed to generate the reverse of all but the last character of the word—RIB. Then all you have to do to reverse the original word is to take the last character, D, and place it at the front of what you already have—DRIB. This reduces the problem of reversing a word to the problem of reversing a shorter word, namely, the BUTLAST of the word. That problem reduces in turn to reversing a still shorter word, namely, the BUTLAST of the BUTLAST, and so on, with shorter and shorter words. You can diagram this process as follows:

REVWORD "BIRD is D ⟷ RIB

or the last character of BIRD added in front of RIB. But

RIB is REVWORD "BIR which is R ⟷ IB

or the last character of BIR added in front of IB. But

IB is REVWORD "BI which is I ⟷ B

or the last character of BI added in front of B. Now put all these together:

```
REVWORD "BIRD = D ⟷ (REVWORD "BIR )
              = D ⟷ R ⟷ (REVWORD "BI )
              = D ⟷ R ⟷ I ⟷ (REVWORD "B )
              = D ⟷ R ⟷ I ⟷ B
```

This strategy, reducing the problem of reversing a word to the problem of reversing BUTLAST of the word, leads to the following recursive procedure:

```
TO REVWORD :X
OUTPUT WORD (LAST :X) (REVWORD BUTLAST :X)
END
```

However, if you execute this procedure, it will not work. Instead Logo runs out of space. The problem is that there is no *stop rule*. Nothing tells REVWORD to stop taking LASTs and BUTLASTs of its input, and the procedure runs until Logo runs out of storage. At some point, REVWORD should simply output an answer directly without reducing the problem to one of reversing a still shorter word. For example, if the word to be reversed is a single character, then REVWORD of the word is the word itself, so you can add to REVWORD the stop rule:

```
IF FIRST :X = :X THEN OUTPUT :X
```

where FIRST :X being equal to :X signals that :X consists of a single character. So here is the complete procedure:

```
TO REVWORD :X
IF FIRST :X = :X OUTPUT :X
OUTPUT WORD (LAST :X) (REVWORD BUTLAST :X)
END
```

10.1.2. Reversing Lists

Similar reasoning can be applied to produce a procedure REVLIST that takes a list as input and returns the list of words in reverse order, as in

```
PRINT REVLIST [OH SAY CAN YOU SEE]
SEE YOU CAN SAY OH
```

As before, the problem reduces to combining the LAST of the input list with REVLIST of the BUTLAST; however, since you will be combining lists rather than words, you should use SENTENCE rather than WORD to form the combination. The stop rule checks for the list being reduced to the empty list, in which case the procedure returns the empty list.

```
TO REVLIST :X
IF :X = [ ] THEN OUTPUT [ ]
OUTPUT SENTENCE (LAST :X) (REVLIST BUTLAST :X)
END
```

You can combine REVWORD and REVLIST to obtain a procedure REVALL that takes a list as input and returns a list of the words in reverse order, with each word reversed, as well:

```
PRINT REVALL [OH SAY CAN YOU SEE]
EES UOY NAC YAS HO
```

All you need to do to implement REVALL is to modify REVLIST so that it REVWORDs the LAST word of its input before combining that with the result of reversing the BUTLAST:

```
TO REVALL :X
IF :X = [ ] THEN OUTPUT [ ]
OUTPUT SENTENCE (REVWORD LAST :X) (REVALL BUTLAST :X)
END
```

10.1.3. Designing Recursive Procedures

The reasoning that led to these procedures is typical of most recursive procedures that involve words and lists:

- There is a *reduction step* that reduces the problem to a similar problem on a shorter word or list (usually the BUTFIRST or BUTLAST of the input).
- There is a *stop rule* that checks for some simple case (usually the input being reduced to a single element, or to the empty word or the empty list).[2]

[2]Notice that in the actual procedure, the stop rule is written before the reduction step. But when you formulate a recursive solution, you most likely discover the reduction step first and then design an appropriate stop rule.

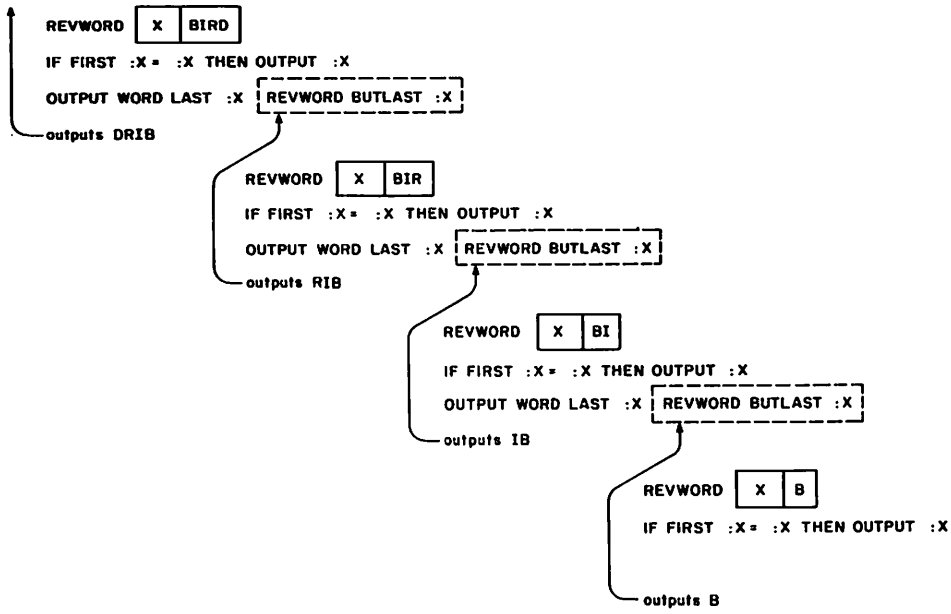


Figure 10.2: Procedure calls in executing REVWORD "BIRD.

Note the use of recursion and the fact that each procedure must explicitly OUTPUT its result to the procedure that calls it, as noted on page 79. Figure 10.2 shows the pattern of inputs and outputs that results from executing

```
REVWORD "BIRD
```

10.2. Recursive Procedures that Manipulate Lists

Logo's list operations FIRST, LAST, BUTFIRST, and BUTLAST are the basic ways to reduce lists to simpler lists. List operations can often be implemented by means of recursive strategies that reduce the problem of performing some operation on a list to the problem of performing a similar operation on the BUTFIRST (or BUTLAST) of the list. This section presents two operations that can be implemented in this way.

10.2.1. The PICK Procedure

One of the most useful operations to have in working with lists is the ability to select an item from a list. Consider the problem of writing a procedure PICK that takes a number and a list as inputs and outputs the designated item from the list: if the number is 1, PICK outputs the first item in the list; if the number is 2, PICK outputs the second item in the list; and so on.

There is a recursive strategy for computing PICK in terms of the operations FIRST and BUTFIRST. You can reduce the problem of picking an item from a list to the problem of picking an item from the BUTFIRST of the list: the *n*th item of a list is the same at the (*n* − 1)st item of the BUTFIRST of the list. The recursive plan is:

- *Reduction Step*: to PICK the *n*th item from a list, PICK the (*n* − 1)st item from the BUTFIRST of the list.
- *Stop Rule*: if *n* = 1, then output the FIRST item in the list.

This strategy can be expressed as the following Logo procedure:

```
TO PICK :N :X
IF :N = 1 OUTPUT FIRST :X
OUTPUT PICK (:N - 1) (BUTFIRST :X)
END
```

Figure 10.3 shows the chain of procedure calls and the inputs and outputs in executing:

```
PICK 3 [A B C D]
```

where picking the 3rd item of [A B C D] reduces to the picking the 2nd item of [B C D] which reduces to picking the lst item of [C D], which is C.[3]

By combining PICK with RAND (page 80) you get a useful operation that selects an item at random from a list of possibilities.

```
TO PICKRANDOM :X
OUTPUT PICK (1 + RAND (LENGTH :X)) :X
END
```

This procedure uses the LENGTH operation that is included in TI Logo II. LENGTH takes a list as input and returns the number of items in the list.[4]

---

[3] If you call PICK with N larger than the length of the list, then the procedure will return the empty list. For example, trying to pick the 5th item of [A B C D] reduces to the 4th item of [B C D], the 3rd item of [C D], the 2nd item of [D], and finally PICK is called with N equal to 1 and X equal to the empty list. At this point PICK tries to compute FIRST of X. In the first release of TI Logo FIRST of the empty list returns the empty list.

[4] LENGTH can also take a word as input, in which case it returns the number of characters in the word. LENGTH is not included in the first release of TI Logo, but it can be implemented as a recursive Logo procedure (see note in Section 7.2).

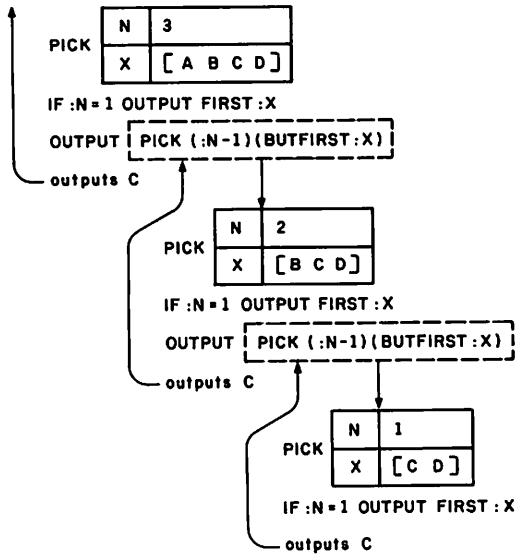


Figure 10.3: Procedure calls in executing PICK 3 [A B C D].

Observe the inputs to PICK and RAND: if the length of the list is n, then RAND (LENGTH :L) returns a number selected at random between 0 and $n - 1$. You should add 1 to this to produce a random number between 1 and n, which becomes the input to PICK.

10.2.2. The MEMBER? Predicate

The MEMBER? predicate takes a word and a list as inputs and checks whether the word is a member of the list, outputting TRUE or FALSE accordingly. The recursive strategy here is that it is easy to check if the desired word is the FIRST item in the list. If it is, then MEMBER? should output TRUE. If not, you check to see if the word is in the BUTFIRST of the list, and so on. If the list ever becomes empty, you have run out of elements to check the word against, so MEMBER? should output FALSE. The resulting procedure is

```
TO MEMBER? :WORD :LIST
IF :LIST.= [] OUTPUT "FALSE
IF :WORD = (FIRST :LIST) OUTPUT "TRUE
OUTPUT MEMBER? :WORD (BUTFIRST :LIST)
END
```

Converting to Pig Latin

As an example of using MEMBER? and recursion, you can write a program that converts a sentence to pig latin. For each word in the sentence, you must move the leading consonants to the end of the word and add "ay" as in

Isthay entencesay isay inay igpay atinlay.

Since you need to strip off consonants, it is useful to have a predicate that checks whether a word begins with a vowel. That's easily done:

```
TO BEGINS.WITH.VOWEL? :W
OUTPUT MEMBER? (FIRST :W) [A E I O U]
END
```

Notice that this outputs TRUE or FALSE because MEMBER? outputs TRUE or FALSE.

Here's a program that converts a single word to pig latin:

```
TO PIG :W
TEST BEGINS.WITH.VOWEL? :W
IFT OUTPUT WORD :W "AY
IFF OUTPUT PIG WORD (BUTFIRST :W) (FIRST :W)
END
```

The cleverness in PIG is the recursive call that ensures that PIG will keep stripping letters off the front of the word until it reaches a vowel. To better understand this point, you should draw a diagram that gives the sequence of recursive calls in computing

```
PRINT PIG "STRING
INGSTRAY
```

Now, if you work word by word, you can convert an entire sentence. The trick is to think recursively again:

```
TO PIGL :S
IF :S = [] OUTPUT []
OUTPUT SENTENCE (PIG FIRST :S) (PIGL BUTFIRST :S)
END
```

```
PRINT PIGL [THIS IS ANOTHER RECURSIVE PROCEDURE]
ISTHAY ISAY ANOTHERAY ECURSIVERAY OCEDUREPRAY
```

The strategy used in PIGL is a standard way to "do something to every item in a list." The idea is to reason as follows. Suppose you have already converted the words in the BUTFIRST of the list. Then you need only

SENTENCE this with the result of converting the first word in the list, and you are done. In this way, the problem of converting the entire list reduces to converting BUTFIRST of the list, which reduces to BUTFIRST of *that* list, and so on, and so on. Finally the problem is reduced to that of converting the empty list, for which the answer is empty.

10.3. Radix Conversion

As a final example of a problem that seems difficult but has a simple recursive solution, we consider the problem of converting an integer written in base 10 notation to some other base, say, base 8. For instance, we would like to find that 65 base 10 is written as 101 in base 8, 100 base 10 is 144 base 8, 1000 base 10 is 1750 base 8, and so on.

