



The Terrapin  
Logo Language  
for the Apple II

**T**  
UTORIAL

The Terrapin  
Logo Language  
for the Apple II

**T**UTORIAL

*Written by  
Virginia Carter Grammer  
and  
E. Paul Goldenberg*

*Edited by  
Mark Eckenwiler  
and  
Peter von Mertens*

Please fill this out for your records:

Name \_\_\_\_\_

Address \_\_\_\_\_

\_\_\_\_\_

Telephone (     ) \_\_\_\_\_

Logo Language Disk # \_\_\_\_\_

Purchase Date \_\_\_\_\_

Store Bought \_\_\_\_\_

Store Telephone Number \_\_\_\_\_

Salesperson \_\_\_\_\_

Terrapin, Inc. gratefully acknowledges the writing and editing contributions to this tutorial by

Leigh Klotz, Jr.  
J. Sheridan McClees  
Nola Sheffer  
Patrick G. Sobalvarro  
Deborah G. Tatar  
Rena Upitis

Designed by Donna Albano and Janet Mumford

Terrapin Logo mascots designed by Virginia Grammer

Copyright © 1982, 1983 Terrapin, Inc.  
All Rights Reserved.

## ***IMPORTANT — PLEASE READ***

Welcome to Terrapin™ Logo! The Logo language is an exciting way of entering the world of computers. There are all kinds of fun things to do—you can create graphics, play word games, make music and learn how to write fascinating programs.

The items that should be included in this package are listed on page B-1. If you run into a problem, our customer service department is always here to help. Our address is: 376 Washington Street, Malden, MA 02148. You may call us Monday - Friday between 9:00 AM - 4:30 PM EST at: (617) 322-4800.

In the interest of giving Ma Bell and the Post Office less business, we suggest that you attempt to solve your problem by consulting the documentation. Many of the calls we get concern printing; the Printing chapter should answer most of your questions. The Index and Glossary of Logo Commands will also serve as handy resources for you.

In the event that you still require assistance, contact us and please be prepared to provide us with as much detailed information as possible about your particular situation—which version of the language you are using, the type of Apple® computer system you have (including printer and type of interface if you have a printing problem), exactly what the computer does or says when the problem occurs, whether or not the event is repeatable, and so on. The more information you can give us, the better able we will be to answer your questions and solve your problem.

### ***Backup and Replacement Policy***

As a registered owner, you are entitled to one backup of your Logo language disk. To obtain a backup copy, just

complete the enclosed warranty card and return it to Terrapin along with a check for \$10.00. You need not send in an order form—save it for ordering books. Owners of TerraPak-10's and TerraPak-20's are not eligible for backup Logo Language disks.

If at any time your disk fails to load, you can replace it for \$10.00. Simply return the damaged disk with your check. (If your Logo Language disk fails within the first ninety days, it will be replaced at no charge; simply return the faulty disk with proof of purchase.) Make sure to include your name and address. We suggest that all disks be sent via registered mail; we are not responsible for disks lost in transit to us.

It is important that you return your warranty card immediately, even if you do not want a backup disk at this time. (Be sure to keep a copy of your warranty card or proof of purchase to send along when you do want your backup disk.) Once you have sent in your warranty card, you will be placed on our mailing list so that you can receive a free subscription to the *Terrapin Times*, our newsletter of Logo-related news and information from across the nation and around the world. Through our mailings, you will also be kept informed about upgrades and new products as they become available.

In addition to offering a quality product and superior customer support, Terrapin also produces a variety of Logo resources to further enhance your use of the language. Please examine the Terrapin price list inserted in the binder's inside pocket. If it is missing, contact us and we'll be happy to send you another.

Happy adventures with Logo!

---

## ***DISCLAIMER OF ALL WARRANTIES AND LIABILITY***

---

This software product and the accompanying materials are sold "AS IS," without warranty as to their performance. The entire risk as to the quality and performance of the computer software program is assumed by the user. The user of this product shall be entitled to use the product for his/her own use, but shall not be entitled to sell or transfer reproductions of the product or accompanying materials to other parties in any way. Terrapin, Inc. reserves the right to make improvements in the product described in this manual at any time and without notice. Neither Terrapin, Inc. nor anyone else who has been involved in the creation and production of this product shall be liable for indirect, special or consequential damages resulting from use of this product.

### ***COPYRIGHT AND TRADEMARK NOTICES***

Portions of the Technical chapter and the Logo software are copyrighted by the Massachusetts Institute of Technology. The Tutorial and changes to the Technical chapter and Logo software are copyrighted by Terrapin, Inc.

It is against the law to copy, photocopy, reproduce, translate, or reduce to any electronic medium or any other medium, in whole or in part, the software and documentation in this package without prior written consent from Terrapin, Inc.

***Copyright © Massachusetts Institute of Technology, 1981.*** Except for the rights and materials reserved by others, the Publisher and Copyright owner hereby grant permission without charge to domestic persons of the United States and Canada for use of this work and related materials in the

United States and Canada after 1995. For conditions of use and permission to use materials contained herein or any part thereof for foreign publication in other than the English language, apply to the Copyright owner or publisher. Publication pursuant to any permission shall contain an acknowledgement of this copyright and an acknowledgement and disclaimer statement as follows:

This material was prepared with the support of National Science Foundation Grant No. SED-7919033.

However, any opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of NSF.

Each school purchasing and putting into use Logo will make the program object code and accompanying manuals and teaching guides, if any, available for inspection by the parents or guardians of the children who will be using Logo in the school.

***Copyright © 1982, 1983, 1988 Terrapin, Inc.  
376 Washington Street  
Malden, MA 02148  
(617) 322-4800***

Terrapin hereby grants permission in advance to copy the computer programs listed in the documentation and on the utilities disk, for personal and archival purposes only.

Terrapin expressly reserves all rights including without limitation copyright and trademark, in and to the Terrapin Logo mascot figures as represented herein, or from any other perspective.

Terrapin, the Terrapin turtle logo, Logo PLUS and the Terrapin mascots are trademarks of Terrapin, Inc.

Apple, Apple IIGS, ImageWriter, and ProDOS are registered trademarks of Apple Computer, Inc.

Grappler is a registered trademark of Orange Micro, Inc.

# CONTENTS

---

## Chapter Titles

<b>Beginning in Logo</b>	B-1 to B-16
<b>Graphics</b>	G-1 to G-91
<b>Computation: Handling Numbers</b>	C-1 to C-26
<b>Words and Lists</b>	W-1 to W-116
<b>Music</b>	M-1 to M-14
<b>Printing</b>	P-1 to P-14
<b>Utilities</b>	U-1 to U-46
<b>Technical</b>	T-1 to T-49
<b>Appendix</b>	A-1 to A-112
Error Messages	A-1
Edit Mode	A-14
Strategies for the Graphics Projects	A-18
Miscellaneous Tips	A-78
Strategies for the Words and Lists Projects	A-81
<b>Logo Command Glossary</b>	L-1 to L-28
<b>Index</b>	I-1 to I-12

## Table of Contents

### BEGINNING IN LOGO

Your Terrapin Logo Package .....	B-1
This Tutorial .....	B-2
Overview: What Can You Do With Logo? .....	B-3
Graphics .....	B-4
Computation .....	B-5
Words and Lists .....	B-5
Music .....	B-5

## Table of Contents

---

Before You Begin .....	B-5
Starting Logo .....	B-6
When Logo Has Started Up .....	B-7
Recovery Process .....	B-8
Using the Keyboard .....	B-9
Upper Case and Lower Case .....	B-11
Preparing a Blank Disk .....	B-11
Copying Disks .....	B-14
Starting Logo: Summary .....	B-16

## *GRAPHICS*

Graphics Mode .....	G-1
Driving the Turtle: FORWARD (FD), BACK (BK), RIGHT (RT), LEFT (LT) .....	G-2
Let Logo Do Your Arithmetic .....	G-5
An Easy Way to Repeat Yourself: <CTRL> P .....	G-5
The Screen: DRAW, NODRAW (ND), TEXTSCREEN (<CTRL> T), SPLITSCREEN (<CTRL> S), FULLSCREEN (<CTRL> F) .....	G-7
Turtle-driving Projects .....	G-8
Color: PENCOLOR (PC) and BACKGROUND (BG) .....	G-8
The Magic of PENCOLOR 6: Erasing .....	G-11
Introduction to Procedure Writing .....	G-12
Primitives vs. Procedures .....	G-12
Naming a Procedure .....	G-13
Writing a Procedure: EDIT Mode: TO, END, <CTRL> C, <CTRL> G .....	G-14
Running a Procedure .....	G-19
Planning and Drawing Your Favorite Square .....	G-21
Projects: Simple Procedures .....	G-25
What goes Into a Procedure .....	G-25

More Primitives: REPEAT, CLEARSCREEN (CS), HOME, PENUP (PU), PENDOWN (PD) .....	G-26
Procedure Projects .....	G-28
Saving Procedures: CATALOG, SAVE, POTS .....	G-29
Clearing the Workspace, Reloading Procedures: READ, GOODBYE, ERASE (ER), ERASE ALL (ER ALL), ERASEFILE .....	G-32
Selective Uses of SAVE, PO, ERASE (ER), and EDIT (ED) .	G-34
Saving, Reading and Erasing Pictures: SAVEPICT, READPICT, ERASEPICT .....	G-35
The Invisible Turtle: HIDETURTLE (HT), SHOWTURTLE (ST) .....	G-37
Summary of Logo Commands Used So Far .....	G-38
More About the Editor: Arrow Keys, <CTRL> P, <CTRL> N, <CTRL> O, <CTRL> A, <CTRL> E, <CTRL> D, <CTRL> X, <CTRL> Y .....	G-40
Summary of Editing Commands .....	G-41
Projects Using Shapes .....	G-42
Listing a Procedure: PRINTOUT (PO), <CTRL> W .....	G-43
Summary of Listing Commands .....	G-44
Heading: A Matter of State .....	G-44
Copying a Procedure .....	G-46
A Magic Number .....	G-46
Projects: More Shapes .....	G-48
Introduction to Variables: Procedures That Take Inputs .....	G-48
Projects: Sizable Shapes .....	G-53
From SQUARE to POLY .....	G-53
Projects: Regular Polygons .....	G-55
Another View of POLY .....	G-55
Circles .....	G-57
Projects: Curves .....	G-57
Using Subprocedures .....	G-58

## Table of Contents

---

Non-stop Procedures: Introduction to Recursion .....	G-61
Projects: Simple Recursion .....	G-62
Recursion: Changing the Input, WRAP, NOWRAP .....	G-62
Projects: Changing Inputs .....	G-65
Stopping With Style: IF-THEN, STOP .....	G-66
Projects: Testing and Stopping .....	G-69
Using the Full Power of Recursion .....	G-69
Recursion Projects .....	G-73
Special Effects and New Utilities .....	G-74
RANDOM Numbers, Numbers from Arithmetic	
Operations, Inputs, Outputs .....	G-76
Projects Using Random .....	G-78
Debugging by Printing Values: PRINT (PR) .....	G-79
Debugging Using PAUSE: <CTRL> Z, CONTINUE (CO) ..	G-81
Negative Inputs .....	G-81
More on Debugging: TRACE, NOTRACE .....	G-83
More About the Turtle: TURTLESTATE (TS),	
HEADING, SETHEADING (SETH), TOWARDS .....	G-83
Position When You Want It: XCOR, YCOR,	
SETX, SETY, SETXY .....	G-85
INSTANT: Logo Turtle Graphics for the Non-reader .....	G-87
Modifying INSTANT .....	G-91

## *COMPUTATION: HANDLING NUMBERS*

Arithmetic Operations .....	C-1
Hierarchy of Operations .....	C-2
Outputs, Integer Operators, Functions: RANDOM,	
RANDOMIZE, ROUND, INTEGER, QUOTIENT,	
REMAINDER, SQRT, SIN, COS .....	C-4
Variables, Global and Local: MAKE .....	C-6
Procedures: TO, END .....	C-8
Interactive Procedures: LOCAL, REQUEST (RQ) .....	C-10

Bringing Values Out of Procedures: OUTPUT (OP) ..... C-13  
 Example of OUTPUT and Recursion:  
     A Procedure to Do Exponentiation ..... C-15  
 Graphing Functions: Sine, Cosine, Tangent, Parabola,  
     Ellipse, SETXY, HOME, DRAW, HT ..... C-19

## **WORDS AND LISTS**

INTRODUCTION ..... W-1  
 Interactive Graphics: READCHARACTER (RC),  
     TOPLEVEL, STOP ..... W-4  
 Projects with RC: Extending QUICKDRAW ..... W-8  
 Changing the Value of a Variable: MAKE, PRINT (PR) .... W-9  
 Projects with MAKE: More Extensions to QUICKDRAW ... W-17  
 Programs that Interact without Waiting: RC? ..... W-18  
 Projects with RC, RC?: Extensions to LOOP ..... W-22

### **INTERACTIVE LANGUAGE**

Don't Skip This Section! MEMBER?, EMPTY? ..... W-23  
 Some Friendly Introductions: SENTENCE (SE),  
     REQUEST (RQ), LPUT, FPUT ..... W-25

Interlude: Clearing the Text Screen with CLEARTEXT .... W-32

Objects: Producing RESULTs as Output,  
     and Using Them as Input ..... W-33

Writing Procedures that Create and Output Objects:  
     OUTPUT ..... W-36

Making One Procedure's Output into Another Procedure's  
     Input: OUTPUT (OP), FIRST, BUTFIRST (BF), LAST,  
     BUTLAST (BL), SENTENCE (SE), WORD ..... W-43

Subprocedures for Cleaner Programming ..... W-46

A Generalization Using Recursion: ITEM ..... W-48

Projects Using ITEM and Recursion ..... W-51

### **DEFINITIONS AND MODELS**

Some Important Primitives Used in this Chapter ..... W-52

Definitions of Words and Lists: CHAR ..... W-56

## Table of Contents

---

Details of Programming in Logo: Variables, Passing Objects, Logo's Way of Understanding Commands, and Logo's Messages When It Doesn't Understand . . . . .	W-60
How Logo Interprets a Command . . . . .	W-67
Using Logo Predicates and Creating New Ones: LIST?, WORD?, MEMBER?, and the Structure of IF, THEN, and ELSE. . . . .	W-70
Projects with Predicates . . . . .	W-74
Ordered Rules . . . . .	W-75
Projects with PLURAL . . . . .	W-79
Quiz Programs: More About REQUEST (RQ) . . . . .	W-81
Projects with REQUEST . . . . .	W-84
Composing Logo Objects: SENTENCE (SE), WORD, LIST, FPUT, LPUT, TEST, IFTRUE (IFT), IFFALSE (IFF) . . . . .	W-86
An Application of LPUT in Interactive Graphics: RUN . . . . .	W-94
Using the History List: Applying a Command (RUN) to Each Element of a List . . . . .	W-96
Projects with History Lists . . . . .	W-99
Substituting One Word for Another in a Sentence: A Procedure with Two Recursive Calls . . . . .	W-100
Projects with Mad-Libs . . . . .	W-105
Understanding Language: Searching for Key Words and Matching Sentences to Templates: ALLOF, ANYOF . . . . .	W-106
Projects with Language Understanding . . . . .	W-115

## *MUSIC*

Preparation: READ . . . . .	M-1
Duration . . . . .	M-2
Pitch. . . . .	M-5
Procedures . . . . .	M-7
Analyses of the Utilities Disk Music Procedures: STOP, FIRST, BUTFIRST (BF), THING, WORD, Top Level . . . . .	M-11

## *PRINTING*

Printing Text .....	P-1
Printing Pictures .....	P-2
Printing Logo Graphics Using Other Programs .....	P-3
Using an ImageWriter Printer .....	P-3
The PRINTSCREEN Command ( <i>Logo PLUS</i> only) .....	P-4
PRINTPICT and LARGEPICT .....	P-5
SMALLCOLOR and LARGECOLOR.....	P-7
Other ImageWriter Printing Tips .....	P-8
SCRIBE: Using a Scribe Printer .....	P-10
Printing to a Parallel Printer .....	P-11
SCREENDUMP: Using a Grappier™ Interface .....	P-11
SWEET-P: Plotter Procedures .....	P-12
RSPLOTTER: Procedures for the Radio Shack Plotter .....	P-14

## *UTILITIES*

About the Utilities Programs .....	U-1
The <i>Logo PLUS</i> Utilities Disk .....	U-1
To Use the Utilities Disk Files.....	U-2
Summary of Utilities Disk Files .....	U-4
Explanation of Utilities Disk Files .....	U-9
Aids to Using Logo	
INSTANT: Single Letter Logo Commands .....	U-9
TMOVE: Another Way to Move the Turtle .....	U-10
TEACH: How to Write Logo Procedures Without Using the Editor.....	U-11
Music System Files	
MUSIC: How to Write and Run Logo Music Procedures .....	U-13
TWINKLE: A Sample Melody.....	U-13

## Table of Contents

---

MUSIC.SRC, MUSIC. BIN: An example of Logo/Assembler Interfacing .....	U-13
Shape Editing Programs	
SHAPE.EDIT: How to Change the Shape of the Turtle .....	U-14
The Logo Shape Editor .....	U-14
ROCKET, ROCKET.AUX, ROCKET.SHAPES: Example of User-Defined Turtle Shapes .....	U-22
Demonstration Programs	
ANIMAL: A Game that Teaches the Computer About Animals .....	U-23
ANIMAL.INSPECTOR: What's in the ANIMAL Knowledge Base? .....	U-24
DYNATRACK: A Game: the Dynamic Turtle on a Frictionless Surface .....	U-25
INSPI.PICT: Sample Logo Picture .....	U-26
TET: A Graphics Procedure of Variable Complexity .....	U-26
Useful Tools	
ARCS: Variable Radii Arc and Circle Procedures .....	U-27
BEFORE: Comparing Words .....	U-28
CURSOR: Procedures for Character Output Control: Position, Flashing, Inverse .....	U-30
PICK: Choosing Things at Random .....	U-31
PPRINT: Word Wrap for Long Lines .....	U-32
STRING: Manipulating Word Segments .....	U-33
WAIT: Causing Delays in your Programs .....	U-35
File-Handling Utilities	
BIN.TO.TEXT: Converting Binary Files to Text Files .....	U-36
FID: File Management Utility: How to Delete, Rename, Lock, and Unlock Files, Set Default File Extension .....	U-37
TEXTEDIT: How to Save, Read, Examine, and Print Files Containing Text .....	U-38
DPRINT: Printing Text Into Disk Files .....	U-40

Logo Files for Assembly Language Interfacing .....	U-43
ADDRESSES, AMODES, ASSEMBLER, OPCODES:	
Interfacing Logo and the Assembler .....	U-43
Useful and Interesting Memory Locations .....	U-43

## *TECHNICAL*

Use of the Logo System .....	T-1
Modes of Using the Screen .....	T-1
Non-Editing Control Characters .....	T-4
Editing .....	T-5
Screen Editing Commands .....	T-8
Using Apple Peripherals .....	T-11
Color Control .....	T-11
The Logo File System .....	T-14
Changing the Turtle Shape .....	T-17
Building Your Own Shape Tables .....	T-17
Assembly Language Interfaces to Logo .....	T-18
.EXAMINE and .DEPOSIT .....	T-19
Writing Your Own Machine-Language Routines .....	T-20
The Logo Assembler .....	T-23
Example: Generating Music .....	T-30
Useful Memory Addresses .....	T-36
Miscellaneous Information .....	T-42
Using the Logo System as a Text Editor .....	T-42
Self-Starting Files .....	T-44
Various System Parameters .....	T-45

## *APPENDIX*

### ERROR MESSAGES

Part I .....	A-1
Part II .....	A-5

## Table of Contents

---

### EDIT MODE

Use of Control Characters for Ease in Editing .....	A-14
Moving the Cursor .....	A-14
Moving the Text .....	A-15
Deleting Text .....	A-16
Restoring Text .....	A-16
Leaving EDIT Mode .....	A-17

### STRATEGIES FOR THE GRAPHICS PROJECTS

Turtle Driving Projects .....	A-18
Procedure Projects .....	A-21
Projects Using Shapes .....	A-24
Projects: More Shapes .....	A-38
Projects: Sizable Shapes .....	A-39
Projects with Regular Polygons .....	A-41
Projects: Curves .....	A-44
Projects: Simple Recursion .....	A-47
Projects: Changing Inputs .....	A-49
Projects: Testing and Stopping .....	A-53
Recursion Projects .....	A-55
Projects Using Random .....	A-65
Mascots: Elephant, Rabbit, Snail .....	A-68
Procedures for Saving Pictures .....	A-72
Developing an Arc Procedure .....	A-75

### MISCELLANEOUS TIPS

Debugging with TRACE, NOTRACE .....	A-78
Adding Remarks in Your Procedures .....	A-79
Switching Disk Drives: SETDISK .....	A-79
Creating Self-Starting Files	
Using the STARTUP Variable .....	A-80
Reading Apple Logo Files .....	A-80

### STRATEGIES FOR THE WORDS

AND LISTS PROJECTS .....	A-81
--------------------------	------

## *LOGO COMMAND GLOSSARY*

## *INDEX*

*B*

*B*

*B*

*B*

*B*

*B*

***B*** ***BEGINNING IN LOGO***

*B*

# B

## BEGINNING IN LOGO

---

### *Your Terrapin Logo Package*

**NOTE:** This section should be read the first time you use your Logo package. If you have used *Terrapin Logo* before, or have a resource person or teacher helping you, skip to the next section, titled This Tutorial.

In your *Terrapin Logo for the Apple* package, you will find:

- 1 Logo Language disk
- 1 Utilities Disk containing demonstration and utility programs
- 1 Tutorial (which you are reading) to help you learn Logo. It will provide you with complete instructions for printing and using the utilities programs, technical information, a Glossary and an Index.
- 1 Quick Reference Card of Logo commands
- 1 Logo Project Card

In the *Logo PLUS* package, you will find all of the above as well as a *Getting Acquainted with Logo PLUS* booklet. Note that a 3.5" *Logo PLUS* disk includes both the Logo Language and Utilities programs.

In order to use Logo, you will also need:

- An Apple II family (Apple II+, IIe, IIc or IIGS) or Apple-compatible computer
  - 64K of memory is required for *Terrapin Logo*
  - 128K of memory is required for *Logo PLUS*
- One or more disk drives
- To save your work, you will need a blank disk.

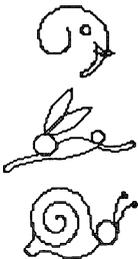
Before using your Utilities Disk, you should make a backup copy because it is possible to damage or erase the Utilities Disk accidentally. Instructions for copying a disk are at the end of this chapter.

### *This Tutorial*

This tutorial will teach you how to use Logo. It is divided into chapters that introduce each of the areas you will want to explore. The Utilities chapter explains the use of the sample and utility programs that are on the Utilities Disk. The Technical chapter contains information about assembly language interfaces for Logo and the internal workings of Logo. You need not read it to start using Logo, but you will find it useful when you are ready for new challenges.

Once you are comfortable with your Apple computer, use this tutorial to learn the basics of programming in Logo. Type in the examples and problems. Think about what you are doing; expect to go over some sections more than once.

Logo puts the user in control from the start. In keeping with that philosophy, this tutorial will suggest but not dictate. If you are ever really stuck for an idea, see the Appendix. It contains examples of all the ideas suggested. In fact, after you try things on your own, look through the Appendix for new ideas, tips and tricks.



In this tutorial you will meet three Logo mascots, all drawn with Logo. The elephant marks things to remember. The rabbit points out neat tricks, short cuts, and quicker ways of doing things. Go slow and be careful when you see the snail. It calls attention to warnings and possible problems. The procedures that draw the mascots are listed in the Appendix.

We have occasionally put some information between pairs of colored bands on the page. It is not necessary to read this information your first time through the tutorial, but you will find it helpful when you return and want further explanations of specific sections.

### *Overview: What Can You Do with Logo?*

Logo is a procedural language. Each procedure is a group of one or more instructions, which the computer can store for reuse. These instructions can be either Logo commands or procedure names. When you have written a procedure to do a task, you can use it in any other procedure you write, without having to rewrite its instructions in that procedure, chain to it, or link it, as you would with other languages.

Logo's strong appeal to schools derives from this structure as well as its underlying educational philosophy. Its simple and open environment encourages a discovery approach to learning. Logo actively involves the student in the learning process, so that the student teaches the computer. Educators attest to the fact that students who use Logo gain a new excitement about learning. The Logo language gives them an opportunity to explore problem-solving, creativity, higher order thinking skills and mathematical reasoning in a unique way.

Terrapin offers educators not only the Logo language for a variety of microcomputers, but also a series of curriculum materials designed to help teachers integrate Logo into their standard curriculum. With products such as *Logo Works: Lessons in Logo*, *The Logo Project Book*, *Logo Data Toolkit* and *Logo Probability*, students can explore math, language, social studies and science using Logo as a tool.

In Logo, you build a system of procedures the way you build your own knowledge base, with new procedures and knowledge building on what already exists. This leads to clearer, more structured thinking and programming, in contrast to the development of one long, complicated procedure (program) which is common in some other languages.

Logo is what is known as an interpreted language. Logo commands produce immediate results. Logo can either execute a command immediately (called IMMEDIATE Mode) or you can use commands in procedures, which can be stored and used as often as you want. Changing or correcting (editing) a procedure is simple in Logo.

If you are familiar with other languages, you will be delighted with the lack of distinction between system commands, Logo commands, and procedures. This is perhaps the most unusual aspect of Logo, and one of the most powerful, from the user's standpoint. Any command you can type to Logo can be used within a Logo procedure, and Logo procedures can even be written to edit themselves.

You can begin to use all of the different types of commands immediately. As you advance in your programming skills, you will gradually discover the vast possibilities Logo offers you.

### *Graphics*

Using Logo graphics, you can draw lines and turn in any direction. With its simple commands you may create figures and drawings of great complexity. In Logo, you do not have the tedious task of figuring point to point coordinates, although Logo can tell you the coordinates at any position.

An introduction to Logo graphics comes first in this tutorial because you need no experience to be able to use these commands. Using the single-letter commands in the INSTANT system, pre-schoolers can create Logo graphics. At the other end of the intellectual spectrum, university professors use Logo to introduce computer science and develop concepts in higher mathematics and physics.

### *Computation*

In addition to the ordinary mathematical computations all languages can handle, Logo's built-in ability to do recursion, which allows a procedure to use itself as a subprocedure, makes it easy to do computations not possible in languages such as BASIC and FORTRAN. You will meet recursion in each of the areas of Logo described in this overview. For a description of mathematical computation, see the chapter titled *Computation: Handling Numbers*.

### *Words and Lists*

Logo's facility with words and lists makes it ideal for writing conversational programs, quizzes, pig-Latin translators, programs that teach, and even programs that learn: in short, all programs that need to manipulate lists of information.

Logo's unique list-processing capabilities give you power over words that is impossible to match in non-list-processing languages such as BASIC, FORTRAN, and Pascal. See the chapter titled *Words and Lists* for what Logo can do and what you can do with it.

### *Music*

Logo makes it easy for you to write tunes and pieces of tunes, or play games with pitch, time, and sequencing of phrases. You can even define your own musical scales, using Logo commands as building blocks. See the chapter titled *Music* for details.

### *Before You Begin*

It is possible to run Logo without a storage or data disk in the disk drive, but you would not be able to save your

work. We encourage you to prepare a blank disk for storing the programs you will be writing.

A blank disk, unlike an audio cassette tape, must be prepared before it can store information. This process is called initializing (or formatting) the disk. Follow the instructions at the end of this chapter for formatting a blank disk.

After the disk has been formatted, remove it from the disk drive, label it immediately, and use it to store your Logo procedures. We will refer to it again in the section When Logo Has Started Up later in this chapter.

### *Starting Logo*

One of the disks packaged with your system is called the Language Disk. It is the disk with the Logo language on it. The other disk, labeled Utilities Disk contains some demonstration and utility programs. They are mentioned where appropriate throughout the tutorial and discussed in detail in the Utilities chapter.

With the Apple turned off,

1. Place the Language Disk in your disk drive with the label facing up and closest to the front.
2. Close the disk drive door firmly.
3. Turn on the monitor.
4. Turn on the Apple. The on-off switch is on the back at the left as you face the machine.

The computer will display information on the screen and the disk drive light will go on. It takes about 20 seconds to load and start Logo. (Since *Logo PLUS* has many additional features, it takes a bit longer to load

than *Terrapin Logo for the Apple*.) If Logo does not seem to load properly, check to be sure that you are using the Language Disk and that the disk drive door is firmly closed.

When it has started, Logo will display the greeting

WELCOME TO LOGO

If Logo does not start up after about one minute, your Language Disk may be damaged in some way, or your disk drive may need adjusting. If other disks work on your disk drive, the problem is most likely with your Language Disk or a card you may have installed in your computer that contains extra memory. If you do not have sufficient memory in your computer to load Logo, you will see a message to that effect.

### *When Logo Has Started Up*

Logo will print its WELCOME TO LOGO message and a ? when it is ready for you. The ? is called a prompt, prompting you to respond with a Logo command. The flashing box is called the cursor. It shows you where the next character you type will appear. Whenever the cursor is flashing, Logo is waiting for you to type something.



(This would be a good time to remove the Logo Language Disk from the disk drive, put it in a safe place, and replace it with the blank disk you have initialized and will be using to store your Logo procedures.)



You give Logo directions by typing commands at the keyboard. Logo reads what you have typed when you press the <RETURN> key. Pressing <RETURN> is like saying "Do it." Nothing will happen until you press <RETURN>.



**NOTE ON POINTED BRACKETS:** When you see pointed brackets < > around a word, press the key on the keyboard with that word on it. Do not spell out the word. When you see <CTRL> C, hold down the <CTRL> key and type the letter C. (The abbreviation <CTRL> is used for the <Control> key.) Think of the <CTRL> key as a different kind of <SHIFT> key.

**SPECIAL NOTE:** Nothing you type can harm the computer or Logo. Even the worst that can happen is not too bad: pressing the <CTRL> and <RESET> keys at the same time while using Logo may take you out of Logo and mean the loss of work you have not yet stored, but it will not harm Logo or the computer. Much of the time you can recover your work after an accidental reset (see below). Don't be afraid to try things.

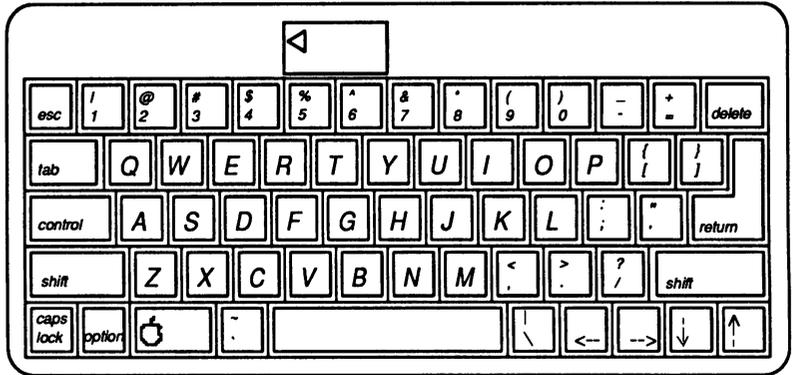
### *Recovery Process*



To recover from a system reset or a Logo bug that leaves you with an asterisk (\*) at the bottom of the screen, first type <CTRL> Y <RETURN> (hold down the <CTRL> key and tap <Y>, then press <RETURN>). Then type <CTRL> G <RETURN>. Usually this will put you back into Logo. If it does not, turn the machine off and start Logo according to the four-step Starting Logo: Summary at the end of this chapter.

When Logo does not understand something you have typed, it will try to help you by typing out a message. Most of the time you will have no trouble figuring out what is wrong, but when you need assistance, turn to the list of Error Messages and their explanations (with examples) in the Appendix.

## KEYBOARD DIAGRAM



The Apple IIGS Keyboard

*Using the Keyboard*

**BEFORE YOU BEGIN:** If you are using an Apple IIe, IIc, or IIGS, the <ESC> and <DELETE> keys can both be used for deleting backwards, one character at a time. If you have an Apple II or II+, you will delete backwards using the <ESC> key, which we will call <DEL> from now on. In addition, you will need to use <SHIFT> N to print a left bracket ([), and <SHIFT> M to print a right bracket (]).

Use the following editing keys to correct typing errors. The <DEL> key moves the cursor to the left and erases the character there. The left- and right-arrow keys move the cursor in the direction they point on the keyboard. Any letter, number, or symbol that you type will appear exactly where the cursor is blinking, even if you have used the arrow keys to move the cursor back into the text. The letters under and after the cursor will move to the right to make room. Try typing this line:

MARY HAD A LITTLE LAMB

Use the <DEL> (or <ESC>) key to erase the last character. Try it a few times. Move the cursor back several letters using the left-arrow key. Notice that this does not erase the letters it travels over. Change the line to read:

```
GARY HAD A LITTLE LAMB
GARY HAD A LITTLE HAM
GERTA HAD A LITTLE HAM SOUP
GERTA HAD XVP26 A LITTLE HAM SOUP
```

Finally, change it back to

```
MARY HAD A LITTLE LAMB
```

See how typing characters in the middle of the line makes the rest of the line move over to make room? You can never accidentally type on top of other characters and cause them to be erased.

Press <RETURN> now. Logo will try to understand the whole line as a series of commands. Since the words MARY HAD A LITTLE LAMB are not Logo commands, Logo will tell you so. Type MARY again.



Tell Logo to ignore what is typed with <CTRL> G (before you press <RETURN>). To do this, hold down the <CTRL> key and press the <G> key. (Remember, the <CTRL> key is like a special <SHIFT> key which is always used with another key.) Logo will print STOPPED! and a new prompt. Typing <CTRL> G is the usual way to stop whatever Logo is doing.



**CAUTION:** At any time, you can exit Logo by turning the machine off; however, by doing so, you will lose all your work unless you have saved it on the disk. You are also likely to lose your work if you press the <RESET> key and have to restart Logo. Use <CTRL> G to stop

programs; stay away from the <RESET> key. But be sure to try the recovery process outlined above if you do press <RESET>.

### *Upper Case and Lower Case*

The Apple IIe, IIc and IIGS, unlike the Apple II and II+, come with the built-in ability to type both upper- and lower-case characters. You can instruct Logo to print text in both upper and lower case on the screen or on paper.



*Terrapin Logo* commands must be entered in upper case. Be sure to put the <CAPS LOCK> key (in the lower left corner of the keyboard) in the down position.

If you are using *Logo PLUS*, however, you can type commands in either upper or lower case. *Logo PLUS* will then convert the commands you enter in lower case into upper case for its own use.

### *Preparing a Blank Disk*

In order for a disk to be able to store information, it must first be formatted or initialized. Once the disk is formatted, the computer will be able to read files from it into the computer's memory, save files to it, display a listing of the files that are on it, and so on.

Think of the formatting process as similar to that of painting lines on a newly-paved parking lot. Each different brand of computer or operating system needs to have lines painted in a different way because each stores information in a unique way. During the formatting process, parking lines are painted on the disk, so to speak, so that the computer can "park" information in an organized way.

*Terrapin Logo* is based on the DOS 3.3 operating system and *Logo PLUS* is based on the ProDOS operating system. Thus, the method of formatting disks differs between the versions. The following two sections explain how to format a disk for use with each version of the Logo language.

Make sure that a blank disk is in the disk drive when the formatting process begins. When a disk is formatted, any information that exists on the disk is permanently erased. You can reuse a disk by formatting it again.

Note that you can transfer files from DOS 3.3 disks to ProDOS disks using the Convert program (described in the *Getting Acquainted with Logo PLUS* booklet) or the ProDOS User's Disk that came with your computer.

### *Formatting Disks for Logo PLUS*

1. Place the *Logo PLUS* Language Disk in the disk drive and turn on the computer.
2. When the turtle picture is displayed on the screen, hold down the <Closed-Apple> or <Option> key on your keyboard until you see the message "Loading accessories...." After a few moments, you will see the Accessories Menu on the screen.
3. Press 1 to select the first option, 1 - Format a disk. Then you will enter the slot and drive of the disk you want to format. Press <RETURN> if your disk drive is connected to slot 6 inside the computer; otherwise enter the appropriate slot number.

Press <RETURN> to accept 1 as the drive where the blank disk will be placed; otherwise, press the appropriate drive number.

4. Place a blank disk in the drive you have specified.
5. Type a new disk (or volume) name that is 15 characters or less.
6. The disk in the drive you have selected will now be formatted.

### *Formatting Disks For Terrapin Logo*

To initialize a blank disk, you can use your *Terrapin Logo for the Apple Utilities Disk* or any other disk that has already been formatted.

1. Place the *Terrapin Logo Utilities* disk in the disk drive and then turn on the computer.
2. When the red light on the disk drive goes out, remove the Utilities disk from the disk drive and put it away in a safe place for future use.

You can ignore the following message, which is displayed on the screen:

TERRAPIN LOGO FILES DISKETTE  
THIS DISKETTE CONTAINS LOGO  
PROGRAMS.

YOU MUST BE RUNNING LOGO ON  
YOUR APPLE TO USE THESE PROGRAMS.

3. Place the blank disk you want to initialize into the disk drive.



**WARNING:** Be sure to remove the Utilities Disk from the disk drive and replace it with a blank disk before proceeding. When you type the INIT HELLO instruction, the disk in the disk drive will be erased.

**DO NOT TYPE INIT HELLO with your Utilities Disk still in the disk drive.**

4. With the blank disk in the disk drive, type

INIT HELLO

and press the <RETURN> key. The disk drive will whir for almost a minute, then the Apple prompt (]) will appear on the screen and the light will go out on the disk drive. Your disk is now initialized.

Formatting Tips:

You will probably find it convenient to format a number of disks at the same time, no matter which method you are using. Be sure to label the disk as soon as you have formatted it. For ProDOS disks, include the volume name on the label for easy reference.

### *Copying Disks*

It is wise to make copies of your Utilities Disk and the disks on which you store your programs. Disks can fail for any number of reasons (dirt, magnets, heat, old age, etc.), and when they fail, it is typically without warning. Although disk failure is not a common occurrence, you should be prepared for the event.

Your Logo language disk can also fail. If it does so within the first ninety days after purchase, Terrapin will replace it without charge. Replacement disks after the initial time period are \$10.00. One backup disk is also available to you for \$10.00 and proof of purchase (warranty card or sales slip).

You can use either of the following methods to copy DOS 3.3 disks (for use with *Terrapin Logo*) or ProDOS disks (for use with *Logo PLUS*). Both programs will copy either type of disk.

*Method #1 - Using the ProDOS User's Disk*

1. Place the ProDOS User's Disk in the disk drive and turn on the computer.
2. At the opening menu, press <F> to select PRODOS FILE (UTILITIES).
3. Then press <V> to select VOLUME COMMANDS.
4. Finally, press <C> to select COPY A VOLUME.
5. Enter numbers for the slot and drive of the disk to copy and the slot and drive of the blank disk. (Press <RETURN> at these prompts to keep the number that the computer is already displaying.)
6. Press <RETURN> to allow the computer to read the volume name from the disk you are copying. (DOS 3.3 disks will have the volume name NON-PRODOS DISK.) Press <RETURN> to keep the volume name that is displayed or enter a new volume name. The copy will now be made. If you have just one disk drive, be prepared to swap disks in the disk drive a number of times.

*Method #2 - Using the DOS 3.3 System Master*

1. Place the DOS 3.3 System Master in the disk drive and turn on the computer. You will see a ] prompt sign on the screen.
2. Type RUN COPYA and press <RETURN> to load and start the copy program. Make sure that you type this instruction in upper case letters.
3. Enter numbers for the slot and drive of the disk to copy (original) and the slot and drive of the blank

disk (duplicate). (Pressing <RETURN> will keep the number that the computer is already displaying.)

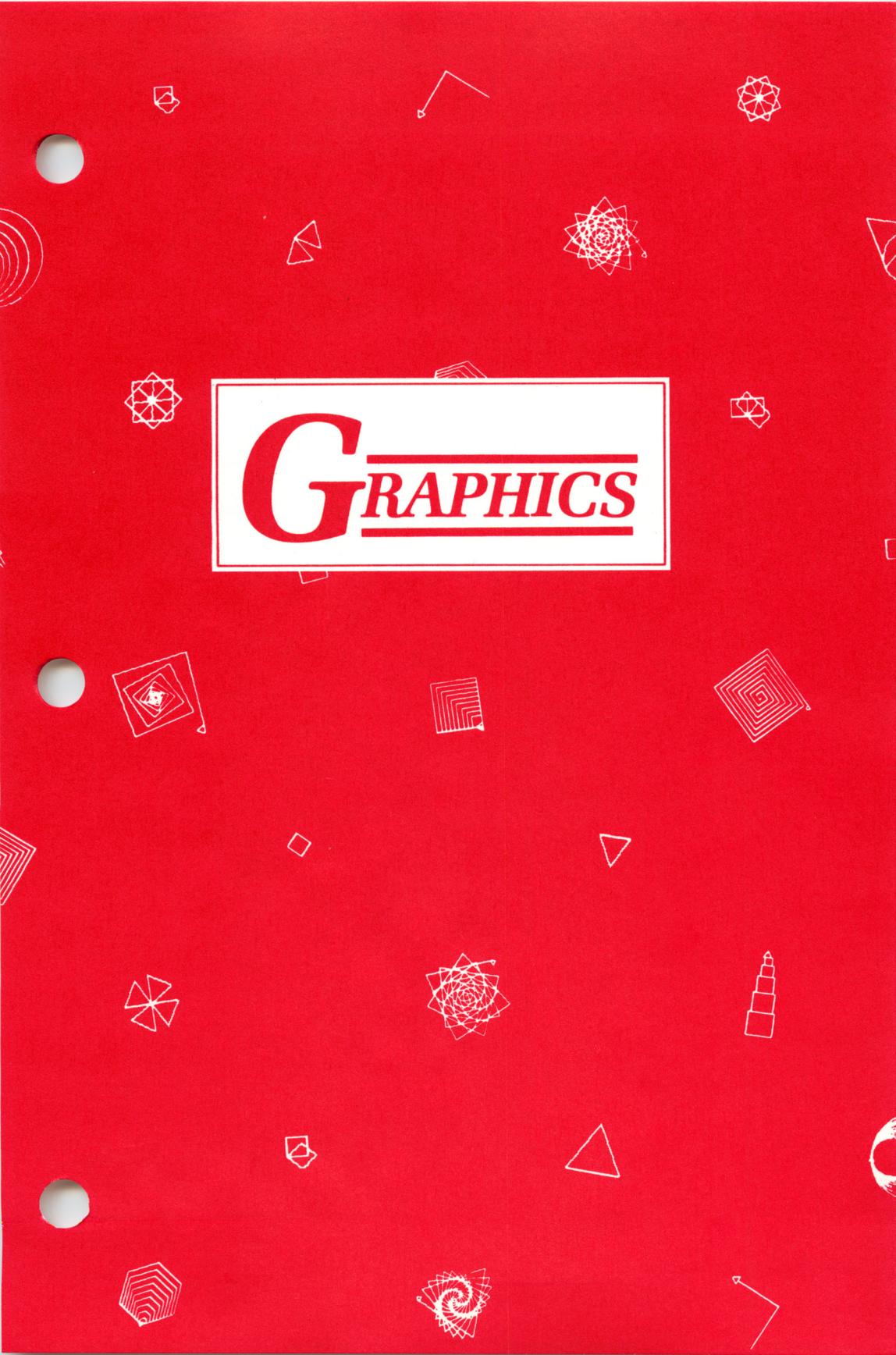
4. Press <RETURN> to begin the copy. If you have only one disk drive, you will swap disks in the disk drive several times before your copy is complete. Pay attention to which disk should be in the disk drive at all times.

#### Using Either Method:

These methods first format the blank disk and then copy the contents of the original disk to the blank disk. Be sure to make a label for your new disk, and include the volume name of ProDOS disks for easy reference in the future.

#### STARTING LOGO: SUMMARY

1. Place the Language Disk in the disk drive.
2. Turn on the Apple and wait approximately 20 seconds while Logo is loaded. If your computer is already on, hold down the <CTRL> and <Open-Apple> keys and press the <RESET> key to restart the computer.
3. After you see the WELCOME TO LOGO message, remove the Language Disk and insert a disk that has been formatted, so that you can save your work.
4. You are ready to proceed with Logo.



**G** *GRAPHICS*

# GRAPHICS

## G

---

Since this tutorial is written for our reading constituency, we have placed the section describing INSTANT for non-reading users at the end of the Graphics chapter.

---

Logo puts the user in control from the start. In keeping with that philosophy, this tutorial will suggest but not dictate. If you are ever really stuck for an idea, see the Procedures section of the Appendix. It contains examples of all the ideas suggested. In fact, after you try things on your own, look through the Appendix for new ideas and tips and tricks.

### *Graphics Mode*

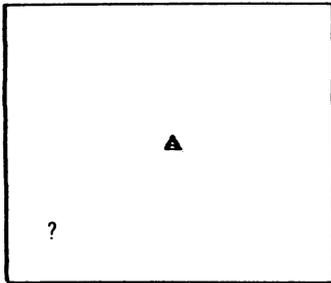
Enter the graphics or DRAW mode by typing DRAW:

DRAW

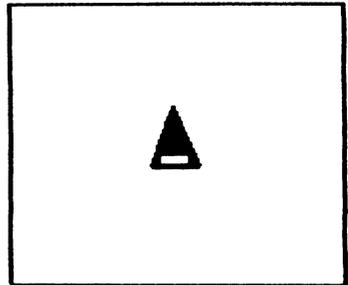
and press the <RETURN> key. (Remember, pointed brackets around a word refer to a key, not a word to be typed.)

A drastic change occurs on the screen; the command you have just typed and all other commands will disappear. A small triangle will appear in the middle of the screen, and the prompt will be in the lower left region of the screen.

Logo is now in DRAW mode. The bottom four lines of the screen are reserved for commands you will type and the rest of the screen is drawing space.



Splitscreen and Turtle

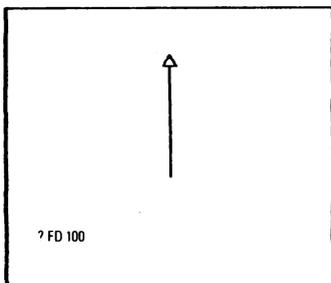


Turtle Enlarged

The small triangle in the middle of your screen is called the turtle. When it first appears, it is pointing upward. You can tell where it is heading by the black bar that runs across its back.

*Driving the Turtle: FORWARD (FD),  
BACK (BK), RIGHT (RT), LEFT (LT)*

You move the turtle with turtle commands. The turtle can leave a trail as it moves, allowing you to produce a picture.



**FORWARD** always moves the turtle in the direction it is pointed. Type

```
FORWARD 100 <RETURN>
```

or the short equivalent

```
FD 100 <RETURN>
```

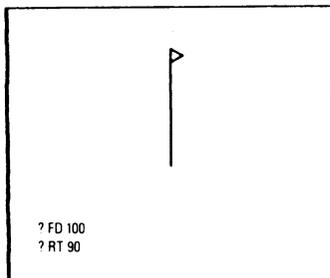


The turtle will move forward one hundred turtle steps. The space between the command and the number is necessary. If omitted, Logo will assume the whole thing to be a procedure name. (Try FD100 without the space.)

If you leave out the number that **FORWARD** is expecting, or the space, or do something else that Logo does not recognize, Logo will try to help you by printing an error message. These are usually self-explanatory, but if you cannot figure out what is wrong, turn to the Appendix where error messages are interpreted with examples.

To make the turtle turn, type the direction of the turn and the number of degrees:

```
RIGHT 90 <RETURN> or RT 90 <RETURN>
```



You told the turtle to turn right 90 degrees (a quarter of a circle). If you type RIGHT 90 again, the turtle will point straight down.

Type

LEFT 90 <RETURN> or LT 90 <RETURN>

From now on, we'll assume you know to press the <RETURN> key after a command.

The turtle will turn in place 90 degrees to its left. Try moving the turtle around yourself. Type BACK (or BK) with a number of steps.



To clear the screen and start over, type DRAW. DRAW erases whatever picture is on the screen and takes the turtle to its starting position. Use DRAW whenever you want to start a new picture.

Play with the turtle some more.

(1) Try some odd distances and turns, such as

```
FD 87
RT 43
FD 26
LT 141
FD 59
```

(2) Draw a square

(3) Try a triangle

Get familiar with the turtle commands. Use the commands or their abbreviations:

Command	Abbreviation
FORWARD	FD
BACK	BK
RIGHT	RT
LEFT	LT

### *Let Logo Do Your Arithmetic*



Whenever Logo expects a number (we call this number its input), you can give it an arithmetic expression to evaluate to get a number. Logo will do the arithmetic for you.

Type                      and Logo figures

FD 10 * 5	FD 50
RT 100/3	RT 33.3333 ...
FD 5 + 5	FD 10

This is useful for both accuracy and precision: the computer will not make a mistake, and the computer will make a division like 100/3 quite precisely.

### *An Easy Way to Repeat Yourself: <CTRL> P*

You can put as many commands on the same line as you want, as long as you separate them with spaces. When you have typed a line and pressed <RETURN> ,

Logo will repeat the line for you if you press <CTRL> P. (Hold down the <CTRL> key and press the <P>). Type



FD 50 RT 30 FD 20 RT 115<RETURN>



Logo draws the line. Type

<CTRL> P      Logo types



FD 50 RT 30 FD 20 RT 115



You press <RETURN> to do it.



Type <CTRL> P <RETURN> as many times as you wish; each time Logo will print and execute the line.

If you put a space at the end of your original instruction, you may also type

<CTRL> P <CTRL> P <RETURN>



This will print out two sets of your instructions. You can repeat the <CTRL> P as many times as you wish, up to 129 characters (9 characters more than 3 lines), as long as there are spaces between the commands. If you have no space at the end of the line, and type <CTRL> P twice, you will get

FD 50 RT 30 FD 20 RT 115FD 50 RT 30 FD 20 RT 115

If there is no space at the end of the line when you type another <CTRL> P (as in the line above), the last com-

mand of the first batch will not be separated from the first command of the second, and Logo will stop and say

THERE IS NO PROCEDURE NAMED 115FD

You can add a space after you type the <CTRL> P, but an easier way to insure a space is to put it there when you type the line (RT 115 <SPACE> <RETURN>).

***The Screen: DRAW, NODRAW (ND), TEXTSCREEN (<CTRL> T), SPLITSCREEN (<CTRL> S), FULLSCREEN (<CTRL> F)***

When Logo is in DRAW mode, the Apple displays four lines of text at the bottom of the screen. To see the commands you have typed that have disappeared under the picture, type

TEXTSCREEN or <CTRL> T

Remember that you must hold the <CTRL> key down while you type the T.

Try typing

<CTRL> T

To get back the split graphics/text screen, type

SPLITSCREEN or <CTRL> S

To show off your drawing without the distracting text, type

FULLSCREEN or <CTRL> F

<CTRL> S will bring back the split screen from either the text or fullscreen.

---

To clear the screen and leave DRAW mode, type NODRAW, abbreviated ND. Type ND <RETURN> right now.  
Type DRAW again to do some graphics projects.

---

### *Turtle-driving Projects*

1. Determine how many turtle steps it takes to get to the top edge of the screen.
2. Determine how many turtle steps from the bottom edge of the screen to the top. From the left edge to the right.
3. (Tricky one) How many steps from the lower left corner of the split screen to the upper right corner?
4. (Trickier still) How many from the lower left corner of the full screen to the upper right?
5. Try each of the commands with a negative number. (Example: FORWARD -100) How else could the turtle make the same move?
6. Can you draw a square? A rectangle?
7. Can you draw your initials?

### *Color: PENCOLOR (PC) and BACKGROUND (BG)*

The turtle has six pencolors and six background colors, plus a switching so-called color that reverses the color it passes over. The colors are numbered from 0 to 6.

Here are the colors and numbers for a black background (BG 0):

Color	Number
Black	0
White	1
Green	2
Violet	3
Orange	4
Blue	5
Reverse	6

The **PENCOLOR** (or **PC**) primitive takes the number of the color as input, and sets the turtle's pencolor to that color. Try typing

```
DRAW  
PC 4  
LT 45  
FD 50  
RT 90  
FD 50
```

To change the background color, type **BACKGROUND** (or **BG**) and the number. **BG 1** gives a white background. **BG 1 PC 0** will give you a black pen on a white background. Try typing

```
BG 5  
RT 135  
FD 62
```

The Apple computer color system determines the use of background colors. Blue and orange, for instance, do interesting things when exposed to violet and green. Combinations which will work as you expect:

**PENCOLOR on BACKGROUND draws**

4	2	green on green (erases)
4	3	green on violet
5	2	violet on green
5	3	violet on violet (erases)
2	4	orange on orange (erases)
2	5	orange on blue
3	4	blue on orange
3	5	blue on blue (erases)

In addition, changing the background color after a picture is drawn may change some of the lines in peculiar ways. Returning to the original background color restores the picture.

To see the effects of the different combinations, set a background color and draw some lines in each of the different colors. Change the background color and do it again.

On a black and white screen, colors 2-5 take on different textures, but black and white remain the same as always.

Logo draws thick lines to obtain clear colors on the Apple. On a black-and-white monitor, for thin white lines on black, use BG 6 and PC 1 through 5. PC 0 is black. (On a color monitor, these lines will not be uniformly white: vertical lines will be red or green, depending on their position.)

### *The Magic of PENCOLOR 6: Erasing*



PC 6 changes black to white and white to black when turtle tracks cross. This means that the turtle can erase a line by going back over it with PC changed to 6. To see how it works, type

```
FD 100  
PC 6  
BK 100
```

Now is the time to see one of the amazing effects you can create.

Type

```
PC 6  
LT 2  
FD 3000
```

Vary the turn and the distance forward for different effects. Try starting the turtle at the edge of the screen...

---

### Something to Try After You Read the Procedures Section

To see the effect of PC 6 with a non-stop procedure, choose one that never takes the same track twice. Clear the screen, hide the turtle, set your pencolor to 6, the reversing color, type the name of your procedure, and hit <CTRL> F so you can watch on the full screen:

DRAW HT PC 6 (procedure name) <CTRL> F

---

## *Introduction to Procedure Writing*

Now that you know how to drive the turtle around and make shapes, we will proceed to giving your shapes names which will become new turtle commands. You will be able to type BOX and get your box picture back, or SQUIGGLE to draw your squiggle.

To do this, you will write procedures.

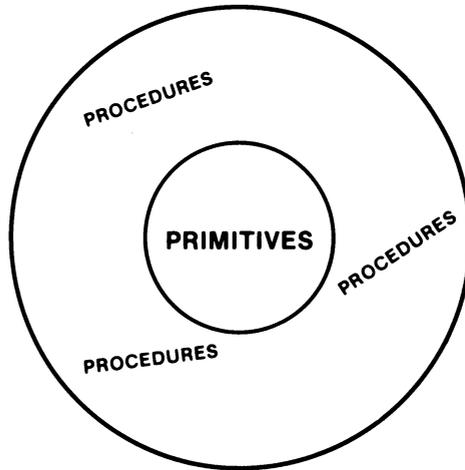
A procedure is a series of commands which you design to achieve a specific purpose. The commands may be composed of procedures and/or Logo primitives.



### LOGO COMMANDS

**PRIMITIVE:** a command that Logo has already been taught

**PROCEDURE:** a command that you teach Logo



Think of the **PRIMITIVES** as the core of the world of **PROCEDURES** you will write.

**FORWARD**, **BACK**, **LEFT**, **RIGHT**, **DRAW**, and **NO-DRAW** are Logo primitives. You used the primitives by typing their names, with numbers if they required them. To use a procedure, you do the same.

### *Naming a Procedure*

Type

```
MOVE <RETURN>
```

Logo tells you

```
THERE IS NO PROCEDURE NAMED MOVE
```

Logo is saying that it does not recognize the word you typed as either a Logo primitive or a procedure name. It does not know how to do that command.

```
? MOVE
THERE IS NO PROCEDURE NAMED MOVE
```

The name of a procedure is the single word that you type to tell Logo to perform the series of commands in the procedure.

Since you choose the name, select one that

1. Reminds you of what the procedure does
2. Is easy to remember
3. Is easy to type
4. Will not be confused with another name

***Writing a Procedure: EDIT Mode: TO, END, <CTRL> C, <CTRL> G***

To write a procedure, start with the name. The tutorial will use the name MOVE, but you may use your own.

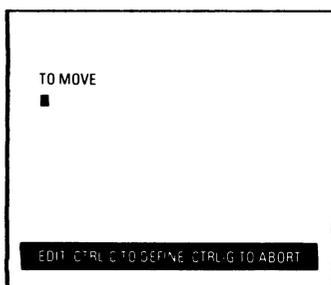
We tell Logo that we're about to write a new procedure by writing TO and the name of the procedure. For example, type:

```
TO MOVE
```

When you press <RETURN>, the screen will change: Logo will clear the screen and print the words

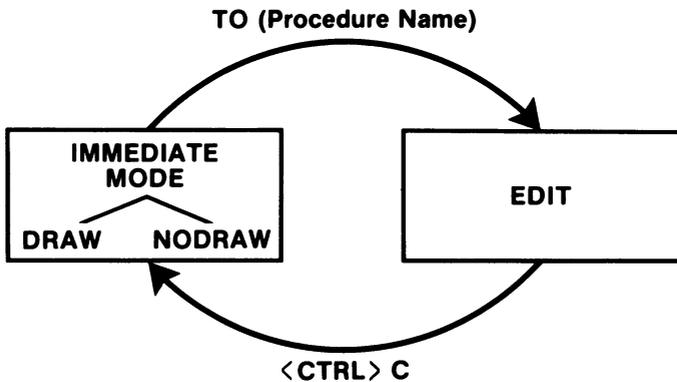
TO MOVE on the first line. Now Logo is in EDIT mode. The cursor will be at the beginning of the next line. At the bottom of the screen there will be a white line with black letters. It always says the same thing:

EDIT:CTRL-C TO DEFINE,CTRL-G TO ABORT



This reminds you that you are in EDIT mode, and tells you the two ways to get out of it: <CTRL> C to Complete the job and <CTRL> G in which any changes you have made in EDIT are Gone.

EDIT mode is very different from IMMEDIATE mode. In IMMEDIATE mode, Logo does the commands that you type (like FORWARD or RIGHT) as soon as you press the <RETURN> key. In EDIT mode, Logo waits for you to define a whole procedure; that is, to write a series of commands that will constitute the new procedure.



While you are in the editor you write the procedure. To use it, you must first get out of the editor by typing `<CTRL> C`, which puts you back into IMMEDIATE mode. (But don't do this yet.)

When you are using the editor, you can use the right and left arrows to move the cursor and `<DEL>` (`<ESC>`) to erase the character at the left of the cursor, just as you can in IMMEDIATE mode.

Type a line of text to practice. For example, you might type

```
FORWARD 33  
RIGHT 55
```

(or their short versions:)  
FD 33  
RT 55

Press the

```
<RETURN>
```

key. Note that it moved the cursor to the next line. In fact, <RETURN> is just another character to the editor: you can even erase it with the <DEL> (<ESC>) key. Press

<DEL> (<ESC>) and then the  
<RETURN>

key again to see this. Press

<DEL> (<ESC>)



until the whole line under TO MOVE goes away. (You can use the <REPT> (repeat) key on the Apple II+ in conjunction with <DEL> (<ESC>) to delete several characters very quickly.)

(See the Appendix and the Technical chapter for a discussion and summary of some other editing commands.)

Now type a series of commands, alternating FORWARD or BACK with RIGHT or LEFT. Remember to include the number of turtle steps or degrees, and to press <RETURN> after each.

For your first time through this tutorial, type either version of MOVE:

TO MOVE	TO MOVE
FORWARD 100	FD 100
RIGHT 15	RT 15
BACK 80	BK 80
RIGHT 25	RT 25

```
TO MOVE
FORWARD 100
RIGHT 15
BACK 80
RIGHT 25

EDIT CTRL C TO DEFINE CTRL G TO ABORT
```



Look over your procedure to be sure that

- (1) the commands are spelled correctly,
- (2) that you have used zeros in your numbers and not the letter O (zeros have slashes through them on the Apple), and
- (3) that there are spaces between the commands and the numbers.



Use the arrows and the `<DEL>` (`<ESC>`) key to fix errors. Use `<REPT>` (repeat) with the arrows to move the cursor quickly. When you finish your repairs, leave the cursor where it happens to be. Logo, unlike other languages, does not require the cursor to be at the end of the listing or even at the end of a line when you leave the EDIT mode.

The white line at the very bottom of the screen tells you the two ways of exiting from the editor and returning to IMMEDIATE mode.

Press `<CTRL> C`.

Logo will Complete your procedure definition: it will return you to IMMEDIATE mode, and will remember your procedure MOVE while you stay in Logo. It will confirm that it has read in your program by saying

**MOVE DEFINED**

If instead, you type <CTRL> G, your work done in EDIT mode will be Gone: Logo will return you to IMMEDIATE mode without accepting the work you did in EDIT. <CTRL> G stops Logo, whatever it is doing. Logo will confirm this state of affairs with

**STOPPED!**  
?

---

Note above that Logo types   PLEASE WAIT...  
                                          then   MOVE DEFINED  
                                          followed by the prompt   ?

(The wait occurs when you write a long procedure. You will not notice the wait with a short procedure like this.)

---

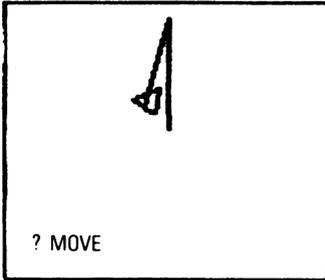
**Congratulations! You have written your first procedure. You have taught the turtle a new command. But wait! It's not time for congratulations yet. Does it work? You must try it.**

***Running a Procedure***

Type

**MOVE<RETURN>**

Just as typing the name of a primitive makes Logo do it, typing the name of a procedure makes Logo do what that procedure says to do. This is called **RUNNING** or **EXECUTING** the procedure.



If you have typed a word incorrectly within your procedure, Logo will try to help you by printing an error message. If you cannot figure out what the problem is, see the Appendix, which explains error messages with examples.



To make a change in your procedure, reenter the EDIT mode by typing TO and the name of your procedure. To change MOVE, type

TO MOVE

The screen will look as it did just before you left EDIT. Logo confirms that you are again in EDIT mode with the white line at the bottom of the screen.

Make your changes using the arrows and <DEL> (<ESC>) key, then exit EDIT with <CTRL> C. You are **DEBUGGING** your procedure (removing errors, called **BUGS**).

Run your procedure by typing its name. And now... Congratulations! It should look like the picture above.

Type `MOVE` again. The turtle will begin at the place it finished and will go in the direction it was pointing. You can also add to the shape on the screen by driving the turtle around with individual commands such as `RIGHT 12` or `FORWARD 55`, but these commands will not be included in the procedure.



You may put as many commands on a line as you wish; separate them with spaces and press `<RETURN>` at the end of the line to run them. If you run over the end of the line, Logo will continue on to the next line. (In `EDIT` mode, Logo puts an exclamation point to remind you that the line is continued).

**CAUTION:** In `IMMEDIATE` mode, Logo will do commands until it sees something it does not recognize. If one of the first commands on a long line of commands is misspelled, it will stop there and you will have to retype the incorrect one and all that came after it.

### *Planning and Drawing Your Favorite Square*

Procedures like `MOVE` draw somewhat random designs. Drawing a specific shape requires more specific thought about the sequence of commands you will write.

**Example:** Define a procedure called `SQUARE` which will draw a square.

Decisions you must make:

The number of

1. steps on a side (your choice)
2. degrees to turn at the corner (Aha!)
3. times to do a side and/or turn (Hmmm)

Things to remember (always):

- Correct spelling of commands
- Space between command and number
- Use zeros in numbers, not the letter O
- Press <RETURN> after each line
- Begin with the name:  
(for this one, type TO SQUARE)
- End your procedure with END



(Logo will put END in for you if you forget it. The only time it is definitely needed is when you define more than one procedure in the editor at the same time.)

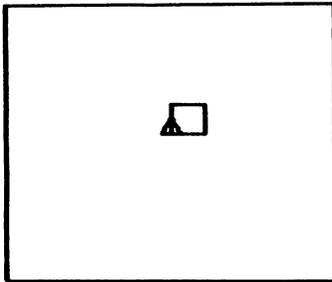
- Exit the editor with <CTRL> C  
(C for Complete)

Analysis:

Decision 1: From your turtle-driving projects, you have a good idea of the size of the screen. Choose a number considerably less than half, so that you can use your square in larger pictures. (Draw your proposed square on the screen with a felt tipped water based pen and make the turtle trace it.)

Decision 2: Only one specific number of degrees will work here; if you don't know what it is, try a few before you begin on SQUARE.

Decision 3: No doubt you know how many times you need to do the side and how many times you need to turn to draw a square. We will discuss other options later on.



SQUARE

Defining SQUARE:

To teach Logo the new command SQUARE, type

TO SQUARE

You are now in EDIT mode. Type in the commands you need, as you determined above. If you make mistakes in typing, use the arrow keys and <DEL> (<ESC>) to correct them. If the mistake is not on the line with the cursor, you must move the cursor to that line to correct it.



Exit from EDIT mode with <CTRL> C (C for Complete).

(Forgive the repetition of (C for Complete); we just don't want you to lose any of the work you have done in EDIT as you would with <CTRL> G (G for Gone...))

Type SQUARE to run it. Move or turn the turtle and run it again, and again. Notice that the turtle draws the square from wherever it happens to be, and starts off on the first side in whatever direction it is heading.



---

Now for a trick or two. You certainly don't want to spend the rest of your life typing SQUARE when you could obtain the same results typing SQ. (Would you want to have to type the whole word FORWARD all the time?) You created the procedure SQUARE using Logo primitives such as FD, BK, LT, and RT. Now you can create a procedure SQ using the new Logo command, the procedure name SQUARE.

Using the editing techniques you have learned, write a procedure SQ that looks like this:

```
TO SQ  
  SQUARE  
END
```

Clear the screen with DRAW and run SQ. Clear it again with DRAW and run SQUARE. You should get the same results with both. Now any time you want to draw a square, type either SQ or SQUARE.

SQ and SQUARE can also be used in procedures any time you wish, and as many times as you wish, just like the Logo primitives.

---

### ***Projects: Simple Procedures***

Write several of your own procedures. Choose appropriate names, but do not use the name MOVE as we will be using that again later.

### ***What goes Into a Procedure***

Any command you can type at the keyboard, as well as any procedure you have written, can be used in a procedure. Some commands have two versions: one is a word spelled out at the keyboard and the other uses the <CTRL> key plus a letter. Use the word in a procedure; the <CTRL> version is only for convenience at the keyboard.

#### **SUMMARY OF COMMANDS USED SO FAR THAT HAVE A CONVENIENT KEYBOARD VERSION**

<b>Procedure Version</b>	<b>Keyboard Version</b>
<b>TEXTSCREEN</b>	<b>&lt;CTRL&gt; T</b>
<b>SPLITSCREEN</b>	<b>&lt;CTRL&gt; S</b>
<b>FULLSCREEN</b>	<b>&lt;CTRL&gt; F</b>

*More Primitives: REPEAT, CLEARSCREEN (CS), HOME, PENUP (PU), PENDOWN (PD)*

The Logo command REPEAT saves you the work of typing a command or series of commands more than once. You tell Logo the number of times you wish to repeat, and enclose the command(s) to be repeated in square brackets.

Try these examples:

```
REPEAT 4 [FD 23]
REPEAT 3 [FD 30 RT 60]
REPEAT 8 [FD 65 RT 135]
REPEAT 20 [RT 50 FD 15 RT 60 FD 10]
```

As you will recall, when you type <CTRL> P, Logo will retype the previous line for you. You press <RETURN>, and Logo will execute it.

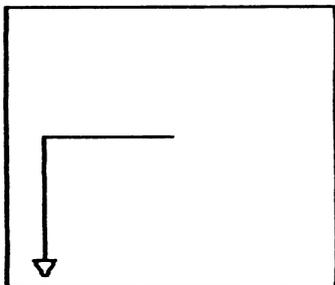
To repeat MOVE 24 times, type

```
REPEAT 24 [MOVE]
```

If the turtle starts in the middle of the screen, the design created by repeating MOVE will go off the edge (and appear on the opposite side). To avoid this, move the turtle before starting the design. 100 steps to the left and 100 steps down turn out to be a good starting point for MOVE, determined by examination and experimentation. Find a good starting point for your procedure.

To walk the turtle to its starting point for MOVE, type

```
LT 90 FD 100 LT 90 FD 100
```



The turtle is there, but it is pointing down. To head it in the right direction to start MOVE, type RT 180.

Now, what about the track it left? (If you type DRAW to get rid of the track, you will also send the turtle home.) To keep it where it is as Logo clears the screen, type CLEARSCREEN (or CS). Now try that REPEAT line with MOVE.

DRAW is a combination of CLEARSCREEN, SHOW-TURTLE (explained later), and HOME, the command that moves the turtle to the center of the screen and turns it to point straight up. Walk the turtle around some, then type HOME to see what happens.

There is another way to move the turtle without leaving a trace. Tell it to pick up its pen with **PENUP (PU)** before you start, and to put it down with **PENDOWN (PD)** when you get there. The line would be

```
PU LT 90 FD 100 LT 90 FD 100 RT 180 PD
```

The turtle arrives ready to draw, without leaving tracks.

---

The names of the primitives **PENUP** and **PENDOWN** come from the robot floor turtle which has the ability to pull its pen up and not draw or put it down and draw.

---

### ***Procedure Projects***

1. Write a setup procedure to move the turtle to its starting point without leaving a track.
2. Write a procedure using **REPEAT** which draws a design with **MOVE**.
3. Write a procedure to draw a four-sided figure.
4. Write a procedure to draw a rectangle.
5. Use your setup and rectangle procedures to draw a rectangle where **MOVE** began.
6. Write a procedure using **REPEAT** that repeats the sequence of drawing a shape with one of your shape procedures and then turns the turtle (then draws the shape and turns ...)



## *Saving Procedures: CATALOG, SAVE, POTS*

You have created a procedure which Logo will remember as long as you do not exit Logo or turn off your Apple. To be able to turn the computer off without losing your work, so that you may be able to use these procedures another day, you must ask Logo to **SAVE** them on a Logo file disk. Use a file disk prepared according to the instructions in the section titled Preparing a Blank Disk.

When you use the **SAVE** command, every procedure in your workspace is saved in a file on your disk. Your workspace is like your desktop. You do your work here, sometimes creating new material, sometimes bringing copies of files out of the drawers. When you finish for the day, you go to the copying machine, make a copy for the file, and file the copy away. Everything you are currently working on is on your desktop (in your workspace). This may include many procedures. When you want to save the contents of your workspace (desktop), use **SAVE** to transfer a copy of it to the disk (desk drawer).

You can use and change procedures only when they are in your workspace, not on the disk. When you are happy with your changes, or finished for the session, you store a copy of the workspace contents back as a file on the disk.

The **SAVE** command copies the entire contents of your workspace into a file on the disk. Just as your procedures have names, the collection of procedures in your workspace, which will be saved in a file, must have a name too, to distinguish it from your other files. Since

you choose the name for the group of procedures in the file, it is a smart idea to choose a file name that tells you what they are. The file name SHAPES might be useful for the first group of procedures you will be writing as you go through this chapter.

### Type

#### SAVE "SHAPES



The double-quote character immediately preceding the word is a crucial part of the file name. You cannot omit it. If you try to store your workspace without it, nothing will be saved, because Logo does not recognize it as a file name without the quote character. If you try to read a file without it, Logo will not find the file.



The quote distinguishes other types of names from procedure names. There is no space between the quote character and the word.



**WARNING:** You can have only one file per file name. Therefore, for the time being, use a new file name each time you save your workspace (such as SHAPES, SHAPES1, SHAPES2). (The Appendix includes more details about saving procedures.) If you had already had a file called SHAPES, the contents of the old file would be erased, replaced by the present contents of your workspace.

If you had nothing in your workspace (which is the case every time you turn on the computer, before you read a file or write a procedure) and typed SAVE "SHAPES, Logo will print out a message telling you there is nothing to save. But if you had one item in

your workspace, Logo would still save the entire contents of your workspace, even though it is almost empty, and the file "SHAPES would be replaced by a copy of the almost empty workspace. The old file "SHAPES on the disk would be gone.

This would be like taking a blank book with only a title page to the copying machine, copying it, and replacing your old files in the drawer with the copies of the blankpaper.

To see the names of the files you have saved on your disk, type

#### **CATALOG**

Everything on the disk will be listed, including the HELLO file which was put there during the initialization process, which must stay there although you will never need to use it again. Each Logo file will have your file name followed by .LOGO. For example, the new entry SHAPES.LOGO will appear on the list.

To print out the titles of your procedures in your workspace, type

#### **POTS**

To print out the commands in a procedure, type PO (procedure name) i.e.

