LOGO
A Language for Learning

commodore
COMPUTER

The Commodore 64 Logo

TUTORIAL

Written by
Virginia Carter Grammer,
E. Paul Goldenberg,
and Leigh Klotz, Jr.

Edited by
Mark Eckenwiler

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

Marlene Kliman
Nola Sheffer
Patrick G. Sobalvarro
Deborah G. Tatar

and especially
J. Sheridan McClees
Peter von Mertens

Designed by Donna Albano and Janet Mumford

Terrapin Logo mascots designed by Virginia Grammer

Logo was implemented on the Commodore 64 by Leigh Klotz, Jr., with assistance from Andy Finkel of Commodore, John Cox, Patrick Sobalvarro, and Devon McCullough of Terrapin. Logo was originally developed at Bolt, Beranek and Newman, Inc. and at the Massachusetts Institute of Technology. The M.I.T. 6502 implementation was accomplished by Stephen Hain, Patrick Sobalvarro, and Leigh Klotz, Jr. under the direction of Professor Harold Abelson.

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

DISCLAIMER

COMMODORE ELECTRONICS LTD. ("COMMODORE") MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THE PROGRAM DESCRIBED HEREIN, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. THIS PROGRAM IS SOLD "AS IS". THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE IS WITH THE BUYER. SHOULD THE PROGRAM PROVE DEFECTIVE FOLLOWING ITS PURCHASE, THE BUYER (AND NOT THE CREATOR OF THE PROGRAM, COMMODORE, THEIR DISTRIBUTORS OR THEIR RETAILERS) ASSUMES THE ENTIRE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION AND ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES. IN NO EVENT WILL COMMODORE BE LIABLE FOR DIRECT, INDIRECT, INCIDENTAL OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE PROGRAM EVEN IF IT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME LAWS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF IMPLIED WARRANTIES OR LIABILITIES FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY.

COPYRIGHT AND TRADEMARK NOTICES

The Logo software and sections of the Reference Guide are copyrighted by the Massachusetts Institute of Technology. The Tutorial and changes to the 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 included in the Commodore Logo 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 publications or publication in other than the English language, apply to the Copyright owner or publisher. Publication pursuant to any permission shall contain an acknowledgment of this copyright and an acknowledgment 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 the 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 © Terrapin, Inc., 1982, 1983
380 Green Street
Cambridge, MA 02139
(617) 492-8816

Terrapin, the Terrapin logo, and the Terrapin mascots are trademarks of Terrapin, Inc.

All references to "Logo" herein refer to the Logo language, developed at the Massachusetts Institute of Technology.

We sincerely hope you enjoy using Commodore Logo. Here are several other Commodore software packages which you should know about:

PILOT for the Commodore 64
This is the most powerful version of PILOT available. You can define your own characters, create colorful, moveable objects called sprites, and create music and a variety of sound effects. With PILOT, educators can write their own courseware to combine fun and entertainment with learning. A RUN-ONLY version of PILOT is also included.

PET EMULATOR
With this program, you will be able to use software that had been designed for the Commodore PET (2.0 BASIC version).

EASYLESSON and EASYQUIZ
These two programs let you easily create lessons and quizzes on topics of your own choice. When you are establishing a pool of questions, you can classify each question in one of seven categories that you define, such as Historical Dates, Famous Quotes, etc. Each of these questions and answers can be "shuffled" before a quiz is given. The quiz can be given on the screen in a "flash card" method or on a piece of paper as a multiple choice or fill in the blank format.

PUBLIC DOMAIN SOFTWARE
The Public Domain software series consists of 27 pre-packaged sets of programs. Altogether, there is a total of 340 educational programs. Subjects such as Business, Computer Science, English, History, Math, Geography, and Technology are included for various grade levels. Included also are challenging games in logic, math, Hi-Q, and words.

INTRODUCTION TO BASIC: PART I
Intro to BASIC: Part I teaches you how to program in the BASIC language on your Commodore 64. The manual gives you step by step instructions so that learning is fun and easy.

GORTEK and the MICROCHIPS
This package teaches young people how to program in BASIC. The program theme is to have you help Gortek teach the Microchips to program the computer in time to ward off an attack from the powerful Zitrons.

SPEED/BINGO MATH (Elementary Grades)
This package features two learning games. Addition, subtraction, multiplication, and division can be learned while having fun. SPEED MATH gives you a time limit to solve a variety of math problems. BINGO MATH asks you to solve a math problem and then use the answer to play bingo. The games can be played against the clock or opponents.

THE WORD MACHINE and THE NAME MACHINE
This is an easy-to-learn and easy-to-use word processing package. Perfect for letters, address lists, memos, and notes, these programs let you overtype, insert, and delete text; personalize form letters; and print in draft, formal, or informal formats.

EASY SCRIPT 64
This is a powerful word processor with table producing capabilities, comprehensive printer controls, easy update facilities, easy document handling, the ability to interact with EASY SPELL 64, and more.

EASY SPELL 64
Easy Spell 64 features the following: the automatic correction of spelling errors, the ability to count the number of words in your manuscript and interact with Easy Script 64, and a built-in 20,000 word dictionary that lets you add words not already stored there.

EASY MAIL 64
With Easy mail 64, you can easily manage your address files. Label printing is also simplified with Easy Mail's ability to search for specific fields/categories. The program's features include entry, change, or deletion of a file by name or number; the capability to print one or two abreast labels; a HELP screen; and the ability to print a complete printout of all the data in each of your records.

DISK BONUS PACK
CASSETTE BONUS PACK
As an introduction to your Commodore 64, both of these packages feature programming aids, music and video demonstration programs, and several educational and personal programs.

SUPER EXPANDER 64
This cartridge is a powerful extension of the BASIC language which gives you the commands needed to easily access and implement Commodore's graphics, music, and sound capabilities. You will be amazed at how quickly and easily you can plot points and lines; draw arcs, circles, ellipses, rectangles, triangles, octagons; paint shapes with specified colors; read game paddle and joystick locations; create music and sound; display text; split screens to display both text and graphics; and program the function keys.

EASY CALC 64
Easy Calc 64 is an easy-to-use electronic spread sheet which features editing functions and HELP screens. With Easy Calc 64, you can also print bar charts and individually formatted tables.

THE MANAGER
The Manager is a general data base for handling your files.

THE MUSIC MACHINE
This cartridge turns your Commodore 64 keyboard into a music synthesizer. You can now utilize all of your computer's music-making capabilities, even if you don't know how to program or play music.

MUSIC COMPOSER
This package teaches everything you need to play songs and create sound effects on the computer. The manual also includes several sample songs for you to play.

THE COMMODORE 64 MACRO ASSEMBLER DEVELOPMENT SYSTEM
This package is designed for experienced Assembly language programmers. Everything you need to create, assemble, load, and execute 6500 series Assembly language code is included.

SCREEN EDITOR
The Screen Editor helps you design software by letting you create and edit your own screens. This programming tool is for users with some computer experience.

Commodore is proud to announce an entire series of EASY FINANCE software packages. The EASY FINANCE series is called "easy" because all of the programs are simple to operate and require no programming experience. Here is a brief description of each:

EASY FINANCE I — LOANS
LOANS shows you how to make the most out of your hard-earned money by calculating 12 different loan concepts for you. Principal, regular payment, last payment, and remaining balance are just some of the functions EASY FINANCE I can determine.

EASY FINANCE II — INVESTMENTS
INVESTMENTS helps you make the right financial decisions by showing you how to make the most out of 16 investment concepts. Functions such as future investment value, initial investment, and internal rate of return can be calculated.

EASY FINANCE III — ADVANCED INVESTMENTS
ADVANCED INVESTMENTS is an advanced version of EASY FINANCE II. It shows you how to make the most out of 16 more investment concepts. Financial terms are clarified and functions such as discount commercial paper, financial management rate of return, and financial leverage and earnings per share are included.

EASY FINANCE IV — BUSINESS MANAGEMENT
This is a business management package that shows managers how to make the right decisions about production, inventory, control, compensation, and much more. Lease purchase analysis, depreciation switch, and optimal order quantity are some of the 21 functions this program can calculate for you.

EASY FINANCE V — STATISTICS
STATISTICS shows you how to make the most out of statistics. This includes payoff matrix analysis, regression analysis forecasting, and apportionment by ratios.

Please contact your local Commodore dealer for additional information on other software available for your Commodore computer.

Thank you for owning a Commodore computer. Now that you are a member of the Commodore family, maybe you'd like to expand your computer's family. Here is a list of additional hardware which is compatible with your Commodore computer:

1525 Printer
This printer is an 80 column, dot-matrix, impact printer for creating printouts and hard-copies from your VIC 20 or Commodore 64. The printer features 30 characters per second print speed and prints graphics and text characters.

1526 Printer
This bi-directional, 80 column, dot-matrix, impact printer is excellent for creating printouts and hardcopies from your computer. The printer features programmable line spacing and a print format interpreter.

1520 Plotter/Printer
This is a four color, high resolution plotter that connects directly to your VIC 20 or Commodore 64 computer. With the 1520 Plotter/Printer you can plot on paper the same unique color graphics that you can create on your screen using Logo graphics!

Commodore Piano Keyboard and DIGI-DRUM®
These exciting accessories for your Commodore 64 add musical flair to your talents. Both of these computerized instruments are excellent for composing, playing or learning music. The Piano Keyboard comes complete with the software needed to turn your computer into a music synthesizer. DIGI-DRUM is an inexpnsive 3-pad electronic drum which creates three types of authentic percussion sounds. Both the Piano Keyboard and DIGI-DRUM may be used with a television set, monitor, or your own audio system.

Commodore Speech Module
The speech module cartridge comes with a built-in vocabulary of 234 words which are easily programmed into sentences. The module "talks" in a pleasant female or male voice. It can generate other types of voices with special vocabularies geared to each software package. The speech module works with disk, tape, and also has a slot for accepting plug-in cartridges.

1701 Monitor
This full color monitor is compatible with the VIC 20, Commodore 64, and other computers. The 1701 Monitor features high quality and resolution video and a built-in speaker with audio amplifier.

1530 Datasette
The 1530 Datasette is a low cost, highly reliable way to store and retrieve programs and data. It features keys for Play, Record, Fast-Forward, Rewind, and Stop. The 1530 Datasette uses standard audio cassette tapes and allows naming of programs and files, verification of programs, and programmable end of tape marker sensing.

Joystick and Paddles
Controls for games and entertainment.

1600 Modem
This telephone interface lets you communicate with other computer systems over your telephone line! The modem package includes cassette-tape terminal software, a free password and one-hour subscription to the CompuServe System® and software controls for duplex, baud rate, and parity. There is also an optional adapter for non-modular phones.

1650 Automatic Modem
This telephone interface features automatic answer and automatic dial. The modem package lets you communicate with other computer systems over the phone lines! It includes cassette-tape terminal with software, a free password and one-hour subscription to the CompuServe System® and software controls for duplex, baud rate, and parity. You need a modular phone or adaptor to use this product.

PET 64
This unique machine combines the many features of the Commodore 64 with the capabilities of the Commodore PET. However, sprites, color and sound are not featured on this machine. The majority of other Logo commands will be compatible.

SX100/DX1100 Portable Color Computers
These new additions to the Commodore family of computers give you the many features of the Commodore 64 only now in a convenient portable style: The model SX100 (single disk drive) and DX100 (double disk drive) are excellent investments for executive business people as well as affordable for today's students. Your Logo program will be compatible with these computers.

CONTENTS

BEGINNING LOGO

Your Commodore Logo Package B-1
This Tutorial B-1

Overview: What Can You Do With Logo? B-2
Graphics B-3
Words and Lists B-4
Computation B-4

Starting Your Commodore 64 .... B-4
Preparing a Blank Disk For Use B-5
Starting Logo B-6
When Logo Has Started Up B-7
Recovery Process B-8
Upper and Lower Case and Graphics Characters:
The Commodore Key B-10
Starting Logo: Summary B-11

GRAPHICS

Graphics Mode G-1
Driving the Turtle: FORWARD (FD), BACK (BK),
RIGHT (RT), LEFT (LT) G-2
Let Logo Do Your Arithmetic G-4
An Easy Way to Repeat Yourself: <CTRL> P, Up Arrow G-4
The Screen: DRAW, NODRAW (ND), TEXTSCREEN
(Function Key <f1>), SPLITSCREEN (Function
Key <f3>), FULLSCREEN (Function Key <f5>) G-5
Turtle-driving Projects G-6
Color: PENCOLOR (PC), BACKGROUND (BG),
SINGLECOLOR, and DOUBLECOLOR G-7
The Magic of PENERASE (PENCOLOR −1): Erasing G-8

Introduction to Procedure Writing G-9
Primitives vs. Procedures G-9
Naming a Procedure G-10
Writing a Procedure: EDIT Mode: TO, END,
<CTRL> C, RUN Key, <CTRL> G G-11
Running a Procedure G-15

Planning and Drawing Your Favorite Square G-16
Projects: Simple Procedures G-18
What goes Into a Procedure G-19
More Primitives: REPEAT, CLEARSCREEN (CS),
HOME, PENUP (PU), PENDOWN (PD) G-19
Procedure Projects G-21

Saving Procedures: CATALOG, SAVE, POTS G-21
Clearing the Workspace, Reloading Procedures:
READ, GOODBYE, ERASE (ER), ERASE ALL,
ERASEFILE G-23
Saving, Reading and Erasing Pictures: SAVEPICT,
READPICT, ERASEPICT G-25
The Invisible Turtle: HIDETURTLE (HT),
SHOWTURTLE (ST) G-26
Changing the Textscreen Color: TEXTCOLOR, TEXTBG G-26
Putting Letters on the Graphics Screen: STAMPCHAR G-27

Summary of Logo Commands Used So Far G-28

More About the Editor: <CTRL> P, <CTRL> N,
<CTRL> O, <CTRL> A, <CTRL> L, <CTRL> D,
<CTRL> K, Arrow Keys G-30
Summary of Editing Commands G-31
Projects Using Shapes G-31
Listing a Procedure: PRINTOUT (PO), <CTRL> W G-32
Summary of Listing Commands G-33
Heading: A Matter of State G-33
Copying a Procedure G-34
A Magic Number G-34
Projects: More Shapes G-35

Introduction to Variables: Procedures That Take Inputs G-36
Projects: Sizable Shapes G-39
From SQUARE to POLY G-40
Projects: Regular Polygons G-41
Another View of POLY G-41
Circles G-42
Projects: Curves G-43
Using Subprocedures G-43

Non-stop Procedures: Introduction to Recursion G-45
Projects: Simple Recursion G-46
Recursion: Changing the Input WRAP, NOWRAP G-46
Projects: Changing Inputs G-48
Stopping With Style: IF-THEN, STOP G-48
Projects: Testing and Stopping G-51
Using the Full Power of Recursion G-51
Recursion Projects G-54

Special Effects and New Utilities G-55
RANDOM Numbers, Numbers from Arithmetic
  Operations, Inputs, Outputs G-55
Projects Using Random G-57
Debugging by Printing Values: PRINT (PR) G-57
Debugging Using PAUSE: <CTRL> Z,
  CONTINUE (CO) G-59
Negative Inputs G-59
More on Debugging: TRACE, NOTRACE G-61
Commenting Your Program: ; G-61
More About the Turtle: DRAWSTATE, HEADING,
  SETHEADING (SETH), TOWARDS G-62
Position When You Want It: XCOR, YCOR,
  SETX, SETY, SETXY G-63

INSTANT: Logo Turtle Graphics for the Non-reader G-64

COMPUTATION: HANDLING NUMBERS

Arithmetic Operations C-1
Hierarchy of Operations C-1
Outputs, Integer Operators, Functions: RANDOM,
  RANDOMIZE, ROUND, INTEGER, QUOTIENT,
  REMAINDER, SQRT, SIN, COS C-3
Variables, Global and Local: MAKE, LOCAL C-5
Procedures: TO, END C-6
Interactive Procedures: LOCAL, REQUEST C-7
Bringing Values Out of Procedures: OUTPUT (OP) C-10
Example of OUTPUT and Recursion:
  A Procedure to Do Exponentiation C-12
Graphing Functions: Sine, Cosine, Tangent,
  Parabola, Ellipse, SETXY, HOME, DRAW, HT C-15

WORDS AND LISTS

INTRODUCTION .... W&L-1
Interactive Graphics: READCHARACTER (RC), TOPLEVEL, STOP .... W&L-3
Projects with RC: Extending QUICKDRAW .... W&L-6
Changing the Value of a Variable: MAKE, PRINT (PR) .... W&L-7
Projects with MAKE: More Extensions to QUICKDRAW .... W&L-12
Interactive Programs without Waiting: RC? .... W&L-13
Projects with RC, RC?: Extensions to LOOP .... W&L-16

INTERACTIVE LANGUAGE .... W&L-17
Don't Skip This Section: MEMBER?, EMPTY? .... W&L-17
Some Friendly Introductions: SENTENCE (SE), REQUEST, LPUT, FPUT .... W&L-18
Interlude: Clearing the Text Screen with CLEARTEXT .... W&L-23
Objects: Producing RESULTs as Output, and Using Them as Input .... W&L-23
Writing Procedures that Create and Output Objects: OUTPUT .... W&L-26
Making One Procedure's Output into Another Procedure's Input: OUTPUT (OP), FIRST, BUTFIRST (BF), LAST, BUTLAST (BL), SENTENCE (SE), WORD .... W&L-31
Subprocedures for Cleaner Programming .... W&L-34
A Generalization Using Recursion: ITEM .... W&L-35
Projects Using ITEM and Recursion .... W&L-38

DEFINITIONS AND MODELS .... W&L-39
Some Important Primitives Used in this Chapter .... W&L-39
Definitions of Words and Lists: CHAR .... W&L-42
Programming: Some Metaphors and Some Review .... W&L-45
Some Details of Programming in Logo: Variables, Passing Objects, Logo's Way of Understanding Commands, and Logo's Messages When It Doesn't Understand .... W&L-46
How Logo Interprets a Command .... W&L-51
Using Logo Predicates and Creating New Ones: LIST?, WORD?, MEMBER?, and the Structure of IF, THEN, and ELSE .... W&L-54

Projects with Predicates .... W&L-57
Ordered Rules .... W&L-57
Projects with PLURAL .... W&L-59

Quiz Programs: More About REQUEST (RQ) .... W&L-61
Projects with REQUEST .... W&L-63
Composing Logo Objects: SENTENCE (SE), WORD, LIST, FPUT, LPUT, TEST, IFTRUE (IFT), and IFFALSE (IFF) .... W&L-65
An Application of LPUT in Interactive Graphics: RUN .... W&L-71
Using the History List: Applying a Command (RUN) to Each Element of a List .... W&L-73
Projects with History Lists .... W&L-75
Substituting One Word for Another in a Sentence: A Procedure with Two Recursive Calls .... W&L-75
Projects with Mad-Libs .... W&L-79

Understanding Language: Searching for Key Words and Matching Sentences to Templates: ALLOF, ANYOF .... W&L-80
Projects with Language Understanding .... W&L-86

SPRITES

Introduction S-1
Getting Ready to Use Sprites S-1
Talking to Sprites: TELL S-1
A Program Using TELL S-2
Which Sprite: WHO S-4
Sprites and the Graphics Screen: TB? S-5
Sprite Shapes S-6
Changing Sprite Size: BIGX, BIGY, SMALLX, SMALLY S-7
Changing Sprite Shapes: SETSHAPE S-8
Identifying the Shape of the Current Sprite: SHAPE S-9
The Sprite Editor: EDSH S-10
Moving Around and Drawing in the Sprite Editor S-11
Summary of Sprite Editor Commands S-13
Saving Shapes S-14
Shape Numbers of the Built-In Shape Files S-14
Demonstration Programs S-15

DINOSAURS S-15
SUBMARINE S-17
RUNNER S-19
Suggestions for Ambitious Programmers S-21

MUSIC

Preparation: READ M-1
Duration M-2
Pitch M-4
Procedures M-5
Envelope M-8
The SOUND Procedure M-9

APPENDIX

ERROR MESSAGES A-1
Part I A-1
Part II A-4

EDIT MODE A-11
Use of Control Characters and Cursor Keys
for Ease in Editing A-11
Moving the Cursor A-11
Moving the Text A-12
Deleting Text A-12
Leaving EDIT Mode A-13

GRAPHICS CHAPTER PROJECTS A-14
Turtle-driving Projects A-14
Procedure Projects A-16
Projects Using Shapes A-19
Projects: More Shapes A-30
Projects: Sizable Shapes A-32
Projects with Regular Polygons A-34
Projects: Curves A-36
Projects: Simple Recursion A-38
Projects: Changing Inputs A-39
Projects: Testing and Stopping A-43
Recursion Projects A-45
Projects Using Random A-53

Mascots: Elephant, Rabbit, Snail A-55
Procedures for Saving Pictures A-58
Developing an Arc Procedure A-60

STRATEGIES FOR THE WORDS AND LISTS PROJECTS A-63

THE COMMODORE LOGO UTILITIES DISK A-88
How to Make a Backup Copy of the Utilities Disk A-88
Summary of Utilities Disk Files A-90
Explanation of Utilities Disk Files A-94
ARCS: Variable Radius Arc and Circle Procedures A-94
B&W: Resets the Color for a Black & White TV A-95
BASE: For Converting from One Base to Another A-95
CCHANGE, CCHANGE.BIN, CCHANGE.SRC:
Changing Colors on the Graphics Screen A-96
COLORS: Variable Names for Colors A-96
FOR: A FOR-NEXT Loop A-97
INSTANT: Single Letter Logo Commands A-97
JOY: For Drawing with the Joystick A-97
LOG: Procedures for Logarithms and Exponents A-97
PLOTTER: Controlling the Commodore 4-Color Plotter A-98
PRINTPICT: Hardcopy Printing of Saved Pictures A-98
STAMPER: Printing Text on the Graphics Screen A-99
STAMPFD: Printing Strings of Characters
on the Graphics Screen A-99
TEACH: How to Write Logo Procedures
Without Using the Editor A-100
TEXTEDIT: How to Read, Examine, and Print Text Files A-101
WHILE: Procedures for WHILE and UNTIL Commands A-104

ADVENTURE: An Adventure Game A-104
ANIMAL: The Game that Teaches
the Computer About Animals A-104
ANIMAL.INSPECTOR: What's in the
ANIMAL Knowledge Base? A-105
DYNATRACK: A Game: the Dynamic Turtle
on a Frictionless Surface A-106
GRAMMAR: A Random Sentence Generator A-106
INSPI: Sample Logo Picture A-107
PIG: A Piglatin Program A-108
SNOW: A Recursive Snowflake Drawing A-108
TET: A Graphics Procedure of Variable Complexity A-109

ADDRESSES, AMODES, ASSEMBLER, OPCODES:
Interfacing Logo and the Assembler A-109
SPRITES, SPRED, ANIMALS, ASSORTED, RUNNER, SHAPES, VEHICLES, DINOSAURS, SUBMARINE, SPRITEDEMOS, VELOCITY A-110
MUSIC, SOUND, TWINKLE A-110

COMMODORE 64 LOGO REFERENCE GUIDE

USE OF THE LOGO SYSTEM A-111
Modes of Using the Screen A-111
Nodraw Mode A-111
Edit Mode A-111
Draw Mode A-112
Editing A-113
Line Editor A-113
Screen Editor A-114
Using Commodore Peripherals A-116
Printing Procedures on a Printer A-116
Color Control A-118
Color Modes A-119
The Logo File System A-119
Disk Files A-119
Saving Pictures A-120

MISCELLANEOUS INFORMATION A-122
Self Starting Files A-122
INST Key: Printing Non-Standard and Reverse Characters A-123
Various System Parameters A-123
Memory Organization Chart A-125

ASSEMBLY LANGUAGE INTERFACES TO LOGO A-126
.EXAMINE and .DEPOSIT A-126
Writing Your Own Machine-Language Routines A-127
The Logo Assembler A-128
Syntax of Input to the Assembler A-129
Saving Assembled Routines on Disk A-131
Example: Changing Colors A-131
Useful Memory Addresses A-134

LOGO SYSTEM PRIMITIVES GLOSSARY

Graphics Commands A-139
Numeric Operations A-143
Word and List Operations A-145
Defining and Editing Procedures A-148
Naming A-149
Conditionals A-150
Control A-151
Input and Output A-152
Filing and Managing Workspace A-154
Debugging A-157
Miscellaneous Commands A-158
The .OPTION Primitive A-159

INDEX

B BEGINNING LOGO

Your Commodore Logo Package

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

In your Commodore Logo package, you will find:

1 Logo Language disk
1 Utilities Disk containing demonstration and utility programs
1 Commodore Logo Tutorial, which you are now reading, including an Appendix with technical information

In order to use Logo, you will also need a Commodore 64 with a VIC-1541 disk drive. To save your work, you will need a blank disk.

Instructions for copying and using the Utilities Disk are in the Appendix. You will need a second blank disk to make a copy of the Utilities Disk.

NOTE: It is possible to run Logo without a disk in the disk drive, but you would not be able to save your work. So we encourage you to prepare a blank disk for storing the procedures you will be writing, as described later in this chapter.

This Tutorial

This tutorial will teach you how to use Logo. The Appendix has several sections. There are explanations of error messages; descriptions of the files on the Utilities disk; documentation on the edit mode of Logo; suggested answers to projects in the Tutorial; and information about the internal workings of Logo. The Glossary contains descriptions of Logo primitives with explanations of what they do. You need not read the Appendix and Glossary to start using Logo, but you will find it useful as a reference tool when you want to continue.

Throughout this tutorial, there are projects. Suggested answers to these projects are in the Appendix. REMEMBER that there are many ways to solve a problem and just because your method is different from ours does not mean it is wrong.

Other books you may wish to read include MINDSTORMS: Children, Computers, and Powerful Ideas, by Seymour Papert; TURTLE GEOMETRY, by Harold Abelson and Andrea diSessa, which is awaiting the day that you think you have done all there is to do in Logo; SPECIAL TECHNOLOGY FOR SPECIAL CHIL-

DREN, by E. Paul Goldenberg, which describes uses in several special needs environments; LEARNING WITH LOGO by Daniel Watt, for more ideas of what to do with Logo, particularly in the classroom . . . a must for teachers and kids; THE TURTLE SOURCEBOOK by Donna Bearden and Jim Muller (Young People's Logo Assoc., Inc.) and Kathleen Martin, Ph.D (U. of Dallas) is filled with examples, worksheets and suggestions on using and enjoying Logo . . . another good book for the classroom.

Also, look forward to the following books specifically for Logo on the Commodore 64: LOGO PRIMER, by Gary G. Bitter and Nancy Watson (Reston Publishing Co.). The Logo Primer provides applications for elementary school students.