There is a clever recursive strategy for solving this problem. Suppose that *n* is some integer and that you want to find the string of digits that represents *n* in base 8. Think about what such a representation means. For example, to say that 100 base 10 is written as 144 base 8 means that

$100 = 1 \times 8 \times 8 + 4 \times 8 + 4 =$ 144 base 8

The key insight is that it is easy to find the *last* digit of the string of digits that represents *n*: this is just the remainder when *n* is divided by 8:

REMAINDER 100 8 is 4

Now suppose you take the string of digits that represents *n* and strip off the last digit. In terms of base 8 representation, that corresponds to shifting everything one place to the right and dropping the last digit. But this corresponds precisely to dividing the number by 8 and dropping the remainder. That is to say, if you take the string of digits that represents *n* in base 8 and leave off the last digit, what you are left with is the string of digits that represents the integer quotient of *n* by 8, written in base 8:

QUOTIENT 100 8 is 12, and 12 represented in base 8 is 14

So now you have a simple description of the string of digits that represents *n* in base 8

- The LAST digit is the remainder of *n* by 8.
- The BUTLAST of the string is the base 8 representation of integer quotient of *n* by 8.

So the problem of representing *n* in base 8 reduces to representing the quotient of *n* by 8 in base 8, which reduces further, and so on. The reductions stop when you reach a quotient that is less than 8, which is represented in base 8 as a single digit.

Here is how to generate *n* in base 8:

- If $n<8$, the result is the digit for *n*.[5]
- Otherwise
  - ★Find the digits that represent the quotient of *n* by 8, and
  - ★ Append to these the remainder of *n* divided by 8.

Using the Logo WORD operation to glue numbers together, this strategy translates into the procedure:

```
TO BASE8 :N
IF :N < 8 OUTPUT DIGIT :N
OUTPUT WORD BASE8 (:N/ 8)
                DIGIT REMAINDER :N 8
END
```

The DIGIT procedure is used to take a single-digit number and convert it to the corresponding character. If you start with a list of these characters:[6]

```
MAKE "DIGITLIST (SE "0 "1 "2 "3 "4
                    "5 "6 "7 "8 "9 )
```

then you can implement DIGIT as:

```
TO DIGIT :N
OUTPUT PICK :N + 1 :DIGITLIST
END
```

(Note the 1 added to N: the *first* digit in the list, for example, is the digit 0.)

Of course, there is nothing special about base 8. You can convert to any base less than 10 in the same way:

```
TO BASE :N :B
IF :N < :B OUTPUT DIGIT :N
OUTPUT WORD BASE (:N/:B) :B
                DIGIT REMAINDER :N :B
END
```

---

[5]Remember that in TI Logo, you must distinguish between a number, say 5, and a character, say "5. For doing arithmetic, you use numbers, and for doing word operations, you use characters. In this case we are assembling the converted number using word operations, so the digit used must be the *character* "5 rather than the number 5.

[6]This is the "hexadecimal" notation commonly used for specifying computer memory addresses.

For example

```
PRINT BASE 1000 2
1111101000
```

For bases larger than 10, you can use the same strategy, except that you will need "digits" representing the single-digit numbers larger than 10. For instance, in base 16 you can represent 10 by the letter A and 11 by the letter B, and so on.[5] All you need to do is add more items to DIGITLIST:

```
MAKE "DIGITLIST (SE "0 "1 "2 "3 "4
                    "5 "6 "7 "8 "9
                    "A "B "C "D "E "F )

PRINT BASE 20000 16
4E20

PRINT BASE 20000 12
B6A8
```

CHAPTER 11

Advanced Use of Lists

We've seen how words can be grouped together into Logo lists. But lists in Logo can be used for more than just collecting words. For example, the random-sentence generator of Section 7.2 picked its nouns from a list:

```
MAKE "NOUNS [DOGS CATS CHILDREN TIGERS]
```

Suppose, however, you want to make sentences using "nouns" that aren't single words. For example, you may want to make sentences about dogs, cats, children, tigers, and pack rats. You can't do this by adding the two words PACK RATS to the above lists as in

```
MAKE "NOUNS [DOGS CATS CHILDREN TIGERS PACK RATS]
```

because making a sentence whose nouns are words picked at random from this list of six items would give results including things like

```
PACK LAUGH
RATS RUN
.
.
.
```

What you need to do is to take the two words PACK RATS and group these together as a *single* item within the list of nouns. You can do this in Logo by

```
MAKE "NOUNS [DOGS CATS CHILDREN TIGERS [PACK RATS]]
```

What you have now is a list of five items. The first four items in the list are words: DOGS, CATS, CHILDREN, TIGERS. The fifth item in the list is *itself* a list [PACK RATS] consisting of two words, PACK and RATS. When you pick items from the list NOUNS, you may get a single word like DOGS, or you may get the two-word list [PACK RATS]. This new value of NOUNS gives the desired results in the sentence generating program of Section 7.2:

```
DOGS BITE
PACK RATS LAUGH
.
.
.
```

The general point here is that in Logo *the items in a list can be, not only words, but also other lists.*

11.1. Hierarchical Structures

If you think of a list of words as a simple list (or one-level list), then the NOUNS list above can be considered to be a two-level list, that is, a list with an element that is itself a list. But there is no reason to stop there. In general, you can have lists whose items are themselves lists whose items are lists, and so on. This general notion of a list in Logo provides lots of power and flexibility in dealing with complex structures. For example, Figure 11.1

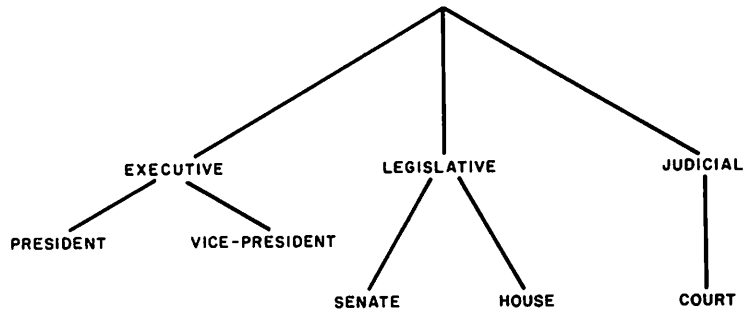


Figure 11.1: Hierarchical Organization of U.S. Government.

shows a *tree structure* that represents part of the organization of the U.S. government.

From our point of view, the important thing about this structure is that it is a *hierarchy*; that is, it consists of parts that themselves consist of parts, and so on. You can represent the tree structure in Figure 11.1 as the Logo list

```
[ [EXECUTIVE [PRESIDENT VICE-PRESIDENT]]
  [LEGISLATIVE [SENATE HOUSE]]
  [JUDICIAL [COURT]] ]
```

This is a list of three items.[1] The first item, which is the list

```
[EXECUTIVE [PRESIDENT VICE-PRESIDENT]]
```

is itself a list of two items, of which the first is the word EXECUTIVE and the second is a list of two words, and so on.

Logo's use of lists is adapted from the programming language Lisp, which was developed for research in artificial intelligence. Lists have proved to be indispensable in programs that deal with symbol manipulation and complex

[1] Note how the list is printed, lining up its three elements in order to make its structure more readable.

data structures, and their presence in Logo and Lisp is largely responsible for the fact that programming in these languages is very different from working in languages like BASIC and Fortran. In those languages, complex data structures must be encoded in terms of numbers, character strings, and arrays. Lists, however, allow many kinds of complex hierarchical structures to be represented directly, and therefore lists play a major role in computer applications dealing with complex data structures. In particular, they are the workhorse of most programs that are heavily involved with symbolic expressions, rather than just numerical data. The projects in Section 11.3 illustrate how lists are used in this way. However, this hardly scratches the surface of what can be done. The book by Winston and Horn [19] provides many examples of the uses of lists in symbol manipulation in the context of the language Lisp.

11.1.1. List Operations

We've already seen how to use the Logo operations FIRST, LAST, BUTFIRST, BUTLAST, and SENTENCE for working with "simple" lists of words. These same operations extend to work with complex lists, as well. For example, suppose you create a complicated list:

```
MAKE "TRY [[A B C] D [E F]]
```

TRY is a list of three items, the list [A B C], the word D, and the list [E F].

```
PRINT :TRY
[A B C] D [E F]
```

Note how TRY is printed. Logo always prints lists without the outermost pair of brackets.

The operations FIRST and LAST output, as usual, the first and last items in a list. In a complex list these items may themselves be lists:

```
PRINT FIRST :TRY
A B C
PRINT LAST : TRY
E F
```

BUTFIRST outputs the list consisting of all elements by the first, and BUTLAST outputs the list consisting of all elements but the last:

```
PRINT BUTFIRST :TRY
D [E F]
PRINT BUTLAST :TRY
[A B C] D
```

Keep in mind that the operations output, in general, new lists, to which you can apply further operations. For example,

```
FIRST FIRST :TRY
```

is the first item of the first item of TRY, which is the first item of [A B C], which is A.

```
FIRST LAST :TRY
```

is the first item of the last item of TRY, which is the first item of [E F], which is E.

```
FIRST BUTFIRST :TRY
```

is the first item of the butfirst of TRY, which is the first item of [D [E F]], which is D. (In general, FIRST of BUTFIRST of any list is the second item of the list.)

```
FIRST BUTFIRST LAST :TRY
```

is the FIRST of the BUTFIRST of the LAST of TRY, which is the FIRST of the BUTFIRST of [E F], which is F.

The four operations FIRST, LAST, BUTFIRST, and BUTLAST are used for extracting pieces of lists. To combine lists into more complex lists, we have the Logo operation FPUT. FPUT takes two inputs, of which the second must be a list. It puts its first input at the beginning of its second input; that is, it outputs a list whose FIRST is the first input and whose BUTFIRST is the second input:

```
PRINT FPUT "A [D E F]
A D E F
PRINT FPUT [A] [D E F]
[A] D E F
PRINT FPUT [A B C] [D E F]
[A B C] D E F
```

LPUT is similar to FPUT, except that it installs its first input as the *last* item in the list:

```
PRINT LPUT "A [D E F]
D E F A
PRINT LPUT [A] [D E F]
D E F [A]
PRINT LPUT [A B C] [D E F]
D E F [A B C]
```

The Logo operation SENTENCE, which we previously used to combine words into lists, can also be used with more complex lists. if SENTENCE is given a number of lists as inputs, it combines all of the elements of the lists into a single list:

```
PRINT SENTENCE [A [B C]] [D E F]
A [B C] D E F
```

This description of SENTENCE makes sense only when all of the inputs to SENTENCE are themselves lists. In order to make this consistent with our previous definition of SENTENCE for combining words into lists we extend the definition as follows: if one of the inputs to SENTENCE is a word, then you replace that word by the one-item list containing that word, and then apply the definition of SENTENCE given above. For example:

```
(SENTENCE "A "B "C )
```

gives the same result as

```
(SENTENCE [A] [B] [C])
```

which is the list [A B C].

```
SENTENCE "A [B [C D]]
```

gives the same result as

```
SENTENCE [A] [B [C D]]
```

which is the list [A B [C D]]. In general, SENTENCE :X :Y gives the same result as FPUT :X :Y if :X is a word and :Y is a list.[2]

Using FPUT, you can construct a useful operation called LIST that takes two inputs and combines them into a list of two items. LIST works by first combining its second input with the empty list using FPUT. This creates a one element list whose only element is the original second input. Next, the first input of LIST is combined with this one-element list to produce a two-element list.

[2]If you are interested only in combining words into lists to be printed (as in most elementary Logo programs), then SENTENCE is the only operation you need for constructing lists. However, when you are interested in using lists as hierarchical data structures, you need the finer control provided by FPUT and LPUT. For example, it is always true that :X if the first item of FPUT :X :Y. But this is not the case with SENTENCE. For instance, if :X is [A B C] and :Y is [D E], then the first item of SENTENCE :X :Y is the word A.

```
TO LIST :A :B
OUTPUT FPUT :A (FPUT :B [ ])
END

PRINT LIST [A B] [C D]
[A B] [C D]
```

11.1.2. Example: Association Lists

One particularly simple form of list is a list of pairs, which can be used to represent simple tables in which values are associated to things:

```
MAKE "TABLE1 [[COLOR PURPLE]
              [SIZE HUGE]
              [WEIGHT [1 TON]]]
```

Such a list of pairs is called an *association list*. The first item in each pair is referred to as the *key*, and second item is the corresponding *value*. The most important function for operating on tables represented as association lists is LOOKUP, which outputs the value corresponding to a given key:

```
PRINT LOOKUP "SIZE :TABLE1
HUGE
```

LOOKUP is implemented by means of an auxiliary function called ENTRY, which outputs the pair in which the key occurs, or outputs the empty list if there is no such pair in the table. LOOKUP then outputs the second item in the ENTRY, or signals an error if the key was not found. ENTRY is implemented by scanning down the list in the usual fashion:[3]

```
TO ENTRY :KEY :TABLE
IF :TABLE = [ ] OUTPUT [ ]
IF :KEY = (FIRST FIRST :TABLE) OUTPUT (FIRST :TABLE)
OUTPUT ENTRY :KEY (BUTFIRST :TABLE)
END
```

LOOKUP is implemented as

```
TO LOOKUP :KEY :TABLE
MAKE "PAIR ENTRY :KEY :TABLE
IF :PAIR = [ ] PRINT [ERROR: KEY NOT IN TABLE]
OUTPUT LAST :PAIR
END
```

[3]This is very similar to the MEMBER? procedure on page 143.