#### **PO BOX**

SUMMARY		
Command	Purpose: Lists	Example
CATALOG	Files on disk	CATALOG
POTS	Procedure titles	POTS
PRINTOUT or PO	Procedure commands	PO BOX

***Clearing the Workspace, Reloading Procedures: READ, GOODBYE, ERASE (ER), ERASE ALL (ER ALL), ERASEFILE***

You may reload procedures into your workspace at any time. The most usual time might be when you begin a new session with Logo, but there will be times when you wish to add the contents of another file to what is already in your workspace. To list on the screen the files which are saved on your disk, type CATALOG, as before. To reload the procedures from your file SHAPES.LOGO, type

READ "SHAPES

The red light on the disk drive will go on, the disk will whirr, and the computer will print out the name of each of your procedures in your file SHAPES and confirm that it has been read into your workspace by printing DEFINED. For instance,

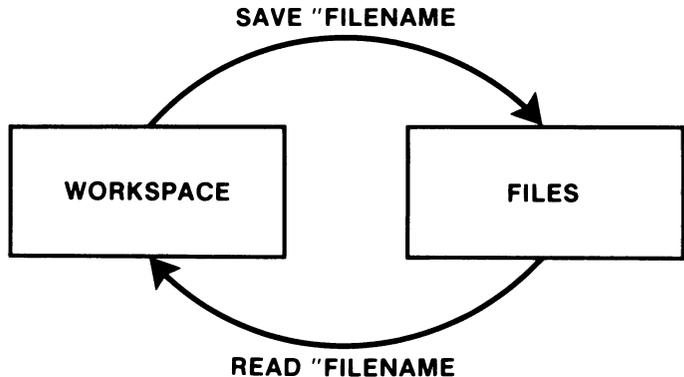
MOVE DEFINED



A word of warning: if you have changed MOVE in your workspace, the version read in from the disk will wipe out the one in your workspace. If you want to keep both versions, rename the one in your workspace using EDIT, before you read in the file. You can change the name in EDIT mode just as you change a command.

To store them all back in SHAPES, type

SAVE "SHAPES



There will be times when you want to clear your workspace, particularly when you want to shift gears and read in another file. If you want to save your current work, save it first.

To clear your workspace, type

ERASE ALL

It is always possible to erase whole files from a disk. The command which does this is `ERASEFILE`. For example, to permanently delete a file called `JUNK`, type

```
ERASEFILE "JUNK
```

to Logo. Always double-check to see that you're not erasing a file you want to keep.

### ***Selective Uses of SAVE, PO, ERASE (ER), and EDIT (ED)***

Certain Logo primitives can take a list of procedures as input instead of just one procedure name. For instance, typing

```
PO [SPIRAL SQUARE TRI]
```

will produce a screen listing of all three procedures. Likewise, typing

```
ER [TRI SQ CIR]
```

will erase all three procedures. In this case, the option of giving `ERASE` a list as its input is convenient but not crucial; the same effect could be achieved by typing

```
ER TRI ER SQ ER CIR
```

Using an extra input list with `SAVE` or `EDIT` is a bit more powerful, as it allows you to do things not otherwise possible.

Suppose you had two similar procedures, TRI and TRI2, and wanted to edit them together. You could type ED ALL, but then every procedure would appear in the editor; also, TRI and TRI2 might not be together. A simpler approach would be to type

```
ED [TRI TRI2]
```

SAVE normally puts all existing procedures into a new file. However, by using an input list you can save a selected list of procedures. If you have several procedures in your workspace and want to SAVE only a few of them, you can type something like

```
(SAVE "FIGURES [SQ TRI STAR])
```

The parentheses are necessary in order to tell Logo to expect an extra input.

### *Saving, Reading and Erasing Pictures: SAVEPICT, READPICT, ERASEPICT*

---

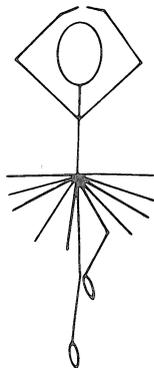
Logo can store complicated pictures on your disk and read them back in much less time than it takes the procedure to draw them. However, there is a tradeoff in disk space. The procedure might take 1 block of disk storage space. The picture will occupy 34 blocks. Only you can decide when this is worthwhile.

To save a picture (whatever is on the drawing part of the screen at the time), assign it a name. We shall use DANCER. To save the

picture part of the screen on the disk under the name DANCER, type

**SAVEPICT "DANCER**

The name you choose can be any name you care to give it. It does not have to be the same name as the procedure that drew it, but it could be. The picture can be the result of running one or several procedures (without clearing the screen between), or driving the turtle around, or a combination. Everything on the picture part of the screen except the turtle is stored with SAVEPICT.



To recall a stored picture (remember, this one will be listed on the disk as DANCER.PICT), type

**READPICT "DANCER**

To remove the picture from the disk forever (not just from the workspace), type

**ERASEPICT "DANCER**

In each case, use the double-quote character before the first character of the name.

---

---

## *The Invisible Turtle: HIDETURTLE (HT), SHOWTURTLE (ST)*

There are two situations in which you might want the turtle to become invisible.



1. To get it out of the way of your picture either during the drawing or after the picture is completed.
2. To speed up the drawing of a picture (the invisible turtle draws faster).

To tell the turtle to become invisible, type

HT (or its long form) HIDETURTLE

To tell it to reappear, type

ST or SHOWTURTLE

Except for being invisible, the hidden turtle works exactly the same as the visible turtle. In particular, it draws when its pen is down and leaves no trace when its pen is up.

## *Summary of Logo Commands Used So Far*

### TURTLE COMMANDS

Command	Abbreviation
FORWARD	FD
BACK	BK
LEFT	LT
RIGHT	RT
HOME	
PENUP	PU
PENDOWN	PD
HIDETURTLE	HT
SHOWTURTLE	ST
PENCOLOR	PC
BACKGROUND	BG

### SCREEN COMMANDS

Command	Abbreviation
CLEARSCREEN	CS
DRAW	
NODRAW	ND
TEXTSCREEN	<CTRL> T
SPLITSCREEN	<CTRL> S
FULLSCREEN	<CTRL> F

## FILE COMMANDS

READ            ERASEFILE  
SAVE  
SAVEPICT  
READPICT  
ERASEPICT

## PENCOLORS on BG 0

PC 0            Black  
PC 1            White  
PC 2            Green  
PC 3            Violet  
PC 4            Orange  
PC 5            Blue  
PC 6            Reverse

Commands used in all Logo domains (Graphics, Music, Computation, etc.):

TO ...	REPEAT	CATALOG
END	<CTRL> P	POTS
READ	EDIT	ERASE
SAVE	<CTRL> C	PO
	<CTRL> G	

<CTRL> P, <CTRL> C, and <CTRL> G are keyboard instructions which cannot be used in procedures.

*More About the Editor: Arrow Keys,  
<CTRL> P, <CTRL> N, <CTRL> O,  
<CTRL> A, <CTRL> E, <CTRL> D,  
<CTRL> X, <CTRL> Y*

In EDIT mode, you will want to move the cursor around the screen in order to make changes and additions. The following keystrokes will help you do this easily.

Use the four arrow keys to move the cursor in the direction of the arrow. Pressing an arrow key will not affect the text that the cursor passes over. Use the left and right arrow keys to move the cursor over characters that are on the same line. Use the up and down arrow keys to move the cursor from line to line.

If you are using an Apple II+, use <CTRL> P to move up to the Previous line (up on the screen), and <CTRL> N to move down to the Next line (down on the screen).

To Open up a space to insert a new line, type

<CTRL> O                    (the letter O)

No matter where the cursor is on the line, the rest of the line will be moved down to the next line, but the cursor will stay put.

To move the cursor to the beginning of the line, type

<CTRL> A

To move the cursor to the end of the line, type

<CTRL> E

To Delete the character under the cursor, type

<CTRL> D

Note that this is the opposite of the <DEL> key which deletes to the left of the cursor.

To kill a line from the cursor to the end, type

<CTRL>X

To Yank back the last line killed, type

<CTRL>Y

Other editing commands are described in the Appendix and in the Technical chapter.

### SUMMARY OF EDITING COMMANDS

	MOVING BACKWARD	MOVING FORWARD
1 character	Left arrow	Right arrow
End of line	<CTRL> A	<CTRL> E
Adjacent line	<CTRL> P or Up arrow	<CTRL> N or Down arrow
	DELETING BACKWARD	DELETING FORWARD
1 character	<DEL> (<ESC>)	<CTRL> D
Line	--	<CTRL> X
FOR EASY INSERTION OF A LINE		
Open line	<CTRL> O	
RESTORING DELETED TEXT		
Line	<CTRL> Y	

## *Projects Using Shapes*

1. Write a procedure (using SQ or SQUARE) that puts a square in each corner of the screen. (Hint: remember PENUP?)(Don't forget PENDOWN)
2. Write a procedure that draws a row of squares.
3. Write a procedure that draws a tower of squares. (Hint: use your row of squares procedure in it)
4. Write a procedure that draws a leaning tower of squares. (use your tower procedure)
5. How about a window with four panes?
6. Write a different procedure to draw the same size square as SQUARE.
7. Using the same sort of analysis used in developing the SQUARE procedure, figure out how you would draw a triangle whose turns are all the same size, then write the procedure.
8. Try #1-4 using triangles.
9. Write procedures to use your 4-sided (not a square) figure to make designs.
10. How about a window with 6 triangular panes?
11. Write a different procedure to draw the same size triangle.

Since all your new procedures (and old) are in your workspace, you can safely save them all in SHAPES by typing SAVE "SHAPES."

### *Listing a Procedure: PRINTOUT (PO), <CTRL> W*

Just as you can print out titles using POTS, you can also PRINTOUT the list of commands in any procedure. Type

PO (procedure name)

to list the commands in any procedure in your workspace. Type

PO (procedure name)

to list any other procedure in your workspace. PO provides a handy, quick way to check on a procedure, but to make changes in it, you must get into EDIT mode as described before. Type

PO ALL

to scroll by the listings of all the procedures in your workspace. Use



<CTRL> W (W for Wait)

to stop the scrolling; each <CTRL> W you press after you stop the scrolling will move one line onto the screen. You may inspect the titles one by one with more <CTRL> Ws, or resume the scrolling by pressing another key.

To printout a selected list of procedures, type

PO [PROC1 PROC2 . . . ]

SUMMARY OF LISTING COMMANDS	
Command	Result
CATALOG	Lists names of files on disk in disk drive
POTS	Lists names of procedures in workspace
PO	(procedure name) Lists commands in named procedure
PO ALL	Lists entire contents of workspace
<CTRL> W	Wait: computer waits for another key to be pressed: press <CTRL> W again for line by line inspection, or any key to resume scrolling.

*Heading: A Matter of State*

It is possible that when you closed your square and triangle, you finished your procedure with FD and did not follow it with a turn. This left the turtle heading in the direction the last side required. This makes it handy to draw successive figures in new positions, but it leads to confusion when you want to use the shape in another procedure.

It is generally good programming practice to leave the turtle in the same state in which you found it. The state of the turtle is its position and heading. It is already in the original position, since you closed the figure. All that is required is to turn the turtle so that it is heading in the original direction. This means one more turn, the same size as the other turns.

---

Consider these three procedures:



```

TO SQ          TO SUPER
  FD 30        REPEAT 8 [SQ RT 45]
  RT 90        END
  FD 30
  RT 90        TO STRANGE
  FD 30        REPEAT 4 [SQ]
  RT 90        RT 45
  FD 30        REPEAT 4 [SQ]
END            END
  
```

Both SUPER and STRANGE draw the same design (although they draw the parts of the design in a different order).

Note that the last turn in SQ, the one that would turn the turtle back to its original heading, is omitted.

If you edit SQ now to add a RT 90 at the end, SUPER will still draw the same design (in yet a new order), but STRANGE will not.

This may seem odd at first because we have not changed STRANGE. However, we DID change the procedure STRANGE uses.

To counteract the effect of adding the RT 90 at the end of SQ, we would have to insert a LT 90 immediately after SQ in each procedure that uses it.

This kind of fix is not always so easy. For example, if the newly introduced extra was a line instead of a turn, it would be harder (in some contexts, impossible) to counteract its effect.

So it is best to leave the turtle heading as it started. This will eliminate many interface bugs (puzzling things that must be fixed in order to use one procedure after another).

---

### *Copying a Procedure*

Your procedures SQUARE and TRIANGLE may now need another command added to them to turn the turtle to its original heading. But you have used SQUARE and TRIANGLE in other procedures; changing them now would spoil the procedures that use them. Take heart; change SQUARE, but give the new version a new name, such as SQUARE1. While in EDIT, change the name slightly (it can be edited like any other part of the procedure), then move down and add the new command. Voila. You now have your original procedure plus a slightly altered copy under a new name.

### *A Magic Number*

Now for a rather basic question: how far around did the turtle turn when it drew the square that left it in the same state that it started from (same position and heading)? (Add up the turns.) How far around did the turtle turn when it drew the triangle that left it in its original state?

You have just discovered a great truth: the turtle will turn the same amount to get back to its original heading, no matter how it goes. The total amount of the turn, adding the turns in one direction and subtracting if it turns the other way, will be the magic number you just discovered. (Of course, if it goes one way and then cancels the turn out completely by going the other way, the total turn will be 0, but it will not have traveled completely AROUND anything, either.) This is called The Total Turtle Trip Theorem: if the turtle travels around an area, no matter what shape, and ends in the same place that it started, heading in the same direction, it always turns the same amount.

You can use the magic number to make shapes with any number of sides. To see the relationship between the magic number and the turns you made in the square, divide the magic number by the number of turns. Let Logo do it for you. On the computer, where we cannot type one character above another on a single line, we use the slash (/) (on same key as the ?) for division. To divide 10 by 5, type

```
10/5
```

Logo will reply

```
RESULT: 2
```

Remember, when Logo requires a number, it can use the result of an arithmetic operation, so you can also use this division as the number required by the Logo primitives FD, BK, LT, and RT. For example,

Command	Equivalent
FD 100/2	FD 50
RT 300/30	RT 10
BK 200/4	BK 50
LT 360/4	LT 90

### *Projects: More Shapes*

1. Using REPEAT and division in your turn command, write another procedure that draws a square.
2. Using REPEAT and division in your turn command, write another procedure that draws a triangle.
3. Using REPEAT and division in your turn command, write a procedure that draws a 5-sided figure.
4. Write a procedure that draws a 6-sided figure.
5. Write a procedure that draws a 7-sided figure.
6. How about a 15-sided figure?

### *Introduction to Variables: Procedures That Take Inputs*

DRAW does the same thing each time it is used. FORWARD is more flexible; it moves the turtle different distances depending on its input.



INPUT is the specific term for the number required by commands like FD, BK, LT, and RT. (Later you will also see INPUTS which are not numbers.)

So far your procedures have always done the same thing each time they were used, but it is possible to write procedures which use some input to tell them, for example, how much to move the turtle.

It would be nice to have a BOX procedure which draws different sized squares, just as we have a line procedure (FORWARD) which draws different lengths of line.

We would expect BOX 10 to produce a small box and BOX 100 to produce a larger box. To describe what happens more fully, we might say:

To draw a box of some dimension,  
we go forward that dimension,  
turn right 90 degrees,  
go forward that dimension,  
turn right 90,  
forward that dimension,  
right 90,  
forward dimension,  
right 90  
and that's it.

The Logo translation of the English is very similar:

```
TO BOX :DIMENSION
  FD :DIMENSION
  RT 90
  FD :DIMENSION
  RT 90
  FD :DIMENSION
  RT 90
  FD :DIMENSION
  RT 90
END
```

Or, we could have said:

To draw a box of some dimension,  
we must, 4 times, go forward that dimension  
and turn right 90 degrees.

which translates into Logo as

```
TO BOX :DIMENSION  
  REPEAT 4 [ FD :DIMENSION RT 90 ]  
END
```

NOTE:



1. The **:** that appears in the procedure must be there every time an input variable is used, attached directly to the variable name without a space between. The dots distinguish the name of a variable from the name of a procedure. We call the colon (**:**) **DOTS** because it is more descriptive. Read **:DIMENSION** as **DOTS DIMENSION**.
2. Variable names are just as much your choice as procedure names. We could have written

```
TO BOX :WIDTH or  
  TO BOX :DIST or even  
  TO BOX :X
```

Of course, the name you choose in the title line must also be the one used within the procedure, so those procedures would have had

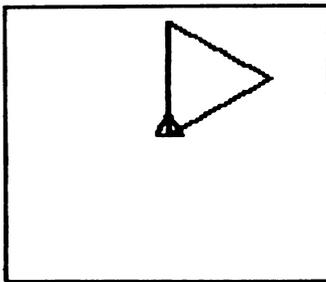
```
FD :WIDTH FD :DIST and FD :X
```

3. Note where the variable-number name must go, in the same place in which you previously put the

constant number. In the procedure TRI, for example, FD 100 becomes FD :LENGTH. To pass the number into the procedure for FORWARD to use, the title now must become TO TRI :LENGTH. The two procedures look like this:

```
TO TRI
  REPEAT 3 [FD 100 RT 120]
END
```

```
TO TRI :LENGTH
  REPEAT 3 [FD :LENGTH RT 120]
END
```



TRI

This TRI procedure is very much like the Logo primitives you have been using. For a triangle of any size, you type TRI and the length of the side.

Try a few triangles of different sizes.



Try typing TRI without a number. Now that TRI is defined with a variable input, Logo looks for that input, just as it does when you type FD or RT. To recall just what inputs a procedure is expecting, type either POTS, to print out the titles of all the procedures in your workspace, or PO (procedure name), to print out the one procedure (for instance PO TRI).

You have a choice now when you want to use TRI in another procedure. You can specify the size of the triangle in the procedure (TRI 75), or you can choose to decide on the size when you run the superprocedure it is in. You must pass the number in to TRI if you do not specify it inside the procedure. For example:



```
TO TWO.TRI          TO TWO.TRI2 :LENGTH
  TRI 75            TRI :LENGTH
  RT 90             RT 90
  TRI 75            TRI :LENGTH
END                 END
```

Note: Two words can be combined with a dot to make a title.

Both versions of TWO.TRI use the same subprocedure TRI. Both versions can make a triangle design with triangle sides of length 75. BUT one version can only draw a size 75 design, while the other can draw designs of any size. The size of its design will depend on the number you give it when you run it.



The variable name :LENGTH may be used in any number of procedures. You are allowed to have only one procedure named SQUARE or TRIANGLE, but both may use the variable name :LENGTH. :LENGTH is what is called a local variable, local to its procedure. A name used in one procedure will not interfere with the same name used in another.

This also means that TWO.TRI2 could have used a different name for the variable than was used internally by TRI.

### *Projects: Sizable Shapes*

1. Write a procedure SQV with variable input and use it in a new procedure SQUARE4 to draw a series of squares of different sizes, all starting at the same place. (Hint: you can add to a picture; you don't have to clear the screen with DRAW everytime you want to draw something more.)
2. Add another set of squares beside the first.
3. Write a procedure that uses a specific size square in it.
4. (Here's a toughie) Write a procedure that draws 4 squares, each 25 steps bigger than the last, and which receives as input the size of the first square when the procedure is run.

### *From SQUARE to POLY*

SQUARE4 (if you did project 1) now has a variable input for the length of the side, but it still has two other numbers, the size of the turn and the number of times the sequence is repeated. Either or both of these numbers could also become variables. (However, if we change either one, it would not draw a square.)

You know from your experiments that 360 is the magic number that takes the turtle all the way around and back to the same heading, no matter what shape it is going around. You also know that the amount of the turn at each corner is 360 divided by the number of turns. Remember too that Logo will do all the work of dividing for you. You may use  $360/4$  as the input for your turn in SQUARE4, for instance.

In other words, the SQUARE4 procedure could be written

```
TO SQUARE4 :LENGTH  
  REPEAT 4 [FD :LENGTH RT 360/4]  
END
```

The 4 in both places is the number of turns. SQUARE4 now has a variable input for the length of the side and one other number that might be changed, the number of turns or sides. What if we made that number a variable, too? The procedure would repeat the side-and-turn sequence that number of times, and would divide 360 by the number for the turn. Sounds all right, but it wouldn't draw a square. It would draw a many-sided figure, (called a polygon) with the number of sides you chose when you ran it. Call it POLY.

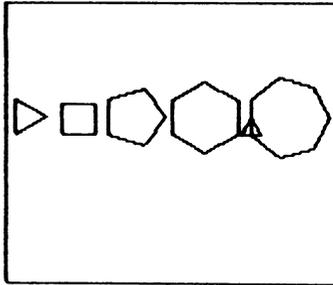
POLY will need two names for the variable inputs, and they should clearly describe what they are for.

:LENGTH would be fine for the length of the side again, and you could use :TURNS for the number of turns (or sides).

Both variable names must appear in the title, to pass the numbers in to where they are used in the procedure. Choose the order you will remember best. They do not have to appear in the title in the order in which they are used in the procedure, but, when you run POLY, the numbers must be typed in the same order as the variables which represent them in the title. POLY 100 4 will be very different from POLY 4 100.

So POLY could look like this:

```
TO POLY :LEN :TURNS
  REPEAT :TURNS [FD :LEN RT 360/:TURNS]
END
```



POLY

### *Projects: Regular Polygons*

Experiment with different inputs to POLY. Write down the ones you like.

1. What is the difference between POLY 100 4 and POLY 4 100? Try them both.
2. Try POLY with the same :LENGTH input and a lot of different numbers for :TURNS.
3. Keep :TURNS the same and try a lot of different numbers for :LENGTH.
4. Make a design using POLY twice, with a different number of sides (:TURNS) each time.
5. Use POLY to make a triangle.
6. What is the largest number you can use for turns? (Hint: hide the turtle for a quicker trip.)

### *Another View of POLY*

Look back at the procedures in which you used division to help you draw 3, 4, 5, 6, and 7-sided figures.

They probably look a lot alike. In English you might describe them this way:

To draw a shape of some specified number of sides, repeat for each side: go forward some distance and turn right 360 divided by the number of sides

Let's use a forward distance of 50. The English translates to Logo:

```
TO SHAPE :NUMBER.OF.SIDES  
Type as one line REPEAT :NUMBER.OF.SIDES  
                  [ FD 50 RT 360/:NUMBER.OF.SIDES ]  
END
```

(Note that the REPEAT statement must be typed on one line.) Type in SHAPE and try it with various inputs. Try



```
SHAPE 3  
SHAPE 4
```



We can also make shapes of various sizes by making the forward distance a variable. Replace the 50 with the variable :DIST and add it to the title:

```
TO SHAPE :NUMBER.OF.SIDES :DIST  
Type as one line REPEAT :NUMBER.OF.SIDES  
                  [FD :DIST RT 360/:NUMBER.OF.SIDES]  
END
```



```
Try  
SHAPE 3 50 and SHAPE 50 3
```

It is important to remember the order of the variables in the title.

This procedure produces the same design as POLY (above). The number of sides will be the same as the number of turns.

## *Circles*

So far we have drawn only straight lines. How does the turtle draw curves? When you consider that all it can do is step and turn, then it must be some combination of steps and turns in curves as well as in straight-sided figures. Experiment with small steps and small turns. Use REPEAT with your little steps and turns to avoid exhaustion. Try some combinations in IMMEDIATE mode, then make procedures of the combinations you like.



Some things to remember:

- the turtle draws faster when hidden (HT)
- <CTRL> G stops the turtle, whatever it is doing
- you know how far the turtle must turn to finish back where it started

## *Projects: Curves*

Try these first, then make procedures of the ones you would like to be able to use. Give your procedures descriptive names, for instance, a 6th-of-a-circle arc to the right might be ARCR6.

1. Use REPEAT to draw a circle, then without clearing the screen, draw another circle with steps twice as big as in the first one. Draw another with the turn twice as big.

2. Draw a circle to the right and an identical one to the left.
3. Figure out the diameter (distance across) of the last circle.
4. Draw a quarter-circle arc to the right.
5. Draw another quarter-circle arc with steps twice as big as the one in #4.
6. Draw a 6th-of-a-circle arc to the left, then a 6th-of-a-circle arc to the right. (Hint: use division, and let Logo do it for you)
7. Write a procedure that uses an arc procedure and straight lines to draw a picture or design.
8. Do these projects using variable inputs for the step size and the number of degrees.

See the section on Procedures for a way to develop an arc procedure. There are also several demonstration arc and circle procedures on the Utilities Disk. See the Utilities Disk section.

### *Using Subprocedures*

A procedure used as a command in another procedure is called a subprocedure. The procedure which uses it is a superprocedure. You have already used SQUARE as a subprocedure when you called it in the superprocedure SQ, and, if you did the projects, you used procedures as subprocedures to draw towers, windows, and a design with arcs and lines.

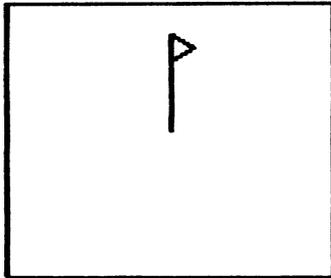
A subprocedure is useful when you want to use a procedure as a new primitive in a variety of procedures, or several times in one procedure. You could write a procedure to do one side of a square (such as FD 73) and

one turn (RT 90). If you called it SQUARESIDE, then your square procedure would look like this:

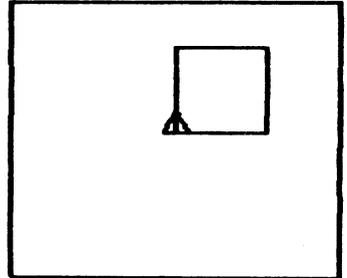
```
TO SQUARE2  
  SQUARESIDE  
  SQUARESIDE  
  SQUARESIDE  
  SQUARESIDE  
END
```

(or perhaps)

```
TO SQUARE2  
  REPEAT 4 [SQUARESIDE]  
END
```



SQUARESIDE



SQUARE2

Any Logo procedure can be a subprocedure. In addition, subprocedures may have subprocedures of their own.

For example:

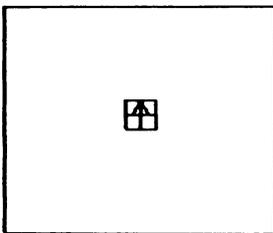
SQUARE2 uses SQUARESIDE as a subprocedure.

We write WINDOW which uses SQUARE2 as a subprocedure.

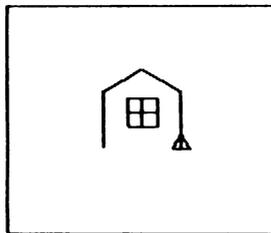
SQUARE2, which has SQUARESIDE as a subprocedure, is now also a subprocedure.

We write HOUSE, which uses WINDOW, and TOWN, which uses HOUSE...

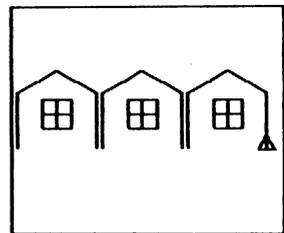
We can build as far as we want; all the procedures except the top one (TOWN) will be used as subprocedures, and all but the bottom one (SQUARESIDE) will use subprocedures. All but TOWN and SQUARESIDE will both use and be subprocedures.



WINDOW



HOUSE



TOWN



The point of this exercise is to show that even now you are writing procedures you can use later on. As you write your way through the tutorial, note the procedures that may be particularly useful to you as subprocedures. You might even want to file them separately

after a while, in a file called NEW.PRIMITIVES (Logo allows you to use periods in your procedure and file names to connect words.) Your arc procedures are good examples of useful primitive-like subprocedures.

### *Non-stop Procedures: Introduction to Recursion*

Your procedures up to now have been very well-behaved and have stopped when you told them to. Now let's try a type of procedure that simply doesn't know when to stop.



As you know, a Logo procedure can use any Logo command, whether it is a primitive or a procedure. This includes a procedure being able to use itself.

The ability of a procedure to call itself is called recursion. We shall work up to the power of recursion with some simple examples. What happens when you tell a procedure to do itself? Let's try it with a square program:

```
TO SQUARE3 :LENGTH
  FD :LENGTH
  RT 90           (Stop me with <CTRL> G)
  SQUARE3 :LENGTH
END
```

What have we told SQUARE3 to do?

1. Draw a side and do a turn
2. Do SQUARE3
  1. Draw a side and do a turn
  2. Do SQUARE3
    1. ...

Only a <CTRL> G typed at the keyboard will stop this runaway square. It will go over and over the same track until you stop it. Not very interesting.

But what would happen if there was a side and a turn that made a design which would not go over itself? Change the amount of the turn. Try a little more or less than the 90 used for a square. Try, for example,



```
FD :LENGTH  
RT 87
```

***Projects: Simple Recursion***

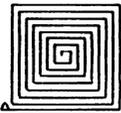
1. Write a recursive procedure that draws a little figure then calls itself.
2. Write a recursive procedure that uses arcs and lines.
3. Use your triangle procedure in a recursive procedure.
4. Write a recursive procedure to draw a star.

***Recursion: Changing the Input  
WRAP, NOWRAP, CONTINUE (CO)***

Another interesting possibility is that of changing the length of the side each time it is drawn. Remember, wherever Logo requires a number, there are several ways to give it one. We have tried actual values (100 for instance) and right now we are using a variable (:LENGTH). The next kind of number to try is a number which Logo will produce for us by doing some arithmetic, for instance, :LENGTH + 3.

```
TO SQUARAL :LENGTH
  FD :LENGTH
  RT 90
  SQUARAL :LENGTH + 3
END
```

When SQUARAL calls SQUARAL, it uses a little bigger number for the length of the side. Now, even with a turn of 90, the design will not repeat itself on the same path.



SQUARAL

What happens when you run this procedure: Type

SQUARAL 5

1. The turtle moves FD 5 for the first side and turns right.
2. Logo runs SQUARAL 5 + 3.

SQUARAL 8

1. The turtle moves FD 8, 3 steps more than the first side and turns.
2. Logo runs SQUARAL 8 + 3.

SQUARAL 11

1. The turtle moves FD 11 and turns.
2. Logo runs SQUARAL 11 + 3.  
and so on.

The second side, and each side after it, will be 3 steps longer than the previous side, and the picture will clearly not be a square.

---

Before long, the line spills off the edge and reappears on the other side of the screen. Logo is in WRAP mode, where the lines wrap around the screen rather than stopping at the edge. This can make interesting effects, particularly with PENCOLOR 6, which reverses the color when lines cross.

Remember, you can use FULLSCREEN or <CTRL> F, SPLITSCREEN or <CTRL> S, and TEXTSCREEN or <CTRL> T to change the amount of drawing space showing on the screen.

To make Logo stop the procedure when the line threatens to get out of bounds, type NOWRAP to put Logo into NOWRAP mode. Now, no matter where the turtle is when you run the procedure, when the design gets too big for the screen, Logo will stop it. (There are more elegant ways to stop recursive procedures mentioned later on. See Stopping With Style.)

The commands WRAP and NOWRAP, like all other Logo commands, can also be used in procedures. Whenever they are used, each stays in effect until the other is used, or until you exit DRAW mode.

---

## ***Projects: Changing Inputs***

Make the changes suggested below and give each changed version a new name. Run each version with several different inputs, large and small (SQUARE 10, SQUARE 100 for instance)

1. Change the amount added to :LENGTH in SQUARAL make it large; make it very small.
2. Subtract an amount from :LENGTH in SQUARAL instead of adding to it.
3. Change the size of the turn a little bit.
4. Multiply :LENGTH by a number. Keep trying until you find one you like. Remember, use the star (\*) for multiplication. (Hint: you can use decimals such as 1.1 or 1.5)
5. Try all of your procedures in WRAP mode and NO-WRAP mode.
6. In WRAP mode, try your procedures with PENCOLOR 6 (PC 6).
7. Write a procedure which takes a variable input and draws one square. (Hint: use REPEAT) Then write a recursive procedure that uses the square procedure as a subprocedure and draws a series of squares which get bigger and bigger.

## *Stopping With Style: IF-THEN, STOP*

Logo can make choices based on what you tell it to do. You can write IF (something) is true, THEN (do something else) (STOP for instance). (If it is not true, it will go directly to the next line. If it IS true, and the instruction is not STOP, it will execute the instruction and THEN go to the next line.)

For example, you would like to be able to specify the number of times a recursive procedure executes, and specify a different number every time you run it. Make the procedure count down from the number you give it, and test the count each time it executes with

```
IF :TIMES = 0 THEN STOP
```

Here is a procedure that draws a square, turns the turtle a little, and does it again.

```
TO DESIGN :TIMES
  IF :TIMES = 0 THEN STOP
  SQUARE 100
  RT 45
  DESIGN :TIMES-1
END
```

This is what happens when you type

□ DESIGN 4

1. Logo tests :TIMES (4) to see if it is zero.
2. Logo runs SQUARE and turns the turtle
3. Logo calls DESIGN 4-1 or DESIGN 3

**DESIGN 3**

1. Logo tests `:TIMES (3)` to see if it is 0
2. Logo runs `SQUARE` and turns the turtle
3. Logo calls `DESIGN 3—1` or `DESIGN 2`

**DESIGN 2**

1. Logo tests `:TIMES (2)` to see if it is 0
2. Logo runs `SQUARE` and turns the turtle
3. Logo calls `DESIGN 2—1` or `DESIGN 1`

**DESIGN****DESIGN 1**

1. tests `:TIMES (1)` to see if it is 0
2. runs `SQUARE` and turns the turtle
3. calls `DESIGN 1—1` or `DESIGN 0`

**DESIGN 0**

1. Logo tests `:TIMES (0)` to see if it is zero and stops. `:TIMES = 0` is finally true.

Control is passed back to each level in turn and the procedure is done. This aspect of recursion will be covered in the next section.

What happens when your friend tries to be funny and runs `DESIGN` with a negative number?(Ah, you tried it ... Well, remember `<CTRL> G`.) You will be pleased to know that you can test for that also. In fact, you can put as many tests as you wish in your procedure. You can test for that negative number by using one of the two other conditions, less than (`<`) or greater than (`>`).

To cover both situations, your negative friend and the normal ending of the procedure, change your test:

```
TO DESIGN :TIMES
  IF :TIMES < 1 THEN STOP
  SQUARE 100
  RT 45
  DESIGN :TIMES-1
END
```

Now DESIGN will stop when :TIMES gets to 0 and will never start if :TIMES is less than 0.

The procedure can still have variable inputs for other values, such as the length of the side of the square:

```
TO DESIGN :TIMES :LENGTH
  IF :TIMES < 1 THEN STOP
  SQUARE :LENGTH
  RT 45
  DESIGN :TIMES-1 :LENGTH
END
```

You can even change the length each time it is called if you wish by incrementing it as it is in SQUARAL.



**NOTE:** Be sure the variable you test in your procedure will eventually reach the test value. For example, in our first version of DESIGN, :TIMES would never have reached 0 if it had started out negative. The first one, in fact, will also fail with a decimal such as 10.3.

If you don't happen to think of this possibility, the procedure may go on and on and on and you won't know why.

This is a common problem in writing procedures: the computer always does what you TELL it to do, whether or not it's what you want it to do. BUGS creep into the procedures of the best of programmers.



Bugs can be fun. You can learn from them, and sometimes what the computer does is more interesting than what you had intended.

### *Projects: Testing and Stopping*

1. Try replacing the 45 in RT 45 with something that depends on :TIMES, such as  $4 * :TIMES$ .
2. Write a procedure to draw a tower of smaller and smaller squares, choosing the number of squares when you run it.
3. In DESIGN, change the input for RT to a variable. (Remember to add the variable name to the procedure title)

### *Using the Full Power of Recursion*

---

To see Logo execute procedures step by step, use TRACE, described in the section on debugging in this chapter and in the Logo Command Glossary.

---

The results of the recursive procedures shown so far could have been achieved with non-recursive procedures. Each one so far has done something and then called itself to do essentially the same thing again. Except for DESIGN, the procedures did not stop by themselves, so they never had the chance to return to the top level.

The power of recursion, and what makes it different from iteration (repetition), is its ability to come back from the last call to itself (called the deepest or lowest level), finishing a job at each level as it returns.

This will be a new concept to many. Logo is one of the few computer languages with this capability.

The following comparison will illustrate this:

```
TO COUNTER :NUMBER
  IF :NUMBER > 2 STOP
  PRINT :NUMBER
  COUNTER :NUMBER + 1
END
```

```
TO COUNT.PLUS :NUMBER
  IF :NUMBER > 2 STOP
  PRINT :NUMBER
  COUNT.PLUS :NUMBER + 1
  PRINT :NUMBER
END
```

Small numbers are used to permit full step-wise explanation.

COUNTER works in the same way as DESIGN. Type

COUNTER 1 and Logo will respond

```
1
2
```

COUNT.PLUS, as its name suggests, does more. This is what happens when you type

**COUNT.PLUS 1**

1. Logo tests to see if :NUMBER (1) greater than 2.
2. Logo prints :NUMBER (1).
3. Logo calls COUNT.PLUS :NUMBER + 1 (2).
4. (The last statement, PRINT :NUMBER, is not executed.)

**COUNT.PLUS 2**

1. Logo tests to see if :NUMBER (2) > 2.
2. Logo prints :NUMBER (2).
3. Logo calls COUNT.PLUS :NUMBER + 1 (3).
4. (The last statement, PRINT :NUMBER, is not executed.)

**COUNT.PLUS 3**

1. Logo tests to see if :NUMBER (3) > 2.
2. Logo stops and returns control to the procedure that called COUNT.PLUS 3, which was COUNT.PLUS 2.

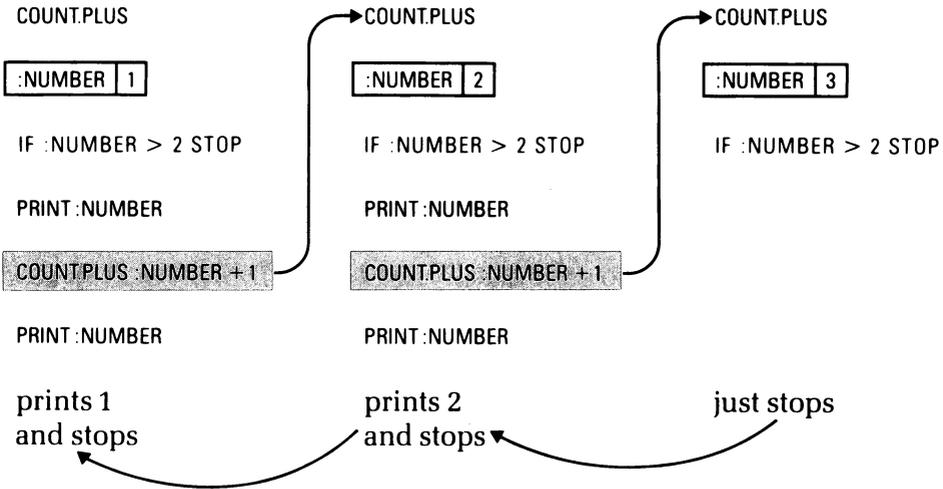
**COUNT.PLUS 2**

5. Logo executes the next statement in COUNT.PLUS 2, which is PRINT :NUMBER. Prints 2.
6. Logo stops and returns control to the procedure that called COUNT.PLUS 2, which was COUNT.PLUS 1.

**COUNT.PLUS 1**

5. Logo executes the next statement in COUNT.PLUS 1, which is PRINT :NUMBER. Prints 1.
6. Logo stops and returns control to the procedure that called COUNT.PLUS 1, which was the main Logo command level.

The diagram shows how all copies of COUNT.PLUS exist at once, each with its own private value for :NUMBER.



The process of recursion is based on one idea:

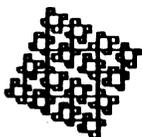


When a procedure (A) calls another procedure (B), the calling procedure (A) puts on hold any instructions which come after the call. When the procedure (B) which is called stops, the calling procedure (A) continues with the rest of its instructions after the call to (B).

What makes recursion so powerful is that this idea applies also to (B) and any procedure (B) calls, and any procedure that THAT procedure calls...

And all of these copies of the procedure co-exist, each with its private portfolio of values. All copies are used and exist as if they were completely different procedures.

An excellent example is the procedure SQS which produces squares with half-size squares on the corners:



```
TO SQS :LENGTH
  IF :LENGTH < 5 STOP
  REPEAT 4 [FD :LENGTH SQS :LENGTH/2 RIGHT 90]
END
```

```
TO SQR :LENGTH
  IF :LENGTH < 5 STOP
  REPEAT 4 [FD :LENGTH RT 90]
  SQR :LENGTH/2
END
```

Note the difference the placement of the recursive call makes in SQR and SQS.

The procedure EXPONENT in the Computation chapter and the procedure TET on the Utilities disk (see the Utilities chapter) are two other examples of good recursive procedures. See also Recursion in Music in the Music chapter.

See chapter 2 in LOGO FOR THE APPLE II, by Professor Harold Abelson, M.I.T., for a further discussion of recursion.

### ***Recursion Projects***

1. Write a set of procedures which draw successively smaller houses. Use subprocedures for the parts of the house.
2. Write a procedure to draw a binary tree. A binary tree is a v-shaped tree with a smaller v-shaped tree on each tip. Develop the procedure for the basic V, then determine where in the procedure you would insert a recursive call to itself to draw a smaller tree.

To stop the procedure, use a test similar to the one used in SQS.

3. Write a procedure that draws a series of successively larger fish, each totally within the next larger.

### *Special Effects and New Utilities*

Remember that PC 6 changes black to white and white to black when turtle tracks cross. Try it with SQUARE3 and SQUARAL.

```
TO SQUARE3 :LEN TO SQUARAL :LEN
  FD :LEN        FD :LEN
  RT 90          RT 90
  SQUARE3 :LEN   SQUARAL :LEN + 3
END              END
```

Clear the screen, hide the turtle, set your pencolor to 6 (the reversing color), type SQUARE3, and hit <CTRL> F so you can watch on the full screen:

```
DRAW HT PC 6 SQUARE3 <CTRL> F
```

If you like the effect, write a procedure which will do it for you at the stroke of a single name. Give the procedure a name and the commands in the line above (use the word FULLSCREEN for the <CTRL> F):

```
TO SUPERSQ
  DRAW HT PC 6 FULLSCREEN SQUARE3
END
```

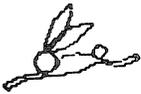
Type SUPERSQ and sit back.

You could also make a separate procedure of the SETUP part. Make this one of your own utility procedures.

```

TO SETUP          TO SUPERSQ
  DRAW            SETUP
  HT              SQUARE3
  PC 6            END
  FULLSCREEN
END

```



Since Logo lets you use primitives and procedures the same way, you can build your own file of new primitives, utility procedures that do the special things that you want to do. This might even include procedures like C which has the single command CATALOG, simply to save typing...

If you can change a color once, you can change it again, both background and pencolor. You can make the change once in a great while, or you can flash from one to another.

Here's a flashy example (NOTSQ is not quite a square)

```

TO NOTSQ
  REPEAT 4 [FD 85 RT 85]
END

```

```

TO FLASH.NOTSQ
  PC 6
  BG 0 NOTSQ
  BG 1 NOTSQ
  BG 2 NOTSQ
  FLASH.NOTSQ
END

```

FLASH.NOTSQ sets the pencolor to 6 (reversing), the background to black, and runs NOTSQ, four lines that

don't quite meet. The background changes to white, four more lines are drawn, the background changes to a color, four more lines, then the whole procedure repeats endlessly. Each time a line crosses a line, the color of that spot is reversed.

### ***RANDOM Numbers, Numbers from Arithmetic Operations, Inputs, Outputs***

The Logo primitive **RANDOM** will give you a number, chosen at random from the group you specify. You specify the group from 0 to your number by giving **RANDOM** the next higher number. For instance, **RANDOM 7** will choose a number from 0 to 6 (just what **PC** and **BG** need).



The number **RANDOM** chooses is called its **OUTPUT**. If you type **RANDOM 7** at the keyboard, Logo will respond with **RESULT: 4** (or some other number from 0 to 6), just as it printed **RESULT: 90** when you typed **360/4**. Both **RANDOM** and the arithmetic operation produce a result, that is, they each put out a number, which is called an **OUTPUT**.

The number **RANDOM** uses is its **INPUT**. You can never leave out an input: the command needs it to work. On the other hand, in **IMMEDIATE** mode, Logo will print an output as a **RESULT** sometimes. However, any time Logo expects to go on, as in a procedure or a **REPEAT** command, it needs to know what to do with that output. Try typing

```
REPEAT 4 [RANDOM 8]
```

and Logo will complain.

This is equivalent to typing

```
REPEAT 4 [5]
```

Give RANDOM's OUTPUT to something that requires an INPUT (such as FORWARD or PRINT), and you are in business:

```
REPEAT 4 [FORWARD RANDOM 8]
```

Ooh la la... it works.

To make the turtle's pen or the background take on a random color, use RANDOM 7 instead of the number. FLASH.NOTSQ could now be

```
TO FLASH.NOTSQ
  PC 6
  BG RANDOM 7
  NOTSQ
  FLASH.NOTSQ
END
```

(You have the choice of editing the old FLASH.NOTSQ or typing ERASE (or ER) FLASH.NOTSQ and typing the new version.)

Here FLASH.NOTSQ sets the pencolor to reverse, picks a random background color, runs NOTSQ, then does the same three steps again and again until you stop it.

To avoid using the reversing (eraser) color #6, use RANDOM 6, which will select numbers from 0 to 5. To

avoid using black as well (color #0), use `1 + RANDOM 5`. This gives you a random number from 1 to 5 because 1 is always added to a random number from 0 to 4.

Try adding one of these lines to one of your procedures:

```
PC 1 + RANDOM 5  
BG 1 + RANDOM 5
```

Note that the number used with PC (PENCOLOR) and BG (BACKGROUND) is the result of an arithmetic operation again, addition this time. Recall that some of the turns in your shape procedures were calculated by division.

Any time a number is required in Logo, it can be given as the result of an arithmetic operation. In Logo, use + and - for addition and subtraction (as usual), the slash (/) for division, and the star (\*) (or asterisk) for multiplication. There are rules you need to know if you use more than one operator (+-/\*) at a time; see the COMPUTATION chapter for details on that.

### *Projects Using Random*

1. Substitute `FORWARD RANDOM 100` for the side in `SQUARE3`.
2. Write a `REPEAT` statement using a `FORWARD` command and a random turn from 0 to 360 degrees.
3. Write a recursive procedure using a `FD` command and a random turn between 90 and 180 degrees.
4. Try some other ranges for turns; choose the most interesting to keep as a procedure.

### *Debugging by printing values: PRINT (PR)*

Logo is one of the easier computer languages to debug (get rid of the errors, called bugs) because large programs are composed of small procedures. It is a lot easier to debug a small procedure than a long, complicated program. Always make sure your procedures are debugged (run correctly by themselves) before you use them in other procedures.



```
TO DESIGN :TIMES :LENGTH
  IF :TIMES = 0 THEN STOP
  SQUARE :LENGTH
  RT 45
  DESIGN :TIMES-1 :LENGTH
END
```

In DESIGN, if you type

```
DESIGN 6.5 100
```

the procedure will never stop.

To find out why, we want to check out :TIMES. It would be nice to print it out each time around.

Use the Logo PRINT (PR) command to check on the value of :TIMES. Type

```
TO DESIGN
```

and add this line (in EDIT mode) just before the test (before IF ...):

```
PR :TIMES
```

(You can remove it after you have debugged the procedure.)

DESIGN now looks like this:

```
TO DESIGN :TIMES :LENGTH
  PR :TIMES
  IF :TIMES = 0 THEN STOP
  SQUARE :LENGTH
  RT 45
  DESIGN :TIMES-1 :LENGTH
END
```

Type <CTRL> C to leave EDIT mode, then type

DESIGN 6.5 100

As it runs DESIGN, Logo will draw the design in the graphics part of the split-screen, and will print the values of :TIMES on the four lines of the text part of the screen.

Because the values are not whole numbers, if you look quickly, you will see them get smaller and smaller and then become negative and get larger and larger. In other words, :TIMES has passed zero and skipped the test because :TIMES was never exactly zero.

Now you know that the bug is in the test that failed to account for this possibility. You can either change the test or add another test. The best thing to do is change the test, since two tests are not really necessary. However, when you change the test, be sure to try out DESIGN with every possibility you can think of. ALWAYS test your procedures using all of the possibilities you can think of.

## ***Debugging Using PAUSE: <CTRL> Z CONTINUE (CO)***

PAUSE or <CTRL> Z stops a procedure in such a way that you can start it again. While it is stopped, you can find out where the (hidden) turtle is by typing SHOW-TURTLE (or ST), hide the turtle with HT, print the procedure out with PO, PRINT variable values, or do a number of other things. To resume running the procedure, type CONTINUE (or CO).

## ***Negative Inputs***

There is also another possibility: remember that friend of yours who likes negative inputs? What happens to DESIGN if :LENGTH is negative? What happens to :TIMES? What happens to the friend?

Well, if :LENGTH is negative, the turtle just backs around in the opposite direction. Logo knows all about negative lengths.

And the friend? Unless he knows how to give that negative input, Logo will give him a (no doubt helpful) error message.



A negative input for the second variable must be in parentheses to show that it is an input and not a number to be subtracted from the first variable, for, as you will recall, inputs can be the results of arithmetic operations. Type

```
DESIGN 5 (-100)
```

Let's set up a situation where the size of the turn between squares depends on the number of :TIMES the

square is drawn, so we can have a complete design. To do this, we replace the 45 with  $360 / :TIMES$ .

```
TO DESIGN :TIMES :LENGTH
  IF :TIMES < 1 THEN STOP
  SQUARE :LENGTH
  RT 360/:TIMES
  DESIGN :TIMES-1 :LENGTH
END
```

Now we have two things which depend on `:TIMES`, `:TIMES` itself, which must always be positive, and the turn between squares, which could be either positive or negative. A negative turn just goes around in the other direction.

How can we fix it so a negative number for `:TIMES` will give us a positive value for `:TIMES`, but keep the negative turn?

To do this, we must write a procedure to test `:TIMES`, then call `DESIGN` with the appropriate values. We also need to use a variable for the turn, so we can keep it negative when `:TIMES` changes to positive. `DESIGN` becomes

```
TO DESIGN :TIMES :LENGTH :TURN
  IF :TIMES < 1 THEN STOP
  SQUARE :LENGTH
  RT :TURN
  DESIGN :TIMES-1 :LENGTH :TURN
END
```

`COMPLETE.DESIGN` is the procedure which handles the details:

Type as one line

```

TO COMPLETE.DESIGN :TIMES :LENGTH
  IF :TIMES < 0 THEN
    DESIGN - :TIMES :LENGTH 360/:TIMES
  ELSE
    DESIGN :TIMES :LENGTH 360/:TIMES
END

```

This is a one-line procedure, shown here on several lines for clarity. It must be typed as one line.

This says that if `:TIMES` is negative, change it to positive when you call `DESIGN`, otherwise leave it alone. In both cases, `:TURN` uses `:TIMES` directly, so if `:TIMES` is negative, `:TURN` is negative; if `:TIMES` is positive, `:TURN` is positive.

### *More on Debugging: TRACE, NOTRACE*



Logo provides a detective system to trace through the procedure with you as the procedure is executed. Logo prints each line on the screen, you press `<RETURN>`, and Logo executes the line. Type `TRACE` to activate `TRACE` mode, `NOTRACE` to get out of it. See the Appendix for a full description of `TRACE` and `NOTRACE`.

### *More About the Turtle: TURTLESTATE (TS), HEADING, SETHEADING (SETH), TOWARDS*

Logo primitives which give information about the turtle are useful for testing. `TURTLESTATE` is a good example, giving a list of four pieces of information.

Type

TURTLESTATE and Logo will reply  
RESULT: [TRUE TRUE 0 1]

- if 1. It is TRUE that the pen is down
- 2. It is TRUE that the turtle is visible
- 3. Background color is 0 (black)
- 4. Pencolor is 1 (white)

Refer to the chapter on Words and Lists for how to test against a member of a list. You can also print the information, i.e. PRINT TURTLESTATE.

Logo uses HEADING for the direction the turtle is pointing. Type

HEADING and Logo will reply  
RESULT: 45.007

or whatever number of degrees the turtle has turned to the right (clockwise) from facing up.

PRINT HEADING, whether used in a procedure or not, will print the number alone. You can use HEADING to stop a procedure after a turn. Example:

```
IF HEADING < 45 STOP
```

Use SETHEADING (SETH) to tell Logo what direction you want the turtle to face:

```
SETHEADING 45
```

turns the turtle as if it had turned 45 degrees to the right from facing straight up.

To change the turtle's heading by a specific amount, use both:

```
SETHEADING HEADING + 5
```

will turn the turtle 5 degrees to the right.

TOWARDS turns the turtle to face a point designated by its coordinates:

```
SETHEADING TOWARDS 100 (-100)
```

turns the turtle to face a point 100 turtle steps to the right ( $x = 100$ ) and 100 turtle steps down ( $y = -100$ ) from the center of the screen. Note that here, too, the negative input is in parentheses to avoid confusion with subtraction. Another way to write a negative second input is to write it as zero minus the number. Example:

```
SETHEADING TOWARDS 100 0-100
```

### ***Position When You Want It: XCOR, YCOR, SETX, SETY, SETXY***

The graphics screen can be thought of as a grid, with X going across and Y going up and down. At the HOME position in the center of the screen, X and Y are zero. X gets larger to the right; Y increases as you go up. X is negative to the left of HOME, and Y is negative below it.

XCOR and YCOR give the X and Y coordinates of the turtle's position on this grid. Type XCOR, YCOR,

**PRINT XCOR**, or **PRINT YCOR** and Logo will print the X or Y coordinate. You may also test against either:

**IF XCOR = 150 STOP**

To move the turtle to a specific coordinate position, use **SETX**, **SETY**, or **SETXY**. Only the position will change; the turtle will not change its heading. Type:

**SETX 100**  
to move the turtle across to  $x = 100$

**SETY 100**  
to move the turtle up or down to  $y = 100$

**SETXY 100 100**  
to move the turtle to the point  $x = 100, y = 100$

**SETXY 100 (-100)**  
to move the turtle to  $x = 100, y = -100$

Use these commands together to move the turtle a specific distance:

**SETX XCOR + 5**

moves the turtle 5 steps to the right without changing its heading.

**SETXY XCOR + 5 YCOR - 5**

moves the turtle 5 steps to the right and 5 steps down, keeping the same heading.

**SETXY** is used in the Computation chapter to draw curves using their equations. To see how to use **SETXY** with joysticks and paddles, see **PADDLE** in the Logo Command Glossary.

## ***INSTANT: Logo Turtle Graphics for the Non-reader***

Your Logo system disk contains a collection of procedures which makes Logo turtle graphics accessible to young children. The INSTANT system uses single character commands which are equivalent to longer Logo commands. You can use colored stickers to identify the appropriate keys for use with INSTANT.

To use INSTANT, turn on the Apple with the Logo Language disk in the disk drive. When Logo is loaded and displaying the question-mark prompt (?), put the Utilities Disk in the disk drive and type

READ "INSTANT (with the “)

Logo will read in the file of procedures used by INSTANT, identifying each as defined. Type

INSTANT (without the “)

The screen will display the commands used in INSTANT as follows:

F MOVES THE TURTLE FORWARD  
R TURNS IT RIGHT  
L TURNS IT LEFT  
D DRAW (CLEARS THE SCREEN)  
U UNDO (ERASES LAST COMMAND)  
N NAMES THE PICTURE  
P SHOWS A PICTURE, ASKS FOR ITS NAME.  
? GIVES HELP

PRESS ANY KEY TO CONTINUE.

When you press a key, the list goes away, the turtle appears, and the blinking cursor moves to the lower left portion of the screen.

Type F to move the turtle forward. Turn the turtle with either R or L.

D restores the screen to its original condition, erasing the whole picture.

To erase just the last command, type U. Logo will re-draw the picture without the most recent command.

### Animation Of A Sort



U makes it possible to do some interesting animation, since every motion of the turtle is relived in the re-drawing, even though it is not visible in the finished drawing. For a Lively Line, try typing

```
F R L L R   F R L L R   F R L L R   U
```

The idea is to wave the turtle back and forth every once in a while, perhaps turn it completely around; let it be indecisive about making a turn... It all comes out again when you type the U.

To name a picture, type N and the name. (Names do not have to be single letters; they can be of any length.) INSTANT will create a Logo procedure which has that name and contains all the steps used in drawing the picture.

To get a picture back, type P and its name. When the picture-drawing procedure is called using P, it is added to the current list of commands and becomes part of a new procedure when N is next used. Using N and P in this way, you can do structured programming in INSTANT.

The following INSTANT session demonstrates this feature. Although the single-letter INSTANT commands do not print out when you type them, they are shown here for convenience. The computer's responses appear in italics.

```
F
F
R
R
N
WHAT DO YOU WANT TO CALL THIS PICTURE?
SIDE
P
WHAT PICTURE DO YOU WANT TO SHOW?
SIDE
P
WHAT PICTURE DO YOU WANT TO SHOW?
SIDE
P
WHAT PICTURE DO YOU WANT TO SHOW?
SIDE
P
WHAT PICTURE DO YOU WANT TO SHOW?
SIDE
N
WHAT DO YOU WANT TO CALL THIS PICTURE?
BOX
```

If you leave INSTANT and print out the procedures SIDE and BOX, you can see that they are basically the same procedures developed in the beginning of this chapter, with minor differences such as three RIGHT 30 commands being used instead of RIGHT 90.

To save a picture on the disk, return to Logo with <CTRL> G and type SAVEPICT " and the name you want for your picture. Example:

SAVEPICT "PUPPY

will save the picture on the screen under the name PUPPY on the disk.

Type

INSTANT

to return to the INSTANT system.

For disk storage of procedures created using INSTANT, you must leave INSTANT and return to Logo:

1. Type <CTRL> G to return to Logo.
2. Type <CTRL> T for the full screen of text (TEXT mode)
3. Type POTS to list the procedures you have created (plus the system procedures you saw defined as they were read in)
4. To write all of the listed procedures to your disk, put your procedure-storage disk in the disk drive, and type

SAVE "INSTANT

All the procedures listed will be written to your disk. In subsequent sessions using `INSTANT`, `READ` “`INSTANT`” from your own disk instead of the Utilities disk. You will have everything you need to run `INSTANT` as well as all previously written original procedures.

If you want only the procedures created by `INSTANT`, you can use `SAVE` with two inputs: a filename and a list of procedures. See the earlier section on Saving Procedures.

### *Modifying INSTANT*

Like many of the programs on the Utilities Disk, `INSTANT` can be expanded or modified to include more complex commands. To add new single-letter commands, edit the `COMMAND` procedure.

**C**OMPUTATION:  
HANDLING NUMBERS

# COMPUTATION: HANDLING NUMBERS

## C

Perhaps you've begun to wonder if Logo can do anything but draw pictures. Have no fear! Like any other full computer language, Logo can perform a variety of numerical operations



Logo uses integers (whole numbers like 4, 67, 1918) and real numbers (numbers with a decimal part like 4.55, 3.14159) without distinguishing between them.  $7/2$  (7 divided by 2) is always 3.5 in Logo.

---

Logo also recognizes "floating point" numbers, which use a form of scientific notation. For instance, 5000 can be represented as  $5E3$  (5 times ten to the exponent 3); likewise, 0.005 can be represented as  $5N3$  (5 times ten to the exponent -3). Floating point notation is useful primarily in representing extremely large numbers. See the Numbers section of the Technical chapter for more details.

---

## *Arithmetic Operations*

When you use a computer, you must type everything on one line. For the operations of addition, subtraction, multiplication, and division, Logo uses the following operators:

		as in
Addition	+	$7 + 5$ (12)
Subtraction	-	$7 - 5$ (2)
Multiplication	*	$7 * 5$ (35)
Division	/	$7 / 5$ (1.4)

The star (or asterisk  $*$ ) is used for multiplication to avoid confusion with the letter x. The slash ( $/$ ) is used to keep division on one line.

Raising to powers (exponentiation) uses the procedure EXPONENT, described below.

Logo will do the arithmetic for you when you give it an operation for its input. When you type:

FD 26 + 42 Logo will move the turtle 68 steps forward;

PRINT 76 \* 42 Logo will print 3192;

RT 360/5 Logo will turn the turtle 72.

### *Hierarchy of Operations*



Doing arithmetic on a line does present some problems, however. There must be rules about which operation is done first. Try these:

PRINT (7 + 5) / 2  
PRINT 7 + 5 / 2

In the first, the 7 and 5 are added, to make 12, then the 12 is divided by 2, which gives 6. In the second, the 5 is divided by 2 first, with the result of 2.5, then the 2.5 is added to the 7, giving 9.5.

#### RULES THE COMPUTER PLAYS BY



1. Parentheses are the first thing the computer looks for in evaluating an arithmetic expression. It does whatever is in the parentheses first, according to the rest of the rules.
2. Multiplication and division are done next, from left to right.

3. Addition and subtraction are done last, also from left to right.

Examples:

1.

$$4 * 3 + 6 / 3 - 2 * (3 + 2)$$

Step 1	4	*	3	+	6	/	3	-	2	*	(	3	+	2)

2.

$$4 * (3 + 6) / (3 - 2) * 3 + 2$$

Step 1	4	*	(	3	+	6)	/	(	3	-	2)	*	3	+	2

3.

$$4 * (3 + 6) / ((3 - 2) * 3 + 2)$$

Step 1	4	*	(	9	)	/	((	3	-	2)	*	3	+	2)

So you see, the order in which the operations are done can make a considerable difference.

***Outputs, Integer Operators, Functions:  
RANDOM, RANDOMIZE, ROUND, INTEGER,  
QUOTIENT, REMAINDER, SQRT, SIN, COS***

Arithmetic operations give a result, called an output. When you type an operation at the keyboard, Logo will tell you that result. Type

24/3                      and Logo will type  
RESULT: 8

RANDOM is another Logo operation which gives a result. It chooses a random number in the group you select. You specify the group by giving RANDOM the next higher number.

Type	Logo will output
RANDOM 10	a number between 0 and 9;
RANDOM 501	a number between 0 and 500.

(Type RANDOMIZE before using RANDOM to avoid identical sequences of random numbers every time you turn on the computer.)

The other integer operators also output. ROUND rounds off a real number to the closest integer:

ROUND 6.4	outputs 6
ROUND 2.7	outputs 3
ROUND -6.4	outputs -6
ROUND -2.7	outputs -3
ROUND 6.5	outputs 7

**INTEGER** gives the integer portion of a real number:

<b>INTEGER 4.3</b>	outputs 4
<b>INTEGER 4.9</b>	outputs 4
<b>INTEGER -4.3</b>	outputs -4
<b>INTEGER -4.9</b>	outputs -4
<b>INTEGER 7/2</b>	outputs 3

**QUOTIENT** gives the integer part of the quotient of two numbers:

<b>QUOTIENT 7 2</b> (the same as <b>INTEGER 7/2</b> )	outputs 3
<b>QUOTIENT 1 2</b>	outputs 0
<b>QUOTIENT -7 2</b>	outputs -3

**REMAINDER** outputs what is left over from the integer division:

<b>REMAINDER 7 2</b>	outputs 1
<b>REMAINDER 2 3</b>	outputs 2

When you use real numbers with **QUOTIENT** or **REMAINDER**, they are **ROUNDED** to integers before the division takes place.

**SQRT** produces the square root of the positive number you give it:

<b>SQRT 160</b>	outputs 12.6491
<b>SQRT 16</b>	outputs 4

SIN and COS output the sine and cosine of the number given in degrees:

SIN 0	outputs 0
SIN 90	outputs 1
COS 0	outputs 1
COS 90	outputs 0

In a procedure you must do something with an output. If you don't, Logo complains that you don't say what to do with it. You might PRINT it, assign it to a variable name, or use it in a graphics command:

RT 360/4	the turtle turns right 90
MAKE "A 360/4	the value of A becomes 90
PRINT :A	Logo prints 90
PRINT QUOTIENT 5 2	Logo prints 2
MAKE "B REMAINDER 5 2	:B becomes 1
PRINT :B	Logo prints 1

### *Variables, Global and Local: MAKE*

In Logo, you may use a variable anywhere you can use a number.



Variable names in Logo may be of any length, made up of any combination of letters, numbers, or the special characters !,"#\$.%&? which leave out only the operators, brackets and parentheses, and the single quote.



The name of the variable is preceded by the single set of double quotes (""). The value of the variable is preceded by dots (: also known as colon).

### Global Variables:

The Logo primitive **MAKE** gives a value to a variable which the variable keeps until it is changed with another **MAKE** command. **MAKE** can be used either in **IMMEDIATE** mode or in a procedure. The value assigned is used in any procedure in which the variable is used; the value is also stored when a copy of the workspace is saved onto a disk. Variables created with **MAKE** are called Global Variables. Examples:

```
MAKE "PI 3.14159 gives the variable :PI the value 3.14159
```

```
PRINT :PI prints 3.14159
```

```
MAKE "MINE "MINK gives :MINE the value MINK
```

```
PRINT :MINE prints MINK
```

```
MAKE "A :PI gives :A the VALUE of :PI (3.14159)
```

```
PRINT :A prints 3.14159
```

### Local Variables:



Local variables are used only in procedures. When a procedure is running, its local variable(s) have a value. When the procedure stops, the variable ceases to exist until the next time the procedure is run. An input to a procedure behaves as a local variable.

You can also make a variable local to the current procedure with the **LOCAL** command (added in version 2.0). Any time **MAKE** is used after the command **LOCAL**, the variable is treated as a local and not a global variable.

---

Local variables are important because they keep the workspace from becoming cluttered. Using global variables when local variables will do wastes memory space that could be used for other purposes. Also, you can have several local variables with the same name in different procedures, but only one global variable with a given name.

---

### *Procedures: TO, END*

Any command you can type at the keyboard can be used in a Logo procedure. To define a procedure, type TO and the name you have chosen. For example, type:

TO CUBE (to obtain a number multiplied by itself 3 times)



The screen will clear, with the procedure title TO CUBE at the top and a white line at the bottom which tells you that you are in EDIT mode and should use <CTRL> C to complete the definition of your procedure. (<CTRL> G means gone.) (To do a <CTRL> C, hold down the <CTRL> key and press the <C> key.)

Type in the rest of the procedure below, and press <CTRL> C. (See the APPENDIX for a discussion of commands used in EDIT mode.)

```
TO CUBE
  PRINT 4 * 4 * 4
END
```

Type

CUBE

Logo will print 64

You can use a variable to extend the usefulness of this procedure. Make it print the cube of whatever number is given it, instead of printing the cube of 4 all the time. Replace each 4 with the variable name and add it to the title, so the value of the variable may be brought into the procedure. You may choose any name for your variable; a descriptive one is most helpful.

```
TO CUBE :NUMBER
  PRINT :NUMBER * :NUMBER * :NUMBER
END
```

CUBE now expects a number. This means that you may not type CUBE by itself any more. When you do, Logo will tell you that you forgot the input (:NUMBER).

Now when we type CUBE with a number, Logo will print the cube of that number.

Type

Logo will print

CUBE 3	27
CUBE 33	35937
CUBE 333	36926037

After CUBE is run, Logo forgets the value of :NUMBER. Try typing

```
PRINT :NUMBER
```

:NUMBER is a local variable and has value only within the procedure in which it is used. :NUMBER could be used in a variety of procedures and have a different value in every one.

### *Interactive Procedures: LOCAL, REQUEST (RQ)*

LOCAL is convenient when you don't want to give a procedure input immediately, but still want to use a local variable. This is frequently the case with interactive procedures, especially if the procedure requires the user to input more than one variable.

An interactive procedure is one that requires user input from the keyboard while the procedure is running. As an example of this, we start with a procedure which randomly picks two numbers and prints their product.

```
TO MULTIPLY
  LOCAL "X LOCAL "Y
  MAKE "X RANDOM 10
  MAKE "Y RANDOM 10
  (PRINT :X [TIMES] :Y [IS] :X * :Y)
END
```

LOCAL specifies that its input (in this case X and Y) be treated as local variable for the rest of the procedure. In the procedure shown, RANDOM is used to pick values for the variables. The last line then prints the variables and their product as part of a sentence.

Note that X and Y do not have values until MAKE is used. To see this, put PRINT :X between the LOCAL and MAKE statements.

---

PRINT usually takes one input which can be a word, a list, or a number. In this case it has five inputs, so parentheses must be used to tell Logo to expect more than the usual number of inputs.

---

To make the procedure continue to pick variables and print answers, add MULTIPLY as the last line in the procedure (but before END). Use <CTRL>G to stop it.

Now you have a procedure that is good at picking numbers and telling you the product, but this isn't an interactive procedure. There is no way for you to do anything while the procedure is running. Let's change things so that you have to type the answer to a question. The line with PRINT will become

```
(PRINT [HOW MUCH IS] :X [TIMES] :Y [?])
```

Of course, you'll want Logo to tell you whether the answer you give is right or wrong. The following procedure will do just that.

```
TO TESTANSWER :ANSWER  
  IF :ANSWER = :X * :Y PRINT [CORRECT] STOP  
  PRINT [INCORRECT]  
END
```

This procedure looks to see if the value of ANSWER equals  $:X * :Y$ . If this is TRUE, the procedure prints CORRECT and stops; otherwise, it prints INCORRECT.

How should you combine the two procedures? If you add TESTANSWER :ANSWER as the line after the PRINT command in MULTIPLY, where does :ANSWER come from?

To allow user input, use REQUEST. This primitive takes input from the keyboard and makes it into a list when <RETURN> is hit.

```
TO MULTIPLY
  LOCAL "X LOCAL "Y
  MAKE "X RANDOM 10
  MAKE "Y RANDOM 10
  (PRINT [HOW MUCH IS] :X [TIMES] :Y [?])
  TESTANSWER FIRST REQUEST
END
```

REQUEST takes what you type and gives it to TESTANSWER as input. The command FIRST is needed because REQUEST makes a list; if FIRST were omitted, the first line of TESTANSWER would compare a number with a list containing a number, and the procedure would print INCORRECT. What we need is the first (in this case, the only) item in the list, which is the number you typed in.

---

Note that TESTANSWER uses X and Y even though they were not declared in its title line. A subprocedure always has access to the variables in the calling procedure(s).

---

So far MULTIPLY and TESTANSWER could have been written as one procedure. But what if we wanted to make the program keep asking for an answer until it got the right one? To do this, we need a recursive call to TESTANSWER so it will keep calling itself until you type the correct answer.

What happens if you add TESTANSWER :ANSWER as the last line of TESTANSWER? Obviously this doesn't work! (Try it to see why.) We need to change :ANSWER in the last line to something else. Hint: look at MULTIPLY.

Why all this fuss about local variables, whether created by LOCAL or declared as procedure inputs in the title line? Global variables take up space. Unless you have a particular need for a variable that stays around in the workspace, use a local variable.

### *Bringing Values Out of Procedures: OUTPUT (OP)*



When the results of running a procedure are to be used by another procedure, which often happens when the purpose of a procedure is doing a computation, those results must be brought out of the procedure for use.

There are two ways of getting values out of a procedure:

1. Create a global variable (described above).
2. Use the Logo primitive OUTPUT.

The Logo primitive OUTPUT returns values from the procedure in which it occurs. The values are returned to the procedure which called that procedure.

If you run a procedure which uses OUTPUT, the procedure will print the OUTPUT on the screen.

If you run a procedure which calls a procedure which uses OUTPUT, only the procedure you ran will receive the information from OUTPUT. It will not be printed unless there is a PRINT statement.

This is similar to what happens when you do arithmetic operations. Type

`3 + 5`

and Logo will print

RESULT: 8

Type

`FORWARD 3 + 5`

and the result 8 only goes to the FORWARD

The turtle moves, but the 8 is not printed on the screen.

We can change the PRINT statement in CUBE to OUTPUT to show this:

```
TO CUBE1 :NUMBER
  OUTPUT :NUMBER * :NUMBER * :NUMBER
END
```

Now if you type

```
CUBE1 3
```

Logo will print

```
RESULT: 27
```

However, if you type

```
FORWARD CUBE1 3
```

the graphics turtle will move forward 27 steps.

***Example of OUTPUT and Recursion:  
A Procedure to Do Exponentiation***

A recursive procedure is a procedure which calls itself as a subprocedure. The procedure EXPONENT, shown below, uses recursion to raise :X to the power of :Y.

```
TO EXPONENT :X :Y
  IF :Y = 0 THEN OUTPUT 1
  OUTPUT :X * (EXPONENT :X :Y-1)
END
```

In the procedure, Y is used as a counter to make sure that X is multiplied together the correct number of times.

How EXPONENT works:

1. Tests for the finish, i.e.  $Y = 0$
2. Multiplies :X by the result of running EXPONENT with the counter decremented.
  1. Tests for the finish
  2. Multiplies :X by the result of running EXPONENT with the counter decremented, and so on until :Y is decremented to 0.

Example:

EXPONENT 3 4

We shall follow the recursion down through all its levels and then trace OUTPUT on its way back up.

Going down, each level is put on hold pending the appearance of a number needed to complete the computation. Coming back up, each number is output to the level above and each computation completed.