Once you are comfortable with your Commodore 64, 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 and 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 also put some information between big bold bands and shifted them to the right of 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 (primitives), which are built into the language, or procedure names, which you define. 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, or having to chain to it, or link it.

You build a system of procedures the way you build your own knowledge base, new procedures and knowledge using and building on what is already in existence. This leads to clearer, more structured programming and thinking, 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 interpretive 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 primitives, and your own 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. Logo procedures can even be written to edit themselves, or other procedures.

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 this opens to you.

Graphics

Logo graphics allows you to 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 co-ordinates, although Logo can tell you the co-ordinates at any position.

Commodore 64 Logo also offers the best available graphics and sprite capabilities, giving you control of up to eight independently-controlled sprites.

Graphics is first in this tutorial because you need no experience to be able to use it. Pre-schoolers, using the single-letter commands in the INSTANT system, can do Logo graphics. At the other end of the intellectual spectrum, Professors Harold Abelson and Andrea diSessa, at M.I.T., use Logo graphics to develop concepts in higher mathematics and physics in their book Turtle Geometry: The Computer as a Medium for Exploring Mathematics.

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 which is impossible to match in non-list-processing languages such as BASIC, FORTRAN, and PASCAL.

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.

Starting Your Commodore 64

The following assumes that the Commodore is already hooked up, and only needs to be turned on. If this is not true, please read the Commodore 64 User's Guide and the VIC 1540 or VIC 1541 User's Manual for directions on connecting the computer, disk drive, and TV together.

Do NOT put a disk in the disk drive before everything is turned on or the disk could be damaged. To start your Commodore 64, (1) turn on the disk drive (the switch is on the right side of the back panel) and also turn on the TV screen or monitor. (2) Turn on the Commodore 64. REMEMBER, only turn on the computer after the connected equipment is turned on.

There should be the Commodore heading at the top of the screen and the word READY, with a blinking cursor below it.

NOTE: It is possible to run Logo without a disk in the disk drive, but you would not be able to save your work. Therefore, we encourage you to prepare a blank disk for storing the procedures you will be writing. If you already know how to format a blank disk, skip to the next section on starting Logo.

Preparing a Blank Disk for Use

A blank disk, unlike an audio cassette tape, must be prepared before it can store information. This process is called formatting the disk.

To format a disk on the Commodore 64:

1. Insert the blank disk you want to format into the disk drive. The disk should be inserted with the label facing up and towards you. (If there is no label, compare with the Logo disks to see where the label should be placed.)

2. Push the disk into the disk drive far enough that it does not pop back out again. Then push down on the disk drive door until it clicks shut. (To open the door, push in and then up.)

3. Type

   ```
   OPEN 15,8,15
   ```

   and press the <RETURN> key.

4. Next, type

   ```
   PRINT#15,"N0:MY DISK,45"
   ```

   and press <RETURN>.

   You can use any phrase in place of MY DISK as the disk name or header (as long as it does not exceed 16 characters) and any two digit number where the 45 is. The disk drive will whir for a little over a minute, then the Commodore prompt READY will appear on the screen and the light will go out on the disk drive.

5. Remove this disk from the disk drive, label it with a felt tip marker immediately, and use it to store your Logo procedures. We will refer to it again in the section When Logo Has Started Up.

More information about inserting and initializing disks is in the Commodore VIC-1541 floppy disk manual (page 8 for insertion of diskette, page 15 for initialization). See the OPEN and NEW commands on the page referred to in your manual for initialization. BASIC programs and Logo procedures can be stored on the same disk.

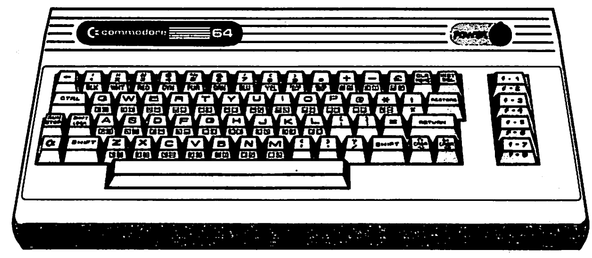
You can initialize disks while running Logo. Type the following command:

```
DOS [N0:MY DISK,23]
```

for the same result as the BASIC sequence above.

Starting Logo

KEYBOARD DIAGRAM



One of the disks packaged with your system is called the Language Disk. It is the disk with the Logo interpreter on it. The other disk, labeled Utilities Disk, contains some demonstration and utility programs. They are mentioned where appropriate in the tutorial and summarized and cross-referenced in the Appendix.

Follow the steps below to start Logo.

(1) The TV's screen should display the word READY with a blinking cursor below it. What else is on the screen depends on what you have done before this point. (2) Place the Language Disk in your disk drive with the label facing up and towards you. (3) Close the disk drive door firmly. If you are not sure how to start the computer and insert a disk, see the Commodore VIC-1541 floppy disk manual, particularly the sections about "powering on" and "insertion of diskette."

Next, type

```
LOAD "LOGO",8
```

and press <RETURN>. The disk drive light will go on and the Commodore will print

```
SEARCHING FOR LOGO
LOADING
READY
```

If this message does not appear, check to be sure that you are using the Language disk and that the disk drive door is firmly closed. Then type,

```
RUN
```

and press <RETURN>. The screen will clear and

```
Loading, please wait ...
```

will appear. It takes a couple of minutes to load and start Logo. When it has started, Logo will print this brief message:

```
  COMMODORE LOGO

 COPYRIGHT (C) 1982,1983 TERRAPIN, INC.
 COPYRIGHT (C) 1981 MIT

 WELCOME TO LOGO!
 ?
```

If Logo does not start up after about two minutes, your language disk may be damaged in some way, or your disk drive may be damaged. Check to see if you are using the correct disk and the drive door is properly closed. If other disks work on your disk drive, the problem is most likely with the diskette.

When Logo Has Started Up

Logo will print its WELCOME 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 Commodore 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 hit <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. (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. You might find a bug while using Logo which 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. This is very very unlikely, so do not be afraid to try things.

RECOVERY PROCESS

The method used to restart Logo without reloading it from the disk is to hold down the <RUN/STOP> key and hit the <RESTORE> key. This should bring the heading COMMODORE LOGO back onto the screen. If this does not happen, you will have to turn off the machine and start again. Sometimes in using the disk drive or the printer, there is a problem. For the disk drive, try typing DOS[I] and hit <RETURN>. If this does not work, you could try turning off the disk drive and turning it back on again. Beware though that there is a slight chance that this could hurt your computer. The same applies for the printer.

When Logo does not understand something 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 do, turn to the list of Error Messages and their explanations (with examples) in the Appendix.

Once in a great while Logo confesses (rightly or wrongly) to a bug and types out explicit messages and recovery instructions. Logo will say something like TYPE R OR C. Type C to Continue with what you were doing before the bug appeared. If you want to erase everything and start over, type R (for Restart). Typing R is like giving Logo the GOODBYE command.

USE THE EDITING KEYS TO CORRECT TYPING ERRORS:

The <DEL> key erases the character to the left of the cursor and moves the cursor and following text one space to the left. The left and right arrow keys move the cursor in the same direction that they point on the keyboard. A left arrow key is in the upper left hand of the keyboard, and a key marked <CRSR> in the lower right hand corner has both a left and a right arrow depending on whether the <SHIFT> key is used. (If you hold down the <DEL> or the <CRSR> arrow keys, they will automatically repeat.) 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 for what is inserted.

Type

MARY HAD A LITTLE LAMB

Use the <DEL> 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 Logo, whatever it is doing.

Upper and Lower Case and Graphics Characters: The Commodore Key

The Commodore 64 has two modes: upper case/graphics and lower/upper case mode. The default setting is for upper case/graphics mode in which upper case letters will normally be displayed when you type. If the <SHIFT> key is used, the graphics symbols on the right front side of the keys are printed instead of upper case letters. For example, if you press <SHIFT> Q, a solid ball is displayed.

To switch to lower case/upper case, press the <COMMODORE> key and the <SHIFT> key. Now, all letters will be lower case except, when you type a letter while holding down the <SHIFT> key, an upper case letter will be displayed.

To get back to the original mode, just hold down the <COMMODORE> key and the <SHIFT> key again. While you are in lower case/upper case mode, holding down the <COMMODORE> key and typing any other key will print the graphic symbol on the left front side of the key.

Note: When in upper case/graphics mode, Logo will recognize commands only in upper case. Also, when in lower/upper case mode, Logo will recognize commands only in lower case. This applies only to the built-in Logo commands, also known as primitives.

Procedure and variable names may use both upper and lower case or upper case and graphics symbols. We recommend that you leave your computer in upper case/graphics mode while using this manual.

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. Use <CTRL> G to stop programs.

STARTING LOGO: SUMMARY

1. Turn on the drive and the Commodore 64.
2. Place Language Disk in disk drive
3. Type LOAD "LOGO",8 and hit <RETURN>.
4. Type RUN and wait approximately 90 seconds while Logo is loaded.
5. After WELCOME TO LOGO is printed, remove the Language Disk and insert your storage disk
6. You are ready to proceed with Logo

G GRAPHICS

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

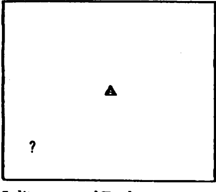
Enter the graphics or DRAW mode by typing DRAW:

```
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 five lines of the screen are reserved for commands you will type and the rest of the screen is drawing space.

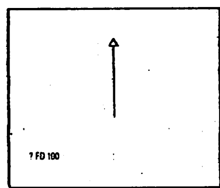


Splitscreen and Turtle

The small triangle in the middle of your screen is called the turtle. When it first appears, it is pointing upward. The back of the turtle has a thicker line than the other two sides so you can tell where it is heading.

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

You move the turtle with turtle commands. The turtle can leave a trail (draw a line) 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

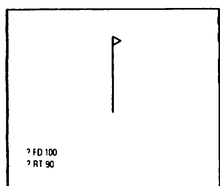
FD 100 <RETURN>

The turtle will move forward one hundred turtle steps. The space between the command and the number is necessary. If it is 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 explained 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. (Please don't worry. We will discuss how to save pictures later on in this manual.)

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) Draw 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. Logo will automatically do the arithmetic for you.

Enter	and Logo calculates
FD 10 * 5	FD 50
RT 100/3	RT 33.3333...
FD 5 + 5	FD 10

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

An Easy Way to Repeat Yourself: <CTRL> P, Up Arrow

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 retrieve the line for you if you press <CTRL> P, or if you press the up arrow key. (Remember, for <CTRL> P, hold down the <CTRL> key and press the <P>.) The up arrow key is easiest to use. Both the up arrow key above the <RETURN> key and the up arrow <CRSR> key below the <RETURN> key can be used. 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.

Try typing

<CTRL> P <SPACE> <CTRL> P <RETURN> or
<Up Arrow> <SPACE> <Up Arrow> <RETURN>

This will print out two sets of your instructions. You can repeat the <CTRL> P or up arrow as many times as you wish, up to 256 characters (a little more than 6 lines), as long as there are spaces between the commands. If you put 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

The last command of the first batch will not be separated from the first command of the second, and Logo will stop and display

THERE IS NO PROCEDURE NAMED 115FD

The Screen: DRAW, NODRAW (ND), TEXTSCREEN (Function Key <f1>), SPLITSCREEN (Function Key <f3>), FULLSCREEN (Function Key <f5>)

When Logo is in DRAW mode, the Commodore 64 displays five lines of text at the bottom of the screen.

> The number of text lines displayed can be changed. See the definition of SPLITSCREEN in the Glossary.

To see the commands you have typed that have disappeared under the picture, type

TEXTSCREEN or function key <f1>

To get back the split graphics/text screen, type

SPLITSCREEN or function key <f3>

To show off your drawing without the distracting text, type

FULLSCREEN or function key <f5>

<f3> will bring back the split screen from either the text- or fullscreen mode.

> To erase any text displayed on the screen, press the <SHIFT><CLR> key. To erase any graphics on the screen, type DRAW. 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

Remember to check the Appendix for additional suggestions.

1. Determine how many turtle steps it takes to get to the top edge of the screen.
2. Determine how many turtle steps there are from the bottom edge of the screen to the top. From the left edge to the right.
3. (Tricky one) How many turtle steps from the lower left corner of the split screen to the upper right corner?
4. (Trickier still) How many turtle steps 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), BACKGROUND (BG), SINGLECOLOR, and DOUBLECOLOR

There are sixteen available pencolors and sixteen background colors. The colors are numbered from 0 to 15. Note that the colors marked on the keyboard are off by 1. This is because colors in Logo start with 0 and on the Commodore 64 they start with 1.

Here are the colors and numbers for both pencolor and background color:

Color	Number
Black	0
White	1
Red	2
Cyan	3
Purple	4
Green	5
Blue	6
Yellow	7
Orange	8
Brown	9
Light Red	10
Gray 1	11
Gray 2	12
Light Green	13
Light Blue	14
Gray 3	15

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 2
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 4
RT 135
FD 62
```

In addition, changing the background color after a picture is drawn may change some of the lines in peculiar ways. Returning to the original backgound 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 take on different textures, but black and white remain the same as always.

In the default setting which you have been using, called SINGLECOLOR, Logo draws with thin lines. The colors can interfere with each other when the lines are too close. The command DOUBLECOLOR will cause Logo to draw thicker lines and allow different pen colors to be drawn closer together without affecting each other. When going from SINGLE to DOUBLECOLOR or vice versa, the graphics screen is cleared and the pen color and background color are set back to their default values. This also happens when you leave DRAW mode and re-enter it.

Try typing

```
DOUBLECOLOR
```

and then experiment drawing with different pen and background colors. Typing DRAW or NODRAW has no effect on the SINGLECOLOR, DOUBLECOLOR setting.

The Magic of PENERASE (PENCOLOR −1): Erasing

PENERASE or PENCOLOR −1 erases the line when turtle tracks cross. This means that the turtle can erase a line by going back over it with PENERASE. To see how it works, type

```
FD 100
PC -1
BK 100
```

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

Type

```
LT 2 FD 30000
PENERASE RT 45 FD 40000
```

Vary the turn and the distance forward for different effects. Try starting the turtle at the edge of the screen. . . . Don't forget to set the PENCOLOR back to a positive number, or you won't be able to draw a line the next time you want to. Note that PENERASE is the same as PENCOLOR −1.

Now that you are familiar with several screen commands, you are ready to continue on with the next sections.

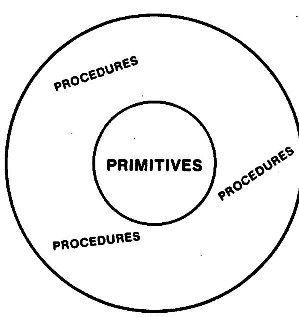
Introduction to Procedure Writing

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.

PRIMITIVE: a command built into Logo
PROCEDURE: a command that you define



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

FORWARD, BACK, LEFT, RIGHT, DRAW, and NODRAW 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

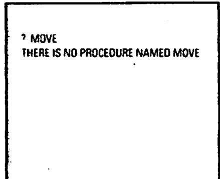
Type

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.



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, RUN Key, <CTRL> G

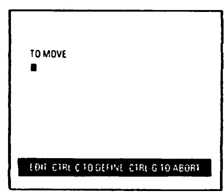
To write a procedure, start with the name. The tutorial will use the name MOVE, but you may use your own.

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
```

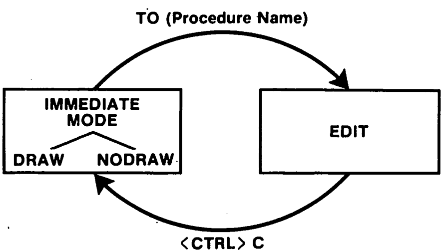
When you press <RETURN>, the screen will change: Logo will clear the screen and print the words TO MOVE on the first line. Now Logo is in EDIT mode. The cursor will be at the beginning of the next line. At the bottom of the screen there will be a white line with black letters. It always says the same 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. Hitting <RUN> will have the same effect as <CTRL> C.

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 can create the procedure. To use the procedure, you must first get out of the editor by typing <CTRL> C or <RUN/STOP>, which puts you back into IMMEDIATE mode. (But don't do this yet.)

When you are using the editor, you can use the arrows to move the cursor and <DEL> 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> key.

Press

<DEL> and then the
<RETURN>

key again to see this. Press

<DEL>

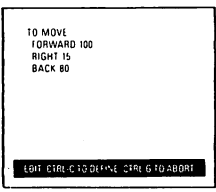
until the whole line under TO MOVE goes away. (Be careful, if you hold down the <DEL> key, it will delete several characters very quickly.)

(See the APPENDIX 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
```



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 Commodore), and
(3) that there are spaces between the commands and the numbers.

Use the arrows and the <DEL> key to fix errors. When you finish your repairs, leave the cursor where it happens to be. Logo, unlike many 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.

You can also use the <RUN/STOP> key instead of <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.)

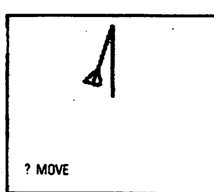
Congratulations! You have written your first procedure. You have taught the turtle a new command. But wait! It's not time for congratulations yet. Does it work? You must try it.

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, re-enter 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> key, and 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. END is needed when you define more than one procedure in the editor at the same time.)

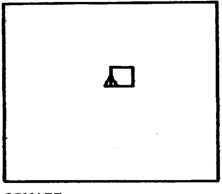
- Exit the editor with <CTRL> C or <RUN/STOP>
  (C for Complete)

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. Tape a sheet of clear plastic over the screen and draw the design you want. Then, make the turtle trace your design.

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> 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), or <RUN/STOP>.

(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 a function key or the <CTRL> key plus a letter. Use the word in a procedure; the function key is only for convenience at the keyboard.

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

Procedure Version	Keyboard Version
TEXTSCREEN	<f1>
SPLITSCREEN	<f3>
FULLSCREEN	<f5>

Remember that SPLITSCREEN can be changed so that there are more lines of text. See the definition of SPLITSCREEN in the Glossary.

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 or the up arrow, Logo will retype the previous line for you. You press <RETURN>, and Logo will execute it.

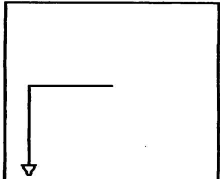
To repeat MOVE 24 times, type

```
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. One hundred steps to the left and 100 steps down turn out to be a good starting point for MOVE. 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, SHOWTURTLE (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, and 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.

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 Commodore. 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).

The SAVE command normally 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.

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

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). 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 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 blank paper.

You can also SAVE some of the procedures in the workspace without saving all of them at once. To do this, use the same format for SAVE, but after the file name, add a list of the procedures which you would like to save. (For now, a list in Logo can be defined as whatever is inside a set of square brackets.) Since SAVE usually takes only one input (the file name), you must use parentheses to tell Logo there is more than one input. For example

(SAVE "PARTSHAPES [SQUARE RECTANGLE])

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

CATALOG

Everything on the disk will be listed. 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 which is short for
PRINTOUT TITLES

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

PO SQUARE or PO MOVE

If you would like to see more than one procedure, use a list after PO, such as

PO [SQUARE RECTANGLE]

SUMMARY

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

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

You may reload procedures into your workspace at any time. The usual time would 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

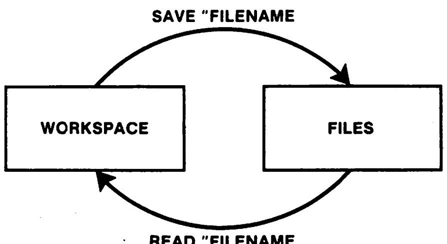
The red light on the disk drive will go on, the disk will whirr, and the computer will print out the name of each of your procedures in your file SHAPES and confirm that it has been read into your workspace by printing DEFINED. For instance,

```
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 all your procedures 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

```
GOODBYE
```

A similar result can be achieved with the command ERASE ALL. You can also use other modifiers with ERASE such as PROCEDURES and NAMES. See the Glossary for an explanation of GOODBYE and ERASE.

In addition, you may erase any individual procedure with ERASE and its name, i.e. ERASE MOVE. Be sure any procedure you want to keep is saved on disk before erasing it from your workspace. If you wish to erase several procedures at once, ERASE can take a list of procedure names just as PO can. For instance

```
ERASE [MOVE RECTANGLE]
```

To remove a file permanently from the disk, use ERASEFILE followed by the file name. For example

```
ERASEFILE "SHAPES1
```

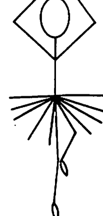
Do not type this unless you wish to get rid of the file SHAPES1 on the disk. ERASEFILE has no effect on your current workspace.

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 38 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, type

```
READPICT "DANCER
```

Picture files are stored differently than Logo files. They are stored as two files, one with the picture and the other with color information. Instead of ending with .LOGO, the files end with .PIC1 and .PIC2. When you use READPICT, it automatically looks for files with these endings.

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.

Changing the Textscreen Color: TEXTCOLOR, TEXTBG

Some monitors and TVs are not easy to read with the default colors used in Logo. If this is a problem, you can change the color of the text using TEXT-COLOR, and change the color of the background using TEXTBG. The same 16

colors can be used that are used on the graphics screen. See your Commodore 64 User's Guide for the color and key chart. Type

```
TEXTCOLOR 0
```

and the cursor will change to black. Anything that you type from now on will be black until you change the color again. TEXTBG works exactly the same way. Type

```
TEXTBG 1
```

and the background will turn white. To get back to the original colors of dark blue letters on a light blue background, use

```
TEXTBG 14
TEXTCOLOR 6
```

By pressing the <CTRL> key and any number key from 1 to 8, the text color will be changed to the corresponding color on the chart below.

Putting Letters on the Graphics Screen: STAMPCHAR

Putting text on the graphics screen can be done with the command STAMP-CHAR. STAMPCHAR works best in SINGLECOLOR (the default) mode. (But if you insist on DOUBLECOLOR, it looks best if the turtle's pencolor is the same as the text color.) To use STAMPCHAR, give it a letter, and it will print it on the screen.

```
STAMPCHAR "F
```

The quote mark is needed to tell Logo that F is a one-letter word and not a procedure that it should run. To see what you have typed, hide the turtle or pick up the pen and move it forward. Changing the PENCOLOR changes the color of the letter stamped.

See the description in the appendix of the Utilities disk file called STAMPER. The file has procedures using STAMPCHAR, with which you can print whole sentences on the graphics screen instead of single letters.

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
PENERASE	PC -1
STAMPCHAR	

FILE COMMANDS

SAVE
READ
ERASEFILE
SAVEPICT
READPICT
ERASEPICT

SCREEN COMMANDS

CLEARSCREEN	CS
DRAW	
NODRAW	ND
TEXTSCREEN	<f1>
SPLITSCREEN	<f3>
FULLSCREEN	<f5>
TEXTCOLOR	
TEXTBG	

PENCOLORS

PC 0	Black
PC 1	White
PC 2	Red
PC 3	Cyan
PC 4	Purple
PC 5	Green
PC 6	Blue
PC 7	Yellow
PC 8	Orange
PC 9	Brown
PC 10	Light Red
PC 11	Gray 1
PC 12	Gray 2
PC 13	Light Green
PC 14	Light Blue
PC 15	Gray 3

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

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

<CTRL> P, <CTRL> C, and <CTRL> G are keyboard instructions which usually cannot be used in procedures. See the explanation of the <INST> key in the Appendix for how they can be included in procedures, along with other control keys and cursor keys.

More About the Editor: <CTRL> P, <CTRL> N, <CTRL> O, <CTRL> A, <CTRL> L, <CTRL> D, <CTRL> K, Arrow Keys

In EDIT mode, you must often move the cursor from one line to another. One way to do this is to use an arrow key. Remember, the arrow keys have an auto-repeat feature, so holding down an arrow key will cause the cursor to move until you release the key.

Another way to move the cursor is to type

<CTRL> P

to go to the Previous line (Up on the screen),

<CTRL> N

to move the cursor to the Next line (Down on the screen).

(This is the same <CTRL> P you used in IMMEDIATE mode to get Logo to retype the previous line for you. Note that many things are different when you are in the editor.)

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

<CTRL> O (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 Last character of the line, type

<CTRL> L

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> K

Other editing commands are described in the APPENDIX.

SUMMARY OF EDITING COMMANDS

	MOVING BACKWARD	MOVING FORWARD
By 1 character	Left arrow	Right arrow
To end of line	<CTRL> A	<CTRL> L
To adjacent line	<CTRL> P or Up arrow	<CTRL> N or Down arrow
	DELETING BACKWARD	DELETING FORWARD
By 1 character	<DEL>	<CTRL> D
By Line	—	<CTRL> K

FOR EASY INSERTION OF A LINE

Open line	<CTRL> O

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)

PO SQUARE

To list several procedures in your workspace, type

PO [procedure name procedure name procedure name]

PO [SQUARE BOX RECTANGLE]

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 the entire contents of your workspace. (PO can also use PROCEDURES, NAMES, and TITLES as its input instead of ALL or a procedure name.) Use

<CTRL> W (W for Wait)

to stop the scrolling. Any key which you hit next, including <CTRL> W, will cause the scrolling to continue. Stop it again with <CTRL> W as before.

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 the entire contents of the 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 Logo 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, and the colon isn't used grammatically as a colon. Read :DIMENSION as DOTS DIMENSION.
2. Variable names are just as much your choice as procedure names. We could have written

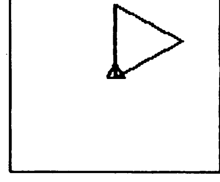
   TO BOX :WIDTH or
   TO BOX :DIST or even
   TO BOX :X

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.

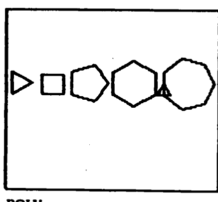
POLY will need two names for the variable inputs, and they should clearly describe what they are for.

: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:

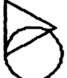
```
TO FIGURE :NUMBER.OF.SIDES
  REPEAT :NUMBER.OF.SIDES [ FD 50 RT 360/:NUMBER.OF.SIDES ]
END
```

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

```
FIGURE 3
FIGURE 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 FIGURE :NUMBER.OF.SIDES :DIST
  REPEAT :NUMBER.OF.SIDES [FD :DIST RT 360/:NUMBER.OF.SIDES]
END
```



Try

FIGURE 3 50 and FIGURE 50 3

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

This procedure is identical to POLY except that the variables have different names and are in a different order in the title.

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.

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.