Another use for association lists that arises in symbol manipulation is for substituting values from a table. The following SUBST procedure takes a list and a table as inputs. For each item in the list that is a key in the table, it replaces the key by the corresponding value. For example, with TABLE1 as above, you would have:

```
PRINT SUBST [HE IS COLOR AND WEIGHS WEIGHT] :TABLE1
HE IS PURPLE AND WEIGHS [1 TON]
```

To define SUBST, we'll begin by writing a procedure SUBST.ITEM that takes an item and a table as input. If the item is a key in the table, then SUBST.ITEM outputs the associated value. Otherwise it outputs the original item. Notice that this is almost the same as LOOKUP except that it returns the original item instead of signaling an error if the item is not in the table.

```
TO SUBST.ITEM :ITEM :TABLE
MAKE "SUBST.PAIR (ENTRY :ITEM :TABLE)
IF :SUBST.PAIR = [] OUTPUT :ITEM
OUTPUT LAST :SUBST.PAIR
END
```

The SUBST procedure itself is implemented by performing SUBST.ITEM on each item in the list, and outputting the list of the results:

```
TO SUBST :LIST :TABLE
IF :LIST = [] OUTPUT []
OUTPUT FPUT (SUBST.ITEM (FIRST :LIST) :TABLE)
            (SUBST (BUTFIRST :LIST) :TABLE)
END
```

Properties

One way to think of an association list is as a collection of the attributes, or "properties," of some object:

```
MAKE "SUPERGRAPE
     [[COLOR PURPLE] [SIZE HUGE] [WEIGHT [1 TON]]]
```

These attributes can be recovered by using the LOOKUP procedure given above. More abstractly, we can forget about the list of pairs, and imagine that we have a procedure PUTPROP, which associates a given property value to a given symbol. For example,

```
PUTPROP "SUPER.GRAPE "COLOR "PURPLE
```

would associate to the symbol SUPER.GRAPE a COLOR property whose value is PURPLE. A corresponding procedure GETPROP would be used to retrieve a property value, so that, for example,

```
GETPROP "SUPER.GRAPE "COLOR
```

would return PURPLE. A typical program that uses properties to manage information might contain a line such as

```
PRINT (SENTENCE [THE COLOR OF]
                :ITEM
                [IS]
                (GETPROP :ITEM   "COLOR ))
```

PUTPROP and GETPROP are readily implemented in terms of association lists, but in some applications, it is better to use other methods for representing properties. In particular, if there are many attributes in a table, performing a LOOKUP will be slow, due to the need to scan a long list. An alternative way to implement properties in Logo, which allows fast access to large tables, is as follows. To associate a property to a symbol, you combine the symbol, the property, and a separator character (e.g., #) to form a new word. Then assign to this word the designated property value. For example, to perform the association

```
PUTPROP "SUPER.GRAPE "COLOR "PURPLE
```

you execute the MAKE command:

```
MAKE "SUPER.GRAPE#COLOR "PURPLE
```

In general, PUTPROP is implemented using this scheme as:

```
TO PUTPROP :SYMBOL :PROPERTY :VALUE
MAKE WORD :SYMBOL
          (WORD "#   :PROPERTY)
     :VALUE
END
```

and the corresponding GETPROP procedure is:

```
TO GETPROP :SYMBOL :PROPERTY
OUTPUT THING WORD :SYMBOL
                  WORD "#   :PROPERTY
END
```

Note that these procedures rely on Logo's ability to assign a value to a symbol that is the result of some computation, rather than typed in literally as is almost always the case with MAKE. Compare the "tricky use of MAKE" shown on page 86.

11.2. Programs as Data

One important kind of hierarchical structure that arises in programming is the structure of a program itself. A Logo procedure can be thought of as a list of lines each of which is a list of words. Using Logo lists, you can write *programs that manipulate other programs*. The basic Logo primitives that enable you to do this are RUN, which executes a list as a Logo command line; DEFINE, which constructs a procedure from list data: and TEXT, which outputs the representation of a procedure. This section explains how these operations work in the context of an extended example—increasing the capabilities of the simple INSTANT program that was introduced in Section 8.2.1.

11.2.1. The RUN Command

The Logo command RUN takes a Logo list as input and executes the list as if the list were a command line typed at the keyboard. For example:

```
RUN [PRINT [HELLO THERE]
HELLO THERE
RUN LIST "PRINT [HELLO THERE]
HELLO THERE
MAKE "COMMAND "PRINT
MAKE "INPUT [HELLO THERE]
RUN LIST :COMMAND :INPUT
HELLO THERE
```

Example: Extending the INSTANT Program

Another situation in which RUN is useful is where you want to build up a list of commands to be executed later. As an example, consider the INSTANT program of Section 8.2.1:

```
TO INSTANT
COMMAND
INSTANT
END
```

```
TO COMMAND
MAKE "COM READCHAR
IF :COM = "F FORWARD 10
IF :COM = "R RIGHT 30
IF :COM = "L LEFT 30
IF :COM = "C CLEARSCREEN
END
```

Suppose you want to add an "undo" feature to the system. That is, typing F, L, and R at the keyboard will cause the turtle to move forward, left, and right as before. In addition, typing U will cause the turtle to undo its previous move.

You can implement the undo operation as follows. As the user of the INSTANT system gives commands, the INSTANT program will not only move the turtle, but will also *remember* the turtle motions that were done by saving them in a list. Then, when the user wishes to undo the last command, INSTANT will clear the screen, remove the last command from the list, and reprocess the remaining commands.[4]

To implement this strategy let's assume you store the turtle commands in a list called HISTORY. For example, if the user types F and then R, HISTORY will be

```
[[FORWARD 10][RIGHT 30]]
```

Notice that HISTORY is a list of lists, in which each entry is the Logo command that should be run to cause the appropriate turtle motion.

The main operation needed now is to take a turtle command and not only do it, but also add it to the HISTORY list. This can be accomplished by

```
TO RUN.AND.RECORD :ACTION
RUN :ACTION
MAKE "HISTORY (LPUT :ACTION :HISTORY)
END
```

LPUT is used to add the new command as the *last* item in HISTORY.

Now change the COMMAND procedure to RUN.AND.RECORD the appropriate response to each key:

[4]There are, of course, many other ways to implement the undo operation. One advantage of the way chosen here is that it extends nicely to allowing the user of the INSTANT system to define programs, as we shall see in Section 11.2.2.

```
TO COMMAND
MAKE "COM READCHAR
IF :COM = "F RUN.AND.RECORD [FORWARD 10]
IF :COM = "R RUN.AND.RECORD [RIGHT 30]
IF :COM = "L RUN.AND.RECORD [LEFT 30]
IF :COM = "C RUN.AND.RECORD [CLEARSCREEN]
END
```

Now, to undo the last command, you remove the last item from HISTORY, clear the screen, and run the rest of the commands:

```
TO UNDO
IF :HISTORY = [] STOP
MAKE "HISTORY BUTLAST :HISTORY
CLEARSCREEN
RUN.ALL :HISTORY
END
```

Note the first line of UNDO, which says that if the HISTORY list is empty, there is nothing to undo. Also note that with this implementation, repeatedly executing UNDO keeps removing more and more items from HISTORY, starting with the last one, the one before that, and so on.

The subprocedure RUN.ALL takes a list of commands as input and runs all the commands in the list in sequence. (Each command in the list must itself be a list.) RUN.ALL uses a recursive strategy. It RUNs the first command in the list and then processes the BUTFIRST of the list.

```
TO RUN.ALL :COMMANDS
IF :COMMANDS = [] STOP
RUN FIRST :COMMANDS
RUN.ALL (BUTFIRST :COMMANDS)
END
```

Now all you need to do is add a line to the COMMAND procedure so that pressing U causes an UNDO operation:

```
TO COMMAND
MAKE "COM READCHAR
IF :COM = "F RUN.AND.RECORD [FORWARD 10]
IF :COM = "R RUN.AND.RECORD [RIGHT 30]
IF :COM = "L RUN.AND.RECORD [LEFT 30]
IF :COM = "C SETUP
IF :COM = "U UNDO
END
```

The complete INSTANT program now simply clears the screen and repeatedly calls COMMAND. You also need to initialize HISTORY to be empty:

```
TO SETUP
MAKE "HISTORY [ ]
TELL TURTLE
CLEARSCREEN
INSTANT
END

TO INSTANT
COMMAND
INSTANT
END
```

SETUP has been added to COMMAND in place of RUN.AND.RECORD [CLEARSCREEN]; now when you clear the screen by typing C, HISTORY is reinitialized as well.

11.2.2. The DEFINE Command

In addition to using Logo list operations to generate individual command lines that can be RUN, you can also write procedures that define other procedures. This is done with the DEFINE command. DEFINE (short form DE) takes two inputs. This first is the name of the procedure to be defined. The second input is a list of lists organized as follows. The first sublist gives the inputs to the new procedure, and there is one additional sublist for each procedure line. For example,

```
DEFINE "TRY [[:X :Y][PRINT :X][PRINT :Y]]
PO TRY
TO TRY :X :Y
PRINT :X
PRINT :Y
END

DEFINE "GREET [[ ] [PRINT [HELLO]]]
PO GREET

TO GREET
PRINT [HELLO]
END
```

Observe that if the procedure is to have no inputs (as in GREET above), the DEFINE list must include an initial empty list for the input specification. Note also that there is no END included in the list of procedure lines.

Example: Another Extension to INSTANT

Most of the time, of course, you use TO rather than DEFINE to create Logo procedures. DEFINE is reserved for those situations in which you want procedure definition to happen within a program. As an example of this, we'll consider another extension to the INSTANT system of Section 11.2.1. This time, we'll allow the user of INSTANT to name drawings and to recall them by name. For example, we may use the letter S for saving drawings. Typing S (for "save") will cause the program to ask the user for a name for the drawing. Later on, the user can ask for a previous drawing to be reshown, say by typing P for "picture." More than one drawing can be saved at once, each with its own name.

You can implement this by having the INSTANT system save a drawing by *defining the drawing as a procedure*, using the name chosen by the user. The list of lines in the procedure is precisely the HISTORY list that you have been using to keep track of what is on the screen. Here is the procedure that implements this "learning" process:

```
TO LEARN
PRINT [WHAT DO YOU WANT TO CALL]
PRINT [THIS PICTURE?]
MAKE "NAME (FIRST READLINE)
DEFINE :NAME (FPUT [ ] :HISTORY)
END
```

The reason for taking the NAME of the procedure to be FIRST of READLINE is that READLINE always outputs the typed line as a list, and DEFINE needs the procedure name to be specified as a word. Also, note that the second input given to DEFINE is FPUT [ ] :HISTORY, since you need to include an empty input list for the procedure being defined. Also, you should make LEARN clear the screen and reinitialize HISTORY to prepare for a new drawing.

The behavior of LEARN is now:

```
WHAT DO YOU WANT TO CALL
THIS PICTURE?
>BOX
```

There is now a procedure called BOX, which, when run, draws the picture that currently appears on the screen.

Now you must add a command that asks for an input line and runs it. This is accomplished by

```
TO ASK
PRINT [WHAT PICTURE DO YOU WANT]
PRINT [TO SHOW?]
RUN.AND.RECORD READLINE
END
```

Notice that the input READLINE line is both run and recorded. Note also that *any* Logo command could be input and executed, not just a call to a procedure created by LEARN.

Finally, you need only add the appropriate lines to the COMMAND procedure so that it will recognize the characters S (for save) and P (for picture) and run the appropriate procedures.

The Complete INSTANT System

Here is a complete listing of the INSTANT system developed in the preceding sections:

```
TO SETUP
MAKE "HISTORY [ ]
TELL TURTLE
CLEARSCREEN
INSTANT
END

TO INSTANT
COMMAND
INSTANT
END

TO COMMAND
MAKE "COM READCHAR
IF :COM = "F RUN.AND.RECORD [FORWARD 10]
IF :COM = "R RUN.AND.RECORD [RIGHT 30]
IF :COM = "L RUN.AND.RECORD [LEFT 30]
IF :COM = "C SETUP
IF :COM = "U UNDO
IF :COM = "S LEARN
IF :COM = "P ASK
END
```

```
TO RUN.AND.RECORD :ACTION
RUN :ACTION
MAKE "HISTORY (LPUT :ACTION :HISTORY)
END

TO UNDO
IF :HISTORY = [ ] STOP
MAKE "HISTORY BUTLAST :HISTORY
CLEARSCREEN
RUN.ALL :HISTORY
END

TO RUN.ALL :COMMANDS
IF :COMMANDS = [ ] STOP
RUN FIRST :COMMANDS
RUN.ALL (BUTFIRST :COMMANDS)
END

TO LEARN
PRINT [WHAT DO YOU WANT TO CALL]
PRINT [THIS PICTURE?]
MAKE "NAME (FIRST READLINE)
DEFINE :NAME (FPUT [ ] :HISTORY)
SETUP
END

TO ASK
PRINT [WHAT PICTURE DO YOU WANT]
PRINT [TO SHOW?]
RUN.AND.RECORD READLINE
END
```

There are many possible modifications and improvements to this system. For a good exercise in manipulating lists, consider the following problem. A typical HISTORY list to be assembled into a procedure might look like:

```
FORWARD 10
RIGHT 30
LEFT 30
FORWARD 10
RIGHT 30
RIGHT 30
RIGHT 30
FORWARD 10
FORWARD 10
```

It would be nice if, before the HISTORY list is made into a procedure, it could be "compressed" so that the procedure that is defined would consist of the command sequence

```
FORWARD 20
RIGHT 90
FORWARD 20
```

Write a procedure COMPRESS that will perform this kind of transformation on a list of turtle commands. Once you have COMPRESS, the LEARN procedure can be rewritten as:

```
TO LEARN
PRINT [WHAT DO YOU WANT TO CALL]
PRINT [THIS PICTURE?]
MAKE "NAME (FIRST READLINE)
DEFINE :NAME (FPUT [ ] (COMPRESS :HISTORY))
SETUP
END
```

11.2.3. The TEXT Command

In some instances, it is useful to have an "inverse operation" to DEFINE, that is, to be able to take a procedure that is already defined and to extract the text of the procedure so that it can be manipulated as a list. This is done with the Logo command TEXT, which takes a procedure name as input and outputs the text of the procedure in the same format as is used in DEFINE.