Going down:

EXPONENT 3 4

1. Check to see if :Y (4) is 0
2. OUTPUT 3 \* the result output by EXPONENT 3 3

Logo must figure out the value of EXPONENT 3 3.

EXPONENT 3 3

1. Check to see if :Y (3) is 0
2. OUTPUT 3 \* the output of EXPONENT 3 2

Logo must figure out the value of EXPONENT 3 2.

EXPONENT 3 2

1. Check to see if :Y (2) is 0
2. OUTPUT 3 \* the output of EXPONENT 3 1

Logo must figure out the value of EXPONENT 3 1.

EXPONENT 3 1

1. Check to see if :Y (1) is 0
2. Output 3 \* the output of EXPONENT 3 0

Logo must figure out the value of EXPONENT 3 0.

EXPONENT 3 0

1. Check to see if :Y (0) is 0; if it is, OUTPUT 1. OUTPUT stops the procedure and outputs the value 1.

Going back up:

The 1 is output to the procedure which called EXPONENT 3 0, which was EXPONENT 3 1. This completes the evaluation in EXPONENT 3 1, which is output to the procedure which called EXPONENT 3 1, which was EXPONENT 3 2. The process is repeated until the top level is reached.

The evaluation of EXPONENT 3 4 on the way down looks like this:

$$\begin{aligned} \text{EXPONENT 3 4} &= 3 * (\text{EXPONENT 3 3}) \\ &= 3 * (3 * (\text{EXPONENT 3 2})) \\ &= 3 * (3 * (3 * \text{EXPONENT 3 1})) \\ &= 3 * (3 * (3 * (3 * \text{EXPONENT 3 0}))) \end{aligned}$$

Since the value output by EXPONENT 3 0 is 1, going back up this becomes

$$\begin{aligned} \text{EXPONENT 3 4} &= 3 * (3 * (3 * (3 * (1)))) \\ &\quad \text{EXPONENT 3 0 outputs 1} \\ &= 3 * (3 * (3 * (3 * 1))) \\ &\quad \text{EXPONENT 3 1 outputs 3} \\ &= 3 * (3 * (3 * 3)) \\ &\quad \text{EXPONENT 3 2 outputs 9} \\ &= 3 * (3 * 9) \\ &\quad \text{EXPONENT 3 3 outputs 27} \\ &= 3 * 27 \\ \text{EXPONENT 3 4 outputs 81} \end{aligned}$$

The 3 is multiplied by itself 4 times, just as prescribed.

Note the use of :Y as a counter which makes sure that EXPONENT is called exactly 4 times, that is, 3 is multiplied by itself 4 times, or raised to the power of 4.

*Graphing Functions:  
Sine, Cosine, Tangent, Parabola, Ellipse,  
SETXY, HOME, DRAW, HT*

It is easy to graph functions of the form  $Y = f(X)$  using the Logo primitive SETXY, which takes as its inputs the :X and :Y positions on the Logo screen.

The heart of each procedure is the evaluation of :Y and the positioning of the turtle ( $f(:X)$  is whatever the function calls for):

```
MAKE "Y f(:X)
SETXY :X :Y
```

This is more elegantly accomplished by combining the two operations. For example:

```
SETXY :X f(:X)
```

Principal considerations include

1. Keeping the curve on the screen
2. Positioning the curve
3. Scaling for visibility

To position the start of the curve, we might want to move :X to the left by the amount :C. Our statement becomes:

```
SETXY :X-C f(:X)
```

To see :Y if it is very small, we might want to multiply it by a visibility factor :D:

```
SETXY :X - :C f(:X) * :D
```

SINE FUNCTION:  $Y = \sin X$

We would like to begin the sine wave at the left edge of the screen (-140), make it large enough to be visible, and stop at the right edge of the screen (+140).

To begin drawing at the left edge and yet have :X start at 0 for the evaluation of :Y, the X position becomes :X-140.

To see :Y, which will vary between 0 and 1, multiply by 100 (actually anything up to 120, the vertical limits of the screen).

The procedure starts out as

```
TO GRAPH.SIN :X
  SETXY :X - 140 100 * SIN :X
END
```

This computes one point and moves the turtle to it. To continue the curve, increment :X by calling GRAPH.SIN with an incremented value:

```
TO GRAPH.SIN :X
  SETXY :X - 140 100 * SIN :X
  GRAPH.SIN :X + 5
END
```

To stop the curve at the right edge of the screen, insert a test for the X position (:X-140):

```
TO GRAPH.SIN :X
  IF :X - 140 > 140 STOP
  SETXY :X - 140 100 * SIN :X
  GRAPH.SIN :X + 5
END
```

To draw a sine wave starting at  $X = 0$ , type

```
GRAPH.SIN 0
```

An axis would improve the graph (DRAW clears the screen and moves the turtle to the center, HT hides the turtle):

```
TO AXIS
  DRAW
  HT
  SETXY 140 0
  HOME
  SETXY -140 0
END
```

Now to draw a sine wave with an X-axis, type

```
AXIS
GRAPH.SIN 0
```

The final improvement for the sine wave is writing a procedure to do that typing for us:

```
TO SINE
  AXIS
  GRAPH.SIN 0
END
```

Finally, to draw a sine wave, type

```
SINE
```

COSINE FUNCTION:  $Y = \cos X$

Substitute COS for SIN in the GRAPH.SIN procedure, changing its name to GRAPH.COS. Write a superprocedure COSINE to call it with AXIS. The easiest way to do this is to edit GRAPH.SIN and SINE. See the editing sections of the Graphics chapter and the APPENDIX.

TANGENT FUNCTION:  $Y = (\sin X) / (\cos X)$

The tangent procedure has some different problems.

Note how :X is incremented slightly if COS :X = 0, to avoid dividing by 0. Since we don't want to stop the procedure in the middle of the screen, PU (PENUP) is used to stop the turtle from drawing when it is simply wrapping around the screen to get to the off-screen points. (When the line goes off the edge of the screen, it continues by entering on the opposite side of the screen. This is called wrapping.) Using PU requires adding PD (PENDOWN) to start drawing again.

```

TO GRAPH.TAN :X
  IF COS :X = 0 THEN MAKE "X :X + 1
  IF :X - 140 > 140 STOP
  MAKE "Y (SIN :X) / (COS :X)
  IF 100 * :Y > 115 PU
  IF 100 * :Y < -115 PU
  SETXY :X-140 :Y * 100
  PD
  GRAPH.TAN :X + 5
END

```

Here Y is evaluated separately because it must be tested before the drawing step.

PARABOLA:  $Y = (X * X) / (4 * A)$

The vertex of this parabola is at 0,0; the axis is vertical. A is the distance from the vertex to the focus. Increasing A makes a wider parabola; decreasing it makes a narrower one.

The general formula for this parabola is

$$(X-H) * (X-H) = 4 * A * (Y-K)$$

where H is the X co-ordinate and K is the Y co-ordinate. H and K are 0 in this example.

In the drawing of the parabola, add PU after AXIS to avoid leaving a trail to the beginning of the curve. (This is the same AXIS procedure that is used in the sine procedure.)

Determine the beginning point in the superprocedure PARABOLA, using the equation again, with 118 the value for Y (about the largest possible value for Y).

```
TO PARABOLA :A
  AXIS
  PU
  GRAPH.P (SQRT (118 * 4 * :A)) :A
END
```

```
TO GRAPH.P :X :A
  MAKE "Y (:X * :X) / (4 * :A)
  IF :Y > 124 STOP
  SETXY :X :Y
  PD
  GRAPH.P :X + 5 :A
END
```

With a positive value for :A, this will draw a parabola above the X axis. To allow use of a negative :A, which would draw a parabola below the X axis, we must use the absolute value of :A (:A without its sign) in calculating the starting position of X, since we cannot take the squareroot of a negative number. We write the procedure ABS to figure the absolute value for us:

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

OUTPUT stops after it outputs. So if X is negative, it will change it to positive; if it is positive it will output it directly. PARABOLA becomes:

```

TO PARABOLA :A
  AXIS
  PU
  GRAPH.P (-SQRT(118 * 4 * ABS :A)) :A
END

```

Since it is a test for Y that stops the procedure, we must revise the test to allow for a negative Y:

```

TO GRAPH.P :X :A
  MAKE "Y (:X * :X) / (4 * :A)
  IF ANYOF (:Y > 124) (:Y < -124) STOP
  SETXY :X :Y
  PD
  GRAPH.P :X + 5:A
END

```

To make a family of parabolas, add a recursive call to PARABOLA (taking care to pick up the pen in between):

```

TO PARABOLA :A
  AXIS
  PU
  GRAPH.P (-SQRT(118 * 4 * ABS :A)) :A
  PU
  PARABOLA :A + 1
END

```

```

ELLIPSE FUNCTION:
Y = B * SQRT (1-(X * X) / (A * A))

```

The center of this ellipse is at 0,0. A is half of the horizontal axis, B is half of the vertical axis.

The general formula for an ellipse is

$$(X-H) * (X-H) / (A * A) + (Y-K) * (Y-K) / (B * B) = 1$$

where H,K are the X and Y co-ordinates of the center, (0,0) in this example.

The ellipse procedure must solve the problem of Y becoming negative as X returns to its original value. Changing the sign of the increment takes care of it.

```
TO GRAPH.ELLIPSE :X :A :B :INC
  IF (:X * :X) > (:A * :A) STOP
  IF :X = :A THEN MAKE "INC (-1)
  SETXY :X :INC * :B * SQRT (1-(:X * :X) / (:A * :A))
  PD
  GRAPH.ELLIPSE :X + :INC :A :B :INC
END
```

The SETXY command must be typed as one line. Use the same AXIS program as you used with the sine procedure.

```
TO ELLIPSE :A :B
  AXIS
  PU
  GRAPH.ELLIPSE - :A :A :B 1
END
```

WORDS & LISTS



## **INTRODUCTION**

So far, all of the procedures that we have described or encouraged you to write have been non-interactive. That is, once you started them, they did what they were designed to do without consulting you further. The most you might ever have done was press <CTRL> G to stop them.

Interactive programs are perhaps the most fun of all, precisely because they interact. They are also potentially the most complex. The reason for this is that while they are underway, they must account for the unpredictable behavior of the person with whom they are interacting.

Interactive movement forms the basis for a variety of video games and simulations. Interactive language adds attractive features to these games, but it can also open up a whole new interest area: mad-libs, quizzes, word games, conversational programs that construct grammatical sentences and “understand” limited amounts of natural language, even foreign languages.

There are two ways you can approach this chapter. You may prefer to go quickly through, skipping all of the indented text, or you may wish to study those portions as you work your way through the chapter. As in other chapters, the indented portions add depth and detail to the presentation.



The procedures you are asked to type in are used throughout the chapter, so be sure to save them on your disk when you decide to take a break, and be sure to read them back in when you start to work on the chap-

ter again. (CHAPTERW would be an appropriate file name, so you can type SAVE "CHAPTERW and READ "CHAPTERW.)

In the graphics chapter, you learned about procedures which had an immediate and visible effect. FD moved the turtle (and left a trace on the screen if the pen was down), DRAW cleared the screen, and so on.

This meant that even without writing procedures, you were able to give Logo several commands in succession and see what their combined effect was. You may even have forgotten what commands you used, but the screen "remembered" their effect.

Procedures spared you considerable typing. They also gave you a way of recording the instructions for your designs. But the designs themselves didn't depend on the procedures. They would have grown just as surely on the screen if you had typed each turtle command line by line.

In this chapter, you will be learning about primitives that manipulate Logo "objects." The effects of these primitives are not graphic and do not accumulate unless you explicitly instruct them to.

These primitives can be explained and used one by one, but their real power is most apparent in combination. As a result, the focus of this chapter must be on building procedures which combine these primitives in different and varied ways.

Even though there are only roughly a dozen important new primitives, and even though only about half are

used with much frequency, there are many, many combinations which can be used in creating sophisticated and interesting programs.

Here are some of the programs that you will learn how to write in this chapter:

- Interactive video programs
- Quiz programs
- Programs that write and “understand” language
- Programs that play games
- Programs that learn

Logo’s facility with words and lists makes it ideal for writing conversational programs, quizzes, pig Latin translators, programs that teach, and even programs that learn: in short, all programs that need to manipulate lists of information.

The chapter is divided into three sections. The first is devoted entirely to interactive video programs, but introduces some procedures and techniques used in the remainder of the chapter.

The second section is devoted to programs that manipulate language (quizzes, sentence generators, etc.) and programs that build and manipulate knowledge bases.

The third section is devoted to building and manipulating more complex knowledge bases, and includes programs that play games and that learn.

## ***Interactive Graphics: READCHARACTER (RC), TOPLEVEL, STOP***

Let's create a program to control the turtle with single key-presses at the keyboard. The initial design will provide only four turtle behaviors, FD, RT, LT, and DRAW, and will control them with F, R, L, and D, respectively.

---

Projects at the end of this section suggest some additional behaviors to control. Further additions will become possible with techniques that you will learn later in this chapter.

The procedures that you will be developing are similar to those in the INSTANT program on your Utilities disk. This program is explained in this guide and in LOGO FOR THE APPLE II, by Harold Abelson.

---

In this design, the turtle will be moved Forward 10 steps each time the F is pressed. Each time R or L is pressed, the turtle will turn Right or Left 15 degrees. (You may choose any amount, of course, not just what is suggested here.) Pressing D executes DRAW.

The first task is to create a procedure that takes a single character as input and controls the turtle on the basis of that character. Its title line might be:

**TO EASYDRAW :CHARACTER**

or to save typing

**TO EASY :CHTR**

The logic is quite simple. If the character is an F, then perform FD 10. In Logo, this is:

**IF :CHTR = "F FD 10**



---

If you prefer, you can add the word THEN, and write

**IF :CHTR = "F THEN FD 10**

Some people find it easier to read a program that has the extra word in it. Others find it more cluttered that way. We will leave it out in this chapter.

---

Similarly, if the character is an R, perform RT 15.

**IF :CHTR = "R RT 15**

There should be some way of telling the program when we want to quit drawing to do something else. The letter Q (for Quit) can be used. If that character is the input, the procedure will perform NODRAW and TOPLEVEL.

NODRAW gets out of draw mode and clears the text screen. TOPLEVEL is the Logo primitive that tells Logo to stop executing a program and return to immediate mode to wait for a new command.



It is important to know the difference between `TOPLEVEL` and `STOP`. `STOP` stops the execution of the procedure in which it is found, but does not stop other procedures that may also be running. `TOPLEVEL` stops an entire program. Every procedure that Logo was running stops, and Logo returns control to the user.

---

The whole procedure might look like this:

```
TO EASY :CHTR
  IF :CHTR = "F FD 10
  IF :CHTR = "R RT 15
  IF :CHTR = "L LT 15
  IF :CHTR = "D DRAW
  IF :CHTR = "Q NODRAW TOPLEVEL
END
```

Define this procedure. Type carefully, making certain that no spaces are left between the `:` and the word `CHTR`, or between the double-quote character and the single letter that follows it. Notice also that there is only one double-quote character on each line.



We will explain in greater detail later, but provide this brief version for the curious. The words

```
"F          in IF :CHTR = "F FD 10 and
"CHAPTERW in SAVE "CHAPTERW
```

are quoted in order to tell Logo not to treat them as procedures. The words `FD` and `SAVE` are executed by Logo, but we want “F to be just plain F, literally, and not have Logo try to execute it as an instruction. Similarly, we want “CHAPTERW to be the name of a file — just a name, not something to do. The quoted word ends at the next blank space, so no final quote is needed.

Do not add a final quote, since Logo will then assume you mean to say something like: If the character is an F followed by a double-quote-mark, then . . . This is not at all what you want. To demonstrate this, type

PR “A”

---

After the procedure is defined — remember to type `<CTRL>C` — you can try it out by typing

```
EASY “F  
EASY “R  
EASY “Q
```

This is definitely not an improvement over typing `FD 10 RT 15 ND`, but it contains all the logic for the program we intended to create. Now what is needed is another procedure — let’s call it `QUICKDRAW` — whose sole purpose is to wait for a key to be pressed at the keyboard and to give that character to `EASY` as an input.

QUICKDRAW will use the Logo primitive READCHARACTER, abbreviated RC, to report what key has been pressed at the keyboard. To make QUICKDRAW continue endlessly (until a Q is pressed), QUICKDRAW calls itself as a subprocedure and looks like this:

```
TO QUICKDRAW
  EASY RC
  QUICKDRAW
END
```

The line EASY RC in QUICKDRAW tells Logo to read a character typed by the person using the computer and to use that character as the input to EASY. EASY figures out what action to perform based on what character it receives. If it gets an R, it performs a RT 15.

Even though all five lines of EASY are executed each time EASY is called, at most one action will be taken, because only one of the IF tests will be true.

### ***Projects with RC: Extending QUICKDRAW***

1. By using the same logic you can add other commands to EASY. Teach the procedure how to control the pen (PU or PD) in a single keystroke. (You might assign U to the command PU and P to the command PD, or you might choose D for PD, in which case you would need something else for DRAW.)
2. Add SHOWTURTLE (ST) and HIDETURTLE (HT).

3. Teach EASY to change the pen color with two keystrokes. The first keystroke (C, for Color) will run a procedure that waits for a second keystroke. If that second keystroke is a 0 through 6, the pen will be set to that color. If any other key is pressed, nothing happens.

The job could, of course, be done with one keystroke, representing each pen color with a different key. You might use the number keys directly, or use letters that represent the color names (for example, W for White, G for Green, etc.).

A disadvantage of using the numbers is that it would be nice to have them available for use as “multipliers” to multiply the effect of the next command. You will learn a technique for doing this in the next section. Choosing letters for each color is acceptable, although it requires that a person remember codes for each color.

4. Use the same technique to change background color.

### ***Changing the Value of a Variable: MAKE, PRINT (PR)***

We must take a short detour from the QUICKDRAW program. When you return to it, you will be able to write procedures which allow multiples of the single key commands in EASY. For example, 3F will make the turtle go forward  $3 * 10$  or 30 turtle steps.

The Logo primitive MAKE is used in several ways. In this section, we will illustrate one way, and in another section of this chapter, when we define words, lists, variables, input, and output more carefully, you will learn more of the subtleties of MAKE.

A metaphor for MAKE: When you say

MAKE "NUM 7 or  
MAKE "PERSON [MARGARET TRUMAN]



it is as though you are creating locations or boxes called NUM and PERSON and tossing a 7 into the first and the list [MARGARET TRUMAN] into the second. To find out what is in a particular box called NUM, the Logo command is THING "NUM or, more commonly, just :NUM, meaning the thing or value that is in the box named NUM.

Of course, you have been using names to refer to values all along. We will use the new metaphor to translate a procedure in a new way.

```
TO SHAPE :LENGTH :SIDES  
  REPEAT :SIDES [FD :LENGTH RT 360/:SIDES]  
END
```

This procedure tells the turtle how to draw a SHAPE whose features will be found in boxes that the procedure refers to as LENGTH and SIDES. The procedure's first instruction is to look in its SIDES box for a number, and REPEAT the following list of commands that number of times — go FORWARD the dimension found in its LENGTH box, and turn RIGHT however many degrees is equal to 360 divided by the number it found in the box named SIDES.

At the moment that you type

SHAPE 73 4 or SHAPE 15 6

Logo puts the 73 or 15 in a location (think of it as a box) that the SHAPE procedure refers to as LENGTH and puts the 4 or 6 into another location that SHAPE refers to as SIDES.



It is important to remember that LENGTH and SIDES are names that SHAPE uses to keep track of these numbers, and that no other procedure knows what SHAPE keeps in the boxes or even that the boxes exist! Further, those boxes cease to exist after SHAPE finishes its work.

---

Please note, however, that if SHAPE had called any subprocedures during its execution, those subprocedures would also have had access to the values in SHAPE's boxes.

---

Before getting back to MAKE, define SHAPE as shown above and then type

**SHAPE 50 5**

While SHAPE is operating, it executes the command `FD :LENGTH`, telling FD to move the turtle forward 50 turtle steps, the number of steps in the box LENGTH. If the 50 is still left in the box after SHAPE has finished drawing its pentagon, you should still be able to use it.

Try typing `FD :LENGTH` to see what Logo will do. Your screen should look like this:

```
FD :LENGTH
THERE IS NO NAME LENGTH
```

Now back to MAKE. MAKE can assign a value to a box or change the value that is in the box, and it can do it equally well in or out of a procedure.

Type MAKE "LENGTH 10 to create a box named LENGTH and place a 10 in it. Type DRAW to clear the screen, and then type FD :LENGTH. The turtle will move forward 10 turtle steps. Type

```
RT 144  
FD :LENGTH
```

This box did not disappear. It still exists and still has a 10 in it. Type

```
PRINT :LENGTH
```

Logo should print 10.

---

This kind of variable, defined outside of a procedure, is called a Global variable. See the explanation of global and local variables in the chapter titled Computation.

---

Since there is already a box called LENGTH with a 10 in it, you might think that you could now type just SHAPE 4 to get a four-sided shape with a size of 10.

If you try that, Logo will complain that SHAPE needs more inputs. Because SHAPE was defined to take two inputs, it must always be given two inputs.

Type

```
SHAPE 50 4
```

When it executed `FD :LENGTH`, how far did the turtle move? Not 10, but 50. And now that the square is drawn, type

`FD :LENGTH`

How far did the turtle move this time? Not 50, but 10. Type `PRINT :LENGTH` to Logo. Again Logo should print 10.

A summary of what happened: You told Logo to `MAKE "LENGTH 10`. Both before and after running `SHAPE` (with its own variable of the same name set to 50), you were able to show that `LENGTH` really did have the value 10. Whether you typed `PRINT :LENGTH` or `FD :LENGTH`, `LENGTH` represented 10.

However, `SHAPE`, even though it had a variable of the same name, did not seem to know about the 10 and did not change it to 50, even though that is what `SHAPE` considered `LENGTH` to be.

Until you have had a chance to write enough procedures and have had more experience with variables and values, they tend to remain confusing, but remembering one principle may help.



When a procedure has variables in its title line, the values of those variables inside the procedure depend entirely on the values given to the procedure as inputs. This is true regardless of the existence or values of variables with the same names that may be found elsewhere in a program.

One more experiment with variables and MAKE before returning to QUICKDRAW. Type

```
PRINT :NUM
```

It should reply:

```
THERE IS NO NAME NUM
```

(If it prints something different from that, type

```
ERNAME "NUM
```

and start again!)

Now type

```
MAKE "NUM 5 and on the next line type  
PR :NUM
```

(PR is the abbreviation for PRINT.) Now it should reply by printing a 5.

Define these two very similar procedures:

```
TO FOO  
  PR :NUM  
  MAKE "NUM 2 * :NUM  
  PR :NUM  
END
```

```
TO FOOL :NUM  
  PR :NUM  
  MAKE "NUM 2 * :NUM  
  PR :NUM  
END
```

After you have defined them and before you run them, type `PR :NUM` again. Logo will still reply 5.

Now, in order and one at a time, type the following commands to Logo. We will explain the mystery afterward.

```
FOO
PR :NUM
FOOL 4
PR :NUM
FOO
PR :NUM
FOOL 3
PR :NUM
```

What's happening?! `FOO` and `FOOL` have absolutely identical insides, and yet their behavior is so very different. You printed the value that is inside the box named `NUM` before and after running each procedure.

`FOO` knew about what was in that box and also changed it, but `FOOL` did neither. Before and after `FOOL`, the value in `NUM` remained the same — even though it appears to have two completely different values inside `FOOL`.

The explanation is in the title line. As mentioned earlier, when a procedure's title line contains a variable name in it, that name refers to a totally private box created just for that procedure.

So `FOO` could use the value of `NUM` that was lying around at the time, and could also change it. But `FOOL`

had access only to its private box, which happened to have the same name, but is altogether a different box.

Whenever the name NUM was used inside FOOL, FOOL took it to mean its own box of that name. It was not the public box named NUM that FOOL printed and changed, but only FOOL's NUM. As soon as FOOL stopped running, it took its NUM with it.

When you then typed PR :NUM again, you had no access to FOOL's private box; instead, you referred to everyone's public box named NUM. The private variable is called a local variable, and the public one is a global variable.

---

Logo version 2.0 includes the command LOCAL, which allows you to create local variables without declaring them in the title line. An example:

```
TO DEMO.LOCAL
  LOCAL "VALUE
  MAKE "VALUE RC
  PR :VALUE
END
```

Consult page C-7 for a full discussion of LOCAL.

---



**Admonition:** Unless you really intend to make a variable public and available for everybody to use and change, don't make global variables. They are trouble-makers (in large programs) precisely because anybody is free to fool around with them.

On the other hand, the great virtue of global variables is that they survive even after a procedure is finished. When you need to have a value remembered even after the procedure that created it is finished working, use a global variable.

Otherwise avoid global variables. It is almost never good style to use MAKE when passing a variable to a procedure in the title line can be done easily.

### ***Projects with MAKE: More Extensions to QUICKDRAW***

5. Teach EASY to recognize digits and use them to multiply the effect of the very next keypress. For example, the effect of typing 3F, should be either FD 30 or REPEAT 3 [FD 10]. You decide which.

If the character 3 is typed to RC, RC's output, which EASY knows as CHTR, can be used both in tests such as IF :CHTR = 3 and in numerical expressions such as :CHTR + 5. You may also find the Logo primitive NUMBER? useful. The test NUMBER? :CHTR is true for all characters 0 through 9.

Project 5 is a reasonable use of MAKE because it requires remembering a number from one execution of EASY to the next. A command like MAKE "MULTIPLE :CHTR will put the current value of CHTR into a box named MULTIPLE.

The contents of the CHTR box will be forgotten when EASY stops, but since MULTIPLE will be a global variable, the value in that box will not be forgotten and can be used until it is changed.

6. Type MAKE "PENPOS [DOWN] and then define and experiment with the following procedure.

```
TO PEN
  IF :PENPOS = [DOWN] PU MAKE "PENPOS [UP]
    ELSE PD MAKE "PENPOS [DOWN]
  PRINT SENTENCE [THE TURTLE'S PEN IS NOW] :PENPOS
END
```

The procedure contains at least one primitive (SENTENCE) that you have not seen before, and an interesting use of a global variable. When you understand how this procedure works, include it in EASY.

7. Write a similar procedure for ST and HT.

### ***Programs that Interact without Waiting: RC?***

Until some key has been pressed, RC cannot output a message saying which key. That is why QUICKDRAW always waits until a character is typed. Every time it runs RC, RC waits until it has something to report back to EASY.

Sometimes, though, you want the program to keep going while waiting for the user to type something. For example, in video-action-games, objects are supposed to keep moving on the screen whether or not the player touches the keys or twiddles the knobs.

Let's design a program in which we drive the turtle like a car. The turtle will always be moving, but we can increase or decrease its speed and can change its direction. In order to have it moving constantly, we will need a loop something like this:

```
TO LOOP
  FD :DIST
  LOOP
END
```

Make DIST have some small initial value, like 1 or 2, by typing MAKE "DIST 2. Then run LOOP. The turtle will slowly crawl across the screen.

To be more flexible, LOOP should check to see if the person has typed anything, and, if so, should take some action before moving the turtle again. RC? is the primitive that checks to see if a character has been typed.

The logic is this: If the person has typed a character,

```
IF RC?
```

then read the character, and control the turtle accordingly:

```
EASY RC
```

So the completed LOOP would look like this:

```
TO LOOP
  IF RC? EASY RC
  FD :DIST
  LOOP
END
```

Define LOOP and experiment with it using your EASY just as it is. How does LOOP behave differently from QUICKDRAW?

As it is written, EASY does not give sensitive control over the speed of the turtle. Pressing F does give a burst of distance, but the turtle settles back to the same slow crawl immediately afterward.

Look at LOOP to see what determines the turtle's speed. Now study EASY to see why it does not alter that speed. Even though EASY is not quite what is needed for this program, still it provides a number of features that are just as appropriate for LOOP as they are for QUICKDRAW.



So that you can make changes to an EASY-like procedure without changing EASY itself (which is just fine for QUICKDRAW), make a copy of EASY using a different title. To do this, edit EASY and change the title in the editor to ACTION. Then, when you define the procedure, it will be named ACTION.

EASY is still around, as before, but a new copy titled ACTION now exists also. If you have been doing the projects, your copy of EASY (and, therefore, ACTION) will no longer look like the original. But if it did, it would look like this:

```
TO ACTION :CHTR
  IF :CHTR = "F FD 10
  IF :CHTR = "R RT 15
  IF :CHTR = "L LT 15
  IF :CHTR = "D DRAW
  IF :CHTR = "Q NODRAW TOPLEVEL
END
```

Do you see that although ACTION controls the turtle's movement, it does not change :DIST, and therefore does not affect the turtle's speed?

Instead of having F move the turtle directly, it could increase the distance that the turtle moves each time through LOOP. The logic might be like this — If the character typed is F

```
IF :CHTR = "F
```

make the distance to travel 2 greater than it was the last time.

```
MAKE "DIST :DIST + 2
```

If F stood for Faster, S could stand for Slower and decrease DIST.

A working version of ACTION might look like this:

```
TO ACTION :CHTR
  IF :CHTR = "R RT 15
  IF :CHTR = "L LT 15
  IF :CHTR = "F MAKE "DIST :DIST + 2 ; FASTER
  IF :CHTR = "S MAKE "DIST :DIST - 2 ; SLOWER
  IF :CHTR = "D DRAW
END
```

When you press the F key, the distance that the turtle will move during each loop increases by 2 steps. The S key decreases the number of steps per loop.

Define ACTION in one of the ways suggested above, and write a START procedure like this one:

```
TO START
  MAKE "DIST 0
  LOOP
END
```

Remember to edit LOOP so that it uses ACTION in place of EASY.

Now type START and experiment with controlling the turtle. With practice, you can learn to control it well enough to draw even complicated figures.

*Projects with RC, RC?: Extensions to LOOP*

8. By changing LOOP so that both the turtle's position and heading are updated each time through the loop, the turtle can then draw curves. Here is how LOOP would look:

```
TO LOOP
  IF RC? ACTION RC
  FD :DIST
  RT :ANG
  LOOP
END
```

Design and make some changes to ACTION and START to take advantage of the new capabilities of LOOP.

9. Add a feature to stop the turtle. Experiment also with three new commands, one of which does MAKE "DIST (- :DIST), another of which does the same for ANG, and the third of which makes both DIST and ANG negative. Try to gain enough skill at controlling the turtle to get it to write your name in cursive script.

## ***INTERACTIVE LANGUAGE***

### ***Don't Skip This Section!*** ***MEMBER?, EMPTY?***

Right now, please define two procedures, `MEMBER?` and `EMPTY?`, that will be used throughout the remainder of the chapter. (If you have Terrapin Logo Version 2.0 or beyond, these are already provided as primitives, so you need not define them yourself and may skip to the next section.)

Type carefully. Make certain that you leave no space between `:` and the word that follows it, and that you do not leave out the question marks in the procedure titles.

```
TO MEMBER? :ELEMENT :OBJECT
  IF EMPTY? :OBJECT OUTPUT "FALSE
  IF :ELEMENT = FIRST :OBJECT OUTPUT "TRUE
  OUTPUT MEMBER? :ELEMENT BUTFIRST :OBJECT
END
```

```
TO EMPTY? :OBJECT
  OUTPUT ANYOF :OBJECT = [] :OBJECT = "
END
```

It is worth the effort to save these procedures in their own separate file as well as with the work you are doing in this chapter. That will allow you to read them into your workspace whenever you need them without reading in everything else you have ever worked on.

For now, these procedures will be explained only as if they were primitives; we will show how they are to be used, but not how they work. Later in the chapter, both will be explained in greater detail.

`MEMBER?` takes two inputs — a word and a list — and outputs the value “TRUE if the word is an element of the list. (You can also give `MEMBER?` a character and a word, and it will return “TRUE if the character is part of the word.)

`EMPTY?` takes one input and outputs “TRUE if the input is the empty list or the empty word. We will explain this in greater detail later on.

To see what `MEMBER?` does, try the following commands.

```
MEMBER? "DOG [THE DOG BARKED]
MEMBER? "CAT [THE DOG BARKED]
MEMBER? "U "AEIOU
MEMBER? "G "AEIOU
MEMBER? "4 "1234XYZ
```

Your screen should look like this:

```
MEMBER? "DOG [THE DOG BARKED]
RESULT: TRUE
MEMBER? "CAT [THE DOG BARKED]
RESULT: FALSE
MEMBER? "U "AEIOU
RESULT: TRUE
MEMBER? "G "AEIOU
RESULT: FALSE
MEMBER? "4 "1234XYZ
RESULT: TRUE
```

If these are not the results you get, check the procedures carefully, character by character, to make certain that they are exactly as shown above. After you have checked, save the procedures.

***Some Friendly Introductions: SENTENCE (SE), REQUEST (RQ), LPUT, FPUT***

If you did project 6 above, you have already seen the Logo primitive SENTENCE used to combine two pieces of text into a single sentence. In the procedure in project 6 the line read

```
PRINT SENTENCE [THE TURTLE'S PEN IS NOW] :PENPOS
```

When PENPOS was [DOWN], the effect of that line was to print

*THE TURTLE'S PEN IS NOW DOWN*

When PENPOS was [UP], the effect of that line was to print

*THE TURTLE'S PEN IS NOW UP*

Define the procedure GREET. You may wish to spell out PRINT and SENTENCE fully or to abbreviate them. Both forms of the procedure are shown.

Fully spelled out:

```
TO GREET :PERSON
  PRINT SENTENCE [NICE TO MEET YOU,] :PERSON
END
```

or abbreviated:

```
TO GREET :PERSON
  PR SE [NICE TO MEET YOU,] :PERSON
END
```

Run the procedure GREET, giving it a person's name as input. For example:

```
GREET [GEORGE]
GREET [GEORGE WASHINGTON]
```

GREET has a simple behavior. Whatever its input, GREET prints a sentence composed of the words NICE TO MEET YOU (with a comma at the end) and that input.

Now we will create a procedure which uses GREET in a brief friendly conversation. The behavior of the new procedure will be a bit more complex. It will start up with no information at all (no input), and will ask the person to type his or her name. Then it will use GREET to greet the person. To do this, it must give GREET an input consisting of whatever the person typed.

Let's write the procedure as we review its behavior. It needs no inputs, so its title line could be TO FRIENDLY. It asks the person it meets to type a name: PR [WHAT'S YOUR NAME?]. It then gives GREET an input consisting of whatever the person types: GREET REQUEST.



REQUEST (abbreviated RQ) is a Logo primitive that tells a procedure 1) to wait for a person to type a line and press <RETURN> and 2) to output that line as a list that can be used by a procedure.

Here, REQUEST's output is used as GREET's input.  
Define FRIENDLY.

```
TO FRIENDLY
  PR [WHAT'S YOUR NAME?]
  GREET REQUEST
END
```

To run it, type FRIENDLY (remember, no input!) and press <RETURN>. When it asks, type your name (and press <RETURN>). You do not need to type brackets or other special decorations; just your name will do. Run it again, but this time, when it asks for your name, press <RETURN> without typing anything at all. Your screen will now look something like this:

```
FRIENDLY
WHAT'S YOUR NAME?
HANNIBAL THE TURTLE
NICE TO MEET YOU, HANNIBAL THE TURTLE
FRIENDLY
WHAT'S YOUR NAME?
```

*NICE TO MEET YOU,*

REQUEST can return an empty list, indicating that the person typed nothing, but GREET is not smart enough to know what to do about that. It would be nicer if GREET could recognize an empty input and respond differently.

Here's a version of GREET that does that. We will use the command EMPTY? to check for bashful typists.

```
TO GREET :PERSON
  IF EMPTY? :PERSON PR [OH! YOU MUST BE QUITE SHY.] STOP
  PR SE [NICE TO MEET YOU,] :PERSON
END
```

Edit GREET to insert the new line and try it again, as you did before.

```
FRIENDLY
WHAT'S YOUR NAME?
HANNIBAL THE TURTLE
NICE TO MEET YOU, HANNIBAL THE TURTLE
FRIENDLY
WHAT'S YOUR NAME?
```

*OH! YOU MUST BE QUITE SHY.*

This time GREET is better about handling the blank response, but it apparently has a terrible memory! After all, it has already met HANNIBAL THE TURTLE, and should have said something more like GOOD TO SEE YOU AGAIN rather than NICE TO MEET YOU.



Helping the computer remember names brings in a whole new idea. For GREET to remember, it must be a learning program. It must keep a list of the people it has already met, and, when it gets a person's name, it must be able to check to see whether that name is on its list. If the person is a member of the list of known people

```
IF MEMBER? :PERSON :KNOWN
```

then GREET should print some appropriate response and then stop.

**PR SE [GOOD TO SEE YOU AGAIN] :PERSON STOP**

If the person is not a member of that list, then GREET should say what it did before. It should also stick the person's name at the end of the list of known people. That is accomplished by taking the list out of the box named KNOWN, tacking the person's name at the end of it, and placing the result back in the box.

**MAKE "KNOWN LPUT :PERSON :KNOWN**

LPUT takes two inputs, an object (in this case PERSON) and a list (in this case KNOWN), and puts the object in the list as the last element. LPUT abbreviates LastPUT, but there is no fully spelled out name of the primitive. (Its companion FPUT, for FirstPUT, will put in an appearance later on.)

Here is GREET as it is now designed.

```

TO GREET :PERSON
  IF EMPTY? :PERSON PR [OH! YOU MUST BE QUITE SHY.] STOP
  IF MEMBER? :PERSON :KNOWN PR SE [GOOD TO SEE
    YOU AGAIN] :PERSON STOP
  PR SE [NICE TO MEET YOU,] :PERSON
  MAKE "KNOWN LPUT :PERSON :KNOWN
END

```

Edit it to include the new changes, and when it is defined, type

**FRIENDLY**

What happens? Ah! Logo complains that there is no list of known people.

Before GREET has met any people, the list may have no names in it, but it must still exist in order to be checked. That is why Logo said

*THERE IS NO NAME KNOWN*

This problem is solved by typing

MAKE "KNOWN []

Type

PRINT :KNOWN

and notice that just an empty line is printed. Now type

FRIENDLY

again. After it finishes greeting you, type

PRINT :KNOWN

again and note what you see. Play with it for a while, perhaps by typing

REPEAT 10 [FRIENDLY]

Introduce new people and reintroduce old people.

Type

PRINT :KNOWN

to see what its memory contains. (If the program is not being friendly, check for errors.)

Finally, some fine points. When the person has been too shy to type a name, let GREET be a bit pushier. Instead of just stopping, it can ask again. How? By running FRIENDLY before stopping. The line might look like

```
IF EMPTY? :PERSON PRINT [DON'T BE SHY. PLEASE TELL  
ME.] FRIENDLY STOP
```

Edit GREET again, making this last change, and experiment with it. Notice that even after you have edited GREET it remembers the people it had met earlier. Any time you want to, you can make it forget everybody by typing

```
MAKE "KNOWN []
```

to empty out its list. You can also type

```
EDIT NAMES
```

and change the contents of the name KNOWN at will. When you do that, make sure when you are finished that all of the left and right brackets match up properly!

There are two more features that would make GREET seem really like an intelligent program. Try typing I DON'T WANT TO TELL YOU, or NONE OF YOUR BUSINESS, or even MY NAME IS PAUL when FRIENDLY asks your name. GREET should certainly not say NICE TO MEET YOU, NONE OF YOUR BUSINESS.

It would be nice if GREET could be given enough knowledge of English to recognize at least these cases and respond properly. Also, it would be nice if both GREET and FRIENDLY had a bit more variety in what they said. You will be able to make both of these improvements by the end of this chapter.

First you must learn some new primitives and programming techniques.

### ***Interlude: Clearing the Text Screen with CLEARTEXT***

Type CLEARTEXT to Logo. While working on this chapter you will often need to clear the text screen. CLEARTEXT has no abbreviation, so you might want to define an abbreviation.

```
TO CT  
  CLEARTEXT  
END
```

After defining CT, and without typing any graphics commands, mess up the text screen some. Typing the following lines should create plenty of mess:

```
ABC  
+  
:
```

Messy enough? Now type CT. Ah, if only all cleaning up were that easy.

## ***Objects: Producing RESULTS as Output, and Using Them as Input***

The best way to come to understand Logo objects well is to use them in a variety of contexts. A formal definition will come later, but some experiments are needed now.

Type

```
5 <RETURN>
[APPLES AND ORANGES] <RETURN>
"BEEP <RETURN>
```

(Don't forget the double-quote at the beginning of "BEEP.)

In each case Logo responds **RESULT:** followed by the object you typed.

```
5
RESULT: 5
[APPLES AND ORANGES]
RESULT: [APPLES AND ORANGES]
"BEEP
RESULT: BEEP
```

---

In two of the cases Logo typed exactly what you typed. But in the third case, Logo typed the word **BEEP** without a double-quote mark.

Here is the explanation. The object you typed was the word **BEEP**. The double-quote mark was merely to tell Logo that you were typing an object and not the name of a procedure.

Typing “FRIENDLY (with the quote-mark) will have the same effect. Typing FRIENDLY (without the quote-mark) will run the procedure that you wrote in the last section.

The double-quote is not part of the object; it is just a marker. Neither numbers nor lists can be procedure titles, so Logo does not need any special markers to help it recognize those as objects.

Also, words that are already inside lists, like APPLES or AND, need no special markers. Logo will not try to run them unless you explicitly tell it to.

---

If an object is “given to” Logo in immediate mode, Logo announces it with the word RESULT:. If an object is “given to” PRINT as an input, PRINT prints the object on the screen.

PRINT, too, changes the appearance of what you type slightly. Using the same three examples, PRINT 5, PRINT [APPLES AND ORANGES], and PRINT “BEEP”, both of the last two are printed without their punctuation.

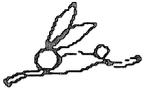
PRINT doesn’t show the marker or the outer brackets that surround a list, but merely the object and the list elements themselves.



Even though we have been playing with a number (5), a list ([APPLES AND ORANGES]), and a word (BEEP) — all abstractions — we think of these very concretely,

as if they were solid objects that can be tossed back and forth among players in a game.

This metaphor is very useful in Logo programming. Procedures are the players. You make up the rules of the game, deciding what the behavior of each procedure will be, what object (if any) a procedure should create, and who should receive the object after it is made.



There are ways of giving objects to Logo in immediate command mode other than by placing them there yourself. You can let a procedure create them and place them there.

At the beginning of the section, you typed `MEMBER? "G "AEIOU` and Logo announced that the object `FALSE` was given to it as a result. Here are some other ways of getting procedures to hand objects to Logo.

```
RANDOM 100  
FIRST :KNOWN
```

The primitive `RANDOM` outputs a random number from 0 up to (but not including) its input. `FIRST` outputs the first element of the object that is its input. (In this case, the object is a list from the box named `KNOWN` that you created earlier in the chapter.)

In the next two lines are two other primitives that output objects. It may be harder to recognize the primitives this time, because you are probably not used to thinking of them as primitives.

```
5 + 6  
:KNOWN
```

The first primitive is the `+`. It takes two inputs, one on each side of it, and outputs their sum if they are numbers.

The second primitive is the `:` (which is a special kind of abbreviation for `THING`). It takes one input, attached to it on the right, and outputs the object that is found in the box of that name. (The box, `KNOWN`, was created earlier as part of the `FRIENDLY` program.)



There are only two ways of creating objects. You can put them there, yourself, or a procedure or primitive can create them.

Some primitives create objects as output and others don't. For example, if you type `FIRST 37`, Logo announces the object that `FIRST` outputs. (What is it?) But if you type `PRINT 37`, the object simply appears on the screen and cannot be used by other commands.

Some primitives require objects as input and others don't. If a primitive does need an object as input, it does not care whether that input is put there by you, or is the result of running another primitive.

So, in the command `PRINT FIRST :KNOWN`, the object that `PRINT` needs as its input is created by `FIRST` and supplied as its output.

### *Writing Procedures that Create and Output Objects: **OUTPUT***

Except for the two procedures `MEMBER?` and `EMPTY?` with which we began this section, you have never written a procedure that creates an object and outputs it for another procedure to work with. Such a

procedure is vastly more powerful and flexible than anything we have discussed up to now.

To begin, let's define the procedures TEN and DOUBLE.

```
TO TEN
  OP 7 + 3
END
```

```
TO DOUBLE :NUMBER
  OP 2 * :NUMBER
END
```

Now, typing TEN to Logo gets the response RESULT: 10. TEN can be used in computations.

Type

```
PRINT DOUBLE TEN
```

The OUTPUT command (or its abbreviation, OP) tells these procedures to stop and “output an object” or “return a value” or “produce a result.” To see what all this means, try the following experiments by typing these lines to Logo:

```
10
TEN
DOUBLE 5
5 * DOUBLE 1
```

When you typed the number 10, you were handing Logo the object 10 directly. The object 10 has the value 10 or results in a 10 lying around. Logo announces that with the message, RESULT: 10.

When you typed the procedure name `TEN`, it computed a value and then it handed the value (the object 10) to Logo. Similarly, when you typed the procedure name `DOUBLE`, you handed it the object 5 to work with. It computed the value 10, and handed that back as the result.

In both of these cases, you may think of the procedures as having replaced themselves by the value they output. That makes the last line especially clear. If `DOUBLE 1` replaces itself with the value 2, then the line becomes `5 * 2`.



---

We often use the word “object” to refer equally to words (including such things as numbers and letters) and lists (including simple sentences or complex data structures).

We do this because Logo can easily combine words and lists to make other words and lists, break words and lists into pieces, or pass words and lists back and forth between procedures as if they were concrete, solid objects.

When we need to specify what kind of object, we refer to “numbers” or “words” or “lists,” but the word “object” refers to them all.

The word “value” sometimes sounds more natural than “object” when we are referring to the result of some computation, but there is really no important difference between the words.

---

Here is a more useful application of the same sorts of procedures.

```
TO PI                                TO CIRCUMF :DIAMETER
  OUTPUT 3.14159                      OP PI * :DIAMETER
END                                    END
```

Having a procedure that figures out the circumference of a circle, given its diameter, has a practical application in Logo. Among other jobs, it can be used in a graphics procedure to draw circles of a given size.

All circles in Logo are drawn by drawing short line segments, turning a little, and repeating the process until the circle closes. The smaller the line segments, of course, the smaller the circle.

But how do we determine the length of the segments if we want a circle of a very specific size? If the circle is composed of twenty segments, then each one is one-twentieth of the circumference. If it is drawn with twelve segments, then each is one-twelfth of the circumference.

Clearly, then, to draw a circle of a specific diameter, we must first know the circumference. Then we can divide it into equal parts, and repeatedly draw one of these parts and turn the appropriate amount.

```
TO CIRCLE :DIAMETER
  ARC 20 ( CIRCUMF :DIAMETER ) / 20
END
```

```
TO ARC :SEGMENTS :CHORD
  REPEAT :SEGMENTS [FD :CHORD RT 18]
END
```

---

Try

CIRCLE 40 RT 180 CIRCLE 40

The following definition of ARC gives a slightly more symmetrical placement of the circles on a vertical line. Figure out why.

```
TO ARC :SEGMENTS :CHORD
  FD :CHORD/2
  RT 18
  REPEAT :SEGMENTS - 1 [FD :CHORD RT 18]
  FD :CHORD/2
END
```

It is useful to have a definition of CIRCLE that curves to the left as well as this one that turns to the right. You can also define half- and quarter-circles using the same ARC procedure.

---

The objects these procedures manipulated were all words -- in fact, only numbers. Now back to lists. Define these two procedures.

```
TO PEOPLE
  OUTPUT [SANDY CHRIS [THE TURTLE] DANA LEE
          PAT DALE]
END
```

```
TO ACTIONS
  OUTPUT [LOVES [DREAMS ABOUT] KISSED
          HATES [CAN'T STAND] LIKES]
END
```

We have chosen the names PEOPLE and ACTIONS as good descriptions of the nature of the procedures. You will be using these procedures often, so you might like to choose names that are shorter or easier to type, like PPL and ACTS, or NOUNS and VERBS, or just N and V.

As you develop more complex programs, it will become especially important that you choose procedure titles and variable names that help you remember what their purpose is. The best policy is to choose names that are easy in two ways: easy to type and easy to remember.

---



These procedures contain instruction lines that are too long to fit neatly on the screen, but you should continue typing normally, without pressing <RETURN> when you get to the edge of the screen. Logo will place a ! at the end of the screen to indicate that your line continues past there, but will continue to show the rest of your typing on the next line.

Type these procedures accurately, being particularly careful about getting the left and right brackets in the correct places. (Notice that they are the square brackets, and not parentheses! The brackets are typed as SHIFT-N and SHIFT-M on the Apple II and Apple II+.)

---



Once you are in the editor, you can type any number of procedures before pressing CTRL-C to define them. (But remember you must type END after each procedure before

starting the next one.) In this case it makes little difference whether you define the procedures one by one or both together. Sometimes, though, you will find it very convenient to be able to look at one procedure while you are defining another.

---

The only behavior of these procedures is to output a list. `PEOPLE` outputs a list of seven names. Six of those names are words, but one of them, `THE TURTLE`, is itself a list of two words.

`ACTIONS` also outputs a list. That list contains only six elements, four of which are single words and two of which are lists of two words each.

To demonstrate that these procedures output objects, type `PEOPLE` to Logo. Then type `PRINT ACTIONS`. Your screen should look something like this.

```
PEOPLE
RESULT: [SANDY CHRIS [THE TURTLE] DANA L
EE PAT DALE]
PRINT ACTIONS
LOVES [DREAMS ABOUT] KISSED HATES [CAN'T
STAND] LIKES
```

When Logo cannot fit everything onto one line, it breaks the line where it must, and continues on the next line. (Note that a ! does not appear at the end of the first line in immediate mode, unlike in edit mode.)

---

*Making One Procedure's Output into Another Procedure's Input: OUTPUT (OP), FIRST, BUTFIRST (BF), LAST, BUTLAST (BL), SENTENCE (SE), WORD*

Clear the text screen.

What object does FIRST PEOPLE output? (Type FIRST PEOPLE to check if you want to.)

Logo also has a procedure BUTFIRST which outputs all but the first element of its input. Type BUTFIRST PEOPLE to see the object it outputs. And what is the FIRST of that object? Type FIRST BUTFIRST PEOPLE to see.

What object would BUTFIRST output if its input is the object created by BUTFIRST PEOPLE. (In other words, what object is created by BF PEOPLE, and what is the BF of that?) Type BF BF PEOPLE or BUTFIRST BUTFIRST PEOPLE to check.

And what is the FIRST of that object? Type FIRST BF BF PEOPLE to see.



Remember to clear the text screen whenever it will help you see what you are doing.

Here are some more experiments to do with PEOPLE and ACTIONS. They are all to get you familiar with some new primitives and passing objects between them. You may type abbreviated forms such as PR, SE, and BF, or fully spelled out forms, whichever you prefer, but don't just sight-read these experiments. Do each of them and compare the results you get to the comments written before or after the experiments.

Logo can copy an object from either end of a list,

**FIRST ACTIONS**  
**LAST ACTIONS**

and from either end of a word

**PR FIRST "CAT**  
**PR LAST "CAT**

---

When **FIRST** or **LAST** receive a word as input, they output the corresponding (first or last) letter of the word. When they receive a list as input, they output the corresponding element of the list.



**BUTFIRST (BF)** and **BUTLAST (BL)** output all but what **FIRST** and **LAST** output. It is important to remember (and a common source of bugs for those who forget) that the **BF** or **BL** of a list is always a list. Thus, the **BUTFIRST** of **[FD 30]** is not 30, but **[30]**.

---

**FIRST** of **ACTIONS** produced an object, a result. Logo can manipulate that object, taking its **FIRST** or **LAST** element, just as it can manipulate any other object.

**PR FIRST FIRST ACTIONS**  
**PR LAST FIRST ACTIONS**

Since **FIRST ACTIONS** is **LOVES**, its **FIRST** is **L** and its **LAST** is **S**.

Type:

**PR SENTENCE PEOPLE ACTIONS**  
**PR SE FIRST PEOPLE FIRST ACTIONS**

**SENTENCE** (abbreviated **SE**) glues any two objects together into a sentence. The sentence of the lists output by **PEOPLE** and **ACTIONS** is a long one. The sentence of the first elements of those lists is **SANDY LOVES**.

Type:

**PR (SE LAST PEOPLE LAST ACTIONS FIRST PEOPLE)**

By surrounding **SENTENCE** and its inputs with parentheses, you can force **SENTENCE** to take more (or fewer) than two inputs. This is often very important in interactive language programs.

The next series of experiments is particularly important as it forms the basis for the vast majority of the procedures you will use most in manipulating words and lists. It is, for example, at the heart of the **MEMBER?** procedure that you defined at the very beginning of this chapter. Clear the text screen. Do each experiment and note its behavior.

**PEOPLE**  
**BF PEOPLE or BUTFIRST PEOPLE**  
**BF BF PEOPLE**  
**BF BF BF PEOPLE**

Be certain you see the pattern in the results of the previous four experiments before going on. Now predict the results of each of these experiments and then check your prediction by running the experiment.

FIRST PEOPLE  
FIRST BF PEOPLE  
FIRST BF BF PEOPLE  
FIRST BF BF BF PEOPLE

Similar patterns hold for LAST and BUTLAST (BL).

PR BUTLAST ACTIONS or PR BL ACTIONS  
PR LAST BL ACTIONS

Finally, you can combine a whole bunch of these operations into one command.

PR (SE FIRST BF BF PEOPLE LAST BL ACTIONS [ME])

SENTENCE glues parts together to make a sentence. We added [ME] to try to add some interest.



Logo also provides the primitive WORD, which glues parts together to make a word. Try this:

PR WORD "C BF "SANDY

Here are some more complicated expressions using WORD.

PR WORD BL FIRST PEOPLE "WICH  
PR WORD BL FIRST ACTIONS LAST BL PEOPLE

### *Subprocedures for Cleaner Programming*

The primitives that Logo provides give immediate access to the first or last element of a list, or to the first or last character of a word, but what about the second, third, or other elements?

One set of procedures to output the SECOND, THIRD, FOURTH, and FIFTH elements of an object is based on the experiments you tried above. Type these in, and try them out with the projects suggested.

```
TO SECOND :OBJ
  OP FIRST BF :OBJ
END
```

```
TO THIRD :OBJ
  OP FIRST BF BF :OBJ
END
```

```
TO FOURTH :OBJ
  OP FIRST BF BF BF :OBJ
END
```

```
TO FIFTH :OBJ
  OP FIRST BF BF BF BF :OBJ
END
```

```
PR (SE FOURTH PEOPLE THIRD ACTIONS THIRD PEOPLE)
PR (SE SECOND PEOPLE FIFTH ACTIONS THIRD PEOPLE)
```

The new procedures allow you to write equivalent commands in different ways. For example, the two following commands have the exactly the same effect:

```
PR (SE FOURTH PEOPLE SECOND ACTIONS FIFTH PEOPLE)
PR (SE FIRST BF BF BF PEOPLE FIRST BF ACTIONS FIRST BF
    BF BF BF PEOPLE)
```

. . . but there are important differences. Not only is the first shorter to type, but it is also much more understandable. Writing understandable programs is a mark of good programming.

## *A Generalization Using Recursion: ITEM*

Although these new procedures vastly simplify both the look and the typing of some list manipulations, they have some drawbacks. The most obvious is that in order to get PAT out of the PEOPLE list, we'd need a procedure SIXTH, and even if we wrote that, there would always be some list that was even longer.

What we really need is one single procedure that can retrieve any member of a list. (If you have version 2.0, you can use the primitive ITEM to do this. Don't skip ahead, though.)

As a first step toward figuring out how to write it, let us carefully describe its behavior in English. Let us call this procedure NTH (as in fourTH, sixTH, sevenTH). We need to tell NTH two things: what number element to find, and what object to choose it from. Perhaps the whole title line will look something like this:

```
TO NTH :N :OBJECT
```

If N is 1, the procedure should just output the first element of the object. That instruction would look like this in Logo.

```
IF :N = 1 OUTPUT FIRST :OBJECT
```

and the whole procedure, so far, would look like this:

```
TO NTH :N :OBJECT  
  IF :N = 1 OUTPUT FIRST :OBJECT  
END
```

Create this procedure. At this stage you can use the procedure to get the first (but only the first) element of an object. Try typing `PR NTH 1 PEOPLE`, and make sure it prints `SANDY`.

That is the simplest situation. To come up with a good way of describing the other situations, let us examine them one by one. If `N` is 2 then we want `NTH` to output the first element of the next shorter object (the `BUTFIRST` of the object). The first element of an object is something `NTH` knows how to output, so it can do the job itself. In Logo, that might be translated this way:

```
IF :N = 2 OUTPUT NTH 1 BF :OBJECT
```

It will turn out that there is a neater way of doing things, but, for now, add that line to your procedure, too, and check to see that `PR NTH 2 PEOPLE` causes Logo to print `CHRIS`. You might also check `PR NTH 1 ACTIONS` and `PR NTH 2 ACTIONS`.

What if `N` is 3? `NTH` already knows how to find the second element of an object. To find the third element, we could simply find the second element in the `BUTFIRST` of the object. In Logo, this is translated:

```
IF :N = 3 OUTPUT NTH 2 BF :OBJECT
```

If we continued in this way, we might add a bunch of instructions that look like this:

```
IF :N = 4 OUTPUT NTH 3 BF :OBJECT  
IF :N = 5 OUTPUT NTH 4 BF :OBJECT  
IF :N = 6 OUTPUT NTH 5 BF :OBJECT  
IF :N = 7 OUTPUT NTH 6 BF :OBJECT
```

But this does not solve the original problem. N might still be some number larger than we account for. Fortunately, there is a generalization we can make. In all of the cases where N is not 1, the procedure figures out what to do by looking for element N-1 in the BUTFIRST of the object.

We will repeat the logic:

TO output the NTH element of an object we need to know N and the OBJECT.

TO NTH :N :OBJECT

If N = 1, we want to OUTPUT the FIRST of the OBJECT.

IF :N = 1 OP FIRST :OBJECT

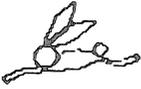
In every other case, we want to OUTPUT the N-1 element (found by using NTH with an input of N-1) of the BUTFIRST of the OBJECT.

OP NTH :N - 1 BF :OBJECT

Thus, the procedure might look like this (with OUTPUT abbreviated as OP):

```
TO NTH :N :OBJECT
  IF :N = 1 OP FIRST :OBJECT
  OP NTH :N - 1 BF :OBJECT
END
```

Edit NTH to make your copy look like this new version and try it out with values of N ranging from 1 to 7 and the PEOPLE list, or with values ranging from 1 to 6 and the ACTIONS list. It even works on words.



Remember that ITEM, which does the same thing as NTH, is provided as a primitive in Terrapin Logo version 2.0.

### *Projects*

10. Write a procedure that takes a number from 1 to 26 as input and outputs the corresponding letter of the alphabet.
11. Using the procedure you wrote in project 10, write a new procedure that takes a list containing a number from 1 to 26 and again outputs the corresponding letter of the alphabet.
12. Using the procedure you wrote in project 11, write a new procedure that takes a list of exactly two numbers ranging from 1 to 26 and outputs a two-letter word with the corresponding letters of the alphabet.
13. Using the procedure you wrote in project 12, write a new procedure that takes a list of exactly three numbers ranging from 1 to 26 and outputs a three-letter word with the corresponding letters of the alphabet.
14. Using the reasoning suggested in this chapter, write a new procedure that takes an arbitrary length list of numbers ranging from 1 to 26 and outputs the word composed of the corresponding letters of the alphabet.
15. Using PEOPLE, ACTIONS, NTH (or ITEM), and Logo primitives PR, SE, and RANDOM, write a procedure that prints random sentences. (Write subprocedures that do parts of the job and then combine them.)

## *Some Important Primitives Used in this Chapter*

---

The following summary gives a brief synopsis of some commonly used primitives. It is by no means an exhaustive listing. If you don't find what you want, consult the Logo Command Glossary.

---

The primitives that manipulate Logo objects can be classified into four categories:

- 1) Those that assemble objects
- 2) Those that decompose objects
- 3) Those that determine the nature of objects (i.e. Predicates)
- 4) Those that pass objects back and forth among procedures, to and from variable names, and between the user and the procedure.

Primitives that assemble Logo objects:

**WORD** — Creates a word (a set of contiguous characters) from two inputs. Inputs may be words, characters, or procedures that output words/characters.

**SENTENCE (SE)** — Creates a list from two inputs. Inputs may be words, lists, or procedures which output words/lists. Unlike **LIST**, **SENTENCE** returns a list containing no sub-lists.

**LIST** — Like **SENTENCE**, creates a list from two inputs. If either input is a list, it will appear as a sub-list in the newly created list.

**FPUT** — Creates a list from two inputs, the second of which must be a list. The new list created by FPUT consists of the first input (a word or list) followed by the elements of the second input.

**LPUT** — Same as FPUT, except that LPUT creates a list consisting of the elements of the second input followed by the first input.

Primitives that decompose Logo objects:

**FIRST** — Outputs the first element of its input. If the input is a word, FIRST outputs a character; if the input is a list, FIRST outputs the first element of the list.

**BUTFIRST (BF)** — Takes one input and outputs all but the first element.

**LAST, BUTLAST (BL)** — Corresponding operations for last element of input.

**COUNT** — Takes a single input, a word or a list. Outputs the number of characters in the word, or the number of elements in the list. (Remember that Logo treats a sub-list as a single element of the larger list.)

**ITEM** — Takes two inputs; the first input must be a number, and the second must be a word or list. Outputs the nth element of the second input.

Note that COUNT and ITEM are not primitives in Logo versions prior to version 2.0.

Primitives that determine the nature of an object:

**WORD?** — Outputs “TRUE if the input is a word; otherwise, outputs “FALSE.

**LIST?** — Outputs “TRUE if the input is a list; otherwise, outputs “FALSE.

**NUMBER?** — Outputs “TRUE if the input is a number; otherwise, outputs “FALSE.

**EMPTY?** — Outputs “TRUE if the input is the empty list or the empty word ([ ] or “ ”); otherwise, outputs “FALSE.

**MEMBER?** — Takes two inputs. Outputs “TRUE if the first input is an element of the second input; otherwise, outputs “FALSE.

Note that **MEMBER?** and **EMPTY?** are not primitives in Logo versions prior to version 2.0.

Primitive that passes an object from one procedure to another:

**OUTPUT (OP)** — Causes a procedure to **STOP** and output an object to another procedure or primitive.

Primitives that pass objects to and from variable names:

**MAKE** — Takes two inputs. The first input becomes the name associated with the value of the second input.

**THING** — Takes a variable name as an input. Outputs the value associated with the name. A colon (:) prefixed directly to a name is the abbreviation for THING. Thus, THING “A is the same as :A.

Primitives that pass objects to and from the user:

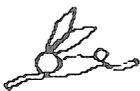
**REQUEST (RQ)** — Waits for the user to type an input line followed by <RETURN>. Outputs the input line as a list to the calling procedure.

**READCHARACTER (RC)** — Takes a character typed at the key board and outputs it as a word to the calling procedure. (Remember that RC does not wait for you to type <RETURN>.)

**RC?** — Outputs “TRUE if a keyboard character is pending; otherwise, outputs “FALSE.

**PRINT (PR)** — Prints its input on the screen (or on the printer, if specified) followed by <RETURN>. Input may be a word or a list. Notice that PRINT strips away all brackets and single-quotes.

**PRINT1** — Prints its input on the screen without <RETURN>. Otherwise, exactly like PRINT.



Also, note that certain Logo primitives can take extra inputs if the entire command is enclosed in parentheses, e.g. (PRINT :LENGTH :HEIGHT :WIDTH). The primitives are LIST, WORD, SENTENCE, PRINT, and PRINT1. In this situation, LIST and SENTENCE may also take one input instead of two.

When using parentheses to indicate extra inputs, be sure to put a space before the closing parenthesis. Otherwise, Logo may assume that the parenthesis is part of the last input and complain that

(primitive) NEEDS MORE INPUTS

### ***Definitions of Words and Lists*** ***CHAR***

We have not yet carefully defined Logo's two types of objects, words and lists. A word, the simplest data object, consists of any continuous string of characters. You've seen several already; here are some other examples:

90  
3.1416  
HI  
ANTIDISESTABLISHMENTARIANISM  
HENRY.THE.8TH  
XYZ  
R2D2

As you can see, numbers are Logo words, long and short English words are Logo words, and even arbitrarily spelled symbols can be Logo words. Experience has already taught you that when you type several Logo words, spaces separate them instead of becoming part of them.

If you need words that contain odd characters like <SPACE> in them, you can surround them with single-quotes. In the experiment that follows, type carefully, remembering to put in all the double-quote

characters and single-quote characters just as they are shown and to type a space between the first A and the B. Clear the text screen and type

```
""A BC'  
PRINT ""A BC'  
[A BC]  
PRINT [A BC]  
LAST ""A BC' and LAST [A BC]
```

Your screen should look like this:

```
""A BC'  
RESULT: 'A BC'  
PRINT ""A BC'  
A BC  
[A BC]  
RESULT: [A BC]  
PRINT [A BC]  
A BC  
LAST ""A BC'  
RESULT: C  
LAST [A BC]  
RESULT: BC
```

Notice that PRINT and other primitives (except OUTPUT) strip away brackets and single-quotes.

The following procedure ODDWORD creates a word of three other words, two of which have spaces in them. Define the procedure, typing carefully. Be sure to type a space before the second parenthesis. (See the preceding glossary if you're not sure why.)

```
TO ODDWORD
  OP ( WORD "'A BA' "'BY B' "OY )
END
```

Now try these experiments with the odd word that ODDWORD outputs.

```
PR ODDWORD
PR NTH 1 ODDWORD
PR NTH 2 ODDWORD
PR NTH 3 ODDWORD
PR NTH 10 ODDWORD
PR LAST ODDWORD
PR WORD NTH 2 ODDWORD ODDWORD
PR WORD "'<space><space><space>' ODDWORD
```

Even though the word that ODDWORD outputs contains spaces, it is a word. Even though it looks like a list when printed, it behaves like a word. The LAST of it is the letter Y, not the word BOY.

A space can be typed, and the single-quote character allows you to insert that space inside a word, but there are some characters that cannot be typed into a procedure at all. An example is the <CTRL> G character. If you were to try typing

```
PR "'<CTRL> G'
```

to Logo, it would say STOPPED! before you reached the second single-quote. But there is a way to include even strange characters like that in a word. The Logo primitive CHAR outputs the character which corresponds to the ASCII code it is given.

The ASCII codes for <CTRL> A through <CTRL> Z are 1 through 26. The codes for capital A through capital Z are 65 through 90, or 64 larger. Thus, you will get the same effect if you type

`PR CHAR 65` or `PR "A`

Empty words — words that contain no characters at all, not even a space — also exist. When typing a command to Logo, one way to indicate you are referring to the empty word is by following a “ with a <SPACE> or <RETURN>. (The <SPACE> separates the word from what follows, and is not part of the word.)

See, for example, the procedure `EMPTY?`, which tests to see if its input `OBJECT` is either the empty word or the empty list.

A list is an ordered collection of Logo objects. Its elements can be words or other lists. Here are some examples of lists:

```
[COLORS [BLUE GREEN YELLOW RED] SIZES [LARGE  
SMALL]]  
[555-2561 617-4436 918-9961]  
[[FD 70] [RT 120] [FD 70] [RT 120] [FD 70] [RT 120]]  
[]
```

The matched left and right square-brackets show the scope of a list. The first list contains four elements, the second and fourth of which are lists themselves and thus are grouped together with the square-brackets.

The second list contains three elements, each a word denoting a telephone number. The third list contains six sublists, each of which contains a Logo command. The fourth list is empty; it contains no elements at all.

Spaces separate elements of the list. The number of spaces signifies nothing, and in fact, more than one space between two elements will be ignored by Logo.

***Some Details of Programming in Logo:  
Variables, Passing Objects, Logo's Way of  
Understanding Commands, and Logo's  
Messages When It Doesn't Understand***

Type this operation to Logo:

**WORD "CAT "S**

As has happened frequently in this chapter, we have suggested you type something to Logo that caused it to respond with the word "RESULT:" followed by the result of the operation. Logo includes the message RESULT: to remind you that it has computed a result, but you have not told it what to do with the result. Compare the effect of this command:

**PR WORD "CAT "S**

Both times, the word CATS appeared, but the second time you told Logo what to do with the result (to print it) and so that is what it did.

You can predict the result of these operations:

**WORD "HORSE "S  
WORD "DOG "S**

In each case, you typed

**WORD somethingorother "S**

suggesting a procedure that might look a bit like this:

**TO PLURAL :SOMETHINGOROTHER  
WORD :SOMETHINGOROTHER "S  
END**

Of course, since names of procedures and variables are arbitrary, you could choose names that are easier to type. NOUN, or IT might be good choices for the variable name.

**TO PLURAL :NOUN  
WORD :NOUN "S  
END**

Why did we switch from quote CAT and quote DOG and quote HORSE to colon NOUN? When you typed

**WORD "CAT "S**

CAT was the word you wanted to attach the S to. In the procedure, the word NOUN only stands for the word you want to attach the S to, but it is not the real word. You still want the procedure to work on words like CAT, DOG, and HORSE.

---

Remember the tiresome joke?

Dale: Bet you've never heard of the word  
"antidisestablishmentarianism!"

Dana: Of course I have.

Dale: Pooh! I bet you can't even spell it.

Dana: Of course I can.

Dale: Go ahead. Let's see if you can spell it.

Dana: A, n, t, i, d, i . . .

Dale: Hah! Wrong already! "It" is spelled  
"i t."

Dale is playing with the confusion between what a word is and what it stands for. When you speak, you change your tone of voice when you need to make that clear. Consider, for example, how you might say the words

"Please say your name"

to Dale if you really wanted Dale to answer "your name" instead of "Dale"? When you write, you use quotation marks to help make your meaning clear. And when you program in Logo, the quotation marks again mean "take this word literally" as they do in written English.



However, Logo's rule is different from the rule in English: in Logo no quotation mark is placed after the quoted word and that one quotation mark applies to only one word. When you need to indicate that more than one word is to be taken literally, you must either separately quote each word, this way

"YOUR "NAME

or enclose all of the words in square brackets,  
this way

[YOUR NAME]

Now type in the procedure:

```
TO PLURAL :NOUN  
  WORD :NOUN "S  
END
```

To run it, type PLURAL followed by a quoted word like this:

```
PLURAL "CAT  
PLURAL "HORSE
```

Your screen will look like this:

```
PLURAL "CAT  
YOU DON'T SAY WHAT TO DO WITH CATS, IN LINE  
  WORD :NOUN "S  
  AT LEVEL 1 OF PLURAL
```

Inside the procedure PLURAL, Logo has created an object and does not know what to do with that object. It is telling you what problem it was having and exactly where it encountered the problem.

Logo tells which of your procedures confused it (in this case, only one of your procedures, PLURAL, was involved, but there might have been more).

It tells the line in which it got stuck. And it tells the "level" at which it got stuck — how many procedures it was already trying to execute when the error occurred.

To see the meaning of level, create another procedure that runs PLURAL.

```
TO TRYLEV  
  PLURAL "CAT  
END
```

Try it.

```
TRYLEV  
YOU DON'T SAY WHAT TO DO WITH CATS, IN LINE  
  WORD :NOUN "S  
  AT LEVEL 2 OF PLURAL
```

Note that the level is now 2. TRYLEV is the first level, the command you typed to the “top level” of Logo. Since PLURAL is “within” TRYLEV, its level is 2. Level can be useful information when debugging complex programs.

Before you can tell Logo what to do with the object it creates inside PLURAL, you have to decide that for yourself. You know how to tell Logo to PRINT the result immediately, but perhaps you want to do something more complicated with the plural word before printing it.

Suppose, for example, you want to create a procedure that brags about your pet like this:

```
I LIKE some-pet-or-otherS.  
some-pet-or-otherS ARE GREAT,  
BUT MY some-pet-or-other IS THE BEST!
```

In the second sentence, we want the plural to be tucked into the sentence before it is printed, and in the first sentence we need to do two things before printing — attach a period to the plural and then stick the whole thing at the end of the sentence.



Since we want to use the object that PLURAL creates in different ways, it would be nice if PLURAL would hand the object back to us to manipulate further as we wish. This is accomplished by telling it to OUTPUT the object.

Edit PLURAL to insert the word OUTPUT (or its abbreviation OP) in the proper place. The procedure will now look like this:

```
TO PLURAL :NOUN
  OUTPUT WORD :NOUN "S
  PRINT [DONE]
END
```

Now run it again as you did before.

```
PLURAL "CAT
PLURAL "HORSE
```

This time, your screen should say:

```
PLURAL "CAT
RESULT: CATS
PLURAL "HORSE
RESULT: HORSES
```

That is precisely what we want. PLURAL has computed the result, and we are still free to decide what to do with it.