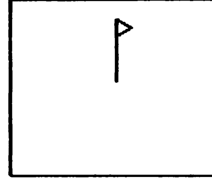
A subprocedure is useful when you want to use a procedure as a new primitive in a variety of procedures, or several times in one procedure. You could write a procedure to do one side of a square (such as FD 73) and one turn (RT 90). If you

called it SQUARESIDE, then your square procedure would look like this:

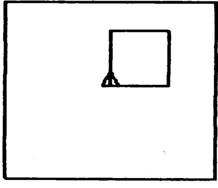
```
TO SQUARE2
  SQUARESIDE
  SQUARESIDE
  SQUARESIDE
  SQUARESIDE
END
```

(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:

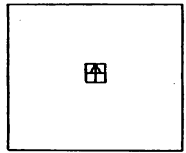
SQUARE2 uses SQUARESIDE as a subprocedure.

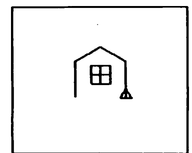
We write WINDOW which uses SQUARE2 as a subprocedure.

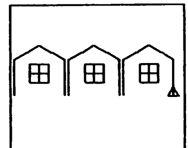
SQUARE2, which has SQUARESIDE as a subprocedure, is now also a subprocedure.

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.

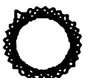
The ability of a procedure to call itself is called recursion. We shall work up to the power of recursion with some simple examples. What happens when you tell a procedure to do itself? Let's try it with a square program:

```
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.

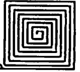
Recursion: Changing the Input WRAP, NOWRAP

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.

What happens when you run this procedure: Type


SQUARAL

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 create some unexpected effects.

Remember, you can use FULLSCREEN or <f5>, SPLITSCREEN or <f3>, and TEXTSCREEN or <f1> to change the amount of drawing space showing on the screen. (See the Appendix to see how SPLITSCREEN can be modified to show different amounts of text at the bottom.)

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 NOWRAP mode.
6. 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 50
  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 50
  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 the Appendix.

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 COUNT.ONE :NUMBER
  IF :NUMBER > 2 STOP
  PRINT :NUMBER
  COUNT.ONE :NUMBER + 1
END

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

COUNT.ONE works in the same way as DESIGN. Type

COUNT.ONE 1 and Logo will respond
1
2

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

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.

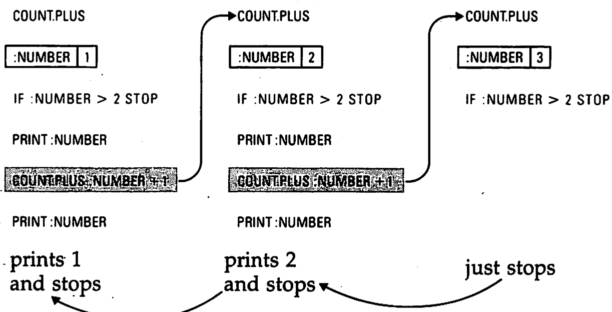
COUNT.PLUS 2

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.



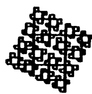
The process of recursion is based on one idea:

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 THEN 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 that the only difference between SQS and SQR is the placement of the recursive call.

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

Recursion Projects

1. Write a set of procedures which draw successively smaller houses. Use sub-procedures for the parts of the house.
2. Write a procedure to draw a binary tree. A binary tree is a v-shape with a smaller v-shape 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

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 3
  BG 0 NOTSQ
  BG 1 NOTSQ
  BG 2 NOTSQ
  FLASH.NOTSQ
END
```

FLASH.NOTSQ sets the pencolor to 3, 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 red, four more lines, then the whole procedure repeats endlessly.

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 16 will choose a number from 0 to 15 (just what PC and BG need).

The number RANDOM chooses is called its OUTPUT. If you type RANDOM 16 at the keyboard, Logo will respond with RESULT: 4 (or some other number from 0 to 15), 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]
```

It works!

To make the turtle's pen or the background take on a random color, use RANDOM 16 instead of the number. FLASH.NOTSQ could now be

```
TO FLASH.NOTSQ
  PC 3
  BG RANDOM 16
  NOTSQ
  FLASH.NOTSQ
END
```

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

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

To avoid using black (color 0), use 1 + RANDOM 15. This gives you a random number from 1 to 15 because 1 is always added to a random number from 0 to 14.

Try adding this line to one of your procedures:

```
BG 1 + RANDOM 15
```

Note that the number used with 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 the order of computation.

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 or <RUN/STOP> 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 (IF :TIMES < 0 THEN STOP) 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 SHOWTURTLE (or ST), hide the turtle with HT, print the procedure out with PO, PRINT variable values, or do a number of things to help you debug your program. You can use any primitive or procedure during a pause.

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. They are :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 continuous line on your screen.

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

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.

Commenting Your Program: ;

When you use a semi-colon in a program, anything that comes after it on the same line is considered a comment. (Remember however that a line may wrap onto another line and still be considered one line.) The comment is not evaluated or printed out when the procedure is run. You only see it when the procedure is printed out.

Putting comments in a program makes it easier for other people to understand what is going on. It also helps when you haven't used a program for a while and can't remember exactly how it works.

More About the Turtle: DRAWSTATE, HEADING, SETHEADING (SETH), TOWARDS

Logo primitives which give information about the turtle are useful for testing. DRAWSTATE is a good example, giving a list of nine pieces of information. Type

DRAW DRAWSTATE and Logo will reply
RESULT: [TRUE TRUE 11 1 DRAW SINGLECOLOR SPLITSCREEN 14 6]

if 1. It is TRUE that the pen is down
2. It is TRUE that the turtle is visible
3. Background color is 11 (grey)
4. Pencolor is 1 (white)
5. Immediate mode is in DRAW mode
6. Color mode is SINGLECOLOR
7. Screen format is SPLITSCREEN
8. Text background color is 14 (light blue)
9. Text color is 6 (dark blue)

Word and List commands, FIRST, BUTFIRST, and ITEM can be used to test a member of the list. For example, the background color is ITEM 3 DRAW-STATE. You can also print the information, i.e. PRINT DRAWSTATE.

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, you can use both:

```
SETHEADING HEADING + 5
```

will turn the turtle 5 degrees to the right. (Note that this is exactly the same as RIGHT 5.)

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 screen is a grid, with X going across and Y going up and down. The HOME position is where both x and y are zero. X gets larger to the right, Y gets larger going up. X is negative to the left of HOME, 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 Glossary.

INSTANT: Logo Turtle Graphics for the Non-reader

Your Logo Utilities 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 Commodore 64 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. The file has a variable called STARTUP which automatically starts the program. 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 THE LAST COMMAND)
N   NAMES THE PICTURE
P   SHOWS A PICTURE, ASKS FOR ITS NAME.
C   FOR CHANGING THE SCREEN COLOR
?   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 redraw 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 redrawing, even though it is not visible in the finished drawing. For a Lively Line, try typing

```
FRLLR   FRLLR   FRLLR   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 may be of any length, not just single letters.) INSTANT will create a Logo procedure with that name, containing all the steps that had been taken when N was typed.

To get a picture back, type P and its name. When the picture (actually a 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. This allows you, using N and P, to do structured programming. You can have procedures calling other procedures using just the one letter INSTANT commands.

The following INSTANT session might look familiar. The letters below are INSTANT commands. When actually running the program, the letters do not print out when you type them. (Here we show the computer's responses in italics.)

F
F
R
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 the graphics section, except for minor changes such as three RIGHT 30 commands being used instead of RIGHT 90.

Typing ? produces the menu again, without damaging the picture.

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

1. Type <CTRL> G to return to Logo.
2. Type function key <f1> 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.

To save a picture on the disk, return to Logo with <CTRL> G and type SAVE-PICT " 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

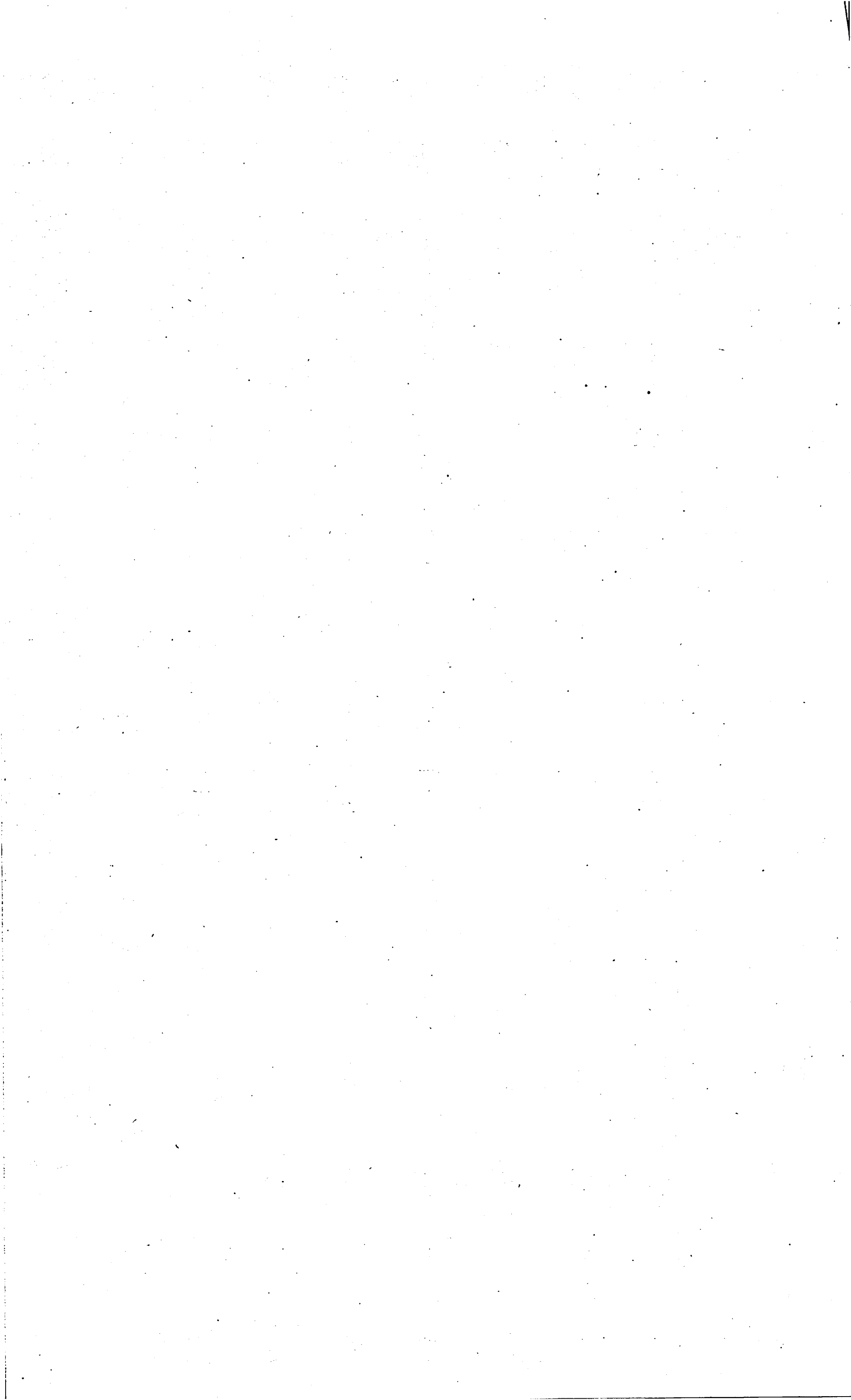
```
INSTANT
```

to return to the INSTANT system.

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 only want the procedures created by INSTANT, you can use SAVE with two inputs, a filename and the list of procedures. See the Saving Procedures section above and SAVE in the Glossary.

You can easily modify INSTANT to provide more complex commands. To change INSTANT, edit the COMMAND procedure.



COMPUTATION: HANDLING NUMBERS

C

The attempt has been made to make each chapter in the tutorial independent with no other chapter as prerequisite. If, however, there are questions, consult the Graphics chapter, which contains the most complete explanations.

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.

Arithmetic Operations

In Logo, you must type arithmetic expressions on one line — not spread out over two or more like when you write them down on paper. For the operations of addition, subtraction, multiplication, and division, Logo uses the following symbols:

		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 right 72 degrees.

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.

If you have trouble remembering the order Logo uses, just think of "My Dear Aunt Sally." Multiplication and division come before addition and subtraction. Examples:

1.	4	*	3	+	6	/	3	−	2	*	(3	+	2)
Step 1	4	*	3	+	6	/	3	−	2	*		5	
Step 2		12		+		2		−		10			
Step 3				14				−		10			
Step 4								4					

2.	4	*	(3	+	6)	/	(3	−	2)	*	3	+	2
Step 1	4	*		9		/		1		*	3	+	2
Step 2		36				/		1		*	3	+	2
Step 3						36				*	3	+	2
Step 4										108		+	2
Step 5												110	

3.	4	*	(3	+	6)	/	((3	−	2)	*	3	+	2)
Step 1	4	*	(	9	)	/	((	1	)	*	3	+	2)
Step 2	4	*	(	9	)	/	(			3		+	2)
Step 3		36				/	(					5	)
Step 4						7.2							

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	Result will be
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:

INTEGER 4.3	outputs 4
INTEGER 4.9	outputs 4
INTEGER −4.3	outputs −4
INTEGER −4.9	outputs −4
INTEGER 7/2	outputs 3

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

QUOTIENT 7 2	outputs 3
QUOTIENT 1 2	outputs 0
QUOTIENT −7 2	outputs −3

Note that QUOTIENT :A :B is not always the same as INTEGER :A/:B, since QUOTIENT rounds its inputs to the nearest integer.

REMAINDER outputs what is left over from the integer division:

REMAINDER 7 2	outputs 1
REMAINDER 2 3	outputs 2
REMAINDER 5 −2	causes an error

The error is caused because Logo considers 5−2 to be its first input and it can't find a second input. There are two ways to get around this.

REMAINDER 5 (−2)	outputs 1
REMAINDER 5 0−2	

When you use real numbers with QUOTIENT or REMAINDER, they are ROUNDed to integers before the division takes place. (This is the case with all Logo primitives which require integer inputs.)

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

SQRT 160	outputs 12.6491
SQRT 16	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, LOCAL

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 leaves 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 and it is stored when a copy of the workspace is saved. Variables created with MAKE are called Global Variables. (There is an exception: If you use MAKE on a variable name which is an input to the current procedure (or the procedure which called the current procedure...) it will change the value of the input variable, rather than creating a global variable.) 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 the procedure is running, the variable has a value. When the procedure stops, the variable no longer exists until it is given a value 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: Any time MAKE is used after the command LOCAL, the variable is treated as a local and not a global variable. See the following two sections on procedures for examples. Also see the Glossary for a more complete description of LOCAL.

> Local variables are important because they keep the workspace from becoming cluttered. Using global variables when local variables would 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 that 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 cube a number means to multiply it 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 (or <RUN/STOP>) 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 an input to a procedure. It acts as 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

LOCAL is useful when you don't want to use an input to your procedure, 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 a simple 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) will be treated as a local variable for the rest of the procedure. The variable name is indicated by a leading quote mark. The local variable has no value until MAKE is used to give it one. (To see this, put a PRINT :X in between LOCAL and MAKE.) In the procedure, RANDOM is used to pick a random value for the variables. The last line of the procedure PRINTs the variables and their product as a part of a sentence.

> PRINT usually takes one input which can be a word, a list, or a number. In this case it has five inputs; the value of the variables :X,:Y; the product of :X*:Y; and two one item lists which are denoted by the brackets. (What would happen if TIMES and IS were not in brackets?) For Logo to know that there is more than one input, the command PRINT and the inputs must be enclosed in parentheses. If you doubt our word, try it without the parentheses and see what happens.

To make the procedure continue to pick variables and print the answers, add MULTIPLY as the last line in the procedure just before END. Don't forget about <CTRL> G to stop it. (The ability of a procedure to call itself is called recursion. For a more complete explanation, see the following section, and also the discussion of recursion in the Graphics chapter.)

We now have a procedure that is very good at picking numbers and telling us their product, but we do not yet have an interactive procedure. There is no way for you to do anything while the procedure is running. Let's change it so that you have to type in the answer instead of the computer doing it for you. Right now, Logo will tell you the answer, so that must be changed or there will be no challenge. The line with PRINT will become

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

Of course, we'll want Logo to tell us whether the answer we give is right or wrong. The procedure listed below will do this for us and it can be used in MULTIPLY.

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

This is very simple. It asks if the value of the variable ANSWER is equal to the value of X times the value of Y. If this is TRUE, it will print CORRECT and stop. If it is false, it will go to the next line and print INCORRECT.

Now then, how do we combine the two procedures? If we add TESTANSWER :ANSWER as the line after the PRINT statement in MULTIPLY which asks how much X times Y is, where does :ANSWER come from?

To have a user input an answer, we need to use REQUEST. REQUEST 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
```

As used here, REQUEST changes what you type (your ANSWER) into the input or variable needed by TESTANSWER. Why does it have FIRST in between TESTANSWER and REQUEST? Remember, REQUEST generates a list. (Essentially, a list is anything enclosed in brackets.) In TESTANSWER we would be comparing a list with a number and the answer would always be INCORRECT. What is needed is the first (and only) item in the list, the number which you type in. Therefore, we use the Logo command FIRST in conjunction with REQUEST.

Note that the values of X and Y were not input to TESTANSWER, but were used by it. A subprocedure has access to the values of the variables in the procedure or procedures which call it, whether the values are passed through the procedure title or not. In its present state, TESTANSWER could not run on its own, unless there were a global variable for X and Y. In addition, if X and Y are not global variables, any procedure which calls TEST-ANSWER must have X and Y as local variables or as inputs in the procedure title.

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 was correct? To do this, we need a recursive call to TESTANSWER so it will keep calling itself until it receives the correct answer. What happens if you make the last line of TESTANSWER be TESTANSWER :ANSWER? When you try this it obviously does not work! So we need to change :ANSWER in the last line to something else. Hint: We have figured it out previously.

Why all this fuss about local variables, whether created by LOCAL or as procedure inputs? Global variables take up space. The less they are used, the more room there is to do other things. Also, procedures which use global variables to store temporary results will often not work properly when used recursively.

A note to teachers: The above procedures are the types of programs that students quickly learn how to modify and create themselves, even in elementary school. They are not meant to be an example of how to develop quizzes for students.

A note to students: If teachers try to make you learn addition facts with the above procedures, fool them by editing the procedures to make them do something else.

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 to do 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 (result) on the screen.

If you run a procedure which in turn calls a procedure that 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. Compare this with

```
FORWARD CUBE 3
```

OUTPUT was not needed in TESTANSWER in the previous section because there was no information that needed to be sent back to the procedure MULTIPLY or which needed to be sent back to itself in a recursive call.

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:

```
EXPONENT 3 4 = 3 *(EXPONENT 3 3)
             = 3 *(3 *(EXPONENT 3 2))
             = 3 *(3 *(3 * EXPONENT 3 1))
             = 3 *(3 *(3 *(3 * EXPONENT 3 0)))
```

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

```
EXPONENT 3 4 = 3 * (3 * (3 * (3 * (1))))
              EXPONENT 3 0 outputs 1
             = 3 * (3 * (3 * (3 * 1)))
          EXPONENT 3 1 outputs 3
             = 3 * (3 * (3 * 3))
     EXPONENT 3 2 outputs 9
             = 3 * (3 * 9)
 EXPONENT 3 3 outputs 27
             = 3 * 27
EXPONENT 3 4 outputs 81
```

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 (−155), make it large enough to be visible, and stop at the right edge of the screen (+155).

To begin drawing at the left edge and yet have :X start at 0 for the evaluation of :Y, the X position becomes :X − 155.

To see :Y, which will vary between 0 and 1, multiply by 100 (actually anything up to 130, the vertical limits of the screen).

The procedure starts out as

```
TO GRAPH.SIN :X
  SETXY :X - 155    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 - 155    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 − 155).

```
TO GRAPH.SIN :X
  IF :X - 155 > 155 STOP
  SETXY :X - 155    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 155 0
  HOME
  SETXY -155 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. This is 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
  LOCAL "Y
  IF COS :X = 0 THEN MAKE "X :X + 1
  IF :X - 155 > 155 STOP
  MAKE "Y (SIN :X) / (COS :X)
  IF 10 * :Y > 115 PU MAKE "Y 12
  IF 10 * :Y < -115 PU MAKE "Y (-12)
  SETXY :X - 155 :Y * 10
  PD
  GRAPH.TAN :X + 5
END
```

Here Y is evaluated separately because it must be tested before the drawing step. If 10*:Y is greater than 120 or less than −120, the turtle will wrap and the turtle will be left at random places on the screen. This will cause unwanted lines when Y comes back in the range of the screen and the pen is put down, so Y is reset to the edge of the screen until its value places it within the screen.

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
  LOCAL "Y
  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 an 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 (See the Glossary for a description of ANYOF):

```
TO GRAPH.P :X :A
  LOCAL "Y
  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). You will also have to remove the DRAW primitive from AXIS.

```
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 (-:INC)
  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 HT
  GRAPH.ELLIPSE -:A :A :B 1
END
```

To speed the drawing of the ellipse change the increment to 2. In ELLIPSE, the 1 becomes a 2 in the third line. In GRAPH.ELLIPSE, SETXY etc. becomes SETXY :X*2 etc.

WORDS AND 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 is the basis of a variety of video games and simulations. Interactive language can not only add attractive features to these games, but can 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 (large quantities of) indented text and come back to it later, 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 chapter 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 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 that are 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 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 further in this guide.

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
"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 9, 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 FIGURE :LENGTH :SIDES
 REPEAT :SIDES [FD :LENGTH RT 360/:SIDES]
END
```

This procedure tells the turtle how to draw a polygon 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

FIGURE 73 4 or FIGURE 15 6

Logo puts the 73 or 15 in a location (think of it as a box) that the FIGURE procedure refers to as LENGTH and puts the 4 or 6 into another location that FIGURE refers to as SIDES.

It is important to remember that **LENGTH and SIDES are names that FIGURE uses to keep track of these numbers,** and that **no other procedure knows what FIGURE keeps in the boxes or even that the boxes exist!** Further, those boxes **cease to exist after FIGURE finishes its work.**

> Please note, however, that if FIGURE had called any subprocedures during its execution, those subprocedures would also have had access to the values in FIGURE's boxes.

Before getting back to MAKE, define FIGURE as shown above and then type

FIGURE 50 5

While FIGURE 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 FIGURE 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 and it has a 10 in it, you might think that you could now type just FIGURE 4 to get a four-sided shape with a size of 10.

If you try that, Logo will complain that FIGURE needs more inputs. Because FIGURE was defined to take two inputs, it must always be given two inputs.

Type

```
FIGURE 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 FIGURE (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, FIGURE, 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 FIGURE 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 the workspace.

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-looking 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 in the workspace 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 and referred to everyone's public box named NUM. The private variable is called a local variable, and the public one is a global variable.

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 troublemakers (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 EASY does not have the variable name MULTIPLE in its title line, 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.

Interactive Programs 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.

By the way, the comments FASTER and SLOWER are just there to make it easier to remember how the program works. A semicolon in an instruction line tells Logo to ignore what follows, allowing you to write notes to yourself as reminders.

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 to start the program 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.

   This procedure is very interesting to experiment with. Try to learn to draw with it.

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! ('Tain't easy!)

INTERACTIVE LANGUAGE

Don't Skip This Section!
MEMBER?, EMPTY?

Right now, we would like you to get to know two very important Logo primitives. While you won't be using them right away, MEMBER? and EMPTY? will appear throughout this section of the tutorial, so you should have a good idea of what they do.

MEMBER? takes two inputs — a word and a list — and outputs the value "TRUE if the word is one of the elements of the list. (You can also give MEMBER? a letter/number and a word, and it will return "TRUE if the letter/ number 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 how MEMBER? works, type these commands exactly:

```
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
```

Some Friendly Introductions: SENTENCE (SE), REQUEST, 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 is, it 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 primitive EMPTY? to check for bashful typists.

```
TO GREET :PERSON
  IF EMPTY? :PERSON PRINT [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 no-response 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 at the end, 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 PRINT [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. Be careful, when you do that, to make sure that when you are finished all of the open and closed 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. The text screen will be cleared off and the cursor will be placed at the upper left. (You can achieve the same effect by typing <SHIFT> <CLR>.)

While working on this chapter you will often need to clear the text screen. To get some practice in, let's mess up the text screen some. Typing the following lines should create plenty of mess:

```
ABC
+
:
```

Messy enough? Now type <SHIFT> <CLR>. 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>
[COMMODORE 64 LOGO] <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
[COMMODORE 64 LOGO]
RESULT: [COMMODORE 64 LOGO]
"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 64 or LOGO, 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 slightly the appearance of what you type. Using the same three examples, PRINT 5, PRINT [COMMODORE 64 LOGO], 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 ([COMMODORE 64 LOGO]), 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.

If your screen is cluttered, clear it with the primitive CLEARTEXT or with <SHIFT> <CLR>.

There are other ways of giving objects to Logo in immediate command mode other than by placing them there yourself. You can let a primitive 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 and primitives 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 primitive or procedure can create them.

Some primitives create objects as output available to other commands and some 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 input 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

So far, you have used commands which output values, such as RANDOM and MEMBER?, but 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 to "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.

A note about terminology: object and value.

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
 OUTPUT 3.14159
END

TO CIRCUMF :DIAMETER
 OP PI * :DIAMETER
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 (or the greater the turn), 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 just 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!)

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 primitive BUTFIRST (abbreviated BF) 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 with the comments written before or after the experiments.

Logo can select an object from either end of a list,

```
FIRST ACTIONS
LAST ACTIONS
```

and from either end of a word.

Type:

```
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. 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 several of these operations into one command.

```
PR (SE FIRST BF BF PEOPLE
       LAST BL BL 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 these commands which include the new primitive WORD.

```
PR FIRST "SANDY       (don't forget the double-quote mark)
PR LAST "VIC          (remember, no space after the ".)
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. (The primitive ITEM does this, but let's write a procedure to see how the process works.)

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. So 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. To print the ninth letter of the word "MASSACHUSETTS, type

```
PR NTH 9 "MASSACHUSETTS
```

Conveniently, you will not have to define NTH each time you use Logo. Remember, the primitive ITEM does exactly what NTH does.

Projects Using ITEM and Recursion

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.)

DEFINITIONS AND MODELS

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 see what you want, consult the 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 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.

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.

Primitive that passes an object from one procedure to another:

OUTPUT (OP) — Causes a procedure to STOP and output an object to another procedure.

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.

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. (Compare with FPRINT in Glossary.)

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 a literal word, and will 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
MUGWUMP
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 rather than 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'
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 and FPRINT) 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 page W&L-41 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 ITEM 1 ODDWORD
PR ITEM 2 ODDWORD
PR ITEM 3 ODDWORD
PR ITEM 10 ODDWORD
PR LAST ODDWORD
PR WORD ITEM 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.

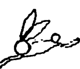
A space can be typed, and the single-quote character allows you to insert that space inside a word, but there are some characters that cannot be typed in to a procedure at all. An example is the <CTRL> G character. If you were to try typing

```
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



> Note that on the Commodore 64, the INST key allows you to type in control characters without using the command CHAR. See the Reference Guide for an explanation of the INST key.