For example, assume that CORNER is defined as

```
TO CORNER :A :B
FORWARD :A
RIGHT :B
END
```

Then TEXT "CORNER is the list

```
[[:A :B] [FORWARD :A] [RIGHT :B]]
```

Using TEXT, you can write procedures that examine and manipulate other procedures.

11.2.4. Adding New Programming Constructs

The ability to use list operations to construct lists, and then to RUN these lists as commands, allows you to add new programming constructs to the basic Logo language. For instance, suppose you would like to have a WHILE command that can be used to keep repeating something over and over as long as some condition is true, as in:

```
WHILE [XCOR < 20] [FORWARD 1]
```

Logo does not include WHILE as a primitive command. But you can use RUN to define your own WHILE command as a procedure that takes two lists as inputs. The first list specifies a condition to be tested, and the second list specifies an action to be repeated over and over as long as the condition remains true. The WHILE procedure first tests if the condition is true by RUNning the condition list. If the result is true, the WHILE procedure RUNs the action list. This sequence is repeated over and over:

```
TO WHILE :CONDITION :ACTION
IF NOT (RUN :CONDITION) STOP
RUN :ACTION
WHILE :CONDITION :ACTION
END
```

As a more complex example, you can implement a FOR procedure that works as follows:

```
FOR [COUNT 1 5] [PRINT :COUNT * :COUNT]
1
4
9
16
25
```

The FOR procedure takes two lists as inputs. The first list is a "FOR list" that specifies a loop variable together with its initial and final values. The second input specifies an action that should be executed for all values of the loop variable between the initial and final values. To implement FOR, you extract from the FOR list the name of the variable and the initial and final values, and pass these on to a subprocedure FOR.LOOP, which does the actual work of looping. It is convenient to write separate, short procedures to extract the parts of the FOR list:

```
TO VAR :FLIST
OUTPUT FIRST :FLIST
END

TO INITIAL :FLIST
OUTPUT FIRST BUTFIRST :FLIST
END

TO FINAL :FLIST
OUTPUT LAST :FLIST
END
```

Then the FOR procedure is written as:

```
TO FOR :FLIST :ACTION
FOR.LOOP (VAR :FLIST)
         (INITIAL :FLIST)
         (FINAL :FLIST)
         :ACTION
END
```

The FOR.LOOP procedure takes as inputs the variable, the initial and final values, and the action to be RUN. It uses MAKE to set the variable to the initial value and RUNs the action. Then it repeats the sequence, with the initial value incremented by 1. As a stop rule, FOR.LOOP tests to see whether the initial value has become greater than the final value. Here is the procedure:

```
TO FOR.LOOP :VAR :INITIAL :FINAL :ACTION
IF :INITIAL > :FINAL STOP
MAKE :VAR :INITIAL
RUN :ACTION
FOR.LOOP :VAR (:INITIAL + 1) :FINAL :ACTION
END
```

Note that in the second line of FOR.LOOP, you say MAKE :VAR, rather than MAKE "VAR, because the name of the variable being set is the value associated with VAR, rather than the literal word VAR. For example, in executing the command

```
FOR [COUNT 1 5] [PRINT :COUNT * :COUNT]
```

the value of VAR is the word COUNT, and COUNT is the variable you want to set using the MAKE command. Compare the "tricky use of MAKE" shown on page 86.

11.3. More Projects Using Lists

This section presents two open-ended projects that make use of Logo's capability to manipulate lists. The first project is a highly simplified version of a famous program called DOCTOR that was developed in the early 1960s by Joseph Weizenbaum. The Logo implementation here uses the association list mechanism that was explained in Section 11.1.2. The second program is a Logo version of a well-known computer "learning" program called ANIMAL, which demonstrates how to use hierarchical list structures to keep track of information.

11.3.1. Example: The DOCTOR Program

The DOCTOR program engages in a "conversation" with the computer user, in which it mimics a nondirective style of psychotherapy. Here is a sample conversation:[5]

DOCTOR
WHAT SEEMS TO BE THE PROBLEM

>EVERYONE HATES ME
YOU SAY EVERYONE HATES YOU

>WHEN I ENTER A ROOM NO ONE WILL LOOK ME IN THE EYE
MANY PEOPLE HAVE THE SAME SORTS OF FEELINGS

>THEY LAUGH AT ME BEHIND MY BACK
YOU SEEM TO THINK THEY LAUGH AT YOU BEHIND YOUR BACK

>EVEN MY MOTHER HATES ME
YOU FEEL THAT EVEN YOUR MOTHER HATES YOU

>EVEN WHEN I WAS A BABY SHE SAID I WAS ALWAYS BAD
PLEASE CONTINUE

How the DOCTOR Program Works

Although at first glance, the DOCTOR program seems to understand and reply to the user's remarks, in reality it is doing nothing of the sort. In fact, the program has two simple methods for generating a response. The first method is to ignore what the user types and simply respond with some sort of hedge like PLEASE CONTINUE or MANY PEOPLE HAVE THE SAME

[5] In order to simplify the program, all punctuation has been omitted. As usual, the computer's typeout has been italicized to distinguish it from the user's responses.

SORTS OF FEELINGS. The second method involves taking the user's reply, changing some common words like "I," "me," and "am" to the corresponding second-person words and appending the transformed response to some qualifying phrase such as YOU SAY or YOU SEEM TO THINK. The program chooses one of these methods at random for each response.

We'll examine these two methods in turn. The first is very simple. What the program prints is just a phrase picked at random from a suitable list of hedges such as

```
MAKE "HEDGES
      [ [PLEASE GO ON]
        [PLEASE CONTINUE]
        [MANY PEOPLE HAVE THE SAME SORTS OF FEELINGS] ]
```

The part of the program that implements the first method is just[6]

```
TO HEDGE
PRINT PICKRANDOM :HEDGES
END
```

The second method is more complicated. You must take the user's typed-in response, change the "I" words to the corresponding "you" words, and append this to a randomly selected qualifier. To perform the "I-you" change, you can use the SUBST procedure in Section 11.1.2, where the substitution TABLE of pairs is made up of first-person pronouns and their second-person counterparts:

```
MAKE "PRONOUNS [[I YOU] [ME YOU] [MY YOUR] [AM ARE]]

TO CHANGE.PERSON :PHRASE
OUTPUT SUBST :PHRASE :PRONOUNS
END
```

So if the collection of qualifiers is given by

```
MAKE "QUALIFY [[YOU SEEM TO THINK]
               [YOU FEEL THAT]
               [YOU SAY]]
```

then the second type of response to the user's input is generated by

[6] We use here the PICKRANDOM procedure (page 143). Notice that although we designed PICKRANDOM to pick a random word from a list of words, the generality of the Logo list operations FIRST and BUTFIRST ensures that the same procedure also works to pick a random element from any list.

```
TO RESPOND :USER.INPUT
PRINT SE (PICKRANDOM :QUALIFY)
         (CHANGE.PERSON :USER.INPUT)
END
```

Now you can put both methods together. You can select between the methods at random, using the test IF (RAND 2) = 0 to generate TRUE or FALSE with equal chances.[7] You can also terminate the conversation if the user types GOODBYE:

```
TO DOCTOR.LOOP
MAKE "USER.INPUT READLINE
IF :USER.INPUT = [GOODBYE]
   PRINT [COME SEE ME AGAIN] STOP
IF (RAND 2) = 0 HEDGE ELSE RESPOND :USER.INPUT
DOCTOR.LOOP
END
```

All that is missing now is a procedure DOCTOR to start things going. This should initialize the lists QUALIFY, HEDGES, and PRONOUNS used above, print an opening remark, and call DOCTOR.LOOP:

```
TO DOCTOR
MAKE "QUALIFY [[YOU SEEM TO THINK]
               [YOU FEEL THAT]
               [YOU SAY]]
MAKE "HEDGES [[PLEASE GO ON]
               [PLEASE CONTINUE]
               [MANY PEOPLE HAVE THE SAME SORTS OF
               FEELINGS]]
MAKE "PRONOUNS [[I YOU] [ME YOU] [MY YOUR] [AM ARE]]
PRINT [WHAT SEEMS TO BE THE PROBLEM]
DOCTOR.LOOP
END
```

Extending the Program

The previous program is only a simple sketch. One immediate extension you'll want to make is to increase its repertoire of HEDGES and QUALIFY, so that the responses are more varied. Another idea is to upgrade the RESPOND procedure not only to change first person words to second person, but also second person to first. For instance, if the user types

[7] Use the RAND procedure from Section 6.2.2.

YOU ARE NOT BEING VERY HELPFUL TO ME

the program should respond with something like

YOU FEEL THAT I AM NOT BEING VERY HELPFUL TO YOU

Another idea is this. Every so often, the program should save away the user's response. Then, a few exchanges later, the program could say something like "Earlier you said that . . ." Still other ideas are to have the program select special responses, when the user mentions certain words, like "computer."

By including more and more of these features, you can make the program's conversations quite elaborate. The responses of Weizenbaum's original DOCTOR program have been occasionally mistaken for those of a real person, and this has led some people to advocate using such programs in the treatment of psychiatric patients. Others, including Weizenbaum, maintain that this would be extremely unethical. For a further discussion of these points see Weizenbaum's book [18].

11.3.2. The ANIMAL Program

ANIMAL is a well-known computer program that asks the user to think of an animal and then tries to guess what animal it is by asking yes-or-no questions. Here is a sample session with the program:

```
ANIMAL
THINK OF AN ANIMAL. I WILL
TRY TO GUESS IT.
DOES IT HAVE LEGS?
>YES

IS IT A CAT?
>YES

LOOK HOW SMART I AM!
LET'S TRY AGAIN. . .
THINK OF AN ANIMAL. I WILL
TRY TO GUESS IT.
DOES IT HAVE LEGS?
>NO

DOES IT CRAWL?
>YES

IS IT A SNAKE?
>YES
```

LOOK HOW SMART I AM!
LET'S TRY AGAIN. . .
.
.
.

The cleverness of the program is that it learns from its mistakes. Here is what happens when it guesses incorrectly:

DOES IT HAVE LEGS?
>NO

DOES IT CRAWL?
>YES

IS IT A SNAKE?
>NO

OH WELL, I WAS WRONG.
WHAT WAS IT?
>EARTHWORM
PLEASE TYPE IN A QUESTION
WHOSE ANSWER
IS YES FOR AN EARTHWORM AND
NO FOR A SNAKE
>DOES IT LIVE UNDERGROUND?
LET'S TRY AGAIN. . .
.
.
.

The next time the program runs across this situation it will behave like this:

DOES IT HAVE LEGS?
>NO

DOES IT CRAWL?
>YES

DOES IT LIVE UNDERGROUND?
.
.
.

So the program becomes smarter and smarter as it is used more and more.

How the ANIMAL Program Works

The key to the program is its knowledge structure. This can be thought of as a tree, as shown in Figure 11.2. The tree is made up of "nodes," where each node consists of a QUESTION to ask, a YES.BRANCH to follow if the answer to the question is yes, and a NO.BRANCH to follow if the answer is no.

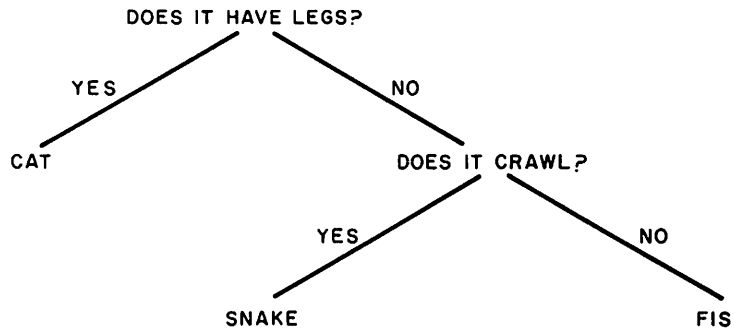


Figure 11.2: Knowledge tree for the ANIMAL program.

The basic operation of the program is to begin at the top node of the tree and work its way down, following the YES.BRANCH or the NO.BRANCH according to the answer to the QUESTION. If the program reaches a node that consists of only a single item, it guesses that as the animal.

When the program guesses incorrectly, it "gets smarter" by expanding the tree. It asks the user for the correct response and a question that distinguishes the correct response from the incorrect response. It then replaces the old single-item node by a new node made up of the user's question, the correct response as the YES.BRANCH and the old incorrect response as the NO.BRANCH. For example, to learn the difference between a snake and an earthworm, the program expands the tree, replacing the SNAKE node by a node whose QUESTION is DOES IT LIVE UNDERGROUND?, whose YES.BRANCH is EARTHWORM, and whose NO.BRANCH is SNAKE.[8]

That's all there is to it.