But what became of the **DONE** that we told **PLURAL** to print? **OUTPUT** tells a procedure not only to return a value, but to stop immediately. If it is important that **PLURAL** announce when it is done, it must print **DONE** before it is done. (It can't do anything after it is done!)

However, if **PLURAL** is to be used inside another procedure, perhaps one that brags about pets, **PLURAL** probably should not print anything anyway. It should do its job quietly, and let the superprocedure that uses it decide what to print and when.

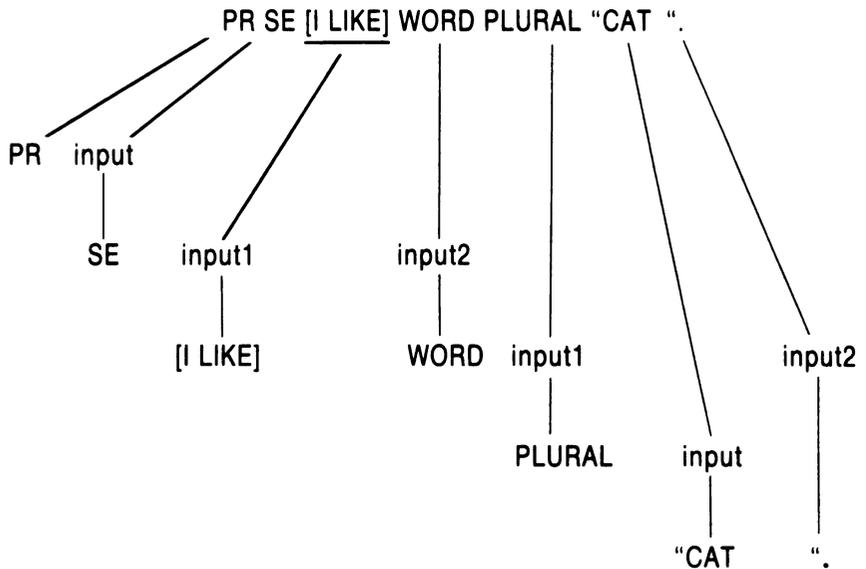
Edit **PLURAL** again to remove the useless line **PRINT [DONE]**.

Try to predict what each of these commands will do, and then type them to see how each works:

```
PR SE [I LIKE] PLURAL "CAT
PR WORD "TOM PLURAL "CAT
PR WORD PLURAL "CAT ".
PR SE [I LIKE] WORD PLURAL "CAT ".
```

## *How Logo Interprets a Command*

It is worth spending a moment to understand how Logo interprets a command as complex as the last one.



Logo reads from left to right, but as you will see by following the diagram above and the discussion below, PLURAL is the first operation to be executed.

First Logo sees the word PR. That means that it will have to print whatever follows. So before executing PR, Logo must read further to see what follows. PR must wait.

Instead of finding an object, Logo encounters another operation, SE. Furthermore, this primitive requires two inputs of its own, so again Logo must read on to find them. PR waits for SE and SE waits for its inputs.

Logo finds the object [I LIKE] as a first input to SE. But SE needs another, so Logo reads further.

Next it finds WORD. Again, this is not an object but an operation. As before, this primitive requires two inputs, so Logo reads still further.

The next thing it finds is, again, not an object but another operation, PLURAL. PLURAL requires one input, so Logo must still look further.

This time Logo finds an object, "CAT — and since PLURAL needs only the one input, it can now execute. It outputs CATS which becomes the first input to WORD. Still, WORD requires a second input which Logo has not yet seen. So, now — after executing PLURAL "CAT — Logo continues to read through the original line and finds the object ". at the end of it.

Logo has now found two objects — CATS and . — to use as inputs to WORD. WORD can now execute, outputting CATS., which becomes the second input to SE. SE can now execute, outputting [I LIKE CATS.] which becomes the input to PR. PR can now execute, printing (not outputting!)

I LIKE CATS.

This left-to-right reading but (seemingly) right-to-left execution can be confusing sometimes. Both of the following command lines will cause Logo to complain. Try them out to see when and where the complaint occurs, and then use an analysis like the one given above to understand what Logo was doing when it had to stop.

```
PR SE [I LIKE] WORD PLURAL [CAT] ".
PR SE [I LIKE] WORD PLURAL "CAT [.]
```

Sometimes the complexity of a line makes it difficult to understand even by the person who first wrote it. Before reading on, try to predict what the following Logo command will do. Then type it in to try it, and read on.

```
PR SE WORD LAST PLURAL "CAT "CAT "CAT
```

When you write complex Logo commands — especially if you are writing them for other people to understand, but often even for yourself — it can be a good idea to use parentheses to help group the parts of the command. Logo will interpret the command according to its rules equally easily with or without the parentheses, but people find the added punctuation helpful.

You should decide for yourself how much parenthesizing to do. Sometimes, using the maximum is best. At times, the maximum looks too cluttered, and just a few are better. The choice is entirely a matter of taste. For example, that last command might be parenthesized in the following ways. Which way makes it visually clearest to you what the command does?

```
PR (SE (WORD (LAST (PLURAL "CAT )) "CAT ) "CAT )
PR SE (WORD LAST (PLURAL "CAT ) "CAT ) "CAT
PR SE (WORD (LAST PLURAL "CAT ) "CAT ) "CAT
```

---

Be sure to type a space between “CAT and ) — otherwise, Logo will read the parenthesis as part of the word and will complain that the primitive needs more inputs, i.e. Logo can’t find a matching right parenthesis.

---

*Using Logo Predicates and Creating New Ones: LIST?, WORD?, MEMBER?, and the Structure of IF, THEN, and ELSE*

All along, we’ve been using IF without any explanation of its structure. The IF statement has three parts:

- 1) The IF itself
- 2) A condition which may be either TRUE or FALSE. (In this case, the condition is LIST? :NOUN which tells whether it is TRUE or FALSE that NOUN is a list.) The condition may include modifiers such as NOT, ALLOF, and ANYOF, either individually or in combination.
- 3) The THEN clause: an action to perform if the condition is TRUE.

An IF statement can also have an additional two parts when desired.

- 1) The word ELSE and
- 2) An action to perform if the condition is FALSE

Finally, as mentioned earlier, the word THEN can be used optionally between the condition and the action-if-true.

Thus, an IF statement can take the following four forms.

IF condition action-if-true

IF condition THEN action-if-true

IF condition action-if-true ELSE action-if-false

IF condition THEN action-if-true ELSE action-if-false



The condition always contains a “predicate,” a Logo primitive or user procedure that answers a True-False question by outputting TRUE or FALSE.

In the case of LIST? :NOUN, the True-False question is “NOUN is a list! True or false?” If the statement is false, LIST? outputs FALSE. If the statement is true, LIST? outputs TRUE.

You will often need to create your own predicates, so it is important to become familiar with their behavior. Type these expressions to Logo:

LIST? PEOPLE

LIST? FIRST PEOPLE

LIST? NTH 3 PEOPLE

Each time, Logo should announce a result, showing that LIST? output a word, either TRUE or FALSE, depending on whether the input was a list or not.

You have used several other predicates. When you used `NUMBER? :CHTR` in the `EASY` procedure for `QUICKDRAW` in project 5, it output `TRUE` or `FALSE` depending on the truth of the statement “`CHTR` is a number.”

In `GREET`, you used the expression `EMPTY? :PERSON`. It worked the same way.

And, in the expression `IF :CHTR = "F`, the equal sign also outputs `TRUE` or `FALSE` depending on the truth of the statement that `CHTR` equals “`F`. (The `=`, like the `+`, comes between its inputs.)

The `RC?` primitive (which takes no inputs), the `WORD?` primitive (which takes one input), and the procedure `MEMBER?` (which takes two inputs) are also predicates.

When you first defined `MEMBER?` and `EMPTY?` we postponed explaining how they work. You are now probably ready for that explanation.

Look first at the procedure `EMPTY?`.

```
TO EMPTY? :OBJECT
  OUTPUT ANYOF :OBJECT = [] :OBJECT = "
END
```

There are two equal-signs in the procedure. Each one outputs `TRUE` or `FALSE`.

The first one outputs `TRUE` if `OBJECT` is the empty list (and `FALSE` otherwise). The second outputs `TRUE` if `OBJECT` is the empty word (and `FALSE` otherwise).

ANYOF takes two (or more) inputs, and it outputs TRUE if *any* of them is TRUE.

Finally, the purpose of the command OUTPUT in the procedure is to tell EMPTY? to output whatever ANYOF outputs. Thus, EMPTY? outputs TRUE if any of these conditions is true:

the OBJECT is [], the empty list  
the OBJECT is " ", the empty word

Otherwise, EMPTY? outputs FALSE.

Now look at the procedure MEMBER?.

```
TO MEMBER? :ELEMENT :OBJECT
  IF EMPTY? :OBJECT OUTPUT "FALSE
  IF :ELEMENT = FIRST :OBJECT OUTPUT "TRUE
  OUTPUT MEMBER? :ELEMENT BUTFIRST :OBJECT
END
```

Surely ELEMENT cannot be a member of OBJECT if OBJECT has no members! So, if OBJECT is empty, MEMBER? should output FALSE.

```
IF EMPTY? :OBJECT OUTPUT "FALSE
```

If ELEMENT is the first member of OBJECT, the procedure need check no further. It can already answer TRUE that ELEMENT is a member of OBJECT.

```
IF :ELEMENT = FIRST :OBJECT OUTPUT "TRUE
```

Now, the recursive step. If there are more elements in OBJECT (because OBJECT is not empty), but ELEMENT is not the first element of OBJECT, it may still be one of the later elements. If it is a member of BUTFIRST :OBJECT, it is clearly a member of OBJECT.

So, if ELEMENT is not the first element of OBJECT, but there are more elements, the procedure may answer the original question — MEMBER? :ELEMENT :OBJECT — by outputting the answer to a simpler question — MEMBER? :ELEMENT BUTFIRST :OBJECT.

**OUTPUT MEMBER? :ELEMENT BUTFIRST :OBJECT**

---

You've probably noticed that every predicate has the -? suffix. We will continue to use this convention throughout the chapter. When you see a primitive or procedure name ending in -?, you'll know that its behavior is to output TRUE or FALSE.

---

### *Projects with Predicates*

16. Define the predicate, TO VOWEL? :LETTER, that outputs TRUE if LETTER is a vowel, and FALSE otherwise.
17. Define the predicate, TO YES?, that requests a typed line from the user and outputs TRUE if that line is a reasonable synonym of "yes," FALSE if the line is a reasonable synonym of "no," and otherwise prints a message requesting clarification and calls itself recursively to try again. Decide on the synonyms you will accept.

## *Ordered Rules*

Testing out PLURAL reveals a number of bugs. Try the following inputs:

PLURAL "DOG  
PLURAL "TURTLE  
PLURAL "FLY  
PLURAL "FINCH  
PLURAL "FISH  
PLURAL "MOUSE  
PLURAL "CHILD  
PLURAL "FOX  
PLURAL [FOX TERRIER]

Two different kinds of bugs can be noted. One is that some of the plurals are not correct. The procedure's only rule is to tack on an S, and it must be taught more about English plurals.

The other bug is that it couldn't handle [FOX TERRIER] at all. In this case, Logo complains that WORD doesn't like [FOX TERRIER] as input in the context of OUTPUT WORD :NOUN "S in the procedure PLURAL.

Logo, of course, is not biased against cute dogs. It is merely trying to say that WORD glues pieces of words — not lists — together to make other words.

To solve this problem the procedure doesn't need more knowledge about English, but rather needs more knowledge about its inputs. We will show a solution to three of the problems and suggest several other problems as projects for you to work on.

First, the FOX TERRIERS. If NOUN is a list, PLURAL should probably do most of its work on the last word of the list.

It should OUTPUT a SENTENCE composed of all BUT the LAST word of NOUN, and the PLURAL of the LAST word of NOUN. The Logo instruction would look like this:

```
IF LIST? :NOUN OP SE BL :NOUN PLURAL LAST :NOUN
```

Edit PLURAL and add that line.

```
TO PLURAL :NOUN
  IF LIST? :NOUN OP SE BL :NOUN PLURAL LAST :NOUN
  OUTPUT WORD :NOUN "S
END
```

Try  
PLURAL [BLUE BIRD]  
or  
PLURAL [RICKETY LADDER]  
in addition to  
PLURAL [FOX TERRIER].

Right now, PLURAL "FOX outputs FOXS. To get it to output FOXES, we might include a simple test to see if X is the last letter of NOUN. If it is, we should attach ES rather than S to NOUN.

```
IF "X = LAST :NOUN OP WORD :NOUN "ES
```

Where shall we put this new instruction? Certainly not as the last instruction, because if it came after the line OUTPUT WORD :NOUN "S, the procedure would

never get to it. In this case, it makes little difference in PLURAL's behavior whether the new instruction comes first or second.

Edit PLURAL and define it to look like this:

```

TO PLURAL :NOUN
  IF LIST? :NOUN OP SE BL :NOUN PLURAL LAST :NOUN
  IF "X = LAST :NOUN OP WORD :NOUN "ES
  OUTPUT WORD :NOUN "S
END

```

Now test it out. Does PLURAL give the right plural for FOX? What about [FOX TERRIER]? And what about [GREY FOX]?

A third problem is teaching the procedure how to handle the really strange cases, like CHILD, MOUSE, FOOT, and SHEEP. First, we must make a list of the exceptions.

```
MAKE "EXCEPTIONLIST [CHILD MOUSE FOOT SHEEP OX]
```

PLURAL must be told something like "If the noun is one of the exceptions . . ."

```
IF MEMBER? :NOUN :EXCEPTIONLIST . . .
```

". . . then output the special plural associated with that particular noun."

```
. . . OUTPUTspecial.plural.something.or.other
```

Where should that special-plural information reside? It could be another procedure:

```
TO EXPLU :NOUN
  IF :NOUN = "CHILD OP "CHILDREN
  IF :NOUN = "SHEEP OP "SHEEP
  IF :NOUN = "MOUSE OP "MICE
  IF :NOUN = "FOOT OP "FEET
  etc.
END
```

In that case the new addition to PLURAL would be:

```
IF MEMBER? :NOUN :EXCEPTIONLIST OP EXPLU :NOUN
```

Another approach, in some ways simpler, is to put each piece of special plural information into a box whose name is the noun itself. So we could put CHILDREN into a box named CHILD, and put SHEEP into a box named SHEEP, etc.

```
MAKE "CHILD "CHILDREN
MAKE "SHEEP "SHEEP
MAKE "OX "OXEN
```

Then IF the NOUN were a member of the EXCEPTIONLIST, PLURAL should OUTPUT the object (THING) inside a box associated with the NOUN. The Logo would look like this:

```
IF MEMBER? :NOUN :EXCEPTIONLIST OP THING :NOUN
```



This is strange-looking code, indeed. What can THING :NOUN mean? If :NOUN is CHILD, then THING :NOUN is the THING of CHILD, and if :NOUN is SHEEP, then THING :NOUN is the THING of SHEEP.

And what is the THING of CHILD? CHILDREN, because earlier you typed MAKE "CHILD "CHILDREN. So, too, the THING of SHEEP is SHEEP.

### *Projects with PLURAL*

18. It matters where you place the new instruction. Below, we show PLURAL defined in three different ways, with the new instruction placed first, second, and third.

Define PLURAL each way and test it out enough to determine which way(s) work. (Why do we not bother even trying it as the fourth instruction?)

```
TO PLURAL :NOUN
IF MEMBER? :NOUN :EXCEPTIONLIST OP THING :NOUN
IF LIST? :NOUN OP SE BL :NOUN PLURAL LAST :NOUN
IF "X = LAST :NOUN OP WORD :NOUN "ES
OUTPUT WORD :NOUN "S
END
```

```
TO PLURAL :NOUN
IF LIST? :NOUN OP SE BL :NOUN PLURAL LAST :NOUN
IF MEMBER? :NOUN :EXCEPTIONLIST OP THING :NOUN
IF "X = LAST :NOUN OP WORD :NOUN "ES
OUTPUT WORD :NOUN "S
END
```

```
TO PLURAL :NOUN
IF LIST? :NOUN OP SE BL :NOUN PLURAL LAST :NOUN
IF "X = LAST :NOUN OP WORD :NOUN "ES
IF MEMBER? :NOUN :EXCEPTIONLIST OP THING :NOUN
OUTPUT WORD :NOUN "S
END
```

19. To teach PLURAL when to add ES at the end, you sometimes must look at the last letter of :NOUN and sometimes at the last two letters. Figure out the rule, and then make PLURAL smart enough to output the correct plural for WISH.

Does it handle [BEST WISH] correctly? Can it handle BOSS? FINCH? Does it still do the right thing for FOX? Are you satisfied with the way it handles FISH?

20. Teach it to do the right thing with FLY.

21. Does PLURAL handle BOY and KEY correctly? If not, fix it.

22. In project 15 above, you wrote a program to generate random sentences out of the nouns in PEOPLE and the verbs in ACTIONS. Without changing any of the details of the program, you can add HE, [MY MOTHER], and certain other nouns and pronouns to PEOPLE, but, as the program stands, it will stop making grammatical sentences if PEOPLE contains elements like YOU, or [CHARLES AND DIANA].

This can be fixed. ACTIONS now contains the verbs [LOVES [DREAMS ABOUT] KISSED HATES [CAN'T STAND] LIKES]. If it were changed slightly, a program similar to PLURAL could add the proper S or D endings when needed. This is what ACTIONS would need to contain: [LOVE [DREAM ABOUT] KISS HATE [CAN'T STAND] LIKE]

First, write a procedure, TO FIXVERB :VERB (the logic will be similar to PLURAL, but not the same) that adds S or ES or nothing to any verb that is its input. Write

another procedure, `TO PAST :VERB` that adds `D` or `ED` (or makes whatever other change is needed) to put the verb in past tense.

Now write a procedure that takes a subject such as `YOU` or `[THE TURTLE]` and figures out whether the verb needs to be “fixed” or not.

With these procedures you can make a better sentence generator.

23. If you know French, you could do the same thing for French verbs. Of course, the rules are more complicated, and you will need to do more designing and more programming.

But you have all the techniques now, and some good strategies. It is probably a good idea to have small procedures, each of which does a specific job, rather than one large procedure that does everything.

A set of procedures that conjugate French verbs can be used in a program that generates French sentences. It can also be used as part of a quiz on French verbs. The next section will deal with quiz programs.

### *Quiz Programs: More About REQUEST (RQ)*

When `REQUEST` is encountered in a procedure, the procedure stops and waits until the user presses `<RETURN>`. Anything that the person has typed prior to the `<RETURN>` is then output by `REQUEST` as a list.



If the person types a dozen words, REQUEST outputs a 12 word list. If the person types nothing, REQUEST outputs an empty list. If the person types a single word, REQUEST outputs a one word list. The important thing to remember is that REQUEST's output is always a list, never a word.

Here is a model of a simple quiz program. QUIZ "gives" the quiz, using QA to handle each question/answer pair. QA is a subprocedure that prints the question, requests an answer from the quizee and if that answer is the official ANSWER, prints "YUP!" and stops. If the answer is not judged to be correct, QA prints the correct answer.

```
TO QUIZ
  PRINT [TEST YOUR BRILLIANCE!]
  QA [WHO IS BURIED IN GRANT'S TOMB?] [GRANT]
  QA [WHY DID THE CHICKEN CROSS THE ROAD?] [TO GET GAS]
  QA [HOW DO YOU SPELL RELIEF?] [CORRECTLY]
END
```

```
TO QA :QUESTION :ANSWER
  PRINT :QUESTION
  IF :ANSWER = REQUEST PR [YUP!] STOP
  PR SE [NOPE! THE ANSWER IS:] :ANSWER
END
```

On the surface, the logic of the addition quiz below is identical to QUIZ. ADDQUIZ "runs" the test, calling ADDQ with each number pair. ADDQ's inputs are two numbers to add. It doesn't need to be told the answer, as QA did, because it can figure out the answer itself.

Its first line prints the question — for example  $7 + 9 =$  — and waits for the answer at the end of the line. The second line waits for the user to type an answer and compares it to the calculated correct answer.

If the user's answer is the same, ADDQ prints YAY! and stops. Otherwise it prints the correct answer. It seems like it ought to work. Yet it has a bug.

Define ADDQUIZ and its subprocedure ADDQ, try the quiz (by typing ADDQUIZ), and see if you can make it work properly before reading on.

```
TO ADDQUIZ
  PRINT [TEST YOUR ADDITION]
  ADDQ 7 9
  ADDQ 8 5
  ADDQ 9 8
END
```

```
TO ADDQ :NUMBER1 :NUMBER2
  PRINT1 ( SE :NUMBER1 "+ :NUMBER2 "' = ' )
(buggy line) IF (:NUMBER1 + :NUMBER2) = REQUEST PR [YAY!] STOP
  PR ( SE "NOPE, :NUMBER1 "+ :NUMBER2 " = :NUMBER1
    + :NUMBER2 )
END
```



Forgetting that REQUEST always outputs a list is a frequent source of bugs. As the procedure ADDQ is currently written, it will never print YAY!. (:NUMBER1 + :NUMBER2) is a number (and therefore a word), while REQUEST outputs a list — the two can never be equal.

To make them comparable, we need to change REQUEST's list into a word. We can do this by taking the FIRST of REQUEST. Thus ADDQ will work if REQUEST is replaced by FIRST REQUEST or its abbreviation FIRST RQ.

Make that change and verify that ADDQ now works by typing

**ADDQUIZ**

*Projects with REQUEST*

24. In general, there is more than one right way to answer a question, yet QUIZ considers only one answer correct. Suppose QUIZ were rewritten this way:

**TO QUIZ**

**PRINT [TEST YOUR BRILLIANCE!]**

**QA [WHO IS BURIED IN GRANT'S TOMB?]**

**[[GRANT] [GENERAL GRANT]]**

**QA [WHY DID THE CHICKEN CROSS THE ROAD?]**

**[[TO GET GAS] [FOR FUN] [TO LAY EGGS]]**

**QA [HOW DO YOU SPELL RELIEF?]**

**[[CORRECTLY] [ROLAIDS]]**

**END**

In each case, a different number of correct answers has been provided. Rewrite QA to account for the choices of answers.

25. The biggest difference between the subprocedure ADDQ for the addition quiz and the subprocedure QA for the general information quiz is that ADDQ does not need to be told the answer to the question. Because

number pairs can be selected at random, even the questions do not have to be specified one by one.

This means that the quiz can keep generating questions as long as desired, without having had to list all the questions beforehand. Write an addition quiz that poses problems with randomly selected numbers no larger than 12, and keeps going until the quizee gets ten of them correct.

26. Add a bit more intelligence to the addition quiz. Let it start by posing addition problems with very small numbers, say under 4. If a person gets three of them correct, the program begins giving slightly larger numbers, and so on. The program stops if a person gets two wrong in a row.

27. Change ADDQ's title line to read TO ADDQ :TRIES :NUMBER1 :NUMBER2 and change the procedure to allow a person two tries at the same problem before the problem is changed.

ADDQ should perhaps say TRY AGAIN if the person gets the wrong answer the first time, but should not give the correct answer until the person gets the problem wrong a second time. Then it should quit and go on to the next problem.

28. Using the procedures PICK and QA that were defined earlier in the solution to project 15, write a STATESQUIZ program that picks question-answer sets off a pre-defined list. You might store the information in a form something like this:

```
MAKE "STATES [[OHIO COLUMBUS] [[NEW YORK]
           ALBANY] [GEORGIA ATLANTA]]
```

29. If you used the exact list shown in project 28, and wrote a working STATESQUIZ, it may be hard to add states that have multi-word capitals to the list. For example, if you now type MAKE "STATES LPUT [IOWA [DES MOINES]] :STATES, the chances are that when STATESQUIZ asks what the capital of Iowa is, it will not accept any answer as correct.

Fix the quiz so that it works, either by redesigning the data-base (:STATES) to be more consistent, or by making the procedures smart enough to handle the inconsistency. (Suggestion: redesigning the database makes the program simpler.)

30. If you have written a French verb conjugator, you can write a quiz similar to ADDQUIZ that selects a verb at random from a list, selects a pronoun, also at random, and asks the person to type in the correct verb form.

*Composing Logo Objects: SENTENCE, WORD, LIST, FPUT, LPUT, TEST, IFTRUE, and IFFALSE*

Here is a procedure, JUNKMAIL, that uses SENTENCE and its abbreviation SE. Define JUNKMAIL, complete with extra spaces as shown below.

```
TO JUNKMAIL :PERSON
  PR SENTENCE [DEAR] :PERSON
  PR [IF YOU ACT RIGHT NOW, YOU HAVE]
  PR [A CHANCE TO WIN A MILLION DOLLARS!]
  PR [WINNING TICKETS, ALREADY MADE OUT]
  PR [IN YOUR NAME, ARE WAITING FOR YOU.]
  PR ( SE [THINK,] :PERSON [, WHAT THAT COULD MEAN!])
END
```

To run it, type JUNKMAIL followed by a list or a word, like this:

```
JUNKMAIL [MS. RACHEL LEVIN]
JUNKMAIL [ABBY]
JUNKMAIL "MIKE
JUNKMAIL PICK PEOPLE
```

Notice, first, its handling of spaces. All of the extra spaces you inserted are missing. Also, because SENTENCE creates a list — outputting DEAR ABBY instead of the word DEARABBY — it appears to leave a space between its inputs. The space, as noted earlier, is not a part of the list, but merely a separator that comes between elements of the list.



SENTENCE always outputs a list. If either input is a word, SENTENCE treats that input as if it were a one-element list. Thus, all four of these expressions output the sentence [DEAR ABBY].

```
SENTENCE "DEAR "ABBY
SENTENCE "DEAR [ABBY]
SENTENCE [DEAR] "ABBY
SENTENCE [DEAR] [ABBY]
```

The last line of JUNKMAIL contains parentheses. By surrounding the primitive SENTENCE and the three objects that follow it, those parentheses tell Logo that the primitive is to accept all three objects as input.



A few Logo primitives — in general, the ones that “associatively combine” their inputs, such as SENTENCE, WORD, and LIST, but

also some others such as PRINT and PRINT1 — have this ability to accept other than their usual number of inputs when surrounded by parentheses.

User-defined procedures cannot be given this feature.

---

The procedure has a formatting bug. We would like it to type,

**THINK, MIKE, WHAT THAT COULD MEAN!**

but the space that separates elements of a list has separated PERSON from the following comma, with this result:

**THINK, MIKE , WHAT THAT COULD MEAN!**

When, in PLURAL, you attached S to one of the words in a list, you were solving a similar problem, but JUNKMAIL adds a new twist.

If we could be certain that PERSON was a Logo word, the change would be simple:

**PR ( SE [THINK,] WORD :PERSON “, [WHAT THAT COULD MEAN!] )**

But this will not work if the input is a list. Since WORD cannot take lists as inputs, the list would first have to be torn apart (using FIRST or LAST to extract the elements, and BUTFIRST or BUTLAST to preserve the rest), and then recomposed (using SENTENCE) after the comma is affixed properly by WORD.

```
PR ( SE [THINK,] BL :PERSON WORD LAST :PERSON ",
      [WHAT THAT COULD MEAN!])
```

Now try JUNKMAIL twice, once with a word and once with a list. What happens?

Since the user is free to input either word or a list, we must take still one more step. We have a choice. One possibility is to test the input with WORD? or LIST? and choose which path to follow depending on the outcome. We can perform either test and write the rest of the IF statement accordingly. So, the logic might be:

```
IF LIST? :PERSON do-the-list-version ELSE
              do-the-word-version
              or
IF WORD? :PERSON do-the-word-version ELSE
              do-the-list-version
```

In either case, the result is a horribly long line that becomes nearly impossible to read. Here is how it might look inside the editor if the LIST? test were used:

```
IF LIST? :PERSON PR ( SE [THINK,] BL :!
PERSON WORD LAST :PERSON ", [WHAT THAT !
COULD MEAN!]) ELSE PR ( SE [THINK,] WO!
RD :PERSON ", [WHAT THAT COULD MEAN!])
```

Logo provides another IF-like construction, TEST, which is useful when several actions must be performed depending on the truth of the tested conditional. TEST is also useful when the actions are very long, as they are in this case.

Here is how the same logic would be written using TEST.

```
TEST LIST? :PERSON
IFTRUE PR ( SE [THINK,] BL :PERSON WOR!
D LAST :PERSON ", [WHAT THAT COULD MEAN!
!])
IFFALSE PR ( SE [THINK,] WORD :PERSON !
", [WHAT THAT COULD MEAN!])
```



There is a less verbose alternative. Since (SE "ABBY) and (SE [ABBY]) both output the list [ABBY], SENTENCE can be used to convert the input, whatever form it started in, into a standard form.

Insert the statement MAKE "PERSON (SE :PERSON) as the first line of JUNKMAIL to force :PERSON to be a list. The parentheses are needed because SE is taking fewer than two inputs. Then, since you know that :PERSON is a list, you need not test and can use just the solution that applies to lists. This application of SE often comes in handy.

LIST, FPUT, and LPUT also compose lists. It is important both to compare their effects by doing some simple experiments (some will be suggested below) and to know why anybody would care about the differences.

First, compare SENTENCE and LIST this way:

```
SE [THIS IS] [A LIST]
LIST [THIS IS] [A LIST]
```

SENTENCE outputs a list whose elements are the elements of its inputs, whereas LIST outputs a list whose elements are its inputs.

When is this important? If you are trying to compose a simple list of words, as in an English sentence, `SENTENCE` is the right choice. Try these:

```
SE [THIS IS A] "SENTENCE
SE "THIS [IS A SENTENCE]
(SE "THIS [IS] "A [SENTENCE])
(SE [THIS IS A SENTENCE])
(SE "THIS "IS "A "SENTENCE )
```

Because `SENTENCE` throws away information about the structure of its inputs, each of these expressions outputs the same list, `[THIS IS A SENTENCE]`. Now try the same sets of inputs using the primitive `LIST` instead of `SE`.

```
LIST [THIS IS A] "SENTENCE
LIST "THIS [IS A SENTENCE]
(LIST "THIS [IS] "A [SENTENCE])
(LIST [THIS IS A SENTENCE])
(LIST "THIS "IS "A "SENTENCE )
```

The structure of the inputs is fully preserved in the output.

```
[ [THIS IS A] SENTENCE ]
[ THIS [IS A SENTENCE] ]
[ THIS [IS] A [SENTENCE] ]
[ [THIS IS A SENTENCE] ]
[THIS IS A SENTENCE]
```

`LIST` is the primitive to use when you need to package objects, unaltered, into a list. Like `SENTENCE`, `LIST` usually takes two inputs, but when parenthesized, it accepts any number greater than zero.

Neither SENTENCE nor LIST allows you to insert an element into an already existing list. This is the job of FPUT and LPUT.

Each takes an object (word or list) as its first argument and a list as its second argument. It then inserts the object into the list either to become the first element of the new list (in the case of FPUT) or the last element of the new list (LPUT), and outputs the new list. Try these:

```
FPUT "THIS [IS A SENTENCE]
LPUT "THIS [IS A SENTENCE]
LPUT [FD 50] [[RT 90] [BK 30] [LT 60]]
```

FPUT and LPUT are important when you are accumulating information gradually and want to keep track of it on a list. This is the reason why LPUT was the proper primitive for storing new names of people that GREET met in the FRIENDLY program that you defined in the section called Some Friendly Introductions.

Because LPUT created its output by packing the new object (in that case, PERSON) into a previously existing list (in that case, KNOWN), its output can later be decomposed back to the original object and list by LAST and BUTLAST respectively.



This inverse relationship of LPUT to LAST and BUTLAST, and of FPUT to FIRST and BUTFIRST is what makes these two primitives so important. This relationship is best shown by an illustration and some experimenting.

The relationship can be summarized this way (type each statement below):

If WOL represents any Logo word or list, e.g.

```
MAKE "WOL [FD 50]
```

and OLD.LIST represents any Logo list, e.g.

```
MAKE "OLD.LIST [[RT 90] [BK 30] [LT 60]]
```

then define NEW.LIST this way:

```
MAKE "NEW.LIST FPUT :WOL :OLD.LIST
```

Now type

```
PR :WOL PR :OLD.LIST PR :NEW.LIST
```

and observe that the following two statements are true:

```
:WOL = FIRST :NEW.LIST  
:OLD.LIST = BF :NEW.LIST
```

Similarly, if you

```
MAKE "D LPUT :WOL :OLD.LIST
```

then these statements are true:

```
:WOL = LAST :D  
:OLD.LIST = BL :D
```

### *An Application of LPUT in Interactive Graphics: RUN*

Look back at the procedure EASY that you defined in the early section called Interactive Graphics. Each time certain characters are pressed, a turtle command is executed.

The screen “remembers” the effect of each command, but the program has no way of knowing what command it executed last. It could not, for example, run through the same sequence of commands again to make another copy of the design on the screen.

Just as FRIENDLY was given a memory, you can add memory to the QUICKDRAW program. Each time a character is pressed, EASY will run the proper command, and also store that command on a list.

Using the simplest combination of the strategies in GREET and in EASY, one might rewrite each line of EASY to look something like this:

```
IF :CHTR = "F THEN FD 10 MAKE "HISTORY FPUT [FD 10]
    :HISTORY
IF :CHTR = "R THEN RT 15 MAKE "HISTORY FPUT [RT 15]
    :HISTORY
IF :CHTR = "L THEN LT 15 MAKE "HISTORY FPUT [LT 15]
    :HISTORY
etc.
```

But there is a way of reducing the amount of repetition. If there was a procedure (let us call it RUN.AND.RECORD) that could take the command as input and be responsible for both the running and

recording of the command, EASY could be written more economically and more understandably as:

```
IF :CHTR = "F RUN.AND.RECORD [FD 10]
IF::CHTR = "R RUN.AND.RECORD [RT 15]
IF :CHTR = "L RUN.AND.RECORD [LT 15]
etc.
```

If RUN.AND.RECORD calls its input MOVE, then the line that records the history of moves might look like this:

```
MAKE "HISTORY (LPUT :MOVE :HISTORY)
```



To run a list that contains a legal Logo command or expression, Logo provides the primitive RUN.

Thus, the procedure that runs and records each move might look like this:

```
TO RUN.AND.RECORD :MOVE
  RUN :MOVE
  MAKE "HISTORY ( LPUT :MOVE :HISTORY )
END
```

To summarize, RUN.AND.RECORD takes an input list containing a Logo command. It RUNs the input, and then tucks it neatly into a list named HISTORY.

Define this new procedure and test it out a few times. As was necessary in the FRIENDLY program, you must first create an empty HISTORY list for RUN.AND.RECORD to add its new moves to.

```
MAKE "HISTORY [ ]
```

Now type these commands. (Use <CTRL>P to repeat the line and the <DEL> key to change the last few characters. It will save you some typing!):

```
RUN.AND.RECORD [FD 30]
RUN.AND.RECORD [RT 120]
RUN.AND.RECORD [BK 10]
RUN.AND.RECORD [RT 24]
RUN.AND.RECORD [BK 5]
```

To print the history list, type

```
PR :HISTORY
```

and notice that it contains a record of the commands that generated the picture on the screen.

```
[FD 30] [RT 120] [BK 10] [RT 24] [BK 5]
```

### *Using the History List: Applying a Command (RUN) to Each Element of a List*

Whole new possibilities are now opened up. Re-running each of these commands will copy the design onto the screen a second time.

Alternatively, you can achieve the effect of “undoing” the last command (BK 5) by erasing the screen, removing the [BK 5] from the history list and running what remains.

---

The INSTANT program on your Utilities Disk uses this strategy. Several of the procedures described in this section are similar to those

used in `INSTANT`. You may want to study that program. See the Graphics chapter for a description of its use.

---

Both of these functions require that you have a procedure capable of doing the same thing — in this case, `RUNning` — to each of the elements of a list.



Normally `RUN` takes a list and executes the command(s) in the list. Here, the list to be run is composed of sub-lists, each of which must be run individually.

The procedure will take the list as input:

```
TO RUN.ALL :COMMANDS
```

If the list is empty, then the job is done, so the procedure stops.

```
IF EMPTY? :COMMANDS STOP
```

If the list is not empty, then perform the required action to the first element of the list.

```
RUN FIRST :COMMANDS
```

And then, following the same logic, deal with the remainder of the list.

```
RUN.ALL BF :COMMANDS
```

Define the procedure `RUN.ALL`.

```
TO RUN.ALL :COMMANDS
  IF EMPTY? :COMMANDS STOP
  RUN FIRST :COMMANDS
  RUN.ALL BF :COMMANDS
END
```



RUN.ALL can be thought of as a model for a whole class of procedures. For instance, you have already seen MEMBER?, NTH, and COUNT. The structure of this kind of procedure is shown in the “ghost” procedure below:

```
TO X.ALL :LIST                                title with input
  IF EMPTY? :LIST STOP                       condition for stopping
  Y FIRST :LIST                               action to take with first element
  X.ALL BF :LIST                              recursive call with BF input
END                                            end
```

Here is a procedure of similar structure which erases a list of procedures.

```
TO ERLIST :PROCS
  IF EMPTY? :PROCS STOP
  RUN LIST "ERASE FIRST :PROCS
  ERLIST BF :PROCS
END
```

---

Type these commands:

```
RUN.ALL :HISTORY
RUN.ALL :HISTORY
REPEAT 2 [RUN.ALL :HISTORY]
PR :HISTORY
```

The picture has changed, but the history list has not. Why? Because RUN.ALL did not record any of the commands it ran; it just ran them.

To “undo” a command, we clear the screen and run all but the last element of the history list. Of course, if the history list is already empty, we cannot undo any more and so should just stop.

Here is a procedure which does that:

```
TO UNDO
  IF EMPTY? :HISTORY STOP
  MAKE "HISTORY BL :HISTORY
  DRAW
  RUN.ALL :HISTORY
END
```

Clear the screen with DRAW and type RUN.ALL :HISTORY. Now type UNDO a few times to see its effect.

### *Projects with History Lists*

31. Edit EASY to take advantage of RUN.AND.RECORD and UNDO. Some changes need to be made in addition to inserting the two new procedures.

Try out all of the features — the old as well as the new — in a variety of combinations to be certain they work together properly. In particular, make certain that UNDO does the right thing when pressed right after you have pressed the D key to erase the screen.

To start up the program with an empty history list, it might be convenient to define this startup procedure:

```
TO STARTUP
  MAKE "HISTORY []
  QUICKDRAW
END
```

32. Add right-curving circles and left-curving circles to QUICKDRAW.

### ***Substituting One Word for Another in a Sentence: A Procedure with Two Recursive Calls***

We will design a procedure that will work like this:

```
SUBST "DOGS "CATS [WE THINK DOGS ARE GREAT]
RESULT: [WE THINK CATS ARE GREAT]
SUBST "X PICK PEOPLE [WE LOVE X MORE THAN ANYBODY]
RESULT: [WE LOVE SANDY MORE THAN ANYBODY]
SUBST "ADV PICK ADVERBS [COLORLESS GREEN IDEAS
  SLEEP ADV]
RESULT: [COLORLESS GREEN IDEAS SLEEP FURIOUSLY]
```

It will serve as a building block for a variety of language activities, and a model for a procedure that can work Mad-Libs.

What is its design? It takes three inputs: a key word it is looking for, a word to replace that one with, and a sentence as a context in which to perform the replacement.

This version of SUBST will replace all occurrences of the key word with the replacement word. Described concretely, it can look through sentences like [WE THINK DOGS ARE GREAT] and wherever it finds DOGS, it substitutes CATS.

The logic is absolutely like the recursive model shown before.

Let's review the model:

```
title with inputs
condition for stopping
action to take with first element
recursive call with BF input
end
```

The title line and stop condition are straightforward. If there is nothing in the sentence CONTEXT, there is nothing to substitute, so the procedure outputs an (identical) empty sentence.

The remaining two lines introduce a new twist. The action to take with the first element is clear: if it is the key word :KEY that we are looking for

```
IF ( FIRST :CONTEXT ) = :KEY
```

the procedure must replace it with :NEW. Replacing the first element of a list means keeping the butfirst. SUBST must output a sentence composed of the new first element with the butfirst of the original CONTEXT. This, by itself, is

```
OP SE :NEW BF :CONTEXT
```

But the object is to catch every occurrence of KEY in CONTEXT. SUBST changed one occurrence at the beginning, but the code line we just wrote takes the but-first of the CONTEXT without checking further.

Instead of BF :CONTEXT itself, what we really want is the result of a continued substitution of NEW for KEY in that BF :CONTEXT. So the action really is

```
OP SE :NEW SUBST :KEY :NEW BF :CONTEXT
```

and the logic of that line is

```
IF ( FIRST :CONTEXT ) = :KEY OP SE :NEW SUBST :KEY :NEW  
BF :CONTEXT
```

If there is no substitution to make, of course, SUBST will keep the first element, but it still must check further in the sentence for later occurrences of the key word. The action in this case is nearly identical to the previous action except that the first element of the list is not changed to NEW but kept as is:

```
OP SE FIRST :CONTEXT SUBST :KEY :NEW BF :CONTEXT
```

Here is the entire procedure:

```
TO SUBST :KEY :NEW :CONTEXT  
  IF :CONTEXT = [] OP []  
  IF ( FIRST :CONTEXT ) = :KEY OP SE :NEW SUBST :KEY  
    :NEW BF :CONTEXT  
  OP SE FIRST :CONTEXT SUBST :KEY :NEW BF :CONTEXT  
END
```

And here are some examples of its use.

SUBST "VERB "LOVES [PAUL VERB CINDY]

SUBST "VERB PICK ACTIONS [THE TURTLE VERB DALE]

SUBST "NAME "CHRIS [NAME KISSED NAME]

SUBST "NAME PICK PEOPLE [NAME WON'T SPEAK TO  
NAME]

SUBST "ADV PICK [STEALTHILY CREATIVELY [WITH  
EXCEPTIONAL SPEED] HUNGRILY] [CATS CAN CLIMB  
TREES ADV BECAUSE OF THEIR SHARP CLAWS]



Although the procedure does everything it is advertised to do, it is not quite right for Mad-Libs. The problem is that in a command like PR SUBST "NAME PICK PEOPLE [NAME WON'T SPEAK TO NAME], both NAMEs are replaced by the same PICK from PEOPLE.

Why? Because the picking is done first. SUBST is presented with one name, selected at random by PICK, to use everywhere it finds the key word.

SUBST is useful as it is (because sometimes it is necessary to specify a particular replacement to make) but for Mad-Libs, it would be better to have a procedure that looked for a key word and each time it found one, selected at random from a list of potential substitutes.

Such a procedure would need inputs giving the key word and context as before, but instead of having a designated substitute, it should be given a list of alternates from which to pick each time the need arises.

TO MAD :KEY :ALT :CONTEXT

The stop rule would be the same.

```
IF :CONTEXT = [] OUTPUT []
```

And if there's no substitution to make, the action is the same.

```
OP SE FIRST :CONTEXT MAD :KEY :ALT BF :CONTEXT
```

Only when a KEY is found must MAD behave differently from SUBST. Compare the corresponding lines.

```
IF ( FIRST :CONTEXT ) = :KEY OP SE :NEW SUBST :KEY :NEW  
  BF :CONTEXT  
IF ( FIRST :CONTEXT ) = :KEY OP SE PICK :ALT MAD :KEY :ALT  
  BF :CONTEXT
```

SUBST is given a fixed substitute as input, whereas MAD picks the alternate itself whenever it needs to. Otherwise, they are identical.

Here is the finished procedure:

```
TO MAD :KEY :ALT :CONTEXT  
  IF :CONTEXT = [] OUTPUT []  
  IF ( FIRST :CONTEXT ) = :KEY OP SE PICK :ALT MAD :KEY  
    :ALT BF :CONTEXT  
  OP SE FIRST :CONTEXT MAD :KEY :ALT BF :CONTEXT  
END
```

And here are some examples of its use.

```
MAD "NAME PEOPLE [NAME KISSED NAME]  
MAKE "ADVERBS [STEALTHILY CREATIVELY [WITH EXCEP-  
  TIONAL SPEED] HUNGRILY]
```

MAD "ADV :ADVERBS [DOGS DO NOT CLIMB TREES ADV OR  
ADV]

MAD "V ACTIONS [PAT V CHRIS, BUT DALE V DANA.]

More can be done with MAD. Since MAD creates and outputs an object (rather than just printing its finished product), that object can be processed further. Try this:

MAD "NAME PEOPLE [NAME V NAME]

The object it produced is something like [SANDY V THE TURTLE]. If this object were made the input to MAD, the V could be replaced with some action. This can be done in one step.

MAD "V ACTIONS MAD "NAME PEOPLE [NAME V NAME]

The output from MAD "NAME PEOPLE [NAME V NAME] becomes the third input to MAD "V ACTIONS

\_\_\_\_\_.

Try

MAD "ADV :ADVERBS MAD "X PEOPLE MAD "V ACTIONS [X  
V AND V X ADV AND ADV]

### *Projects with Mad-Libs*

33. Create a MADLIB procedure that takes one input, a text, and looks for Verbs, Nouns, Proper Names, ADVerbs, and ADJectives to substitute. You might use [THE ADJ N V MY ADJ N PN ADV] as a test text.

34. Punctuation in a sentence will interfere with MAD the way it is now written. For example, MAD “V ACTIONS [PAT V CHRIS, BUT DALE V DANA.] will work, but MAD “PN PEOPLE [PN LOVES PN, BUT PN CAN’T STAND PN.] will not. (You may want to try it to see it fail.)

The substitution must be more sophisticated to handle punctuated sentences. It must look at each word in the sentence and perform tests to determine if it is a key word. Then, after choosing an alternative, the procedure must affix the proper punctuation to that new word.

Write a version of MAD that works correctly in both of the contexts shown above.

35. As MADLIB is now written, it finds substitutes for a fixed set of key words. A slightly more versatile program would take two inputs, the context (as always) and a list of key words to look for. Then it would systematically look through the context for instances of each of the key words and make the proper substitutions. Write a procedure that will do this.

*Understanding Language: Searching for Key Words and Matching Sentences to Templates.*  
*ALLOF, ANYOF*

FRIENDLY, when we last left it, expected a nice, tame answer to its question, “WHAT’S YOUR NAME?” It would respond unintelligently if you typed MY NAME IS PAUL or NONE OF YOUR BUSINESS when it asked. Here is a scenario that looks more intelligent, as if FRIENDLY really understands language.

FRIENDLY  
*WHAT'S YOUR NAME?*  
MY NAME IS PAUL.  
*HI, PAUL! IT'S NICE TO MEET YOU.*

FRIENDLY  
*WHO ARE YOU?*  
NONE OF YOUR BUSINESS  
*YOU SEEM TO BE IN A BAD MOOD. BYE.*

FRIENDLY  
*WHAT'S YOUR NAME?*  
WHAT'S IT TO YOU?  
*I WAS JUST CURIOUS.*

FRIENDLY  
*MY NAME IS LOGO. WHAT'S YOURS?*  
PLEASE CALL ME PAUL.  
*GOOD TO SEE YOU AGAIN, PAUL.*

FRIENDLY  
*HI, WHAT'S YOUR NAME?*  
PAUL  
*AH, YOU'RE BACK. HI, PAUL.*

FRIENDLY is exhibiting a number of behaviors we regard as intelligent. It is not confused by punctuation. Also, its phrasing is flexible. But, most important, it has always responded appropriately.

How can we design it so that it will reliably recognize the name in an arbitrary phrase? We might start by trying to figure out how people do that.

Do we listen to all the words in the sentence and look up each one on a list of possible names? Unlikely. If a Martian said to you, “Hi, my name is Xqpsnpfltk,” you might not be able to repeat the name, but you’d know you were being told one.



You’d know because you understood the rest of the sentence and realized that whatever that sound was that came at the end, that had to be this creature’s name.

All is not hopeless. Although we cannot expect to write a procedure that is capable of understanding all of English, we can analyze the likely language that this particular conversation will contain.

If the procedure encounters something we have anticipated, it can give a specific appropriate answer. Otherwise, it will have to give a neutral answer.

Here’s how it might work. First we list some possible phrases it may see. One limitation we will impose is that people always respond only with their first name, and not with first and last, or title and last, etc. (That complication comes later.)

Cooperative responses might include:

<name>

My name is <name>

People call me <name>

Please call me <name>

<name> is my name

I am <name>

Uncooperative responses should include:

None of your business!  
I won't tell you.  
I don't want to tell  
I'm not telling you.  
What's it to you?  
Go away

Let's work with the cooperative responses first. Suppose we create a series of templates based on likely response patterns. If we had a procedure that could match what the person types to each of the templates, and, where it found a match, record what word corresponded to the "wild card" <name>, that would be a big help.

For example, suppose we had a procedure MATCH? which would tell if a sentence matched a template. For example, the actual sentence

**MATCH? [MY NAME IS PAUL]**

used with the template

**[MY NAME IS @NAME]**

with the wild card identified by the at-sign, would give the result TRUE.

Suppose, furthermore, that if the sentence and template do match, then the matching word in the sentence (in this case, PAUL) and the name of the wild card it corresponded to (in this case, there is only one, @NAME) are saved in a special variable named @@MATCHES. Then, after this match,

@@MATCHES would have the value [[@NAME PAUL]].

Let's also suppose we have a way of looking for a wild card in this list and outputting the word associated with it; thus LOOKUP "@NAME would output PAUL. If we had such procedures, then we could write a language understander that looked like this.

```
TO OUTPUT.NAME :SENT
  IF MATCH? :SENT [MY NAME IS @NAME] OP LOOKUP "@NAME
    :@@MATCHES
  IF MATCH? :SENT [@NAME IS MY NAME] OP LOOKUP "@NAME
    :@@MATCHES
  IF MATCH? :SENT [I AM @NAME] OP LOOKUP "@NAME
    :@@MATCHES
  IF MATCH? :SENT [@JUNK CALL ME @NAME] OP LOOKUP
    "@NAME :@@MATCHES
  IF 1 = COUNT :SENT OP FIRST :SENT
    OP [I WAS JUST CURIOUS]
END
```

The first three lines explain themselves. If the sentence typed by the person to FRIENDLY is of any of those forms, a match will occur, and LOOKUP will find the name.

The fourth line has two wild cards in it. It takes care of both PLEASE CALL ME PAUL and PEOPLE CALL ME PAUL.

The fifth line assumes that if the person answers with only a single word, that word is probably the name. And the sixth line is a "punt." If no other strategy worked, this answers "neutrally" with a nothing answer.

There are some problems with this procedure as it was written. The most striking is that it can supply either the right answer (a name) which must then be tucked into some reply by GREET (depending, for example, on whether GREET has met the person before or not) or an entire reply which should not be further altered.

GREET, of course, can tell the two situations apart, as the name is a word, and the full reply is a list.

Second, we have not dealt at all with the “uncooperative responses.” More on those later. Meanwhile, how do MATCH? and LOOKUP work?

MATCH? will need two inputs — the sentence in question, and the template to check it against.

**TO MATCH? :SENTENCE :TEMPLATE**

It will need to make sure that the variable @@MATCHES is cleaned out before checking to see if the sentence matches the template.

**MAKE @@MATCHES []**

Finally, it performs the check.

**OP CHECK :SENTENCE :TEMPLATE**

So the procedure looks like this:

```
TO MATCH? :SENTENCE :TEMPLATE
  MAKE " @@MATCHES []
  OP CHECK :SENTENCE :TEMPLATE
END
```

But, we've put off the major part of the work! How does CHECK check?! It, too, must take both the sentence and template as inputs.

TO CHECK :S :T

If these two do match, it should output TRUE. If they don't, it should output FALSE. (This is not, of course, all it does. It must also identify what element of the sentence corresponded to the "wild card" in the template, but we will worry about that later.) A trivial case of matching is when both the sentence and the template are empty.

IF ALLOF :S = [] :T = [] OP "TRUE

If they are not both empty, but one of them is empty, then they surely do not match.

IF ANYOF :S = [] :T = [] OP "FALSE

If the first element of the sentence and the first element of the template are the same, then a match is possible, but not definite. In this case, the answer is to be found in checking the remaining elements of the sentence and the template for a match.

IF (FIRST :S) = FIRST :T OP CHECK BF :S BF :T

Likewise, if the first element of the template is a wild card, then a match is possible, but not definite. Again, the answer is to be found in checking the remaining elements of the sentence and the template for a match.

In this case, however, the procedure must do one additional thing. It must record what the first element of the sentence was when it encountered the wild card as the first element of the template.

```
IF WILD? FIRST :T (RECORD FIRST :T FIRST :S) OP CHECK BF
:S BF :T
```

Notice that both WILD? and RECORD are just tossed in there as if we already knew how they should work. We don't, and Logo has no such primitives to help us with, but we can design those procedures later.

At present, all we are trying to do is handle the top level logic of CHECK. Surely, if WILD? and RECORD existed, this line would be what we want.

Finally, if the first of T is neither wild nor matches the first of S, then there is no match, so we output FALSE.

This is how the procedure looks so far.

```
TO CHECK :S :T
  IF ALLOF :S = [] :T = [] OP "TRUE
  IF ANYOF :S = [] :T = [] OP "FALSE
  IF (FIRST :S) = FIRST :T OP CHECK BF :S BF :T
  IF WILD? FIRST :T (RECORD FIRST :T FIRST :S) OP CHECK BF
  :S BF :T
  OP "FALSE
END
```

What WILD? does depends on how we choose to indicate a wild card. Since we have decided that wild cards begin with the at-sign character, WILD? need only check for that character as the first character of its input.

```
TO WILD? :WORD
  OP "@" = FIRST :WORD
END
```

RECORD creates a list of the key and the matched word, and tucks that list into @@MATCHES to be retrieved when needed by LOOKUP.

```
TO RECORD :KEY :MATCHEDWORD
  MAKE "@@MATCHES LPUT LIST :KEY :MATCHEDWORD
  :@@MATCHES
END
```

And LOOKUP will look systematically through each element of @@MATCHES until it finds one whose first element is the key word. It will then output the second element. Notice how similar its structure is to the model recursive procedures you have seen before.

```
TO LOOKUP :KEY :LIST
  IF :LIST = [] OP ""
  IF :KEY = FIRST FIRST :LIST OP LAST FIRST :LIST
  OP LOOKUP :KEY BF :LIST
END
```

Now try running OUTPUT.NAME a few times.

```
OUTPUT.NAME [MY NAME IS ASHER]
OUTPUT.NAME [PLEASE CALL ME ISHMAEL]
OUTPUT.NAME [WHAT'S IT TO YOU?]
OUTPUT.NAME [PAUL]
```

### *Projects with Language Understanding*

36. Add OUTPUT.NAME to the FRIENDLY program. FRIENDLY must still be capable of responding differently to old people and new people, and must have the added ability to pull names out of the contexts in which they are typed. Do not yet worry about other details (e.g. punctuation) yet.

37. To add a bit more sensitivity to the uncooperative responses, you might design a procedure that looks for “negative words” in the sentence, words like WON’T, NONE, DON’T, NOT, and the like, and outputs a “neutral” response to a negative.

Such a response might be YOU SEEM TO BE IN A BAD MOOD, or SORRY I ASKED. Add that to OUTPUT.NAME in such a way that no other changes need to be made to GREET or FRIENDLY.

38. Wherever fixed phrases are now used, teach the program to vary them using PICK and a phrase list. If it is necessary to embed the name in a phrase, SUBST can do the work.

39. Finally, fix the program not to get stumped by punctuation.

40. As CHECK is currently written, [MY NAME IS @NAME] would match [MY NAME IS ASHER] but would not match [MY NAME IS ASHER LEV] because only one word can match a wild card.

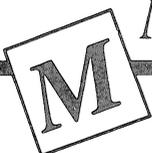
Likewise, [@JUNK CALL ME @NAME] would not match [CALL ME ISHMAEL], because some word must be present to match @JUNK.

A better CHECK program would recognize two kinds of wild cards, one that matches to exactly one word in the sentence (the situation we already have), and another that matches to any number of words in the sentence, including 0.

The new wild card would have to be symbolized differently, perhaps by a number-sign prefix. So CHECK would be able to tell that both CALL ME ISHMAEL and MY CLOSE FRIENDS CALL ME SIR OLIVER match the template [#JUNK CALL ME #NAME], but that [MY NAME IS HARRIET BEECHER STOWE] fits another template.

The image shows the front cover of a red book. The cover is decorated with a repeating pattern of white musical notes, including eighth notes, quarter notes, and sixteenth notes, scattered across the surface. On the left side, there are three circular punch holes. In the center, there is a white rectangular box with a thin black border. Inside this box, the word "MUSIC" is written in a large, bold, red serif font. The letter "M" is significantly larger than the other letters, and the word "USIC" is positioned to its right, sharing the same baseline. The overall design is clean and thematic, clearly indicating the book's subject matter.

**MUSIC**



# MUSIC

---

The Music chapter of this tutorial assumes that you are familiar with your Apple keyboard and some Logo primitives (commands Logo already knows). For explanations of commands and concepts not explained in detail here, see the Graphics chapter. In addition, the Logo Command Glossary lists all Logo primitives and their uses.

Some background: a Logo procedure is a series of instructions to the computer stored for recurrent use. A procedure can be used in other procedures just as if it were a Logo primitive.

Procedures are stored in files on disks. The SAVE command stores the entire contents of the workspace to the disk as a file with the name which you give it. See the Saving Procedures section in the Graphics chapter. Care should be taken to SAVE work before turning the computer off, as this clears the workspace.

## *Preparation: READ*

The Utilities disk contains the procedures required to make music in Logo, as well as the demonstration procedures we shall use. Start Logo as described in the Beginning in Logo chapter. Then insert your copy of the Utilities disk. Type

READ "MUSIC <RETURN> (only one quote)

(Logo does not hear what you type until you press the <RETURN> key.)

Wait for the light on the disk drive to go off and the question mark prompt to appear on the screen.

There are three music files on the Utilities disk. You have just read one file, MUSIC, into the workspace. MUSIC automatically reads in another of the music files called MUSIC.BIN, which contains the machine-language routine which does the actual work of making the Apple produce a sound. The third file is a demonstration file called TWINKLE, which will be used later.

To start, we will experiment with pitch and duration, concepts with which most people are familiar.

### *Duration*



Duration is usually thought of as how long a note is held. (You could also think of it as the time from the beginning of one note to the beginning of the next.) To start, let's use a procedure named TONE which plays a note for a specified duration. Type

```
TONE 40  
TONE 80  
TONE 80 TONE 80  
TONE 80 TONE 40 TONE 80
```

Typing TONE over and over gets tiresome. To save yourself work, try using the REPEAT command. For instance:

```
REPEAT 10 [TONE 40]  
REPEAT 10 [TONE 20]  
REPEAT 10 [TONE 20 TONE 40]
```



There is an even better way: the TONES procedure. TONES takes a list of durations to play. (A list for Logo is any list of items enclosed in square brackets.) Type

```
TONES [40 40 40 40]
TONES [80 80 80 80]
TONES [40 80 40 80 40 80]
```

You have probably noticed that the numbers for durations are related in a special way. For instance, a duration of 80 is twice as long as one of 40 and similarly, 60 is twice as long as a duration of 30. Try typing

```
TONES [80 80 80 80 40 40 40 40]
TONES [30 30 30 30 60 60]
TONES [40 40 20 20 40]
TONES [80 80 40 40 80]
```

Notice that the last two examples sound the same except that the last one goes slower. Another way of saying this is that the relative durations of the last two examples are the same.

If you know the symbols used in music notation, you will see that they relate to each other in a manner similar to that of the numbers we have been using. If 40 were used as a whole note, then 20 would be a half note, 10 a quarter note and 5 an eighth note.

Of course, we could have used another number instead of 40 as our whole note. If we made 60 a whole note, then 30 would be a half note, and 15 a quarter note. Using 40 as the whole note instead of 60 will speed up the tempo.

You don't have to change the durations to be able to change the tempo (the speed at which the piece is played). The procedure TEMPO allows you to do this. (The default value for TEMPO is 100.) Speeding up or slowing down the tempo will make the durations sound shorter or longer, but each duration will still have the same relation to the other durations as it did before. Try

```
TONES [80 80 40 40 80]
TEMPO 50
TONES [80 80 40 40 80]
TEMPO 100
TONES [80 80 40 40 80]
```

You can use TONES to create any rhythm you want. For example, try

```
TONES [80 40 40 80 40 40 80]
```

You can use <CTRL>P to bring back copies of the previous line so that it will play several times in a row. However, let's use REPEAT instead.

```
REPEAT 3 [TONES [80 40 40 80 40 40 80]]
REPEAT 3 [TONES [60 15 15 60 15 15 60]]
```

Notice how the beats seem to regroup when you REPEAT these patterns several times instead of playing them just once.

You can vary the duration within a wide range. Numbers higher than 1000 can be used but a duration of 1000 is very long. You can also use decimal numbers as well as whole numbers for durations.

## *Pitch*

Now let's try varying the pitch while keeping the durations constant. To do this we use a different procedure named `PLAY`, which takes as input a list of pitches and a list of durations. The first duration is paired with the first pitch, the second duration with the second pitch and so on. Type

```
PLAY [1 2 3 4 5 6][40 40 40 40 40 40]
```



(Notice that there is an extra duration in the second list. Logo ignores extra durations in input lists, but it will give an error if there are more pitches than durations.)

Each pitch is a half step higher than the one before. This is called the chromatic scale. An octave is divided into twelve pitches, each a half step apart. The numbers below correspond to the more common musical notation as follows.

1	2	3	4	5	6	7	8	9	10	11	12	13
C	C#	D	D#	E	F	F#	G	G#	A	A#	B	C

You should notice that the letters that are not followed by a # sign (read sharp sign in music notation) correspond to the white keys on the piano. The numbers that are paired with these letters (1, 3, 5, 6, 8, 10, 12, and 13) make the C major scale. You can make a major scale starting with any number using the same relations among the numbers as in the C major scale.

C Major Scale	1	3	5	6	8	10	12	13
D Major Scale	3	5	7	8	10	12	14	15

An interesting project for more advanced users is to make a procedure that will generate a major scale starting on any pitch. The major scale is one of the diatonic scales. Try typing

```
PLAY [1 3 5 6 8 10 12 13][30 30 30 30 30 30 30 30 30]
PLAY [3 5 7 8 10 12 14 15][30 30 30 30 30 30 30 30 30]
```

Notes are numbered from 1 to 24. However, adding a plus sign (+) to a number will make the note one octave higher. Similarly, adding a minus sign makes the note an octave lower.

For instance, 1+ is an octave higher than 1. (Do you see that 1+ is the same as 13?)

Try your own combinations of pitches and durations with PLAY. The pitches can only be whole numbers, but you can use any number, even a decimal or fraction, for durations.



There is also a procedure named SING which takes only a list of pitches and plays each with a constant duration. If you are experimenting with just pitches, this procedure will save you a lot of typing. The two PLAY commands above could easily have been done instead as

```
SING [1 3 5 6 8 10 12 13]
SING [3 5 7 8 10 12 14 15]
```

Rests are another feature which you can use with PLAY or SING. Type in the letter R in place of one of the

itches, and no sound will be played for the duration corresponding to that pitch. Type

```
PLAY [1+ 1+ R 1+ R 1+] [40 40 40 40 40 40]
SING [1+ 1+ R 1+ R 1+]
```

## ***Procedures***

Typing everything out each time can become tiresome, even if you do make use of <CTRL>P and the REPEAT command. Writing procedures will allow you to keep a record of the tunes you create.

For instance, to make a comparison of the chromatic and major scales easier, you might want to create the two following procedures so you can play them again easily in any order you want.

```
TO C
PLAY [1 2 3 4 5 6 7 8 9 10 11 12 13]
    [30 30 30 30 30 30 30 30 30 30 30 30 30]
END
```

```
TO M
PLAY [1 3 5 6 8 10 12 13]
    [30 30 30 30 30 30 30 30]
END
```



Here is a neat way of turning a PLAY statement into a procedure.

- a. Find a tune you like using PLAY.
- b. Press <CTRL>P to print it out again, but do not press <RETURN>.

- c. Use <CTRL>A to move the cursor to the beginning of the line.
- d. Type the name of your procedure, e.g. TO TUNE1, followed by a space. The rest of the line will move to the right as you insert new words.
- e. Press <RETURN>. You will be in EDIT mode with everything you typed on the first line (the title line).
- f. Move the cursor to the space between the title and the word PLAY.
- g. Press <RETURN>. The part of the line to the right of the cursor will move down to the next line.
- h. Press <CTRL>C to define your procedure.
- i. Type the procedure name (e.g. TUNE1) to play it.



Following is an example of a procedure that will give you quick feedback, which is useful if you are trying to pick out a tune.

```
TO QUICK  
PLAY (LIST RC) [15]  
QUICK  
END
```

Type QUICK to start the procedure and <CTRL>G when you want to stop it. Now typing any single digit number (except 0) will play a note.

It is useful to make procedures that will play phrases or pieces of a song. We will call these tune blocks. Then you can link these tune block procedures together to make the whole song, like putting together the blocks in a jigsaw puzzle. As an example, type

```
TO BELL1  
PLAY [5+ 1+ 3+ 8][80 80 80 80]  
END
```

Now type BELL1 if you haven't already. The tune is part of Westminster Chimes. So far we only have the first part of it and the rhythm doesn't seem quite right. Try increasing the duration of the last note to see if that sounds better. Instead of trying to add the entire tune into the procedure BELL1, you can break it up into blocks and write a superprocedure which uses them. This also allows you to use any of the blocks over again. The superprocedure could look like this:

```
TO BELL
BELL1
BELL2
BELL3
BELL2
END
```

So far we have BELL1. The following procedure makes the third block in the tune.

```
TO BELL3
PLAY [5+ 3+ 1+ 8] [80 80 80 160]
END
```

Notice that the only change is that two of the pitches are reversed, but even a small change makes an important difference. We leave BELL2 to you to create. (Hint: Try rearranging these same pitches in another way, keeping the durations in the same order.)

For a similar example, read in the file TWINKLE by typing

```
READ "TWINKLE
```

You have probably already guessed what tune this file will play. If you haven't, or even if you have, type STAR. Type PO STAR to see what the superprocedure looks like. Each of the subprocedures for STAR is a tune block.

```
TO STAR
STAR1
STAR2
STAR3
STAR3
STAR1
STAR2
END
```

The STAR superprocedure is designed in the same way as the BELL superprocedure. If you print out STAR1, STAR2, and STAR3, you can see that each uses the procedure RHYTHM1 to specify its duration list. Notice what happens if you change the rhythm

```
from [40 40 40 40 40 40 120]
to   [60 20 60 20 60 20 120]
```

You can still recognize the original tune but this makes a varied version of it. It works because each pair of 40's is changed to 60 and 20, and both 40+40 and 60+20 add up to 80. Notice that it sounds like a waltz now instead of a march. Try reversing the 60 and the 20 so it is 20 60 20 60 20 60 120. It sounds strange, right, almost like a new tune? This is because the durations make the pitches group together in a different way.

- ▷ Try playing STAR1, 2 and 3 in various orders to see if you can make a new tune. Don't forget the possibility of repeating the same block twice. Here is one example of a different tune.

STAR3 STAR2 STAR1 STAR1 STAR3 STAR2

You already know what the durations are for STAR1. Now, see if you can figure out what the pitches are without looking. The only pitches you will need are 1, 3, 5, 6, 8, and 10, pitches in the C Major Scale. To experiment, use the QUICK procedure shown earlier.

Try creating tune blocks for other tunes that you know. Instead of numbering the blocks in the proper order, pick a random order and see if your friends can figure out how the blocks fit together. Most familiar tunes use only the pitches of a major scale.

*Analyses of the Utilities Disk Music  
Procedures: STOP, FIRST, BUTFIRST (BF),  
THING, WORD, Top Level*

PLAY: a recursive procedure to play a list of notes with a list of durations.

```
TO PLAY :PITCHES :DURS
IF EMPTY? :PITCHES STOP
PLAY.NOTE ( FIRST :PITCHES ) ( FIRST :DURS )
PLAY (BF :PITCHES ) ( BF :DURS )
END
```

Line 1: title, including the local variables :PITCHES and :DURS, which represent values input when the procedure is run.

Line 2: IF-THEN statement without the optional THEN. Line 2 says IF it is true that there are no more pitches to be played (i.e. the list :PITCHES is empty, THEN STOP running this procedure and return control to whatever called it, which might be another procedure or the user (also called “top level”).

Line 3: Run the procedure PLAY.NOTE, using for inputs the first elements of the lists :PITCHES and :DURS. The local variables in the title line of PLAY.NOTE, :PERIOD and :DURATION, are given the values FIRST :PITCHES and FIRST :DURS, respectively.

Line 4: Run PLAY again, using the rest of the list :PITCHES and the rest of the list :DURS as the two inputs.

BF is short for BUTFIRST. BF :PITCHES is the list :PITCHES without its first element. USING BF recursively, as it is used here, enables one to work through a list element by element. The procedure will stop in Line 2 when the list is exhausted.

Use of the list operators FIRST and BUTFIRST is explained in the Words and Lists chapter.

**TONE, TONES, SING:** All these procedures are special versions of **PLAY**. Note that **TONE** and **SING** use the primitive **SENTENCE (SE)** to put information into a list before passing it to **PLAY**, which requires lists for its inputs.

For a full discussion of **SENTENCE**, see the **Words and Lists** chapter.

**PLAY.NOTE:** a procedure to play one note.

```
TO PLAY.NOTE :PERIOD :DURATION
MAKE "PERIOD THING WORD "# :PERIOD
.CALL2 :TONE :PERIOD :DURATION*:BASE.PERIOD
/:PERIOD
END
```

Line 1: The title, including the local variables **:PERIOD** and **:DURATION**.

Line 2: Line 2 pastes a **#** onto the front end of the **:PERIOD** brought into the procedure. The Logo primitive **WORD** makes one word out of **#** and whatever came in as **:PERIOD**; for instance, **#** and **5** make **#5**.

The **THING** of a variable is the value associated with that name. **THING "PERIOD** is the same as **:PERIOD**. **THING** is used when there is no actual variable name to put the dots on.

**MAKE** gives this value to the global variable **:PERIOD**.

Line 3: Runs the procedure .CALL.2 using the following for inputs:

For :ADDR—the value of the global variable :TONE  
(defined by SETUP)

For :INPUT1—the value of the local variable :PERIOD

For :INPUT2—a value obtained by the calculation  
shown



# **P** PRINTING

---

This chapter contains information on printing text and graphics using a variety of printers and plotters.



**NOTE:** The following examples assume that your printer is attached to slot 1 of your computer. If your printer is connected to a different slot, use that number rather than 1 in these examples. If you are using an Apple IIGS computer and your printer is connected to the Printer Port, set the Slot 1 option to PRINTER PORT at the Control Panel.

## *Printing Text*

The process of printing text is quite straightforward, regardless of the type of printer you have.

The command `OUTDEV` allows you to turn on and off any slot inside the computer. A printer is generally connected to slot 1 or a printer port, which looks to the computer like slot 1. Typing `OUTDEV 1` activates the printer; typing `OUTDEV 0` deactivates it.

While the printer is activated, any Logo instruction that you enter will send output to the printer.

To print a catalog listing of your disk, for example, you could type

```
OUTDEV 1
CATALOG
OUTDEV 0
```

To obtain a paper printout of a procedure called `CIRCLE`, you could type



OUTDEV 1  
PRINTOUT CIRCLE  
OUTDEV 0

You can print out procedures, names of procedures, names of variables or the entire contents of your workspace (both procedures and variable names) using the following instructions.

PO TITLES (POTS) lists the titles of all procedures.  
PO PROCEDURES lists all procedures, but no names.  
PO NAMES lists just variable names and their values.  
PO ALL lists all procedures and names.

### *Printing Pictures*

Use the printing utilities described in this chapter to print a Logo picture immediately after you create it, or first save the picture on a disk and print it later. The SAVEPICT command will store the graphics screen image to disk. An extension of .PICT is added to the filename you give a picture in order for Logo (and you) to differentiate it from other files.

Printed pictures will have slightly different proportions from the screens that produce them. That is because the aspect ratio (squareness of the dots that make up the image) of a printer is different from that of a video monitor or television. The standard value for .ASPECT is .8. For better hardcopy results, you might want to determine the proper aspect ratio to use with your printer. For example, type .ASPECT .9 before drawing your screen picture (using .ASPECT after drawing your picture has no effect). Note that this results in elongated screen graphics that come out properly on the printer. Frequently you can compromise by setting the

aspect ratio to be between that of the monitor and that of the printer, with negligible bad effects.

### *Printing Logo Graphics Using Other Programs*



When you save a Logo design using the SAVEPICT command, it is stored on the disk as a standard picture file. There are a number of commercially available programs that can read Logo pictures and print them using a variety of printers. In many cases, you can enhance your Logo picture using the graphics tools that these programs provide.



*Terrapin Logo for the Apple* saves its pictures using the DOS 3.3 operating system, so you may be able to load them directly into other DOS 3.3 programs. *Logo PLUS* saves its pictures using ProDOS, so you will need to read the pictures it saves into ProDOS-based packages. However, since you can convert picture files from one operating system to the other with the ProDOS User's Disk, you have the flexibility to use many different programs.