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.)

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 401-9961]
[[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.

Programming — Some Metaphors and Some Review

Since we focus most of our attention in this chapter on the behavior of procedures, it is worth spending a moment clarifying the language that we use.

Procedures have a "behavior." Very early you learned the primitive behaviors of DRAW and FORWARD. You designed and created graphics procedures with a more complex behavior.

Throughout the chapter, we have treated designing a program as a matter of clearly describing a behavior to yourself in your own best language (that might be English) and then translating from that description into the machine's best language (in this case, Logo).

As you gain skill in Logo, you may find yourself "thinking in Logo" and skipping the translation step.

You've noted that some primitives, like DRAW and PU (PENUP), need no information to do their jobs. Other primitives, like FD and PR, cannot act without further input — a distance for FORWARD to move the turtle, and some object for PRINT to print.

In most of the graphics procedures that you wrote, you provided the inputs to FD yourself. In some cases, as in the case of BOX below, you provided FD's fixed input (50) at the time that you defined the procedure.

In other cases, as in STAR, you provided FD with an input that remained undefined until you ran the procedure. When you ran the procedure, you provided STAR with an input, and STAR passed that input along to FD.

```
TO BOX
 REPEAT 4 [FD 50 RT 90]
END
```

```
TO STAR :SIZE
 REPEAT 5 [FD :SIZE RT 144]
END
```

Another way for a procedure to receive its input is from another procedure. Inputs to graphics procedures were only rarely the outputs of other procedures, but you have already seen this kind of object passing used extensively in this chapter.

The manipulation of objects constantly involves passing messages from operation to operation: one procedure's output can be another procedure's input.

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 "antidisestablish-mentarianism!"
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 there is a beginning quote, but no quote after the word. The end of the word is shown by a space. 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
```

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, say 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].

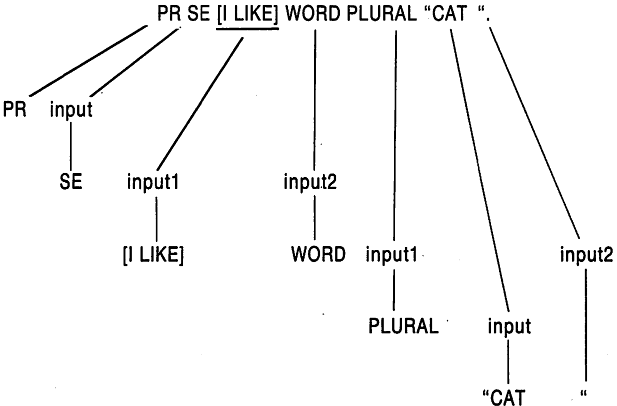
Try to predict what each of these commands will do, and then type them to see how each works:

```
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 above.

Logo reads from left to right, but as you will see by following the diagram and the discussion on the next page, 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 first 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 complain that SENTENCE 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

Testing out PLURAL reveals a number of bugs. Try the following inputs:

```
PLURAL "CAT
PLURAL "DOG
PLURAL "TURTLE
PLURAL "FATHER
PLURAL "FLY
PLURAL "OCTOPUS
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]

All along, we've been using IF without any explanation of its structure. In the procedure you just defined, 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. (See the Glossary for detailed explanations.)

3) The THEN clause: an action to perform if the condition is TRUE. (In this case, the action is OP SE BL :NOUN PLURAL LAST :NOUN .)

An IF statement can also have an additional two parts when desired.

1) The word ELSE

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 in the procedure you just defined, 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, and so it is important to become familiar with their behavior. Type these expressions to Logo:

```
LIST? PEOPLE
LIST? FIRST PEOPLE
LIST? ITEM 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 MEMBER? primitive (which takes two inputs) are also predicates.

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

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.

Actually, if you place the new instruction first, PLURAL will take more time to output the plural of [SILVER FOX] than if you place the new instruction second. (Figure out why!) But, in a procedure like this, you'd never notice the difference.

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."

```
... OUTPUT special.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.

In the first project below, you define PLURAL with the new instruction and test it out.

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 it 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 "'= ' )
 IF (:NUMBER1 + :NUMBER2) = REQUEST  PR [YAY!] STOP            (buggy line)
 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 data-base 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.

    If you are more ambitious, you might generate a random sentence, leaving out the verb, and have the person type in the correct verb.

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 instead of the two inputs that SENTENCE normally expects.

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.

type as one line

```
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 a 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 (WORD? or LIST?) 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.

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 allow 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 to which RUN.AND.RECORD can add its new moves.

```
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.

We will now create procedures to accomplish these functions.

> 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 Appendix 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. 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

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:	TO SUBST :KEY :NEW :CONTEXT
condition for stopping:	IF :CONTEXT = [ ] OP [ ]
action to take with first element:	if key replace with new
recursive call with BF input:	continue looking
end	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 butfirst 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.

type as one line

```
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 EXCEPTIONAL 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 **that** 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 this

```
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 two 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 keywords to look for. Then it would systematically look through the context for instances of each of the keywords 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. 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. Thereafter, @@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 interpreter 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 stands now. 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 at all dealt 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
```

The empty test is very often the important stop condition for recursive procedures manipulating language.

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 friends 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.

SPRITES

Introduction

By now, you've probably begun to wonder whether turtle graphics was more than a simple amusement. Perhaps you wanted to make the turtle look a bit more interesting. Maybe you even wished you could control more than one.

You're in luck! Commodore 64 Logo gives you the ability to design and control eight separate moveable objects called sprites.

Sprites can do anything the turtle can do — hide, draw, write text on the graphics screen, and so on. (In fact, the turtle is really a sprite, too.) Moreover, sprites open up new possibilities for animation and 3-D simulation. To find out more about sprites and how to use them, read on.

Getting Ready To Use Sprites

In order to use sprites, place the Utilities Disk in the drive. Then, type

```
READ "SPRITES
```

Next, get into draw mode by typing

```
DRAW
```

If the turtle is not visible now, type

```
ST or SHOWTURTLE
```

Each sprite is identified by its own sprite number (0-7). The turtle is sprite 0. When the SPRITES file is first read in and turtle graphics commands (i.e. FORWARD, BACK, RIGHT, etc.) are typed, sprite 0 (the turtle) is the sprite that responds.

Talking to Sprites: TELL

Sprites will obey the sprite commands covered in this chapter as well as regular turtle graphics commands.The TELL command lets you use any of the eight available sprites. For example, after the SPRITES file is read and DRAW is typed, the only sprite that can be seen on the screen is sprite 0, the turtle. Type the following:

```
TELL 1
```

Now sprite 1 will respond to any turtle graphics or sprite commands that are typed in. If

```
ST
```

is typed, sprite 1 will appear at the center of the screen. Sprites 1 through 7 start out hidden with pens up. Since no special shapes have been defined at this point, sprites 1 through 7 will appear as solid boxes. This shape can be changed as will be seen later.

If sprite 0 (or some other sprite) is already at the center of the screen, it may be necessary to move sprite 1 (with a FD or BK command) in order to see it better. For example, type FD 100. Before sprite 0 will respond to turtle or sprite commands again, TELL 0 must be typed in.

In general, TELL makes a particular sprite (0 through 7) the "current" sprite, i.e., the sprite that responds to any turtle graphics or sprite commands. When Logo starts, sprite 0 (the turtle) is the current sprite. This sprite remains current until another sprite is specified with the TELL command.

A Program Using TELL

The following three procedures illustrate how one might use TELL to give commands to different turtles.

Suppose you wanted to program a race between a kangaroo and a bug. You can read in pre-defined shapes from the Utilities Disk to replace the box shapes. Type in

```
READ "ANIMALS
```

to give the sprites animal shapes. (The various predefined sprite shapes are listed later.) The file ANIMALS only contains variables with the names of animals. Each name has a value corresponding to a sprite. For instance, KANGAROO has a value of 2. ANIMALS automatically reads in the shapes file ANIMAL.SHAPES. The other shape files, VEHICLES, ASSORTED, RUNNER, and SHAPES, also use this two file setup.

The following procedure might be used to move the kangaroo (sprite shape 2).

```
TO HOP :N
  RT 90
  REPEAT :N [FD 10 LT 90 FD 10 RT 90 FD 3 RT 90 FD 10 LT 90]
  LT 90
END
```

Procedure HOP will cause a sprite to take a big step to the right, jump up, take a little step to the right, and jump down again a specified number of times. Before you use HOP or any procedure which is to affect a sprite's screen movement, be sure to type ST so the sprite will be displayed (i.e., TELL 2 ST HOP 10).

The next procedure could be used to move the bug (sprite shape 3 on the file ANIMALS).

```
TO CRAWL :N
  RT 90
  REPEAT :N [FD 1 FD RANDOM 24]
  LT 90
END
```

Procedure CRAWL will cause a sprite to take a small step to the right, and then move a random number of steps to the right. Before you use CRAWL, don't forget to TELL 3 ST first.

Finally, here is the RACE procedure:

```
TO RACE
  TELL 2 SETXY -125 30
  TELL 3 SETXY -125 (-30)
  REPEAT 30 [TELL 2 HOP 1 TELL 3 CRAWL 1]
END
```

In the race procedure, sprites 2 and 3 start out toward the left margin of the screen and then move across the screen in their unique ways. HOP, CRAWL, RACE, and several other procedures described in this section are in the file SPRITEDEMOS.

Type the following to load SPRITEDEMOS into the workspace.

```
READ "SPRITEDEMOS
```

Then type

```
RACE
```

to see the race between the kangaroo and the bug.

Which Sprite: WHO

The WHO command returns the number of the current sprite. For instance, type

```
WHO
```

and Logo's response should be RESULT: 3, since that is the last sprite used in RACE. If you have used TELL in between running RACE and typing WHO, the number you used with TELL should be the result. Type

```
TELL 2
WHO
```

and Logo will respond

```
RESULT: 2
```

The following procedure, ASK, lets you give commands to a sprite without changing the current sprite.

```
TO ASK :N :COMMAND.LIST
  LOCAL "CURRENT
  MAKE "CURRENT WHO
  TELL :N RUN :COMMAND.LIST
  TELL :CURRENT
END
```

The ASK procedure takes as arguments a sprite number (0 through 7) and a list of commands that the sprite is to perform. ASK then uses a local variable to save the number of the current sprite (which can be identified by WHO). Next, the specified sprite (N) is made the current sprite, and the list of commands is run. Finally, the sprite that was current when the procedure was first called is again made current.

ASK can be found in the files SPRITEDEMOS and SPRITES. Type the following to see the bug change color, draw, and hop:

```
ASK 3 [PC 4 PD HOP 10]
```

Now type WHO. The response RESULT: 2 specifies that sprite 2 is still the current sprite even though we used sprite 3 with the ASK procedure.

Sprites and the Graphics Screen: TB?

Like the turtle, sprites are distinct from the graphics screen behind them. Sprites can move around the screen without disturbing any images that have been drawn on the graphics screen.

For example, if several lines have been drawn on the screen, a sprite can move or land anywhere without changing or erasing anything. Similarly, the presence of images on the graphics screen doesn't necessarily change the behavior of a sprite.

There is a way, however, for a sprite to "interact" with the graphics screen. The TB? procedure in the SPRITES file (which you have already read in) gives the value TRUE if the current sprite is positioned over a filled-in dot on the graphics screen and gives the value false if the sprite is positioned over a dot that is the screen background color.

The following three procedures cause a sprite to move randomly around the inside of a square. TB? is used to insure that the sprite does not escape. If the sprite touches a side of the box (i.e. if TB? gives the value "TRUE, the sprite changes direction and continues moving through the interior of the box, rather than crossing the boundary and escaping. The procedures are in the file "SPRITEDEMOS. If you have not already done so, type

```
READ "SPRITEDEMOS
```

to load the file, and

```
TRAP 100
```

to run the program.

```
TO TRAP :100
  CS BACKGROUND 3
  PENCOLOR 0 ST SETH 0
  BOX :SIDE
  PU
  FD :SIDE/2 RT 90
  FD :SIDE/2 LT 90 PD
  PENCOLOR 6
  TRAPLOOP
END

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

TRAP sets up the screen and the current sprite for drawing. TRAP then tells the current sprite to draw a box and get inside of it, and then calls on TRAP-LOOP to move the current sprite around inside of the box.

```
TO TRAPLOOP
  IF RC? STOP
  IF TB? BK 1 RT RANDOM 180
  FD 1
  TRAPLOOP
END
```

TRAPLOOP moves the current sprite around inside the box until any key is pressed. Each time the current sprite moves forward, TB? is called to check if it is touching a dot on the graphics screen. If the current sprite is touching a side of the box, TB? will give the value TRUE. In that case, the sprite will move back into the box and will be assigned a random direction in which to continue.

Note: TB? works only in SINGLECOLOR mode.

Sprite Shapes

When Logo starts up, sprites 1 through 7 are automatically given box shapes. However, as we have already seen, a sprite can be any shape.

Commodore 64 Logo comes with five sets of seven sprite shapes.

— ANIMALS	Animal shapes
— VEHICLES	Vehicle shapes
— SHAPES	Geometric shapes
— ASSORTED	Assorted shapes
— RUNNER	Shapes designed for animation of a woman running

Each of these names has two files associated with it, one a normal Logo file and the other a shape file. The Logo file is loaded as usual. For example, READ "ANIMALS will load the file ANIMALS.LOGO. After loading, it will automatically load the shapes file, in this case ANIMALS.SHAPES.

The ANIMALS.SHAPES file actually contains the shapes; the ANIMALS file contains only seven variables (DINOSAUR, KANGAROO, BUG, etc.) with values ranging from 1 to 7 corresponding to the sprite shapes. The purpose is to make the sprites easier to use, allowing you to type ASK :DINOSAUR [FD 10] instead of having to know that the dinosaur is sprite 1.

If you want only the shapes, but not the associated variables in the ANIMALS file, use the READSHAPES procedure in the file named SPRITES (i.e., READ-SHAPES "ANIMALS). You can also load the shapes in directly by using the command BLOAD "ANIMALS.SHAPES, which is exactly what READSHAPES does. Again, there is a complete list of built-in sprite shapes near the end of this chapter.

Changing Sprite Size: BIGX, BIGY, SMALLX, SMALLY

Four of the commands in the SPRITES file, BIGX, BIGY, SMALLX, and SMALLY, let you change the proportions of sprites.

When the file SPRITES is first read in, the sprites are the smallest size (i.e. with size SMALLX and SMALLY). Typing BIGX when the current sprite is the smallest size will cause it to double its width by expanding to the right; typing BIGY will cause the current sprite to double its height by expanding downward.

Conversely, typing SMALLX after the current sprite has already been extended in the X (width) direction, will cause it to halve its width by shrinking to the left; typing SMALLY when the current sprite has been expanded in the Y (length) direction will make it halve its length by shrinking upward.

In summary, each sprite has four possible sizes:

SMALLX SMALLY — The smallest sprite size, which is the default

SMALLX BIGY — Smallest width, doubled height

BIGX SMALLY — Doubled width, smallest height

BIGX BIGY — Doubled width, doubled height

See the dinosaur and submarine demonstration programs documented at the end of this chapter for examples of effective use of the sprite size commands.

Changing Sprite Shapes: SETSHAPE

When a file of sprite shapes is loaded, each sprite has a unique shape. Using the SETSHAPE command, it is possible to give more than one sprite the same shape. SETSHAPE takes a sprite shape number (0 through 7) as its argument and sets the shape of the current sprite to the sprite shape which has that number.

For example, in order to program a family of three dinosaurs going out for a picnic, three sprites would have to have the dinosaur shape, which is sprite shape 1 from the shape file ANIMALS. After ANIMALS is first read in, sprite 1 will have the dinosaur shape, sprite 2 will have the kangaroo shape, and sprite 3 will have the bug shape.

After typing in

```
TELL 2 SETSHAPE 1
TELL 3 SETSHAPE 1
```

sprites 1, 2, and 3 will have the same shape (i.e. the dinosaur shape from the file ANIMALS).

Note: If another file of sprite shapes were loaded, sprites 1, 2, and 3 would still all have shape 1. However, shape 1 would now correspond to the new sprite shape 1.

For example, if the shape file VEHICLES were read in by typing

```
READSHAPES "VEHICLES
```

sprites 1, 2, and 3 would all have the truck shape, which is shape 1 in the VEHICLES file.

The following two procedures have already been read in from the SPRITE-DEMOS file. Their purpose is to line up sprites 1 through 7 on the right side of the screen, displaying each sprite in its corresponding sprite shape. These procedures, SHOW and SHOWLOOP, are especially convenient to use after a file of sprite shapes has just been loaded because they set the sprites back to their original sprite shapes and display all of the sprites.

```
TO SHOW
  LOCAL "OLD
  MAKE "OLD WHO
  TELL 0 HT
  SHOWLOOP 1 120 100
  TELL :OLD
END

TO SHOWLOOP :CURRENT :X :Y
  IF :CURRENT = 8 STOP
  TELL :CURRENT SETSHAPE :CURRENT
  SETXY :X :Y
  SMALLX SMALLY ST
  SHOWLOOP :CURRENT + 1 :X :Y - 30
END
```

SHOW uses a local variable to save the current sprite and then calls on SHOWLOOP to display the sprites, beginning with sprite 1 at position (120, 100). At the end of SHOW, the sprite that was originally current is made current again. Try typing SHOW.

Identifying the Shape of the Current Sprite: SHAPE

The SHAPE primitive gives the sprite shape number of the current sprite. After running the SHOW procedure, you would get the following responses to the SHAPE command:

```
TELL 0 SHAPE
RESULT: 0
TELL 1 SHAPE
RESULT: 1
TELL 2 SHAPE
RESULT: 2
```

etc.

If you had typed the above before typing SHOW, the response for sprites 2 and 3 would have been RESULT:1 because of the SETSHAPE 1 commands used earlier. To see that SHAPE tells you the sprite's shape but not necessarily the sprite's number, type

```
TELL 4 SHAPE
SETSHAPE 6
```

Now you will get the following response to the SHAPE command:

```
SHAPE
RESULT: 6
```

To see that you are still using sprite 4 (which has shape 6) and not sprite 6, type WHO. The response should be RESULT: 4. Remember, WHO tells you which sprite is current, and SHAPE tells you which shape the current sprite is carrying.

The Sprite Editor: EDSH

Commodore 64 Logo comes with five files of built-in sprite shapes: ANIMALS, VEHICLES, SHAPES, ASSORTED, and RUNNER. Using the sprite editor in the file SPRED, it is possible to modify these shapes and to create original sprite shapes.

To use the sprite editor, type

```
READ "SPRED
```

The file SPRED also contains the sprite commands that are in the file SPRITES. Therefore, reading in SPRED will load all the commands necessary to use both sprites and the sprite editor.

Remember to tell Logo which sprite shape you want to edit. (Use the TELL command.) Be careful! The shape used by the current sprite at the time EDSH is typed will be edited. If, for example sprite 7 were current and had been assigned shape 5 with a SETSHAPE 5 command, the shape that would be edited would be shape 5, not shape 7.

If you have just read the sprite editor into a new workspace, the current sprite is sprite 0, the turtle. If you don't use TELL, you might temporarily edit the turtle shape but be unable to save your design.

Once the file SPRED has been read in, the sprite editor is invoked with the command

EDSH

which is short for EDitSHape. When EDSH is typed, several things happen. First, the screen clears and the current sprite appears in the middle of the screen. Next, a large box containing an image of the current sprite is drawn on the left side of the screen, with dots for blanks and balls for filled-in pixels.

It is sometimes easiest to design sprite shapes by drawing them out on graph paper before typing them into the sprite editor. Make a box 24 blocks wide and 21 blocks high. Then you can fill in blocks to see how a sprite will look.

Moving Around and Drawing in the Sprite Editor

The sprite editor enables you to create sprite shapes by filling in and blanking out pixels. The interior of the sprite editor box is 24 characters wide and 21 characters high. Each character position represents a pixel of the sprite. Therefore, when a character position is filled in (or blanked out) on the sprite editor box, the corresponding pixel of the sprite is filled in (or blanked out).

MOVING

The sprite editor cursor is initially positioned in the upper left corner of the Sprite Editor box. The cursor can be moved by the left, right, up, and down cursor keys, and with <RETURN>. Pressing the <HOME> key will bring the cursor back to the upper left corner.

NOTE: Sometimes the sprite editor executes commands a little slowly. If several keys are pressed in succession (for example, if <up arrow> is pressed five times to move the cursor up five lines), the sprite editor may not respond immediately. However, within a second or two, the sprite editor should have performed all the commands typed in.

FILLING IN PIXELS

The asterisk key <*> or the plus key <+> will fill in both the space under the cursor and the corresponding pixel of the current sprite. The <*> key will advance the cursor one pixel to the right, and the <+> key will leave the cursor over the space that was just filled in.

BLANKING OUT PIXELS

The space bar or the minus <−> key will clear both the space under the cursor and the corresponding pixel of the current sprite. The space bar will advance the cursor one pixel to the right, and the <−> key will leave the cursor over the space that was just blanked out.

The <DEL> key will clear the pixel immediately to the left of the cursor and will move the cursor one pixel to the left.

The shape in the sprite editor and any sprites with that shape number can be blanked out by holding down <SHIFT> and pressing the <CLR> key. Be very careful with this command!

CHANGING SPRITE SIZE

When using the Sprite Editor, it is possible to see how the sprite currently being edited will look in each of its four possible sizes. Typing X will toggle the width of the sprite between its small (24 pixels) and big (48 pixels) width size, and typing Y will toggle the length of the sprite between its small (21 pixels) and big (42 pixels) length size.

REVERSING THE SPRITE SHAPE

Typing <CTRL> 9 causes the sprite editor to reverse the shape: that is, all filled-in pixels are erased and all blank pixels are filled in. To restore the shape, type <CTRL> 9 again.

EXITING THE SPRITE EDITOR

To exit the sprite editor and save your work, press the <RUN/STOP> key or <CTRL> C. The current sprite and any other sprites with the same shape number as the current sprite will now have the shape that was just created with the Sprite Editor.

The sprite's shape changes as you edit it, so you can not type <CTRL> G and regain the shape as it was before you started. If you want the original version, you must read the shape file from the disk again.

SUMMARY OF
SPRITE EDITOR COMMANDS

Key	Effect
<CRSR>↓	Moves cursor down one line
↑, <CRSR>↑	Moves cursor up one line
<CRSR>→	Moves cursor one space to the right
←, <CRSR>←	Moves cursor one space to the left
<RETURN>	Moves cursor to the beginning of the next line
<HOME>	Moves cursor to the upper left corner
*	Fills pixel, moves one space to the right
+	Fills pixel
<SPACE>	Erases pixel, moves one space right
<DEL>	Moves one space left and erases pixel
–	Erases pixel
<CLR>	Erases all pixels, homes cursor
X	Toggles BIGX/SMALLX
Y	Toggles BIGY/SMALLY
<CTRL> 9	Toggles shape reverse
<CTRL> C <RUN/STOP>	Defines shape, exits editor

Saving Shapes

The current set of seven sprite shapes can be saved in a permanent file. To do this, type SAVESHAPES (a procedure in the file SPRITES) and the name of the new shape file being created. For example, suppose you use the shape editor to create a panda for shape 1 and a penguin for shape 5. These new animal shapes (1, 5) and the original shapes (0, 2, 3, 4, 5 and 7), could be saved by typing in

```
SAVESHAPES "CRITTERS
```

The new shape file will be saved under the name CRITTERS. (Note: Any previous shape file called CRITTERS will be deleted.) When

```
READSHAPES "CRITTERS
```

is typed in, any sprites with sprite shape 1 will be assigned the panda shape, and any sprites with sprite shape 5 will be assigned the penguin shape.

Shape Numbers of the Built-In Shape Files

ANIMALS.SHAPES file

1 DINOSAUR
2 KANGAROO
3 BUG
4 DOLPHIN
5 HORSE
6 CAT
7 BUTTERFLY

SHAPES.SHAPES file

1 FRAME
2 BBALL (Big Ball)
3 SBALL (Small Ball)
4 SQUARE
5 TRIANGLE
6 HEART
7 BOX

VEHICLES.SHAPES file

1 TRUCK
2 CAR
3 BICYCLE
4 SUBMARINE
5 AIRPLANE
6 BOAT
7 BALLOON

ASSORTED.SHAPES file

1 TARGET
2 TRUCK
3 ROCKET
4 BALLOON
5 BOW
6 ARROW
7 MAN

The file RUNNER.SHAPES contains seven shapes that were designed for an animation sequence of a woman running.

Demonstration Programs

The Commodore 64 Logo package includes four demonstration programs. Here is a list of each of the demonstration programs with a brief description:

SPRITEDEMOS	Contains examples of the short procedures used throughout this chapter.
DINOSAURS	A dinosaur family goes out for a picnic.
SUBMARINE	A submarine travels through a colorful ocean.
RUNNER	An animated display of a woman running.

The remainder of this chapter describes how the procedures in DINOSAURS, SUBMARINE, and RUNNER work.

DINOSAURS

The procedures for this demonstration program are located in the file DINOSAURS. Type

```
READ "DINOSAURS
```

Since DINOSAURS is a self-starting file, the demonstration will run automatically.

The secret to a self-starting file is to make a variable named STARTUP. The value associated with the variable STARTUP must be a list. This list is run when the file is read in. The value of STARTUP here is [DEMO], so the procedure DEMO executes immediately.

Note that DEMO contains the instruction READ "ANIMALS. When ANIMALS.LOGO is read in as a result, its own STARTUP value is also executed, causing Logo to read in ANIMALS.SHAPES.

For a full explanation of self-starting files and the use of the STARTUP variables, see section 2.1 of the Reference Guide.

Pressing any key will stop the execution of the program at the end of a cycle and <CTRL> G will stop it immediately.

As DEMO runs, it reads in other files, hides unused sprites, and sets the shape and size of the others. Then it calls STROLLDEMO.

```
TO DEMO
  READ "SPRITES
  READ "ANIMALS
  EACH [0 4 5 6 7][HT]
  EACH [1 2 3][SETSHAPE 1 PU]
  TELL 3 BIGX BIGY
  TELL 2 SMALLX BIGY
  TELL 1 SMALLX SMALLY
  STROLLDEMO
END

TO STROLLDEMO
  IF RC? THEN SPLITSCREEN STOP ELSE STROLL
  STROLLDEMO
END
```

STROLLDEMO runs the STROLL procedure until a key is pressed. Note the use of the procedure EACH in both DEMO and STROLL. EACH is used to tell several sprites to do the same thing.