Using Lists

The ANIMAL program can be conveniently written in Logo, because lists are just the right tool for representing the knowledge tree. You can think of the tree as a list called KNOWLEGE that has three elements: a QUESTION, a YES.BRANCH, and a NO.BRANCH. Of course YES.BRANCH and

[8]Of course, if the user types in *wrong* information, then the program will get stupider instead of smarter. Also, the program we shall describe below does not check for *inconsistent* responses on the part of the user. Extending the program to do so is a good project.

NO.BRANCH may themselves be lists that have the same structure. And so you have sublists and sublists, until you finally reach branches that are words, which give the actual animals to be guessed.

Here is a Logo list that represents the tree shown in Figure 11.2:

```
[ [DOES IT HAVE LEGS?]
  CAT
  [ [DOES IT CRAWL?]
    SNAKE
    FISH]]
```

When snake is distinguished from earthworm, the list becomes

```
[ [DOES IT HAVE LEGS?]
  CAT
  [ [DOES IT CRAWL?]
    [ [DOES IT LIVE UNDERGROUND?]
      EARTHWORM
      SNAKE]
    FISH] ]
```

With the program's knowledge structured in this way, you can extract the QUESTION, YES.BRANCH, and NO.BRANCH parts of a given node by using the following procedures:

```
TO QUESTION:NODE
OUTPUT FIRST :NODE
END

TO YES.BRANCH :NODE
OUTPUT FIRST (BUTFIRST :NODE)
END

TO NO.BRANCH :NODE
OUTPUT LAST :NODE
END
```

To construct a node from the three constituent parts, we can use the LIST procedure given in section 11.1.1, as follows:

```
TO MAKE.NODE :QUESTION :YES.BRANCH :NO.BRANCH
OUTPUT FPUT :QUESTION (LIST :YES.BRANCH :NO.BRANCH)
```

The Main Procedure

Here is the procedure that starts the program:

```
TO ANIMAL
PRINT [THINK OF AN ANIMAL. I WILL]
PRINT [TRY TO GUESS IT]
CHOOSE.BRANCH :KNOWLEDGE
PRINT [LET'S TRY AGAIN. . .]
ANIMAL
END
```

It prints the instructions, does the guessing, and continues this over and over. The real work is done by the CHOOSE.BRANCH procedure, which is meant to be called with a node as input. It is initially called with the node that is the entire KNOWLEDGE list of the program:

```
TO CHOOSE.BRANCH :NODE
IF (WORD? :NODE) GUESS :NODE STOP
MAKE "RESPONSE ASK.YES.OR.NO (QUESTION :NODE)
IF :RESPONSE = [YES]
   CHOOSE.BRANCH (YES.BRANCH :NODE) STOP
CHOOSE.BRANCH (NO.BRANCH :NODE)
END
```

CHOOSE.BRANCH implements precisely the technique explained above. It asks the question associated with the node and then continues with the YES.BRANCH or the NO.BRANCH according to the result of the question. When it reaches a node that is a single word, it uses that as its guess. (The GUESS procedure, which actually makes the guess, is discussed below.) Notice how the "continues with . . ." part of the strategy is implemented by a CHOOSE.BRANCH calling itself recursively using the appropriate branch as the new node.

Asking Questions

The following procedure is used to ask a yes-or-no question. It takes the question as input and returns either [YES] or [NO].

```
TO ASK.YES.OR.NO :QUESTION
PRINT :QUESTION
MAKE "INPUT READLINE
IF :INPUT = [YES] OUTPUT [YES]
IF :INPUT = [NO] OUTPUT [NO]
PRINT [PLEASE TYPE "YES" OR "NO" ]
OUTPUT ASK.YES.OR.NO :QUESTION
END
```

If the user responds with something other than YES or NO, the procedure repeats the question, using the same "try again" method as with the READNUMBER procedure on page 98.

"A" or "An"

One nicety that the program must handle when making guesses is to distinguish between animal names that begin with vowels and those that do not. If the guess if "snake," the program should ask "Is it *a* snake?" while, if the guess is "earthworm," the program should ask "Is it *an* earthworm?" The following procedure helps to do this. It takes a word as input and outputs a sentence consisting of the word preceded by "a" or "an" as appropriate:

```
TO ADD.A.OR.AN :WORD
TEST MEMBER? (FIRST :WORD) [A E I O U]
IFT OUTPUT SENTENCE "AN :WORD
IFF OUTPUT SENTENCE "A :WORD
END
```

The program uses the MEMBER? procedure described on page 143. Compare the BEGINS.WITH.VOWEL? procedure on page 143.

Making a Guess

When CHOOSE.BRANCH reaches a node with only a single animal, it calls the GUESS procedure with that animal as input.

```
TO GUESS :ANIMAL
MAKE "FINAL.QUESTION
        (SE [IS IT] (ADD.A.OR.AN :ANIMAL) [?])
MAKE "RESPONSE ASK.YES.OR.NO :FINAL.QUESTION
IF :RESPONSE = [YES]
  PRINT [LOOK HOW SMART I AM!] STOP
GET.SMARTER :ANIMAL
END
```

GUESS first formulates the appropriate "Is it (a or an) . . . ?" question and gets the response. If the guess is correct, the program brags about how smart it is and stops, returning eventually to the ANIMAL procedure, which starts the next round. If the guess is wrong, the program must grow smarter.

Getting Smarter

Getting smarter consists, first of all, of asking the user for the right animal and for a question that distinguishes the right animal from the wrong one. Observe how the "a or an" choice is needed to construct the request for a question.

```
TO GET.SMARTER :WRONG.ANSWER
PRINT [OH WELL, I WAS WRONG.]
PRINT [WHAT WAS IT?]
MAKE "RIGHT.ANSWER (LAST READLINE)
PRINT [PLEASE TYPE IN A QUESTION]
PRINT [WHOSE ANSWER]
PRINT (SENTENCE [IS YES FOR]
                (ADD.A.OR.AN :RIGHT.ANSWER) [AND] )
PRINT (SENTENCE [NO FOR]
                (ADD.A.OR.AN :WRONG.ANSWER))
MAKE "QUESTION READLINE
EXTEND.KNOWLEDGE :QUESTION
                 :RIGHT.ANSWER
                 :WRONG.ANSWER
END
```

Once the new question and the two answers are in hand, the program proceeds to extend its knowledge. The KNOWLEDGE list is extended by replacing the old node—consisting of just the old answer—by a branching node consisting of a new question with the new animal as the YES.BRANCH and the old question as the NO.BRANCH.

```
TO EXTEND.KNOWLEDGE :NEW.QUESTION :YES.ANSWER :NO.ANSWER
MAKE "KNOWLEDGE
      REPLACE :KNOWLEDGE
              :NO.ANSWER
              (MAKE.NODE :NEW.QUESTION
                         :YES.ANSWER
                         :NO.ANSWER)
END
```

Finally, there is the procedure that does the actual replacement. This takes as inputs:

- A list that represents a tree of QUESTION—YES.BRANCH—NO.BRANCH nodes
- A node to be replaced
- The thing to replace it with

The output of REPLACE is a copy of the tree with the old node replaced by the designated replacement.

```
TO REPLACE :TREE :NODE :REPLACEMENT
IF :TREE = :NODE OUTPUT :REPLACEMENT
IF WORD? :TREE OUTPUT :TREE
OUTPUT (MAKE.NODE QUESTION :TREE
                   REPLACE (YES.BRANCH :TREE)
                            :NODE
                            :REPLACEMENT
                   REPLACE (NO.BRANCH :TREE)
                            :NODE
                            :REPLACEMENT)
END
```

REPLACE is the most difficult procedure in the ANIMAL program. It uses a recursive strategy somewhat as in the SUBST procedure (Section 11.1.2), but more complicated. The idea is that if the tree itself is the node to replace, you output the replacement. Otherwise, the new tree should be formed from the original tree's QUESTION, together with the result of performing the replacement recursively in the YES.BRANCH and the NO.BRANCH. This reduces the substitution to operations on smaller and smaller subtrees of the original. Finally, when you reduce to nodes that are individual words, you should output the words themselves.

Running the Program

Figure 11.3 shows the structure of procedure calls for the entire ANIMAL program.

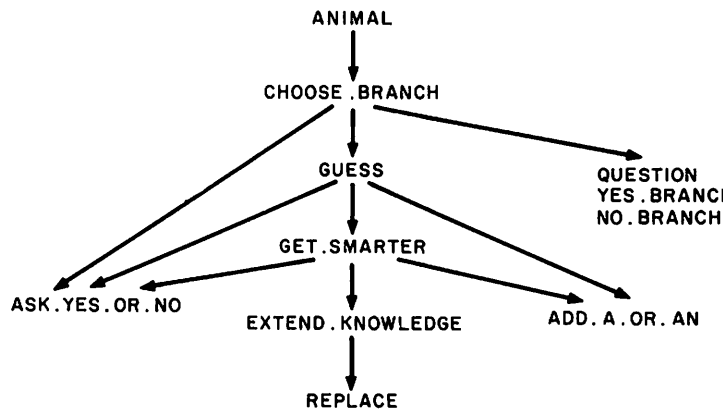


Figure 11.3: Structure of procedure calls in the ANIMAL program.

To run the program, you initialize the KNOWLEDGE list to any animal you like, and run the ANIMAL procedure. For example,

```
MAKE "KNOWLEDGE "FISH
ANIMAL
```

When you are done with your session, you can save your workspace, which will include the final value of the KNOWLEDGE tree. Thus when you read in the program to play with it the next time, the wisdom gained during previous sessions will not be lost.

Modifications

ANIMAL is a lot of fun to use as it is, especially as you begin to input funny animals and outrageous questions. There are also many good projects that arise from extending the program's capabilities. One idea is to vary the responses that the program gives. Another is to have the program request not a literal yes-or-no question, but rather a fact that distinguishes between the two animals. Then the program can use this fact to generate its own question. This requires that you build in some knowledge about grammar, somewhat like the capabilities in the DOCTOR program. Finally—and this is really open-ended—you can try to make the program smart enough to distinguish legitimate responses from wisecracks. Here is a sample dialogue with a more sophisticated animal program:[9]

```
DOES IT HAVE HORNS?
YES

IS IT A BUFFALO?
NO

WELL, I'M NOT TOO SHARP TODAY. I GIVE UP.
JUST WHAT KIND OF BEAST DID YOU HAVE IN MIND?
A GAZELLE

TELL ME SOMETHING ABOUT A GAZELLE THAT IS NOT TRUE ABOUT A
BUFFALO
A GAZELLE IS GRACEFUL

THAT WAS FUN. WANT TO TRY AGAIN?
MAYBE
HEY, CAN YOU GIVE ME A YES OR NO ANSWER
YES
```

[9]The dialogue given here is with a program by B. Greenberg [11].

DOES IT HAVE HORNS?
NO

IS IT A BUTTERFLY?
NO

WELL, I'M NOT TOO SHARP TODAY. I GIVE UP. JUST WHAT KIND OF BEAST DID YOU HAVE IN MIND?
A PIG

TELL ME SOMETHING ABOUT A PIG THAT IS NOT TRUE ABOUT A BUTTERFLY
A PIG IS SLOVENLY

THAT WAS FUN. WANT TO TRY AGAIN?
YES

DOES IT HAVE HORNS?
NO

IS THIS ANIMAL SLOVENLY?
YES

IS IT A PIG?
YES

HEY, HEY, I SURE AM CLEVER, HUH?
THAT WAS FUN. WANT TO TRY AGAIN?
YES

DOES IT HAVE HORNS
YES

IS THIS ANIMAL GRACEFUL?
YOU ARE NOT GRACEFUL
HEY CAN YOU GIVE ME A YES OR NO ANSWER?
NO

IS IT A BUFFALO?
NO

WELL, I'M NOT TOO SHARP TODAY. I GIVE UP.
JUST WHAT KIND OF BEAST DID YOU HAVE IN MIND?
BULL

TELL ME SOMETHING ABOUT A BULL THAT IS NOT TRUE ABOUT A BUFFALO
WHY DON'T YOU TELL ME SOMETHING, YOU ELECTRONIC MORON?
AW, BE SERIOUS. I ASKED YOU A REAL QUESTION
IT WOULD MARRY A COW
.
.
.

CHAPTER 12

Glossary of Logo Primitive Commands

This chapter lists the primitive commands included in the TI Logo system together with their abbreviations and examples of how many of them are used. As in the rest of this book, when we wish to emphasize the distinction between what the user types and what the computer responds, we have printed the latter in italics.

12.1. Graphics Commands

These are Logo's commands for controlling the graphics screen using the turtle, sprites, and tiles.

BACK Abbreviated BK
Example:

```
BACK 100
{turtle moves backward 100 units}
```

Takes one number as input and moves the active turtle or sprite that many units in the opposite direction from which it is facing.

BACKGROUND Abbreviated BG
Example:

```
TELL BACKGROUND
SETCOLOR :RED
{screen background will now be red}
```

Used with TELL to direct graphics commands to the background.

BIG Takes no input. Changes all sprite 32 × 32 units on a side, rather than their usual 16 × 16 size. See SMALL and SIZE. (Not included in the first release of TI Logo.)