Contact Terrapin, Inc. for information regarding the use of Logo graphics with other software packages. Technical Tip memos are available that describe how to use Logo graphics with The Print Shop™ program, published by Broderbund, and the Newsroom™ program, published by Springboard.

### *Using an ImageWriter Printer*

Before you begin, check to see that the DIP switches on the ImageWriter printer are set to their standard default values.

### Printer:

SW1: Switch 8 open

SW2: Switches 1-2 closed, switches 3-4 open

If you are using a Super Serial Card, make sure that these are the settings of the DIP switches located there.

### Super Serial Card:

SW1: OFF OFF OFF ON OFF ON ON

SW2: ON OFF OFF ON ON OFF OFF

### *The PRINTSCREEN Command*

*(Logo PLUS only)*

If you are using *Logo PLUS*, you can print pictures directly from Logo using the built-in PRINTSCREEN command. (This command is not available in *Terrapin Logo for the Apple*. Use the PRINTPICT and LARGEPICT programs described below instead.)

Before typing the command, make sure that the printer is on, set to the online or select position and that paper is ready.

PRINTSCREEN takes two numbers as inputs. The first number indicates the slot to which your printer is attached to your computer. (Apple //c owners should always specify 1.) The second number indicates the size of the picture. An input of 1 gives a standard size picture; an input of 2 prints a double-size picture.

For example, if your printer is connected to slot 1 inside your computer and you want to print a double-size picture, you would type

```
PRINTSCREEN 1 2
```

If you want to print an inverse or color picture, or if you are using a printer other than an ImageWriter, you must use the printing programs from the Utilities Disk, described in the rest of this chapter.

*PRINTPICT and LARGEPICT:  
Printing Black-and-White Graphics*

PRINTPICT uses the file DUMP.OBJ and prints a small-size picture (3.5 x 2.5 inches).

LARGEPICT uses the file LARGE.OBJ and prints a picture that is twice as large in each direction (about 7 x 5 inches).

Read in the file you want (either PRINTPICT or LARGEPICT). The appropriate .OBJ file will be loaded automatically. If you are using 64K of memory, you will want to read in this file before you begin working on a picture. Unless you first save it, your picture will be destroyed if you read this file after creating your design.



Now you can print the graphics screen whenever you want. It makes no difference whether the pictures are drawn from scratch or read in from disk.

To print the graphics screen, type PRINTPICT followed by two inputs: the slot number for your printer and a print option number.

Slot number - Most printers are connected to slot 1 inside the computer. Apple IIc users should always specify slot 1. Apple IIGS users should specify slot 1 if the printer port is being used. Set the SLOT 1 option to PRINTER PORT at the Control Panel. If a printer card

is used and is in slot 1, set the SLOT 1 option to YOUR CARD instead.

Option number - With a print option value of 0, white or colored dots will be printed. With a print option value of 1, the picture's negative will be printed.

A typical command for regular printing with an Image-Writer connected in slot 1 is:

```
PRINTPICT 1 0
```



---

Note: If you have read in the LARGEPICT file, you can type either the PRINTPICT or LARGEPICT command to print a double-sized picture; the result is identical, no matter which instruction you use. The reason for having two ways of issuing the same command is to accommodate both those who are used to typing PRINTPICT to print a picture and those who want to type a procedure name that reflects the name of the file they are using. In the PRINTPICT file, the only available command is PRINTPICT, which prints a single-sized picture.

---



#### A Few Hints:

For most drawings, the inverse option will wear out the printer ribbon more quickly.

Small-size printing is 3 to 4 times faster than double-size and much easier on the printer ribbon. (Also, the resulting hardcopy occupies less space on the refrigerator door.)



**Beware!** If you are running Logo with only 64K of memory, reading a Logo file (such as a picture-printing file) erases the graphics screen. Unless you have a 128K machine, read PRINTPICT or LARGEPICT before creating or reading in a picture.

***SMALLCOLOR and LARGECOLOR:  
Printing Color Graphics***

SMALLCOLOR uses the file IRIS.BIN and prints a small-size picture (3.5 x 2.5 inches).

LARGECOLOR uses the file CHROMA.BIN and prints a picture twice as large in each direction (about 7 x 5 inches).

These files permit you to print full color Logo graphics. You will need an ImageWriter II printer with a color ribbon and an Apple computer with 128K. It makes no difference whether the printer file is loaded before or after the screen picture is drawn.

Each printer file contains a procedure with the same name as the file. To print the graphics screen, first load the file you want, then type either SMALLCOLOR or LARGECOLOR followed by two inputs: the slot number for your printer and a print option number.

Slot number - See the discussion of slots in the previous section concerning black-and-white printing.

Option number - A print option value of 0 will cause white or colored dots on the screen to be printed (white dots appearing as black). A print option value of 1 will cause all but the white dots on the screen to be printed.

A typical command for regular printing on an ImageWriter II in slot 1 is:

`SMALLCOLOR 1 0` or `LARGECOLOR 1 0`

The `SMALLCOLOR` and `LARGECOLOR` files need to be read in only one time (unless, of course, the procedures are erased with a `GOODBYE` or similar command). After you have read in the file you want, you can print pictures at any time using the appropriate `SMALLCOLOR` or `LARGECOLOR` command.



These printing routines are not recommended for use with a black ribbon. Satisfactory black-and-white results may be achieved on the ImageWriter II using the `PRINTPICT` utilities described above. Likewise, pictures printed using a black ribbon (whether or not they contain colored dots on the screen) will be produced much more quickly using `PRINTPICT`.

Read the hints in the preceding section for additional helpful information.

### *Other ImageWriter Printing Tips*

*A letter R printed with graphics:*

If you are using an Apple IIc, you may find that a letter R is printed at the bottom of the picture. To prevent this, add a semi-colon to the beginning of the line

`(PRINT1 CHAR 9 CHAR 82 CHAR 13)`

at the end of the `PRINTPICT`, `SMALLCOLOR`, or `LARGECOLOR` procedure.

With the semi-colon added, the line should look like this:

```
; (PRINT1 CHAR 9 CHAR 82 CHAR 13)
```



The semi-colon is a signal to Logo to ignore the rest of the line. You could also delete the line, but commenting it out with a semi-colon allows you to get the line back again by just removing the semi-colon mark. Do not change this line if you are using a machine other than the Apple IIc.

*Long lines of text that are overprinted:*

If your Logo procedures contain very long instruction lines (over 80 characters), the ImageWriter will print the second line on top of the first. In order to print lines that are legible, you must enter a special command to prepare the printer. (To cancel the commands given below, simply turn the printer off and then on again.)

If you are using a Super Serial Card, enter the following instructions at the Logo ? prompt sign:

```
OUTDEV 1
(PRINT1 CHAR 9 "C CHAR 13)
OUTDEV 0
```

If you are connecting your ImageWriter to an Apple IIGS using the Printer Port, enter the following instructions at the Logo ? prompt sign:

```
OUTDEV 1
(PRINT1 CHAR 27 CHAR 68 CHAR 32 CHAR 0
  CHAR 13)
OUTDEV 0
```

Change the value of "Add LF after CR" to NO at the Printer Port option of the Control Panel to prevent short lines in your procedures from being double-spaced.

*SCRIBE: Using a Scribe Printer*

The PRINTPICT files described above will work for the Scribe as well as the ImageWriter. To begin, make sure that all DIP switches on the Scribe are set to the UP position. This is the factory default setting.

To print using the Scribe's black ribbon, read in either of the printer files PRINTPICT or LARGEPICT. Follow the instructions above to print the graphics screen using the PRINTPICT procedure.

For a single-color picture using the Scribe's color ribbon, you can use the procedures RED, YELLOW, and BLUE in the file named SCRIBE. First read the file by typing READ "SCRIBE, then run the desired procedure just before printing the picture. A typical command to print a red picture with a Scribe printer in slot 1 would be:

```
RED PRINTPICT 1 0
```

If no color is specified, a yellow picture will be printed. Color commands will be ignored when a black ribbon is installed. Do not use the ImageWriter color printing files SMALLCOLOR and LARGECOLOR. They were not designed for the Scribe printer.



---

### *Printing to a Parallel Printer*

To print graphics using a parallel printer, you need a graphics interface card. The SCREENDUMP program (see below) supports the Grappler™ interface card. If you have a different interface card, refer to your interface card manual, which will provide instructions for printing graphics. Then modify the SCREENDUMP program with the appropriate control codes for your particular setup (type HELP after loading SCREENDUMP for assistance).

#### *SCREENDUMP : Using a Grappler™ Interface Card*

The procedure SCREENDUMP in this file can be used with a Grappler card (from Orange Micro, Inc.) to print Logo screen pictures. To use it, read in the file, draw (or read in) the desired picture, and type SCREENDUMP. (Remember that reading in a file erases the graphics screen if you are using only 64K of memory. So, unless you are reading in a picture that you have previously saved, make sure that you load the SCREENDUMP file before drawing your picture.)

After reading in the SCREENDUMP program, you may want to type

HELP

for complete instructions as well as help in modifying the SCREENDUMP procedure for use with other interface cards.

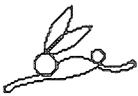
When printing Logo graphics with this card, there are several optional codes that can be included in the printer command to alter the printout. The basic graphics

printing command for use with a Grappler card connected to slot 1 is:

```
OUTDEV 1
(PRINT1 CHAR 9 "G CHAR 13)
OUTDEV 0
```

The letter G in this command stands for graphics. Other optional codes include:

"GD to print a double-size picture  
"GR to rotate the picture 90°, placing it sideways on the screen  
"GI to print the inverse or negative of the picture



The optional code letters D, R, and I can be added to the "G code in any order. For example, a command to print a double-size, rotated graphics design would be:

```
OUTDEV 1
(PRINT1 CHAR 9 "GRD CHAR 13)
OUTDEV 0
```

Those with Epson printers must include the letter R whenever a double-size command is given in order for the picture to be centered on the page.

### *SWEET-P: Plotter Procedures*

The procedures in this file make it easy to create Logo graphics on the Sweet-P Personal Plotter (from Enter Computer).

To start, read in the file SWEET-P and type

```
SETUP
```

The names of most procedures resemble their Logo screen graphics equivalents; for example, PFD moves the plotter pen forward, PPU does a plotter PENUP, and so on. For a listing of commands, type

**HELP**

The procedure POLYSPI is given as an example of a plotter superprocedure. Its only difference from procedures you've already seen is that it uses PFD and PRT instead of FD and RT.

The Sweet-P has the ability to print text as well as graphics. The following procedures make it easy to vary the size and direction of text printing.

**PRINTTEXT** Takes an input (either a word or list) and prints it on the plotter.

**TEXTSIZE** Controls the magnification of text size. The input range is 1 to 255, although values much larger than 90 will create characters too big for a regular sheet of paper. Default value is 1.

**TEXTORIENT** Sets the direction for text printing. The possible inputs are 0, 90, 180, and 270. Default is 0.

The procedure CHANGEPEN is provided so you can tell a procedure to pause for manual pencolor change.

You can also vary the magnification of graphics figures. To do so, change the global variable EXPANSION by typing, for example

## MAKE "EXPANSION 7.6

A value of 7.6 will give the plotter page the same turtle-step dimensions as the graphics screen. (This means that the default value, 3, allows more turtle steps in all directions than is possible on the monitor.)

### ***RSPLOTTER: Procedures for the Radio Shack Plotter***

The procedures in this file allow you to control the Radio Shack four-color plotter (model CGP-115) from Logo. (Although this plotter is no longer produced, Terrapin provides this file for current owners of the product.)

As with the SWEET-P procedures, most procedure names resemble their Logo screen graphics equivalents. A few differences should be noted, however. You do not have to change pens manually on the Radio Shack plotter. Instead, the procedure PPC is provided.

In addition, the procedure NEWPAGE allows you to feed clean paper from the roll into the plotter.

Finally, the Radio Shack plotter can print text as well as graphics, but must first be put into the appropriate text or graphics mode. The procedures TXTMODE and GRMODE permit you to switch back and forth with ease.

*U* TILITIES

### *About the Utilities Programs*

The Utilities Disk contains many sample programs, useful tools and demonstrations. Using these programs, you will be able to make Logo music, introduce young children to Logo in an easy way, play games written in Logo, print to a variety of printers and plotters, or explore assembly language programming with Logo.

Before you begin, make a copy of your Utilities Disk, because it is possible to damage it or erase it accidentally. Put the original away in a safe place. Refer to the first chapter, *Beginning in Logo*, for complete instructions for making a backup copy.

All procedures in the files on the Utilities Disk may be considered to be examples or learning models. You are encouraged to analyze their ideas and use their constructions. Note particularly the brevity of Logo procedures (small programs), the constant use of subprocedures, and the use of procedure names that describe the procedure explicitly.

In addition to serving as models, the procedures in the files on the Utilities Disk fill a variety of roles.

#### *The Logo PLUS Utilities Disk*

The *Logo PLUS* Utilities Disk contains a somewhat different set of programs from those described here. A few of the programs are not present on the *Logo PLUS* Utilities Disk because they are not needed; their functions are identical to commands that are built into *Logo PLUS*. For example, the built-in shape editor of *Logo PLUS* replaces the need for the shape edit utility program.

Additional programs have been added to the *Logo PLUS Utilities Disk* because they relate to or demonstrate the enhanced features of the *Logo PLUS* language. These added files contain character fonts, sample shapes files and interesting project ideas. Files that are not present on the *Logo PLUS Utilities Disk* are marked here with a +, and additional files are described in the separate *Getting Acquainted with Logo PLUS* booklet.

### *To Use the Utilities Disk Files*

1. Start Logo using the Language Disk, then remove the Language Disk from the disk drive and put it away.
2. Insert your backup copy of the Utilities Disk into the disk drive.
3. To list the files on the disk, type CATALOG. You will see the first part of the listing; press the space bar to see the remainder. If you type CATALOG while Logo turtle graphics are on the screen, you will only see four lines of text at the bottom of the screen. Press the space bar to complete the listing, if necessary, then press <CTRL> T to see the full text screen. Return to the graphics screen using <CTRL> S.
4. To read a file from the disk, after the ? prompt, type  
READ "(filename without .LOGO extension)

Example:

```
?READ "TEACH      (only one quote, please)
```

Logo will read the file, confirming the presence of each procedure as it reads it in by printing its name and the word DEFINED. Example:

```
?READ "TEACH
TEACH1 DEFINED
TEACH DEFINED
?
```

Some programs are self-starting; that is, they begin running by themselves once you read the file. Other programs do not automatically begin and require you to type the name of the initial procedure. The initial procedure for most of the demonstration programs on the Utilities disk is the same as the name of the file. For utility programs that need to be initialized, the procedure is usually called SETUP. If Logo prints the ? prompt when it finishes reading the file, you must start the program yourself. The descriptions of the programs below explain how to start each one.

You may want to place one or more of the Utility programs on your own data disk in order to incorporate them into programs you have written. Several of the programs, such as the ImageWriter printing programs and the MUSIC program, require some machine language code to run and these programs automatically load the correct binary file from the disk. This binary file must be on the same disk as the Logo program that uses it. Use the system disk that came with your computer to copy a binary file to another disk. The summary below indicates the Logo programs that use binary files. (Keep in mind that these utility programs are copyrighted and may not be given away.)

## *Summary of Utilities Disk Files*

- + The programs marked with a + sign are not contained on the Utilities Disk for *Logo PLUS*. Refer to the *Getting Acquainted with Logo PLUS* booklet for complete information about *Logo PLUS* Utilities programs.

### Printing Utilities:

Refer to the Printing chapter for complete instructions for printing text and graphics using Logo.

- |            |                                                                                                                                                       |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------|
| PRINTPICT  | Utility for printing small-sized graphics using the ImageWriter printer. (Uses the DUMP.OBJ file.)                                                    |
| LARGEPICT  | Utility for printing large-sized graphics using the ImageWriter printer. (Uses the LARGE.OBJ file.)                                                   |
| SMALLCOLOR | Utility for printing small-sized graphics in color using the ImageWriter II printer. (Uses the IRIS.BIN file.)                                        |
| LARGECOLOR | Utility for printing large-sized graphics in color using the ImageWriter II printer. (Uses the CHROMA.BIN file.)                                      |
| SCRIBE     | Utility for printing color pictures on the Scribe printer. See also the PRINTPICT and LARGEPICT utilities, which can be used with the Scribe printer. |

- SCREENDUMP** Utility for printing graphics to parallel printers using the Grappler™ and other interface cards.
- SWEET-P** Procedures for controlling the Sweet-P Personal Plotter.
- RSPLOTTER** Procedures for controlling the Radio Shack 4-Color Plotter.

Aids to Using Logo:

- INSTANT** System of single letter Logo commands which makes Logo graphics available to non-readers, among others. INSTANT and its use are also described in the Graphics chapter.
- TMOVE** Procedure for moving the turtle using arrow keys.
- TEACH** System of writing Logo procedures without using the editor.

Music System Files:

- MUSIC** Procedures for creating music using notes of specified pitch and duration. See the Music chapter for a complete description and tutorial. (Uses the MUSIC.BIN file.)
- MUSIC.SRC** Assembler language and Logo procedure MCODES.
- TWINKLE** Procedures to play “Twinkle, Twinkle, Little Star.”

The music system is an example of Logo/assembly language interfacing, explained in the Technical section of this tutorial.

Shape Editing Programs:

Instructions for creating your own turtle shape are described later in this chapter. See the Technical chapter for a detailed discussion of this program.

Owners of *Logo PLUS* should refer to the *Getting Acquainted with Logo PLUS* booklet for instructions on using the built-in shape editor in that version.

**SHAPE.EDIT**      System for changing the shape of the  
+                      graphics turtle, a useful feature for  
                            games and animation.

**ROCKET, ROCKET.AUX, ROCKET.SHAPES,**  
+                      A demonstration of how to use a turtle  
                            with a different shape. The shape was  
                            created using the **SHAPE.EDIT**  
                            utilities program.

Demonstration Programs:

**ANIMAL**              Game that adds your information  
                            about animals to its knowledge base.  
                            (An interesting discussion of the  
                            structure and procedures of this  
                            program can be found in the section  
                            on advanced use of lists in Logo for  
the Apple II, by Harold Abelson,  
                            professor of mathematics at M.I.T.)

**ANIMAL.INSPECTOR**

Procedures for examining the ANIMAL knowledge base.

**DYNATRACK** Game using principles of physics to simulate a ride around a frictionless race track.

**INSPI.PICT** Example of a design created with Logo. The procedure that created this picture is listed later in this chapter.

**TET** Example of a simple recursive procedure which draws a complex design.

**Useful Tools:**

**ARCS** Collection of procedures for drawing arcs with variable radii. Another procedure for drawing an arc is developed in the Procedures section of the Appendix.

**BEFORE** Procedures for determining which of two words comes first alphabetically.

**CURSOR** Collection of procedures for outputting the cursor's position.

**PICK** Randomly selects a letter from a word or a word from a list.

**PPRINT** Procedures to wrap words in long lines of text around the screen rather than be split in the middle.

**STRING** Procedures for manipulating segments of words.

**WAIT +** Procedure for causing a timed delay.

**File Handling Utilities:**

**BIN.TO.TEXT +** File that will allow you to convert your *Terrapin Logo for the Apple* files from binary files into text files for use with word processors and modems.

**FID** File utility program for deleting, renaming, locking, and unlocking files.

**TEXTEDIT +** Procedures for using the Logo system as a text editor.

**DPRINT +** Collection of procedures for printing text to a disk file.

**Logo Files for Logo/Assembler Interfacing:**

**ADDRESSES** File of names describing addresses in the Logo interpreter for the assembler.

**AMODES** File of names describing the 6502 addressing modes.

**ASSEMBLER** Logo assembler procedures.

**OPCODES** File of names describing the 6502 mnemonics for the assembler.

The Technical chapter describes Logo/Assembler interfacing in detail.

## *Explanation of Utilities Disk Files*

Many of the following program descriptions include sample procedures that you can try. Please note that if the second line of an instruction is indented, you should not press <RETURN> at the end of the first line.

### *PRINTING UTILITIES*

Please refer to the Printing chapter for instructions on using the following programs that are provided on the Utilities disk. Note that *Logo PLUS* includes a built-in command to print graphics to an ImageWriter printer.

### *AIDS TO USING LOGO*

#### *INSTANT: Single Letter Logo Commands*

This collection of procedures makes the Logo system easy to use even for very young children. After you READ "INSTANT, you can use single-character commands to manipulate the turtle and define procedures. Each keystroke is acted upon immediately. Typing F, for example, makes the turtle move forward a small amount, leaving a trail. R makes it turn to the right. Repeating a sequence of F's and R's will draw a square.

The INSTANT system allows you to store the commands you have typed as a procedure. When you type N, it will define a procedure to draw the picture currently on the screen. For example, if you draw a square using R and F, and name the result SQUARE (using the N command), the INSTANT program will define a procedure SQUARE with FORWARD and

RIGHT commands in it. If you press P, the program will ask you the name of a procedure to run (picture to show), and run that procedure. Here is a table of INSTANT commands:

?	Help
D	Clears the screen
F	Go forward
L	Turn left
N	Names a new picture
P	Asks for the name of a picture to show
R	Turn right
U	Undo the last command

If you need to start the program again from the Logo ? prompt sign, type INSTANT, the name of the main procedure for this program.

The INSTANT program is an example of how easy it is to create “languages” with simple Logo programs. It also serves as an example of Logo programming style, and of the use of RUN and DEFINE. You can easily modify INSTANT to provide more complex commands.

For a complete discussion of this program, see the INSTANT section of the Graphics chapter in this tutorial.

### *TMOVE: Another Way to Move the Turtle*

The program TMOVE allows the user to move the turtle using the four arrow keys. TMOVE is similar to INSTANT in that it looks for a single keystroke and causes an action based on the key that is pressed. Unlike INSTANT, however, it does not keep track of the keys

that have been pressed, nor does the turtle draw when it moves, although it could be modified to do so.

The six keys that can be used are the four arrow keys, the H key to return the turtle to its Home position, and the <ESC> key to end the program.

The distance (number of screen dots) that the turtle moves with each keystroke is currently set to five by the instruction MAKE "DISTANCE 5 in TMOVE. This number can be changed to cause the turtle to move a larger or smaller distance.

TMOVE may help younger children and those with special needs to more easily position the turtle on the screen before drawing a design.

### ***TEACH: How to Write Logo Procedures Without Using the Editor***

TEACH is used to define procedures whenever you want to avoid the complexities introduced by using the editor. It has the additional advantage of prompting the user for information. The instructions developed in the immediate mode are copied into a procedure using TEACH.

To use this program, type READ "TEACH to load it and type TEACH to begin.

Consider this procedure, which will draw a variable-sized square.

```
TO SQUARE :SIZE  
  REPEAT 4 [FD :SIZE RT 90]  
END
```

The following dialog shows how to define the same procedure using TEACH. Type what appears in the computer font, as you are prompted. What TEACH prints is in italics. If there are no variable inputs to the procedure, press <RETURN> at the second prompt.

?TEACH	press <RETURN>
<i>NAME OF PROCEDURE</i> > SQUARE	press <RETURN>
<i>INPUTS (IF ANY)?</i> :SIZE	press <RETURN>
>REPEAT 4 [FD :SIZE RT 90]	press <RETURN>
>END	press <RETURN>
<i>SQUARE DEFINED</i>	
?	

To run SQUARE, type

SQUARE 50

The graphics screen is not cleared when TEACH is used, as it sometimes is when the editor is used. (The 64K version of Logo clears the graphics screen when returning from the editor; the 128K version leaves it intact.) In graphics mode, TEXTSCREEN (<CTRL> T) will show the previous typing, which might be hidden by the picture. SPLITSCREEN (<CTRL> S) will return the picture and four lines of text.

Remember that reading a file from the disk clears the screen if your computer has only 64K of memory. Before beginning to type any instructions that you might want to copy into a procedure using TEACH, you may want to read TEACH in from the disk.

The TEACH system uses the two procedures described below—TEACH and TEACH1.

TEACH asks for and receives the name of the procedure

and any inputs, then passes the information on to TEACH1.

TEACH1, a recursive procedure, receives the lines of the procedure (after the prompt >), testing each for END. When END is received, TEACH1 completes the defining of the procedure, and passes control back to TEACH, which announces the procedure defined.

### *MUSIC SYSTEM FILES*

#### *MUSIC:*

##### *How to Write and Run Logo Music Procedures*

To run the music programs, type READ "MUSIC. For a short demonstration, type FRERE to hear the "Frere Jacques" melody. For more information about using the music-making tools, see the Music chapter of this tutorial. *Logo PLUS* contains its own primitive NOTE for playing a single note. However, *Logo PLUS* users can use these music tools as well.

#### *TWINKLE:*

##### *A Sample Melody*

To use this program, first read in the file MUSIC, as described above. Then read the TWINKLE file and type STAR to hear the song.

#### *MUSIC.SRC, MUSIC.BIN:*

##### *An Example of Logo/Assembler Interfacing*

MUSIC.SRC is a Logo file, which contains the assembly language program for playing notes. MUSIC.BIN is a binary file that is automatically loaded by the MUSIC

file. MUSIC.BIN must be present on any disk that uses the MUSIC tools. If you copy MUSIC.LOGO to another disk to use with Logo programs you have written, you must also copy the MUSIC.BIN file using your computer's system disk. The Technical chapter contains more detailed information on the Logo music system.

### *SHAPE EDITING PROGRAMS*

#### *SHAPE.EDIT: How to Change the Shape of the Turtle (Terrapin Logo only)*

Although Logo is not designed for animation, it is possible to change to shape of the turtle in order to simulate animation effects. Ordinarily, the best screen motion that you can obtain is by moving the turtle. If you want to make a circle move across the screen, it will be very slow to repeatedly draw and erase the circle, moving the position of the circle little by little. However, if you change the shape of the turtle itself, so that the turtle looks like a circle, then you can make the circle move across the screen by simply moving the turtle.

The following instructions will help you use the shape editor that is on the *Terrapin Logo for the Apple Utilities Disk*. A more technical discussion of shape editing can be found in the Technical chapter.

Note that *Logo PLUS* includes a built-in shape editor with more features and capabilities.

#### *The Logo Shape Editor*

The turtle shape was designed by building a shape table.

Other shapes can be created by constructing other shape tables. However, constructing shape tables is a tedious process and requires some technical knowledge.

One of the programs contained on the Utilities Disk is a shape editor. This shape editor, written in Logo, allows you to design a shape by drawing it directly on the screen, and also includes functions for changing the currently displayed turtle shape and its size. The program then automatically assembles the shape into a shape table that you can use easily.

### *Creating a Shape*

When defining a new shape, always start with a fresh Logo workspace with no procedures defined.

To read in the shape editor, type

```
READ "SHAPE.EDIT
```

The SHAPE.EDIT file contains many procedures that allow you to create and use new shapes for the turtle. One of these procedures is called MAKESHAPE, which allows you to define or create new shapes.

To begin designing your own shape, enter a MAKESHAPE command, which takes one input—the name of the shape you will design. For example:

```
MAKESHAPE "BOX
```

When you press <RETURN> after this instruction, you will enter the shape editor, a blank graphics screen, with a message reminding you of the shape you are defining. In general, you will use the arrow keys to move around the screen, lighting dots to create a pattern for a new

turtle shape. Use the following commands (similar to those of the editor) to construct shapes.

- U            Penup
- D            Pendown
- arrow keys    Move in the direction of the arrow and draw a line if the pen is down.
- <CTRL> P    Move up and draw a vertical line if the pen is down. (Equivalent to up arrow.)
- <CTRL> N    Move down and draw a vertical line if the pen is down. (Equivalent to down arrow.)
- <CTRL> C    Exit the shape editor and define the shape. A message that says your shape is done will appear. Your turtle will be set to the new shape.
- <CTRL> G    Exit without permanently defining the shape. Use this command to cancel the definition of a shape if you wish to start over. (You can set the turtle to the shape as defined so far, but the next time you define a new shape, this one will be lost.)
- <DELETE>  
or <ESC>    Delete the previous few commands. Because the previous byte in the shape table is deleted, one, two or three dots may be erased.
- 1...9        Change the size of the shape. Typing 3 is equivalent to entering SIZE 3 outside the editor after the shape is defined. It may be helpful to switch between the size you want and a larger size, which is easier to see, while designing a shape.



When creating the new shape, be sure to pick up the pen before backing up over a line that has already been drawn, otherwise it will be erased. Since these lines are being drawn using pen color 6, lines that are traced over will be erased and lines will be drawn where none exist.

Once you've defined a turtle shape (BOX, in this example), the turtle will assume this shape when you exit the shape editor. You can use SETSHAPE to change the turtle back to its original shape or to any other shape you have defined. The SETSHAPE procedure takes one input (the shape name) and changes the turtle to that shape.

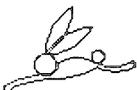
You can restore the turtle to its original triangular shape, by typing

```
SETSHAPE 0
```

Then change it back to its new shape by typing

```
SETSHAPE :BOX
```

At any time you can change the size in which shapes are shown by using the SIZE procedure. SIZE 1 is the standard size. Type SIZE followed by a number from 1 to 9 to change the size of the turtle shape.



Perhaps you want to set the turtle's shape at random from the set of available shapes. You can do this by first making a list of shape names, including zero for the turtle shape itself. For example, type

```
MAKE "SHAPES [0 ROCKET BOX]
```

You will have to create a special variable if you want to use the turtle shape 0 by typing

MAKE "0 0 (both these characters are zero)

Now enter these two procedures:

```
TO PICK.SHAPE
  SETSHAPE THING (WORD PICK :LIST)
END
```

```
TO PICK :LIST
  OUTPUT ITEM 1 + RANDOM COUNT :LIST :LIST
END
```

Now you can type PICK.SHAPE to set the turtle's shape at random.

Typing another MAKESHAPE command will cause a new shape to be defined. You cannot edit previously defined shapes. If you wish to erase all shapes and start over, type SETUP.

### *Stamping Shapes*

The SHAPE.EDIT file contains procedures for stamping the image of the current turtle shape onto the screen. The STAMP procedure gets the current shape from the SS procedure and stamps it on the screen at the current location.



When you use STAMP to stamp the image of the turtle, you will at first see neither the stamped turtle nor the currently active turtle. However, they are both actually there, but hidden; they cancel each other out while at the same position.

This unexpected event occurs because Logo uses pen color 6 (the reverse pen color) to draw and erase the turtle. Remember that when you give an instruction to

draw a line with pen color 6, the turtle will erase a line that is already there and draw a line if none is present. The same phenomenon is occurring here. The turtle shape (in pen color 6) is covering an identical turtle shape that has already been stamped on the screen. These images temporarily cancel each other out, and you see nothing on the screen. As soon as you give a command to move the turtle, however, you will see both the stamped turtle and the active turtle.

The following procedure, STAMPRANDOM, uses the STAMP procedure to stamp the current shape at random places on the screen.

```
TO STAMPRANDOM
  STAMP PU
  SETXY (120 - RANDOM 240)(110 - RANDOM 220)
  PD STAMPRANDOM
END
```

### *Saving Shapes*

To save on disk all the shapes you have defined, use the SAVESHAPES command, written as a Logo procedure. It takes the name of the file as input. Be sure to use a quote, just as you would using the SAVE command. For example, type

```
SAVESHAPES "BOX
```

The SAVESHAPES procedure creates two files on the disk. For example, SAVESHAPES "BOX will create BOX.SHAPES and BOX.AUX.LOGO. The file BOX.SHAPES contains the computer's internal shape table (the actual appearance of the shape). The file BOX.AUX.LOGO contains the names of the shapes and the following procedures: SETSHAPE, .SHAPE,

SIZE, INITSHAPES, plus any procedures you have defined. Unless you are writing assembler procedures for manipulating the shapes themselves, you probably don't want to include any extra procedures in this file.

The shapes file that ends with ".AUX" contains a procedure called INITSHAPES, which automatically loads the proper ".SHAPES" file that sets up the defined shapes. To use the shapes you saved in a file called BOX, therefore, you would need to enter these instructions:

```
READ "BOX.AUX
INITSHAPES
```

You could also write a procedure that takes any filename as input and loads and initializes the shapes to be ready for use. Such a procedure might look like this:

```
TO USE.SHAPES :FILENAME
  READ WORD :FILENAME ".AUX
  INITSHAPES
END
```

Running this procedure with an instruction like USE.SHAPES "BOX, will load the correct file and define the shapes that are stored in it.



For greater flexibility, you should keep the procedures you write to actually use the shapes in a separate file from the shapes themselves. This means that you should not define procedures to use shapes while you're still using the shape editor. Instead, after defining your shapes, you should save them and clear your workspace by typing GOODBYE.

If you accidentally do save your procedures in with the shapes you created, you can edit the resulting ".AUX" file and separate the procedures into two different

files—one containing SETSHAPE, .SHAPE, SIZE and INITSHAPES, and the other containing procedures you have written to use the shapes.

*A Sample Session*

```
?GOODBYE
?READ "SHAPE.EDIT
?MAKESHape "BLOCK
Define a shape here
?MAKESHape "TIRE
Define another shape here
?SAVESHAPES "BLOCKS
```

At this point, Logo will ask you to place your data disk in the disk drive. This disk should be the one on which you want to store the shapes files and the programs that will use them. After Logo saves the shapes, it will ask you to put the disk containing the shape editor back into the drive. You should then place the Utilities Disk (or a copy of it) into the drive and press <RETURN>. Logo will pause for a while as it reads the shape editor back into memory.

There should now be two new files on the files disk that you used: BLOCKS.AUX.LOGO and BLOCKS.SHAPES. Type GOODBYE to erase everything in the workspace so you can start fresh. To retrieve the shapes you created, place the disk on which you saved your shapes in the drive and type

```
READ "BLOCKS.AUX
INITSHAPES
SETSHAPE :BLOCK
```

You now have two options. The first may seem more convenient to you. The second, however, will give you more flexibility in the long run.

1. You could write procedures to use these shapes and then save everything. That way, all you have to do to read in (and use) the shapes is to type `INITSHAPES`. The disadvantage to this method is that it is more difficult to use these same shapes in a different program.
2. The alternative is to start with a fresh workspace by typing `GOODBYE`, write the procedures that use the shapes, and include the `READ` and `INITSHAPES` commands in the procedures so the shapes are read in when the program starts running. When testing the program, it will bring the shapes into the procedures workspace. Therefore, when you are satisfied, you can use `ERASE` to erase the following: `NAMES`, `SETSHAPE`, `.SHAPE`, and `INITSHAPES`. (Don't erase any variable names that are required by the procedures you have written.) Then save the file. This gives you the ability to use the same shapes more easily in different programs.

The following section briefly describes a program that uses the second technique of separating the files.

***ROCKET, ROCKET.AUX, ROCKET.SHAPES:  
Example of User-Defined Turtle Shapes  
(Terrapin Logo Only)***

On the *Terrapin Logo* Utilities Disk is a demonstration program called `ROCKET`. Type `READ "ROCKET` and then `ROCKET` to begin the demonstration. The moving rocket is the turtle, defined using `SHAPE.EDIT` as described in the section above.

The `ROCKET` procedure reads the `ROCKET.AUX` file and calls the procedure `INITSHAPES`. The

INITSHAPES procedure automatically sets up the shapes. After you run the procedure, type

```
SETSHAPE :ROCKET  
DRAW
```

and try moving the rocket-turtle around. (See the Graphics chapter.) Try changing its size using the procedure SIZE that is provided. For example, type

```
SIZE 5
```

The rocket will move, no matter how large it is (SIZE 1 is the normal size). However, edited turtle shapes will make only 90 degree turns, although the trail each leaves behind will go in the proper direction.

Type PO ROCKET to see how the demo works. To run it again without reloading the entire file, type SHOW.

### *DEMONSTRATION PROGRAMS*

#### *ANIMAL: A Game that Teaches the Computer About Animals*

This program attempts to augment its knowledge about the animal kingdom by playing a game in which it tries to guess the animal you are thinking of. It asks various questions, such as "Does it have wings?" You answer with "Yes" or "No." If it doesn't guess correctly, it will ask you for your animal's name and a question to distinguish that animal from the animal it guessed. This information it adds to its knowledge tree for the next game.

To play, type

```
READ "ANIMAL
```

and then type ANIMAL to begin.

When you are finished playing, stop the program using <CTRL> G. If you want to save what you have done, type SAVE "ANIMAL; the next time you play, it will know the animals you taught it. The animal game on the Utilities Disk already knows several animals. To make it start out fresh, after the Logo ? prompt sign type INITIALIZE.KNOWLEDGE. Then type ANIMAL to start the game from scratch.

The ANIMAL game is a good example of brief, single purpose procedures. The main procedure ANIMAL prints the greeting, then uses GUESS with the stored :KNOWLEDGE. After a round of the game, it prints another greeting, uses WAIT for a pause, then begins again by calling itself, ANIMAL.

#### *ANIMALINSPECTOR:*

*What's in the ANIMAL Knowledge Base?*

The procedures in this file are for examining the ANIMAL knowledge base. The procedure INSPECT.KNOWLEDGE, which prints out the ANIMAL program's "knowledge" about animals in an easily readable form, is intended as a learning aid to be used with the ANIMAL program described above. In its use of recursion, it is similar to tree-drawing programs, since it actually follows the tree of the ANIMAL program's knowledge as it prints it out. Look at the procedures in this file as an example of recursive programming.

The global variable :KNOWLEDGE in the file ANIMAL is the knowledge base examined. Therefore, it is necessary to read in the file ANIMAL to use the ANIMAL.INSPECTOR.

You can examine the ANIMAL knowledge base by typing

```
INSPECT.KNOWLEDGE
```

Here is a brief description of the ANIMAL.INSPECTOR procedures:

```
INSPECT.KNOWLEDGE
```

Uses INSPECT1 with the stored :KNOWLEDGE, beginning at level 0.

```
INSPECT1 :KNOWLEDGE :LEVEL
```

Calls itself and IPRINT to inspect and print each branch of the knowledge tree.

```
IPRINT
```

Creates a formatted display of the ANIMAL tree of knowledge. Type <CTRL> W (hold down the <CTRL> key and tap <W>) to stop the scrolling (to read the tree). Press any key to resume scrolling.

*DYNATRACK:*

*A Game: the Dynamic Turtle  
On a Frictionless Surface*

Steering without friction is a very different world, as people riding on rocket power have discovered. Dynatrack puts you on a rocket sled on a frictionless track and gives you the power to do two things:

1. You can turn the sled, BUT it will keep moving in the old direction, moving sideways.
2. You can give it a burst of rocket power. The force will be applied in the direction in which it is pointing, BUT, since it was already moving, the resultant direction will be somewhere between the original direction and the direction in which you are pointing.

This is one of the trickiest games you will meet. It requires strategy more than eye-hand coordination. The object is to move around the track without touching the lines.

Type READ "DYNATRACK and then follow the directions to play. The program begins automatically.

The dynamic turtle keeps moving when you give it a "kick." Type R to turn it right, L to turn it left, K to kick it in the direction it is facing. If the turtle is moving in another direction when you "kick" it, the direction of movement will be changed. Each time you "kick" it, it will accelerate and be harder to control.

The main procedure is called DYNATRACK. Type DYNATRACK to begin the program again when you see the Logo ? prompt sign.

### *INSPI.PICT: Sample Logo Picture*

To see the picture, type

```
READPICT "INSPI
```

This picture was generated by running the following procedure four times with the turtle pointing at

different angles, and with different pen colors:

```
TO INSPI :DISTANCE :ANGLE :INCREMENT
  FD :DISTANCE
  RT :ANGLE
  INSPI :DISTANCE :ANGLE + :INCREMENT
  :INCREMENT
END
```

Try using the instruction `INSPI 8 0 13` to recreate this picture.

*TET:*  
*A Graphics Procedure of Variable Complexity*

TET is a good example of a recursive procedure. It draws tetrahedra on the points of tetrahedra. The largest tetrahedron is of the size specified. On its points are drawn half-size tetrahedra, on their points are drawn quarter-size tetrahedra, and so on, to the depth specified. A depth of 1 draws only the one large tetrahedron. TET takes two inputs; the first sets the size and the second sets the depth (or number of levels of recursion). Try, for example, `TET 50 3`.

Spaces in the procedure listing below are to help isolate the individual commands. They are not a necessary (or usual) inclusion. The REPEAT statement must be typed as one line (without a <RETURN> in it).

```
TO TET :SIZE :DEPTH
  IF :DEPTH = 0 STOP
  REPEAT 3 [LEG :SIZE
    TET :SIZE * .5 :DEPTH - 1
    RT 150 FD :SIZE
    RT 150 LEG :SIZE RT 180]
END
```

```
TO LEG :SIZE
  FD :SIZE / ( 2 * COS 30)
END
```

### *USEFUL TOOLS*

#### *ARCS:*

#### *Variable Radii Arc and Circle Procedures*

To use the arc and circle procedures provided on the Utilities Disk, type the procedure name followed by the required number of inputs. Examples:

RARC 50 90 for a 90 degree (quarter circle) arc to the right with a radius of 50.

RARC1 1 90 for a 90 degree (quarter circle) arc to the right with a radius of  $360/(2 \text{ PI})$  or about 57.2.

RCIRCLE 30 for a circle to the right with a radius of 30.

Substitute LARC, LARC1, and LCIRCLE for arcs and circles to the left.

The Procedures section of the Appendix offers an additional variable radius arc procedure.

ARC Procedures:

#### **RARC :RADIUS :DEGREES**

Procedure which draws an arc to the right with given :RADIUS covering the number of :DEGREES indicated. Uses RARC1.

**LARC :RADIUS :DEGREES**

Procedure which draws an arc to the left with given :RADIUS covering the number of :DEGREES indicated. Uses LARC1.

**RCIRCLE :RADIUS**

Procedure which draws a circle to the right with given :RADIUS. Uses RARC.

**LCIRCLE :RADIUS**

Procedure which draws a circle to the left with given :RADIUS. Uses LARC.

**RARC1 :SIZE :DEGREES**

Procedure which draws an arc to the right with a radius equal to :SIZE x 360/(2 PI). Uses CORRECTARCR.

**LARC1 :SIZE :DEGREES**

Procedure which draws an arc to the left with a radius equal to :SIZE x 360/(2 PI). Uses CORRECTARCL.

**CORRECTARCR :SIZE :AMOUNT**

Procedure which makes a small correction with each step of RARC1.

**CORRECTARCL :SIZE :AMOUNT**

Procedure which makes a small correction with each step of LARC1.

The CORRECTARC procedures compensate for the error introduced by trying to make a fractional number of repetitions in the ARC1 procedures, in the line:

```
REPEAT QUOTIENT :DEGREES 5 [FD :SIZE * 5 RT 5]
```

### ***BEFORE: Comparing Words***

The BEFORE file contains a set of procedures that allow you to compare words and numbers. The main procedure, called BEFORE?, takes two words as input and returns TRUE if the first comes alphabetically before the second; otherwise, it returns FALSE.

Examples:

```
PRINT BEFORE? "ANTELOPE "ANTEATER  
FALSE
```

```
PRINT BEFORE? "DENMARK "FINLAND  
TRUE
```

```
PRINT BEFORE? 39 103  
TRUE
```

When numbers and words are both used in the same BEFORE? instruction, the number is always evaluated as coming first.

```
PRINT BEFORE? 1492 "COLUMBUS  
TRUE
```

This tool will help you write programs that alphabetize lists of words or sort numbers.

### ***CURSOR: Procedures for Character Output Control: Position, Flashing, Inverse***

These procedures are for controlling character output to the screen. In *Terrapin Logo for the Apple*, the CURSOR primitive is the only one provided for controlling the location of the text cursor on the screen. In *Logo PLUS*, CURSORPOS is a built-in command. This first set of procedures performs operations not directly supported in *Terrapin Logo*.

---

**Character Control Procedures:**

**CURSOR.HV** Outputs a list of two elements: the cursor's horizontal position and its vertical position.

```
PRINT CURSOR.HV  
0 19
```

*Logo PLUS* equivalent:  
PRINT CURSORPOS

**CURSOR.H** Outputs the cursor's horizontal position.

```
PRINT CURSOR.H  
0
```

*Logo PLUS* equivalent:  
PRINT FIRST CURSORPOS

**CURSOR.V** Outputs the cursor's vertical position.

```
PRINT CURSOR.V  
23
```

*Logo PLUS* equivalent:  
PRINT LAST CURSORPOS

The following procedures are useful in both *Terrapin Logo for the Apple* and *Logo PLUS*.

**FLASHING** All characters printed after this command will flash alternately black on white, white on black. This includes what Logo prints as well as everything typed at the keyboard. Use **NORMAL** to restore to white on black. To enter this mode type:

```
FLASHING
```

**INVERSE** All characters printed after this command (by Logo or from the keyboard) will appear in inverse video, black on white. To enter this mode, type:

**INVERSE**

**NORMAL** Restores the normal mode of white on black. Type:

**NORMAL**

Everything printed after this will be white on black.

*PICK: Choosing Things at Random*

The PICK tool allows you to pick a letter at random from a word, or a word at random from a list of any length. Using PICK, you can choose an individual at random from a list, combine words to make amusing sentences of varying grammatical structures or explore probability.

Although the Logo procedure for this purpose is short, it may be beyond a beginning programmer's understanding; thus it is provided on the disk as a tool.

```
TO PICK :OBJ  
  OUTPUT ITEM 1 + RANDOM COUNT :OBJ :OBJ  
END
```

To use PICK, read the file PICK from the Utilities Disk. Here are some examples of ways to use PICK.

Printing the result of flipping a coin:

```
PRINT PICK [HEADS TAILS]
```

Tossing a die:

```
PRINT PICK [1 2 3 4 5 6]
```

Picking a person at random from a list:

```
PRINT PICK [ABE BONNIE CARLOS DIANA]
```

Creating subject-predicate sentences:

```
MAKE "SUBJECT [GOLDFISH COMPUTERS [ICE  
    CREAM CONES] LLAMAS]
```

```
MAKE "PREDICATE [BOUNCE SWIM FLY WIGGLE  
    PRETEND MELT]
```

```
PRINT SENTENCE PICK :SUBJECT PICK :PREDICATE
```

```
ICE CREAM CONES WIGGLE    (one possible example)
```

### *PPRINT: Word Wrap for Long Lines*

Normally, when Logo displays lines of text that are longer than 40 characters, the words at the ends of the lines may be split in the middle. A tool can be written in Logo that causes words to wrap neatly around the edge of the screen and not be divided. The PPRINT tool checks the position of the next word in a list and places it on the next line if it won't fit on the current line. This is sometimes referred to as "pretty-printing."

To use the PPRINT tool, read the PPRINT file. Use PPRINT (or its PPR abbreviation) in the same way that you would use PRINT (or PR). The difference is that the words that are in the list will wrap around the screen. Long lines are harder to edit than shorter ones, but it is advantageous to have the PPRINT tool available, especially when you are getting input of unknown length from the user.

The following examples show the difference between PRINT and PPRINT. (Also included is a procedure for the abbreviation PPR, equivalent to PPRINT.)

```
PRINT [THE QUICK BROWN FOX JUMPS OVER A  
LAZY DOG.]  
THE QUICK BROWN FOX JUMPS OVER A LAZY DO  
G.
```

```
PPRINT [THE QUICK BROWN FOX JUMPS OVER A  
LAZY DOG.]  
THE QUICK BROWN FOX JUMPS OVER A LAZY  
DOG.
```

### *STRING: Manipulating Word Segments*

The file **STRING** contains a tool for isolating a set of letters within a word. The procedure **STRING** takes three inputs: two numbers and a word, which could be a number. **STRING** returns to you a set of letters between the numbers representing the position of the letters in the word. For example:

```
PRINT STRING 4 6 "ANTEATER  
EAT
```

The letters **EAT** are the fourth through sixth letters of the word "ANTEATER.

Other examples:

```
PRINT STRING 3 4 1928  
28
```

The following example shows how a date format can be changed using **STRING** as a tool. If the variable name "DATE has been assigned the value of 08-22-88 using

the following instruction:

```
MAKE "DATE "08-22-88
```

Then the following instructions can be used. (The computer will display the text in italics.)

```
MAKE "MONTH ITEM (STRING 1 2 :DATE) [January  
February March April May June July August  
September October November December]
```

```
PRINT :MONTH
```

*August*

```
MAKE "DAY STRING 4 5 :DATE
```

```
PRINT :DAY
```

*22*

```
MAKE "YEAR WORD 19 STRING 7 8 :DATE
```

```
PRINT :YEAR
```

*1988*

```
PRINT (SENTENCE :MONTH WORD :DAY ", :YEAR)
```

*August 22, 1988*

### *WAIT: Causing Delays in your Programs (Terrapin Logo Only)*

The WAIT tool that is provided on the *Terrapin Logo Utilities Disk* is a convenient way to place a timed delay into a program. (Since WAIT is a built-in command in *Logo PLUS*, this file is not included on the *Logo PLUS Utilities Disk*.)

WAIT takes one input, a positive integer representing tenths of a second. Therefore, the instruction WAIT 10 pauses a program for 1 second; the instruction WAIT 60 causes a delay of 6 seconds, and so on.

Note that you cannot stop a wait that is in progress using <CTRL> G.

## *FILE HANDLING UTILITIES*

### *BIN.TO.TEXT: Converting Binary Files to Text Files (Terrapin Logo only)*

Files containing Logo procedures are saved in a binary format for greater speed in reading and saving them. However, there may be times when you would prefer your Logo programs to be in a standard text format. As text files, they can be read into a word processing program or sent to another computer through a modem. The BIN.TO.TEXT program makes this conversion for you.

Note that since *Logo PLUS* already saves its programs as text files, this file is not provided on the *Logo PLUS* Utilities Disk.

The BIN.TO.TEXT program is not written in Logo. Therefore, you must save the program you wish to convert before leaving Logo. To use it, place the *Terrapin Logo* Utilities Disk in the disk drive and restart the computer. At the ] prompt sign, type

```
RUN BIN.TO.TEXT
```

You will see a menu with options to convert your file or exit. Press 1 and <RETURN> to convert a file.

You will be asked for the slot and drive number of the disk containing the Logo program and the name of the program you want to convert. Do not include the .LOGO extension that Logo adds to filenames.

Then you will enter the slot and drive number of the disk on which you want to place the new file. (You can

place the new file on the same data disk, if you would like.) Then you will be asked to type a name for the text file. You can use a new name or the same name you typed for the Logo file, since the .LOGO extension will not be added to the new filename.

Make sure the disks are in place and press <RETURN>. You can cancel the process at this time by pressing <ESC>. When the process is complete, you will return to the opening menu.

To read your newly created text file into a word processor or to use it with a modem, you must either use software programs that are based on the DOS 3.3 operating system, or you must convert the file to ProDOS using the ProDOS User's Disk before reading it into a ProDOS-based software program. See your computer's documentation for instructions.

*FID: File Management Utility: How to Delete, Rename, Lock, and Unlock Files, Set Default File Extension (Terrapin Logo only)*

FID is a file utility program written in Logo. It allows you to catalog the disk, rename files, and delete files, all with single-keystroke commands. (Since *Logo PLUS* contains built-in commands for these functions, this utility file is not necessary.)

Each file command asks for the name of a file, and appends to it the current "file extension." The "." command in FID allows you to change the file extension. Useful file extensions for Logo are "LOGO," "PICT," and "BIN."

After you read in the FID file, it will automatically begin. Use the character indicated by FID to list, delete, rename, lock, or unlock files on the disk, or to set the default file extension. Use the ? command to view the list of available commands. In the event of a disk error, restart the program by running the procedure FID. (See DOS in the Logo Command Glossary.)

***TEXTEDIT:***

***How to Save, Read, Examine, and Print  
Files Containing Text (Terrapin Logo only)***

These procedures allow you to use the Logo editor to read and save files of English text. The TEXTEDIT commands allow you to read text into the editor, save the contents of the editor to disk, print the contents of the editor, and so on. *Logo PLUS* contains built-in primitives that replace the need for this Utilities program.

To use Logo as a text editor, first read the file by typing

READ "TEXTEDIT

To start a text file, type

TO <RETURN>

This puts you into the editor, with the white line across the bottom of the screen.

Type the text you want, making use of the editing commands described in the Edit section of the Appendix.



**NOTE THE DIFFERENCE HERE:** When you have finished and are ready to print or save the text, leave the editor by typing

**<CTRL> G**

Do **NOT** type the **<CTRL> C** used to define procedures.

Use **SAVETEXT** (described below) immediately to save the file on the disk. You can always replace it with a corrected version, but if you accidentally erase it from your workspace before you save it, you must retype the text in its entirety.

Read the file back from the disk using **READTEXT** (also described below).

**WARNING:** To work on the file again after using **READTEXT**, type **EDIT** (or **ED**). If you type **TO** when there is already text in the editor, it will be erased. If you have not yet saved it, it will be lost completely. However, if it is on the disk, it only means reading it in again.

**READTEXT :FILE** Reads a Logo file into the editor. Type **ED** to enter the editor or display the contents of the editor on the screen.

```
READTEXT "WASHINGTON  
SHOWTEXT  
EDIT
```

**SAVETEXT :FILE** Saves the contents of the editor to the disk in the file named. To store text in the file **GEORGE**, type

SAVETEXT "GEORGE

**SHOWFILE :FILE** Reads the file named and prints it out on the screen. SHOWFILE is a combination of READTEXT and SHOWTEXT. Example:

SHOWFILE "GEORGE

**PRINTFILE :FILE** Reads the file into the editor and prints it to the printer. If your printer is not controlled from Slot 1, change the value of :PRINTER. Example: if your printer card is in Slot 7, type

MAKE "PRINTER 7

To print the contents of the file GEORGE to the printer, type

PRINTFILE "GEORGE

**PRINTTEXT** Prints to the printer the text currently in the editor. Uses SHOWTEXT. Example: to print the contents of the editor, type

PRINTTEXT

If your printer is not controlled from Slot 1, change the value of :PRINTER. (See PRINTFILE.)

**SHOWTEXT**  
(*a Logo primitive*)

Prints on the screen the text currently in the editor.

*DPRINT: Printing Text Into Disk Files  
(Terrapin Logo only)*

The procedures in DPRINT can be used to write text into disk files. TEXTEDIT (see previous section) contains procedures for saving, reading, examining, and printing the files. For most uses, the TEXTEDIT procedures are more appropriate and easier to use.

To create a new text file, type

```
OPEN  "(name of file)
DPRINT [what you want to type into the file]
CLOSE
```

Example:

```
OPEN "SESAME
DPRINT [THIS IS A TEST]
CLOSE
```

The text is stored with the CLOSE procedure. To see the file listed on the disk, type CATALOG. To see the contents of the file, use the procedure SHOWFILE in TEXTEDIT. Example:

```
READ "TEXTEDIT      (if these procedures are not in memory)
SHOWFILE "SESAME
```

```
THIS IS A TEST
```

To add to a file, use OPEN.FOR.APPEND instead of OPEN. Example:

```
OPEN.FOR.APPEND "SESAME
DPRINT [TESTING APPENDING]
CLOSE
```

SHOWFILE "SESAME

*THIS IS A TEST  
TESTING APPENDING*

**WARNING:** If you use OPEN with an existing file, the newly entered text will overwrite the text already in the file. Example:

OPEN "SESAME           (assuming the text above is in it)  
DPRINT [HI]  
CLOSE

SHOWFILE "SESAME

*HI  
IS A TEST  
TESTING APPENDING*

The following procedures are contained in the DPRINT file.

**OPEN :FILE**       Takes a file name as input. The input to DPRINT (see below) will be printed into the Logo file with this name.

**CLOSE**           Closes the open file. All output will be written to the file.

**DPRINT :ITEM**   Will cause the item to be printed into the file.

**OPEN.FOR.APPEND :FILE**  
Used instead of OPEN. Will cause everything printed with DPRINT to be appended to the existing file rather than writing over it.

### Additional notes:

The files created by these procedures can be printed and read into the editor using the procedures in the file `TEXTEDIT`. If you use `READ` on such a file, Logo will attempt to execute its contents as Logo commands.

You cannot read or save a file or use the editor or graphics while a file is open for `DPRINTing`. Also, you cannot print more than 8192 characters; any extra characters will be ignored.

### *LOGO FILES FOR ASSEMBLY LANGUAGE INTERFACING*

#### *ADDRESSES, AMODES, ASSEMBLER, OPCODES: Interfacing Logo and the Assembler*

**ASSEMBLER** The Logo assembler procedures.

**AMODES** The file of names describing the 6502 addressing modes.

**OPCODES** The file of names describing the 6502 mnemonics for the assembler.

**ADDRESSES** The file of names describing addresses in the Logo interpreter for the assembler.

Refer to the Technical chapter for detailed information about these programs.

### *Useful and Interesting Memory Locations*

There are a number of memory locations that even non-technical Logo users will find interesting. The following is a simplified and partial list of memory locations used by Logo. A complete listing can be found in the Technical chapter.

To use any of these variable names, first read the ADDRESSES file from the Utilities Disk.

The use of the Logo primitive .DEPOSIT changes the value stored at the memory location indicated by the variable. Placing a new value in a memory location will cause the Logo system to behave differently, as described below. The primitive .EXAMINE allows you to look at the contents of a location in memory.

**WARNING:** If values other than those listed below or other memory locations are used, there is no telling what behavior your computer will assume. If strange and unusual events occur, it is best to reboot your computer, which will reset the memory to its original state.

#### *Reading a File into the Editor*

**.DEPOSIT :SAVMOD 1**

Causes Logo not to define the procedures in a file that is read in. The text is placed in the editor. Type ED to enter the editor; <CTRL> C will cause the procedures to be defined as usual. Use SAVE to save the current contents of the edit buffer.

**.DEPOSIT :SAVMOD 0**

Cancels special READ and SAVE mode.

---

The TEXTEDIT procedures described earlier depend on the use of :SAVMOD.

*Displaying Outer Brackets Around Lists*

.DEPOSIT :BKTFLG 1

Causes Logo to retain outer brackets around lists, displaying them when variables are printed.

.DEPOSIT :BKTFLG 0

Returns to normal mode where outer brackets are not displayed.

*Disabling Special <CTRL> Characters*

.DEPOSIT :NOINTP 1

Disables Logo's special "interrupt" characters <CTRL> F, <CTRL> S, <CTRL> T, <CTRL> G, and <CTRL> W. These keystrokes can then be read using their ASCII codes (6, 19, 20, 7 and 23 respectively).

.DEPOSIT :NOINTP 0

Restores these keystrokes to their normal mode.

.DEPOSIT :NOINTP 255

Disables <CTRL> Z in addition to those disabled by a value of 1 (see above).

*Determining the Position of the Text Cursor*

.EXAMINE :CH

Returns the horizontal position of the text cursor.

.EXAMINE :CV

Returns the vertical position of the text cursor.

The following operation, for use with *Terrapin Logo for the Apple*, returns a list containing the horizontal and vertical position of the text cursor. (*Logo PLUS* contains a `CURSORPOS` primitive that performs this function.)

```
TO CURSORPOS
  OUTPUT LIST .EXAMINE :CH .EXAMINE :CV
END
```

*Inverse and Flashing Text*

- .DEPOSIT :INVFLG 0  
Sets text to inverse mode (black letters on white background).
- .DEPOSIT :INVFLG 64  
Sets text to flashing mode.
- .DEPOSIT :INVFLG 255  
Sets text to normal mode.

See also the `CURSOR` file on the Utilities Disk.

***T*** TECHNICAL

## *Use of the Logo System*

The Logo system includes a full interpreter for the Logo language, a complete text editor for editing procedure definitions, and an integrated “turtle graphics” system. This section provides notes on how these different functions interact.

### *Modes of Using the Screen*

The Logo system uses the display screen in three different ways, or “modes.”

#### *NODRAW Mode*

This is the mode in which the system starts. Logo prompts the user for a command with a question mark, followed by a blinking square called the “cursor.” You may type in command lines, terminated with RETURN. Logo executes the line and prints a response, if appropriate.

Whenever the cursor is visible and blinking, Logo is waiting for you to type something, and will do nothing else until you do.

The system includes a flexible line editor, which allows you to correct any typing errors in a command line that you have typed in draw or nodraw mode. The available keystrokes, described on page T-6, correspond to the <ESC> and <DELETE> keys, arrow keys, <CTRL> A, <CTRL> D, <CTRL> E, <CTRL> G, <CTRL> X, <CTRL> P.

#### *EDIT Mode*

Using the commands TO or EDIT places Logo in edit mode. For example, if you enter Logo and type

TO POLY :SIDE :ANGLE

followed by <RETURN>, the system will enter the screen editor with the typed line of text on the screen. Logo indicates that it is in edit mode by printing "EDIT: CTRL-C to define, CTRL-G to abort" in reverse-color letters at the bottom of the screen.

At this point you can use all of the editing keystrokes described beginning on page T-8 to create and/or edit the text for the procedure. Typing <CTRL> C will exit the editor and cause the procedure to be defined according to the text you have typed. If you are using 128K, you will return to the screen from which you entered the editor; if you are using 64K, you will return to nodraw mode. Typing <CTRL> G aborts the edit. Logo will return to nodraw mode without any procedures being defined. If you begin editing a procedure, and decide that you don't want to change it after all (or would like to start over), type <CTRL> G. The procedure you were editing will not be changed.

In this mode, <RETURN> is just another character, which causes the cursor to move to the next line. In edit mode, <CTRL> C causes Logo to evaluate the contents of the edit buffer just as <RETURN> in other modes causes Logo to evaluate the line just typed.

To edit the most recently defined procedure, type just EDIT (or its abbreviation, ED). In draw mode with a 64K computer, this edits the current definition of the procedure most recently defined or PO'd. In all other cases, however, typing EDIT with no inputs returns to edit mode with the contents of the edit buffer intact. For example, after a READ or SAVE, everything read or saved will be in the edit buffer. If you had aborted the definition of a procedure with <CTRL> G, the edit buffer's contents at the time you typed <CTRL> G will

still be in the edit buffer; you can retrieve it by typing EDIT not followed by a procedure name. Typing EDIT followed by the procedure name would edit the procedure as it was last defined.

Typing TO with no procedure name puts you into an empty edit buffer.

### *DRAW Mode*

In draw mode, you use the turtle for drawing on the screen. If you attempt to execute any turtle command while in nodraw mode, the system will enter draw mode before executing the command. The NODRAW command (abbreviated ND) exits draw mode and enters nodraw mode. Actually, there are different types of draw mode.

Splitscreen mode is the normal way in which draw mode is used. Four lines at the bottom of the screen are reserved for text, and the rest of the screen shows the field in which the turtle moves. The turtle field actually extends to the bottom of the screen and so is partially masked by the four-line text region. In fullscreen mode the text region disappears and you can see the entire turtle field. You can still type commands, but they will not be visible. If the system needs to type an error message, it will first enter splitscreen mode so that the message will be visible.

You can use the characters <CTRL> F and <CTRL> S to switch back and forth between splitscreen and fullscreen mode. Pressing <CTRL> F while in splitscreen mode will enter fullscreen mode. Pressing <CTRL> S will restore splitscreen mode. It is also sometimes convenient to be able to switch back and forth under program control. The commands SPLITSCREEN and FULLSCREEN are provided for

this purpose.

In draw mode, Logo displays just four lines of text. This can be an inconvenience, since error messages are sometimes longer than four lines. If you type `<CTRL> T` while in graphics mode, the turtle picture will disappear and you can use the entire screen for text, just as in nodraw mode. The difference is that you are actually still in draw mode: turtle commands can be executed, although you will not see the picture being drawn. The `<CTRL> T` command is useful for looking at long catalog listings, procedures that you have displayed or error messages. `<CTRL> T` is equivalent to the `TEXTSCREEN` primitive. The only way to make the graphics screen visible after using `<CTRL> T` is to type `<CTRL> F` to return to fullscreen mode, or `<CTRL> S` to go back to splitscreen mode.

`TEXTSCREEN` is different from `NODRAW`. `NODRAW` clears the text screen, clears the graphics screen, and resets the pencolor, turtle visibility, pen state, background color, and wrapping mode to their default values.

Here is a list of control characters not related to editing functions. All are available in draw mode and nodraw mode. Some exist in edit mode, also, and are specially indicated.

Non-Editing Control Characters:

- `<CTRL> G` In edit mode, exits the editor without processing the edited text. In draw or nodraw mode, stops execution and returns control to the user.
- `<CTRL> S` In graphics mode, gives mixed text/graphics screen.

- <CTRL> T** In graphics mode, gives full graphics screen.
- <CTRL> W** Pauses program execution, or causes it to Wait. Repeatedly typing **<CTRL> W** will cause Logo to print line by line (or the next list element if lists are being printed). Typing any character other than **<CTRL> W** or **<CTRL> G** will resume normal processing. Hold down **<CTRL> W** (or use it with the repeat key on an Apple II+) to obtain “slow motion” effects.
- <CTRL> Z** Causes Logo to pause. You may type anything and Logo will execute it as if it were a line of the current procedure. Type **CO** or **CONTINUE** to continue.
- <CTRL>-SHIFT-M** Restores output to the screen. (See **OUTDEV**.)
- <CTRL>-SHIFT-P** This generates an underscore (**\_**) character (only with the Apple II+ computer). It is a regular printing character, available in all three modes.

### *Editing*

The Logo system contains a fully-integrated screen editor, and a compatible line editor. The screen editor is used for defining Logo procedures in edit mode, and the line editor is used for typing Logo commands to be executed in draw and nodraw modes.

### *Line Editor*

While you are typing an instruction to Logo, there are many ways in which you can make corrections to the

line before you press <RETURN>.

right and left  
arrow keys    Move the cursor right or left along the line. If you want to insert characters anywhere in the line, simply move the cursor there with the arrow keys, and type what you want.

<ESC> or  
<DELETE>    Erases the character to the left of the cursor.

<CTRL> A    Moves the cursor to the beginning of the line. If you add text now, the characters will push the rest of the line to the right; nothing will be lost or overwritten.

<CTRL> D    Deletes the character at the cursor

<CTRL> E    Moves the cursor to the end of the line.

<CTRL> X    Deletes all characters from the cursor to the end of the line.

<RETURN>    Ends the line and has Logo act upon it. It is not necessary for the cursor to be at the end of the line; all characters you see on the line will be read by Logo.

Lines typed to Logo may wrap around to the next screen line. The editing commands will still work on them exactly as if the line did not spread over more than forty characters. A line can contain up to 256 characters. However, variables defined at toplevel (at the ? Logo prompt) can be combined using WORD or SENTENCE to create variables that contain more than 256 characters. The limit is the length of the edit buffer, which is 8192 characters for *Terrapin Logo for the Apple* and 4096 characters for *Logo PLUS*.

Logo remembers the most recently typed line in both draw and nodraw modes so that you can insert it into the current line by typing <CTRL> P. In *Terrapin Logo for the Apple*, filing commands, such as SAVE and READ, cause Logo to forget the last line typed. In *Logo PLUS*, all lines can be recalled using <CTRL> P.

### *Screen Editor*

Logo's screen editor allows you to define procedures. Enter the editor by typing EDIT, ED or TO, followed by the name of the procedure you wish to define or edit.

Once Logo is in "edit mode" the characters you type will appear on the screen. Pressing <RETURN> will cause the cursor to move down to the next line. (If the cursor was not at the end of the line, it will split the current line into two lines.) It will not cause Logo to execute the line.

Various other commands are available for editing the line on which the cursor appears, and moving to other lines. To move down to the next line, press the down-arrow key or type <CTRL> N (N is for Next). To move up to the previous line, press the up-arrow key or type <CTRL> P (P is for Previous).

Lines may be of any length, as long as they fit in the edit buffer. Lines which are longer than the screen width "wrap around" to the next line. You can tell they are continued lines because an exclamation point ("!") is shown in the last screen column. This mark is not a part of the procedure being typed, and serves only as a reminder that the line does not actually end at that point. There is a slight difference here between edit mode and draw or nodraw mode; only edit mode displays the exclamation marks.

Although Logo procedures are seldom more than a few lines long, the text you may edit is not limited to one screen page. If the text you are typing begins to overflow the current page, the system will automatically shift the display so that the current line is in the middle of the screen. If you use keystrokes to move to either the top or bottom edge of the screen, the next page will appear. The <CTRL> F key inside edit mode moves immediately to the next page of text. To move back to the previous page, type <CTRL> B. If you are on the first page, <CTRL> B will move to the top of it; similarly, on the last page, <CTRL> F will move to the end. If the text you are editing is more than one page long, you can use <CTRL> L to center the current line on the screen.

To exit the editor and have the procedure you typed be defined, type <CTRL> C. To exit without having the procedure defined, type <CTRL> G. After typing <CTRL> G, you can return to the editor with ED or EDIT, and have all the text still there, providing you didn't use graphics or filing in the meantime. (If your computer has 128K of memory, then using the graphics screen will not affect the editor, and vice versa.)

The text you type in the editor doesn't have to be part of a procedure. See the information about TEXTEDIT in the Utilities chapter for using Logo to edit text.

The following is a summary of the available editing commands.

#### Screen Editing Commands:

arrow keys      Move the cursor one character to the left, right, up or down a line without erasing any characters.

- 
- <ESC> or <DELETE>** Rubs out the character immediately to the left of the cursor and moves the cursor one space to the left.
- <CTRL> A** Moves the cursor to the beginning of the current line. <CTRL> A was chosen for this command because it lies at the beginning of a row of the keyboard, and is the first letter in the alphabet.
- <CTRL> B** When editing more than one screenful of text, moves the cursor one screenful of text backwards, or to the beginning of the buffer if not that much text precedes the cursor.
- <CTRL> C** Exits the editor and processes the edited text.
- <CTRL> D** Deletes the character at the current cursor position, that is, the character over which the cursor is flashing.
- <CTRL> E** Moves the cursor to the end of the current line.
- <CTRL> F** When editing more than one screenful of text, moves the cursor one screenful of text forward, or to the end of the buffer if not that much text follows the cursor.
- <CTRL> G** In edit mode, exits the editor without processing the edited text. In all modes, stops execution and returns control to the user.
- <CTRL> L** In edit mode, scrolls the text so that the line containing the cursor is at the center of the screen.

- <CTRL> N** or down-arrow Moves the cursor down to the Next line.
- <CTRL> O** Opens a new line at the cursor position. That is, **<CTRL> O** is equivalent to typing **<RETURN>** and then pressing the left arrow key. It is most useful for adding new lines in the middle of procedures.
- <CTRL> P** or up-arrow In edit mode, moves the cursor to the Previous line. In draw or nodraw mode, retrieves previous input line so that it can be edited and/or re-executed. Note for users of *Terrapin Logo for the Apple*: In draw and nodraw mode, file-system commands cause Logo to forget the previous line.
- <CTRL> X** Deletes all characters on the current line to the right of the cursor. Use **<CTRL> X** several times in a row to delete or cut multiple lines of text, up to a limit of 256 characters. Use **<CTRL> Y** to restore them or paste them in a different location. Use **<CTRL> X** with **<CTRL> Y** to cut and paste text in the editor. See **<CTRL> Y** for information on how to retrieve text deleted in this way.
- <CTRL> Y** Yanks back the lines previously deleted using **<CTRL> X**. If you delete a line and move the cursor to delete another line, then the first deleted line is forgotten. If you delete several lines in a row using **<CTRL> X** without moving the cursor, then **<CTRL> Y** will remember them all. Use **<CTRL> X** with **<CTRL> Y** to cut and paste text in the editor.

## *Using Apple Peripherals*

Logo's ordinary input and output operations deal with the Apple keyboard, the screen, and one disk drive. There are also commands for reading input from up to four game paddles that can be attached to the Apple. (See the PADDLE and PADDLEBUTTON primitives.)

In addition, Logo provides the OUTDEV primitive for accessing output devices other than the screen. This command takes one input that specifies a slot on the Apple board at which a peripheral interface card should be attached. It is also possible to use OUTDEV to designate a user-supplied assembly language routine that should be called in place of the normal character output routine.

The OUTDEV command causes any subsequent output that would normally go to the Apple screen to be directed at the device in the specified slot. Unlike the PR # command of the BASIC language, OUTDEV does not direct everything you type to the alternate device. The Logo screen editor and toplevel line editor are unaffected. Using OUTDEV with an input of 0 will reset the output device to the screen.

Typing <CTRL> SHIFT-M will redirect output to the screen. It is equivalent to executing an OUTDEV 0, but takes effect immediately, even if Logo is in the process of printing something to a printer. (If the computer is "hung" waiting for the printer to receive a character, however, this keystroke will not take effect. Typically, this condition occurs when the printer is off or otherwise disabled.)

## *Color Control*

If you have a color TV monitor, you can use the

PENCOLOR command (abbreviated PC) to change the color of the lines that the turtle draws. You can also use the BACKGROUND command (abbreviated BG) to make the turtle draw on backgrounds of various colors. Both PENCOLOR and BACKGROUND take a number 0 through 6 as input. The correspondence of colors to numbers is as follows:

<u>number</u>	<u>color</u>	
0	black	(The actual color that appears on the screen corresponding to any of these color names can vary greatly depending on the adjustment of the monitor. Also, if you have a black and white monitor, the “colors” will appear as striped vertical lines.)
1	white	
2	green	
3	violet	
4	orange	
5	blue	

Drawing with PENCOLOR 6 “reverses” the color of all dots that the turtle passes over. The actual color produced depends on both the background color and the color of the dot the turtle is passing over but in all cases, reversing the color of a dot and then reversing it again will restore the original color. PENCOLOR 6 is most useful with black-and-white graphics.