```
TO STROLL
  TELL 3 PENCOLOR 5 SETXY 60 21
  TELL 2 PENCOLOR 9 SETXY 100 21
  TELL 1 PENCOLOR 8 SETXY 130 0
  EACH [1 2 3][SETH 0 ST]
  SCENERY
  REPEAT 4 [TELL 3 CLUNK 1 TELL 2
            CLUNK 1 TELL 1 SKIP 3] SETX - 35
  TELL 3 EAT
  TELL 2 SETX - 65 EAT
  TELL 1 EAT
END
```

STROLL begins by giving the dinosaurs different colors and lining them up, one behind the other toward the right of the screen. Then SCENERY draws trees at the left of the screen. The two big dinosaurs move across the screen with the CLUNK procedure; the baby moves with the SKIP procedure. When

the dinosaurs reach the left of the screen they nibble on the tree branches with procedure EAT.

```
TO TREE :SIZE
  PD FD :SIZE TOP :SIZE BK :SIZE PU
END

TO TOP :SIZE
  IF :SIZE < 7 STOP
  LT 25
  FD :SIZE
  TOP :SIZE * 0.65
  BK :SIZE
  RT 50 FD :SIZE
  TOP :SIZE *0.65
  BK :SIZE
  LT 25
END
```

TREE and TOP (used by SCENERY) together draw trees. TREE draws the base of the tree and then calls TOP to draw the rest. TOP is simply a variation of the binary tree procedure you encountered in the Graphics chapter. (See also the solutions to Graphics Projects in the Appendix.)

Note that when STROLLDEMO is run, the baby dinosaur always passes in front of and never behind its parents. Sprites obey a priority system: sprite 0 has the highest priority and sprite 7 has the lowest priority. A sprite with a lower number will obscure one with a higher sprite number if the two sprites overlap. To see this more clearly, type

```
EACH [1 2 3][HOME]
```

SUBMARINE

In this demonstration program, a submarine spirals through a sea of colors. Type

```
READ "SUBMARINE
```

to load the file; since it has a variable named STARTUP, it will automatically begin to execute. Pressing any key will stop the execution of the program at the end of a cycle, and, of course, <CTRL> G will stop it immediately.

Some procedures in the file SUBMARINE are listed below.

```
TO DEMO
  READ "SPRITES
  READ "VEHICLES
  EACH [0 1 2 3 5 6 7][HT]
  TELL :SUBMARINE
  FULLSCREEN

SUBMARINEDEMO
END
```

DEMO reads the necessary files, hides the turtle and other sprites that aren't used, and runs SUBMARINEDEMO.

```
TO SUBMARINEDEMO
  SPIRAL
  IF RC? THEN SPLITSCREEN STOP ELSE SUBMARINEDEMO
END
```

SUBMARINEDEMO causes the procedure SPIRAL to run until a key is pressed.

```
TO SPIRAL
  BACKGROUND 6 PENCOLOR 7
  PU HT
  SETH 0 HOME
  SMALLX SMALLY ST
  EXPAND 2 100
  REPEAT 30 [PENCOLOR RANDOM 16 FD RANDOM 100 RT RANDOM 360
             RANDSIZE BACKGROUND RANDOM 16]
  SETXY 120 0
  SMALLX SMALLY SETH 0
  STAMPCIRCLE 120
  CONTRACT 100 2
  HOME
  REPEAT 20 [BACKGROUND RANDOM 16 PENCOLOR RANDOM 16]
  PENCOLOR 7
  CS REPEAT 20 [BACKGROUND RANDOM 16] BACKGROUND 6
END
```

SPIRAL begins by setting up the screen and the current sprite for the journey. The sprite begins its trip at the center of the screen. First, it travels in an outward spiral with EXPAND, then moves across a flashing screen as it randomly changes color, size, and direction. Next the sprite stamps a circular path on the screen with STAMPCIRCLE and spirals back to the center of the screen with CONTRACT. Finally, the graphics screen changes color forty times, in the middle of which the current sprite changes color.

RUNNER

The RUNNER demonstration program displays an animation of a woman running. Type

```
READ "RUNNER
```

to read the file. For very unusual results, try suppressing the STARTUP list (in this case DEMO) by holding down the Commodore key while RUNNER is loading, and try running the program with a different file of sprite shapes. Pressing any key will stop the execution of the program at the end of the cycle; <CTRL> G will stop it immediately. The procedures in RUNNER are listed below.

```
TO DEMO
  READ "SPRITES
  READSHAPES "RUNNER
  ASK 0 [HT]
  EACH [1 2 3 4 5 6 7][PC 1 PU]
  RUNDEMO
END

TO RUNDEMO
  LOCAL "STANCE
  BACKGROUND 4
  SHOW
  TELL 1 HOME BIGX BIGY SETHEADING 90 PENCOLOR 1
  WAIT 3
  MAKE "STANCE 8
  REPEAT 6 [MAKE "STANCE :STANCE - 1 TELL :STANCE HT]
```

(continued on next page)

```
  TELL 1
  SETSHAPE 2 WAIT 1 / 2
  SETSHAPE 3 RUNN
  SETSHAPE 2 WAIT 1 / 2
  SETSHAPE 1 WAIT 1
  SETSHAPE 7 WAIT 4
  SHOW
END
```

RUNDEMO begins by setting up the screen for the runner. First, procedure SHOW displays the shape sprites 1 through 7 at the right of the screen. Shapes 1, 2, and 3 show the runner preparing to run. Shapes 4, 5, and 6 show her running, and shape 7 shows her sitting down. Next the runner moves to the middle of the screen, and after a pause the other sprites are hidden. The runner assumes her starting position and then runs across the screen using procedure RUNN.

When a key is pressed, the runner sits down. After a pause, procedure SHOW is called, and the seated runner then joins sprites 1 through 6 at the right side of the screen.

```
TO RUNN
  IF RC? STOP
  REPEAT 11 [RUNLOOP 4]
  RUNN
END
```

RUNN causes the runner sprite to move across the screen (with RUNLOOP) until a key is pressed. When a key is pressed, RUNN stops and RUNDEMO continues.

```
TO RUNLOOP :SHAPE
  IF :SHAPE = 7 STOP
  FD 30
  SETSHAPE :SHAPE
  RUNLOOP :SHAPE + 1
END
```

RUNLOOP has the runner move forward while taking on the three running shapes.

```
TO WAIT :SECONDS
  REPEAT :SECONDS * 300 [ ]
END
```

WAIT causes a pause of an input number of seconds.

Suggestions for Ambitious Programmers

The examples given above only scratch the surface of potential sprite applications. Try combining sprite commands with music for a fresh approach.

The most obvious use for sprites is the creation of games. If you have a joystick or a set of paddles, these can be used to manipulate sprites more easily.

> See the Glossary listings for JOYSTICK, JOYBUTTON, PADDLE, and PADDLEBUTTON. You might also look at the file JOY on the Utilities Disk for more inspiration.

Above all, remember that sprites can do anything the turtle does. An interesting program might make use of the fact that sprites can STAMPCHAR, and that TB? outputs the value TRUE if the current sprite is in contact with any filled-in background pixels. (Note that TB? makes no distinction between lines drawn on the screen and characters stamped there.)

Finally: You may have noticed that TB? does not tell you when sprites are in contact with each other. Not to worry! The procedure TS? in the SPRITES file serves this purpose. You might even decide to combine the detecting functions of TS? and TB? in a procedure like

```
TO TOUCH?
  OP ANYOF TB? TS?
END
```

M MUSIC

Your Commodore 64 can play music and make sound effects. The Logo package includes programs which allow you to use these features, although it cannot play more than one melody line at a time. Not only can the pitch and duration be controlled, but the "envelope" can as well. The factors creating the envelope are what makes a note sound like a bell, a guitar, a harpsichord, etc.

The music chapter of this tutorial assumes that you are familiar with your Commodore 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 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. The workspace is cleared when the computer is turned off.

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 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 red 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. This contains procedures allowing you to make different sounds easily. MUSIC automatically reads in another of the music files called SOUND. The SOUND file contains the SOUND procedure which does the actual work of making the Commodore 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. We will use a slightly more general definition. Duration is the time from the beginning of one note to the beginning of the next. To start, let's use a procedure named SSH which imitates a snare drum sound. Type

```
SSH 5
SSH 10
SSH 10 SSH 10
SSH 10 SSH 5 SSH 10
```

If you don't hear any sounds, check to be sure the volume is turned up on the TV or monitor and that the cables are connected properly.

Using SSH only once, it is hard to tell how long it is without a second SSH. SSH 5 and SSH 10 sound alike when nothing follows them. When several SSH's are played one after another, the duration of each becomes obvious. However, typing SSH over and over gets tiresome. Another possibility is using the REPEAT command. For instance,

```
REPEAT 10 [SSH 10]
REPEAT 10 [SSH 5]
REPEAT 10 [SSH 5 SSH 10]
```

There is an even better way, the SSHER procedure. SSHER takes a list of durations to play. (A list for Logo is any list of items enclosed in square brackets.) Type

```
SSHER [10 10 10 10 10]
SSHER [5 5 5 5 5]
SSHER [5 10 5 10 5 10]
```

You have probably noticed that the numbers for durations are related in a special way. For instance, a duration of 10 is twice as long as one of 5 and similarly, 8 is twice as long as a duration of 4. Try typing

```
SSHER [8 8 8 8 4 4 4 4 4 4]
SSHER [6 6 6 6 6 6 12 12 12]
SSHER [8 8 4 4 8]
SSHER [12 12 6 6 12]
```

Notice that the last two examples sound the same except that the last one goes slower.

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 16 were used as a whole note, then 8 would be a half note, 4 a quarter note and 2 an eighth note. Of course, we could have used another number instead of 16 as our whole note. If we made 12 a whole note, then 6 would be a half note, and 3 a quarter note. Using 12 as the whole note instead of 16 will speed up the tempo.

You don't have to change the durations to be able to change the tempo. The procedure TEMPO allows you to do this. (The default value for TEMPO is 20.) Even though the durations remain the same, slowing down the tempo will make the durations longer, but each duration will still have the same relation to the other durations as it did before. Try

```
SSHER [8 8 4 4 8]
TEMPO 40
SSHER [8 8 4 4 8]
TEMPO 20
SSHER [8 8 4 4 8]
```

You can use SSHER to create any rhythm you want. For example, try

```
SSHER [10 5 5 10 5 5 10]
```

You can use the up arrow key 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 [SSHER [10 5 5 10 5 5 10]]
REPEAT 3 [SSHER [12 3 3 12 3 3 12]]
```

Note how the beats regroup if you play this pattern together several times instead of just once as you did before.

You can vary the duration within a wide range. Numbers higher than 100 can be used but a duration of 100 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 [0 1 2 3 4 5][15 15 15 15 15 15 15]
```

(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.

0	1	2	3	4	5	6	7	8	9	10	11	12
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 0, 2, 4, 5, 7, 9, 11, 12 that are paired with these letters 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 0 2 4 5 7 9 11 12
D Major Scale 2 4 6 7 9 11 13 14

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 [0 2 4 5 7  9 11 12][15 15 15 15 15 15 15 15 15]
PLAY [2 4 6 7 9 11 12 14][15 15 15 15 15 15 15 15 15]
```

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 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 [0 2 4 5 7  9 11 12]
SING [2 4 6 7 9 11 13 14]
```

Rests are another feature which you can use with PLAY or SING. Type in the letter R in place of one of the pitches, and no sound will be played for the duration corresponding to that pitch. Type

```
PLAY [0 0 R 0 R 0][10 10 10 10 10 10]
SING [0 0 R 0 R 0]
PLAY [0 0 0 0 R 7 R 4][10 5 5 10 10 10 10 10]
```

Procedures

Typing everything out each time, even if you do make use of the up arrow key and the REPEAT command, can get tiresome.

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 [0 1 2 3 4 5 6 7 8 9 10 11 12][15 15 15 15 15 15 15 15 15 15 15 15 15]
END

TO M
  PLAY [0 2 4 5 7 9 11 12][15 15 15 15 15 15 15 15 ]
END
```

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 any single digit number you type 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 [4 0 2 -5][10 10 10 10]
END
```

(Notice that you can use negative numbers as well. Minus 1 (−1) is a half step below 0, −2 is a whole step below 0 and so on, just as 1 and 2 are a half and a whole step above 0.)

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 doubling 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 [4 2 0 -5][10 10 10 20]
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 the durations are the same for all the blocks. Now see what happens if you change the rhythm

from [8 8 8 8 8 8 24]
to [12 4 12 4 12 4 24]

You can still recognize the original tune but this makes a varied version of it. It works because each pair of 8's is changed to 12 and 4, and both 8+8 and 12+4 add up to 16. Notice that it sounds like a waltz now instead of a march. Try reversing the 12 and the 4 so it is 4 12 4 12 4 12 24. 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 0, 2, 4, 5, 7, 9, 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.

Envelopes

To change the way the music sounds, you have several options. The first thing you can change is the wave form. There are four waveforms: triangle, saw-tooth, pulse, and noise. So far we have been using the triangle waveform. The WAVE procedure allows you to change the form. If you type WAVE 33 the sawtooth pattern will be chosen. Try typing BELL or STAR to hear the difference.

Waveform	Number
Triangle	17
Sawtooth	33
Pulse	65
Noise	129

The noise form you have already encountered. It produces the noise in SSH and SSHER.

ATTACK/DECAY

All four waveforms are affected by changes in the attack and decay variables. The attack is the rate at which the volume of the sound rises to its peak. The decay is the rate at which the volume of the sound falls from its peak. Both attack and decay can range from 0 to 15. However, except for long notes, an attack above 10 doesn't work too well because the volume rises so slowly. The decay likewise doesn't work too well below 3 because the volume of the sound will decrease too quickly. There are two procedures used to control attack and decay and not surprisingly they are named ATTACK and DECAY. Each takes a number as input. Try the following.

```
ATTACK 5 BELL1
ATTACK 10 BELL1
ATTACK 0 BELL1
DECAY 0 BELL1
DECAY 15 BELL1
DECAY 9 BELL1
```

Note that with a long decay (higher number), the sound lasts for the entire duration of the note. This means that you can hear the duration of the note without the note being in context with other notes.

SUSTAIN/RELEASE

The sound will decay from its peak to a mid-range volume. The level of that volume is called the sustain level. The rate at which the sound drops to the sustain level is the decay and the rate at which the sound drops from the sustain level at the end is the release. As with attack and decay, the sustain and release can vary from 0 to 15 and are controlled by procedures named SUSTAIN and RELEASE.

PULSE

Changing the pulse has an effect on waveform 65 only, the pulse (or square) waves. The procedure PULSE takes one input. This input can vary from 0 to 2048. If this variable is 0 or near 0, there is almost no sound. Type

```
WAVE 65
PULSE 0 BELL1
PULSE 100 BELL1
PULSE 500 BELL1
PULSE 1000 BELL1
PULSE 2000 BELL1
```

Note: Sometimes when you first use waveform 65, it makes little sound. Just type PULSE 100 or some other input where the input is not 0.

The SOUND Procedure

This section is only for those who want to know more about the low level procedure SOUND which actually generates the sounds. It is not something which you need to know to be able to use the music capabilities.

SOUND takes five inputs in the following order: pitch, duration, attack/decay, sustain/release, and waveform. The PLAY procedure (which uses the NOTE procedure) will modify the pitch you give it so that SOUND can use it. The numbers for the pitches we have been using, when they are converted into numbers which SOUND can use, are in the 3,000 to 20,000 range. Duration here uses the same numbers as PLAY and SSHER do. The two values attack/decay and sustain/release both are calculated the same way. The decay ranges from 0 to 15 and the attack is based on multiples of 16.

Number Attack or Sustain Uses	Number Decay or Release Uses	Number SOUND Uses
0	0	0
0	1	1
0	5	5
1	0	16
1	1	17
2	0	32
2	2	34

The last variable used by SOUND (WAVEFORM) is the same as you have used in PLAY.

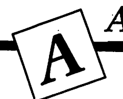
To get a feel for the variety of sounds your Commodore 64 can make, try the following procedure.

```
TO RANDNOISE
  SOUND RANDOM 32000 RANDOM 30 RANDOM 255 RANDOM 255 ITEM(1 + RANDOM
3)[17 33 129]
  RANDNOISE
END
```

You can modify this so it does different ranges of the variables, or you might add the waveform 65. Another idea is to use local variables which print out so you can see the configuration for a particular sound you like.

As with the sprites, you could add a line so the procedure stops or pauses if you hit a key. If you type <CTRL>G to stop a procedure, sometimes the sound is left on. Just play another note (use PLAY or SOUND) and it will stop.

This chapter serves only as an introduction to the musical capabilities of your Commodore 64. If you would like to know more, see the chapter in the Commodore 64 Programmer's Reference Guide named "Programming Sound and Music on your Commodore 64", or the chapter "Creating Sound" in the Commodore 64 User's Guide.



APPENDIX

ERROR MESSAGES

Error messages are Logo's way of trying to help the user find errors. Common errors are misspellings and wrong usage. This list of error messages has two parts. In both parts, capital letters indicate the unchanging part that Logo displays to you; the parentheses indicate what will vary depending on the circumstances. Both parts of the list are alphabetical according to the first unchanging word.

The first part of the list includes those messages which will 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.

PART I

(procedure) DIDN'T OUTPUT

Example:

```
FORWARD SQUARE 5
SQUARE DIDN'T OUTPUT
```

```
SQUARE FD 100
FD DIDN'T OUTPUT
```

This occurs when a procedure or primitive command that needs an input is followed by another procedure or primitive that does not output anything. Logo will run the second command assuming it will output something suitable as input to the first command. If the second command does not output anything, then the error message results. In the examples above, it looks like a number was left out between the two commands and thereby caused the error.

This error generally 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 with the wrong type of data. Here Logo is trying to multiply a name (specified by ") instead of a value (specified by :).

(primitive) DOESN'T LIKE (data) AS INPUT.
IT EXPECTS TRUE OR FALSE

Example:

IF :X - 10 THEN PRINT [YOU CRASHED!]
IF DOESN'T LIKE 2 AS INPUT. IT EXPECTS TRUE OR FALSE

It looks like :X = 10 was intended. . .

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. For instance, IF :X THEN. . . instead of IF :X=3 THEN. . .

(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 which 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 first input obtained by subtraction.

This error can also occur because parentheses are allowed as part of a word in Logo. If there is no space between a word and a closing parenthesis, Logo will treat the closing parenthesis as part of the word and assume that the closing parenthesis is missing. Therefore, Logo complains, as it will with the following example.

(PRINT "HELLO "HANDSOME)

(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, LOCAL, 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.

DISK ERROR

Example:

PRINTER
CATALOG
[ ] — DISK ERROR

The Logo Language Disk files cannot be listed with CATALOG. You will also get this message when the disk is damaged.

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 NOT FOUND

Example:

```
READ "AMINAL
AMINAL — FILE NOT FOUND
```

AMINAL is not a file on the disk in the disk drive. Perhaps you simply misspelled ANIMAL. Check your spelling and type CATALOG to see what IS on 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; for instance, (SETXY 3) is missing the Y coordinate.

NO STORAGE LEFT!

You have used up all the storage. Erase some unnecessary procedures. Before you erase any procedures, check to see if you have a recursive call that is not at the end of a procedure, and which does not have a way to stop. If so, fix this bug; there may not actually be a storage problem.

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 less than 10 ** (−38).

PROCEDURE NESTING TOO DEEP

You have exceeded the limit for nesting procedures (which is over 200).

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.

Sometimes you will get this message when you don't expect it. It usually means that you have left out an OUTPUT statement in some procedure, possibly on the last line of a recursive procedure.

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.

THE DISK IS FULL

Example:

```
SAVE "NEWPROC
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;
2. You can use another disk which is not full;
3. You can trim the amount you are saving by erasing procedures from your workspace. This may not work; use it only as a last resort if you can't find any blank disks.

THE DISK IS WRITE PROTECTED

You tried to write on a write-protected disk. This might mean you forgot to put your own disk in the disk drive, and have left the Logo Language disk in place.

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 DISK IN THE DISK DRIVE

Example:

```
READ "MYFILE
```

This error message is self-explanatory. You may also get this message in situations where there is a disk in the drive but the door is open, or perhaps the disk drive may be turned off.

If you get this error message when there is a disk in a turned-on drive and the door is closed, then the disk drive has become confused. Open the door, remove the disk, and then put the disk back into the drive and start the process over again. Consult your Commodore disk drive manual if the problem continues.

THERE IS NO LABEL (whatever you used)

Example:

THERE IS NO LABEL QUAD:, IN LINE

You have used GO to 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. If you are using GO to make the result of a long IF statement appear on a separate line, then try using the TEST/IFTRUE combination instead.

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 (whatever you typed)

Example:

THERE IS NO PROCEDURE NAMED FD100

When Logo does not recognize a primitive, it looks for a procedure name. This error message is usually caused by a typographical mistake, or forgetting to read in a file.

THERE'S NOTHING TO SAVE

Example:

```
SAVE "MYSTUFF
THERE'S NOTHING TO SAVE
```

There are no procedures or global variables in the workspace to save. Logo would rather tell you now than have you be disappointed when you read the (empty) file back in from the disk.

TOO MANY PROCEDURE INPUTS

You have exceeded the limit on procedure inputs (which is over 100). This will be exceedingly rare.

TOO MUCH INSIDE PARENTHESES

Logo uses this when it cannot figure out some parenthesized expression. Interior parentheses may be incorrectly placed.

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.

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 OUTPUT, PRINT, FORWARD, etc. This corresponds to RESULT: in immediate mode. Add the missing instruction (possibly PRINT in the example) to the line. As in the RESULT: message previously described, the missing OUTPUT statement may be in a subprocedure of the one in which the error is reported.

EDIT MODE: USE OF CONTROL CHARACTERS AND CURSOR KEYS FOR EASE IN EDITING

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 vertical arrow keys move the cursor up and down from line to line.
<CTRL> N	NEXT: Moves the cursor down to the next line. Same as down arrow key.
<CTRL> P	PREVIOUS: Moves the cursor up to the previous line. Same as up arrow key.
<CTRL> A	Moves the cursor to the beginning of the line.
<CTRL> L	End of Line: 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.
<HOME>	Scrolls the text so that the line with the cursor is in the middle of the screen, if there is more than one page of text. 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> 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> K	KILL: Deletes from the cursor to the end of the line. If the cursor is at the beginning of the line, <CTRL> K kills the whole line, leaving a blank line. An additional <CTRL> K will remove the blank line.

Leaving EDIT Mode

<CTRL> C or <RUN/STOP>	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.

GRAPHICS CHAPTER

Turtle Driving Projects

1. through 4. Screen size:

Hint: type key <f5> to see when the whole turtle goes off the top or bottom and appears at the other edge of the screen. Type <f1> 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:

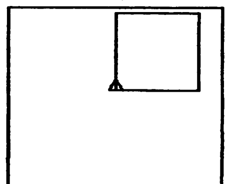
To get to the first corner: use half the distance across (from 2) to get to the edge, and half the distance down to get to the bottom. Write down this list of instructions in case you do not get the true diagonal on the first try. Then aim the turtle at the opposite corner.

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>

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:

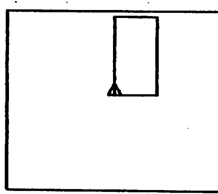
1) FD 100 RT 90 FD 50 RT 90 <RETURN>
   <CTRL> P <RETURN>

(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>

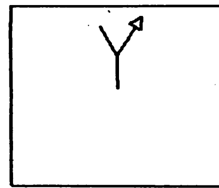


Rectangle

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 100
  RT 90
  BK 100
  PD
  FULLSCREEN
END
```

gives the same final result as

```
TO SETUP
  DRAW
  LT 90
  FD 100
  RT 90
  BK 100
  CS
  FULLSCREEN
END
```

Use PU / PD to avoid having to get rid of the track.

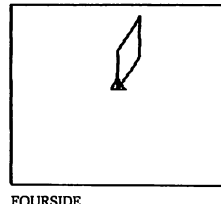
2. Design with MOVE repeated:

```
TO MOVE
  FD 100
  RT 15
  BK 80
END
```

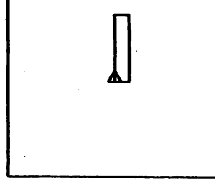
```
TO MOVEIT
  REPEAT 24 [MOVE]
END
```

3. A four-sided figure:

```
TO FOURSIDE
  REPEAT 2 [FD 60 RT 30 FD 60 RT 150]
END
```



FOURSIDE



RECT1

4. Rectangles:

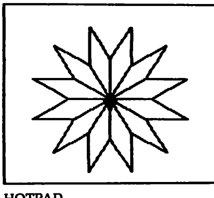
```
TO RECT
  REPEAT 2 [FD 100 RT 90 FD 50 RT 90]
END

TO RECT1
  REPEAT 2 [FD 110 RT 90 FD 10 RT 90]
END
```

5. Setup and a rectangle:

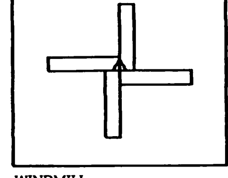
```
SETUP          SETUP
RECT           RECT1
```

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

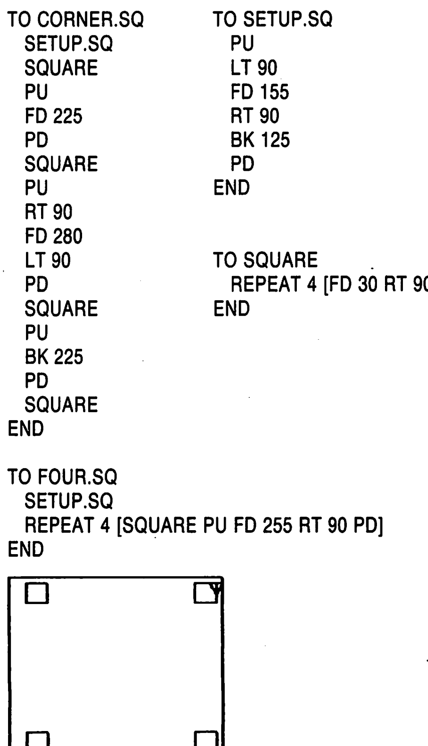
1. A square in each corner of the screen:

```
TO CORNER.SQ
  SETUP.SQ
  SQUARE
  PU
  FD 225
  PD
  SQUARE
  PU
  RT 90
  FD 280
  LT 90
  PD
  SQUARE
  PU
  BK 225
  PD
  SQUARE
END
```

```
TO SETUP.SQ
  PU
  LT 90
  FD 155
  RT 90
  BK 125
  PD
END
```

```
TO SQUARE
  REPEAT 4 [FD 30 RT 90]
END
```

```
TO FOUR.SQ
  SETUP.SQ
  REPEAT 4 [SQUARE PU FD 255 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,

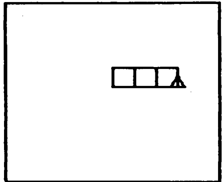
```
TO ROW.SQUARE
  REPEAT 3 [SQUARE RT 90 FD 30 LT 90]
END
```

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
```

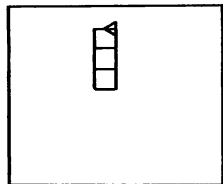
Lengthening the distance forward would produce a row of separated squares.



ROW.SQ

3. Tower of squares:

```
TO SQUARE.TOWER          TO SQUARE.TOWER.LEFT
  LT 90                    RT 90
  ROW.SQUARE               ROW.SQUARE.LEFT
END                      END
```



SQUARE.TOWER

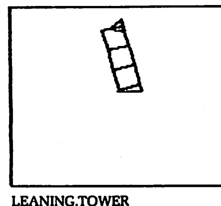
4. A leaning tower:

```
TO LEANING.TOWER     TO BASE
  BASE                 RT 90
  SQUARE.TOWER         FD 30
END                    LT 105
                       FD 10
                     END
```

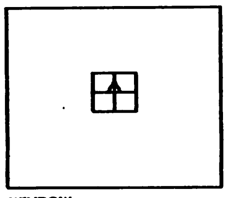
Discover the distances in a procedure like BASE by trying different ones.

5. A window with four panes:

```
TO WINDOW
  REPEAT 4 [SQUARE LT 90]
END
```



LEANING.TOWER



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

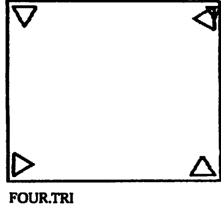
```
TO TRI
  REPEAT 3 [FD 30 RT 120]
END
```

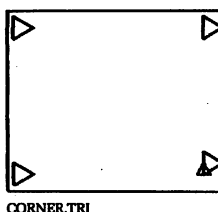
8. 1 – 4 using triangles:

(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.SQ
  REPEAT 4 [TRI PU FD 255 RT 90 PD]
END
```

```
TO CORNER.TRI
  SETUP.SQ
  TRI
  PU
  FD 225
  PD
  TRI
  PU
  RT 90
  FD 280
  LT 90
  PD
  TRI
  PU
  BK 225
  PD
  TRI
END
```

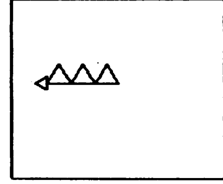

FOUR.TRI


CORNER.TRI

Notice that the two procedures produce different results with triangles. The orientation of a triangle makes a difference. Notice also that the name SETUP.SQ is inappropriate and should have been given a more general or accurate name.