CARRY Example:

```
CARRY :TRUCK
{active sprite now has the TRUCK shape (number 2)}
```

Takes one numeric input in the range 0 through 25. (Numbers outside this range will be reduced modulo 26, that is, reduced to the remainder after dividing by 26.) Tells the active sprite to "carry" the corresponding shape.

CLEARSCREEN Abbreviated CS
Takes no inputs. Clears the screen.

COLOR Takes no inputs. Outputs the color number of the active sprite or tile. If the turtle is active, outputs the turtle's pen color.

COLORBACKGROUND Abbreviated CB
Example:

```
COLORBACKGROUND :BLUE
```

is equivalent to

```
TELL BACKGROUND SETCOLOR :BLUE
```

except that it does not alter the active sprite, as does using TELL. COLORBACKGROUND takes one numeric input in the range 0 through 15. (Numbers outside this range will be reduced modulo 16, that is, reduced to the remainder after dividing by 16.) It sets the screen background to the corresponding color.

DOT Example:

```
DOT 30 30
```

Takes two numeric inputs, *x* and *y* coordinates, and places a dot at the designated point on the turtle screen.

EACH Example:

```
TELL :ALL
EACH [SETHEADING 10 * YOURNUMBER]
```

Takes a list of commands as inputs, and runs the list for each active sprite. The operation YOURNUMBER when used within the list returns the number of the sprite.

FORWARD Abbreviated FD
Example:

```
FORWARD 50
{turtle or sprite moves forward 50 units}
```

Takes one numeric input. Moves the currently active turtle or sprite the designated number of units in the direction in which it is facing. Draws a line if the turtle's pen is down.

FREEZE Takes no inputs. Stops motion of all sprites on the screen. Motion is resumed with THAW.

HEADING Example:

```
SETHEADING HEADING + 10
{rotates the turtle 10 degrees clockwise}
```

Takes no inputs. Outputs heading of the currently active turtle or sprite as a number between 0 and 360.

HIDETURTLE Abbreviated HT
Takes no inputs. Makes the turtle pointer disappear.

HOME Takes no inputs. Moves the turtle to the center of the screen, pointing straight up. Moves the active sprite to the center of the screen without changing the heading.

LEFT Abbreviated LT
Example:

```
LEFT 90
{turtle rotates 90 degrees counterclockwise}
```

Takes one numeric input. Rotates the currently active turtle or sprite that many degrees counterclockwise.

LOOKLIKE Takes one numeric input. Synonym for CARRY.

MAKECHAR Abbreviated MC
Takes one numeric input in the range 0 through 255. (Numbers outside this range will be reduced modulo 256, that is, reduced to the remainder after dividing by 256.) Enables you to define or edit the corresponding character shape. See Section 4.3.

MAKESHAPE Abbreviated MS
Takes one numeric input in the range 0 through 25. (Numbers outside this range will be reduced modulo 26, that is, reduced by the remainder after dividing by 26.) Enables you to define or edit the corresponding sprite shape. See Section 4.2.

NOTURTLE Takes no inputs. Exits turtle mode.

NUMBEROF Example:

```
PRINT NUMBEROF WHO
```

Takes one input. Usually used in conjunction with WHO to return the number of the active sprite.

PENDOWN Abbreviated PD
Takes no inputs. Causes the turtle to leave a trail when it moves.

PENERASE Abbreviated PE
Takes no inputs. Causes the turtle to erase (that is, change to the background color) any points that it passes over.

PENREVERSE Abbreviated PR
Takes no inputs. Causes the turtle to reverse any point it passes over. The

effect is that the turtle will draw, unless it is retracing a line, in which case the line will be erased.

PENUP Abbreviated PU
Takes no inputs. Causes the turtle to move without leaving a trail.

PUTTILE Abbreviated PT
Example:

```
PUTTILE 100 16 12
{tile number 100 appears at the center of the screen}
```

Takes a tile number and row and column numbers as inputs. Places the tile at the designated row and column.

RIGHT Abbreviated RT
Example:

```
RIGHT 45
{turtle rotates 45 degrees clockwise}
```

Takes one numeric input. Rotates the active turtle or sprite that many degrees clockwise.

SETCOLOR Abbreviated SC
Example:

```
TELL SPRITE 5
SETCOLOR :RED
{sprite 5 is now red}
TELL TILE 100
SETCOLOR [6 15]
{tile 100 now has foreground color red
and background color white}
```

For sprites, takes as input a number in the range 0 through 15. (Numbers outside this range will be reduced modulo 16, that is, reduced to the remainder after dividing by 16.) Changes the active sprite to that color. The COLOR of the turtle is the color in which it draws. With tiles or the turtle SETCOLOR can also take as input a list of two color numbers, which specify the foreground and background colors.

SETHEADING Abbreviated SH
Example:

```
SETHEADING 180
{turtle now faces straight down}
```

Takes one numeric input. Rotates the active sprite or turtle to point in the direction specified. The input is interpreted as a number in degrees. Zero is straight up, with heading increasing clockwise.

SETSPEED Abbreviated SS
Example:

```
TELL SPRITE 10
SETSPEED 100
```

Takes as input a number in the range −127 through 127. Sets the speed of the active sprite.

SHAPE Takes no input. Returns the shape number of the active sprite.

SHOWTURTLE Abbreviated ST
Takes no inputs. Makes the turtle pointer appear.

SIZE Takes no inputs. Outputs 16 if sprites are currently SMALL and 32 if they are BIG. (Not included in the first release of TI Logo.)

SMALL Takes no inputs. Makes sprites 16 × 16 units in size. See BIG. (Not included in the first release of TI Logo.)

SPEED Takes no inputs. Outputs the speed of the currently active sprite.

SPRITE Example:

```
TELL SPRITE 5
SETSPEED 100
```

Takes one numeric input in the range 0 through 31. Used with TELL in order to direct graphics commands to a sprite.

SV Example:

```
TELL SPRITE 5
SV 30 30
```

Takes two numeric inputs, which are used to set the *x* and *y* velocity components of the active sprite.

SX Takes one numeric input. Moves the currently active sprite or turtle horizontally to the specified coordinate.

SXV Takes one numeric input. Sets the *x* velocity component of the active sprite.

SXY Example:

```
SXY 80 50
{turtle moves to position (80,50)}
```

Takes two numeric inputs. Moves the currently active sprite or turtle to the specified point, where (0,0) is center of screen.

SY Takes one numeric input and moves the currently active sprite or turtle vertically to the specified coordinate.

SYV Takes one numeric input. Sets the *y* velocity component of the active sprite.

TELL Examples:

```
TELL SPRITE 1
TELL TILE 50
TELL TURTLE
TELL BACKGROUND
TELL [1 5 8]
TELL 10
```

Used to direct subsequent graphics commands to an object, which becomes the "active object." If used with a list of numbers, commands are directed to all sprites in the list. TELL used with a number (as in the final example above) designates a sprite.

THAW Takes no inputs. Restores motion that was stopped by FREEZE.

TILE Example:

```
TELL TILE 100
SETCOLOR :RED
```

Used with TELL in order to designate an active tile.

TURTLE Takes no inputs. Used with tell in order to specify the turtle.

WHERE Takes no inputs. If the turtle is the currently active object, outputs a list of three numbers: the *x*-coordinate, *y*-coordinate, and heading.

WHO Takes no inputs. Outputs the currently active graphics object (as specified by the previous TELL).

XCOR Example:

```
SETX XCOR + 10
{moves the turtle 10 units to the right}
```

Takes no inputs. Outputs the *x* coordinate of the turtle or currently active sprite.

XVEL Takes no inputs. Outputs the *x* velocity component of the currently active sprite.

YCOR Takes no inputs. Outputs the *y* coordinate of the turtle or currently active sprite.

YOURNUMBER Abbreviated YN
Takes no inputs. Outputs the number of the currently active sprite. Normally used inside a command list with EACH.

YVEL Takes no inputs. Outputs the *y* velocity component of the currently active sprite.

12.2. Numeric Operations

These are Logo's built-in facilities for performing operations with numbers. Numbers handled by Logo must be integers in the range −32767 through 32767.

\+ Example:

```
PRINT 5 + 2
7
```

Takes two numbers as inputs, and outputs their sum.

\− Example:

```
PRINT 5 - 2
3
PRINT 1 + (-2)
-1
```

With two numeric inputs, outputs their difference. With one numeric input, outputs its negative.

* Example:

```
PRINT 5 * 2
10
```

Takes two numeric inputs, and outputs their product.

/ Example:

```
PRINT 5 / 2
2
PRINT 6 / 2
3
```

Outputs its first input divided by its second. Truncates any fractional part.

DIFFERENCE Example:

```
PRINT DIFFERENCE 10 6
4
```

Takes two numeric inputs. A prefix operation equivalent to −.

PRODUCT Takes two inputs. A prefix operation equivalent to *.

QUOTIENT Takes two inputs. A prefix operation equivalent to /.

RANDOM Takes no input. Outputs a random number in the range 0 through 9.

SUM Takes two inputs. A prefix operation equivalent to +.

12.3. Word and List Operations

In addition to numbers, Logo also includes operations for dealing with words (strings of characters) and lists (structured collections of data).

BUTFIRST Abbreviated BF
Example:

```
PRINT BUTFIRST [THIS IS A LIST]
IS A LIST
PRINT BUTFIRST "ABRACADABRA
BRACADABRA
```

If input is a list, outputs a list containing all but the first element. If input is a word, outputs a word containing all but the first character. BUTFIRST of the empty list returns the empty list. BUTFIRST of a single-character word returns the empty list.

BUTLAST Abbreviated BL
Example:

```
PRINT BUTLAST [THIS IS A LIST]
THIS IS A
PRINT BUTLAST "ABRACADABRA
ABRACADABR
```

If input is a list, outputs a list containing all but the last element. If input is a word, outputs a word containing all but the last character. BUTLAST of the empty list returns the empty list. BUTFIRST of a single-character word returns the empty list.

FIRST Abbreviated F
Example:

```
PRINT FIRST [THIS IS A LIST]
THIS
PRINT FIRST "ABRACADABRA
A
```

If input is a list, outputs the first element. If input is a word, outputs the first character. FIRST of the empty list returns the empty list.

FPUT Example:

```
PRINT FPUT [A B] [C D]
[A B] C D
```

The second input must be a list. Outputs a list consisting of the first input followed by the elements of the second input.

LAST Example:

```
PRINT LAST [THIS IS A LIST]
LIST
PRINT LAST "ABRACADABRAX
X
```

If input is a list, outputs the last element. If input is a word, outputs the last character. LAST of the empty list returns the empty list.

LENGTH Example:

```
PRINT LENGTH "ELEPHANT
8
PRINT LENGTH [ALPHA BETA GAMMA]
3
PRINT LENGTH [A [B C D] [E F]]
3
```

If input is a word, outputs the number of characters in the word. If input is a list, outputs the number of items in the list. (Not included in the first release of TI Logo.)

LPUT Example:

```
PRINT LPUT "Z [W X Y]
W X Y Z
PRINT LPUT [A B] [C D]
C D [A B]
```

Second input must be a list. Outputs a list consisting of the elements of the second input followed by the first input.

REVERSE Example:

```
PRINT REVERSE "APPLESAUCE
ECUASELPPA
PRINT REVERSE [ALPHA BETA GAMMA]
GAMMA BETA ALPHA
PRINT REVERSE [A [B C] [D E]]
[D E] [B C] A
```

If input is a word, outputs the characters of the word in reverse order. If input is a list, outputs a list of the items in reverse order. (Not included in the first release of TI Logo.)

ROTATE Example:

```
PRINT ROTATE "APPLESAUCE
PPLESAUCEA
PRINT ROTATE [ALPHA BETA GAMMA]
BETA GAMMA ALPHA
PRINT ROTATE [A [B C] [D E]]
[B C] [D E] A
```

If input is a word, outputs the word with the first character moved to the end; that is, outputs

WORD (BUTFIRST :X) (FIRST :X)

If input is a list, outputs the list with the first item moved to the end; that is, outputs

LPUT (FIRST :X) (BUTFIRST :X)

(Not included in the first release of TI Logo.)

SENTENCE Abbreviated SE
Example:

```
PRINT SENTENCE "HELLO "THERE
HELLO THERE
PRINT SENTENCE [THIS IS] [A LIST]
THIS IS A LIST
PRINT (SENTENCE "THIS [IS] [A LIST])
THIS IS A LIST
PRINT SENTENCE [[HERE IS] A] [NESTED LIST]
[HERE IS] A NESTED LIST
```

Takes a variable number of inputs. (The default is two.) If inputs are all lists, combines all their elements into a single list. If any inputs are words, they are regarded as one-word lists in performing operation.

WORD Example:

```
PRINT WORD "MISH "MASH
MISHMASH
```

Takes two inputs. Outputs a word that is the concatenation of the characters of its inputs (which must be words).

12.4. Defining and Editing Procedures

TO and EDIT are the most commonly used operations for creating and changing procedures. But Logo includes some other operations that allow more advanced manipulation of procedure definitions.