If you don’t explicitly give any BACKGROUND or PENCOLOR commands, Logo will default to BACKGROUND 0 and PENCOLOR 1.

To erase a line that has been drawn, set the pen color to match the background color and draw over the same line again. Reset the pen color to continue with your design. *Logo PLUS* contains a PENERASE (PE) command that performs this function for you, allowing you to erase a line when you redraw over it. Cancel out of PENERASE mode using the primitive PENDOWN (PD) or changing the pencolor with PENCOLOR (PC).

### *Drawing on Colored Backgrounds*

When drawing on a colored background (2 through 5), only two of the four colors—green, violet, blue, orange—are available. When the background is green or violet, blue and orange cannot be used: PENCOLOR 4 will draw in green and PENCOLOR 5 will draw in violet.

When the background is blue or orange, PENCOLOR 2 will draw in orange and PENCOLOR 3 will draw in blue. Also, if you draw a picture on the screen and then change the background color, the colors of the lines in the picture may change, or the lines may become distorted in unexpected ways; however, returning to the background color in which the lines were drawn will always restore their original appearance. These strange effects are the result of a compromise with the Apple computer color system, which does not allow, for example, green dots to appear very close to orange dots.

### *Drawing without Color Control*

In order to obtain clear colors with the Apple computer, the Logo system must draw lines more thickly than would otherwise be necessary. This means that drawings will not look as precise as they could if one drew only thin lines. If you don't care about color, you will obtain better looking drawings by using thin lines. To do this, select BACKGROUND 6. In BACKGROUND 6, PENCOLOR 0 gives black, 1 through 5 give "white," and 6 gives "reverse." The reason that "white" is in quotes is that "white" lines may not always appear white on a color monitor. In particular, "white" vertical lines will be either red or green, depending on their position.

### *Adding Text to the Graphics Screen*

*Logo PLUS* includes commands for placing text of varying styles on the graphics screen as well as a built-in screen editor. Text can be entered in plain, bold, underlined or italicized text and the color of the text can be set using the primitive PC. *Logo PLUS* users should refer to the *Getting Acquainted with Logo PLUS* booklet for instructions on using these primitives.

Users of *Terrapin Logo for the Apple* can stamp plain text characters on the graphics screen using the HIRES.TOOLS program that is contained in the Utilities II package, available separately from Terrapin.

### *The Logo File System*

The Logo file system allows you to save procedure definitions on floppy disk. A user may have many files on a single disk, and the files are distinguished by the fact that they are named. The names of the files are listed in the disk catalog.

The standard file-handling commands in Logo are SAVE, SAVEPICT, READ, READPICT, ERASEFILE and ERASEPICT. These and other disk operations, such as LOCK, UNLOCK, RENAME, etc., can also be performed using the DOS primitive. *Logo PLUS*, implemented in ProDOS, includes many other file handling primitives, described in the *Getting Acquainted with Logo PLUS* booklet.

### *Disk Files*

When you use Logo, you should normally have a Logo file diskette mounted in the disk drive. File diskettes may be created as described in the chapter Beginning in Logo. In addition to the following sections, users of

*Logo PLUS* may also want to refer to the *Getting Acquainted with Logo PLUS* booklet for information regarding the use of directories.

If you want to save your procedure definitions, use the **SAVE** command. For example,

```
SAVE "MYSTUFF
```

will save all the procedure definitions and names currently in the workspace in a file named **MYSTUFF**. There can be both a procedure and a file with the same name; however **SAVE** saves everything in the workspace and will not save only the procedure by that name. If you already had a file of that name, the old one will be deleted.

The **READ** command takes a file name as input and reads the procedures and names from that file into the workspace. The procedures and names will be added to the ones currently in workspace. If you are using Logo with 64K of memory, be aware that reading or saving a file while in draw mode will first move you to nodraw mode. This is because, with just 64K of memory, the editor and the graphics screen must share the same memory location.

Notice that the file names given as inputs to **SAVE** and **READ** are preceded by a quote and have no following quote.

The **CATALOG** command lists all the files on the disk. Logo workspace files will be listed with the characters ".LOGO" appended to the name. For example, the file created by the instruction

```
SAVE "MYSTUFF
```

will be listed in the catalog as MYSTUFF.LOGO. Do not include the .LOGO part of the name when you use the READ or SAVE commands.

To remove a file from the disk, use the ERASEFILE command, which takes as input the name of the file to be erased. As with READ and SAVE, the filename must be preceded by a " mark.

### *Saving Pictures*

In addition to saving procedure definitions, Logo also allows you to save a graphics screen image on the disk so that it can be read back in and displayed. To do this, use SAVEPICT and READPICT.

SAVEPICT, which is similar to SAVE, takes a name as input. It saves on the disk the picture currently on the turtle graphics screen. (SAVEPICT should only be done when you are in graphics mode.) READPICT reads in a picture that was saved by SAVEPICT, and displays this picture on the screen.

When you do a CATALOG you will notice that “.PICT” is appended to picture files just as the “.LOGO” is appended to regular Logo files. Do not include the “.PICT” part of the name when you use the READPICT command. (However, you do need the “.PICT” if you access the file from outside of Logo.)

To erase a picture from the disk, use the ERASEPICT command, which takes a quoted picturename as input.

Logo picture files are stored in the standard binary file format. In memory, they are located in the primary graphics page.

## *Changing the Turtle Shape*

### *Building Your Own Shape Tables*

Instructions for using the shape editor can be found in the Utilities chapter. This section involves a more technical discussion of how the shape edit program works.

The Logo turtle is drawn using the Apple “shape” mechanism, which allows specification of shapes by tables of two- and three-bit vectors as described in the Applesoft Programming Reference Manual. You can design your own shapes for Logo to move around on the screen in place of the turtle.

To set up your own shape table, deposit the location of the first element of the table in the address USHAPE. (See the explanation of Logo addresses later in this chapter.) The size of the turtle or of the created shape can be changed with the one-byte size code contained in address SSIZE. The default value, 1, is best for the regular turtle; however, values of 2 or greater often make user-defined shapes more visible.

Unlike the general Apple shape mechanism, the Logo interface to shapes allows user-defined shapes to be displayed only at 0, 90, 180 and 270 degree headings. The heading at which the shape is displayed is determined by the quadrant in which the “turtle” is facing and can be changed by turning the “turtle” with the usual LEFT and RIGHT commands. The format of shape tables is as described in the Applesoft Reference Manual, except that the header information in the shape table should be omitted. Begin each shape table directly with the vectors. Terminate it normally.

You can construct a shape table by hand and use .DEPOSIT to store it in the Logo area reserved for user code, and then set USHAPE and SSIZE. Note that you can make more than one user-defined shape and switch between them by changing USHAPE.

Note that the shape editor is itself a collection of Logo procedures that work by using .EXAMINE and .DEPOSIT according to the scheme outlined above. You can read in the procedures from the Utilities disk file SHAPE.EDIT and use them as a guide to writing similar functions.

### *Assembly Language Interfaces to Logo*

The Logo system for the Apple has been designed to be both powerful and easy to use. Writing and executing programs in the Logo language using its built-in primitives should be sufficient for most purposes. However, there are situations in which it is desirable to extend the capabilities of the system by getting direct access to machine language.

**Warning:** This chapter will only be useful and intelligible to people who are familiar with assembly language programming on the Apple.

The Logo system has various “hooks” built into it that enable users to directly access memory locations in the Apple and to interface assembly language routines to Logo programs. The Logo Utilities disk includes a 6502 machine language assembler that aids in doing this. These hooks into Logo allow you to create simple animation effects by supplying a new shape to be displayed in place of the Logo turtle. They also allow

you to modify the behavior of the Logo editor so that it can be used as a regular text editor rather than as a procedure editor, to access disk files in non-standard ways and to generate music.

Note: Throughout this chapter, we will use the convention of specifying hexadecimal numbers as prefixed by a dollar sign, e.g., \$9E is 158 decimal.

### *.EXAMINE and .DEPOSIT*

These two commands are essentially the usual Apple PEEK and POKE routines. One difference is that the addresses should always be specified as positive numbers. Apple PEEK and POKE require addresses above 32K to be given as negative numbers.

.EXAMINE takes an address as an input and returns (as a number) the byte stored in that address.

.DEPOSIT takes two inputs, an address and a numeric value, and deposits the value in the byte specified by the address.

These commands are useful for communicating with special-purpose I/O devices, especially when the facility supplied by OUTDEV is insufficient. Needless to say, .DEPOSITing into arbitrary memory locations can cause Logo to crash or do other unfriendly things.

Note that the addresses used with these commands are ordinary Logo numbers, which are expressed in base 10, even though it is customary to think of Apple addresses as written in hexadecimal notation. For many purposes it would be useful to write a conversion routine that converts from hexadecimal to base ten. (One is provided in Terrapin's Utilities II package.)

That way, you could type, for example

```
.EXAMINE HEX "9E
```

rather than

```
.EXAMINE 158
```



When Logo is running, the upper 16K “language card bank” is active. The memory locations \$C0B0-\$C08F (49280-49295) contain soft switches for this upper 16K bank area. You can look at memory locations in this area using .EXAMINE, but using .DEPOSIT with an address in this area will cause unpredictable effects.

### *Writing Your Own Machine-Language Routines*

You can interface your own machine-language routines to Logo by using the .CALL primitive. .CALL takes two inputs: the first is the address of the routine, and the second is an integer input that the routine may examine. The routine may output an integer or output nothing. The .CALL primitive always requires two inputs, regardless of whether the user routine chooses to examine the second one.

.CALL transfers control to the address specified by its first input. Naturally, before doing this, you should assemble an appropriate routine and store it at the address. The available memory for user machine code begins at \$99A0 and extends to \$9AA0. You can do the assembly by hand and store the routine using .DEPOSIT, but you will find it much more convenient to make use of the Logo assembler described later in this section.

When your routine begins executing, the page zero locations NARG1 and NARG1 + 1 contains the first input to .CALL, which is just the address of the routine itself. NARG1 + 2 and NARG1 + 3 are guaranteed to contain zero at the time the routine is called. The routine may use locations NARG1 through ANSN4 + 3 as temporary storage locations, without worrying about restoring them before returning. These storage locations are volatile; that is, Logo may change these locations between successive calls to your routine. Locations USERPZ through \$FF are not used by Logo and so can be used by your routines as non-volatile, page-zero storage.

NARG2 through NARG2 + 3 contain the second input to .CALL, stored as a four-byte fixnum in two's complement form. Thus .CALL (\$ "99A0) 3 would result in the following values in memory:

NARG2	NARG2+1	NARG2+2	NARG2+3
3	0	0	0
NARG1	NARG1+1	NARG1+2	NARG1+3
\$A0	\$99	0	0

Substituting -1 for 3 would make NARG2 through NARG2 + 3 contain \$FF. To output an integer, store the integer to be returned (using the above format) in the four locations with NARG2 through NARG2 + 3, and jump to location OTPFX2. If the number is stored in some other set of 4 consecutive page-zero variables (such as NARG1), load Y with the address and jump to OTPFIX.

To output the Logo word "TRUE, jump to OTPTRU; similarly, jumping to OTPFLS will cause your routine to output "FALSE. To output no value, simply end the routine with an RTS instruction.

Here is an example which reads the state of the Open-Apple key or Button 0 (by addressing the Open-Apple key's location \$C0) and returns "TRUE or "FALSE, depending on whether the Open-Apple key is pressed. The code here is written in standard 6502 assembler format. To use it you will have to assemble it by hand and deposit the instructions in memory (see the next section).

```

                ORG $99A0
OAKEY          EQU $C061
OTPTRU        EQU <see Addresses.Logo>
OTPFLS        EQU <see Addresses.Logo>
APPLEKEY:     LDA OAKEY
                BMI PRESSED
                JMP OTPFLS
PRESSED:      JMP OTPTRU
    
```

Now you can set the Logo variable `APPLEKEY` to the address of the label `APPLEKEY` and execute this new “primitive” by typing

```
.CALL :APPLEKEY 0
```

(Note that an input is needed, even though it is ignored.) This will work just like a normal primitive or procedure—

```
PRINT .CALL :APPLEKEY 0
```

will print TRUE or FALSE.

When a machine language routine has determined some error condition that would make it inappropriate to return to the Logo procedure that called it, it jumps to `PPTTP`, which effectively executes the Logo `TOPLEVEL` primitive.

## *The Logo Assembler*

The Logo assembler is a 6502 assembler that is written in the Logo language. The assembler is stored on the Logo utilities disk in the file ASSEMBLER. (The ASSEMBLER program in turn reads data stored on the Logo disk in auxiliary files AMODES and OPCODES.) To use the assembler, simply read this file into Logo as you would any normal Logo file and then run a procedure called SETUP:

```
READ "ASSEMBLER
SETUP
```

To assemble a routine, you write the routine in the format of a Logo procedure, using the Logo editor. For example, the Open-Apple key example listed earlier would be written as the procedure:

```
TO APPLEKEY.CODE
  [MAKE "OAKEY $ "C061]
  [MAKE "OTPTRU <see Addresses.Logo>]
  [MAKE "OTPFLS <see Addresses.Logo>]
  APPLEKEY: LDA OAKEY
             BMI PRESSED
             JMP OTPFLS
  PRESSED:  JMP OTPTRU
END
```

Notice that there are differences in syntax between the input accepted by the Logo assembler and the standard 6502 assembler. The syntax of code for the assembler is explained later in this section.

Once you have defined the procedure you now assemble it by typing

## ASSEMBLE "APPLEKEY.CODE

ASSEMBLE will now assemble the instructions and place them in the default location (\$99A0). Also, any labels in the code (such as APPLEKEY, above) will now be defined as Logo symbols. So now you can call the routine by

```
.CALL :APPLEKEY 0
```

### *Using the Assembler to Write I/O Routines*

While it is possible to use the .EXAMINE and .DEPOSIT primitives to operate most peripheral devices, machine language routines are required for others. If the peripheral device is one which has a built-in "driver," then you can use the OUTDEV primitive. OUTDEV takes as input a slot number 1 through 7 as an input and directs the Logo character input or output routines to the device at the specified slot. (OUTDEV 0 directs output to the screen.)

Some devices, however, may require special routines to handle input and output. If you specify OUTDEV with an input greater than 8, the input will be interpreted as the address of a routine in memory that should be called in place of Logo's regular character input or output routine.

Many peripherals use a technique called "handshaking" to assure that the computer does not try to send data to them (or read data from them) too fast. The following program will interface Logo to such a device. We assume that STATUS is the memory-mapped I/O address on the peripheral card indicating the status of the device. In this case, bit 7 is high if the device is ready

to receive a character. DATA is the address where bytes to be sent should be stored.

Once you have assembled this routine, you may access the peripheral by executing OUTDEV:TYOWAIT.

```
TO CODE
  [MAKE "STATUS <address>]
  [MAKE "DATA <address>]
  TYOWAIT: LDX STATUS
           BPL TYOWAIT
           STA DATA
           RTS
END
```

A character output routine like this, which is meant to be called via OUTDEV, should expect that the A register will contain the byte to be output. Here is another example output routine. This one causes all ! characters to be printed as spaces:

```
TO IOCODE
  XCLOUT: CMP #"!
          BNE OUTCHAR
          LDA #32
  OUTCHAR: JMP COUT
END
```

### *Syntax of Input to the Assembler*

In order to take advantage of some aspects of the Logo language, the Logo assembler uses a format slightly different from most assemblers. Each assembly language program is stored as a Logo procedure, although this procedure cannot be executed directly. The following paragraphs concisely describe the Logo assembler format; a study of the examples provided will

better explain how to write assembly language programs to interface with Logo.

Labels within the program are indicated by a postfixed colon. References to page-zero memory locations that are not indirect-indexed (LDA (FOO ,X)) or indexed-indirect (LDA (FOO) ,Y) must have an exclamation point before the label or expression that is on page zero. (If you forget the exclamation point, the instruction will be coded as absolute references, and will occupy one more byte.) There must be a space following every ! (indicating page-zero reference) or # (indicating immediate mode), and after every label or reference to a label. The operand of an instruction may be a word (a reference to a label), a number, a list, or a single-letter word beginning with a quote. If the latter case, the operand is the ASCII value of the letter.

Anything inside a list is evaluated as a regular Logo expression. If the list is the first thing on the line, it is not allowed to output a value, and is evaluated for “side-effect” (label assignment) only. If it is an operand (follows the name of an instruction), it is expected to output something. Thus, arithmetic expressions such as :FOO + 3, where FOO is a label or regular Logo symbol, may be used provided they are enclosed in square brackets. Of course, references to the values of labels inside square brackets must have dots (:) before them, and spaces have their normal significance. All labels are Logo variables. DOT is a Logo variable whose value is the current location being assembled.

The HI8 and LO8 procedures, which return respectively the high and low eight bits of a number, are also useful inside lists. Use them like this:

```
LDA # [LO8 :SOURCE]
STA ! DEST
LDA # [HI8 :SOURCE]
STA ! [:DEST+1]
```

The \$ procedure takes as input a word that is a hexadecimal number and outputs the number that it represents. Thus, hex numbers may be included in programs by placing a call to the \$ inside a list. Use the MAKE primitive to assign values to labels.

If you use octal or binary numbers a lot, you might want to change the value of the Logo word \$BASE. This is the base used by the \$ procedure. Changing it to 2 gives you binary, and so on. You can do this within the source for an assembly language program with [MAKE "\$BASE 2].

You can assemble arbitrary bytes into code by placing the number on the line with nothing (except perhaps a label) preceding it.

Here is a simple program in normal assembler syntax:

```
ORG      EQU $99A0
NARG2    EQU $9E
BELL     EQU $1C40
PASS:    LDA NARG2
         CMP #$04
         BEQ YES
         RTS
YES:     JMP BELL
END
```

And in the Logo assembler syntax:

TO CODE

```
PASS:    LDA ! NARG2 ; ! means page zero. Note space after !.
          CMP # ["04]
          BEQ YES
          RTS
YES:     JMP BELL
END
```

To assemble this program, load in the assembler and type **SETUP** and **READ "ADDRESSES"**. The following will assemble the above program, with a default origin of \$99A0. (To assemble at some other start address, assign the value to the Logo variable **ORG**.)

**ASSEMBLE "CODE"**

This will generate a listing file on the screen and deposit the code in memory. The labels are available as Logo symbols for use with **.CALL**, **.DEPOSIT**, and **.EXAMINE**. To invoke the above routine, type

```
.CALL :PASS 4
```

to beep the bell, and **.CALL :PASS <anything but 4>** to do nothing. This is sort of a secret “password” program.

If you try to assemble long programs, you may run out of memory. One way to get more memory is to load in only those instructions that your program uses. In a fresh Logo, read in the **OPCODES** file from the utilities disk and erase the instructions (using **ERNAME "BIT**, for example) that you don't plan to use. Then, rewrite this as the new **OPCODES** file. (Of course, you should not do this on the original Logo disk. Save copies of the original assembler files on an ordinary Logo file disk and run the assembler using these copies.)

*Saving Assembled Routines on Disk*

With the DOS primitive, you can save the actual machine code that the assembler generates. The following will save your assembled routines in a file called ROUTINES.

```
DOS [BSAVE ROUTINES.BIN,A$99A0,L$100]
```

To load the routines into Logo, type

```
DOS [BLOAD ROUTINES.BIN]
```

The BIN is short for BINARY, and might help you remember that the file is a saved machine-language file. Keep in mind that in addition to saving the actual machine code, you should save the Logo variables that define the addresses used by .CALL. One way to do this is to type EDIT NAMES, then exit the editor with <CTRL> G and execute ERASE ALL. Re-enter the editor by typing just EDIT and edit the definitions to include only the ones you still want. Then exit the editor with <CTRL> C. Save the file by typing SAVE "ROUTINES. Then, to reload your routine, type READ "ROUTINES and DOS [BLOAD ROUTINES.BIN].

### *Example: Generating Music*

This section presents an example of an assembly language extension to Logo. Although *Terrapin Logo for the Apple* has no primitives for playing music, you can use the .CALL feature to interface a machine-language routine to produce pitches with the Apple II speaker. (*Logo PLUS* has a built-in NOTE primitive.) The speaker produces a narrow pulse each time the location to which it is mapped, \$C030 (49200), is referenced. Try repeatedly reading this location from Logo using .EXAMINE. (Due to the way the speaker is interfaced, depositing in the location has no effect. Additionally, the speaker generates clicks only on every other reference. This brings the pitch down one whole octave, but does not affect the intervals of pitches played.)

In order to play pitches, a program has to examine this location many times each second. The number of clicks produced per second is called the frequency. To make the pitches sound equally spaced, the ratio of successive frequencies must be constant; that is, the frequencies must be in geometric progression. In Western music, which has twelve pitches to the octave, this constant must be such that the frequency doubles after twelve pitches; thus, the ratio of successive pitches in the scale is the twelfth root of 2, or approximately 1.05946.

Closely related to the concept of frequency is that of period. The period of a pitch is simply the reciprocal of its frequency. Given the period, it is possible to play the corresponding pitch by repeatedly generating a narrow pulse (click), and then waiting for the period to expire. The program which does that will have to be written in machine language, since it must run very quickly.

To make Logo play music, we need to write some procedures. Let's say there should be a procedure called "PLAY," and that it should take two inputs: a list of pitches and a corresponding list of durations. The pitches should be numbers specifying the number of chromatic steps above or below a center pitch. The durations should be lengths of time for the note to sound, with 1 being the shortest, 2 being twice as long, and so on.

```
TO PLAY :PITCHES :DURS
  IF :PITCHES = [] STOP
  PLAY.NOTE (FIRST :PITCHES) (FIRST :DURS)
  PLAY (BF :PITCHES) (BF :DURS)
END
```

Even though we're not exactly sure how notes will be played, we can assume that `PLAY.NOTE` actually plays a note (given the pitch number and duration) because that's what we're going to write it to do.

Since the notes are played by a machine-language program that requires the period of the pitch, we must find some way of associating the periods of various pitches with their representation in the `PLAY` procedure. A table would be one good way of doing this. For each note, there is an entry in the table that contains the period. We'll construct our table as Logo words, and have the periods as the things associated with the names. We'll choose some arbitrary name for this table, and then have the individual notes be represented by words that begin with the table name and have the number of the pitch at the end of the name. So, if we call the table "#," the period for note number 3 is in the Logo variable called "#3." We'll assign a special value to mean rest, and use the Logo variable #R to store this value, so that "R" will cause a rest in the `PLAY` procedure.

Additionally, it would be useful to be able to specify notes above or below the center octave in some convenient notation. We have chosen postfixed plus and minus signs to indicate different octaves. In inputs to the PLAY procedure 4 means the fourth pitch above the center tone. 4+ means the same pitch an octave above it, and 4- the pitch one octave below. (You can worry about an appropriate notation for extending the range to more than these three octaves if you wish.) The following MAKE.PITCHES procedure associates each pitch with the proper period. It takes as input the number of the highest octave and the period of the highest pitch in that octave.

```

TO MAKE.PITCHES :PERIOD
  MAKE.OCTAVE 11 "+" :PERIOD
  MAKE.OCTAVE 11 " " :PERIOD * 2
  MAKE.OCTAVE 11 "-" :PERIOD * 4
  MAKE "#R 16384
END

```

```

TO MAKE.OCTAVE :PITCH :OCTIND :PERIOD
  IF :PITCH = 0 STOP
  MAKE (WORD "# :PITCH :OCTIND) :PERIOD
  MAKE.OCTAVE :PITCH-1 :OCTIND :PERIOD * 1.0596
END

```

This is about as far as we can proceed in Logo before we know the specifics of the implementation of the note-generating routine. This machine-language routine should sound a note with a specific period for a certain length of time. As mentioned before, the way to generate a tone on the Apple speaker is to cause a click, wait for the period to expire, and keep doing this until the note is supposed to be over.

The heart of our machine language routine will be a subroutine called **CLICK** (listed below). This routine is called repeatedly with the (16 bit) period in locations **PER.H** and **PER.L**. It copies them to another place so that they will still be valid next time around **CLICK** is called.

One way to cause notes to have a certain duration would be to call the **CLICK** routine a certain number of times. Calling it twice that many times would result in a note twice as long. That is fine if only one period (pitch) is used. Unfortunately, the **CLICK** routine by its very nature takes a different amount of time for different periods (i.e., different pitches); therefore, the routine that plays notes must convert the actual duration to the number of clicks to generate.

If we pick a base pitch, and scale the durations of all other pitches from that one, our problems will be solved. The number of times to call the **CLICK** routine for a given duration and period is  $\text{:DURATION} * (\text{:BASE.PERIOD} / \text{:PERIOD})$ . This scaling is called normalization. It is much easier to do the required multiplication and division in Logo than in machine-language, so we'll calculate this scaling factor in Logo. The machine-language routine will take this number as an input, and call the **CLICK** routine that many times. Here is the entire machine-language program for playing notes:

```
TO MCODE
[MAKE "SPKR $ "C030]
[MAKE "DUR.L :NARG2]
[MAKE "DUR.H :NARG2+1]
[MAKE "PER.L :NARG2+2]
[MAKE "PER.H :NARG2+3]
[MAKE "COUNT.L :USERPZ]
```

```

[MAKE "COUNT.H :USERPZ+1]
[(PRINT [STARTING ADDRESS:] :ORG)]
TONE:      LDA  ! DUR.L
           ORA  ! DUR.H
           BEQ  EXIT           ;A duration of 0 means no note.
           LDA  ! DUR.L
           SEC
           SBC  # 1
           STA  ! DUR.L
           LDA  ! DUR.H
           SBC  # 0
           STA  ! DUR.H
           JSR  CLICK
           JMP  TONE

EXIT:      RTS
CLICK:    LDA  ! PER.L
           STA  ! COUNT.L
           LDA  ! PER.H
           STA  ! COUNT.H
           BIT  ! COUNT.H
           BVS  PDLOOP
           LDA  SPKR           ;Click
PDLOOP:   LDA  ! COUNT.L
           ORA  ! COUNT.H
           BEQ  EXIT
           LDA  ! COUNT.L
           SEC
           SBC  # 1
           STA  ! COUNT.L
           LDA  ! COUNT.H
           SBC  # 0           ;propagate carry
           STA  ! COUNT.H
           JMP  PDLOOP

[(PRINT "LENGTH: :DOT - :ORG "BYTES. )]
END

```

The loops that make up the body of the **CLICK** and **TONE** routines both have an interesting property: Every iteration takes the same amount of time as every other. Some methods of writing these loops would have them run faster or slower when **DUR.H** (or **PER.H**) was 0. Those methods would cause durations of 400 clicks not to be twice as long as durations of 200.

Note that in the **CLICK** routine there are some instructions that are outside the loop, and are executed once for each period of the tone. The time they take has an effect on the pitches produced. It is just like adding a small amount to every period. To counteract this, we subtract a small amount from each period. This factor is the **FUDGE** constant.

Some method of interfacing the machine-language routine to the Logo procedures is needed. The **.CALL** primitive is provided for just this purpose, but allows passing only one input. What we need is a way to pass both the period (**PER.H** and **PER.L**) and the duration (**DUR.H** and **DUR.L**). A careful look will show that this adds up to 32 bits, which is the number of bits **.CALL** can pass to the machine-language programs it calls. If we arrange memory locations so that **DUR.L/DUR.H** are the low two bytes of the input to **.CALL**, and **PER.L/PER.H** the high two, the following procedure will give the two inputs to machine-language routines:

```
TO .CALL.2 :ADDR :INPUT1 :INPUT2
.CALL :ADDR (ROUND :INPUT2) + (ROUND
      :INPUT1) * 65536
END
```

Note the use of the **ROUND** primitive. Were it not called, non-integer periods would cause the result of the multiplication not to be a multiple of 65536, interfering

with the duration. The `PLAY.NOTE` procedure is a combination of this procedure and the normalization step mentioned before. (The three inputs to `.CALL.2` are put on separate lines below, to make them more readable.)

```
TO PLAY.NOTE :PERIOD :DURATION
MAKE "PERIOD THING (WORD "# :PERIOD)
.CALL.2 :TONE
      :PERIOD - :FUDGE
      (:DURATION * :BASE.PERIOD / :PERIOD)
END
```

### *The Music demonstration program*

There are two music-related files on the Utilities Disk. One is a Logo file called `MUSIC`, and the other is a file of saved machine-language routines called `MUSIC.BIN`. To try out the music demo, type `READ "MUSIC` and `SETUP`. All the procedures shown here are included. See also the Music and Utilities chapters for more information.

### *Useful Memory Addresses*

This section contains brief descriptions of addresses in the Logo program that serve as “hooks” for modifying Logo with `.EXAMINE` and `.DEPOSIT` and for interfacing assembly language programs to Logo as described earlier. The actual values of the addresses are contained in a file called `ADDRESSES` that is included on the Logo utilities disk. Beware that the actual values of these addresses may change with new releases of Logo. Executing `READ "ADDRESSES` in Logo will define the addresses as normal Logo variables whose values are integers. (Comments in the file also give the

values of the addresses in hexadecimal notation.) When using an address, it should be preceded with the character : as in .EXAMINE :EPOINT.

Page zero locations:

- |               |                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
|---------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>EPOINT</b> | Location of the current character in the edit buffer. Used by the editor and the EDOUT routine.                                                                                                                                                                                                                                                                                                                                                        |
| <b>ENDBUF</b> | The address of the last character in the edit buffer, plus one. The disk saving routines (see SAVMOD) save from \$2000 to the address in this location.                                                                                                                                                                                                                                                                                                |
| <b>SAVMOD</b> | If the contents of this location is 0, READ and SAVE work normally. If it is non-zero, SAVE saves whatever is in the edit buffer (which can be text other than procedures and names) and READ reads a file into the edit buffer from disk, but doesn't evaluate it.                                                                                                                                                                                    |
| <b>BKTFLG</b> | If this location contains 1, then Logo attempts to print out objects in a manner such that they can be read back in. This is useful when you are printing to the EDOUT device. All lists will be printed with brackets around them; none will be printed in "sentence" form and single-quoted words will be printed with their single quotes. PO NAMES will print out Logo variables and their values with MAKE "VARIABLE 3 instead of "VARIABLE is 3. |

- NOINTP** Controls the action of the special “interrupt” characters `<CTRL> F`, `<CTRL> G`, `<CTRL> S`, `<CTRL> T`, and `<CTRL> W`. If the location contains zero (the default), these keys have their normal action in draw and nodraw mode. If it contains 1, these characters have no special meaning, and will be recognized by `READCHARACTER`. `<CTRL> Z` and `<CTRL> SHIFT-M` are still enabled. To disable them also, deposit 255 in `NOINTP`.
- CH, CV** These locations contain the current cursor location, in columns and rows, respectively. `.EXAMINE :CH` outputs the current horizontal cursor position. See the `CURSOR` primitive.
- OTPDEV** Contains the address of the routine currently being used for character output.
- INPDEV** Like `OTPDEV`, but for character input.
- USHAPE** Pointer to user-defined turtle shape. See the shape-editing section.
- SSIZE** Shape size for turtle or user shapes. Default = 1.
- INVFLG** Determines whether characters will be white-on-black (default, contents = 255), black-on-white (contents = 0), or flashing (contents = 64).
- NARG2** Second input to `.CALL`. Four bytes.

---

NARG1	First user-available temporary location. All memory from here to ANSN4 + 3 is available for user routines.
ANSN4	Last user-available temporary location. This and the next three bytes are free.
USERPZ	First user-available permanent page zero location. From this location to \$FF is not used by Logo.
HIMEM	(.EXAMINE :HIMEM) + 256 * (.EXAMINE :HIMEM + 1) outputs the highest address available for user machine-language programs. It is set by the DOS command MAXFILES 1 to \$9AA5. See VZZZZZ.

Other useful addresses:

OTPFX2	Jumping to this address will cause .CALL to output the integer stored in NARG2 through NARG2 + 3.
OTPFIX	Like OTPFX2, but returns to Logo with the value of the integer stored in the four successive bytes starting with the page-zero location pointed to by the Y register.
OTPTRU	Jump to this routine to output "TRUE.
OTPFLS	Output "FALSE.
GETRM1	Loading from or storing to this location twice, e.g., LDA GETRM1, LDA GETRM1, enables Logo locations in extended memory and disables Monitor ROM.

**KILRAM** Referencing this location enables the Apple monitor ROM. It is enabled during .CALL execution unless explicitly disabled.

**PPTTP** An alternate exit for user machine-language routines. Jumping to this address runs the Logo primitive TOPLEVEL. It is useful to return to Logo in this manner when some error condition has occurred, making it inappropriate to continue executing the .CALLing procedure.

**COUT** Logo's normal screen character-output routine. Prints the character in the A register on the screen.

**EDOUT** Routine to place the character in the A register in the edit buffer. Deposits A in location pointed to by EPOINT and increments EPOINT. Returns without doing anything if EPOINT is greater than \$3FFF. Can be used with OUTDEV to cause Logo to place text directly in the edit buffer.

**PNTBEG** Routine to reset EPOINT to beginning of the edit buffer. Use before outputting to the buffer (with OUTDEV :EDOUT) the first time.

**ENDPNT** Routine to set ENDBUF to EPOINT. Use when finished printing to the buffer. See PNTBEG, EPOINT, ENDBUF.

---

BELL	Routine to beep the bell. Use PRINT1 CHAR 7 to beep from Logo.
HOME	Homes the cursor and clears the screen.
CLREOP	Clear from cursor position to end of screen.
SCROLL	Scroll.
CLREOL	Clear to end of line.
FILLEN	Contains length of last file loaded.
FILBEG	Start address of last file loaded.
VZZZZZ	$(.EXAMINE :VZZZZZ) + 256 * .EXAMINE :VZZZZZ + 1$ outputs the lowest address available for user machine-language programs. Although this address may be below \$99A0, you should not place code in the intermediate region, since future releases of Logo may use that area of memory.

## *Miscellaneous Information*

### *Using the Logo System as a Text Editor*

The Utilities chapter provides information on the use of the TEXTEDIT file. The following discussion is a more technical analysis of the program.

The SAVE primitive normally saves the names and procedures in the workspace by placing them in the edit buffer, and then saving the buffer on disk. If you want to write arbitrary text on disk to be loaded back into the edit buffer without being evaluated, you can set the "save mode" flag; it controls the action of READ and SAVE.

Normally the memory location SAVMOD contains zero. When you deposit any non-zero number, SAVE will save the current contents of the edit buffer (rather than saving workspace), and READ will not evaluate the edit buffer after reading it in from disk.

Nothing in Logo but GOODBYE (or changing the SAVMOD contents to 0) resets this flag. The actual address of the flag may be found in the file ADDRESSES on the Utilities Disk, which can be accessed by typing READ "ADDRESSES. This should be done before creating the text in the edit buffer, or it will be lost. Exit the edit buffer by presing <CTRL> G. Then, before saving the file, you should type

```
.DEPOSIT :SAVMOD 1
```

If you are going to use the Logo procedure editor as a text editor for an entire session, you might want to type this in at the beginning. If you should want to read or

save some procedures (or names), just type

```
.DEPOSIT :SAVMOD 0
```

and things will be back to normal.

To get the file back to work on at a later date, READ "ADDRESSES from the utilities disk, type .DEPOSIT :SAVMOD 1, and READ "MYFILE from your own disk, then type ED followed by a RETURN. Do not type TO after reading the file or you will start in a new empty buffer and have to read in the file again.

### *Printing the Edit Buffer*

The following procedures will print the contents of the edit buffer to the peripheral in slot SLOT. Before using it, you will need to make ENDBUF have the value listed in the Logo ADDRESSES file.

If you have been using the Logo screen editor as a text editor, typing HARDCOPY after typing the <CTRL> G editor command will print the contents of the edit buffer.

```
TO HARDCOPY :SLOT
  OUTDEV :SLOT
  PRINTMEM 8192 256 * (.EXAMINE :ENDBUF + 1) +
    .EXAMINE :ENDBUF
  OUTDEV 0
END
```

```
TO PRINTMEM :FROM :TO
  IF :FROM > :TO STOP
  PRINT1 CHAR .EXAMINE :FROM
  PRINTMEM :FROM + 1 :TO
END
```

### *Self-Starting Files*

There are two methods for creating self-starting files in Logo.

One method involves the use of the variable name `:STARTUP`, which has special meaning in Logo. Anytime a file is read in, Logo looks for a global variable `:STARTUP`. If this variable is present, Logo will run it. The value of the variable must be a list and can contain any number of instructions. For example, to create a file that automatically runs a main procedure called `SETUP`, type the following:

```
MAKE "STARTUP [SETUP]
```

Then save the file in the standard way. When the file is read back in, the `SETUP` procedure will be executed as soon as all the procedures have been evaluated.

Before the `STARTUP` variable feature was added to the language, it was necessary to follow the steps in this next process, which still might be of interest to some readers.

You can use `:SAVMOD` to append arbitrary text to the end of procedures and names to be saved on disk. This is useful if you have a program that you want to start automatically every time a certain file is loaded in.

For example, suppose you want a procedure called `SETUP` to be run automatically every time you read in the file called `GAME`. This can be accomplished by arranging things so that the command `SETUP` is executed automatically each time the `GAME` file is read in. To do this, define all the procedures needed for `GAME`. Type `EDIT ALL` to get the entire workspace

into the edit buffer. Then, go to the end of the buffer (using <CTRL> F) and insert commands you want executed directly (SETUP, for example). Then, type <CTRL> G to exit the editor and then type

```
.DEPOSIT :SAVMOD 1  
SAVE "GAME  
.DEPOSIT :SAVMOD 0
```

Now, whenever the GAME file is loaded, the procedures will be defined and the SETUP instruction that you appended to the end of the edit buffer will be executed.

### *Various System Parameters*

This section contains information about this specific implementation of Logo. Although it is not necessary to know what is presented here in order to use Logo, the curious may find these facts interesting and useful.

### The Graphics Screen

When pointing straight up, the turtle can go 121 steps before wrapping around to the bottom of the screen. It can go 120 steps downward before wrapping around to the top. It can go 140 steps when pointing the the left, and 140 when going to the right. If you change the aspect ratio (see the .ASPECT primitive), then the allowable vertical range will change, but the horizontal range will remain the same.

The turtle-graphics screen is stored in memory in the primary high-resolution graphics page.

## Numbers

The smallest number on which Logo can perform operations is `IN38`, and the largest is `9.9999E38`. The largest positive number which is not “floating point” is `2147483647`, and the largest negative is `-214783647`.

## ASCII Values

There is a correspondence between the characters available in the Logo character set and the numbers 0-255. The `ASCII` primitive, if given a word of one letter, outputs the number associated with that letter. The `CHAR` primitive is the inverse, returning a single-letter word when given a positive integer. The character represented by 0 (often called “null”) is special in Logo: it represents the empty word. Just as `SENTENCE` ignores empty lists as input, `WORD` ignores the empty word. It is impossible to make a word which contains the empty word, unless that word is itself the empty word.

The `READCHARACTER` primitive, abbreviated `RC`, reads a key from the keyboard and outputs a single-letter word. There are certain “interrupt” keys that will never be output by `RC`. These are `<CTRL> F`, `<CTRL> S`, `<CTRL> T`, `<CTRL> SHIFT-M`, `<CTRL> W`, `<CTRL> Z`, and `<CTRL> G`. The functions these keys provide are available whenever Logo is in draw or nodraw mode.

The following table shows the ASCII values of selected keys. To find out the ASCII value of any key, type `PRINT ASCII RC`, and type the key.

---

<u>Key</u>	<u>ASCII Value</u>
ESC	27
DELETE	127
left arrow	8
right arrow	21
up arrow	11
down arrow	10
space bar	32
Return	13

Sometimes it is useful to be able to disallow <CTRL> G, or to use some interrupt character for purposes other than the function to which it is assigned. For these cases, Logo provides a hook for turning off the special meanings of all the above mentioned interrupt characters, except for <CTRL> Z and <CTRL> SHIFT-M. .DEPOSITing 1 in location NOINTP disables interrupt characters; .DEPOSITing 0 re-enables them. (Warning: .DEPOSITing 255 in the location will disable all interrupt characters.)

When interrupt characters have been disabled, the READCHARACTER primitive will output on any key pressed (except of course, <CTRL> Z and <CTRL> SHIFT-M). A typical use of this feature is a system like the INSTANT program included on the Logo Utilities Disk. The program could disable interrupt characters and assign its own meanings to the characters normally reserved for special immediate actions in Logo. Another occasion where disabling interrupts is useful is in procedures which do things which must be undone before returning to toplevel. If the user presses <CTRL> G during the execution of a procedure which temporarily changes OUTDEV to some other device, all output from then on (until another OUTDEV or <CTRL> SHIFT-M) would be directed to the alternate

device. The following procedure, which uses the NOINTP feature, can be executed without fear of causing “STOPPED!” or “PAUSE” messages to be sent to the device.

```
TO TCMD :CMD :ARG
  .DEPOSIT :NOINTP 255
  OUTDEV 7 ;device in slot 7
  (PRINT :CMD :ARG)
  OUTDEV 0
  .DEPOSIT :NOINTP 0
END
```

### Line Length

Lines typed into Logo in the line editor may not be more than 256 characters long (although longer lines can be created by concatenating names using the primitive WORD). Additionally, the list that is input to RUN and REPEAT, and each sub-list in the second input to DEFINE must abide by this restriction. In contrast, lines typed in the screen editor (using TO) may be of any length, as long as it fits in the edit buffer. Similarly, lines read in from disk files may be of any length. The edit buffer in *Terrapin Logo for the Apple* is 8192 characters long; the *Logo PLUS* editor will hold 4608 characters.

### Storage in Logo

Logo stores procedures much more efficiently than most other languages. Each procedure is stored as a list of lines. The lines are lists of other lists and words. Each word takes up the same amount of space every time it is used, no matter how many characters it has. Thus, there is almost no penalty for using long, descriptive procedure and variable names. (The first time a long

name is used, the amount of space it occupies is related to its length. Every occurrence thereafter occupies 2 bytes, which is a pointer to the complete name. Space can be saved, therefore, by using a limited vocabulary, especially when a lot of text is being stored. Use the procedures in the TEXTEDIT file to display text without using up workspace.)

(Actually, this implementation of Logo usually stores procedures as arrays of arrays, since that method takes half as much space; however, when there isn't enough contiguous memory, Logo uses the list-of-lists method. It is possible for the curious to tell how procedures are stored: If each line is indented one space, the procedure is stored in the array form. If not, it is stored in the rarer list form.)

When Logo runs out of storage space, it enters a process called garbage collection. This simply means that Logo is finding out what parts of memory are not being used, and makes a big list of all of them. Then, when Logo needs to use a memory location, it takes it off of this list. Since Logo can't do anything else (like run your procedures) when it is garbage collecting, the process can interfere with certain programs where real-time response is important. If this becomes annoying, place calls to the .GCOLL primitive at natural pauses in the program.

A

A

A

A

A

A

A

**A**PPENDIX

A

A

A

A

A

A

A

A

A

A

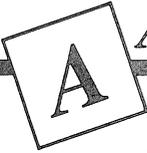
A

A

A

A

A



# APPENDIX

---

## *ERROR MESSAGES*

Error messages are Logo's way of trying to help the user find errors, those things which Logo does not understand. They may be misspellings or wrong usage.

The list of error messages, as given here, is divided. In both parts, capital letters indicate the unchanging part that Logo types to you; what is in the parentheses will vary depending on the circumstances. Both parts of the list are alphabetized according to the first unchanging word.

The first part of the list includes those messages that start with different words at different times; here you must look for that part of the message after the variable portion.

The second part includes the messages that always start out the same way.

For *Logo PLUS* error messages, consult the *Getting Acquainted with Logo PLUS* booklet for correct usage of the commands that relate to the messages you receive.

### PART I

(procedure) DIDN'T OUTPUT

Example:

```
FORWARD SQUARE 5  
SQUARE DIDN'T OUTPUT
```

```
SQUARE FD 100  
FD DIDN'T OUTPUT
```

This occurs when Logo cannot find the input for some-

thing that requires an input, either a procedure or a primitive. (It looks at the next object on the line and complains that it didn't get anything from that object to use as an input. To produce something that could be used as an input, that object would have to output it.)

This message usually means that you forgot to type the input Logo was looking for.

(primitive) DOESN'T LIKE (data) AS INPUT

Example:

```
PRINT 5 * "SIDE
* DOESN'T LIKE SIDE AS INPUT
```

This occurs when you try to do an operation on the wrong type of data. Here Logo is trying to multiply a name (specified by ") instead of the value of name (specified by :).

(primitive) DOESN'T LIKE (data) AS INPUT.
IT EXPECTS TRUE OR FALSE

Example:

```
IF 2 THEN PRINT 5
IF DOESN'T LIKE 2 AS INPUT. IT EXPECTS
TRUE OR FALSE
```

The primitives IF, NOT, ALLOF and ANYOF expect only expressions which will evaluate to TRUE or FALSE, such as :X = 3 (which is either true or false). You probably neglected to type the rest of the test.

(message) , IN LINE  
(line)  
AT LEVEL (level) OF (procedure name)

Example:

THERE IS NO PROCEDURE NAMED FD100, IN LINE  
FD100  
AT LEVEL 1 OF SQUARE

The (message) here is another error message, with this larger message pinpointing the location of the error, by printing the line, level and procedure in which it occurred.

(name) IS A LOGO PRIMITIVE

Example:

FIRST IS A LOGO PRIMITIVE

Logo reserves the words that are Logo primitives and does not allow them to be used as procedure names. Choose another name for your procedure.

(procedure) NEEDS MORE INPUTS  
(primitive) NEEDS MORE INPUTS

Examples:

SQUARE NEEDS MORE INPUTS  
FD NEEDS MORE INPUTS

SQUARE required more inputs than were used; FD was used without an input. With a procedure, this can happen when the second input is negative and is used without parentheses. The parentheses are necessary to distinguish a second input from a single input obtained by subtracting the second number from the first.

(arithmetic-operator) NEEDS SOMETHING BEFORE IT

Example:

```
PRINT / 8
/ NEEDS SOMETHING BEFORE IT
```

The number to be divided by 8 is omitted.

(primitive) SHOULD BE USED ONLY INSIDE A  
PROCEDURE

Example:

```
OUTPUT SHOULD BE USED ONLY INSIDE A  
PROCEDURE
```

OUTPUT, STOP and GO cannot be used in immediate mode (top level). They have meaning only in a procedure.

## PART II

CAN'T DIVIDE BY ZERO

Example:

```
PRINT :X / :Y
CAN'T DIVIDE BY ZERO
```

:Y is (no doubt inadvertently) zero. This message occurs with QUOTIENT, REMAINDER and /. Get around this by testing :Y to see if it is zero before the division.

DIRECTORY NOT EMPTY (*Logo PLUS only*)

You cannot erase a directory that still contains programs or pictures. You will need to erase the files

before you can erase the directory. (You can first move them to another directory with the COPY command.)

DIRECTORY NOT FOUND (*Logo PLUS only*)

You have used a pathname that includes a directory name that is not on the current volume. Use SETPREFIX or SETDISK to switch to another volume, or check your spelling.

DIRECTORY EXISTS (*Logo PLUS only*)

You have tried to create a directory using a name of a directory that already exists. Choose another name for your new directory.

DISK ERROR

Example:

```
CATALOG  
CATALOG [] - - DISK ERROR
```

You will get this message when you try to access a disk with no disk in the disk drive, or try to SAVE to a protected or unformatted disk. Also, the Language Disk files cannot be listed with CATALOG.

DISK IS FULL

Example:

```
SAVE "NEWMUSIC  
THE DISK IS FULL
```

When the disk is full and Logo will not save your workspace, you have several choices:

1. You can type CATALOG, locate a file you no longer

need, and erase it with ERASEFILE. (Erase pictures with ERASEPICT.)

2. You can use another disk that is not full.
3. You can trim the amount you are saving by erasing procedures from your workspace with ERASE.



You will also receive a DISK FULL message if you try to save *Terrapin Logo* programs to a disk that has been formatted for ProDOS. *Terrapin Logo* is based on DOS 3.3 and data disks must be formatted for that operating system to use. *Logo PLUS* is ProDOS-based. See the Beginning in Logo chapter for instructions for properly formatting a blank disk for your Logo files.

DISK IS WRITE PROTECTED

You tried to write on a write-protected disk. This might mean you forgot to remove the Language Disk. A 5.25" disk is write-protected if the notch on the left side of the disk (as it enters the disk drive) is covered.

END SHOULD BE USED ONLY IN THE EDITOR

Example:

```
PRINT 5 END
END SHOULD BE USED ONLY IN THE EDITOR
```

You have done one of these things:

1. tried to use END in IMMEDIATE mode
2. put END on a line with something else in a procedure
3. put it in the list used by the Logo primitive DEFINE.

ELSE IS OUT OF PLACE

Example:

```
PRINT 5 ELSE PRINT :C
ELSE IS OUT OF PLACE
```

ELSE has no meaning in this context. It must be used in an IF...THEN...ELSE statement.

FILE IS LOCKED

Example:

```
ERASEFILE "MUSIC  
THE FILE IS LOCKED
```

Locking a file is a way to protect it from inadvertent erasing. However, do not lock a file you will be changing; you can READ from locked files, but you cannot SAVE to them.

The file cannot be erased while it is locked. To unlock a file on a disk, use FID on the Logo Utilities Disk or the primitive DOS, in an instruction like:

```
DOS [UNLOCK MUSIC.LOGO]
```

Type CATALOG to list the files on the disk. Files with \* before them are locked.

FILE NOT FOUND

Example:

```
READ "NUSIC  
FILE NOT FOUND
```

NUSIC is not a file on the disk in the disk drive. (The file MUSIC might be.) Check your spelling before despairing. Type CATALOG to see what IS on the disk.

If you are using *Logo PLUS*, make sure that the prefix is set to the directory containing the file you want to use.

**FILE STILL OPEN** (*Logo PLUS only*)

You have tried to open, rename or erase a file that is already open. Use the command **CLOSE** to close it.

**INVALID FILE NAME** (*Logo PLUS only*)

You have tried to save a file using a name that contains invalid characters. Filenames must begin with a letter and can include only letters (A-Z), numbers (0-9) and periods (.). Try saving the file again with a name that does not contain other characters. You will also receive this message if you are using *Logo PLUS* and try to catalog a DOS 3.3-formatted disk.

**INVALID SHAPE NUMBER** (*Logo PLUS only*)

You have tried to set the turtle's shape to one that has not yet been defined or with a number greater than 100. Use the number of a defined shape, or perhaps read a different shapes file from the disk.

**LINE GIVEN TO DEFINE TOO LONG**  
**LINE GIVEN TO REPEAT TOO LONG**  
**LINE GIVEN TO RUN TOO LONG**

You have exceeded the maximum length of a line used by **DEFINE**, **REPEAT** or **RUN**, which is 256 characters.

**MISSING INPUTS INSIDE ()'S**

Example:

(FORWARD)  
**MISSING INPUTS INSIDE ()'S**

The procedure or primitive in the () was used with too few inputs.

**NO PICTURE TO PRINT (*Logo PLUS only*)**

You have given the PRINTSCREEN command, but have no picture on the graphics screen to print. You receive this message, rather than having to wait for the printer to go through the motions of printing a blank screen.

**NO SHAPES AVAILABLE (*Logo PLUS only*)**

You have tried to change the turtle's shape, but no shapes are available. Try using READSHAPES to read some from the disk or create some using EDSHAPE.

**NO STORAGE LEFT!**

You have used up all the storage. The exclamation mark means that this is very unusual. Erase some unnecessary procedures. If this doesn't help, SAVE the workspace and type GOODBYE.

**NUMBER TOO LARGE OR TOO SMALL**

An arithmetic operation has resulted in a number too large or too small for Logo, i.e. greater than  $10^{38}$  or  $10^{-38}$ . Use different numbers.

**PROCEDURE NESTED TOO DEEP**

You have exceeded the limit for nesting procedures (which is over 200). This will be rare. Send a copy of your procedure to Terrapin, Inc. for its museum of the unusual.

RESULT: (data)

Example:

12 \* 10

RESULT: 120

Besides giving you a quick way to calculate, Logo is also telling you that you have not specified what is to be done with the results of the computation. This is important to note if you are intending to use the line in a procedure.

SHAPE TABLE FULL (*Logo PLUS only*)

The area reserved for user-defined shapes is full, causing you to lose the shape that was being defined at the time. When you define a new shape, you will see a message telling you how much space remains. Careful use of this information will help prevent you from completely filling the shape table space.

STACK OVERFLOW (*Logo PLUS only*)

This message will occur if you run out of stack space, where Logo keeps copies of procedures as they are created during the running of recursive programs. You may have unwittingly placed the recursive call to the procedure other than immediately before the END statement. Read about recursion to determine if your recursive procedure is written properly. If you have in fact written the procedure correctly, you have run out of stack space and should consider alternate methods of achieving the results you want. This message might also occur if you are trying to edit a shape that is too large to be defined (very unlikely to happen).

THE : IS OUT OF PLACE AT (something)

Example:

```
PRINT X:  
THE : IS OUT OF PLACE AT X
```

The : has no meaning in this position. Logo realizes that PRINT expects an input, and sees the : which, in the right place, denotes a variable. You probably meant :X.

THEN IS OUT OF PLACE

Example:

```
PRINT 5 THEN PRINT 6  
THEN IS OUT OF PLACE
```

THEN has no meaning in this context. THEN must be used in an IF...THEN statement.

THERE IS NO LABEL (whatever you used)

Example:

```
THERE IS NO LABEL QUAD
```

You have used GO to a label that is not specified in the procedure. You can avoid this by not using GO. To fix it, add the missing label to your procedure.

THERE IS NO NAME (whatever you used)

Example:

```
PRINT :X  
THERE IS NO NAME :X
```

X has not been defined, or is used only in a procedure

and is local to it. This will also occur if you forget to list the variables in the title line of a procedure.

THERE IS NO PROCEDURE NAMED (what you typed)

Example:

THERE IS NO PROCEDURE NAMED FD100

When Logo does not recognize a primitive, it looks for a procedure name. Mistyping accounts for most instances of this message; forgetting to read in the file is another possibility.

TOO MANY PROCEDURE INPUTS

You have exceeded the limit on procedure inputs (which is over 200). This will be exceedingly rare. Send a copy of the procedure that generated this message to Terrapin, Inc. for its museum.

TOO MUCH INSIDE PARENTHESES

Logo uses this when it cannot figure out some parenthesized expression. Interior parentheses may be incorrectly placed, or the number of opening and closing parentheses may not be the same.

TURTLE OUT OF BOUNDS

Example:

FD 200  
TURTLE OUT OF BOUNDS

In NOWRAP mode, the turtle would go off the screen if it moved, so it doesn't move.

**UNRECOGNIZED DOS COMMAND**  
*(Terrapin Logo only)***Example:**

```
DOS [TICKLE]
[TICKLE] -- UNRECOGNIZED DOS COMMAND
```

Either the file name specified in a **READ** or **SAVE** command is invalid, or an invalid command was used with the Logo DOS primitive.

**VOLUME NOT FOUND** *(Logo PLUS only)*

You have used a pathname that includes the name of a volume that is not currently in a disk drive. Use the **ONLINE** command to see which volumes are available. If you begin a pathname with a slash, *Logo PLUS* assumes the first name to be that of a volume name. Remove this initial slash if this is not the case.

**YOU DON'T SAY WHAT TO DO WITH** *(data)***Example:**

```
YOU DON'T SAY WHAT TO DO WITH 25, IN LINE
:SIDE * :SIDE
AT LEVEL 1 OF SQUARE
```

The line is missing a command such as **OUTPUT**, **PRINT**, **FORWARD**, etc. This corresponds to **RESULT:** in immediate mode. Add the missing instruction (in the example, perhaps **PRINT**) to the line.

## ***EDIT MODE***

### ***USE OF CONTROL CHARACTERS FOR EASE IN EDITING***

The EDIT mode discussion in the Technical chapter includes a listing of the keyboard editing commands.

The <CTRL> key is used like the <SHIFT> key. Hold it down while you type the character indicated. (<CTRL> N: hold down <CTRL> and type <N>.)

### ***Moving the Cursor***

These commands move the cursor without changing the text.

**Arrow Keys** The Left Arrow moves the cursor to the left, and, if it is at the beginning of a line, up to the end of the previous line.

The Right Arrow moves the cursor to the right, and, if it is at the end of a line, down to the beginning of the next line.

The Up Arrow moves the cursor up to the line above. (Apple II+ owners use <CTRL> P.)

The Down Arrow moves the cursor down to the next line. (Apple II+ owners use <CTRL> N.)

<CTRL> A Moves the cursor to the beginning of the line.

- <CTRL> E END:** Moves the cursor to the end of the line.
- <CTRL> F FORWARD:** When editing more than one screenful of text, moves the cursor one screenful forward, or to the end of the buffer, whichever comes first.
- <CTRL> B BACK:** When editing more than one screenful of text, moves the cursor one screenful back, or to the beginning of the buffer, whichever comes first.
- <RETURN>** typed at the end of a line: moves the cursor to the next line.

### *Moving the Text*

These commands move the text without changing it or changing the position of the cursor in the text.

- <RETURN>** typed in the middle of a line: moves the cursor and the text after it on the line to the next line.
- <CTRL> O OPEN:** Opens a new line at the cursor position. The cursor remains on the open line. Equivalent to typing **<RETURN>**
- <CTRL> P.** Use it to add new lines in the middle of a procedure.
- <CTRL> L** Scrolls the text so that the line with the cursor is in the middle of the screen. Useful for seeing a particular sequence completely on the screen.

## *Deleting Text*

These commands delete text. Deleted text is not recoverable. When text is deleted within a line, the rest of the line moves to the left.

- <DEL>** Each stroke of the **<DEL>** (**<ESC>**) key deletes the character to the left of the cursor. **<DEL>** used at the beginning of the line deletes the **<RETURN>** from the previous line, and joins the two lines.
- <CTRL> D** **DELETE**: Deletes the character under the cursor. When used at the end of a line **<CTRL> D** deletes the **<RETURN>**.
- <CTRL> X** Deletes from the cursor to the end of the line. If the cursor is at the beginning of the line, **<CTRL> X** kills the whole line.

## *Restoring Text*

- <CTRL> Y** **YANK**: Recalls the most recently deleted line. This feature is especially useful when you want to move a line to a different location in the procedure. Also, if you are writing a procedure with several similar lines, try this trick: type the first line, delete it, and then restore it several times. Then you can make the minor changes necessary in each line.

### *Leaving EDIT Mode*

- <CTRL> C COMPLETE:** Exits EDIT mode with changes intact. Use it when you complete a procedure or changes to a procedure.
  
- <CTRL> G GONE:** Exits EDIT mode without making any changes to your procedure. Use it when you change your mind about making changes or have just done a lot of typing without realizing you were still in EDIT mode.

When using Logo as a text editor, **<CTRL> G** is the only way to exit from the editor.

## *Procedures*

### *GRAPHICS CHAPTER*

#### *Turtle Driving Projects*

1. through 4. Screen size:

Hint: type <CTRL> F to see when the whole turtle goes off the edge and appears at the other edge of the screen. Type <CTRL> T to see the whole list of numbers (distances). Add up the numbers (or tell Logo to: 100 + 50 + ...), type DRAW and type FD (the total number) to check it out. You could also say FD 100 + 50 + ..., but you would not know what it totalled.

3. and 4. Diagonals:

To get to the first corner: use half the distance across (from 2) to get to the edge, and half the distance down to get to the bottom. Write down this list of instructions in case you do not get the true diagonal on the first try. Then aim the turtle at the opposite corner.

5. Command with a negative number and the equivalent:

Try FD -20 and BK 20

6. Square examples (Type <RETURN> only where indicated):

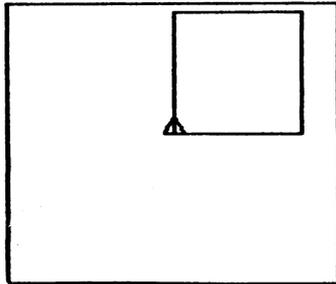
```
1) FD 100    RT 90 <RETURN>
   <CTRL> P  <SPACE> <CTRL> P
             <SPACE> <CTRL> P <RETURN>
```

Type as one line

```
2) FD 100 RT 90 <RETURN>
   FD 100 RT 90 <RETURN>
   FD 100 RT 90 <RETURN>
   FD 100 RT 90 <RETURN>
```

```
3) FD 100 RT 90 <RETURN>
   <CTRL> P <RETURN>
   <CTRL> P <RETURN>
   <CTRL> P <RETURN>
```

```
4) FD 100 RT 90 FD 100 RT 90 FD 100
   RT 90 FD 100 RT 90 <RETURN>
```



Square

### Rectangle examples:

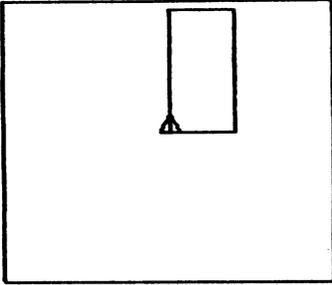
```
1) FD 100 RT 90 FD 50 RT 90 <RETURN>
   <CTRL> P <RETURN>
```

(Why does it take only one repetition for the rectangle but three for the square?)

```
2) FD 100 RT 90 <RETURN>
   FD 50 RT 90 <RETURN>
   FD 100 RT 90 <RETURN>
   FD 50 RT 90 <RETURN>
```

Type as one line, with only the one <RETURN>:

```
3) FD 100 RT 90 FD 50 RT 90 FD 100  
   RT 90 FD 50 RT 90 <RETURN>
```

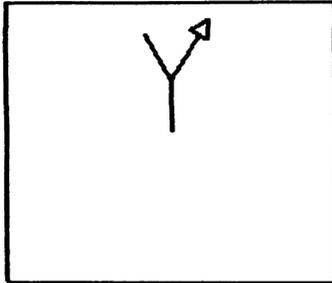


Rectangle

These instructions leave the turtle in its starting position, which is a very good idea. Keep it in mind when you write procedures. It makes it easier to plan how one procedure follows another when you want to use several, as in drawing something that requires both a square and a triangle.

7. Some straight line initials (<RETURN> is assumed after each line):

```
L:  LT 90 FD 50 RT 90 FD 100  
I:  FD 100  
V:  LT 15 FD 100 BK 100 RT 30 FD 100  
T:  FD 100 LT 90 FD 25 BK 50  
Y:  FD 50 LT 30 FD 50 BK 50 RT 60 FD 50
```



Initial Y

These instructions leave the turtle at the end of the initial. Later the tutorial will tell you how to move the turtle without leaving a track. (See section which includes PENUP and PENDOWN.)

### *Procedure Projects*

#### 1. Trackless SETUP:

```
TO SETUP
DRAW
PU
LT 90
FD 140
RT 90
BK 110
PD
FULLSCREEN
END
```

gives the same final result as

```
TO SETUP
  DRAW
  LT 90
  FD 140
  RT 90
  BK 110
  CS
  FULLSCREEN
END
```

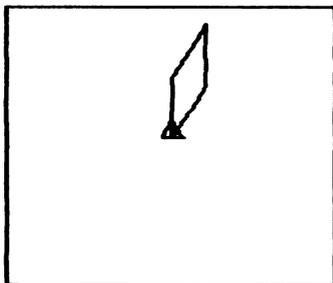
Use PU / PD to avoid having to get rid of the track.

2. Design with MOVE repeated:

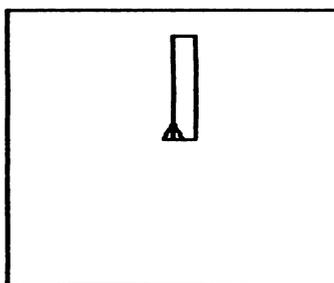
```
TO MOVE          TO MOVEIT
  FD 100         REPEAT 24 [MOVE]
  RT 15         END
  BK 80
  RT 25
END
```

3. A four-sided figure:

```
TO FOURSIDE
  REPEAT 2 [FD 60 RT 30 FD 60 RT 150]
END
```



FOURSIDE



RECT1

4. Rectangles:

```
TO RECT
  REPEAT 2 [FD 100 RT 90 FD 50 RT 90]
END
```

```
TO RECT1
  REPEAT 2 [FD 110 RT 90 FD 10 RT 90]
END
```

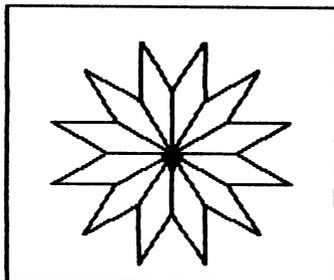
5. Setup and a rectangle:

```
SETUP          SETUP
RECT           RECT1
```

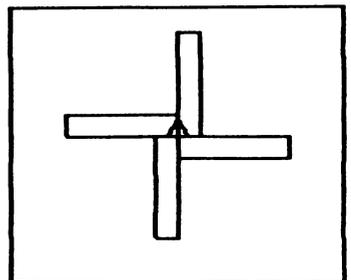
6. REPEAT, a shape, and a turn:

```
TO HOTPAD
  REPEAT 12 [FOURSIDE RT 30]
END
```

```
TO WINDMILL
  REPEAT 4 [RECT1 RT 90]
END
```



HOTPAD



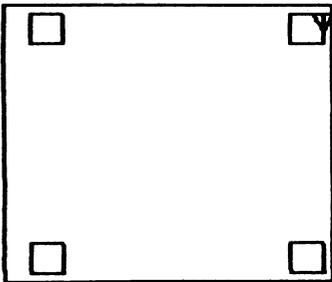
WINDMILL

### *Projects Using Shapes*

1. A square in each corner of the screen:

```
TO CORNER.SQ      TO SETUP
  SETUP          PU
  SQUARE        LT 90
  PU            FD 140
  FD 200        RT 90
  PD            BK 110
  SQUARE        PD
  PU            END
  RT 90
  FD 220
  LT 90
  PD            TO SQUARE
  SQUARE        REPEAT 4 [FD 30 RT 90]
  PU            END
  BK 200
  PD
  SQUARE
END

TO FOUR.SQ
  SETUP
  REPEAT 4 [SQUARE PU FD 230 RT 90 PD]
END
```



CORNER.SQ

Note how in the first version, the turtle walks around the screen getting to the location of the closest corner, while in the second version it starts each square from the corner. It is always more elegant and more understandable if you can figure out a pattern and repeat it.

2. Keeping that in mind, let's see what would draw a square and place the turtle in position to draw another in a row.

SQUARE RT 90 FD 30 LT 90 would do it,

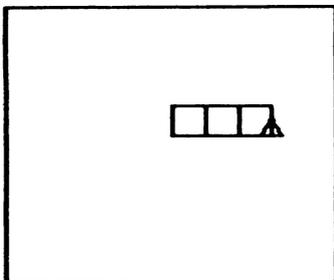
```
TO ROW.SQUARE
  REPEAT 3 [SQUARE RT 90 FD 30 LT 90]
END
```

and, if the turtle turned LT 90 first, so would

SQUARE FD 30

```
TO ROW.SQUARE.LEFT
  LT 90
  REPEAT 3 [SQUARE FD 30]
END
```

Lengthening the distance forward would produce a row of separated squares.

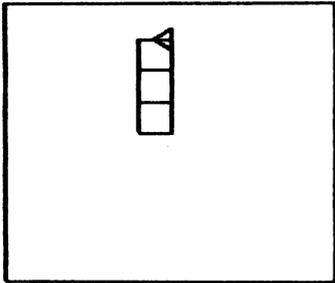


ROW.SQ

3. Tower of squares:

```
TO SQUARE.TOWER  
  LT 90  
  ROW.SQUARE  
END
```

```
TO SQUARE.TOWER.LEFT  
  RT 90  
  ROW.SQUARE.LEFT  
END
```



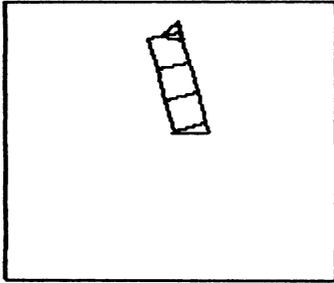
SQUARE.TOWER

4. A leaning tower:

```
TO LEANING.TOWER  
  BASE  
  SQUARE.TOWER  
END
```

```
TO BASE  
  RT 90  
  FD 30  
  LT 105  
  FD 10  
END
```

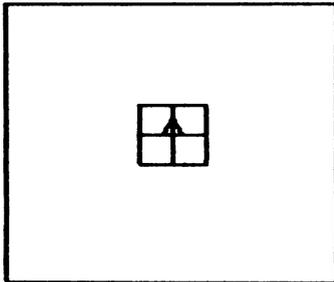
Discover the distances in a procedure like BASE by trying different ones.



LEANING.TOWER

5. A window with four panes:

```
TO WINDOW  
  REPEAT 4 [SQUARE LT 90]  
END
```



WINDOW

## 6. Square

```
1) TO SQ2
    FD 30
    RT 90
    FD 30
    RT 90
    FD 30
    RT 90
    FD 30
    RT 90
END
```

```
2) TO SQ3
    REPEAT 4 [FD 30 LT 90]
END
```

## 7. Analyzing the problem of drawing a triangle:

Decisions (as described in the text):

1. Sides will be 30 steps.
2. Have to try a few different numbers for the turn
3. Want 3 sides

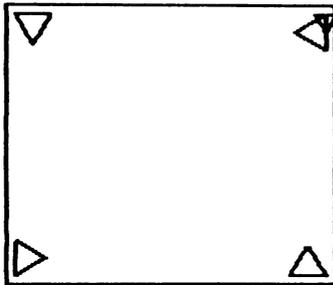
```
TO TRI
    REPEAT 3 [FD 30RT 120]
END
```

## 8. 1—4 using triangles:

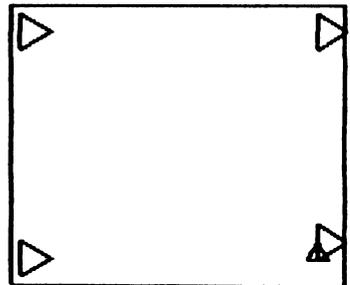
(1) A triangle in each corner of the screen: Substitute the triangle procedure for the square procedure (and change the name):

```
TO FOUR.TRI
  SETUP
  REPEAT 4 [TRI PU FD 230 RT 90 PD]
END
```

```
TO CORNER.TRI
  SETUP
  TRI
  PU
  FD 200
  PD
  TRI
  PU
  RT 90
  FD 220
  LT 90
  PD
  TRI
  PU
  BK 200
  PD
  TRI
END
```



FOUR.TRI



CORNER.TRI

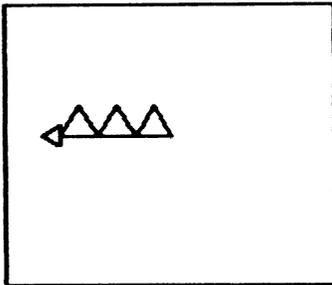
Notice that the two procedures produce different results with triangles. The orientation of a triangle makes a difference.

(2) A row of triangles:

Turn LT 90 (or RT 30) first to lay the triangle down to make it easier to connect the triangles.

```
TO ROW.TRI
  LT 90
  REPEAT 3 [TRI FD 30]
END
```

```
TO ROW.TRI.RIGHT
  RT 30
  REPEAT 4 [TRI RT 60 FD 30 LT 60]
END
```



ROW.TRI

In the first, the turtle is heading in the direction of the first side when it starts out. In the second, it has to turn each time to head in the right direction. Which is easier to understand? Try to make your procedures as simple as possible.