(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

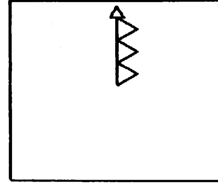
In the first, the turtle is heading in the direction of the first side when it starts out. In the second, it has to turn each time to head in the right direction. Which is easier to understand? Try to make your procedures as simple as possible.

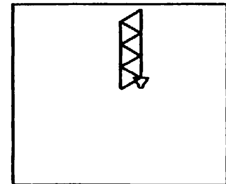
(3) A tower of triangles:
There are several choices:

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
```

```
TO TRI.TOWER2
  RT 90
  ROW.TRI
  RT 60
  FD 30
  RT 120
  RT 90
  ROW.TRI
END
```


TRI.TOWER1

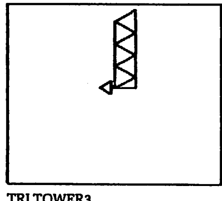

TRI.TOWER2

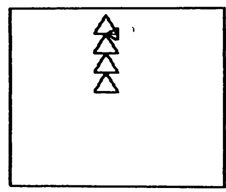
TO TRI.TOWER3: Add to TRI.TOWER2 (before END)

```
FD 15
RT 90
FD 30
```

FD 30 is slightly too long. Adjust it by trial, or by calculation if you know how.

```
TO TRI.TOWER4
  LT 90
  REPEAT 3 [TRI RT 60 FD 30 LT 60 BK 15]
END
```


TRI.TOWER3


TRI.TOWER4

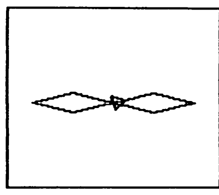
Note that the turtle draws the triangle, turns and moves to the top, then turns again and backs into position to draw the next one.

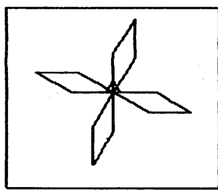
(4) A leaning tower of triangles:
Turn turtle and draw either ROW.TRI,
TRI.TOWER or TRI.TOWER2.

9. Designs using FOURSIDE:

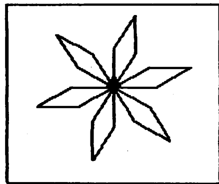
NOTE: These designs were named after they were drawn.

```
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
```


BOW.TIE


PINWHEEL

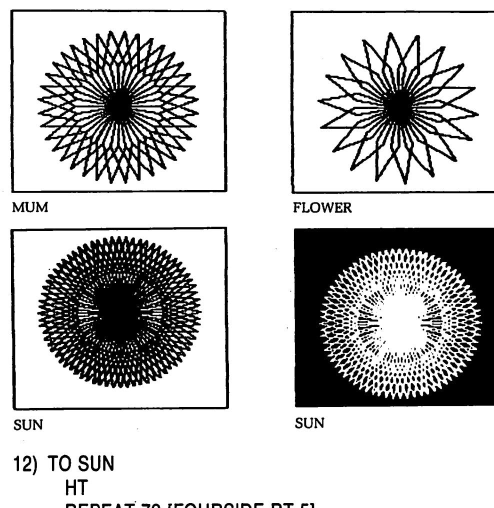
```
5)  TO FIVE
      REPEAT 5 [FOURSIDE RT 72]
    END

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

11) TO MUM
      HT
      REPEAT 36 [FOURSIDE RT 10]
    END
```



MUM FLOWER

SUN SUN

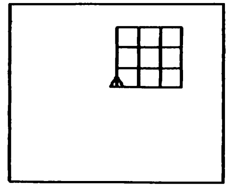
```
12) TO SUN
      HT
      REPEAT 72 [FOURSIDE RT 5]
    END
```

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.

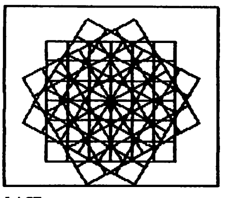
Progressively more complicated designs:

Using ROW.SQ:

```
1) TO NINE
     HT
     REPEAT 4 [ROW.SQUARE LT 90]
   END

2) TO LACE
     HT
     REPEAT 12 [NINE RT 30]
   END
```



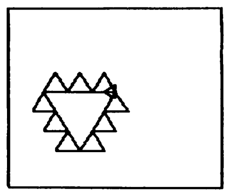
NINE



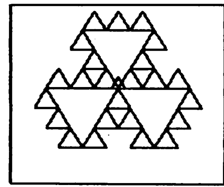
LACE

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
```



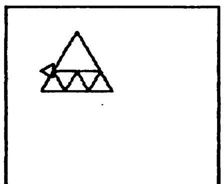
JAG.TRI



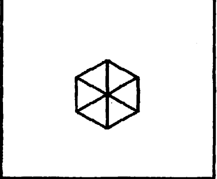
JAG3

10. A window with 6 triangular panes:

```
TO TRI.WINDOW              TO TRI.WINDOW2
  ROW.TRI                    REPEAT 6 [TRI RT 60]
  RT 120                   END
  FD 90
  REPEAT 2 [RT 120 FD 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 RT 120]
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)

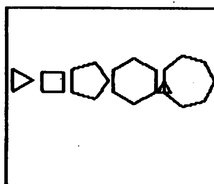
```
TO PENTA
  REPEAT 5 [FD 30 RT 360/5]
END
```

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

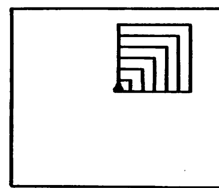
6) A pentadecagon (15 sides)

```
TO FIFTEEN
  REPEAT 15 [FD 30 RT 360/15]
END
```

Projects: Sizable Shapes

1. SQUARE4 to draw squares of various sizes:

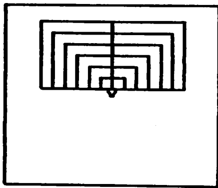
```
TO SQUARE4
  SQV 10
  SQV 20
  SQV 30
  SQV 40
END

TO SQV :LENGTH
  REPEAT 4 [FD :LENGTH RT 90]
END
```



SQUARE4

2. Another set of squares beside the first:

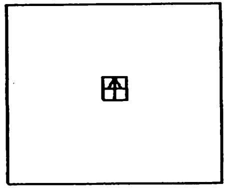
```
TO TWO.SQUARES
  SQUARE4
  LT 90
  SQUARE4
END
```



TWO.SQUARES

3. A procedure using a specific size square:

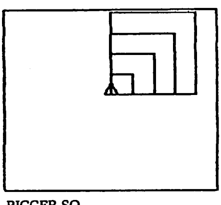
```
TO WINDOW1
  REPEAT 4 [SQV 30 RT 90]
END
```



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
```

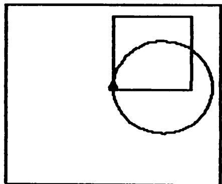


BIGGER.SQ

Projects with Regular Polygons

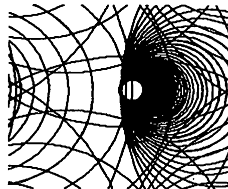
1. POLY 4 100 and POLY 100 4

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



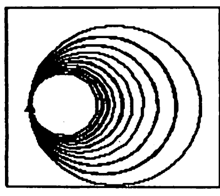
POLY 4 100 and POLY 100 4

2. POLY with the same :LEN and varying :TURNS



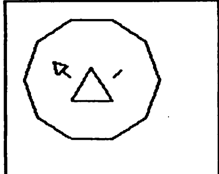
POLY: Same :LEN, varying :TURNS

3. POLY with the same:TURNS and varying :LEN



POLY: Same :TURNS, varying :LEN

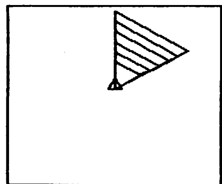
4. POLY twice, with different :TURNS



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

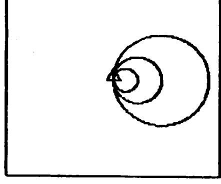
There is no largest number . . . The figure becomes a rough circle at 15, and after that, larger numbers increase the exactness of the curve, but after a while there is no more visible improvement and the only effect is to make the turtle go more slowly and the circle to get larger (with the same length 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.

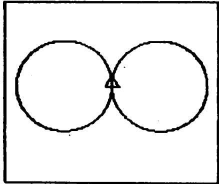
```
DRAW
REPEAT 360 [FD 1    RT 1]
REPEAT 360 [FD 2    RT 1]
REPEAT 180 [FD 1    RT 2]
```



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:

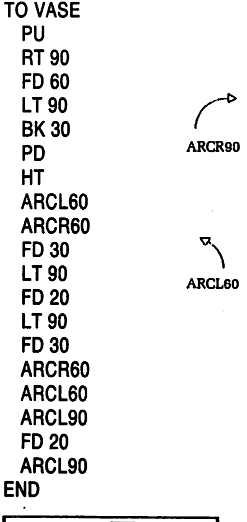
```
REPEAT 360/4 [FD 2    RT 1]
```

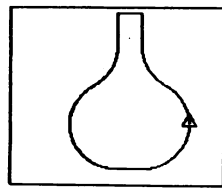
6. Sixth-of-a-circle arc to the left and right (make them into procedures and call them ARCR60 and ARCL60):

```
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
```



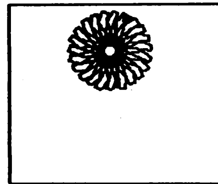

VASE

8. You can combine POLY and one of the ARC procedures to get procedures ARCR and ARCL which take variable step size and degrees. Also see the description at the end of this Procedures section of the Appendix about developing arcs. There are arc procedures on the Utilities disk as well, which are described in the Utilities disk section of the Appendix.

Projects: Simple Recursion

1. A recursive procedure that draws a little figure, then calls itself:

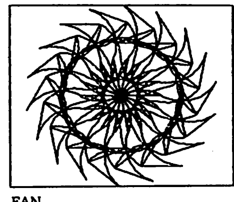
```
TO FIGURE
  FD 60 RT 49 FD 10 RT 80 FD 5 RT 90
  FIGURE
END
```



FIGURE

2. A recursive procedure that uses arcs and lines (The arc procedures used here are developed at the end of this section of the Appendix):

```
TO FAN
  PU
  RT 20
  PD
  REPEAT 3 [RARC 50 60 LARC 50 90 BK 50 LT 90]
  FAN
END
```

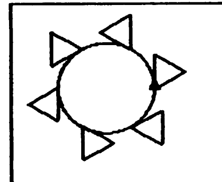


FAN

FAN

3. A recursive procedure using a triangle:

```
TO MILLWHEEL
  TRI
  ARCL60
  MILLWHEEL
END
```

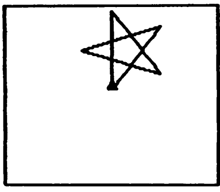


MILLWHEEL

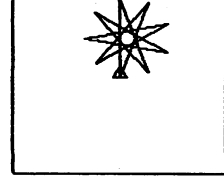
4. Stars:

```
TO STAR
  FD 75 RT 144
  STAR
END
```

```
TO STAR9
  FD 75 RT 160
  STAR9
END
```



STAR



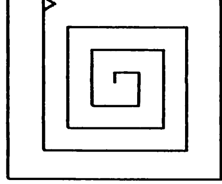
STAR9

Projects: Changing Inputs

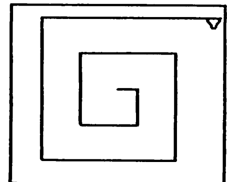
1. SQUARE with a larger increment:

```
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



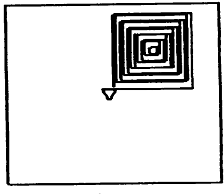
SQUARE2 With +25

SQUARE with a smaller increment:

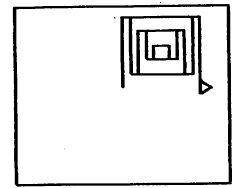
```
TO SQUARE3 :LENGTH
  FD :LENGTH RT 90
  SQUARE3 :LENGTH + 1
END

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

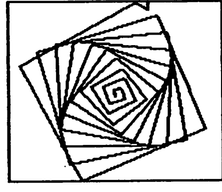


SQUARE6 With−10

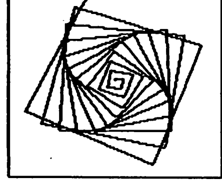
Note what happens when the length of the side becomes very small and then negative. . .

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



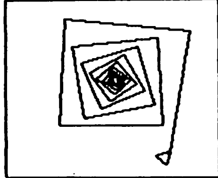
SQUARE8 With RT 87

Now you begin to see some of the power of changing the input in a recursive procedure.

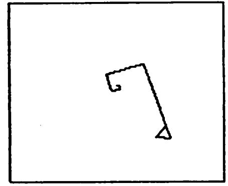
4. SQUARE with the input changed by multiplication:

```
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

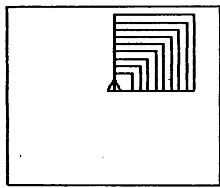


SQUARE10 With * 2

5. SQUARE, SQUARE1, . . . SQUARE10 in both WRAP and NOWRAP mode.

6. Using a SQUARE procedure with variable input (such as SQV) in a procedure that draws successively larger squares.

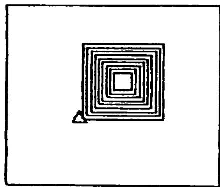
```
TO LARGER.SQUARES :LENGTH
  SQV :LENGTH
  LARGER.SQUARES :LENGTH + 10
END

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
```



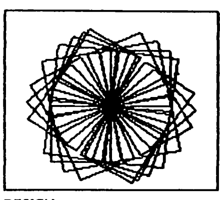
LARGER.SQUARES (Centered)

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

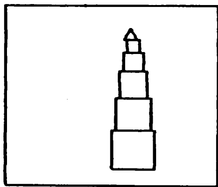
2. A tower of increasingly smaller squares, number of squares chosen when procedure is run, with a setup procedure to start lower on the screen (Type SET.TOWER, then type TOWER.OF.SQUARES 5 55):

```
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
```

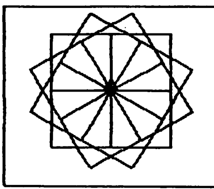
After drawing each square, the turtle moves up the side of the square just drawn, turns, moves half the size of the increment (so the next square is centered), and turns again, ready to begin the next square.

3. DESIGN with a variable turn:

```
TO DESIGN1 :TIMES :LENGTH :TURN
  IF :LENGTH < 0 THEN STOP
  IF :TIMES < 1 THEN STOP
  SQUARE :LENGTH RT :TURN
  DESIGN1 :LENGTH :TIMES - 1 :TURN
END
```



TOWER.OF.SQUARES



DESIGN1

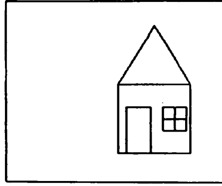
Recursion Projects

1. Successively smaller houses:

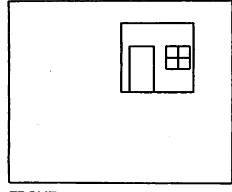
Begin by designing one house with a variable for a unit of size, to be determined later. The parts will require some instructions between them for positioning, but that too can wait. For a start, just describe what will be in the picture.

```
TO HOUSE :SIZE
  FRONT :SIZE
  ROOF :SIZE
END
```

```
TO FRONT :SIZE
  WALLS :SIZE
  DOOR :SIZE
  WINDOW :SIZE
END
```



HOUSE



FRONT

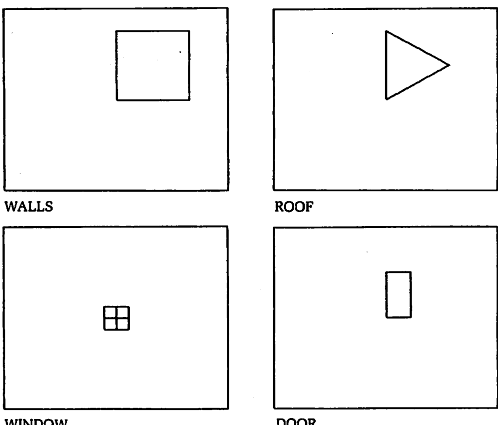
Now is the time to decide the size relationship of the components. Test each of these to be sure it works correctly before you begin on the interfacing instructions that make the parts go together.

```
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
```




```
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
```

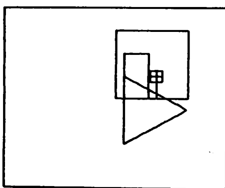
Now comes the fitting together of the parts.

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.

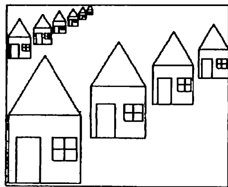
Now all that remains is to create the procedure HOUSES which will run the other procedures when you type HOUSES.

```
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
```



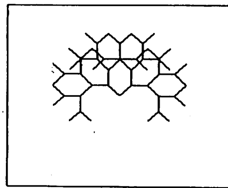
HOUSES .75

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:

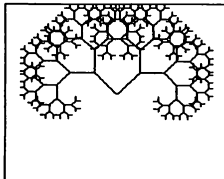
```
TO TREE :LENGTH :FORKS
  IF :FORKS = 0 STOP
  RT 45
  FD :LENGTH
  TREE :LENGTH :FORKS - 1
  BK :LENGTH
  LT 90
  FD :LENGTH
  TREE :LENGTH :FORKS -1
  BK :LENGTH
  RT 45
END
```

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

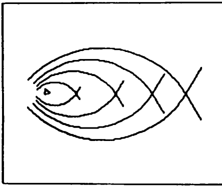
Each of these makes a different design. To alter it even more, consider making it with one side different from the other, perhaps doubling the length of the branches or changing the turn.

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. These procedures utilize procedures from the ARCS file on the Utilities disk.

```
TO FISH :SIZE
  RT 30
  PU
  ARCR :SIZE * 3 10
  PD
  ARCR :SIZE * 3 110
  TAIL :SIZE
  ARCR :SIZE * 3 110
END

TO SETUP.FISH
  PU
  LT 90
  FD 100
  RT 90
  PD
END

TO FISH.IN.FISH :SIZE
  IF :SIZE > 50 STOP
  FISH :SIZE
  PU
  ARCR :SIZE * 3 10
  LT 60
  FD :SIZE/3
  RT 90
  FISH.IN.FISH :SIZE + 10
END

TO EYE
  PU
  RT 90
  FD 70
  LT 90
  FD 15
  LT 90
  FD 5
END

TO FISHES
  SETUP.FISH
  FISH.IN.FISH 20
  EYE
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.



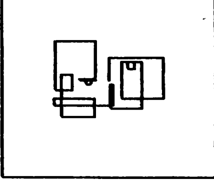
FISHES

Projects Using Random

1. SQUARE3 using FD RANDOM 100 in SQUARESIDE:

```
TO SQUARESIDE
  FD RANDOM 100
  RT 90
END
```

```
TO SQUARE3
  SQUARESIDE
  SQUARE3
END
```

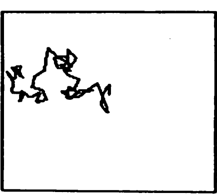


2. REPEAT using a random turn between 0 and 360:

```
REPEAT 50 [FD 20 RT RANDOM 360]
```

3. A recursive procedure using a random turn between 60 and 150:

```
TO WORM
  FD 20
  RT 60 + RANDOM 90
  WORM
END
```



WORM

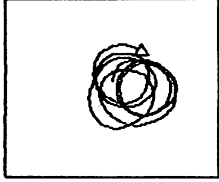
To specify a range BETWEEN two numbers, add the beginning number of the range (here 60) to the amount of the range (90, for a range of from 60 to 150). The computer will always choose a number within the amount of the range (here 90) and add it to the beginning number (here 60), to obtain a number within the specified range (here 60 + 0 to 60 + 90, or 60 to 150).

4. Other ranges of turn:

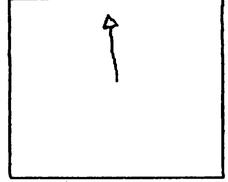
```
TO WANDER
  FD 2
  RT RANDOM 10
  WANDER
END
```

```
TO WIGGLE
  FD 5
  RT -10 + RANDOM 20
  WIGGLE
END
```

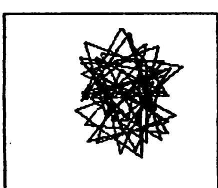
```
TO VARY
  FD 30
  RT 120 + RANDOM 30
  VARY
END
```



WANDER



WIGGLE



VARY

Mascots: Elephant, Rabbit, Snail

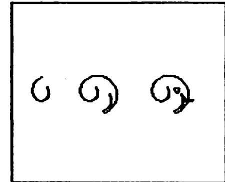
No lions and tigers and bears, but an elephant (that's for remembrance), a rabbit (denoting speed and ingenuity), and a snail (go slow. . . slow. . . slow).

The arc procedures used here are on the Utilities Disk. The procedures in the upcoming arc development section, which have slightly different names, could also have been used.

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
  CIRCLER 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
  CIRCLEL 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
```

Evolving the Rabbit

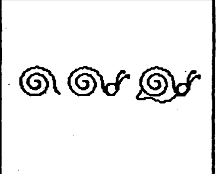
```
TO EARS
  EAR
  RT 150
  EAR
END
```

```
TO EAR
  ARCR 30 60
  RT 120
  ARCR 30 60
END
```

```
TO HEAD
  EARS
  ARCL 6 540
END
```

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
  ARCR 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
  DRAW
  FRAME
  H
  RUN SENTENCE :PROCEDURE [ ]
  TURTLE
  SAVEPICT :PROCEDURE
END
```

```
TO H
  PU
  HOME
  PD
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
```

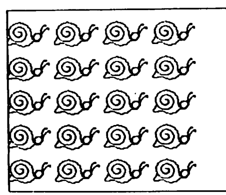
STORE clears the screen, draws the frame, moves the turtle to the HOME position, then runs the procedure TOWN. The SENTENCE :PROCEDURE [ ] makes a list out of the procedure name, so it can be RUN by another procedure. It turns the command into RUN [TOWN]. The procedure TURTLE draws a little turtle, since 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
```

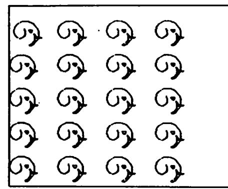


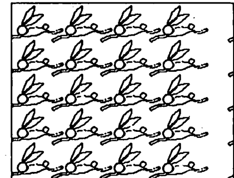
DROVE of Snails

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 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 a 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 1/6th 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

```
(DEGREES/360)*(2 * 3.14159 * :RADIUS)
```

The arc procedure would be

Type as one line

```
TO RARC :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 RARC :RAD :DEG
  REPEAT .0174532 * :DEG * :RAD
    [FD 1 RT 57.295827 / :RAD]
END
```

The circle procedure becomes

```
TO RCIRCLE :RADIUS
  RARC :RADIUS 360
END
```

LCIRCLE would use LARC, the same as RARC with LT substituted for RT. If you wanted to be silly, you could write

```
TO LARC :RADIUS :DEGREES
  MAKE "RADIUS :RADIUS * (-1)
  RARC :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):

```
TO RARC :RADIUS :DEG :STEP
  REPEAT (.0174532*:DEG*:RADIUS)/:STEP
    [FD :STEP RT (57.295827 * :STEP) /:RADIUS]
END
```

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-keystroke 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 9, sets the color accordingly.

PENCOLOR can take a number from 0 to 15 as an input, but for now we'll restrict ourselves to the range 0-9, since RC reads only one character.

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
     etc.
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, we can just set the PENCOLOR to whatever CHTR happens to be.

```
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
 PC :CHTR
END

TO SETPENCOLOR :CHTR
 IF NUMBER? :CHTR THEN 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 9, 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 techniques.