DEFINE Abbreviated DE
Example:

```
DEFINE "PTSUM [[:X :Y] [PRINT :X] [PRINT :X + :Y]]
```

defines the procedure

```
TO PTSUM :X :Y
PRINT :X
PRINT :X + :Y
END
```

Takes two inputs. First is a name, and second is a list whose elements are a list of inputs and a list for each line, and defines a procedure accordingly. Note that you normally use TO rather than DEFINE in order to define procedures. DEFINE is useful for writing procedures that define other procedures, as in the extended INSTANT system described in Section 11.2.1.

EDIT Example:

```
EDIT SQUARE
{sets up procedure SQUARE for editing}
```

Enters the procedure editor with a given procedure. If no input is specified, enters the editor with a blank screen.

END Terminates a procedure definition that is typed into the editor. It is not necessary to type END at the end of the final definition, but if you are defining more than one procedure at a time, the separate procedure definitions must be separated by END statements.

TEXT Example:

```
TO PTSUM :X :Y
PRINT :X
PRINT :X + :Y
END

PRINT TEXT "PTSUM
[:X :Y][PRINT :X][PRINT :X + :Y]
```

Takes a procedure name as input and outputs procedure text as a list, whose format is as described under DEFINE.

TO Begins procedure definition. Enters edit mode.

12.5. Conditional Expressions

Logo includes two basic facilities for allowing the user to write programs that perform tests and do different things depending on the outcomes. One is the IF. . . THEN . . . ELSE construct that is common to many computer languages. The other, TEST. . . IFT. . . IFF, is less common but often simpler to use.

BOTH Example:

```
PRINT BOTH (1 + 1 = 2) (5 = 4)
FALSE
```

Takes two inputs. Each input should be either TRUE or FALSE. Outputs TRUE if both are TRUE; otherwise outputs FALSE.

EITHER Example:

```
PRINT EITHER (1 + 1 = 2) (5 = 4)
TRUE
```

Takes two inputs and outputs TRUE if at least one is TRUE; otherwise outputs FALSE.

ELSE Used in IF. . . THEN . . . ELSE.

IF Example:

```
IF :X = 5 THEN STOP ELSE PRINT "HELLO
```

Used in the basic conditional form IF {condition} THEN {action1} ELSE {action2}. The {condition} is tested. If it is true, {action1} is performed. If it is false, {action2} is performed. The word THEN is optional. The ELSE {action2} part need not be present.

IFF Executes rest of line only if result of preceding TEST was false. See TEST.

IFT Executes rest of line only if result of preceding TEST was true. See TEST.

NOT Example:

```
IF NOT (1 = 2) PRINT "HELP
HELP
```

Outputs TRUE if its input is FALSE, FALSE if its input is TRUE.

TEST Example:

```
TEST "AB = WORD "A "B
IFF PRINT "NO
IFT PRINT "YES
YES
```

Tests a condition to be used in conjunction with IFT and IFF.

THEN Used with IF. . . THEN . . . ELSE . . .

12.6. Predicates Used with Conditional Expressions

The conditional expressions of the previous section make use of predicates, or operations that output either TRUE or FALSE. A predicate can be any procedure that outputs the word TRUE or the word FALSE. Here are the predicates that are built into Logo.

> Example:

```
IF :X > :Y STOP
```

Outputs TRUE if its first input is greater than its second, FALSE otherwise.

< Outputs TRUE if its first input is less than its second, FALSE otherwise.

= Example:

```
PRINT 20 = 10 + 10
TRUE
PRINT "A = [A]
FALSE
PRINT [A B] = SENTENCE "A "B
TRUE
```

If both inputs are numbers, compares them to see if they are numerically equal. If both inputs are words, compares them to see if they are identical character strings. If both inputs are lists, compares them to see if their corresponding elements are equal. Outputs TRUE or FALSE accordingly.

FALSE Outputs the word "FALSE. (Not included in the first release of TI Logo.)

GREATER Prefix form of >.

IS Example:

```
IF IS 7 3+4 PRINT [YES]
```

Takes two inputs. A prefix operation equivalent to =.

LESS Prefix form of <.

NUMBER? Outputs TRUE if its input is a number, FALSE otherwise.

THING? Outputs TRUE if its input has a value associated with it.

TRUE Outputs the word "TRUE. (Not included in the first release of TI Logo.)

WORD? Outputs TRUE if its input is a word, FALSE otherwise.

12.7. Controlling Procedure Execution

GO Example:

```
TO TRIANGLE :STRING
IF FIRST :STRING = :STRING THEN STOP
LOOP: PRINT :STRING
MAKE "STRING BUTFIRST :STRING
GO "LOOP
END
```

Compare this example with the TRIANGLE procedure of Section 6.3. GO takes a word as input and transfers to the line with that label. You can only GO to a label within the same procedure. Labels are defined by typing them at the beginning of the indicated line followed by a colon. GO is very rarely used in Logo programming.[1]

OUTPUT Abbreviated OP
Takes one input. Causes the current procedure to stop and output the result to the calling procedure.

REPEAT Example:

```
REPEAT 3 [PRINT "HELLO ]
HELLO
HELLO
HELLO
```

Takes a number and a list as input. RUNs the list the designated number of times.

[1] GO is occasionally useful, but is easily abused and can lead to obscure programs. In Logo, you can almost always avoid the need to use GO by taking advantage of REPEAT and/or procedure calls. Iteration constructs WHILE, FOR, and so on, can also be implemented by using RUN, as illustrated in Section 11.2.4. As the TRIANGLE procedure above shows, one can, in fact, use Logo to program in a style that is typical of most BASIC programs. That would be like pouring ketchup over caviar.

RUN Example:

```
MAKE "X [PRINT]
RUN SENTENCE :X 5
5
```

Takes a list as input. Executes the list as if it were a typed-in command line. The number of characters in the list (i.e., the number of characters you would get if you printed it) given to RUN must not exceed the maximum number of characters allowed in a top-level command line, which is 255 characters in the current implementation.

STOP Causes the current procedure to stop and return control to the calling procedure.

12.8. Input and Output

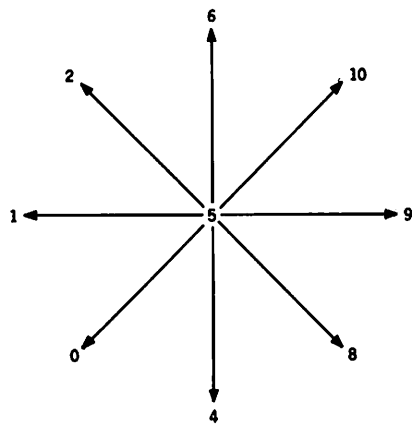
BEEP Takes no input. Starts the computer playing a tone. (Turn the tone off with NOBEEP.)

CHARNUM Abbreviated CN
Takes a character as input and outputs the code number of that character, as defined in the table in Section 4.3.3.

JOY Example:

```
PRINT JOY 1
9
```

Takes one input number, specifying joystick 1 or 2. Outputs a number which depends on the joystick position as shown:



If the joystick's button is pressed when the command JOY 1 or JOY 2 is executed, the number output will be the indicated number plus 16. You can use this to create the effect of an on-off button with the joystick. For example,

```
IF (JOY 1 > 10) CLEARSCREEN
```

will clear the screen whenever this command is given with joystick 1's button pressed down. (The button effect is not included in the original TI Logo.)

Warning: JOY 1 and JOY 2 will output incorrect values if the ALPHA LOCK key is down.,

NOBEEP Takes no inputs. Stops the tone started by BEEP.

PRINT Example:

```
PRINT "HI
HI
PRINT [HELLO OUT THERE]
HELLO OUT THERE
```

Prints its input and moves cursor to the next screen line. When PRINT prints lists, the outermost pair of brackets is not printed.

PRINTCHAR Abbreviated PC
Takes a tile number as input, and prints the corresponding tile (character) at the current cursor position.

RC? Takes no inputs. Outputs TRUE if a keyboard character is pending (i.e., the character input buffer is not empty); otherwise outputs FALSE.

READCHAR Abbreviated RC
Takes no inputs. Outputs the least recent character in the character buffer, or if empty, waits for an input character.

READLINE Abbreviated RL
Takes no inputs. Waits for an input line to be typed, terminated with ENTER. Outputs the line (as a list).

TYPE Like PRINT, but does not move cursor to the next line after printing.

WAIT Takes one numeric input and pauses the computer for that many sixtieths of a second.

12.9. Naming

CALL Example:

```
CALL 7 "LUCKYNUMBER
CALL [ALPHA BETA GAMMA] "TESTWORDS
```

Equivalent to MAKE with the order of the inputs reversed.

MAKE Example:

```
MAKE "APPLE 50
PRINT :APPLE
50
```

Takes two inputs, the first of which must be a word. Assigns the second input to be the value associated with the first input.

THING Example:

```
MAKE "APPLE 50
PRINT THING "APPLE
50
```

Outputs the value of its input (which must be a word). THING "XXX can be abbreviated as :XXX.

12.10. Filing and Managing Workspace

Workspace consists of all currently defined procedures and all names and their associated values. Workspaces can be stored in files on disk or on cassette tape.

ERASE Example:

```
ERASE SQUARE or ERASE "SQUARE
{gets rid of the procedure named SQUARE}
ERASE :X or ERASE "X
{gets rid of the variable named X}
```

Warning: ERASE "X erases *both* a variable *and* a procedure named X.

PA Prints all procedures and names.

PN Prints all currently defined names.

PO Example:

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

Takes a procedure name as input and prints the definition of the procedure.

PP Prints the tile lines of all currently defined procedures.

PRINTOUT Takes no inputs. Prints all your procedures on a thermal printer or RS232 printer. See Section 5.2. (Not included in the first release of TI Logo.)

RECALL Takes no inputs. Reads information from the cassette tape or the disk. See Section 5.2.

SAVE Takes no inputs. Transmits information to cassette tape or disk. See Section 5.2.

12.11. Music Primitives

TI Logo II includes the ability to generate music in up to three-voice harmony. You construct music by using commands, such as NOTE, and place the notes in a music buffer. Afterwards, you use the command PLAYMUSIC to play the notes that have been placed in the buffer.

CHROMATIC Changes meaning of pitch designations. See MAJOR.

DRUM Example:

DRUM [3 4 6 8]

Takes a list of numbers as input, and signals a "drumbeat" with the designated durations between beats. "Beats" are placed in the music buffer to be played by PLAYMUSIC.

LEGATO Controls "dead time" inserted between notes. See STACCATO.

LOOPMUSIC Plays the music in the buffer repeatedly. You can continue to execute Logo commands while music is playing. To stop music, use SETVOICE 0.

MAJOR As opposed to CHROMATIC. Changes the meanings of the pitch designations. In MAJOR mode, 0 is middle C and each unit is a note on the C scale. In CHROMATIC mode, each unit is a half-step. CHROMATIC is the default.

MUSIC Example:

MUSIC [0 3 5 7] [4 2 2 8]
or
MUSIC [0 3 5 7] 4

Takes as input two lists: a list of pitches and a list of durations, and places these in the music buffer. If the lists are not of the same length, the longer one is truncated. If a single number is specified as the duration, that duration

is used for each of the pitches. Volume is taken as the value specified by the previous SETVOLUME command.

NOTE Example:

NOTE 3 8 7

Takes three numbers as inputs, specifying the duration, pitch, and volume for a note, and places that note in the music buffer.

PLAYNOTE This is equivalent to playing a note from the music buffer, and then WAITing for the duration of the note. Consecutive PLAYNOTE commands will play consecutive notes. This command can function with only one voice at a time. If the music buffer contains notes for more than one voice, PLAYNOTE will use the notes for the current voice as designated by SETVOICE. PLAYNOTE can be used to synchronize music playing with other Logo commands, as is illustrated in Section 9.3.

PLAYMUSIC Abbreviated PM
Plays the music in the buffer. Logo music plays simultaneously while commands are executed, so that after giving the PLAYMUSIC command, you can proceed to execute other Logo commands while the music is playing.

REST Takes a number as input and inserts a rest of that duration in the music buffer.

SETTEMPO Takes a number as input and sets the tempo in counts per minute. With a tempo of T, a note of duration D will last (60/T) *D seconds. The default value of T is 300.

SETVOICE Takes a number 0 through 4 as input. 1, 2, or 3 select one of the three voices. Subsequent note commands will be directed to that voice. An input of 4 selects the noise generator. An input of 0 clears the music buffer.

SETVOLUME Takes a number 0 through 15 as input and sets the volume. 0, the default volume, is the softest, 15 the loudest.

STACCATO In contrast to LEGATO, the default condition. Controls "dead time" inserted between notes. For LEGATO, a dead time of 5/60 second will be used. For STACCATO, the note will sound for 5/60 second and the remainder will be dead time. For notes of duration less than 6/60 second, $(n - 1)/60$ will be used in place of 5/60.

12.12. Debugging Aids

CONTINUE Takes no inputs. Can sometimes be used to resume execution from a paused state (entered via AID or DEBUG).

DEBUG Takes no inputs. Controls an option whereby errors will enter a pause state, rather than return to top command level. See Section 5.3.

TRACEBACK Takes no inputs. When called within a procedure, prints the chain of procedure calls from the current procedure back to top level.

12.13. Editing Commands

This section describes the special keys that are used with the procedure editor. Each key is used while simultaneously pressing the FCTN key.

arrow keys Move the cursor one space up, down, right, or left.

BACK Exits the editor and processes definitions.

BEGIN Moves the cursor to the beginning of the current line.