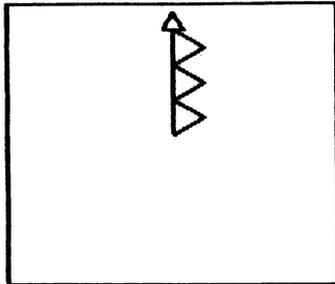
(3) A tower of triangles:

There are several choices:

- 1) Turning the row of triangles will produce a tower of triangles balancing on their points.
- 2) Drawing another row, fitted into the first, will produce a tower with triangles pointing in opposite directions, either balanced on a point,

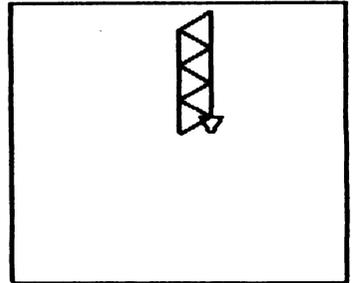
- 3) or with a base.  
 4) Drawing triangles with each base balanced on the point of the one below requires a new procedure.

```
TO TRI.TOWER1
  RT 90
  ROW.TRI
END
```



TRI.TOWER1

```
TO TRI.TOWER2
  RT 90
  ROW.TRI
  RT 60
  FD 30
  RT 120
  RT 90
  ROW.TRI
END
```

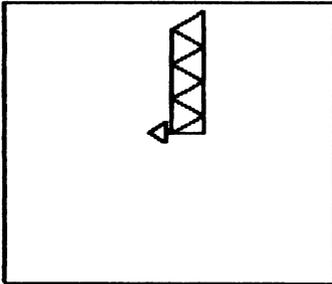


TRI.TOWER2

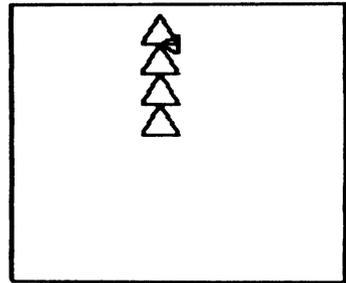
```
TO TRI.TOWER3: Add to 2) (before END)
  FD 15
  RT 90
  FD 30
```

FD 30 is slightly too long. Adjust it by trial.

```
TO TRI.TOWER4
  LT 90
  REPEAT 3 [TRI RT 60 FD 30 LT 60 BK 15]
  TRI
END
```



TRI.TOWER3



TRI.TOWER4

(The REPEAT statement must be typed as one line, with only one <RETURN>, at the end.) Note that the turtle draws the triangle, turns and moves to the top, then turns again and backs into position to draw the next one.

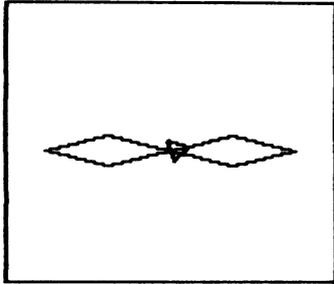
(4) A leaning tower of triangles:  
Turn turtle and draw either ROW.TRI,  
TRI.TOWER or TRI.TOWER2.

### 9. Designs using FOURSIDE:

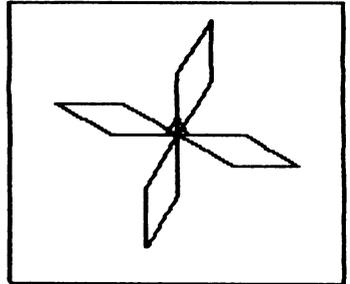
NOTE: These designs were named after they were drawn.

- 1) TO PROPELLER  
    REPEAT 2 [FOURSIDE RT 180]  
    END
- 2) TO BOW.TIE  
    LT 105  
    REPEAT 2 [FOURSIDE RT 180]  
    END
- 3) TO TRI.PROP  
    REPEAT 3 [FOURSIDE RT 120]  
    END

- 4) TO PINWHEEL  
 REPEAT 4 [FOURSIDE RT 90]  
 END



BOWTIE

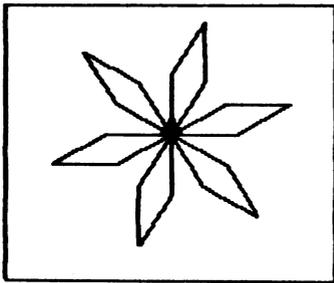


PINWHEEL

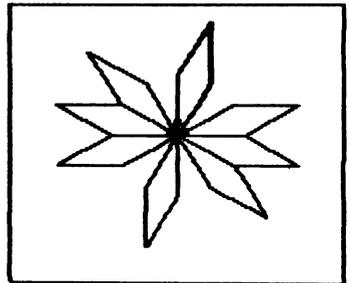
- 5) TO FIVE  
 REPEAT 5 [FOURSIDE RT 72]  
 END

- 6) TO SUPER.PINWHEEL  
 REPEAT 6 [FOURSIDE RT 60]  
 END

- 7) TO BIRD  
 PINWHEEL  
 SUPER.PINWHEEL  
 END



SUPER.PINWHEEL

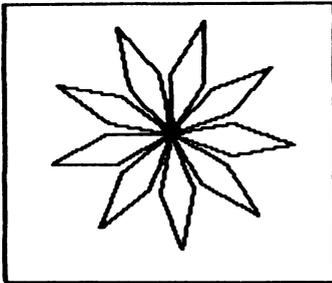


BIRD

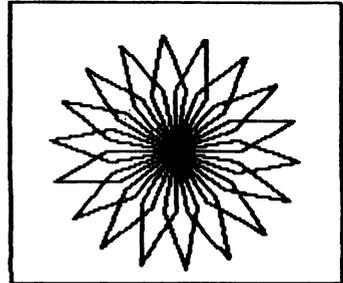
8) TO FLEUR  
REPEAT 9 [FOURSIDE RT 40]  
END

9) TO HOTPAD  
REPEAT 12 [FOURSIDE RT 30]  
END

10) TO FLOWER  
REPEAT 18 [FOURSIDE RT 20]  
END



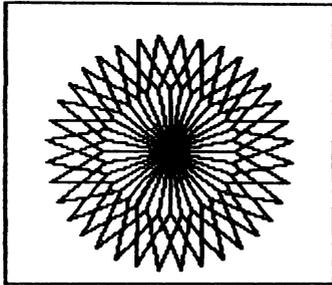
FLEUR



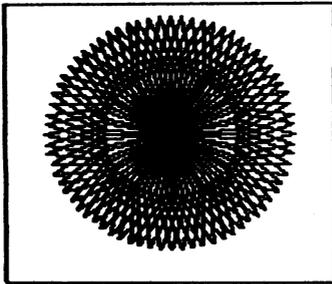
FLOWER

11) TO MUM  
HT  
REPEAT 36 [FOURSIDE RT 10]  
END

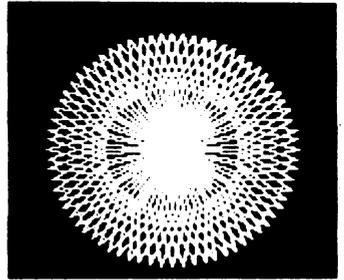
12) TO SUN  
HT  
REPEAT 72 [FOURSIDE RT 5]  
END



MUM



SUN



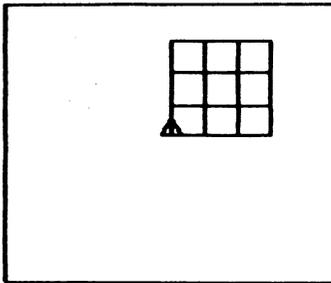
SUN

Except for BIRD, these are all essentially the same procedure, with a different turn. But see what different designs they are! HT (HIDETURTLE) makes the drawing go faster.

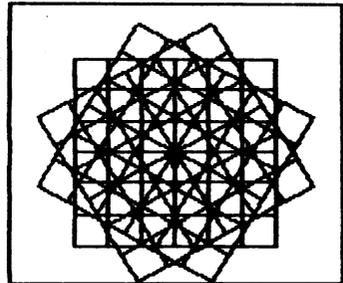
Progressively more complicated designs:

Using ROW.SQ:

- 1) TO NINE  
HT  
REPEAT 4 [ROW.SQ LT 90]  
END
- 2) TO LACE  
HT  
REPEAT 12 [NINE RT 30]  
END



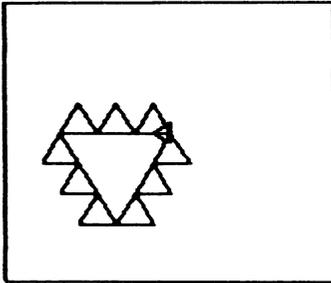
NINE



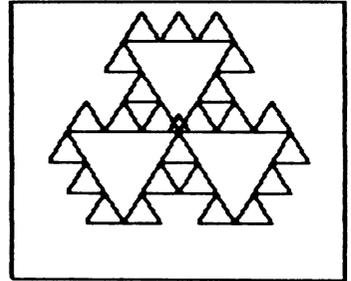
LACE

Using TRI.TOWER:

- 1) TO JAG.TRI  
LT 90  
REPEAT 3 [TRI.TOWER1 LT 120]  
END
- 2) TO JAG3  
REPEAT 3 [JAG.TRI LT 30]  
END



JAG.TRI



JAG3

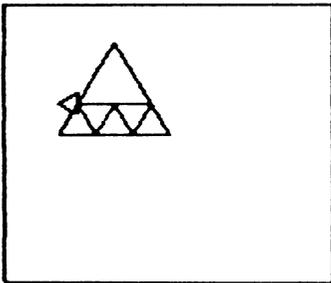
10. A window with 6 triangular panes:

```

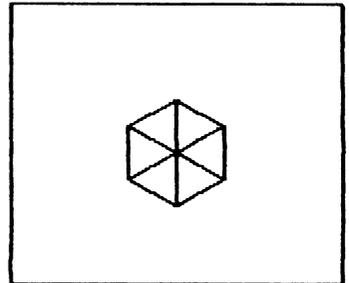
TO TRI.WINDOW
  ROW.TRI
  RT 120
  FD 90
  REPEAT 2 [RT 120 FD 60]
END
    
```

```

TO TRI.WINDOW2
  REPEAT 6 [TRI RT 60]
END
    
```



TRI.WINDOW



TRI.WINDOW2

11. Some triangle procedures:

```
TO TRI
  FD 30
  RT 120
  FD 30
  RT 120
  FD 30
  RT 120
END
```

```
TO TRI2
  REPEAT 3 [FD 30 LT 240]
END
```

***Projects: More Shapes***

1.—3. Using REPEAT and division:

1) A square

```
TO SQ1
  REPEAT 4 [FD 30    RT 360/4]
END
```

2) A triangle

```
TO TRI3
  REPEAT 3 [FD 30    RT 360/3]
END
```

3) A pentagon (5 sides)

```
TO PENTA
  REPEAT 5 [FD 30    RT 360/5]
END
```

4) A hexagon (6 sides)

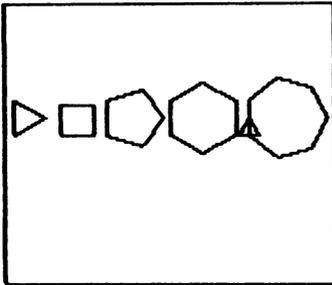
```
TO HEXA
  REPEAT 6 [FD 30    RT 360/6]
END
```

## 5) A septagon (7 sides)

```

TO SEPTA
  REPEAT 7 [FD 30    RT 360/7]
END

```



Polygons

## 6) A pentadecagon (15 sides)

```

TO FIFTEEN
  REPEAT 15 [FD 30    RT 360/15]
END

```

*Projects: Sizable Shapes*

## 1. SQUARE4 to draw squares of various sizes:

```

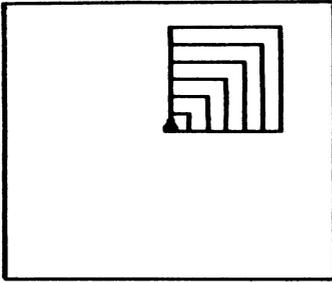
TO SQUARE4
  SQV 10
  SQV 20
  SQV 30
  SQV 40
END

```

```

TO SQV :LENGTH
  REPEAT 4 [FD :LENGTH RT 90]
END

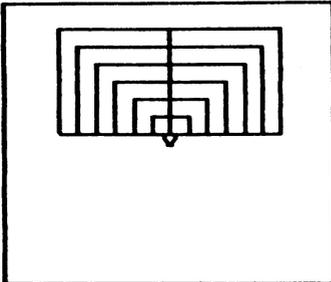
```



SQUARE4

2. Another set of squares beside the first:

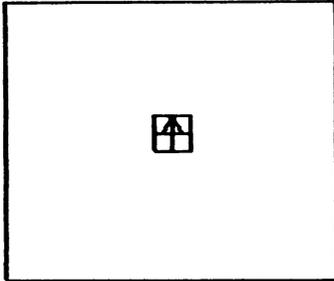
```
TO TWO.SQUARES  
  SQUARE4  
  LT 90  
  SQUARE4  
END
```



TWO SQUARES

3. A procedure using a specific size square:

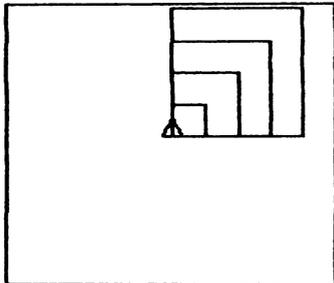
```
TO WINDOW1  
  REPEAT 4 [SQV 30 RT 90]  
END
```



WINDOW1

4. 4 squares, each 25 bigger than the last, with size of the first square input:

```
TO BIGGER.SQ :LENGTH
  SQV :LENGTH
  SQV :LENGTH + 25
  SQV :LENGTH + 50
  SQV :LENGTH + 75
END
```

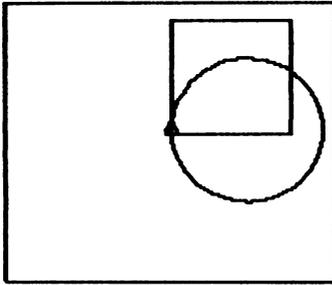


BIGGER.SQ

### *Projects with Regular Polygons*

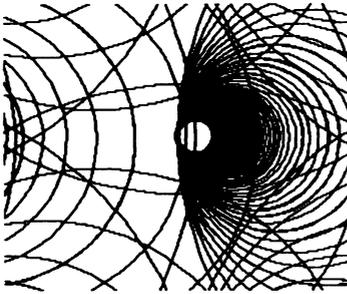
1. POLY 4 100 and POLY 100 4:

```
TO POLY :LEN :TURNS
  REPEAT :TURNS [FD :LEN RT 360/:TURNS]
END
```



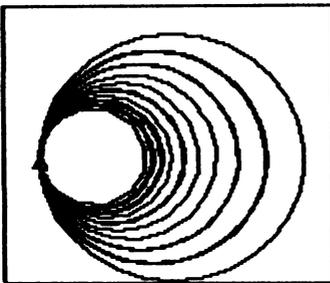
POLY 4 100 and POLY 100 4

2. POLY with the same :LEN and varying :TURNS:



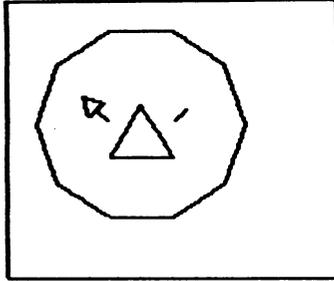
POLY: Same :LEN, varying :TURNS

3. POLY with the same :TURNS and varying :LEN:



POLY: Same :TURNS, varying :LEN

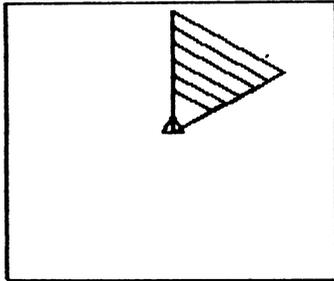
## 4. POLY twice, with different :TURNS:



Using POLY Twice with Different :TURNS

## 5. Using POLY to make a triangle:

POLY 100 3



POLY Triangles

## 6. The largest number you can use for :TURNS:

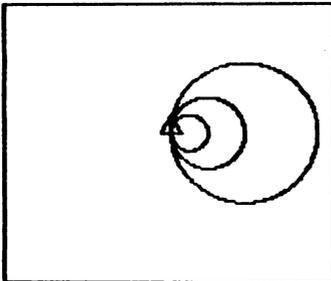
There is no largest number ... The figure becomes a rough circle at 15, and after that, larger numbers increase the exactness of the curve, but after a while there is no more visible improvement and the only effect is to make the turtle go more slowly and the circle to get larger (with the same length of side). Monitors do not have a high enough resolution to distinguish between a many-sided figure and a circle. The only reason you might want to be that exact (and slow)

would be for printing the designs on paper. The designs shown in the tutorial were drawn with the turn indicated in the procedures with them. The mascots (rabbit, elephant, and snail) were drawn with slower arc procedures for better resolution.

***Projects: Curves***

1. Circles: 2nd with step twice as big, 3rd with turn twice as big.

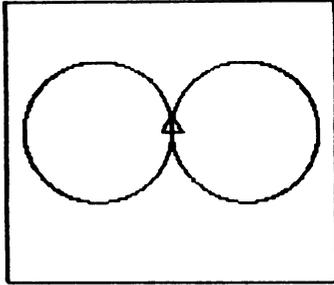
```
DRAW  
REPEAT 360 [FD 1 RT 1]  
REPEAT 360 [FD 2 RT 1]  
REPEAT 180 [FD 1 RT 2]
```



Circles

2. Circles to right and left:

```
DRAW  
REPEAT 360 [FD 1 RT 1]  
REPEAT 360 [FD 1 LT 1]
```



Circles Left and Right

3. To figure out the diameter (distance across) of a circle, turn the turtle 90 and walk it across. You can see the line better if you type HT (HIDETURTLE).

4. Quarter-circle arc to the right (make it into a procedure and call it ARCR90):

```
REPEAT 360/4 [FD 1 RT 1]
```

5. Quarter-circle arc with steps twice as big:

```
REPEAT 360/4 [FD 2 RT 1]
```

6. Sixth-of-a-circle arc to the left and right (make them into procedures and call them ARCR60 and ARCL60):

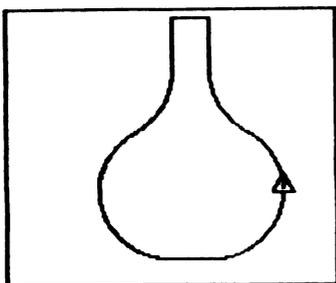
```
REPEAT 360/6 [FD 1 LT 1]
```

```
REPEAT 360/6 [FD 1 RT 1]
```

7. A procedure which uses an arc procedure and straight lines:

TO VASE

```
PU
RT 90
FD 60
LT 90
BK 30
PD
HT
ARCL60
ARCR60
FD 30
LT 90
FD 20
LT 90
FD 30
ARCR60
ARCL60
ARCL90
FD 20
ARCL90
END
```

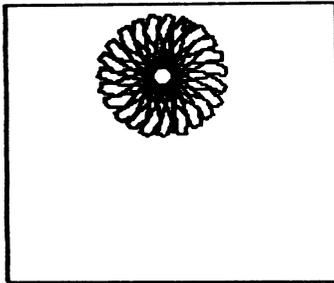


VASE

## *Projects: Simple Recursion*

1. A recursive procedure that draws a little figure, then calls itself:

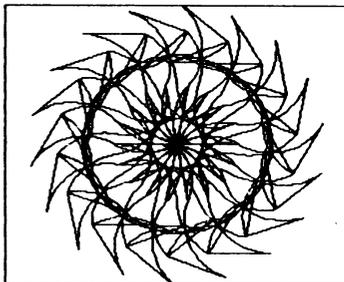
```
TO FIGURE
  FD 60 RT 49 FD 10 RT 80 FD 5 RT 90
  FIGURE
END
```



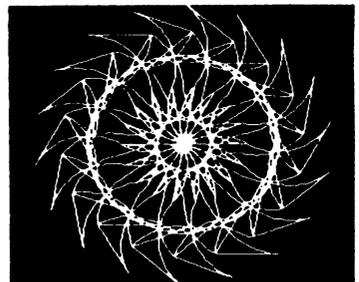
FIGURE

2. A recursive procedure that uses arcs and lines:

```
TO FAN
  PU
  RT 20
  PD
  REPEAT 3 [ARCR 50 60 ARCL 50 90 BK 50 LT 90]
  FAN
END
```



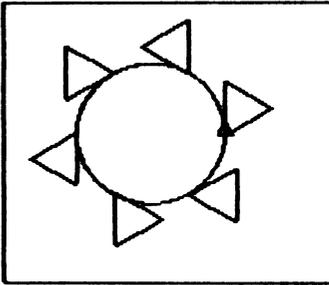
FAN



FAN

3. A recursive procedure using a triangle:

```
TO MILLWHEEL  
  TRI  
  ARCL60  
  MILLWHEEL  
END
```

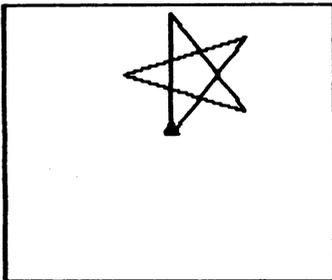


MILLWHEEL

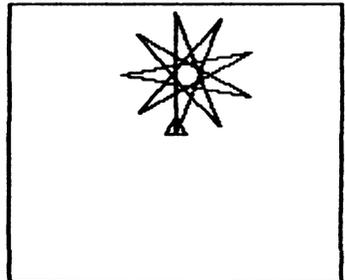
4. Stars:

```
TO STAR  
  FD 75 RT 144  
  STAR  
END
```

```
TO STAR9  
  FD 75 RT 160  
  STAR9  
END
```



STAR



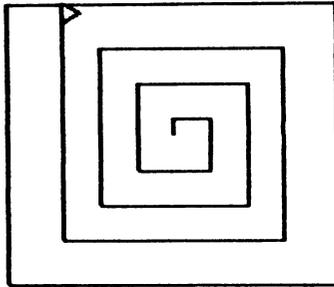
STAR9

## *Projects: Changing Inputs*

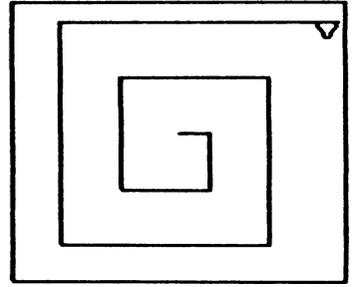
### 1. SQUARE with a larger increment:

```
TO SQUARE1 :LENGTH  
  FD :LENGTH RT 90  
  SQUARE1 :LENGTH + 15  
END
```

```
TO SQUARE2 :LENGTH  
  FD :LENGTH RT 90  
  SQUARE2 :LENGTH + 25  
END
```



SQUARE1 With + 15



SQUARE2 With + 25

### SQUARE with a smaller increment:

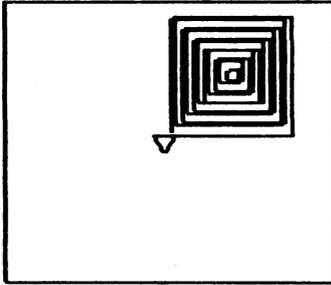
```
TO SQUARE3 :LENGTH  
  FD :LENGTH RT 90  
  SQUARE3 :LENGTH + 1  
END
```

```
TO SQUARE4 :LENGTH  
  FD :LENGTH RT 90  
  SQUARE4 :LENGTH + 3  
END
```

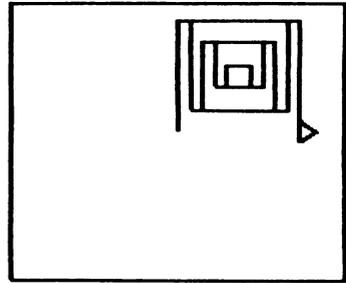
SQUARE with an increment subtracted:

```
TO SQUARE5 :LENGTH  
  FD :LENGTH RT 90  
  SQUARE5 :LENGTH -5  
END
```

```
TO SQUARE6 :LENGTH  
  FD :LENGTH RT 90  
  SQUARE6 :LENGTH -10  
END
```



SQUARE5 With-5



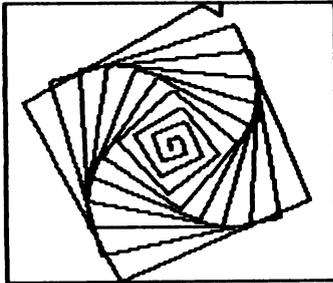
SQUARE6 With-10

Note what happens when the length of the side becomes very small and then negative...

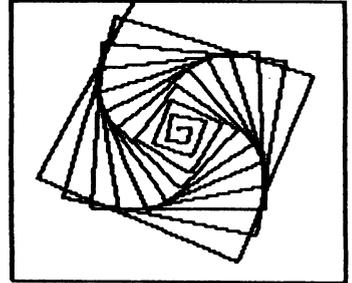
3. SQUARE with a slightly different turn:

```
TO SQUARE7 :LENGTH  
  FD :LENGTH RT 93  
  SQUARE7 :LENGTH + 5  
END
```

```
TO SQUARE8 :LENGTH  
  FD :LENGTH RT 87  
  SQUARE8 :LENGTH + 5  
END
```



SQUARE7 With RT 93



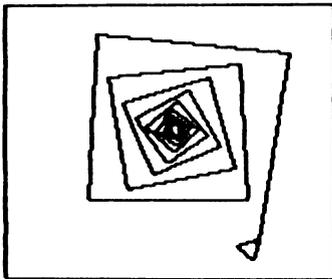
SQUARE8 With RT 87

Now you begin to see some of the power of changing the input in a recursive procedure.

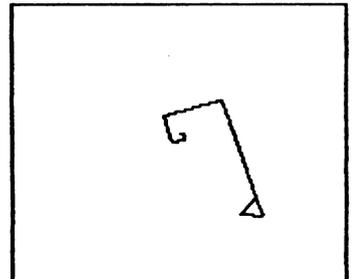
#### 4. SQUARE with the input changed by multiplication:

```
TO SQUARE9 :LENGTH
  FD :LENGTH RT 93
  SQUARE9 :LENGTH * 1.1
END
```

```
TO SQUARE10 :LENGTH
  FD :LENGTH RT 87
  SQUARE10 :LENGTH * 2
END
```



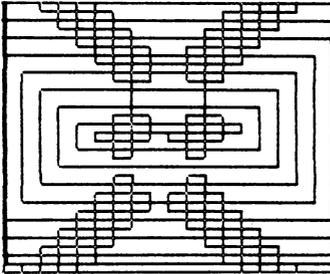
SQUARE9 With \* 1.1



SQUARE10 With \* 2

#### 5. SQUARE, SQUARE1, ... SQUARE10 in both WRAP and NOWRAP mode.

6. All the SQUAREs in WRAP and PC 6 (PENCOLOR 6): The designs will continually change. Sample picture here catches only one moment in the succession of changes.

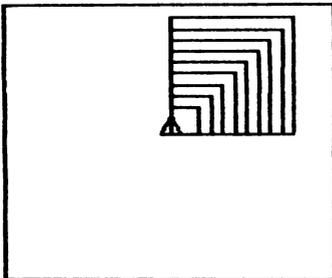


A Squaral in Wrap Mode

7. Using a SQUARE procedure with variable input (such as SQV) in a procedure that draws successively larger squares.

```
TO LARGER.SQUARES :LENGTH  
  SQV :LENGTH  
  LARGER.SQUARES :LENGTH + 10  
END
```

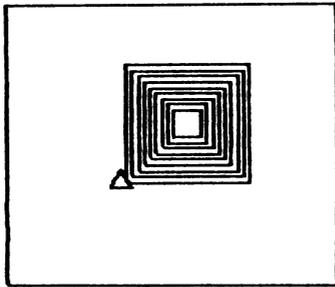
```
TO SQV :LENGTH  
  REPEAT 4 [FD :LENGTH RT 90]  
END
```



LARGER.SQUARES

If you wanted to center your squares, instead of drawing them with two common sides, you would move the turtle between squares:

```
TO LARGER.SQUARES :LENGTH
  SQV :LENGTH
  PU LT 90 FD 5 RT 90 BK 5 PD
  LARGER.SQUARES :LENGTH + 10
END
```



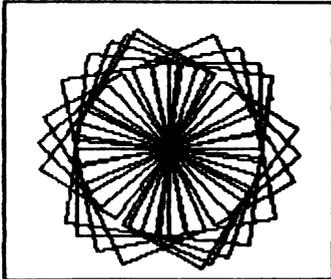
LARGER.SQUARES (Centered)

Note that the turtle turns left, moves the distance of half the increment, turns right and backs into position, moving the distance of half the increment again. The backing up saves an extra turn.

### *Projects: Testing and Stopping*

#### 1. Replacing the 45 in RT 45:

```
TO DESIGN :TIMES :LENGTH
  IF :TIMES < 1 STOP
  SQV :LENGTH
  RT :TIMES * 4
  DESIGN :TIMES-1:LENGTH
END
```

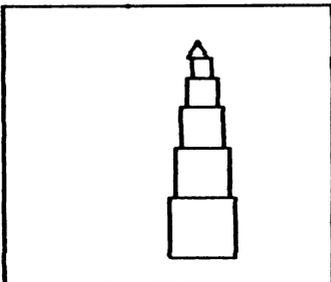


DESIGN

2. A tower of increasingly smaller squares, number of squares chosen when procedure is run, with a setup procedure to start lower on the screen (Type SET.TOWER, then type TOWER.OF.SQUARES 5 55):

```
TO TOWER.OF.SQUARES :NUM :LEN
  IF :NUM = 0 THEN STOP
  SQV :LEN
  FD :LEN RT 90 FD 5 LT 90
  TOWER.OF.SQUARES :NUM-1      :LEN-10
END
```

```
TO SET.TOWER
  PU BK 100 PD
END
```

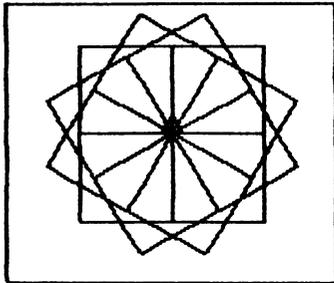


TOWER.OF.SQUARES

After drawing each square, the turtle moves up the side of the square just drawn, turns, moves half the size of the increment (so the next square is centered), and turns again, ready to begin the next square.

### 3. DESIGN with a variable turn:

```
TO DESIGN1 :LENGTH :TIMES :TURN
  IF :LENGTH < 0 THEN STOP
  IF :TIMES < 1 THEN STOP
  SQUARE :LENGTH RT :TURN
  DESIGN1 :LENGTH :TIMES -1 :TURN
END
```



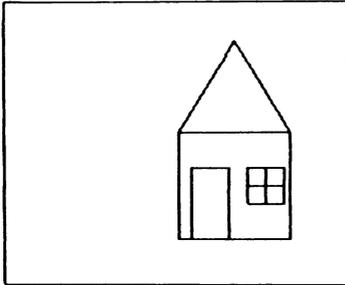
DESIGN1

## *Recursion Projects*

### 1. Successively smaller houses:

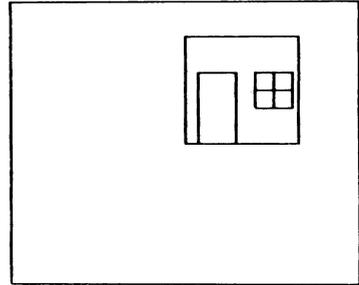
Begin by designing one house with a variable for a unit of size, to be determined later. The parts will require some instructions between them for positioning, but that too can wait. For a start, just describe what will be in the picture.

```
TO HOUSE :SIZE  
  FRONT :SIZE  
  ROOF :SIZE  
END
```



HOUSE

```
TO FRONT :SIZE  
  WALLS :SIZE  
  DOOR :SIZE  
  WINDOW :SIZE  
END
```



FRONT

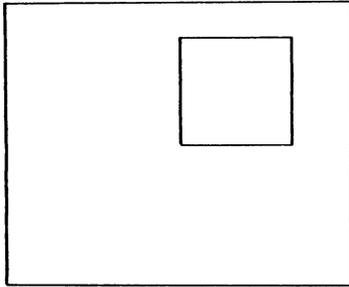
Now is the time to decide the size relationship of the components. Test each of these to be sure it works correctly before you begin on the interfacing instructions that make the parts go together.

```
TO WALLS :SIZE  
  SQUARE :SIZE * 3  
END
```

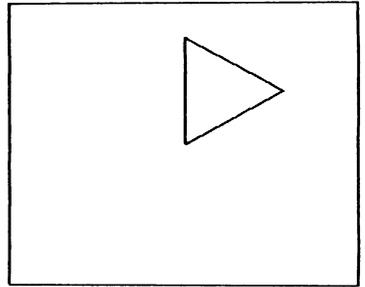
```
TO ROOF :SIZE  
  TRI :SIZE * 3  
END
```

```
TO WINDOW :SIZE  
  REPEAT 4 [SQUARE :SIZE/2 RT 90]  
END
```

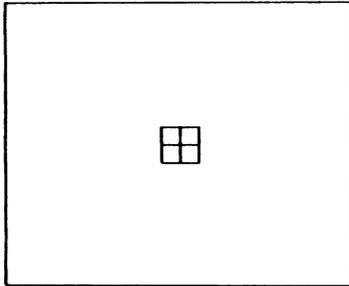
```
TO DOOR :SIZE  
  RECT :SIZE * 2 :SIZE  
END
```



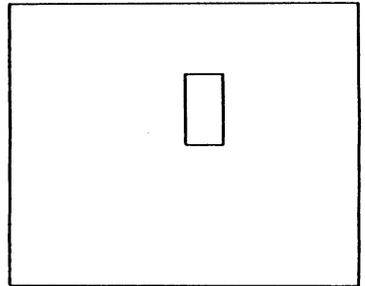
WALLS



ROOF



WINDOW



DOOR

```
TO TRI :LENGTH
  REPEAT 3 [FD :LENGTH RT 120]
END
```

```
TO SQUARE :LENGTH
  REPEAT 4 [FD :LENGTH RT 90]
END
```

```
TO RECT :LEN :WIDTH
  REPEAT 2 [FD :LEN RT 90 FD :WIDTH RT 90]
END
```

Now comes the fitting together of the parts.

In each case, the turtle finishes in its starting position. This makes it much easier to figure out how to get to where the next part is drawn.

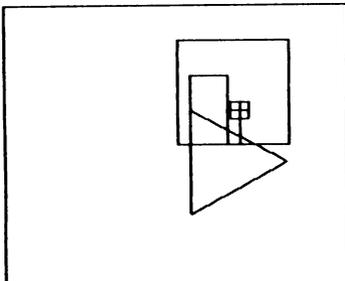
One possible solution:

```
TO HOUSE :SIZE
  FRONT :SIZE
  FD :SIZE * 3
  RT 30
  ROOF :SIZE
  LT 30
  BK :SIZE * 3
END
```

```
TO SETUP
  FULLSCREEN
  PU
  LT 90
  FD 135
  RT 90
  BK 115
  PD
END
```

```
TO FRONT :SIZE
  WALLS :SIZE
  RT 90
  FD :SIZE/3
  LT 90
  DOOR :SIZE
  PU
  RT 90
  FD :SIZE * 2
  LT 90
  FD :SIZE * 1.5
  PD
  WINDOW :SIZE
  PU
  BK :SIZE * 1.5
  LT 90
  FD :SIZE * 2 + :SIZE/3
  RT 90
  PD
END
```

SETUP moves the turtle to the lower left corner of the screen to draw the first house.



Interface Bug in House

The next problem is the procedure which will use HOUSE to draw a succession of smaller houses and stop.

```
TO H :SIZE
  IF :SIZE < 2 STOP
  HOUSE :SIZE
  PU
  RT 90
  FD :SIZE * 3.4
  LT 90
  FD :SIZE * 2
  PD
  H :SIZE * .75
END
```

The 3.4, 2, and .75 were determined by trial and error, to see what came out the best on the screen.

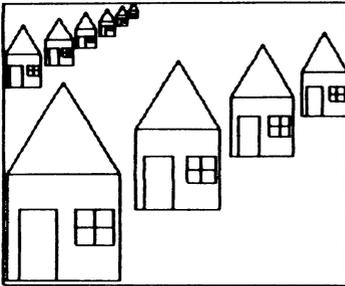
Now all that remains is to create the procedure HOUSES which will run the other procedures when you type HOUSES.

```
TO HOUSES
  HT
  SETUP
  H 30
END
```

To extend this so that you can determine the size reduction when you run the procedure, use a variable instead of the .75:

```
TO H :SIZE :FACTOR
  IF :SIZE < 2 STOP
  HOUSE :SIZE
  PU
  RT 90
  FD :SIZE * 3.4
  LT 90
  FD :SIZE * 2
  PD
  H :SIZE * :FACTOR :FACTOR
END
```

```
TO HOUSES :FACTOR
  HT
  SETUP
  H 30 :FACTOR
END
```



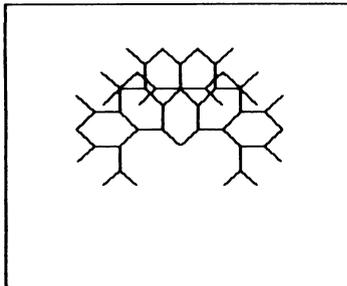
HOUSES .75

Now you have the option of making larger and larger houses, defying perspective, but you will need a test for maximum size to make the procedure stop.

## 2. A binary tree:

The basic pattern:

```
TO TREE :LENGTH  
  RT 45  
  FD :LENGTH  
  BK :LENGTH  
  LT 90  
  FD :LENGTH  
  BK :LENGTH  
  RT 45  
END
```



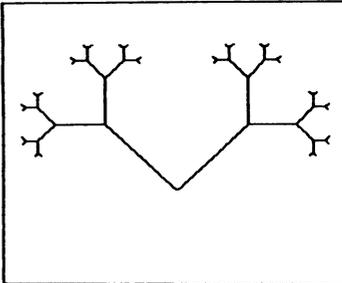
TREE 20 5

Note that the turtle finishes in its starting position.

If you want to draw another one of these at each tip, then you must determine when the turtle is at the tip and call the procedure again. Each FD :LENGTH takes the turtle to a tip, so it is after each FD that the procedure should be called again.

One way to stop this procedure so it can recurse and draw the whole tree, is to specify the number of forks:

```
TO TREE :LENGTH :FORKS
  IF :FORKS = 0 STOP
  RT 45
  FD :LENGTH
  TREE :LENGTH :FORKS -1
  BK :LENGTH
  LT 90
  FD :LENGTH
  TREE :LENGTH :FORKS -1
  BK :LENGTH
  RT 45
END
```



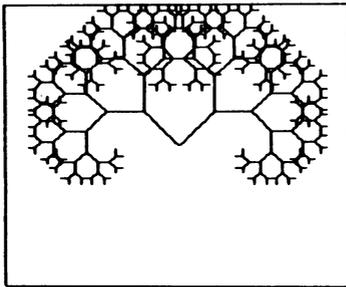
TREE1 80

A tree with successively smaller branches could be told to stop when :LENGTH reached a certain size:

```
TO TREE1 :LENGTH
  IF :LENGTH < 5 STOP
  RT 45
  FD :LENGTH
  TREE1 :LENGTH / 2
  BK :LENGTH
  LT 90
  FD :LENGTH
```

```
TREE1 :LENGTH /2
BK :LENGTH
RT 45
END

TO TREE2 :LENGTH
IF :LENGTH < 5 STOP
RT 45
FD :LENGTH
TREE2 :LENGTH * .75
BK :LENGTH
LT 90
FD :LENGTH
TREE2 :LENGTH * .75
BK :LENGTH
RT 45
END
```



TREE2 40

Each of these makes a different design. To alter it even more, consider making it with one side different from the other, perhaps doubling the length of the branches or changing the turn.

There is a good discussion of binary trees in LOGO FOR THE APPLE II, by Professor Harold Abelson, M.I.T.

### 3. A fish in a fish in a fish.

First draw one fish, then try it in different sizes to be sure they will fit together. Then, as in the houses problem, write the procedure which fits them together.

```
TO FISH :SIZE
  RT 30
  PU
  RARC :SIZE * 3 10
  PD
  RARC :SIZE * 3 110
  TAIL :SIZE
  RARC :SIZE * 3 110
END

TO FISH.IN.FISH :SIZE
  IF :SIZE > 40 STOP
  FISH :SIZE
  PU
  RARC :SIZE * 3 10
  LT 60
  FD :SIZE/3
  RT 90
  FISH.IN.FISH :SIZE + 10
END

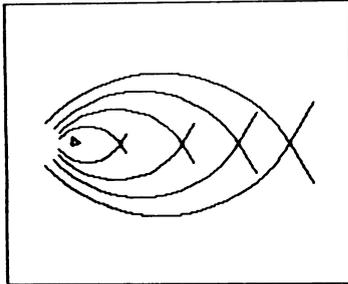
TO FISHES
  SETUP.FISH
  FISH.IN.FISH 10
  EYE
END

TO SETUP.FISH
  PU
  LT 90
  FD 100
  RT 90
  PD
  END

TO EYE
  PU
  RT 90
  FD 40
  LT 90
  FD 8
  LT 90
  BK 10
  RT 30
  FD 5
  END

TO TAIL :SIZE
  FD :SIZE
  BK :SIZE
  RT 60
  BK :SIZE
  FD :SIZE
  END
```

EYE wanders about to put the turtle in an appropriate place for the eye of the smallest fish.



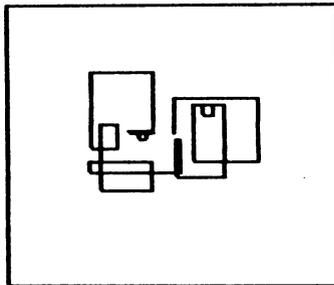
FISHES

### *Projects Using Random*

1. SQUARE3 using FD RANDOM 100 in SQUARESIDE:

```
TO SQUARESIDE
  FD RANDOM 100
  RT 90
END
```

```
TO SQUARE3
  SQUARESIDE
  SQUARE3
END
```



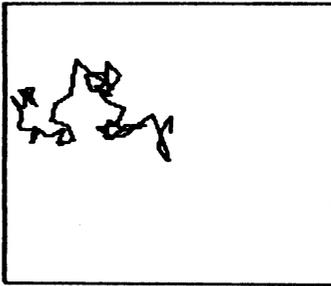
SQUARE3 with RANDOM 100

2. REPEAT using a random turn between 0 and 360:

```
REPEAT 50 [FD 20 RT RANDOM 360]
```

3. A recursive procedure using a random turn between 90 and 120:

```
TO WORM  
  FD 20  
  RT 90 + RANDOM 30  
  WORM  
END
```



WORM

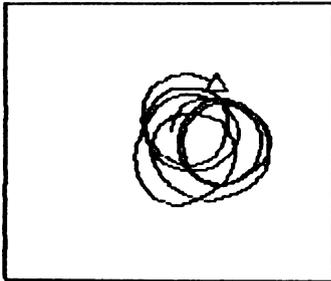
To specify a range BETWEEN two numbers, add the beginning number of the range (here 90) to the amount of the range (30, for a range of from 90 to 120). The computer will always choose a number within the amount of the range (here 30) and add it to the beginning number (here 90), to obtain a number within the specified range (here  $90 + 0$  to  $90 + 30$ , or  $90 - 120$ ).

4. Other ranges of turn:

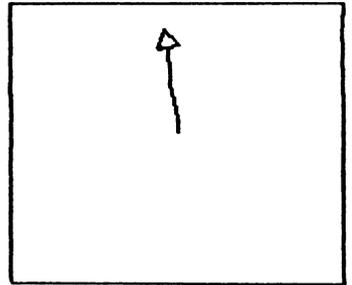
```
TO WANDER  
  FD 2  
  RT RANDOM 10  
  WANDER  
END
```

```
TO WIGGLE  
  FD 5  
  RT -10 + RANDOM 20  
  WIGGLE  
END
```

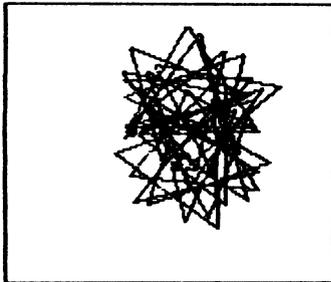
```
TO VARY  
  FD 10  
  RT 120 + RANDOM 30  
  VARY  
END
```



WANDER



WIGGLE



VARY

### *Mascots: Elephant, Rabbit, Snail*

No lions and tigers and bears, but an elephant (that's for remembrance), a rabbit (denoting speed and ingenuity), and a snail (go slow... slow... slow).

The arcs used are described in the arc development section. To use the arc procedures on the Utilities Disk, change ARCR to RARC and ARCL to LARC in each of the procedures below.

#### Elephant

```
TO ELEPHANT :SIZE
  HT
  ELEPHANT.EAR :SIZE
  TRUNK :SIZE
  TUSK :SIZE
  EYE :SIZE
END
```

```
TO TUSK :SIZE
  ARCL 10 * :SIZE 70
  RT 160
  ARCR 10 * :SIZE 50
END
```

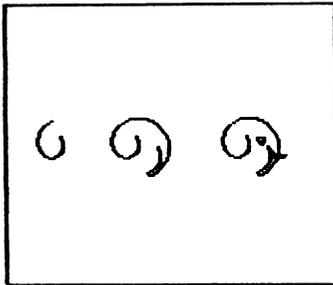
```
TO ELEPHANT.EAR :SIZE
  RT 160
  FD 3 * :SIZE
  ARCR 7 * :SIZE 180
  ARCR 13 * :SIZE 90
END
```

```
TO TRUNK :SIZE
  ARCR 17 * :SIZE 180
  ARCR :SIZE 180
  ARCL 10 * :SIZE 100
  RT 180
END
```

```

TO EYE :SIZE
  PU
  RT 60
  ARCL 10 * :SIZE 60
  PD
  RCIRCLE 2 * :SIZE
END

```



Evolving the Elephant

For the mascot elephant, :SIZE = 1.

### Rabbit

```

TO RABBIT
  HT
  HEAD
  ARCL 7.5 90
  RT 60
  BODY
END

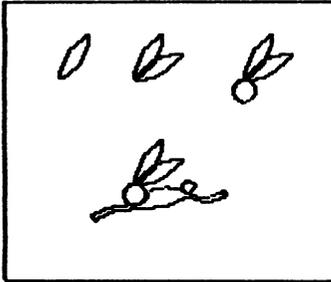
TO BODY
  ARCR 20 60
  LCIRCLE 3.5
  ARCL 20 60
  ARCR 1.5 180
  ARCR 20 60
  LT 60
  ARCR 50 30
  ARCL 50 30
  ARCR 1.5 180
  ARCR 50 30
END

```

```
TO EARS
  EAR
  RT 150
  EAR
END
```

```
TO EAR
  ARCR 30 60
  RT 120
  ARCR 30 60
END
```

```
TO HEAD
  EARS
  ARCL 6 540
END
```



Evolving the Rabbit

### Snail

```
TO SNAIL
  HT
  SNAIL.BODY
  SNAIL.HEAD
  RT 180
  ARCR 5 (270-HEADING)
  SNAIL.FOOT
END
```

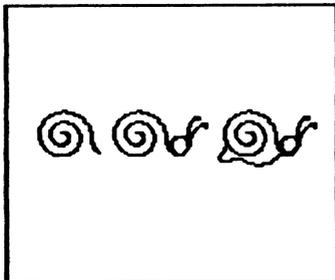
```
TO POLYARC :SIZE :TIMES
  IF :TIMES = 0 THEN STOP
  ARCR :SIZE 60
  POLYARC :SIZE + 1 :TIMES-1
END
```

```
TO SNAIL.BODY
  POLYARC 1 15
  ARCL 10 60
END
```

```
TO ANTENNA
  ARCR 15 60
  ARCR 1 360
  PU
  RT 180
  ARCL 15 60
  RT 180
  PD
END
```

```
TO SNAIL.HEAD
  ARCL 5 475
  ANTENNA
  ARCL 5 20
  ANTENNA
END
```

```
TO SNAIL.FOOT
  ARCR 5 40
  LT 100
  ARCL 15 90
  ARCL 10 60
  ARCR 3 120
  RT 60
  ARCL 8 90
END
```



Evolving the Snail

## *Procedures for Saving Pictures*

The illustrations in the Graphics Procedures section were drawn (2/3 scale) and stored on the disk with the following procedures:

```
TO STORE :PROCEDURE          TO H
  DRAW                       PU
  FRAME                       HOME
  H                           PD
  RUN SENTENCE :PROCEDURE [ ] END
  TURTLE
  SAVEPICT :PROCEDURE
END
```

```
TO TURTLE
  LT 90 BK 6
  REPEAT 3 [FD 12 RT 120]
END
```

```
TO FRAME
  PU SETXY -90 (-85) SETHEADING 0 PD
  REPEAT 2 [FD 160 RT 90 FD 180 RT 90]
END
```

Example: type

```
STORE "TOWN
```

STORE clears the screen, draws the frame, moves the turtle to the HOME position, then runs the procedure TOWN. The SENTENCE :PROCEDURE [ ] makes a list out of the procedure name, so it can be RUN by another procedure. It turns the command into RUN [TOWN]. (See the chapter on Words and Lists.) The procedure TURTLE draws a little turtle, since SAVEPICT does not draw the turtle. SAVEPICT stores the picture on the disk under the procedure name.

Here is a set of procedures used to generate droves of wild animals. This also illustrates a use for SETXY.

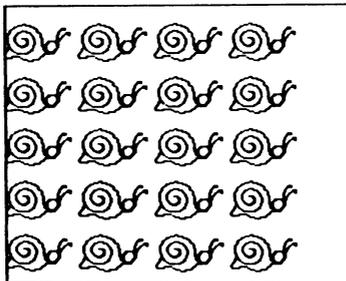
```
TO DROVE :ANIMAL
  FULLSCREEN
  QUAD :ANIMAL (-90)
END
```

```
TO QUAD :PROC :Y
  IF :Y > 90 STOP
  LINE (-125) :Y :PROC
  QUAD :PROC :Y + 45
END
```

```
TO LINE :X :Y :PROC
  IF :X > 55 STOP
  PU
  SETXY :X :Y
  PD
  SETHEADING 0
  RUN SE :PROC [ ]
  LINE :X + 60 :Y :PROC
END
```

To draw a lot of little pictures, type DROVE and the name of the procedure that draws the picture. For example, type

```
DROVE "SNAIL
```

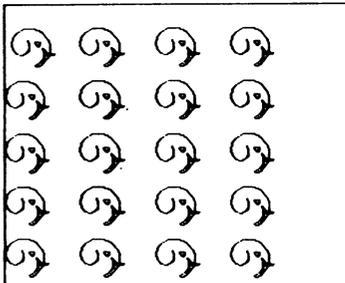


DROVE of Snails

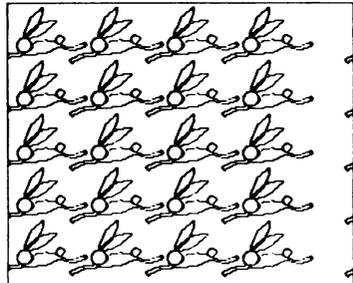
DROVE shows you the whole screen, since the drawing begins in the lower left corner, and calls QUAD with a Y value of -90, close to the bottom of the screen. DROVE is in charge of the whole project.

QUAD tests to be sure you are not going to be drawing off the top of the screen ( $Y > 90$ ), then calls LINE with a value for X (-125) which will start the drawing near the left edge of the screen. When LINE has finished, QUAD moves into position for the next line of pictures and calls LINE again. QUAD uses LINE several times to draw rows of pictures.

LINE tests to be sure you are not drawing off the right side of the screen, then takes the beginning value of X and the value of Y, and moves to that position. LINE then uses RUN to call the procedure that draws the picture, and calls itself with a new position to the right (incremented value of :X, same value of :Y). LINE draws one row of pictures.



DROVE of Elephants



DROVE of Rabbits

## *Developing an Arc Procedure*

It is easiest to develop a circle procedure, then generalize it to do arcs. Then you can use the arc procedure to do everything, including circles.

We want a circle procedure which will depend on the radius, so that we can specify the size by giving the radius when the procedure is run. We work from the fact that the circumference of a circle equals the radius times 2 PI:  $C = 2 \text{ PI (times) } R$ , or, translating for the computer,  $C = 2 * 3.14159 * R$ .

In Logo, every drawing is some combination of steps and turns, so the circle must also consist of steps and turns. A circle of a certain fixed size is drawn by

```
REPEAT 360 [FD 1 RT 1]
```

The 360 comes from the turn of 1; to turn 360 degrees with a turn of 1 degree requires 360 turns, or  $360/1 = 360$ .

The 360 might also be said to represent the circumference, the distance around. We can substitute for it the equivalent  $2 * 3.14159 * R$ . This makes the circumference depend on the radius, as we wanted.

The turn must also be changed to be a function of the radius; if we use the same step and turn as before, we will not have changed the size of the circle. How can we figure out what the turn should be?

With a turn of 1 degree, we figured out the number of turns by dividing 360 degrees by that amount, to get

360 turns. If we use the same relationship, we see that the amount of turn is 360 divided by the number of turns.

The number of turns in our new model is  $2 * 3.14159 * R$ , so the amount of the turn will be  $360 / 2 * 3.14159 * R$ .

Our circle statement (type as one line) becomes

Type as one line    `REPEAT 2 * 3.14159 * :RADIUS  
                  [FD 1 RT 360/(2 * 3.14159 * :RADIUS)]`

Our circle procedure becomes

Type as one line    `TO RCIRCLE :RADIUS  
  REPEAT 2 * 3.14159 * :RADIUS  
    [FD 1 RT 360/(2 * 3.14159 * :RADIUS)]  
END`

Type the REPEAT statement as one line, with only one <RETURN>, at the end. Substitute LT for an LCIRCLE procedure.

To change the circle procedure to an arc procedure, we must change the number of turns to draw the fraction of the circle the arc represents. How do we figure that fraction?

A 60 degree arc is  $60/360$ , or  $1/6$ th of a circle. The fraction of the circle which is any arc then, would be represented by (its size) / 360. If we call its size :DEGREES, then  $:DEGREES / 360$  would be the fraction of the circle which is the arc of the size :DEGREES. ( $360/360 =$  the circle)

The number of turns would be the fraction of the circle represented by the arc, times the number required by the full circle, or

$$(\text{DEGREES}/360) * (2 * 3.14159 * \text{RADIUS})$$

The arc procedure would be

Type as one line

```
TO ARCR :RAD :DEG
  REPEAT (:DEG/360) * (2 * 3.14159 * :RAD)
    [FD 1 RT 360/(2 * 3.14159 * :RAD)]
  END
```

Simplifying by doing the arithmetic gives

```
TO ARCR :RAD :DEG
  REPEAT .0174532 * :DEG * :RAD [FD 1 RT 57.295827 / :RAD]
  END
```

The circle procedure becomes

```
TO RCIRCLE :RADIUS
  ARCR :RADIUS 360
  END
```

LCIRCLE would use ARCL, the same as ARCR with LT substituted for RT. If you wanted to be silly, you could write

```
TO ARCL :RADIUS :DEGREES
  ARCR -:RADIUS (-:DEGREES)
  END
```

Now all the arc and circle procedures are based on one, and only one, procedure. Making the radius negative has the effect of making the turn negative, or LT.

To increase the resolution of the picture, really only desirable when you are going to print a design on paper, decrease the size of the step. Replace the original 1 with :STEP and add the variable to the title.

To keep our procedure drawing arcs with the specified radius, we must multiply the turn by the :STEP and consequently, divide the number of turns by :STEP, giving us (name changed to avoid confusion with the non-variable step version):

```
Type as one line  TO RARC :RADIUS :DEG :STEP
                   REPEAT (.0174532 * :DEG * :RADIUS)/:STEP
                     [FD :STEP RT (57.295827 * :STEP) /:RADIUS]
                   END
```

### *Debugging with TRACE, NOTRACE*

TRACE allows you to watch the execution of your procedure line by line. Logo prints a statement, waits for you to type a character, then executes the statement. TRACE also tells you when it is starting a subprocedure, and tells you what the inputs are.

In TRACE mode, type <CTRL> G, as usual, to stop a procedure. <CTRL> Z will make it PAUSE; type CO (or CONTINUE) to resume. Type NOTRACE to stop tracing.

TRACE and NOTRACE may be used in a procedure to trace just a portion of it.

## *Adding Remarks in Your Procedures*

When you use descriptive procedure names and variable names, and write short procedures and subprocedures, your need for remarks throughout your procedures is lessened, and in many cases, eliminated.

However, for those remarks that simply must go in, precede them with a semi-colon (;) as in the (not to be taken seriously as an) example:

```
TO SQUARE  
FD 100;GOES FORWARD 100  
RT 90;GOES RIGHT 90  
SQUARE;CALLS ITSELF  
END
```

## *Switching Disk Drives: SETDISK*

Occasionally you may want to use more than one disk drive in your Logo system. Use the SETDISK command to switch back and forth between drives. SETDISK takes two inputs, a drive number and a slot number, and causes all subsequent file operations to be done in that drive. For example, SETDISK 2 6 transfers control to the second drive in a two-drive system. Default is SETDISK 1 6.

## *Creating Self-Starting Files Using the STARTUP Variable*

It is possible to write Logo files that begin executing immediately after being read into the workspace. There is an interesting way of doing this using the address SAVMOD, found in the Technical chapter; however this way is also rather difficult.

A much easier way to create self-starting files is to use a STARTUP variable. Simply include in the file a global variable consisting of a list of the procedure to be started automatically. For example, if Logo encounters the message

```
MAKE "STARTUP [DEMO]
```

while reading in a file, the procedure DEMO will begin automatically.

## *Reading Apple Logo Files*

Using Terrapin Logo, you can read files created with Apple Logo from Apple Computer. It's as easy as typing

```
READ "FILENAME
```

Of course, you will need to alter the syntax of some procedures to make them run correctly.

You may encounter a message similar to THERE IS NO PROCEDURE NAMED PPROP. If so, read in the file using the READTEXT procedure which is found in the TEXTEDIT file on the Utility disk.

Now type ED <return> and delete the offending command. Now type <CTRL>C to define the procedures.

A separate product available through Terrapin, **Utilities II**, contains an Apple Logo Translator program which does this work for you. Contact Terrapin for more information.

### *STRATEGIES FOR THE WORDS AND LISTS PROJECTS*

1. Here is one version.

```
TO EASY :CHTR
  IF :CHTR = "F FD 10
  IF :CHTR = "R RT 15
  IF :CHTR = "L LT 15
  IF :CHTR = "D DRAW
  IF :CHTR = "U PU
  IF :CHTR = "P PD
END
```

2. Use the same strategy, adding lines like

```
IF :CHTR = "S ST
IF :CHTR = "H HT
```

3. For a two-stroke method, EASY would need to contain a line such as

```
IF :CHTR = "C SETPENCOLOR RC
```

As in QUICKDRAW, RC grabs a character from the user, and SETPENCOLOR examines that character and, if it is a number from 0 to 6, sets the color accordingly.

SETPENCOLOR could be written several ways. One way that uses no new techniques is this:

```
TO SETPENCOLOR :CHTR
  IF :CHTR = 0 PC 0
  IF :CHTR = 1 PC 1
  IF :CHTR = 2 PC 2
  IF :CHTR = 3 PC 3
  IF :CHTR = 4 PC 4
  IF :CHTR = 5 PC 5
  IF :CHTR = 6 PC 6
END
```

Logo, however, makes life much simpler. If the character is not a number, it certainly is not a 0, 1, 2, 3, etc., and so we need not make all of those tests separately. This is worded concisely in Logo:

```
IF NOT NUMBER? :CHTR STOP
```

Then, if it is a number less than 7, it must be a 0 through 6, and we can just set the PENCOLOR to whatever CHTR happens to be.

```
IF :CHTR < 7 PC :CHTR
```

And that is all the procedure needs to do. Here are two ways to write that procedure.

```
TO SETPENCOLOR :CHTR
  IF NOT NUMBER? :CHTR STOP
  IF :CHTR < 7 PC :CHTR
END
```

```
TO SETPENCOLOR :CHTR
  IF NUMBER? :CHTR THEN IF :CHTR < 7 PC :CHTR
END
```

As a frill, the line in EASY could be:

```
IF :CHTR = "C PRINT1 [WHAT COLOR?] SETPENCOLOR RC
```

Look up PRINT1 in the Logo glossary.

4. You can use exactly the same strategy as above. Because the test for the second character is the same for setting the background color as for setting the pen color, it might make sense to use one procedure for both.

The problem is that after the procedure has verified that the character is a 0 through 6, it must know not only what character was typed, but also which to set, pen or background color.

Here is a procedure that can do both, but it involves more advanced techniques than we have yet explained in the tutorial. Don't worry! You can choose either to use the ones fully explained, or jump the gun and try the new technique.

```
TO SETCOLOR :WHICHCOLOR :CHTR
  IF NOT NUMBER? :CHTR STOP
  IF :CHTR > 6 STOP
  IF :WHICHCOLOR = [PEN] PC :CHTR ELSE BG :CHTR
END
```

The lines in EASY would need to be slightly different, stating which color, PEN or BACKGROUND, was to be changed. Here is one set of possibilities.

```
IF :CHTR = "C PRINT1 [WHAT COLOR?] SETCOLOR [PEN]
RC
IF :CHTR = "B PRINT1 [WHAT COLOR?] SETCOLOR
[BACKGROUND] RC
```

5. Recognizing and using digits can be done several ways. The simplest (if not most elegant) way to write EASY would be to add a bunch of lines like this:

```
IF :CHTR = 2 MAKE "MULTIPLE 2
IF :CHTR = 3 MAKE "MULTIPLE 3
IF :CHTR = 4 MAKE "MULTIPLE 4
IF :CHTR = 5 MAKE "MULTIPLE 5
IF :CHTR = 6 MAKE "MULTIPLE 6
IF :CHTR = 7 MAKE "MULTIPLE 7
IF :CHTR = 8 MAKE "MULTIPLE 8
IF :CHTR = 9 MAKE "MULTIPLE 9
```

Of course, all these lines say essentially the same thing, namely: "If the character is a number, make MULTIPLE that number." That can be translated straightforwardly into Logo with the much more compact statement.

```
IF NUMBER? :CHTR MAKE "MULTIPLE :CHTR
```

Inserting this new logic into EASY requires that we use the new value, and so the lines that move the turtle must now incorporate MULTIPLE thus:

```

IF :CHTR = "F FD 10 * :MULTIPLE
IF :CHTR = "R RT 15 * :MULTIPLE
IF :CHTR = "L LT 15 * :MULTIPLE

```

Alternatively, the lines could be

```
IF :CHTR = "F REPEAT :MULTIPLE [FD 10] etc.
```

Finally, we always want to reset the multiple to 1 so that it doesn't spill over from one command to the next.

Here is how the procedure might look.

```

TO QUICKDRAW
  EASY RC
  QUICKDRAW
END

```

```

TO EASY :CHTR
  IF :CHTR = "F FD 10 * :MULTIPLE
  IF :CHTR = "R RT 15 * :MULTIPLE
  IF :CHTR = "L LT 15 * :MULTIPLE
  IF :CHTR = "D DRAW
  IF :CHTR = "U PU
  IF :CHTR = "P PD
  IF :CHTR = "H HT
  IF :CHTR = "S ST
  IF :CHTR = "C PRINT1 [WHAT COLOR?] SETCOLOR [PEN]
  RC
  IF :CHTR = "B PRINT1 [WHAT COLOR?] SETCOLOR [BG]
  RC
  MAKE "MULTIPLE 1
  IF NUMBER? :CHTR MAKE "MULTIPLE :CHTR
END

```

EASY sets MULTIPLE to 1 every time it is executed. As already mentioned, this is so that L or F or R will mean the same as 1L or 1F or 1R each time unless some other number is typed.

The placement of the MAKE "MULTIPLE 1 line is important. It must be placed after the lines that use the value of MULTIPLE and before the line that sets MULTIPLE to values other than 1. Otherwise the special values of MULTIPLE would persist too long or be erased too soon.

A second thing to notice is that EASY cannot use MULTIPLE before setting it the first time. So before QUICKDRAW can be started, MULTIPLE must be given a value (presumably the value 1). This startup procedure seems convenient:

```
TO QD
  MAKE "MULTIPLE 1
  QUICKDRAW
END
```

6. The procedure PEN picks the pen up if it is already down, and puts it down if it is already up. We say it "toggles the pen state." To include it in EASY, only one line is needed:

```
IF :CHTR = "P PEN
```

The line IF :CHTR = "U PU can be eliminated, because P now takes care of both PD and PU.

Since PEN uses the variable PENPOS, QD (the setup procedure written earlier) should initially set the pen position to [DOWN].

```
TO QD
  MAKE "MULTIPLE 1
  MAKE "PENPOS [DOWN]
  QUICKDRAW
END
```

It is also possible to set up a toggle that works without setting a global variable with MAKE. Look up TURTLESTATE in the Logo glossary, and learn about FIRST (from the glossary or later in this chapter) to understand this alternate version of PEN which we are calling TOGGLEPEN.

```
TO EASY :CHTR
  IF :CHTR = "F FD 10
  IF :CHTR = "R RT 15
  IF :CHTR = "L LT 15
  IF :CHTR = "D DRAW
  IF :CHTR = "P TOGGLEPEN
END
```

```
TO TOGGLEPEN
  IF FIRST TS PU ELSE PD
END
```

TS is the abbreviation for TURTLESTATE. The first element of the list that TS outputs tells whether the turtle's pen is up or down. If it is down (if FIRST TS is TRUE) TOGGLEPEN puts it up, otherwise it puts it down.

In this case, since no global variable is involved, no additions to QD would need to have been made.

```
7. TO TOGGLE.SHOWN
  TEST :SHOWN = [SHOWN]
  IFTRUE HT MAKE "SHOWN [HIDDEN]
  IFFALSE ST MAKE "SHOWN [SHOWN]
END
```

It is not necessary to tell the user whether the turtle is shown or not, so the PRINT statement was not added. Since the values [SHOWN] and [HIDDEN] now serve only as information to the procedure (they will not be printed as information to the user), it would be more “natural” to use TRUE and FALSE to state whether the turtle was shown.

The logic would then be this: If the turtle is shown (that is, if SHOWN is TRUE) then hide the turtle, else show it. In either case, make SHOWN whatever it was not; use the primitive NOT to make it FALSE if it is TRUE, or TRUE if it is FALSE.

```
TO TOGGLE.SHOWN
  IF :SHOWN HT ELSE ST
  MAKE "SHOWN NOT :SHOWN
END
```

Finally, a strategy using TURTLESTATE and avoiding the use of global variables works for showing and hiding the turtle as well as for the pen position.

Again, this strategy makes use of techniques we have not yet described, but which you can look up if you want to begin learning about them now.

TOGGLE.SHOWN using TURTLESTATE would look like this:

```
TO TOGGLE.SHOWN
  IF FIRST BUTFIRST TS HT ELSE ST
END
```

See TURTLESTATE, and learn about BUTFIRST (in the glossary or later in this chapter).

8. ACTION no longer needs to control the turns directly, but can handle turning the way it handles speed. So, it might look like this:

```
TO ACTION :CHTR
  IF :CHTR = "R MAKE "ANG :ANG + 2 ;TURN RIGHT
  MORE
  IF :CHTR = "L MAKE "ANG :ANG - 2 ;TURN LEFT MORE
  IF :CHTR = "F MAKE "DIST :DIST + 2 ; FASTER
  IF :CHTR = "S MAKE "DIST :DIST - 2 ; SLOWER
  IF :CHTR = "D DRAW
END
```

START now has to initialize one more global variable, ANG, to something sensible, and might look like this:

```
TO START
  MAKE "DIST 0
  MAKE "ANG 0
  LOOP
END
```

It might also be nice if the D key really reset everything. As the program currently stands, D will clear the screen, but still leave the turtle flying around in whatever way it last flew. It might be reasonable to change

```
IF :CHTR = "D DRAW
to
IF :CHTR = "D CLEAR
```

and then to write a procedure CLEAR which reinitializes the global variables and clears the screen.

```
TO CLEAR
  MAKE "ANG 0
  MAKE "DIST 0
  DRAW
END
```

9. The feature to stop the turtle must reinitialize ANG and DIST without clearing the screen. Here is one.

```
TO RESET
  MAKE "ANG 0
  MAKE "DIST 0
END
```

Then the lines in ACTION would be

```
IF :CHTR = "D CLEAR
```

to accomplish the previous task of clearing the screen, and

```
IF :CHTR = ". RESET
```

to stop the turtle without clearing the screen. (The command character to stop the turtle is the period.)

Here are lines for reversing the rotation of the turtle, reversing the direction of the turtle and reversing both. Insert them and play with them. The effects are very interesting.

```
IF :CHTR = "T MAKE "ANG (- :ANG) ; REVERSES TURN
IF :CHTR = "M MAKE "DIST (- :DIST) ; REVERSES
    MOVEMENT
IF :CHTR = "B MAKE "DIST (- :DIST)
    MAKE "ANG (- :ANG) ; REVERSES BOTH
```

```
10. TO DECODE :N
    OP NTH :N "ABCDEFGHIJKLMNOPQRSTUVWXYZ
    END
```

There is another way that doesn't involve "counting" with NTH (and therefore is faster). CHAR is a Logo primitive that takes an integer as input and outputs the character whose ASCII code is that integer. The ASCII code for A is 65. For B, it is 66; for C, 67, and so on. So another way to write DECODE is:

```
TO DECODE :N
    OP CHAR (:N + 64)
    END
```

```
11. TO ONENUM :LIST
    OP DECODE FIRST :LIST
    END
```

```
12. TO TWONUM :LIST
    OP WORD DECODE FIRST :LIST ONENUM
    BF :LIST
    END
```

```
13. TO THRENUM :LIST
    OP WORD DECODE FIRST :LIST TWONUM
    BF :LIST
    END
```

14. Here is the logic. If I have only one number in my list, I know exactly what to do. As in ONENUM, I simply OP DECODE FIRST :LIST.

If my list is longer than that, I cannot handle it all at once, so I get ready to glue together the decoding of the first number (which I can do immediately) and the decoding of a slightly shorter list.

Since the exact same reasoning applies to the slightly shorter list, the same procedure can be used. Either it can now handle the list directly (because there is only one number left in it), or it, too, gets ready to glue on its little piece and defers the rest of the job to another step. Here is the procedure it generates.

```
TO ANYNUM :LIST
    IF ( BF :LIST ) = []
        OP DECODE FIRST :LIST
    OP WORD DECODE FIRST :LIST
    ANYNUM BF :LIST
    END
```

15. This could all be done in a single procedure with one long and ugly line that looks something like this:

```
TO RANDSENT
  PR (SE NTH 1 + RANDOM 7 PEOPLE
      NTH 1 + RANDOM 6 ACTIONS
      NTH 1 + RANDOM 7 PEOPLE)
END
```

The repetitive elements and the difficulty of seeing which words go with which make it useful to write a helpful subprocedure. Good style makes it easy to change and extend the program if you want to. Here is a first attempt:

```
TO RANDSENT
  PR SENTENCE WHO DIDWHAT
END
```

```
TO WHO
  OP PICK 7 PEOPLE
END
```

```
TO DIDWHAT
  OP SE DIDIT WHO
END
```

```
TO DIDIT
  OP PICK 6 ACTIONS
END
```

```
TO PICK :LISTSIZE :LIST
  OP NTH 1 + RANDOM :LISTSIZE :LIST
END
```

A problem with this way of doing things is that if **ACTIONS** or **PEOPLE** are edited, and the number of items in their lists is changed, **WHO** and **DIDIT** must also be edited, because they make explicit assumptions about the length of the lists they get.

This is not good programming practice, but fortunately **LISTSIZE** can always be determined from **LIST** just by counting, if we had a procedure that could count the elements in a list.

The procedure **COUNT**, which takes a list (or a word) as its input, does exactly this. (In Terrapin Logo version 2.0, **COUNT** is defined as a primitive.)

```
TO COUNT :OBJ
  IF :OBJ = [] OP 0
  OP 1 + COUNT BF :OBJ
END
```

To see what **COUNT** does, type

```
COUNT [L O G O]
COUNT [LOGO]
COUNT "LOGO
```

Because **PICK** can use **COUNT** to determine the list's size, it no longer needs to be told the size, and so **LISTSIZE** can be dropped from the title line. Where that information was needed in the body of the old version, **COUNT :LIST** can be substituted. The result is a procedure that looks like this.

```
TO PICK :LIST
  OP NTH 1 + RANDOM (COUNT :LIST) :LIST
END
```

Because PICK now takes only one input — the actual list — WHO and DIDIT need to be edited to use PICK properly.

```
TO WHO          TO DIDIT
  OP PICK PEOPLE OP PICK ACTIONS
END             END
```

The resulting program not only solves the problem raised earlier — namely, that PEOPLE and ACTIONS can be edited freely without requiring changes to be made in WHO and DIDIT — but it also looks “cleaner.”

It is a general rule of good programming that by designing the “low level procedures” (such as PICK) properly, the higher level procedures (such as WHO) become cleaner, better organized, and easier to understand and debug.

16. As with all procedures, there are lots of possible designs. Here is one for VOWEL?.

```
TO VOWEL? :LETTER
  IF :LETTER = "A OP "TRUE
  IF :LETTER = "E OP "TRUE
  IF :LETTER = "I OP "TRUE
  IF :LETTER = "O OP "TRUE
  IF :LETTER = "U OP "TRUE
  OP "FALSE
END
```

But the logic is that IF the :LETTER is any one of A, E, I, O, or U, then OP "TRUE, otherwise OP "FALSE. This might be more concisely expressed as

```
TO VOWEL? :LETTER
  IF MEMBER? :LETTER [A E I O U] OP "TRUE
  OP "FALSE
END
```

But remember, MEMBER? is a predicate itself. It already outputs TRUE or FALSE, exactly what we want VOWEL? to output. So, VOWEL? can also be written:

```
TO VOWEL? :LETTER
  OP MEMBER? :LETTER [A E I O U]
END
```

or even

```
TO VOWEL? :LETTER
  OP MEMBER? :LETTER "AEIOU
END
```

18. It is tempting to write a YES? procedure modeled on VOWEL? like this:

```
TO YES?
  OP MEMBER? REQUEST [ [YES] [YUP] [Y] [SURE]
  [YEAH] ]
END
```

but all life is not that simple. What if the person types [I SUPPOSE SO]? The procedure would translate that as if it were a clear NO, when it is probably YES, or at least ambiguous. Alas, we must work harder.