```
TO SETCOLOR :WHICHCOLOR :CHTR
 IF NOT NUMBER? :CHTR 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 slew 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 this 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 DRAWSTATE 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 DRAWSTATE THEN PU ELSE PD
END
```

The first element of the list that DRAWSTATE outputs tells whether the turtle's pen is up or down. If it is down (if FIRST DRAWSTATE is TRUE) TOGGLEPEN puts it up; otherwise it puts the pen 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 DRAWSTATE 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 DRAWSTATE would look like this:

```
TO TOGGLE.SHOWN
 IF FIRST BUTFIRST DRAWSTATE HT ELSE ST
END
```

See DRAWSTATE, 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 as well as clearing 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 a 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 ITEM :N "ABCDEFGHIJKLMNOPQRSTUVWXYZ
END
```

There is another way that doesn't involve "counting" with ITEM (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 THREENUM :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 ITEM 1 + RANDOM 7 PEOPLE
        ITEM 1 + RANDOM 6 ACTIONS
        ITEM 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 PERSON DIDWHAT
END

TO PERSON
 OP PICK 7 PEOPLE
END

TO DIDWHAT
 OP SE DIDIT PERSON
END

TO DIDIT
 OP PICK 6 ACTIONS
END

TO PICK :LISTSIZE :LIST
 OP ITEM 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, PERSON 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 could always be determined from LIST just by counting if we had a procedure that could count the elements in a list.

The primitive COUNT, which takes a list (or a word) as an input, does exactly this: 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 ITEM 1 + RANDOM (COUNT :LIST) :LIST
END
```

Because PICK now takes only one input — the actual list — PERSON and DIDIT need to be edited to use PICK properly.

```
TO PERSON                    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 PERSON 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 PERSON) become cleaner, better organized, and easier to understand and debug.

16. As with all procedures, there are lots of ways of designing them. Here are a few designs 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
```

A final note: The very first method is both "inelegant" and slower than the rest. It is quite unlikely that the speed difference will matter **at all** unless the procedure is used **very** frequently in a larger procedure.

Even a procedure, for example, that figures out where to insert a hyphen in a word at the end of a line, will have to test only a few of the letters in the word before discovering where the syllable boundaries are. It will probably not be noticeable which version of VOWEL? is being used.

The criteria for good programming depend on purpose. For the most part, elegance and readability are the best standards, and also make for compact and fast programs.

Occasionally, speed and elegance require different styles. How to choose? Go for elegance until speed becomes a noticeable problem; then switch criteria and speed up your programs.

17. 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.

There is a different and more elegant way of doing this using the LOCAL primitive.

```
TO YES?
 LOCAL "RESPONSE
 MAKE "RESPONSE REQUEST
 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
```

LOCAL allows you to use a local variable without it being an input listed in the title line. Here, the MAKE command which comes after LOCAL will change only the local value and not affect the global value (if any). For a more detailed discussion, see the section on LOCAL in the Computation chapter.

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! We must still write NEEDS.ES?

```
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
```

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. Why would you use the LOCAL command? (See the section in Computation.)

```
TO PRESENT :SUJET :VERBE
 LOCAL "ROOT
 LOCAL "END
 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
          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.

type as one line

```
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 data-base :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 adding 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
```

In its entirety 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"? The BUTLAST of the word NO would falsely match the keyword, and NO would be replaced by a noun with an O stuck at the end!

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 the-text-resulting-from-MADding-the-text-with-the-first-key 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 of all, 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.

40. Sorry. From here on in, you are on your own!

THE COMMODORE LOGO UTILITIES DISK

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.

To copy the Utilities Disk, use the COPY/ALL program if you have two disk drives, or the 1541 BACKUP program if you have one disk drive. These programs are on the disk which comes with the VIC 1541 disk drive. If you don't have this disk, ask your local dealer for a copy. If you have access to a Commodore 4040 dual-disk system, you may wish to use it to make a backup copy of the Utilities Disk.

Following are instructions for making a backup with one disk drive. This program is used when you have turned on the Commodore 64 and are in BASIC, not after you have loaded Logo.

1. Insert the VIC 1541 disk in the disk drive and type LOAD "1541 BACKUP",8 <RETURN>.

2. After the Commodore 64 responds with SEARCHING, LOADING, READY, type RUN and hit <RETURN>.

3. Type the letter B and hit <RETURN>.

4. The program will show the cursor after the words DISK DESTINATION. Type a disk name such as MYDISK and hit <RETURN>.

5. Type a 2 character ID that is different from the disk you are copying and hit <RETURN>.

6. The program will tell you to insert the destination disk in the drive. When you do this and hit <RETURN>, it will be formatted, erasing anything that was on it. Be careful!

7. The program will tell you to insert the source disk (the disk to be copied). When you hit <RETURN>, it will ask you to verify that the disk to be copied is in the drive. When you hit <RETURN> again, the contents of the disk will be loaded into the Commodore 64's memory.

8. It will tell you to insert the destination disk and hit <RETURN>. The contents of the Commodore 64's memory will be put on this new disk. These last two steps may need to be repeated several times if there is a lot on the disk. The program will tell you when to switch disks.

To Use the Utilities Disk Files

1. Start Logo using the Commodore Logo 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. If the listing is more than one page, the beginning will scroll by and disappear. To stop the scrolling, use <CTRL> W as a toggle to start and stop scrolling.

4. To read a file from the disk, after the ? prompt type

   READ "(file name 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
   TEACH DEFINED
   TEACH.BODY DEFINED
   ASK DEFINED
   ?
   ```

5. To use the file, type the name of a procedure, as described below. Some programs are self-starting; that is, they begin executing by themselves once you read the file. Most programs are not, 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 which are intended to be initialized and used later, 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.

Summary of Utilities Disk Files

The procedures in the Utilities Disk files provide excellent models for analysis. Much can be learned from their structures and techniques. Note in particular the brevity of Logo procedures, the constant use of subprocedures, and the use of procedure names which describe the procedure explicitly.

In addition to serving as models, the procedures in the files on the Utilities Disk perform a variety of functions.

Utilities:

ARCS	Collection of procedures for drawing arcs with variable radii. Another procedure for drawing an arc is developed in the Graphics Projects section of the Appendix.
B&W	Will reset the screen colors for a black and white TV screen.
BASE	Procedures for converting from one base to another, such as from decimal to binary.
CCHANGE CCHANGE.BIN CCHANGE.SRC	These three files are an example of the assembly language interface. The files allow you to change lines from one color to another after they have been drawn.
COLORS	Preset variable names for the colors.
FOR	A For-Next loop.
INSTANT	System of single letter Logo commands which makes Logo graphics available to non-readers, among others. INSTANT and its use are described at the end of the Graphics chapter. It also serves as an example of Logo programming style and of the use of RUN and DEFINE.
JOY	Procedures that allow you to draw on the screen with a joystick.
LOG	Procedures which print the log 10, natural log, and powers of e (2.718. . .) to three or four significant digits.

PLOTTER	Procedures for controlling the Commodore 4-color plotter.
PRINTPICT	A program written in BASIC for printing saved pictures on the printer. You must exit Logo in order to use this file.
STAMPER	Procedures for printing text on the graphics screen.
STAMPFD	Allows you to stamp the same character multiple times over a specified distance on the graphics screen.
TEACH	System of writing Logo procedures without using the editor.
TEXTEDIT	Procedures for using the Logo system as a text editor.
WHILE	Two procedures called WHILE and UNTIL which give you those programming functions.

Demonstration Programs:

ADVENTURE	An adventure game which demonstrates some advanced list processing.
ANIMAL	Game which adds your information about animals to its knowledge base. The structure and procedures of this program are good examples of advanced use of lists.
ANIMAL.INSPECTOR	Procedures for examining the ANIMAL knowledge base.
DYNATRACK	Game using principles of physics to simulate a ride around a frictionless race track.
GRAMMAR	Procedures which generate random sentences. There is also a set of procedures to compose postcards for you. Both use the same general approach and share some sub-procedures.

INSPI.PIC1 INSPI.PIC2	Picture on utilities disk to show use of READPICT and SAVEPICT.
PIG	Takes a sentence and converts it into pig Latin. A good example of word and list manipulation. Try modifying the file so it handles the letter Y properly.
SNOW	Draws a picture of a snowflake by modifying a triangle.
TET	Example of a simple recursive procedure which draws a complex design.

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 Assembler section in the Reference Guide describes Logo/Assembler interfacing.

Sprite Files:

SPRITES	Contains procedures for reading in sprite shape files, changing the size of sprites, telling several sprites what to do, and telling if the current sprite is in contact with an object on the screen.
SPRED	Contains a sprite editor allowing you to make your own sprite shapes.
ANIMALS	The file contains variables corresponding to the various sprite shapes in ANIMAL.SHAPES. Also automatically reads in the shapes in ANIMAL.SHAPES.
ANIMALS.SHAPES	Contains animal sprite shapes.

ASSORTED ASSORTED.SHAPES	These files contain names and shapes for miscellaneous sprite shapes. ASSORTED automatically reads in the shapes from ASSORTED.SHAPES.
RUNNER	A sprite demonstration showing a woman running. Uses the shapes in RUNNER.SHAPES.
RUNNER.SHAPES	Sprite shapes of a woman running.
SHAPES SHAPES.SHAPES	These files contain the names and shapes of a variety of geometric shapes. SHAPES will automatically read in the shapes from SHAPES.SHAPES.
VEHICLES VEHICLES.SHAPES	These files contain the names and shapes of different vehicles. VEHICLES will automatically read in the shapes from VEHICLES.SHAPES.
DINOSAURS	A demonstration program showing a family of dinosaurs walking across the screen.
SUBMARINE	A demo program showing a submarine moving around the screen and drawing.
SPRITEDEMOS	Contains procedures described in the chapter on Sprites.
VELOCITY	Allows you to give sprites velocities.

See the chapter on Sprites for more details about these files.

Music Files:

MUSIC	Procedures that play lists of notes and allow you to change the parameters of the sound such as attack, decay, sustain, and release.
SOUND	Procedure used to create the sound. This file is read in automatically by the MUSIC file.
TWINKLE	Contains procedures to play "Twinkle, Twinkle, Little Star" as an example of how to program music on the Commodore 64.

See the chapter on Music for more details.

Explanation of Utilities Disk Files

ARCS:
Variable Radius Arc and Circle Procedures

Additional variable radius arc procedures are developed in the Graphics Projects section of the Appendix.

To use the arc and circle procedures provided on the Utilities Disk, type the name with numbers for the inputs required. Examples:

ARCR 50 90	for a 90 degree (quarter circle) arc to the right with a radius of 50.
ARCR1 1 90	for a 90 degree (quarter circle) arc to the right with a radius of 360/(2 PI) or about 57.2.
CIRCLER 30	for a circle to the right with a radius of 30.

Substitute ARCL, ARCL1, and CIRCLEL for arcs and circles to the left.

ARC Procedures

ARCR :RADIUS :DEGREES
Procedure which draws an arc to the right with given :RADIUS and length :DEGREES. Uses ARCR1.

ARCL :RADIUS :DEGREES
Procedure which draws an arc to the left with given :RADIUS and length :DEGREES. Uses ARCL1.

CIRCLER :RADIUS
Procedure which draws a circle to the right with given :RADIUS. Uses ARCR.

CIRCLEL :RADIUS
Procedure which draws a circle to the left with given :RADIUS. Uses ARCL.

ARCR1 :SIZE :DEGREES
Procedure which draws an arc to the right with a radius equal to :SIZE × 360/(2 PI). Uses CORRECTARCR.

ARCL1 :SIZE :DEGREES
Procedure which draws an arc to the left with a radius equal to :SIZE × 360/(2 PI). Uses CORRECTARCL.

CORRECTARCR :SIZE :AMOUNT
Procedure which makes a small correction with each step of ARCR1.

CORRECTARCL :SIZE :AMOUNT
Procedure which makes a small correction with each step of ARCL1.

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]

B&W:
Resets the Color for a Black & White TV

The B&W file contains one procedure, also named B&W. When this file is read in, it runs automatically and the default colors are changed to black and white, which can be read more easily on a black and white TV screen. Be careful! This procedure also does a GOODBYE, so anything already in the workspace will be lost.

BASE:
For Converting from One Base to Another

The following procedures convert numbers from one base to another. Some of the procedures are for converting between frequently used bases.

BASE :BASE :NUM
Converts the base 10 number :NUM into base :BASE. For example

```
BASE 2 3
RESULT: 11
```

UNBASE :BASE :BNUM
Outputs the base 10 number that is equivalent to the number :BNUM in base :BASE. If the base is greater than 10, BNUM might have letters in it as well as numbers. In those cases where BNUM contains a letter, a double quote mark must precede the number. This method of dealing with numbers containing letters is used in the $ procedure below as well.

BINARY :NUM
Converts the base 10 number :NUM into a binary number.

HEX :NUM
Outputs the hexadecimal number that corresponds to the base 10 number :NUM.

$:NUM
Outputs the base 10 number corresponding to the hexadecimal number :NUM. If letters are used as part of the hex number, the value for :NUM must be preceded by a double quote mark.

```
$ 2000
RESULT: 8192
```

```
$ "F
RESULT: 15
```

E :ADDR
This is simply an abbreviation for .EXAMINE, so that typing only E and a number is necessary.

CCHANGE, CCHANGE.BIN, CCHANGE.SRC: Changing Colors on the Graphics Screen

These three files are an example of the assembly language interface. For more details, see the Reference Guide section on the assembler.

To use these files, type READ "CCHANGE. This file will load and then call the other two files. CCHANGE contains a procedure of the same name which takes two inputs. The first is the number of a color, and the second is the number of the color to which the first will be changed. For instance, CCHANGE 2 7, will change all red lines (2) to yellow lines (7).

COLORS: Variable Names for Colors

This file contains the names and values for the 16 colors, which can be seen when you type PO NAMES. The variable RED has the value 2, so the pen color could be set to red by typing PENCOLOR :RED.

FOR:
A FOR-NEXT Loop

The procedure FOR in the file FOR takes four inputs. The first is the name of the iteration variable, for example "I. The next two inputs are numbers giving the start and end of the iterative loop. The last input is the action to be done during the loop.

INSTANT:
Single Letter Logo Commands

See INSTANT section of the Graphics chapter in this tutorial. 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.

JOY:
For Drawing with the Joystick

To start the file, type PLAY after the file is read in.

PLAY
clears the screen and calls JOYPLAY.

JOYPLAY
Uses MOVE to check the setting of the joystick and move the turtle accordingly. Checks to see if the joybutton is on or off and raises or lowers the pen in response.

MOVE
Takes the output from the JOYSTICK command, turns it into a direction in degrees and moves the turtle in that direction.

LOG:
Procedures for Logarithms and Exponents

There are three main procedures, LOG, LOG10, and EXP, as well as several subprocedures. These procedures for calculating logs and exponents are accurate to 3 or 4 significant digits.

LOG10 :N
Outputs the base 10 logarithm of :N.

LOG :N
Outputs the natural log of :N.

EXP :N
Outputs e (2.718. . .) raised to the power of :N.

PLOTTER:
Controlling the Commodore 4-color Plotter

This file allows you to access a 4-color plotter through Logo. The names of the procedures resemble their Logo screen graphics equivalents, e.g. PFD for Plotter FD, PPU for Plotter PU, and so on. Type HELP for a listing of commands.

Some of the plotter commands have no exact Logo equivalents. When you type HELP, you are told to type SETUP. This command puts the plotter in graphics mode and centers the pen on the page. See the procedure POLYSPI for an example of how to create plotter graphics.

There is also a text mode on the plotter; type TXTMODE to leave graphics. (The command to return to graphics mode is GRMODE. Observe that GRMODE, unlike SETUP, does not center the pen.) To print words or lists, use DPRINT just as you would use PRINT. The command DPO differs from PO slightly: if a single procedure is to be printed out, its name must be preceded by a double quote.

Text may also be printed from graphics mode. Consult your plotter manual for the appropriate commands, and look at SETORIGIN to get an idea of how to write plotter commands into a Logo procedure.

PRINTPICT:
Hardcopy Printing of Saved Pictures

To use PRINTPICT you must have already saved your picture onto disk (see SAVEPICT in the Glossary). Turn the Commodore 64 on and place the Utilities Disk in the drive. Type

```
LOAD "PRINTPICT",8
RUN
```

You will be asked for the name of the picture you want to print. Remove the Utilities Disk from the drive and insert the disk with the picture files on it. In the case of a picture stored with SAVEPICT "SPIRAL, you would now type

```
SPIRAL
```

PRINTPICT will then print the picture described in file SPIRAL.PIC1. Note that SPIRAL.PIC2, which contains color information, is not used here.

STAMPER: Printing Text on the Graphics Screen

This file uses the command STAMPCHAR to put letters on the graphics screen. However, instead of only using one letter at a time, these procedures permit whole words and sentences to be typed.

STAMP :THING
Orients the turtle so that it will move from left to right. Uses STAMP1 to print a word or list. (Remember that a word must be preceded by a quote mark and a list must be enclosed in square brackets.) Resets the turtle's heading to what it was before STAMP ran when the word or list is finished printing. In addition, STAMP checks to see if the pen is up or down, and if necessary picks up the pen so it won't draw a line over the letters. It then resets the pen at the end of the procedure. If you want to print in some other direction than left to right, this program has to be modified. Change the line SETH 90 PU, or simply use STAMP1..

STAMP1 :THING
Prints a word or list on the screen. It moves FD 8 after stamping each letter. This works nicely if the turtle is moving horizontally, but if the turtle is moving diagonally or vertically, the distance has to be larger to prevent overwriting. The exact distance varies with the heading.

STAMPFD: Printing Strings of Characters on the Graphics Screen

STAMPFD :D :CHAR
This procedure needs two inputs. The first is the distance which you want the turtle to move. The second is the character which you want stamped. (Remember, a letter must be preceded by a double-quote.) STAMPFD will then calculate the number of characters it can print in the distance specified and stamp the character in a line on the screen.

STAMPBK :D :CHAR
Equivalent to using STAMPFD with a negative number for :D.

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 advantages of prompting the user for information and not clearing the turtle graphics screen.

To define the following procedure using TEACH

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

type what appears below in the usual computer font. What TEACH prints is in italics. If there are inputs, put them on the same line as the NAME, just as you would when writing a procedure.

```
? TEACH
NAME: COUNTDOWN :N
> IF :N = 0 STOP
> PRINT :N
> COUNTDOWN :N - 1
> END
COUNTDOWN DEFINED
?
```

To run COUNTDOWN, type

```
COUNTDOWN 5
```

The screen is not cleared when TEACH is used as it is when the editor is used. Instructions developed in IMMEDIATE mode can be copied into a procedure using TEACH. In GRAPHICS mode, TEXTSCREEN (key<f1>) will show the previous typing, possibly hidden by the picture. SPLITSCREEN (key<f3>) will return the picture and five lines of text.

WARNING: Reading a file from the disk DOES clear the screen. Therefore, read TEACH in from the disk before beginning to type any instructions that you might want to copy into a procedure using TEACH.

The TEACH system uses three procedures:

TEACH asks for and receives the name of the procedure and any inputs, and then passes the information on to TEACH.BODY.

TEACH.BODY, a recursive procedure, receives the lines of the procedure (after the prompt >), testing each for END. When END is received, TEACH.BODY completes the defining of the procedure and passes control back to TEACH, which announces the procedure defined.

ASK takes a prompt as input, prints it, and outputs REQUEST, unless it is an empty list, in which case it tries again.

TEXTEDIT:
How to Save, Read, Examine, and Print Text Files

The Logo system is set up to read and save files that contain Logo procedures. By modifying the system, one can use Logo to read and save text files and thus use the Logo editor purely as a text editor.

To use this file, the procedures must be in the workspace before any text is typed in the editor. So type

```
READ "TEXTEDIT
```

Normally to enter the editor you type TO followed by the name of the procedure to be created or edited. To begin working with text in Logo, it is necessary for the edit buffer in memory to be empty. Entering the empty edit buffer to start a text file is achieved by typing

```
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.

When typing the text, you will probably want to type in upper and lower case. Normally when you type on the Commodore, all the letters are upper case; if you use the <SHIFT> key, you get the graphics symbols on the front side of the key. To switch into upper/lower case, hold down the Commodore key and hit the <SHIFT> key. To switch back, repeat the same process. See the section called Commodore Key under Keyboard in the Getting Started chapter of the Commodore 64 User's Guide, and the Beginning Logo chapter of this Tutorial.

NOTE THE DIFFERENCE HERE: When you have finished and are ready to print or save the text, leave the editor by typing

```
<CTRL> G
```

This exits the editor without making any changes to it or trying to evaluate it. For this reason, the text will be saved and printed exactly as it appears. It is not reformatted as procedures are. 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 the text from your workspace before you save it, you must retype the whole thing.

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) followed by a <RETURN>. If you type TO when there is text in the editor, it will be erased. This also happens if you then edit a procedure or enter the graphics mode. If you have not yet saved your text, it will be lost completely. However, if it is on the disk, you can simply read it in again.

Type

```
EDIT
```

to work on the file again. (See warning above.)

The following procedures are in the TEXTEDIT file. They make use of .OPTION commands for SAVE, READ, and PRINTER. (See also the explanation of the .OPTION command in the Glossary.) Normally, SAVE places the contents of the workspace in the edit buffer and then saves the buffer on disk.

The .OPTION will change SAVE so that it places the current contents of the buffer in the disk file instead of first erasing the buffer and putting the contents of the workspace in it.

READTEXT :FILE — Reads a Logo text file into the editor. In this example, "SESAME has two lines in it. Example:

```
READTEXT "SESAME
SHOWTEXT
```

This is a test.
Do you like tests?

SAVETEXT :FILE — Saves the contents of the editor to the disk in the file named. To store the lines above in a different 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 from disk into the editor and prints the file. Uses SHOWFILE. To print the contents of the file SESAME on the printer, type

```
PRINTFILE "SESAME
```

SHOWTEXT — Prints on the screen the text which is in the editor. See examples above. Uses PRINT.MEM.

PRINTTEXT — Prints on the printer the text which is in the editor. Uses SHOWTEXT. Example: to print the contents of the editor, type

```
PRINTTEXT
```

PRINT.MEM :FROM :END
The procedure used by SHOWTEXT.

WHILE:
Procedures for WHILE and UNTIL Commands

To see an example of how to use these procedures, type HELP.

WHILE :COND :ACTION
Will perform the action as long as the condition is TRUE. Both inputs must be lists. The first input must be a statement that results in TRUE or FALSE. The second list may contain any Logo commands. The procedure checks the condition, runs the action, and then checks the condition again and so on.

UNTIL :COND :ACTION
Is similar to WHILE. Here, if the condition changes from FALSE to TRUE, the procedure will stop. The procedure runs until the condition is true, whereas WHILE runs while the condition is true.

ADVENTURE:
An Adventure Game

Just type READ "ADVENTURE and the file will start automatically and give instructions. This is a good example of list processing.

ANIMAL:
The Game that Teaches the Computer About Animals

ANIMAL is a game in which the computer tries to guess the animal you are thinking of by asking you questions. 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 is added to its knowledge tree for the next game.

When you are finished playing, you can type SAVE "ANIMAL, and the next time you play, it will know the animals you taught it. To make it start out fresh, run the procedure INITIALIZE.KNOWLEDGE.

To play ANIMAL, type

READ "ANIMAL then

ANIMAL

The ANIMAL game is a good example of brief, single purpose procedures:

ANIMAL — Prints the greeting, then uses GUESS with the stored :KNOWLEDGE. After a round of the game, prints another greeting, uses WAIT for a pause, then begins again by calling itself.

ANIMAL.INSPECTOR: What's in the ANIMAL Knowledge Base?

This file is intended as a learning aid to be used in the discussion of the ANIMAL game. 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.

Any time you stop ANIMAL, you can examine the knowledge base available to ANIMAL, by typing

```
INSPECT.KNOWLEDGE
```

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 — Does a pretty print of the ANIMAL tree of knowledge. Type <CTRL> W (hold down the <CTRL> key and press <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 dirction 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 co-ordination.

Type

READ "DYNATRACK and then
DYNATRACK

and follow directions to play.

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 it is moving in another direction when you "kick" it, the direction of movement will be changed, but it will take more than one kick to change to the direction in which it is pointing.

Try modifying the game to make the dynaturtle do other things. If you erase the line DRAWTRACK in the procedure DTRACK, and the line CHECK.DTRACK in the procedure DT.DTRACK, the turtle will move all over the screen.

GRAMMAR:
A Random Sentence Generator

There are two main procedures, SENT and POSTCARD, which use other procedures in this file. SENT generates a paragraph of random length with sentences of random construction. POSTCARD prints a short postcard with randomly selected greeting, body, and closing.

The procedure RTRACE is a simple trace procedure that lets you see what is happening as the program runs. Type RTRACE and then SENT or POSTCARD. To turn it off, type RTRACE again.

POSTCARD
Starts by making a global variable with the name WORLD and the value of a list containing the word POSTCARD. It next uses the procedure R to replace the POSTCARD value with a list containing the words GREETING BODY CLOSING. It then uses R again to randomly select a greeting (such as Dear June) to replace the word GREETING, and so on.

SENT
Also starts by making a global variable named WORLD, whose value is a list containing the word PARAGRAPH. Using the procedure R, SENT will replace [PARAGRAPH] with a list containing the word SENTENCE a random number of times. Then R is used to replace each word SENTENCE with a sentence structure picked from a list of structures. The parts of the sentence are then replaced from lists until words are filled in for all the parts of the sentence and the paragraph is printed out. This procedure takes up to a minute to run so be patient.

INSPI: Sample Logo Picture

To see the picture, type

```
READPICT "INSPI
```

The procedure which drew it was run four times with the turtle turned 90 degrees each time and the pen color changed. Each time it was run, it was stopped after it started to double back on itself.

To find out more detail about saving pictures see the Graphics Chapter section called Saving, Reading and Erasing Pictures.

```
TO INSPI :DIST :ANGLE :INCREMENT
  FD :DIST
  RT :ANGLE
  INSPI :DIST :ANGLE + :INCREMENT :INCREMENT
END
```

Type

```
INSPI 8 0 13
```

PIG: A Pig Latin Program

The pig Latin program is an elegant example of the use of recursion and the word and list handling capabilities of Logo. However, the program is not perfect. For instance, how can words containing the letter Y be handled properly?

To start this program, type PIG.

PIG
Prints a line asking you to type in a sentence. The sentence you type becomes the list which is the input to PIGSENT.

PIGSENT :LIST
Outputs the list converted to pig Latin. It does this by calling PIGWORD to convert the first word in the list and then calling itself, PIGSENT, with a new input of all but the first word in the list. This is repeated with the new shorter list and so on until the entire list has been converted.

PIGWORD :WORD
Its input is a word from the list used by PIGSENT. PIGWORD uses VOWEL? to check if the first letter of the word is a vowel, and if it is, it adds AY to the word. If the first letter is a consonant, PIGWORD calls itself with a new input: the new input is the old word with the first letter moved to the end of the word. This continues until it finds a vowel. (What happens if there is no vowel in the word?)

VOWEL?
Outputs true or false if its input is equal to A,E,I,O,U, or Y.