CLEAR Deletes all characters on the current line, from the cursor rightwards.

DEL Deletes the character at the current cursor position.

ERASE Deletes the character to the left of the cursor, and moves the cursor one space to the left.

PROC'D Moves the cursor to the right end of the current line.

12.14. Other Special Keys

This section describes special keys used in Logo other than for editing.

AID Stops procedure execution and enters a pause break. See Section 5.3.

BACK Stops execution and returns control to top level. Also used to exit shape editors.

QUIT Resets the computer, destroying all programs and data in memory. Don't press QUIT unless you are finished using Logo.

ERASE Deletes the character to the left of the cursor and moves the cursor one space to the left.

12.15. Miscellaneous Commands

BYE Leaves TI Logo.

CONTENTS Outputs a list of all words currently being used in the workspace. (Not included in the first release of TI Logo.)

.HELP Prints a list of all the keywords in TI Logo. (Not included in the first release of TI Logo.)

.GC Forces a garbage collection, reclaiming unused storage. (Not included in the first release of TI Logo.)

.NODES Outputs the number of currently free nodes. This is a measure of how much storage is available in workspace. (Not included in the first release of TI Logo.)

; Causes the rest of the line not to be evaluated. (Can be used to include comments in procedures.)

12.16. Error Messages

When Logo encounters an error, it signals that fact by halting program execution and printing a message of the form:

{message}
AT LEVEL {level}*LINE* {line} of {procedure}

For example:

TELL ME HOW TO FORWAXD
AT LEVEL 1 LINE 2 OF BOX

In general, {message} is a description of the error, {line} is the line number at which the error occurred, {procedure} is the name of the procedure containing that line, and {level} tells "how many levels away from top level" Logo was running when the error occurred. That is to say, level 0 means that Logo was executing a line directly typed in, level 1 means executing a line in a procedure that was called at level 0, level 2 means executing a line in a procedure that was called at level 1, and so on.

- *TELL ME HOW TO* {something} This happens when Logo does not recognize the name of the procedure you are trying to run. Common causes are that you forgot to define the procedure in question, or that you used the wrong name. Typing errors also commonly cause this. For example, if you type FORWAXD 100 instead of FORWARD 100, you will get the error *TELL ME HOW TO FORWAXD*.

- {something} *HAS NO VALUE* This happens when you refer to the value of a name, but there is no such name in the environment. The causes are similar to those for the "no procedure" error message. Another cause is confusion between the *local* variables in a procedure and the global variables. For example, defining and running the procedure

```
TO INC :X
OUTPUT :X + 1
END
```

creates a variable X that is local to INC, but this does not mean that there is a global variable named X.

- *TELL ME MORE* A procedure was called with too few inputs.

- *NOTHING BEFORE THE* {infix-operator} This happens when an infix operator is called with nothing before it. For example,

```
PRINT * 3
```

will give the error *NOTHING BEFORE THE* *.

- {primitive} *DOESN'T LIKE* {data} *AS INPUT* This happens when you try to use an operation with a kind of data that it cannot handle. For example,

```
PRINT 1 + "X
```

results in + *DOESN'T LIKE X AS INPUT.*

- *TELL ME WHAT TO DO WITH* {data} This occurs in procedures when you generate some data and then don't say what to do with it. (In most cases, you probably meant to OUTPUT it.) For example:

```
TO SQUARE :X
:X * :X
END
```

```
SQUARE 5
TELL ME WHAT TO DO WITH 25
AT LEVEL 1 LINE 1 OF SQUARE
```

People often make this error in writing recursive procedures:

```
TO FACTORIAL :N
IF :N = 0 OUTPUT 1
:N * (FACTORIAL :N - 1)
END
```

```
PRINT FACTORIAL 1
TELL ME WHAT TO DO WITH 1
AT LEVEL 2 LINE 1 OF FACTORIAL
```

The problem here is that FACTORIAL should have an OUTPUT at the beginning of its second line.

• {procedure}
DIDN'T OUTPUT

This happens when you try to use the value returned by a procedure, but the procedure didn't output anything. For example,

```
TO PRINT.SQUARE :X
PRINT :X * :X
END

FORWARD PRINT.SQUARE 4
16
PRINT.SQUARE DIDN'T OUTPUT
```

• *OUT OF SPACE*

This happens when you have used up all available storage.

• *YOU TRIED TO DIVIDE BY ZERO*

This happens when the QUOTIENT or / operation is called with zero as the divisor.

• {object}
CAN'T {something}

This happens when you try to perform a graphic operation when the current object is not of the type that can do that operation. For example,

```
TELL TURTLE
SETSPEED 100
TURTLE CAN'T SETSPEED
```

• *OUT OF INK*

The turtle has used up all available tiles for drawing. To continue drawing, you must first clear the screen.

• *STOPPED*

Occurs when you have pressed the BACK key to stop a procedure.

• *PAUSED*

Occurs when you have pressed the AID key to temporarily halt a procedure.

• *A LABEL IS OUT OF PLACE*

• *THEN IS OUT OF PLACE*

• *ELSE IS OUT OF PLACE*

These three messages all mean that you used the indicated primitive in a context in which it doesn't make sense. (A label is signaled by the :.) Some lines that would generate such messages are

```
PRINT 5 + X:
FORWARD 100 THEN PRINT 5
```

• {something} *WAS GIVEN INSTEAD OF TRUE OR FALSE*

A command which needs TRUE or FALSE as input was given another value instead. This can occur if you forget to include an = in the input to IF or TEST as in this example:

```
IF HEADING 0 STOP
```

• {primitive} *MUST BE IN A PROCEDURE*

This happens, for example, if you use the OUTPUT, STOP, or GO commands directly at top level rather than in a procedure.

• *PROCEDURE NOT BEING DEFINED*

This means you tried to use END as a command in a procedure line. (You most likely meant to use STOP instead.) Another way to get this error is to explicitly include an END command in the list of lines given to DEFINE.

• *WHERE IS THE LABEL*

This happens if you try to GO to a label that was not defined in the procedure.

• *UNEXPECTED ")"*

Logo has run across a close parentheses for which there was no corresponding open parentheses.

MISMATCHED BRACKETS

Logo has run across a close bracket for which there was no corresponding open bracket.

TOO MANY SUBLISTS

You tried to type in a list that was too deep (i.e., too many levels of open brackets). In the current implementation, the maximum is 14.

OUT OF NOTES

The music buffer is full. You must reset it (using SETVOICE 0) before adding more notes.

• *SENTENCE IS TOO LONG*

Occurs when the output of a SENTENCE command results in a list that has too many elements.

References

1. Abelson, H. and diSessa, A. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics.* MIT Press, Cambridge, MA, 1981.

2. Bamberger, J. "The Development of Musical Intelligence I: Strategies for Representing Simple Rhythms." Memo 342, MIT Artificial Intelligence Laboratory, 1975.

3. Bamberger, J. "The Development of Musical Intelligence II: Children's Representation of Pitch Relations." Memo 401, MIT Artificial Intelligence Laboratory, 1976.

4. Bamberger, J. "Logo Music Projects: Experiments in Musical Perception and Design." Memo 523, MIT Artificial Intelligence Laboratory, 1979.

5. Bowles, K. *Problem Solving Using Pascal.* Springer-Verlag, New York, 1977.

6. diSessa, A. "Unlearning Aristotelian Physics: A Study of Knowledge-Based Learning." *Cognitive Science* (in press).

7. Feurzeig, W., Papert, S., Bloom, M., Grant, R., and Solomon, C. "Programming Languages as a Conceptual Framework for Teaching Mathematics." Report 1889, Bolt, Beranek and Newman, Inc., November, 1969.

8. Feurzeig, W., Goldenberg, E. P., Lukas, G., Manis, V., Rubenstein, R., and Stachel, R. "The Logo-S Language and the Portable Logo System." Bolt, Beranek and Newman, Inc., 1980.

9. Goldberg, A., Robson, D., and Ingalls, D.H.H. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1982.

10. Goldenberg, P. *Special Technology for Special Children.* University Park Press, Baltimore, 1979.

11. Greenberg, B. "Notes on the Programming Language Lisp." MIT Student Information Processing Board, 1978.

12. Howe, J.A.M., O'Shea, T., and Lane, F. "Teaching Mathematics through Logo Programming: An Evaluation Study." Department of Artificial Intelligence, University of Edinburgh, 1977.

13. Kay, A. "Microelectronics and the Personal Computer." *Scientific American* (September 1977).

14. Papert, S. and C. Solomon. "NIM: A Game-Playing Program." Memo 254, MIT Artificial Intelligence Laboratory, 1970.

15. Papert, S. *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books, New York, 1980.

16. Papert, S., diSessa, A., Watt, D., and Weir, S. "Final Technical Report to the National Science Foundation: Documentation and Assessment of a Children's Computer Laboratory." Memos 52, 53, MIT Logo Project, 1980.

17. Weir, S. "Logo and the Exceptional Child." *Kilobaud Microcomputing* 5, 9 (September 1981).

18. Weizenbaum, J. *Computer Power and Human Reason*. W. A. Freeman & Co., San Francisco, 1976.

19. Winston, P. and Horn, B. *Lisp*. Addison-Wesley, Reading, MA, 1981.

KEYBOARD REFERENCE GUIDE

Note that the key sequences required to access special functions depend on the type of computer console you have.

TI-99/4	TI-99/4A	
AID **(SHIFT A)**	**AID** **(FCTN 7)**	Causes the computer to pause.
BACK **(SHIFT Z)**	**BACK** **(FCTN 9)**	■ Leaves the Save and Recall Modes and returns the computer to the mode it was in. ■ Stops a procedure. ■ Leaves the Edit Mode, MAKESHAPE and MAKECHAR.
BEGIN **(SHIFT W)**	**BEGIN** **(FCTN 5)**	Moves the cursor to the beginning of the line in the Edit Mode.
CLEAR **(SHIFT C)**	**CLEAR** **(FCTN 4)**	■ Clears the MAKESHAPE and MAKECHAR grids. ■ Erases what is above and to the right of the cursor in the Edit Mode.
DELETE **(SHIFT F)**	**DELETE** **(FCTN 1)**	■ Erases what is above the cursor. ■ Moves a line up one line if the cursor is at the end of the line in the Edit Mode.
ERASE **(SHIFT T)**	**ERASE** **(FCTN 3)**	■ Erases what is one space to the left of the cursor. ■ Moves a line up one line if the cursor is under the first character of a line in the Edit Mode.
PROC'D **(SHIFT V)**	**PROC'D** **(FCTN 6)**	Moves the cursor to the end of the line in the Edit Mode.
↑ **(SHIFT E)**	↑ **(FCTN E)**	■ Moves the cursor up one line in the Edit Mode. ■ Blackens a square on the MAKESHAPE and MAKECHAR grids as the cursor moves up one square.
← **(SHIFT S)**	← **(FCTN S)**	■ Moves the cursor left one space in the Edit Mode. ■ Blackens a square on the MAKESHAPE and MAKECHAR grids as the cursor moves left one square.
→ **(SHIFT D)**	→ **(FCTN D)**	■ Moves the cursor right one space in the Edit Mode. ■ Blackens a square on the MAKESHAPE and MAKECHAR grids as the cursor moves right one square.
↓ **(SHIFT X)**	↓ **(FCTN X)**	■ Moves the cursor down one line in the Edit Mode. ■ Blackens a square on the MAKESHAPE and MAKECHAR grids as the cursor moves down one square.
SPACE	**SPACE**	■ Leaves a blank space in the type in the Sprite and Turtle Modes. ■ Reviews file names in the Save and Recall Modes.
[**(SHIFT 4)**	] **(FCTN R OR SHIFT 4)**	Types a left bracket.

(Continued)

[**(SHIFT 5)**	] **(FCTN T OR SHIFT 5)**	Types a right bracket.
QUIT (SHIFT Q)	**QUIT (FCTN =)**	Stops TI LOGO and returns to the master title screen.

↑ ↑ ↑ ↑ ↑ ↑

INDEX

General Interest
Computer

The evolving family of computer languages known as Logo was envisioned by its designers as a virtually unlimited educational tool. Now a reality, Logo is simple enough for a young child to learn, yet sophisticated enough to teach adults many serious computer science concepts.

Written by one of the original developers of Logo at MIT, this easy-to-understand book:

- ★ Provides full coverage of all the features of TI Logo
- ★ Details the just-released implementation of Logo called TI Logo II, which features "sprite" graphics and music capabilities
- ★ Contains many sample programs

Hobbyists, educators, parents, and students can learn to make full and innovative use of their computer's teaching potential through chapters covering subjects such as defining procedures...animation...how to keep track of procedures and save them in files...how to write procedures that use data...the Logo music system... fine points of Logo syntax...and much more.

From primitive commands to advanced programming, this timely book will show how to transform the computer into a flexible tool to aid in learning, in playing, and in exploring.

About the Author

Harold Abelson is associate professor of electrical engineering and computer science and heads the Educational Computing Group at M.I.T., where he received his doctorate in 1973. His other books include *Calculus of Elementary Functions* (1970), *Turtle Geometry* (1981), *Apple Logo* (1982), and *Logo for the Apple II* (1982).

McGraw-Hill Book Company
Serving the Need for Knowledge
1221 Avenue of the Americas
New York, NY 10020

ISBN 0-07-038459-2