Here is a suggestion.

```
TO YES?
  OP YESSUB? REQUEST
END
```

```
TO YESSUB? :RESPONSE
  IF MEMBER? :RESPONSE [ [YES] [YUP] [Y] [SURE]
    [YEAH] ] OP "TRUE
  IF MEMBER? :RESPONSE [ [NO] [NOPE] [N] ] OP
    "FALSE
  PRINT1 [PLEASE ANSWER "YES" OR "NO":]
  OP YES?
END
```

This is recursive in a new way. YES? is not defined in terms of itself, nor is YESSUB? — but each is defined in terms of the other! Make sure you understand how these two procedures work together.

18. Either of the first two work properly. To see what is wrong with the third version, try PLURAL "OX.

19. It would be convenient to have a procedure that returned the last two letters of a word. Of course, if there is only one letter in the word, LASTTWO must output the whole thing.

```
TO LASTTWO :WORD
  IF " = BL :WORD OP :WORD
  OP WORD LAST BL :WORD LAST :WORD
END
```

Now we can write a rule for handling words that need ES endings. Let's replace

```
IF "X = LAST :NOUN OP WORD :NOUN "ES
```

with

```
IF NEEDS.ES? :NOUN OP WORD :NOUN "ES
```

Cheating! NEEDS.ES? hasn't been written yet.

```
TO NEEDS.ES? :NOUN
  IF ( ANYOF "S = LAST :NOUN
      "X = LAST :NOUN
      "Z = LAST :NOUN ) OP "TRUE
  OP ANYOF "CH = LASTTWO :NOUN
      "SH = LASTTWO :NOUN
END
```

Alas, the formatting which makes the design so clear on paper is all lost in Logo's editor!

```
20. IF "Y = LAST :NOUN OP WORD BUTLAST
:NOUN "IES
```

21. Ah, but not if the letter before the Y is a vowel!

```
IF "Y = LAST :NOUN OP YPLU :NOUN
```

```
TO YPLU :NOUN
  IF VOWEL? LAST BL :NOUN OP WORD :NOUN "S
  OP WORD BUTLAST :NOUN "IES
END
```

22. The big difference between **FIXVERB** and **PLURAL** is in their handling of lists. In the case of nouns, it was always the **LAST** element of the list that needed to be pluralized, but in the case of the verbs in **ACTIONS**, it is always the **FIRST** element that needs the modification. So the important line to change is the one that begins

```
IF LIST?
```

For **FIXVERB**, it might look like this:

```
IF LIST? :VERB OP SE FIXVERB FIRST :VERB BF :VERB
```

**PAST** and **FIXVERB** appear to have absolutely identical logic, but their exceptions are different. This brings up an interesting problem. The solution used in **PLURAL** was to create global variables which contained the proper form of exceptional words. What happens with verbs like **HAVE** or **GO** which have different exceptions for present and past forms? Although there is always a way to solve the problem if you notice it, the use of global variables is prone to surprising bugs until you notice the conflict.

```
TO PRESENT :SUBJ :VERB
  IF "BE = :VERB OP EXCEPTION.BE :SUBJ
  IF ( ANYOF "I = :SUBJ
    "YOU = :SUBJ
    "WE = :SUBJ
    "THEY = :SUBJ ) OP :VERB
  OP FIXVERB :VERB
END
```

Try to write **EXCEPTION.BE** yourself!

23. An extra level of analysis is needed in order to determine which class of verbs (which conjugation) is involved.

Here is a simplifying structure for the top level. It uses global variables in a risky way, but the structure will be fairly clear.

```
TO PRESENT :SUJET :VERBE
  MAKE "ROOT BL BL :VERBE ; SEPARATE ROOT
  MAKE "END LASTTWO :VERBE ; SEPARATE CONJ.
  MARKER
  ; AND NOW, HANDLE EACH CASE SEPARATELY
  IF "ER = :END OP ER.PRES :SUJET :ROOT
  IF "IR = :END OP IR.PRES :SUJET :ROOT
  IF "RE = :END OP RE.PRES :SUJET :ROOT
END
```

In the following case, make a further distinction.

```
TO IR.PRES :SUJET :ROOT
  IF "O = LAST :ROOT OP OIR.PRESENT :ROOT
  OP XIR.PRESENT :ROOT
END
```

The rest is yours.

24. The relevant change to make is this

```
IF MEMBER? REQUEST :ANSWER PR [YUP!]
```

25. This version of ADDQUIZ takes a number as input and keeps giving problems until that many problems have been answered correctly.

```

TO ADDQUIZ :TIMES
  IF :TIMES = 0 STOP
  IF ADDQ RANDOM 13 RANDOM 13 ADDQUIZ :TIMES - 1
  ELSE ADDQUIZ :TIMES
END

```

```

TO ADDQ :N1 :N2
  PRINT1 ( SE :N1 "+ :N2 "" = ' )
  IF (:N1 + :N2) = FIRST RQ PR [YAY!] OP "TRUE
  PR ( SE "NOPE, :N1 "+ :N2 "" = :N1 + :N2 )
  OP "FALSE
END

```

Notice that the only differences in ADDQ are that it outputs TRUE if the answer is correct and FALSE otherwise.

26. Here is one form. Are there bugs? Is there a cleaner way?

```

TO ADDQUIZ :MAX :TIMESRIGHT :TIMESWRONG
  IF :TIMESWRONG = 2 STOP
  IF :TIMESRIGHT = 3 ADDQUIZ :MAX + 1 0 0 STOP
  IF ADDQ RANDOM :MAX RANDOM :MAX
    THEN ADDQUIZ :MAX :TIMESRIGHT + 1
    :TIMESWRONG STOP
  ELSE ADDQUIZ :MAX :TIMESRIGHT
  :TIMESWRONG STOP
  ADDQUIZ :MAX :TIMESRIGHT :TIMESWRONG + 1
END

```

Start it by typing

```
ADDQUIZ 4 3 0
```

27. The logic we are trying to add is this: ADDQ is told what the problem is and how many tries the person has already made.

```
TO ADDQ :TRIES :N1 :N2
```

If that number (TRIES) is 2, ADDQ should give the correct answer and output FALSE.

```
IF :TRIES = 2 PR ( SE :N1 " + :N2 " = :N1 + :N2 ) OP  
"FALSE
```

Otherwise, ADDQ should state the problem as before and allow the person another try. If the person gets the right answer, ADDQ says YAY and outputs TRUE, as it did before.

```
PRINT1 ( SE :N1 " + :N2 " = ' )  
IF (:N1 + :N2) = FIRST RQ PR [YAY!] OP "TRUE
```

But if the person gets the wrong answer, ADDQ should say "try again," give the same problem as before, and know that the person has taken one more try at answering it.

```
PRINT [TRY AGAIN]  
OP ADDQ :TRIES + 1 :N1 :N2
```

Of course, ADDQUIZ must start ADDQ by telling it that no tries have yet been made.

```
IF ADDQ 0 RANDOM :MAX RANDOM :MAX etc.
```

The completed program might look like this.

```

TO ADDQUIZ :MAX :TIMESRIGHT :TIMESWRONG
  IF :TIMESWRONG = 2 STOP
  IF :TIMESRIGHT = 3 ADDQUIZ :MAX + 1 0 0 STOP
  IF ADDQ 0 RANDOM :MAX RANDOM :MAX
    ADDQUIZ :MAX :TIMESRIGHT + 1
    :TIMESWRONG STOP
  ELSE ADDQUIZ :MAX :TIMESRIGHT
    :TIMESWRONG + 1 STOP
END

```

```

TO ADDQ :TRIES :N1 :N2
  IF :TRIES = 2 PR ( SE :N1 " + :N2 " = :N1 + :N2 )
  OP "FALSE
  PRINT1 ( SE :N1 " + :N2 "' = ' )
  IF (:N1 + :N2) = FIRST RQ PR [YAY!] OP "TRUE
  PRINT [TRY AGAIN]
  OP ADDQ :TRIES + 1 :N1 :N2
END

```

28. PICK can select some element from the STATES list. Each element of the STATES list contains both a question as its FIRST and an answer as its LAST (or BUTFIRST). This is just what QA needs. The hitch is that if we simply type

```
QA FIRST PICK :STATES LAST PICK :STATES
```

Logo will run PICK twice, and each time PICK is run it may pick a different element from the list! QA needs to take the FIRST and LAST (or BUTFIRST) of the same element.

The first thing to resolve is whether we use the LAST or BUTFIRST of the element. It makes a big difference, since the LAST is a word and the BUTFIRST is a list.

Since QA compares its :ANSWER with a REQUEST (which is always a list), we might as well use BF. One way STATESQUIZ might work is this:

```
TO STATESQUIZ
  REPEAT 5 [MAKE "QLIST PICK :STATES QA FIRST :QLIST
    BF :QLIST]
END
```

An alternative that is neater in a few ways is this:

```
TO STATESQUIZ
  REPEAT 5 [STATEQA PICK :STATES]
END
```

```
TO STATEQA :QLIST
  QA FIRST :QLIST BF :QLIST
END
```

29. The BF of [IOWA [DES MOINES]] is [[DES MOINES]] but we want [DES MOINES] to compare to the sentence typed to REQUEST. In this case, we would have been better off taking the LAST rather than the BUTFIRST. How do we resolve the problem?

The real problem is that the database :STATES has both words and lists as possible answers. This makes it difficult to check for equality.

If the answer-part of each element of the :STATES list was always a list, we could consistently choose the FIRST for the question, and the LAST for the answer.

So, we make states differently:

```
MAKE "STATES [[OHIO [COLUMBUS]] [[NEW YORK]
              [ALBANY]] [GEORGIA [ATLANTA]] [IOWA
              [DES MOINES]]]
```

And we redefine STATEQA

```
TO STATEQA :QLIST
  QA FIRST :QLIST LAST :QLIST
END
```

30. 'Tis all yours!

31. The changes would be in the form:

```
IF :CHAR = "F RUN.AND.RECORD SE "FD 10 * :MULTIPLE
IF :CHAR = "R RUN.AND.RECORD SE "RT 15 * :MULTIPLE
IF :CHAR = "L RUN.AND.RECORD SE "LT 15 * :MULTIPLE
```

There are two subtleties. One is that the command lines read:

```
IF :CHAR = "F RUN.AND.RECORD SE "FD 10 * :MULTIPLE
```

and not (more simply)

```
IF :CHAR = "F RUN.AND.RECORD [FD 10 * :MULTIPLE]
```

The reason is that although the second version will RUN correctly, the command that will be LPUT on the history list will be, literally, [FD 10 \* :MULTIPLE] rather than the desired [FD 30] or whatever it is.

RUN and REPEAT are the only primitives that are capable of evaluating what is inside a list. Everything else just treats it as text without meaning.

Also, remember that TOGGLEPEN must be edited to record its ups and downs.

32. Lines like IF :CHAR = "< RCIRCLE :SIZE would be needed, but you must provide the mechanism for setting :SIZE just as you had for the forward and turning commands.

If you allow ARC (first introduced in the section on OUTPUT) to take an angle input as well as the two it now takes, SEGMENTS and CHORD, the new procedures RCIRCLE and LCIRCLE can then be defined by using ARC with angles of 18 and -18 respectively.

33. The procedure itself is very straightforward. It depends on lists of the verbs, nouns, proper names, and so forth.

So far, procedures to output verbs and proper names have been created, as has a global variable containing adverbs. The following definition of MADLIB further assumes procedures NOUNS and ADJECTIVES that must be created on the model of ACTIONS and PEOPLE.

```

TO MADLIB :TEXT
  OP MAD "V ACTIONS MAD "N NOUNS MAD "PN
  PEOPLE MAD "ADV :ADVERBS
  MAD "ADJ ADJECTIVES :TEXT
END

```

34. With the example that was given, all that is needed is to check both the words themselves (i.e., PN LOVES PN<comma> BUT PN CAN'T STAND PN<period>) and the butlast of the words (i.e., P LOVE PN BU P CAN' STAN PN). All of the PNs will be caught this way. The test

```
IF BL FIRST :CONTEXT = :KEY
```

will do that job. If the butlast of the word is KEY, then the last will be the punctuation mark. By picking an alternate and wording the punctuation mark to the end of it,

```
WORD PICK :ALT LAST FIRST :CONTEXT
```

the original punctuation has been restored. Finally, this word must be integrated into the developing sentence just as if the punctuation problem had not occurred.

```
OP SE WORD PICK :ALT LAST FIRST :CONTEXT
MAD :KEY :ALT BF :CONTEXT
```

Altogether the new line of the procedure is:

```
IF BL FIRST :CONTEXT = :KEY
  OP SE WORD PICK :ALT LAST FIRST :CONTEXT
  MAD :KEY :ALT BF :CONTEXT
```

There is a problem. What if one of the keywords were N, as in problem 33, and one of the words of the sentence were "NO"? But last of the word NO would falsely match the keyword, and NO would be replaced with a noun!

A more complex and sophisticated procedure could be written, but the best solution is to make keywords clearly distinct from text. If keywords all began with some non-text character, so that they could never be generated from a text word (as happened when N was generated from NO), the problem would be solved.

Recommendation: Begin keywords with <period>.

Thus, madlib sentences would look like this:

[.PN LOVES .PN, BUT .PN CAN'T STAND .PN.]

Note that MAD never tests for the special keyword marker. The marker just serves to prevent mishaps.

Does the order in which the tests are performed matter?

```
TO MAD :KEY :ALT :CONTEXT
  IF :CONTEXT = [] OP []
  IF ( FIRST :CONTEXT ) = :KEY OP SE PICK :ALT
    MAD :KEY :ALT BF :CONTEXT
  IF BL FIRST :CONTEXT = :KEY OP SE WORD PICK :ALT
    LAST FIRST :CONTEXT MAD :KEY :ALT BF :CONTEXT
  OP SE FIRST :CONTEXT MAD :KEY :ALT BF :CONTEXT
END
```

35. Let's title the procedure this way.

```
TO MADLIB :TEXT :KEYS
```

The logic is that if there are no keywords at all to find and replace, then the text must be returned as it is.

```
IF EMPTY? :KEYS OP :TEXT
```

If there are keys to replace, then

1) using the first of them, replace each instance of it in the text with a suitable alternative (this is accomplished by MAD) and

2) use that as the text in which to search for the remaining keys. This is the purpose of MADLIB, itself, and is thus the recursive step.

Worded more like the program, we are to output the MADLIB of and a list of the remaining keys.

Skipping over a detail, the Logo might look something like this:

```
OP MADLIB ( MAD FIRST :KEYS somethingorother :TEXT )  
BF :KEYS
```

The “somethingorother” needs some thinking.

In previous situations, the key words bore no relation to the procedures or variables that contained the corresponding lists. This is inconvenient, since there is no way to know from looking at the key word, just where to find its substitutes.

But that can be corrected. Abandon the old design of having V refer to a procedure ACTIONS, and ADV to a variable ADVERBS.

From now on, we must be consistent about using either procedures or variables. Further, the keyword will be the name of the variable or the title of the procedure.

Choosing to go with global variables, we can then say that if MAD's KEY is the first of MADLIB's KEYS, MAD's ALT will be the THING of the first of MADLIB's KEYS. MADLIB would then look like this:

```
TO MADLIB :TEXT :KEYS
  IF EMPTY? :KEYS OP :TEXT
  OP MADLIB (MAD FIRST :KEYS THING FIRST :KEYS
    :TEXT) BF :KEYS
END
```

If we chose to use procedures titled by KEY, then MAD's ALT would be the result of RUNNING the first of MADLIB's KEYS.

```
TO MADLIB :TEXT :KEYS
  IF EMPTY? :KEYS OP :TEXT
  OP MADLIB (MAD FIRST :KEYS RUN (SE FIRST :KEYS)
    :TEXT) BF :KEYS
END
```

The most important element here became the willingness to abandon some old designs and rethink the relationship between parts of the problem.

36. GREET needs to look at what OUTPUT.NAME gives it and determine, first, if the result is a name or a response. Here is a possible method:

```
TO RESPOND :NAME.OR.PHRASE
  IF WORD? :NAME.OR.PHRASE GREET
  :NAME.OR.PHRASE STOP
  PRINT :NAME.OR.PHRASE
END
```

```
TO FRIENDLY
  PR [WHAT'S YOUR NAME?]
  RESPOND OUTPUT.NAME REQUEST
END
```

37. Just before the neutral answer ( OP [I WAS JUST CURIOUS] ) the procedure must look for negatives, and should respond appropriately if it finds any.

```
IF FIND? [WON'T NONE DON'T NOT NO] :SENT OP
  [SORRY I ASKED]
```

FIND? is simply a fancy MEMBER?

```
TO FIND? :ITEMS :LIST
  IF EMPTY? :ITEMS OP "FALSE
  IF MEMBER? FIRST :ITEMS :LIST OP "TRUE
  OP FIND? BF :ITEMS :LIST
END
```

38. Any of a number of strategies will work. Be of good cheer! The task of deciding which approach to take should be simple for anyone who has gotten this far.

39. If punctuation only comes at the ends of words, removing it is quite simple.

```
TO NOPUNC :WORD
  IF MEMBER? LAST :WORD [“ , . ! ?] OP BL :WORD
  OP :WORD
END
```

A more general solution, more powerful but slower, is:

```
TO NOPUNC :WORD
  IF EMPTY? :WORD OP “
  IF MEMBER? FIRST :WORD [“ , . ! ?] OP NOPUNC BF
  :WORD
  OP WORD FIRST :WORD NOPUNC BF :WORD
END
```

In either case, change FIRST :S to NOPUNC FIRST :S throughout the CHECK procedure.

41. Sorry. From here on in, you are on your own!

L

L

L

L

L

L

L

*LOGO COMMAND*  
*GLOSSARY*

L

L

L

L

L

L

L

L

L

L

L

L

L

# L

## LOGO COMMAND GLOSSARY

---

+ Additional commands available only in *Logo PLUS*.  
Refer to the *Getting Acquainted with Logo PLUS* booklet for a summary of these commands.

### Categorized Listing of Commands

#### Graphics Commands

BACK	+ PENERASE
BACKGROUND	PENUP
CLEARSCREEN	RIGHT
+ COLORUNDER	SETHEADING
+ DOT	+ SDOT
+ DOT?	+ SDOT?
+ DOTXY	+ SDOTXY
DRAW	SETX
+ FILL	SETXY
FORWARD	SETY
FULLSCREEN	SHOWTURTLE
HEADING	SPLITSCREEN
HIDETURTLE	TEXTSCREEN
HOME	TOWARDS
LEFT	TURTLESTATE
NODRAW	WRAP
NOWRAP	XCOR
PENCOLOR	YCOR
PENDOWN	+ ZOOM

#### Graphics Screen Text Commands

(*Logo PLUS* only)

GCURSOR	GREADCHARACTER
GCURSORPOS	GREQUEST
GMODE	GSTYLE
GPRINT	GWRITE
GPRINT1	<Open-Apple> W

### Shape Commands (*Logo PLUS* only)

CLEARSHAPES	STAMP
COPYSHAPE	STAMPXY
EDSHAPE	TCOLOR
LOCKHEADING	TOTALSHAPES
SETCOLOR	TSHAPE
SETSHAPE	TSIZE
SETSIZE	UNLOCKHEADING

### Numeric Operations

+	NUMBER?
-	QUOTIENT
*	RANDOM
/	RANDOMIZE
>	REMAINDER
<	ROUND
ATAN	SIN
COS	SQRT
INTEGER	

### Word and List Operations

=	LAST
BUTFIRST	LIST
BUTLAST	LIST?
COUNT	LPUT
EMPTY?	MEMBER?
FIRST	SENTENCE
FPUT	WORD
ITEM	WORD?

### Naming

LOCAL	THING
MAKE	THING?

**Conditionals**

ALLOF	IFTRUE
ANYOF	NOT
ELSE	TEST
IF	THEN
IFFALSE	

**Control**

GO	RUN
GOODBYE	STOP
OUTPUT	TOPLEVEL
REPEAT	+ WAIT

**Input and Output**

ASCII	PADDLEBUTTON
CHAR	PRINT
CLEARINPUT	PRINT1
CLEARTEXT	+ PRINTSCREEN
CURSOR	RC?
+ CURSORPOS	READCHARACTER
+ LOWERCASE	REQUEST
+ NOTE	SHOWTEXT
OUTDEV	+ UPPERCASE
PADDLE	

**Debugging**

CONTINUE	+ SHOW
NOTRACE	TRACE
PAUSE	+ TRACEBACK

**Managing the Workspace**

+ COPYDEF	+ NAMES
DEFINE	PRINTOUT
EDIT	TEXT
END	+ TITLES
ERASE	TO
ERNAME	

### Handling Files

- + APPEND
- + BLOAD
- + BSAVE
- CATALOG
- + CLOSE
- + COPY
- + DELETE
- + DISKREAD
- + DISKWRITE
- DOS
- + DPOSITION
- + DPRINT
- + EOF?
- + ERASEDIR
- ERASEFILE
- ERASEPICT
- + ERASESHAPES
- + FILE?
- + LOCK
- + MAKEDIR
- + ONLINE
- + OPEN
- + PREFIX
- READ
- + READFONT
- READPICT
- + READSHAPES
- + READTEXT
- + RENAME
- SAVE
- SAVEPICT
- + SAVESHAPES
- + SAVETEXT
- SETDISK
- + SETPREFIX
- + UNLOCK
- + VCAT

### Miscellaneous Commands

- .ASPECT
- .BPT
- .CALL
- .CONTENTS
- .DEPOSIT
- .EXAMINE
- .GCOLL
- .NODES
- ; (semicolon)

### Command Summary

- + Addition
- Subtraction (two inputs) and  
negation (one input).
- \* Multiplication
- / Division (always outputs a decimal  
value).

---

=	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. (In this case, a space is needed before the = sign.) If both inputs are lists, compares them to see if their corresponding elements are equal. Outputs TRUE or FALSE accordingly.
>	Outputs TRUE if its first input is greater than its second, otherwise FALSE (inputs must be numbers).
<	Outputs TRUE if its first input is less than its second, otherwise FALSE (inputs must be numbers).
; (semicolon)	Causes the rest of the line not to be evaluated. Useful for including comments in procedures and procedure titles.
ALLOF	Takes a variable number of inputs (default is two) and outputs TRUE if all are TRUE. If there are more than two inputs, there must be an opening parenthesis before ALLOF, and a space and a closing parenthesis after the last input.
ANYOF	Takes a variable number of inputs (default is two) and outputs TRUE if at least one is TRUE. If there are more than two inputs, there must be an opening parenthesis before ANYOF, and a space and a closing parenthesis after the last input.

- ASCII** Takes a character as input and outputs the number that is the ASCII code of that character.
- .ASPECT** Changes the vertical scale at which Logo graphics are drawn. Takes one numeric input and uses this to change the scale factor. The default value for the factor is 0.8. This command is included because not all TV monitors have the same amount of vertical deflection. Consequently, turtle programs that are supposed to draw squares and circles may instead appear to draw rectangles and ellipses. If so, the **.ASPECT** command can be used to attempt to compensate for the distortion. Note that changing the factor will change the limits for permissible y-coordinates. If a value too different from 0.8 is used, lines will be drawn at the correct angle, but the turtle pointer may not always appear to be pointing exactly along the line.
- ATAN** Takes two inputs and then outputs (in degrees) the arctangent of the quotient. The output ranges from 0 to less than 360, with the quadrant corresponding to the signs of the two inputs. If the second input is negative, it must be enclosed by parentheses. See also **TOWARDS**.
- BACK** Moves the turtle in the opposite direction from which it is pointing by the amount specified.  
Abbreviated: **BK**.

- BACKGROUND** Takes a number 0 through 6 as input and sets the color of the graphics screen background. Abbreviated: BG.
- .BPT** Breaks out of Logo into the Apple monitor (for use in Logo system debugging.) For *Terrapin Logo*, useful entry addresses are 1BF9, which is a “cold start” address to use after Logo has been started before; and the “warm start” address 1BFC, for attempting to recover after a system crash. For *Logo PLUS*, the “cold start” address is 4003; the “warm start” address is 4006. After restarting Logo at the cold start address, all procedures are lost; it is just like typing GOODBYE. The warm start address leaves all variables and procedures intact. In fact all local variables still have the values they had at the time Logo was interrupted. To return to Logo using these addresses, type the characters for the address followed by the letter G for GO (for example, 4003G). The best way for most users to return to Logo is to type <CTRL> Y, <RETURN> then <CTRL> G, <RETURN>. This is equivalent to a “warm start.”
- BUTFIRST** If the input is a list, outputs a list containing all but the first element. If the input is a word, outputs a word containing all but the first

	character. Gives an error when called with the empty word or the empty list. Abbreviated: BF.
<b>BUTLAST</b>	If the input is a list, outputs a list containing all but the last element. If the input is a word, outputs a word containing all but the last character. Gives an error when called with the empty word or the empty list. Abbreviated: BL.
<b>.CALL</b>	Calls a machine language subroutine in memory. The address of the subroutine is the first input; the second input is stored in a memory location for the routine to examine. This primitive allows users to provide their own special-purpose primitives and interface them to Logo. See the Technical chapter.
<b>CATALOG</b>	Prints the names of files on the currently mounted disk.
<b>CHAR</b>	Takes an integer as input and outputs the character whose ASCII code is that integer.
<b>CLEARINPUT</b>	Clears the character input buffer of any typed-ahead characters.
<b>CLEARSCREEN</b>	Clears the graphics screen. Does not change the turtle's position, the pen state, or whether the turtle is hidden or shown. Abbreviated: CS.
<b>CLEARTEXT</b>	Clears the text screen and places the cursor in the upper left corner.

- .CONTENTS** Returns a list of all words known to Logo. This includes names of variables, procedures, and words used in procedures. One use might be an editing program that, for each procedure defined, asks you whether you want to delete it. **TEXT** and **THING?** are useful primitives to use with the elements of this list. Caution: Use of this primitive interferes with garbage collection of “truly worthless atoms”. These are the no-longer-used words that Logo has in memory, usually as the result of typing errors. If you run short of memory, it might be because an old list from **.CONTENTS** is around somewhere keeping Logo from recovering the storage associated with no-longer-needed words. Before using **.CONTENTS**, you should type **.GCOLL** to force a garbage collection, leaving you with as tidy a workspace as possible
- CONTINUE** Resumes execution after a **PAUSE** or **<CTRL> Z**. Abbreviated: **CO**.
- COS** Outputs the cosine of its input (an angle in degrees).
- COUNT** Takes a word or list as input and returns the number of elements contained in that input. Examples: **COUNT "LOGO** returns 4; **COUNT 12** returns 2 (the number of characters in the word 12); **COUNT [1 2 3 4 5 6]** returns 6;

- COUNT [CHILI [HOT DOG] HAMBURGER] returns 3. The list [HOT DOT], although made up of more than one element itself, is counted as one element of the input list.
- CURSOR** Takes two inputs, column and row, and positions the cursor there. Columns are 0-39, rows are 0-23. 0,0 is upper left. Use the CH and CV locations in the ADDRESSES file to determine the cursor's current position. See the Utilities chapter.
- DEFINE** Takes two inputs. First is a name, second is a list. Each element of this list must be a list itself. The first element is the list of inputs to the procedure. (If there are no inputs to the procedure, the first element should be the empty list.) Each subsequent element is a list corresponding to one line of the procedure being defined. For example, DEFINE "TRIANGLE [[:SIZE] [REPEAT 3 [FD :SIZE RT 120]]]. See TEXT. Note that one normally uses TO rather than DEFINE in order to define procedures.
- .DEPOSIT** Takes two numeric inputs, an address and a value, and deposits a byte of data at the designated memory location. See the Technical chapter.

**DOS**

Takes one input (word or list), and interprets it as commands to DOS. DOS [RENAME GMAE.LOGO, GAME.LOGO] will rename something saved with SAVE "GMAE. To "unlock" locked files (those that appear with an asterisk in the CATALOG listing) type, for example, DOS [UNLOCK ADDRESSES.LOGO]. The following DOS commands are available in this manner: DELETE, CATALOG, LOCK, UNLOCK, RENAME, BLOAD, BRUN, BSAVE. See the Apple DOS manual for information on the syntax of DOS commands.

**DRAW**

Clears the graphics screen, homes the turtle to the center of the screen, shows the turtle, and puts the pen down. It does not change the background or pen color.

**EDIT**

Enters edit mode. If a procedure name is included as an input, that procedure will be in the editor. If no input is specified, enters edit mode with the previous contents of the screen editor buffer, or the most recently defined (or PO'd) procedure if the previous contents are unretrievable. Can also take a list of procedures to place in the editor or the auxiliary words ALL, NAMES, or PROCEDURES. The Appendix also contains a description

- of keystroke commands inside the editor. Abbreviated: ED.
- ELSE** Used in IF...THEN...ELSE.... See IF.
- EMPTY?** Takes one input, a word or list, and returns TRUE if the value is either the empty word " or the empty list [ ]; otherwise returns FALSE.
- 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.
- ERASE** Erases designated procedure from workspace. Can also take qualifiers ALL, NAMES, PROCEDURES or a list of procedures. Signals an error if there is no procedure with the given name. For convenience, the input to erase is not evaluated (i.e. Logo will not try and run the procedure being erased.); to erase a procedure called LOOKUP, type ERASE LOOKUP. See also RUN. Abbreviated: ER.
- ERASEFILE** Removes from the disk a file saved with SAVE. Takes a file name as input, which must begin with a " mark.

- ERASEPICT** Removes a picture that has been saved on the disk using **SAVEPICT**. Takes a picture name as input, which must begin with a " mark.
- ERNAME** Takes a name as input and removes that name from the workspace. Signals an error if the name is not used. Note that unlike **ERASE**, the input to **ERNAME** is evaluated. Thus, to erase the name **TEMP**, type **ERNAME "TEMP**. If you type **ERNAME TEMP**, Logo assumes **TEMP** to be a procedure and tries to run it.
- .EXAMINE** Takes one input. Outputs the value of the byte at the specified address. See also the Technical chapter.
- FIRST** If the input is a list, outputs the first element. If the input is a word, outputs the first character. Gives an error when called with the empty word or the empty list.
- FORWARD** Moves the turtle in the direction in which it is pointing by the amount specified. Abbreviated: **FD**.
- FPUT** Takes two inputs. Second input must be a list. Outputs a list consisting of the first input followed by the elements of the second input. Therefore, if the first input is a list, for example **FPUT [A B] [C D]**, the result will be **[[A B] C D]**. See also **LPUT**, **LIST**, and **SENTENCE**.

FULLSCREEN	In graphics mode, gives full graphics screen. See SPLIT-SCREEN and TEXTSCREEN. Equivalent to <CTRL> F.
.GCOLL	Forces a garbage collection in order to free up the maximum amount of usable workspace.
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.)
GOODBYE	Clears workspace and restarts Logo. It does not, however, clear the machine-language area.
HEADING	Outputs the turtle's heading as a decimal number. The heading ranges from 0 to less than 360. When the turtle has a heading of 0 it is pointing straight up.
HIDETURTLE	Makes the turtle pointer disappear. Abbreviated: HT.
HOME	Moves the turtle to the center of the screen, pointing straight up.
IF	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. The <condition> must be a Logo expression which outputs "TRUE or "FALSE. A Logo variable whose value is "TRUE or "FALSE satisfies this condition, as do various testing functions such as <, >, =, ALLOF, ANYOF, NOT, and primitives that end in a ? mark. Both <action1> and <action2> may be any number of Logo expressions.
- IFFALSE** Executes rest of line only if result of preceding TEST was false. Abbreviated: IFF.
- IFTRUE** Executes rest of line only if result of preceding TEST was true. Abbreviated: IFT.
- INTEGER** Takes one numeric input and outputs the integer part, ignoring the fractional part.
- ITEM** Takes two inputs, a number (represented here by *n*) and a word or list, and outputs the *n*th element of the list. An error message is displayed if *n* exceeds the number of elements in the list.
- LAST** If input is a list, outputs the last element. If input is a word, outputs the last character. Gives an error when called with the empty word or the empty list.

- LEFT** Rotates the turtle. Takes an input that specifies the number of degrees to rotate. Abbreviated: LT.
- LIST** Takes a variable number of inputs (two by default) and outputs a list of the inputs. Therefore, if the first and second inputs are lists, for example LIST [A B] [C D], the result will be [[A B][C D]]. If there are more than two inputs, there must be an opening parenthesis before LIST, and a space and closing parenthesis after the last input. See also FPUT, LPUT, and SENTENCE.
- LIST?** Outputs TRUE if its input is a list. See also WORD? and NUMBER?.
- LOCAL** Takes a word as input, allowing for the creation of local variables not declared in the title line of a procedure. When MAKE is used after the command LOCAL, the named variable is treated as a local variable and is not visible to the entire workspace. Use only in procedures.
- LPUT** Takes two inputs. Second input must be a list. Outputs a list consisting of the elements of the second input followed by the first input. Therefore, if the first input is a list, for example LPUT [A B] [C D], the result will be [C D [A B]]. See also FPUT, LIST, and SENTENCE.

<b>MAKE</b>	Takes two inputs, the first of which must be a word. It treats the word as a variable name, and makes the second input be the value of the variable.
<b>MEMBER?</b>	Takes two inputs, which can be a single character, a word or a list. Returns TRUE if the first input is an element of the second input. For example, "A is a member of "CAT, "CAT is a member of [BIRD CAT DOG] and [D E F] is a member of [[A B C] [D E F][G H I]]. Otherwise returns FALSE. For example, 2 is not a member of [1 3 5] and "AT is not a member of "CAT.
<b>.NODES</b>	Outputs the number of currently free nodes. To obtain a true count of free memory, type .GCOLL before typing .NODES.
<b>NODRAW</b>	Exits graphics mode, giving a clear text page with the cursor homed in the upper left-hand corner of the screen. Abbreviated: ND.
<b>NOT</b>	Outputs TRUE if its input is FALSE; outputs FALSE if its input is TRUE.
<b>NOTRACE</b>	Turns off tracing. (See TRACE.)
<b>NOWRAP</b>	Exits wrapping mode. Any command that would normally cause the turtle to move off one edge of the screen and onto the opposite edge instead results in an error.

- NUMBER?** Outputs TRUE if its input is a number. See also WORD? and LIST?.
- OUTDEV** Takes as input a number designating a slot on the Apple mother board. After executing this command, everything except the instruction typed will be sent to the device plugged into the designated slot. The output may also be echoed to the screen, depending on the particular computer setup. OUTDEV 0 specifies output to the screen. A “slot number” number greater than 255 is interpreted as the address of a user-supplied assembly language routine to be called in place of the usual character output primitive. Typing <CTRL> SHIFT-M will restore output to the screen.
- OUTPUT** Takes an input. Causes the current procedure to stop and output the input to the calling procedure. If the input has to be evaluated, it outputs the result of that evaluation.  
Abbreviated: OP.
- PADDLE** Takes a number 0 through 3 as input, which specifies the paddle. Outputs a number 0-255 depending on the setting of the appropriate paddle dial. One example that can be used with either two paddles or a joystick is SETXY PADDLE 0 PADDLE 1. This instruction sets the turtle’s position to the output of

the joystick, which can be controlled by hand.

## PADDLEBUTTON

Take a number 0 through 2 as input and outputs TRUE or FALSE depending on whether the button on the corresponding paddle is pressed. One example of its use is IF PADDLEBUTTON 0 = "TRUE THEN CLEARSCREEN. On the Apple II, paddle 3 does not have an associated paddle button. The <Open-Apple> key on Apple computers is equivalent to PADDLEBUTTON 0; the <Closed-Apple> or <Option> key is equivalent to PADDLEBUTTON 1.

## PAUSE

Stops execution and allows command lines to be evaluated in the current local environment. Equivalent to interrupt character <CTRL> Z. Execution is resumed with CONTINUE, provided no errors have occurred.

## PENCOLOR

Takes a number from 0 through 6 and sets the color of the lines that the turtle will draw. Abbreviated: PC.

## PENDOWN

Causes the turtle to leave a trail when it moves. This is the default state and it is changed by PENUP. Abbreviated: PD.

## PENUP

Causes the turtle to move without leaving a trail. Abbreviated: PU.

- POTS** Abbreviation for PRINTOUT TITLES. See PRINTOUT.
- PRINT** Variable number of inputs (default is 1). Prints the input on the screen. Lists are printed in “sentence” form, without the outermost level of brackets. The next PRINT will print on the next line of the screen. If there are multiple inputs, as in (PRINT 1 2 3), the inputs will be printed on one line, separated by spaces. Note that for multiple inputs, the entire statement must be enclosed in parentheses. If the input to PRINT is a procedure, it will not print the procedure, but will execute the procedure assuming the procedure will output something to print. See also PRINTOUT. Abbreviated: PR.
- PRINT1** Like PRINT, but does not terminate output line with a return. With multiple inputs, does not print spaces between elements.
- PRINTOUT** If given a procedure name as input, prints out the text of the procedure. If given no input, prints out the last procedure defined, edited or printed out. For convenience, the input is not evaluated; thus to see a procedure called CIRCLE, you would type PO CIRCLE and not PO "CIRCLE. Can also take auxiliary words: ALL, NAMES, TITLES, PROCEDURES or a list of proce-

dures names. POTS is an abbreviation for PRINTOUT TITLES.

See also PRINT and RUN.

Abbreviated: PO.

## QUOTIENT

Outputs the integer quotient of its two inputs. (If the inputs are not integers, it first rounds them to the nearest integer.) If the second input is negative, it must be enclosed by parentheses.

## RANDOM

Takes one input—a positive integer  $n$ —and outputs an integer between 0 and  $n-1$ . Identical sequences of calls to RANDOM will yield repeatable sequences of random numbers each time Logo is restarted unless the seed for the random number generator is RANDOMIZED.

## RANDOMIZE

Randomizes the seed for RANDOM. If given an explicit input, sets the random number seed to that number. For example, after each execution of (RANDOMIZE 259) the same sequence of random numbers will be generated, different numbers result in different sequences. Note that () are needed around RANDOMIZE if an input is used, such as (RANDOMIZE 259).

## RC?

Outputs TRUE if a keyboard character is pending (i.e., if READCHARACTER would output immediately, without waiting for the user to press a key), otherwise outputs FALSE.

- READ** Reads a file from disk. Destroys any graphics display if only 64K is available. Takes file name as input, which must begin with a " mark.
- READCHARACTER** Outputs the least recent character in the character buffer, or if empty, waits for an input character. See CLEARINPUT. See the explanation of the INSTANT program on the Utilities disk for an example of its use. Abbreviated: RC.
- READPICT** Reads a picture that has been stored on disk and displays it on the graphics screen. Takes picture name as input, which must begin with a " mark.
- REMAINDER** Outputs the integer remainder of its first input divided by its second. (If the inputs are not integers, it first rounds them to the nearest integer.) If the second input is negative, it must be enclosed by parentheses.
- REPEAT** Takes a number and a list as input. RUNs the list the designated number of times.
- REQUEST** Waits for an input line to be typed by the user and terminated with RETURN. Outputs the line as a list. Abbreviated: RQ.
- RIGHT** Rotates the turtle. Takes an input that specifies the number of degrees to rotate. Abbreviated: RT.

- ROUND** Outputs the nearest integer to its input.
- RUN** Takes a list as input. Executes the list as if it were a typed in command line. Note: 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 the toplevel command line, which is 255. Otherwise, an error is signalled. To write a procedure that erases a named procedure (for use in interactive programming, for example), use the following format (which can also be used with PRINTOUT):
- ```
TO ERPROC :PROCEDURE
  RUN LIST "ER :PROCEDURE
END

ERPROC "SAMPLE
```
- SAVE** Saves the contents of the workspace on disk. Destroys any graphics display if only 64K is available. Takes file name as input, which must begin with a " mark. To save a selected group of procedures, include them in a list as part of the SAVE command, like this: (SAVE "SHAPES [SQUARE TRI HEX]) You must include the parentheses.
- SAVEPICT** Save on disk the picture on the screen. Takes picture name as input, which must begin with a " mark.

- SENTENCE** Variable number of inputs (default 2). If inputs are all lists, combines all their elements into a single list. If any inputs are words (or numbers), they are regarded as one-word lists in performing this operation. Therefore, if the first and second inputs are lists, for example `SENTENCE [A B] [C D]`, the result will be `[A B C D]`. If there are more than two inputs, there must be an opening parenthesis before `SENTENCE`, and a space and closing parenthesis after the last input. See also `FPUT`, `LPUT`, and `LIST`. Abbreviated: `SE`.
- SETDISK** Takes two numeric inputs—a drive number and a slot number—and activates the disk drive at that location. All subsequent file operations will use that disk drive. Default is `SETDISK 1 6`.
- SETHEADING** Rotates the turtle to the direction specified. Input determines number of degrees. Zero is straight up, with heading increasing clockwise. Abbreviated: `SETH`.
- SETX** Moves the turtle horizontally to the specified coordinate.
- SETXY** Takes two numeric inputs. Moves the turtle to the specified point. 0,0 is screen center. When the y-coordinate (second input) is negative, it must be enclosed by parentheses.

|             |                                                                                                                                                                                                                                                                                                                                                                                |
|-------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SETY        | Moves the turtle vertically to the specified coordinate.                                                                                                                                                                                                                                                                                                                       |
| SHOWTEXT    | Prints the contents of the edit buffer to the current output device, which is normally the screen. Send output to a printer using OUTDEV.                                                                                                                                                                                                                                      |
| SHOWTURTLE  | Makes the turtle pointer appear. This is the default state and it is changed by HIDE TURTLE. Abbreviated: ST.                                                                                                                                                                                                                                                                  |
| SIN         | Outputs the sine of its input (an angle in degrees).                                                                                                                                                                                                                                                                                                                           |
| SPLITSCREEN | In graphics mode, gives mixed text/graphics screen. See FULLSCREEN and TEXTSCREEN. Equivalent to <CTRL> S.                                                                                                                                                                                                                                                                     |
| SQRT        | Takes a positive number as input and outputs the square root of that number.                                                                                                                                                                                                                                                                                                   |
| STOP        | Causes the current procedure to stop and return control to the calling procedure. Note that STOP does not mean the same thing as END. STOP is a primitive which when executed causes the current procedure to stop executing, and returns control to the previous procedure (or toplevel). END is used in the editor to indicate where a procedure ends. It is never executed. |
| TEST        | Tests a condition to be used in conjunction with IFTRUE and                                                                                                                                                                                                                                                                                                                    |

|            |                                                                                                                                                                                                                                                                                                                                |
|------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|            | <p>IFFALSE. TEST takes one input, which must be either TRUE or FALSE. The result of the most recent TEST in each procedure is used by IFTRUE and IFFALSE, and is local to the current procedure.</p>                                                                                                                           |
| TEXT       | <p>Takes a procedure name as input and outputs procedure text as a list. The procedure name must start with " or Logo will run the procedure. If the procedure has not been defined, TEXT outputs []. If instead the input is the name of a Logo primitive, it outputs the primitive's name (i.e., the input). See DEFINE.</p> |
| TEXTSCREEN | <p>In graphics mode, gives full text screen. See SPLITSCREEN and FULLSCREEN. Equivalent to &lt;CTRL&gt; T.</p>                                                                                                                                                                                                                 |
| THEN       | <p>Used with IF... THEN... ELSE... See IF.</p>                                                                                                                                                                                                                                                                                 |
| THING      | <p>Outputs the value of its input, which must be a word. THING "XXX is equivalent to :XXX. (Note that this command allows you extra levels of evaluation.)</p>                                                                                                                                                                 |
| THING?     | <p>Outputs TRUE if its input has a value associated with it.</p>                                                                                                                                                                                                                                                               |
| TO         | <p>Begins procedure definition. Takes a variable number of inputs. Enters edit mode with the procedure named by the first input. Any following inputs are taken as inputs to the</p>                                                                                                                                           |

- procedure named by the first input. With no inputs at all, TO enters edit mode with an empty edit buffer.
- TOPLEVEL** Aborts the current procedure and all calling procedures and returns control to immediate mode. Note the difference between TOPLEVEL and STOP. STOP stops just the current procedure and continues execution with the calling procedure, whereas TOPLEVEL aborts execution of the whole program. It is not used very often in Logo programming.
- TOWARDS** Takes two numbers as inputs. These are interpreted as the x- and y-coordinates of the point on the screen. TOWARDS outputs the heading from the turtle to the point. That is, SETHEADING TOWARDS :X :Y will make the turtle face towards point x,y. Compare with ATAN.
- TRACE** Takes no input. Causes Logo to pause before executing each procedure, and print the name of the procedure and its inputs. Typing any character other than <CTRL> G or <CTRL> Z will cause Logo to go on to the next line. Typing <CTRL> G will cause Logo to abort to toplevel. <CTRL> Z will pause, and CO will continue execution.

- TURTLESTATE** Takes no inputs. Outputs a list of four items giving information about the state of the turtle. The format of the list is as follows: The first element is TRUE or FALSE for pen down or pen up, then TRUE or FALSE for show or hide turtle, then background color, then pen color. Abbreviated: TS.
- WORD** Variable number of inputs (default is 2). Outputs a word that is the concatenation of the characters of its inputs (which must be words). If there are more than two inputs, there must be an opening parenthesis before WORD, and a space and closing parenthesis after the last input.
- WORD?** Outputs TRUE if its input is a word. Since numbers are treated as words, the result will also be TRUE for a number. See also LIST? and NUMBER?.
- WRAP** Places the graphics system in wrapping mode. Any time the turtle moves off the edge of the screen, it reappears at the opposite edge. Wrap mode is the default, and is exited by the NOWRAP command.
- XCOR** Outputs the turtle's x-coordinate as a decimal number.
- YCOR** Outputs the turtle's y-coordinate as a decimal number.

# INDEX

---

' W-57  
" G-30, C-6, W-33  
: G-50, C-6, W-61  
+ G-5, C-1, L-4  
- C-1, L-4  
\* G-5, C-1, L-4  
/ G-5, C-1, L-4  
() C-2  
<> B-8  
> G-67, L-5  
< G-67, L-5  
= G-66 to G-67, L-5  
? B-7, G-87  
[] B-9, G-26  
; P-8 to P-9, A-79, L-5  
! T-7

.ASPECT, P-2, T-45, L-6  
.BPT, L-7  
.CALL, T-20, T-30, L-8  
.CONTENTS, L-9  
.DEPOSIT, U-44, T-19, T-36,  
L-10  
.EXAMINE, U-44, T-19, T-30,  
T-36, L-13  
.GCOLL, T-49, L-14  
.NODES, L-17

## A

Abbreviations, G-2, G-24  
ABS, C-24 to C-25  
Absolute value, C-24 to C-25  
Addition, G-5, C-1  
ADDRESSES, U-43, T-36, T-42  
to T-43  
Addresses, useful, U-44 to U-46

ALLOF, W-70, W-112, L-5  
AMODES, U-43, T-23  
ANIMAL program, U-23  
ANIMAL.INSPECTOR  
program, U-24  
ANYOF, W-70, W-112, L-5  
Apple IIc, P-5, P-8  
Apple IIGS, P-1, P-5, P-9  
APPLEKEY example, T-22 to  
T-24  
Apple Logo, A-80  
Arcs, G-57 to G-58, U-28 to  
U-29, A-75 to A-78  
Arithmetic, G-5, G-47 to G-48,  
C-1  
Arrow keys, B-9, G-16, U-10,  
U-16, T-6, T-8, A-14  
ASCII, W-59, U-45, T-46,  
A-91, L-6  
ASCII values, selected table of,  
T-47  
Aspect ratio, P-2, T-45  
Assembler, T-18, T-23, T-25,  
T-26  
Assembler/Logo interfacing,  
U-43, T-18 to T-41  
ATAN, L-6

## B

BACK, G-2, L-6  
BACKGROUND, G-8, G-10,  
T-12, T-13, L-7  
Beep, T-41  
BEFORE, U-30  
BEFORE?, U-30  
BELL, M-8, T-41

BF, W-43, M-12, L-7  
BG, G-8, G-10, T-13, L-7  
Binary files, U-3  
Binary tree, G-73, A-61 to A-63  
BIN.TO.TEXT, U-36  
BK, G-2, L-6  
BL, W-43, L-8  
Black-and-white pictures,  
    printing, P-5 to P-7  
Blank disk, preparing for use,  
    B-11 to B-14  
Brackets <>, B-8  
Brackets [ ], B-9, G-26  
Bugs, G-29, A-1  
BUTFIRST, W-43, W-92,  
    M-12, L-7  
BUTLAST, W-43, W-92, L-8

C

<CAPS LOCK> key, B-11  
CATALOG, G-29, G-31 to  
    G-32, P-1, U-2, T-15, L-8  
Changing inputs, G-62  
CHAR, W-59, A-91, L-8  
Characters, interrupt, U-45,  
    T-38, T-46  
Circles, G-57 to G-58, A-75 to  
    A-78  
Clearing the workspace, G-32 to  
    G-34  
CLEARINPUT, L-8  
CLEARSCREEN, G-26, G-27,  
    L-8  
CLEARTEXT, W-32, L-9  
CLICK, T-33 to T-35  
CLOSE, U-41, U-42

<Closed-Apple> key, L-19  
CO, G-81, L-9  
Color, G-9 to G-10, G-75 to  
    G-78, T-11, T-13  
Color pictures, printing, P-7 to  
    P-8  
Commands, editing, G-40 to  
    G-41, T-6, T-8 to T-10,  
    A-14 to A-17  
Comments, W-21, A-79  
Computation, B-5, C-1  
Conditional, G-66 to G-69  
CONTINUE, G-81, L-9  
Control commands, See  
    <CTRL>  
Control Panel (IIGS), P-1, P-5,  
    P-10  
Converting files from DOS 3.3  
    to ProDOS, B-12  
Coordinates, graphics, G-85 to  
    G-86  
COPYA, B-15 to B-16  
Copying disks, B-14 to B-16  
Copying files, U-3  
Copying procedures, G-46  
COS, C-4, C-6, C-19, C-22, L-9  
Cosine, C-19, C-22  
COUNT, A-94, L-9  
CS, G-26, G-27, L-8  
<CTRL> key, B-8, B-10  
<CTRL> A, G-40 to G-41, T-6,  
    T-9, A-14  
<CTRL> B, T-8, T-9, A-15  
<CTRL> C, B-8, G-14, G-18,  
    U-16, U-39, T-2, T-8, T-9,  
    A-17

- <CTRL> D, G-40 to G-41, T-6, T-9, A-16
  - <CTRL> E, G-40 to G-41, T-6, T-9, A-16
  - <CTRL> F, G-7, T-3, T-8, T-9, T-38, A-15
  - <CTRL> G, B-8, B-10, G-14, G-19, G-62, U-16, U-39, T-2, T-4, T-8, T-9, T-38, T-42, T-47, A-17
  - <CTRL> L, T-8, T-9, A-15
  - <CTRL> N, G-40 to G-41, U-16, T-10, A-14
  - <CTRL> O, G-40 to G-41, U-16, T-10, A-15
  - <CTRL> <Open-Apple> <RESET>, B-16
  - <CTRL> P, G-5, G-40 to G-41, T-7, T-10, A-14
  - <CTRL> S, G-7, U-12, T-3, T-4, T-38
  - <CTRL> SHIFT-M, T-5, T-11, T-38, T-47
  - <CTRL> SHIFT-P, T-5
  - <CTRL> T, G-7, U-12, T-4, T-5, T-38
  - <CTRL> W, G-43, T-5, T-38
  - <CTRL> X, G-41, T-6, T-10, A-16
  - <CTRL> Y, B-8, G-41, T-10, A-16
  - <CTRL> Z, G-81, T-5, T-38, T-47
  - Curriculum materials, B-3
  - Cursor, B-7, G-18, U-30 to U-31, U-45, T-38, A-14
  - CURSOR, L-10
  - CURSOR.H, U-31
  - CURSOR.HV, U-31
  - CURSORPOS, U-31, U-46
  - CURSOR.V, U-31
  - Curves, G-57 to G-58
  - Cut (see also <CTRL> X), T-10
- D
- Data disk, B-5 to B-6
  - Debugging, G-20, G-79 to G-81, G-83, A-1 to A-13
  - DEFINE, U-10, L-10
  - <DELETE> key, B-9, G-17 to G-17, U-16, T-6, T-9
  - Demonstration programs, U-23 to U-28
  - DIP switch settings, P-3 to P-4
  - Disk, backup copy, B-14 to B-16
  - Disk drives, switching, L-24
  - Disk preparation, B-6
  - Disk, Utilities, see Utilities Disk
  - Division, G-5, C-1
  - DOS, L-11
  - DOS 3.3, B-12, B-14, P-3, U-37, T-14, A-6
  - DOS 3.3 System Master Disk, B-15
  - Dots, G-50, C-6
  - DPRINT, U-41 to U-43
  - DRAW, G-1, G-4, G-7, C-19, C-21, L-11
  - Draw mode, G-1
  - Driving the turtle, G-2
  - DROVE, A-73
  - Duration, M-2
  - DYNATRACK, U-25 to U-26

- E**
- ED, G-34 to G-35, L-11
  - EDIT, G-34 to G-35, T-2, L-11
  - Edit buffer, T-48
  - Editing commands, summary,
    - G-41, T-6, T-8 to T-10,
    - A-14 to A-17
  - Edit mode, G-14 to G-19, T-1
  - Editor, G-40 to G-41
  - Education, B-3
  - Elephant mascot, B-2, A-68
  - Ellipse, C-19, C-25 to C-26
  - ELSE, W-70 to W-71, L-12
  - EMPTY?, W-23, L-12
  - Empty list, W-59 to W-60
  - Empty word, W-59
  - END, G-14, G-22, C-8, L-12
  - Epson printer, P-12
  - ERASE (ER), G-32 to G-34,
    - L-12
  - ERASE ALL, G-32 to G-33,
    - L-12
  - ERASE NAMES, L-12
  - ERASE PROCEDURES, L-12
  - ERASEFILE, G-34, T-16, L-12
  - ERASEPICT, G-35 to G-36,
    - T-16, L-13
  - Erasing, G-11
  - Erasing pictures, G-35 to G-36,
    - T-16
  - ERNAME, L-13
  - Error messages, B-8, W-63, A-1
    - to A-13
  - Errors, typing, B-9
  - <ESC> key, B-9, U-16, T-6, T-9
  - Executing a procedure, G-20
  - EXPONENT, C-15 to C-18
  - Exponentiation, C-15 to C-18
- F**
- FALSE, G-66 to G-69, W-70,
    - L-15
  - FD, G-2, G-3, L-13
  - FID, U-37 to U-38
  - Files, G-29, G-33, M-1
  - FIRST, W-43, W-92, M-12,
    - L-13
  - FLASHING, U-31, U-46, T-38
  - Floating point arithmetic, C-1,
    - T-46
  - Formatting a blank disk, B-6,
    - B-11 to B-14
  - FORWARD, G-2, G-3, L-13
  - FPUT, W-29, W-92, L-13
  - FRERE, U-13
  - Functions, C-4 to C-5
  - FULLSCREEN, G-7, L-14, T-3
  - Fullscreen mode, T-3
- G**
- Garbage collecting, T-49
  - Global variables, C-6 to C-7,
    - W-12 to W-17
  - Glossary of Logo Commands,
    - L-1 to L-28
  - GO, L-14
  - GOODBYE, G-32, G-34, L-14
  - Graphics, B-4, G-1, A-18
  - Graphics commands, summary,
    - G-5
  - Graphics mode, G-1

Graphics screen, T-45  
 Graphics screen, text on, T-14  
 Graphing functions, C-19 to  
 C-26  
 Grappler interface card, P-11 to  
 P-12

## H

HARDCOPY, T-43  
 Harmony, M-14  
 Heading, G-44  
 HEADING, G-83 to G-85, L-14  
 HIDETURTLE, G-37, C-19,  
 C-21, L-14  
 Hierarchy of operations, C-2 to  
 C-3

History lists, W-95 to W-99  
 HOME, G-26, G-27, C-19,  
 C-21, L-14  
 HT, G-37, C-19, C-21, L-14

## I

IF, G-66 to G-69, W-70, L-14  
 IFFALSE (IFF), W-89, W-90,  
 L-15  
 IFTRUE (IFT), W-89, W-90,  
 L-15  
 ImageWriter printer, P-3, P-9,  
 U-3  
 ImageWriter II printer, P-7  
 Immediate mode, B-4, G-15  
 to G-16  
 Initializing a disk, B-6, B-11 to  
 B-14  
 INITSHAPES, U-20 to U-23

Input, G-48, G-76  
 Input, changing, G-62  
 Input, negative, G-81  
 INSPI, U-26  
 INSTANT, B-4, U-9 to U-10,  
 T-47  
 Integer, C-1  
 INTEGER, C-4 to C-5, L-15  
 Integer operators, C-4 to C-5  
 Intelligent language interpreter,  
 W-106 to W-114  
 Interrupt characters, T-38, T-46  
 INVERSE, U-32, U-46  
 Invisible turtle, G-37, U-18  
 I/O (Input/Output), T-24  
 ITEM, W-48, L-15

## K

Keyboard, B-9  
 Keys, editing, G-40 to G-41,  
 A-14 to A-17, T-6, T-8 to  
 T-10  
 Keys, special, B-9 to B-11

## L

Language card, T-20  
 Language Disk, B-1  
 LARC, U-28 to U-29  
 LARGECOLOR, P-7 to P-8  
 LARGEPICT, P-5 to P-7  
 LAST, W-43, W-92, L-15  
 LCIRCLE, U-28 to U-29  
 LEFT, G-2, G-4, L-16  
 Levels of execution, W-63  
 Line editor, T-5 to T-6

Line length, T-6, T-7, T-48  
LIST, W-52, W-90, L-16  
LIST?, W-54, W-76, W-89,  
L-16  
Listing a procedure, G-43  
Listing: Summary of commands,  
G-44  
List processing, B-5  
Lists, W-59  
Load, see READ  
LOCAL, C-7 to C-13, L-16  
Local variables, C-6 to C-13  
Logo for the Apple II, G-73,  
A-63  
Logo for the Apple, Terrapin,  
see *Terrapin Logo for the  
Apple*  
*Logo PLUS*, B-1, B-6, B-11,  
B-12 to B-14, P-3, P-4, U-2,  
U-14, U-30, U-36, U-46,  
T-6, T-7, T-14, T-30, T-48,  
A-1, A-4 to A-10, A-13,  
L-1 to L-4, L-7  
Logo, starting, B-6, B-16  
Lower case letters, B-11  
LPUT, W-29, W-92, W-95,  
L-16  
LT, G-2, G-4, L-16

## M

Mad-lib, W-100 to W-105  
Magic number, G-46  
Major key, M-5  
MAKE, C-7 to C-13, W-9, L-17  
MAKE.OCTAVE, T-32  
MAKE.PITCH, T-32

MAKESHAPE, U-15  
Mascots, B-2, A-68 to A-71  
MEMBER?, W-23, W-73, L-17  
Memory, addresses of  
interesting locations, U-44  
to U-46  
Messages, error, A-1 to A-13  
Mode, DRAW, G-1  
Mode, IMMEDIATE, B-4, G-15  
to G-16  
Mode, NODRAW, G-7, G-8,  
T-1  
Multiplication, G-5, C-1  
Music, B-5, M-1 to M-14  
MUSIC, U-3, U-13 to U-14,  
T-36  
MUSIC.BIN, U-13, T-36  
MUSIC.SRC, U-13  
Music, assembly language  
extension, T-30 to T-36  
Music notation, M-4  
Music procedures (Utilities  
Disk), M-1 to M-14, U-5

## N

Naming, G-14, G-30, G-50,  
G-52, C-6  
ND, G-7, G-8, L-17  
Negative inputs, G-81  
Newsroom program  
(Springboard), P-3  
NODRAW, G-7, G-8, T-3, T-4,  
L-17  
Nodraw mode, T-1  
NORMAL, U-32, U-46  
NOT, W-70, A-82, A-88, L-17

- NOTE, U-13  
NOTRACE, G-69, G-83, A-78,  
L-17  
NOWRAP, G-62, G-64, L-17  
NUMBER?, W-17, A-82, L-18  
Numbers, in Logo, T-46  
Numeric operations, C-1 to C-3
- O
- Object, W-33 to W-36, W-86  
<Open-Apple> key, L-19  
OP, C-13 to C-18, W-37, W-43,  
W-65, L-18  
OPCODES, T-23, T-28  
OPEN, U-41 to U-42  
OPEN.FOR.APPEND, U-41 to  
U-42  
Operations, C-1 to C-3  
Operators, C-1, C-4 to C-5  
<Option> key, L-19  
OUTDEV, P-1, T-11, T-24,  
L-18  
Output, G-76, C-4 to C-6  
OUTPUT, C-13 to C-18, W-37,  
W-43, W-65, L-18  
Overview of Logo, B-3 to B-5
- P
- PADDLE, T-11, L-18  
PADDLEBUTTON, T-11, L-19  
Parabola, C-19, C-23 to C-25  
Parallel printers, P-11 to P-12  
Parentheses, W-55, W-69, W-87  
to W-88  
Paste (see also <CTRL> Y),  
T-10  
PAUSE, G-81, L-19  
Pausing, caused by garbage  
collection, T-49  
PC, G-8, G-12, L-19  
PC 6, G-11 to G-12, G-64,  
G-74, U-17, U-18, U-19,  
T-12  
PD, G-26, G-28, L-19  
PENCOLOR, G-8 to G-12, T-12  
to T-13, L-19  
PENDOWN, G-26 to G-28,  
L-19  
PENUP, G-26, G-28, L-19  
PICK, U-32 to U-33  
Pictures, printing, P-2 to P-14  
Pictures, saving on disk, G-35 to  
G-36, P-2  
Pitch, M-5  
Planning a procedure, G-21 to  
G-23, G-49 to G-50  
PLAY, M-5, T-31  
PLAY.NOTE, M-13, T-31,  
T-36  
Plotter files, P-12 to P-14  
PLURAL project, W-79  
PO, G-29, G-31 to G-32, G-43,  
L-20  
PO ALL, G-43, P-2, L-20  
Pointed brackets, B-8  
PO NAMES, P-2, L-20  
POLY, G-53 to G-57  
POTS, G-29, G-31 to G-32, P-2,  
L-20  
PPRINT, U-33 to U-34

- Predicates, W-72 to W-74
- Preparing a blank disk, B-6, B-11 to B-14
- Primitive, G-12 to G-13
- Primitives, glossary of, L-1 to L-28
- PRINT (PR), G-79, W-12, W-55, L-20
- PRINT1, W-55, W-88, L-20
- Printer Port, P-1, P-5, P-9
- Printers, P-1 to P-14
- PRINTFILE, U-40
- Printing pictures, P-2 to P-14
- Printing procedures, P-1
- PRINTOUT, G-43, L-20
- PRINTOUT ALL, G-43, P-2, L-20
- PRINTOUT NAMES, P-2, L-20
- PRINTOUT PROCEDURES, P-2, L-20
- PRINTOUT TITLES, G-29, G-31 to G-32, P-2, L-20
- PRINTPICT, P-5 to P-7
- PRINTSCREEN, P-4
- Print Shop, The (Broderbund), P-3
- PRINTTEXT, U-40
- Procedural language, B-3
- Procedure, B-3, G-12 to G-13
- Procedure copying, G-46
- Procedure naming, G-13 to G-14
- Procedure printing, P-1
- Procedure saving on disk, G-29
- Procedure writing, G-12 to G-18
- Procedures, graphics, A-18 to A-78
- Procedures that take inputs, G-48
- Procedures, words and lists, A-81 to A-112
- ProDOS, B-12 to B-14, P-3, U-37, T-14, A-6
- ProDOS User's Disk, B-12, B-15, P-3
- Projects: Changing Inputs, G-65, A-49
- Projects: Curves, G-57, A-44
- Projects: History lists, W-99
- Projects: ITEM, W-51
- Projects: Language understanding, W-115
- Projects: Mad-libs, W-105
- Projects: MAKE, W-17
- Projects: More Shapes, G-48, A-38
- Projects: PLURAL, W-79
- Projects: Predicates, W-74
- Projects: Procedure, G-28, A-21
- Projects: RC, W-8, W-22
- Projects: Recursion, G-73, W-51, A-55
- Projects: REQUEST, W-84
- Projects: Simple recursion, G-62, A-47
- Projects: Sizable shapes, G-53, A-39
- Projects: Testing and stopping, G-69, A-53
- Projects: Turtle driving, G-8, A-18

- Projects using RANDOM, G-78, A-65  
Projects using shapes, G-42, A-24  
Projects with regular polygons, G-55, A-41  
Prompt, B-7  
PU, G-26, G-28, L-19
- Q
- Quiz programs, W-81 to W-85  
QUOTIENT, C-4, C-5, L-21
- R
- Rabbit mascot, B-2, A-69  
RARC, U-28 to U-29  
RANDOM, G-76, C-4, L-21  
RANDOMIZE, C-4, L-21  
Random numbers, G-76, U-32 to U-33, L-21  
RC, W-8, W-18, M-8, T-46, A-81, L-22  
RC?, W-18, W-72, L-21  
RCIRCLE, U-28 to U-29  
READ, G-32 to G-33, T-15, T-37, L-22  
READCHARACTER, W-8, W-18, T-46, L-22  
Reading pictures, G-35 to G-36, T-16, T-16, L-22  
Reading procedures, G-32 to G-33, T-15, T-37, L-22  
READPICT, G-35 to G-36, T-16, L-22  
READTEXT, U-39
- Real numbers, C-1  
Recalling lines, G-5, M-4  
Recovery process, B-8  
Recursion, G-61 to G-73, C-15 to C-18, W-48, W-100 to W-105, M-12  
Recursion projects, G-62, G-73  
Recursive designs, G-73, U-27  
REMAINDER, C-4, C-5, L-22  
Remarks, P-8 to P-9, A-79, L-5  
REPEAT, G-26, L-22  
Repeating with <CTRL> P, G-5, M-4  
REQUEST, W-26, W-81 to W-84, L-22  
<RESET> key, B-8, B-10  
Restarting Logo, B-16  
Rests (in music), M-6, T-31  
RESULT:, W-60  
<RETURN> key, B-7, G-17, T-2, T-6, A-15  
RIGHT, G-2, G-3, L-22  
ROCKET, U-22 to U-23  
ROUND, C-4, L-23  
RQ, W-26, W-81 to W-84, L-22  
RSPLOTTER, P-14  
RT, G-2, G-3, L-22  
RUN, W-94 to W-99, U-10, A-72 to A-74, L-23  
Running a procedure, G-19 to G-21

- S
- SAVE, G-29 to G-31, T-15,  
T-37, L-23
- SAVEPICT, G-35 to G-36,  
T-16, L-23
- SAVETEXT, U-39 to U-40
- Saving assembled routines, T-29
- Saving pictures, G-35 to G-36,  
P-2, A-72
- Saving procedures, G-29 to  
G-32, T-16
- Saving text, U-38 to U-40
- SAVMOD, T-37, T-42
- Scales, M-5
- Screen, G-7
- SCREENDUMP, P-11 to P-12
- Screen editor, T-7
- SCRIBE printer, P-10
- Self-starting files, T-44, A-80
- SENTENCE (SE), W-25, W-43,  
W-86, M-13, L-24
- SETDISK, A-79, L-24
- SETHEADING (SETH), G-83  
to G-85, L-24
- SETSHAPE, U-17
- Setup, G-28
- SETX, G-85 to G-86, L-24
- SETXY, G-85 to G-86, C-19 to  
C-26, L-24
- SETY, G-85 to G-86, L-24
- SHAPE.EDIT, U-14 to U-22
- Shapes, creating, U-15 to U-17
- Shapes, reading, U-20 to U-21
- Shapes, saving, U-19 to U-21
- Shapes, stamping, U-18 to U-19
- Shape table, U-14, T-17
- <SHIFT> key, B-8
- <SHIFT> M, B-9
- <SHIFT> N, B-9
- SHOWFILE, U-40
- SHOWTEXT, U-40, L-25
- SHOWTURTLE, G-37, L-25
- SIN, C-19, C-20 to C-22, L-25
- Sine, C-19, C-20 to C-22
- SING, M-6
- Single quote, W-57
- SIZE, U-16 to U-17, U-23
- SMALLCOLOR, P-7 to P-8
- Snail mascot, B-2, A-70
- Spaces in Logo lines, G-3, G-6,  
G-18
- Special effects, G-74 to G-78
- SPLITSCREEN, G-7, U-12,  
T-3, L-25
- Splitscreen mode, T-3
- Square, G-21 to G-25
- SQRT, C-4 to C-5, L-25
- ST, G-37, L-25
- Starting Logo, B-6
- Starting Logo summary, B-16
- STARTUP variable, T-44, A-80
- State, G-44
- STOP, G-66 to G-69, W-6,  
M-33, L-25
- STOPPED!, G-19
- Stopping a procedure, see  
<CTRL> G
- Storage in Logo, T-48
- STRING, U-34 to U-35
- Structured programming, B-3
- Subprocedures, G-58 to G-61
- Subtraction, G-5, C-1

- Summary: commands with keyboard versions, G-25  
 Summary: editing commands, G-41  
 Summary: listing commands, G-44  
 Summary: Logo commands used so far, G-38 to G-39  
 Summary: starting Logo, B-16  
 Summary: turtle commands, G-5  
 Summary: words and lists primitives, W-52  
 Superprocedure, G-58  
 Super Serial Card, P-4, P-9  
 SWEET-P plotter, P-12 to P-13  
 Syncopation, M-10
- T**
- Tangent, C-19, C-22 to C-23  
 TEACH, U-11 to U-13  
 Templates, W-111 to W-116  
*Terrapin Logo for the Apple*, B-1, B-7, B-11, B-12 to B-14, P-3, P-6, U-14, U-30, U-36, T-6, T-7, T-14, T-30, T-48, A-6, A-13, L-7  
 TEST, W-89, L-25  
 Testing: IF-THEN-ELSE, G-66 to G-69, W-70  
 TET, U-27 to U-28  
 TEXT, L-26  
 TEXTEDIT, U-38 to U-40, T-8, T-42, T-49  
 Text editor, using Logo as, T-42 to T-43  
 Text on the graphics screen, T-14  
 TEXTSCREEN, G-7, U-12, T-4, L-26  
 THEN, G-66 to G-69, W-5, W-71, L-26  
 THING, W-78, M-13, L-26  
 THING?, L-26  
 TMOVE, U-10 to U-11  
 TO, G-14, C-8, T-3, L-26  
 TONE, M-2  
 TONES, M-3  
 Top Level, M-12  
 TOPLEVEL, W-4, W-6, T-40, L-27  
 Total Turtle Trip Theorem, G-47  
 TOWARDS, G-83, G-85, L-27  
 TRACE, G-69, G-83, A-78, L-27  
 Tree, G-73, A-61  
 TRUE, G-66 to G-69, W-70, L-15  
 TS, G-83 to G-84, L-28  
 Tune blocks, M-8  
 Turtle, G-2  
 Turtle commands, G-5  
 Turtle driving projects, G-8, A-18  
 Turtle, floor, G-28  
 TURTLESTATE, G-83 to G-84, A-88, L-28  
 TWINKLE, M-9, U-13  
 Typing errors, correcting, B-9 to B-10

- U
- Upper case letters, B-11
  - Utilities Disk, B-1, B-2, B-4, B-12 to B-13, M-1, U-1 to U-46
  - Utilities Disk files: explanation, U-9 to U-46
  - Utilities Disk files: summary, U-4 to U-8
  - Utilities Disk, Logo PLUS, U-1 to U-2
  - Utilities Disk music procedures, M-11 to M-14, U-13 to U-14
  - Utilities Disk: use, U-1 to U-3
  - Utilities, writing your own, G-74 to G-76
- V
- Value, W-38
  - Variables, G-48 to G-52, C-6 to C-7, W-12 to W-17, W-60
  - Variables, global, C-6 to C-7
  - Variables, local, C-6 to C-7
- W
- WAIT, U-35
  - Wild card, W-109
  - WORD, W-43, W-46, M-13, L-28
  - WORD?, W-54, W-72, W-89, L-28
  - Word processor, using Logo as, U-38 to U-40
  - Words, W-56
  - Words and lists, B-5
  - Word wrap, W-33
  - Workspace, G-29, G-32, M-1
  - Workspace, clearing, G-32 to G-34
  - WRAP, G-62 to G-64, L-28
  - Writing a procedure, G-12 to G-18, C-8 to C-9
- X
- XCOR, G-85 to G-86, L-28
- Y
- YCOR, G-85 to G-86, L-28
- Z
- Zero vs. letter O, G-18



Terrapin, Inc.  
376 Washington Street  
Malden, MA 02148  
(617) 322-4800