SNOW: A Recursive Snowflake Drawing

The SNOW procedure uses a triangle as its basis and modifies it. The DEMO procedure steps through the levels one by one to give you an idea of how the snowflake is formed.

SNOW :SIZE :LEVEL
SIZE refers to the length of a side and LEVEL refers to the number of times or recursive levels which the program goes through. The procedure looks like a triangle procedure except that SIDE replaces FD.

SIDE :SIZE :LEVEL
The turtle will only move when LEVEL is 0, so that all of the sides (line segments) will be of the same SIZE. The side of the triangle is broken into thirds, and the first and last third are drawn as they would be normally. The middle third is not drawn, but is treated as the base of a triangle one third the size of the original. The other two sides of this smaller triangle are drawn outward from the original triangle. At the next level of recursion each line segment is broken into thirds using the same process. The more levels, the more the sides are broken up into smaller pieces.

TET:
A Graphics Procedure of Variable Complexity

TET is a good example of a recursive procedure. It draws tetrahedra on the points of tetrahedra. TET takes two inputs, SIZE and LEVELS. 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 level of recursion specified. A level of 1 draws only the one large tetrahedron. Try

```
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
```

ADDRESSES, AMODES, ASSEMBLER, OPCODES:
Interfacing Logo and the Assembler

See the section on the assembler in the Reference Guide for a detailed explanation of the Logo/Assembly language interface.

SPRITES, SPRED, ANIMALS, ASSORTED, RUNNER, SHAPES, VEHICLES, DINOSAURS, SUBMARINE, SPRITEDEMOS, VELOCITY:

See the chapter on Sprites for more information about these files.

MUSIC, SOUND, TWINKLE:

See the chapter on Music for more information about these files.

1. 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.

1.1 Modes of Using the Screen

The Logo system uses the display screen in three different ways, or "modes."

1.1.1 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 that allows you to correct any typing errors in a command line which you have typed in DRAW or NODRAW mode. The available editing operations are the ones described in section 1.2.2 corresponding to the keys: DEL, INST, CLR, the up- and left-arrow keys, the CRSR arrow keys, CTRL-A, CTRL-D, CTRL-L, CTRL-G, CTRL-K, CTRL-P.

1.1.2 Edit Mode

Executing the commands TO or EDIT places Logo in edit mode. For example, if you 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. The screen will change color to remind you that Logo is in edit mode.

At this point you can use all of the editing operations described in section 1.2.2 to create and/or edit the text for the procedure. Typing CTRL-C or the RUN/STOP key will exit the editor, cause the procedure to be defined according to the text you have typed, and enter 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 the editor mode, RETURN is just another character which causes the cursor to move to the next line. CTRL-C (or RUN/STOP) causes Logo to evaluate the contents of the edit buffer just as RETURN in DRAW and NODRAW modes causes Logo to evaluate the line just typed. See the section on Editing below.

To edit a previously defined procedure, type EDIT followed by the name of the procedure you wish to edit. No quotation marks are required for this and related commands (PRINTOUT and ERASE). The following line will edit the POLY procedure defined above, when typed in either DRAW or NODRAW mode:

```
EDIT POLY
```

You need not enter the editor if you want only to examine a procedure; the PRINTOUT command (abbreviated PO) will show the definition of a procedure. Type PO followed immediately by the procedure name (as above). Printing out a procedure is often more convenient than using the editor, since it doesn't clear the screen.

To edit the most recently defined procedure, type just EDIT (or its abbreviation, ED). In DRAW mode, typing EDIT brings the current definition of the procedure most recently defined or printed out (using PRINTOUT or PO) into the edit buffer. In NODRAW mode, 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. Also, if you had aborted the definition of a procedure with CTRL-G, the edit buffer 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 is currently defined.

1.1.3 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. Five 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 five-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 function keys f5 and f3 to switch back and forth between splitscreen and fullscreen mode. Pressing f5 while in splitscreen mode will enter fullscreen mode. Pressing f3 will restore splitscreen mode.

This five-line text display is sometimes an inconvenience, since error messages are occasionally too long to fit. If you press the f1 key while in graphics mode, the turtle picture will disappear and the screen will display all text, just as in NODRAW mode. The difference is that Logo is actually still in DRAW mode: turtle commands can be executed, although you will not see the picture being drawn. To make the graphics screen visible after using f1, type f5 to return to fullscreen mode, or f3 to go back to splitscreen mode.

Since it is useful to switch among these modes under program control, the primitives TEXT-SCREEN, SPLITSCREEN and FULLSCREEN are provided. Note that the TEXTSCREEN command is different from NODRAW. NODRAW clears the text screen, clears the graphics screen, and resets all the graphics parameters (pencolor, turtle visibility, pen state, background color, and wrapping mode). You may return to splitscreen or fullscreen mode with no changes after execution of the TEXTSCREEN command, but not after exiting draw mode with NODRAW.

Here is a list of control characters and function keys not related to editing functions. All are available in draw mode and nodraw mode, even while procedures are running. Some exist in edit mode, also, and are specially indicated.

The special actions of the following keys may be inhibited if necessary; see section 4.12.

Non-editing Control Characters and Function Keys

(f1)	In draw mode, this function key gives full text screen.
(f3)	In draw mode, this function key gives mixed text/graphics screen.
(f5)	In draw mode, this function key gives full graphics screen.
CTRL-G	In edit mode, exits the editor without processing the edited text. In draw or nodraw mode, stops execution and returns control to toplevel.
CTRL-W	Stops program execution. Typing any character other than CTRL-G will resume normal processing, including CTRL-W. The CTRL-W acts as a switch. The first time you type it, the program will stop execution or printing of a listing will stop. The second time you type it, program execution or printing will start again. See also TRACE (section 4.10) for a method of executing procedures slowly.
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.

1.2 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.

1.2.1 Line Editor

While you are typing a line of characters to Logo, you can ignore the line editor until you need it. If you mistype a character, you can rub it out with the DEL key. If you forget to put a word at the beginning of the line, you can place the cursor there with CTRL-A and type the missing word. The characters will push the rest of the line to the right; nothing will be lost or overwritten. If you want to insert characters anywhere in the line, simply move the cursor there with the arrow keys, and type

what you want. To go to the end of the line, type CTRL-L. To delete the character at the cursor, use CTRL-D.

To finish the line and have Logo act upon it, type RETURN. 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. To delete all characters from the cursor to the end of the line, use CTRL-K.

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. Specifically, at the right screen edge, the right CRSR arrow key will still move to the next character in the line, even though it is at the left edge of the screen and on the next row of characters.

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 the CRSR up arrow key (or CTRL-P).

1.2.2 Screen Editor

For defining procedures, Logo has a screen editor which you enter by typing EDIT, ED, or TO, followed by the name of the procedure you wish to define.

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. In edit mode you can delete the RETURN, just as you can any other character, and restore the split line to one line.

Various other commands are available for editing the line on which the cursor appears and moving to other lines. To move to the next line, type the CRSR down key, or CTRL-N. To move to the Previous line, type the CRSR up key, the up-arrow key (next to RESTORE), or CTRL-P.

Lines may be of any length, as long as they fit in the edit buffer. Lines which are longer than 40 characters "wrap around" the screen. 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.

Note that there is a slight difference 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 the up and down CRSR arrow keys or type CTRL-P or CTRL-N and 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 the HOME key to center the current line on the screen.

To exit the editor and have the procedure or procedures you typed in be defined, type CTRL-C, or hit the RUN/STOP key. 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 (provided you didn't use graphics or filing in the meantime).

The text you type in the editor doesn't have to be a procedure. See the Tutorial section on text editing to find out how to use Logo for editing text, saving it on disk, and printing it.

Here is a summary of the editing commands available:

Keyboard Editing Commands

HOME — In edit mode, scrolls the text so that the line containing the cursor is at the center of the screen. This key has no effect if the text is shorter than one page. Use this key to move text around on the Commodore screen until it is placed conveniently by typing HOME followed by one of up/down cursor motion keys as necessary.

SHIFT-CLR — In immediate mode, clears the text screen and aborts the current line being typed.

SHIFT-INST — Quotes the following character. Allows control characters and other characters to be inserted in Logo words. SHIFT-INST followed by CTRL-2 (WHITE) will insert the special character for changing the character color to white. Another example:

```
TO REV
PRINT "SHIFT-INST CTRL-9 HELLO SHIFT-INST CTRL-0
END
```

As usual in Logo, there should be no spaces after the quote mark when you type in the example above. SHIFT-INST allows you to insert any character except return (including cursor motion keys and CTRL-G).

DEL — Rubs out the character immediately to the left of the cursor and moves the cursor one space to the left.

CRSR keys up, down, left, right — Move the cursor one character to the left, right, up or down, without rubbing out any character. Note that although the CRSR up and CRSR left keys require the shift key, the left arrow key (located near the 1 key) and the up arrow key (located near RESTORE) serve the same function (without auto-repeat).

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. 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-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 toplevel.
CTRL-K	Deletes all characters to the right of the cursor on the current line.
CTRL-L	Moves the cursor to the end of the current line.
CTRL-N	Moves the cursor down to the next line. Equivalent to the CRSR down arrow key.
CTRL-O	Opens a new line at the cursor position. That is, CTRL-O is equivalent to typing RETURN and then CTRL-P. It is most useful for adding new lines in the middle of procedures.
CTRL-P	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. Equivalent to the CRSR up arrow key.

1.3 Using Commodore Peripherals

Logo's ordinary input and output operations deal with the Commodore keyboard, the screen, and one disk drive. There are also commands for reading input from up to four game paddles or two joysticks that can be attached to the Commodore. (See the PADDLE, PADDLEBUTTON, JOYSTICK, and JOYBUTTON primitives.)

1.3.1 Printing Procedures on a Printer

The following examples will work if you have a VIC-1525E serial printer attached to your Commodore 64 at serial address 4. If you have a VIC-1525 printer (manufactured for VIC-20 computers), see your Commodore dealer for update kit information.

To obtain a paper printout of a procedure called CIRCLE, you could type

```
PRINTER
PRINTOUT CIRCLE
NOPRINTER
```

PRINTOUT ALL or PO ALL will list all procedures and names. A list of procedure names or variables may be used in place of a single procedure name. This allows ordering of the printout sequence.

The following command line will print out the procedures BIRD, HEAD, WINGS, TAIL, and LEGS, in that order:

```
PRINTER
PO [BIRD HEAD WINGS TAIL LEGS]
NOPRINTER
```

To list all the procedures, but no names (global variables), type

```
PO PROCEDURES
```

Here is a procedure which prints out procedures to the printer:

```
TO HPO :PROCEDURE
  PRINTER
  RUN LIST "PRINTOUT :PROCEDURE
  NOPRINTER
END
```

HPO means "hardcopy printout." Note that since HPO is a regular procedure which gets evaluated inputs, you must type a quotation mark before the name of the procedure you wish to print out. Since PRINTOUT does not evaluate its input, it is impossible to pass procedure names to it through a variable name. Therefore, we construct a list and then RUN that list. Use HPO like this:

```
HPO "CIRCLE
HPO [BIRD TAIL]
```

The simplest way to produce a printout of the turtle graphics screen is to save the picture on disk using the SAVEPICT command. The Commodore VIC-1525E printer can print a saved image of the turtle graphics screen. You can store pictures on disk with the SAVEPICT (section 1.5.2) command. SAVEPICT saves the picture as two files: one whose name ends in ".PIC1" and the other in ".PIC2". The .PIC1 file contains the picture elements (pixels) and one additional byte at the end indicating background color and mode (SINGLECOLOR or DOUBLECOLOR). The .PIC2 file contains color information and is used only by the READPICT command.

The PRINTPICT program on the Utilities Disk will print a saved picture on the VIC-1525E printer. Note that you must exit Logo to use this program.

Since 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, figures that look square on the screen will come out rectangular when printed on paper. If you are producing output especially for printing, you might want to determine the proper aspect ratio to use with your 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. See the description of the .ASPECT primitive, section 4.11.

1.4 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 15 as input. (PENCOLOR also accepts −1 as input, which is equivalent with PENERASE. See the PENERASE primitive.)

The actual color that appears on the screen corresponding to any of these color names can vary greatly depending on the adjustment of the TV screen. If you have a black and white monitor, the "colors" will appear as striped vertical lines. See "Drawing in Black and White" below.

These considerations aside, the correspondence of colors to numbers is:

number	color
0	black
1	white
2	red
3	cyan
4	purple
5	green
6	blue
7	yellow
8	orange
9	brown
10	light red (LRED)
11	dark gray (DGRAY)
12	medium gray (MGRAY)
13	light green (LGREEN)
14	light blue (LBLUE)
15	light gray (LGRAY)

Drawing in PENERASE (or PENCOLOR −1) makes the turtle draw with an eraser instead of a pen: all dots or lines it crosses are erased.

If you don't explicitly give any BACKGROUND or PENCOLOR commands, Logo will default to BACKGROUND 11 (light grey) and PENCOLOR 1 (white).

The Utilities Disk contains a file of Logo names for these colors. After reading the file COLORS, you will be able to refer to these colors by names:

```
READ "COLORS
PO NAMES
BACKGROUND :BLUE
PENCOLOR :WHITE
```

Drawing in Black and White
If you have a black and white monitor or television set, you may wish to use Logo in Black and White mode, in which all background, text, and pen colors are either black or white. Immediately after starting Logo, insert the Utilities Disk and type

```
READ "B&W
```

Logo will restart in Black and White mode, and will remain in it until loaded from disk again. Calls to GOODBYE will not affect Black and White mode.

1.4.1 Color Modes

The commands SINGLECOLOR and DOUBLECOLOR modify the way the turtle draws lines. The SINGLECOLOR setting allows the turtle to draw in only one color per 8 × 8 pixel region. (A pixel is a dot of light on the screen, about equivalent to a turtle step.) If you draw over a line in a different color, a small section of the old line will change color. Lines drawn in SINGLECOLOR mode are thinner and more precise, but the colors are harder to see than in DOUBLECOLOR mode. Additionally, the STAMPCHAR primitive works best in SINGLECOLOR mode.

All sixteen colors may be used in both SINGLECOLOR and DOUBLECOLOR modes. Note that when switching from one mode to the other, the graphics screen is cleared, and PENCOLOR and BACKGROUND are reset to the default values. The default values can be changed with the .OPTION command. See section 4.12.

Logo starts up in SINGLECOLOR mode. To compare the effects of these two modes, try running a turtle graphics procedure, and then type DOUBLECOLOR and run it again.

In doublecolor splitscreen mode with certain background colors, a thin horizontal line will appear separating the graphics area from the text area. This is normal and should be no cause for concern.

1.5 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; the files are distinguished by their names. The names of the files are listed in the disk catalog.

1.5.1 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 Tutorial chapter "Beginning Logo" and in the VIC-1541 floppy disk drive manual. To save your workspace, use the SAVE command. For example, the following will save all the procedure definitions and names currently in the workspace in a file

named MYSTUFF. If the disk already had a file with the name you chose, the old file will first be deleted.

```
SAVE "MYSTUFF
```

SAVE saves everything in the workspace and will not save just a procedure by the same name as the input to SAVE. File names and procedure names are independent. SAVE may take two inputs if SAVE and the inputs are enclosed in parentheses. The first input is the file name and the second a list of procedures to save. The list may additionally contain the words NAMES or PROCEDURES to invoke saving all variables or all procedures:

```
(SAVE "ROCKET [ROCKET ROCKTOP ROCKMIDDLE ROCKBOTTOM])
```

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. (Logo filing makes use of the same memory area as for drawing pictures and editing procedures. Issuing any filing command while in draw mode will first move you to nodraw mode.)

Notice that the file names given as inputs to SAVE and READ are preceded by a quotation mark and have no following quote. If you leave off the quote, Logo will assume you mean to call a procedure, and expect that procedure to output a file name.

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

```
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. To delete other types of files, use the DOS command, as described later.

1.5.2 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 two files are generated, one with ".PIC1" appended to the name, and the other with ".PIC2" appended. Do not include the .PIC1 or .PIC2 part of the name when you use the READPICT command, since Logo assumes it any time you use a picture filing command.

When you use READPICT, Logo automatically restores the background color and color mode (SINGLECOLOR or DOUBLECOLOR). Anything already on the graphics screen will be erased.

To erase a picture from the disk, use the ERASEPICT command, which takes as input the name of the picture to be erased.

See section 1.3.1 to find out how to get paper printouts of these files.

2. Miscellaneous Information

2.1 Self-starting files

The Logo READ primitive reads a file from disk and defines the procedures and names it contains. Usually, you must run some procedure to start a program running. In some cases, you might want to have programs which start automatically on loading. A file containing such a program is called a "self-starting file." To make a self-starting file, create a variable named STARTUP containing a list of Logo commands that you want to be RUN after Logo reads in the file.

```
MAKE "STARTUP [START]
```

This assumes that there is a procedure named START. Saving the workspace with the Logo SAVE command causes STARTUP to get saved with it, just as are all other global variables.

When you READ the file back into the workspace, Logo will attempt to execute the commands in STARTUP. If :STARTUP is not a list, nothing happens, and no error is generated.

If the file had been saved as DEMO, you would type

```
READ "DEMO
```

The procedures load and then :STARTUP runs. In this case :STARTUP runs the procedure START. (The file INSTANT on the Utilities disk is set up this way.)

If you have a value of STARTUP in the workspace and you do not want it to be overwritten when a new file having a STARTUP variable is read in, use the following procedure:

```
TO LOCALREAD :FILE
  LOCAL "STARTUP
  READ :FILE
END
```

This procedure causes the variable STARTUP in the file being READ to be treated as a local variable. After it runs and the procedure stops, it disappears leaving the value that STARTUP had before the file was read.

Using READ in a procedure instead of immediate mode causes another change; normally as each procedure is read into the workspace, Logo prints a message saying that that procedure is DEFINED. In this case, nothing is printed out, except that the prompt is printed after the file is read in.

Sometimes you might need to read in and edit the contents of a self-starting file. To keep :STARTUP from being RUN, hold down the Commodore key until the red light on the disk drive goes out. The STARTUP variable will still be defined, but no special action will occur.

2.2 INST key — Printing Non-Standard and Reverse Characters

The INST key allows you to add non-standard characters to Logo words. This includes the CTRL characters as well as the cursor motion keys. For example, INST followed by a CTRL-2 (white) will cause any further letters typed to be white. Returning to the default color can be done using the INST and CTRL-7. Following is another example which will set the characters to reverse, print HELLO in reverse, and turn off the reverse setting.

```
TO REV
  PRINT "SHIFT-INST CTRL-9 HELLO SHIFT-INST CTRL-0
END
```

You may use the INST key to insert the two solid arrow keys and various other function keys. Any key other than RETURN may be inserted in this manner.

Another use for the characters inserted with the INST key is in procedures which read characters from the keyboard and act on them. For example

```
TO PLAY
  COMMAND READCHARACTER
  PLAY
END

TO COMMAND :CHAR
  IF :CHAR="F FD 10 STOP
  IF :CHAR="B BK 10 STOP
  IF :CHAR="SHIFT-INST SHIFT-CLR CLEARSCREEN STOP
  IF :CHAR="SHIFT-INST DEL PENERASE STOP
  IF NUMBER? :CHAR PENCOLOR :CHAR
END
```

Note that you need not use the INST key when actually running the above procedures, only when typing them in.

2.3 Various System Parameters

This section contains various esoteric information about Logo and about this specific implementation. It is certainly not necessary to know what is presented here in order to use Logo; these topics are covered for the curious.

The Graphics Screen

When pointing straight up, the turtle can go 129 steps before wrapping around to the bottom of the screen. It can go 130 steps downward before wrapping around to the top. It can go 160 steps when pointing to the left, and 159 when going to the right. If you change the aspect ratio (see the .ASPECT primitive listed in the Glossary), then the allowable vertical range will change, but the horizontal range will remain the same.

Numbers
The smallest positive number on which Logo can perform operations is 1.999N38, and the largest is 1.7E38. The largest positive number which is not "floating point" is 2147483647, and the largest negative is −214783647. Sequences of digits which normally represent numbers outside this range are considered to be the names of procedures.

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. 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. Some special output devices receive control codes in the form of characters. Such devices may require an ASCII 0; it may be printed with the .CTYO command (see section 4.8).

The READCHARACTER primitive, abbreviated RC, reads a key from the keyboard and outputs a single-letter word. There are certain "interrupt" keys that will not normally be output by RC. These are F1, F3, F5, CTRL-W, CTRL-Z, and CTRL-G. You may disable the special actions of these keys and enable RC to output them. See section 4.12.

To find out the ASCII value of any other key, type PRINT ASCII RC and RETURN, and type the key. Also, the Commodore 64 User's Manual contains a table of characters and their equivalents.

Line length
Lines typed in to Logo at the toplevel editor may not be more than 256 characters long. Additionally, the list which is input to RUN and REPEAT, and each sub-list in the second input to DEFINE must abide by this restriction.

Lines typed in the screen editor (as with TO procedurename) 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 is 8192 characters long. If you try to SAVE or EDIT procedures which take up more than 8192 characters, Logo will print the error message "TOO MANY CHARACTERS TO EDIT" or "TOO MANY CHARACTERS TO SAVE." The best way to proceed is to EDIT or SAVE a selected list of procedures, rather than the whole workspace. See the documentation on the EDIT and SAVE primitives.

Storage in Logo
In Logo, each procedure is stored as a list of lines, which are themselves lists of words and other lists. (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 by PO'ing the procedure: 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. This information is completely arcane.) Each use of a word takes up one storage node, no matter how many characters it has. The names themselves are stored only once, in the Logo list of all words (accessible via the .CONTENTS primitive). Thus, there is almost no penalty for using long, descriptive procedure and variable names.

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 storage node, 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.

2.4 Memory Organization Chart

This chart describes how the Logo system uses the available address space in the Commodore 64.

Location	Use	Size
0000-00FF	Page 0 (Shared)	1/4K
0100-01FF	Stack	1/4K
0200-03FF	Kernel internal storage	1/2K
0400-07FF	Text Screen (matrix)	1K
0800-0BFF	Graphics matrix	1K
0C00-0DFF	8 Sprites	1/2K
0E00-0FFF	Logo internal storage	1/2K
1000-1FFF	Recursion Stack	4K
2000-4000	Graphics screen	8K
4000-AFFF	Logo Interpreter	28K
B000-DFFF	Nodespace	12K
E000-EBFF	Typecodes	3K
EC00-FDFF	Text Strings	4 1/2K
FE00-FF00	Input line	1/4K
FF00-FF8F	UNUSED	1/8K
FF90-FFFF	Kernel, 6510 vectors	1/8K

3. Assembly Language Interfaces to Logo

The Logo system for the Commodore 64 has been designed to be both powerful and easy to use. Writing and executing programs in Logo using the primitives listed in the Glossary 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/intelligible to people who are familiar with assembly language programming on the 6502 CPU. If you have an interest in this section but no experience in assembly language programming, read the Commodore 64 Programmer's Reference Guide, which contains an introduction to machine language.

The Logo system has various "hooks" built in to it that enable users to directly access memory locations and to interface assembly language routines to Logo programs. The Logo Utilities Diskette includes a 6502 machine language assembler that aids in doing this.

Another hook built in to Logo allows you to create simple animation effects by supplying a new shape to be displayed in place of the Logo turtle. Another hook allows 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 and to access disk files in non-standard ways. See the section on the .OPTION primitive (section 4.12) to find out about these and other hooks into the Logo system.

3.1 .EXAMINE and .DEPOSIT

These two commands are essentially the BASIC language PEEK and POKE routines. .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 in cases where the facility supplied by PRINTER/NOPRINTER 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 machine 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. (Throughout this section, we use the convention of specifying hexadecimal numbers as prefixed by a dollar sign, e.g., $9E is 158 decimal.) That way, you could type, for example

```
.EXAMINE HEX "9E
```

rather than

```
.EXAMINE 158
```

The Commodore 64 is based on the 6510 microprocessor. The 6510 has the same instruction set as the 6502, but (as used in the Commodore 64) has a powerful "memory mapping" facility of which Logo makes considerable use. By this means, several numbers can be stored in apparently the same

location. The contents of the least significant three bits of location 1 control which set of memory addresses is in use. The .DEPOSIT and .EXAMINE primitives have an option which controls which memory map is in effect during their execution. Normally this map is 6, which allows reading and writing to the I/O region of memory ($D000-$DFFF), and examining the KERNEL ($E000-$FFFF). Another configuration available is 4, which lets you access the Logo user storage space, so you can look at and alter Logo objects (lists, words and procedures). In memory map 2, locations $D000-$DFFF refer to the character-generator ROM. You can use this mode to read the definitions of the Commodore 64 character set.

Note that the .DEPOSIT and .EXAMINE commands have separate memory map options, so you must change both if reading and writing a bank of memory is necessary. Additionally, .DEPOSIT-ing to KERNEL or character set ROM is not recommended, since it will instead alter read/write memory at the same location. See section 4.12 to find out how to set these and other options. The Commodore 64 Programmer's Reference Guide has a full discussion of memory maps in the section "Memory Management on the Commodore 64."

3.2 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 $0C40 and extends to $0DFF. (Note that this is the same region of memory reserved for sprite shapes, so if you use sprite shapes you cannot load assembly language routines.) 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 in section 3.3.

When your routine begins executing, the 6510 CPU will be in memory map 6, and 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 FREEPZ through FREEPZ+32 as temporary storage locations, without worrying about restoring them before returning. These storage locations are volatile; that is, Logo will use these locations between successive calls to your routine.

NARG2 through NARG2+3 contain the second input to .CALL, stored as a four-byte fixnum in two's complement form. Thus .CALL HEX "0C40 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
$40	$0C	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 no value, simply end the routine with an RTS instruction.

When a machine language routine has encountered some error condition that would make it inappropriate to return to the Logo procedure that called it, it should jump to PPTTP, which effectively executes the Logo TOPLEVEL primitive.

Here is a simple example which prints "HI" on the screen, using the Logo-supplied character output routine. 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 (but see section 3.3 below).

```
                ORG $0C40
TPCHR           EQU <see ADDRESSES file>
HI:             LDA #'H
                JSR TPCHR
                LDA #'I
                JSR TPCHR
                LDA #$0D
                JSR TPCHR
                RTS
                END
```

Now you can set the Logo variable HI to the address of the label HI and execute this new "primitive" by typing

```
.CALL :HI 0
```

Note that an input is needed, even though it is ignored.

3.3 The Logo Assembler

The Logo assembler is a 6502 assembler that is written in Logo. 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. It is "self starting", which means it initializes itself via the magic variable :STARTUP as described in the paragraph on Self Starting Files in the Miscellaneous Information section.

```
READ "ASSEMBLER
```

Write the routine you wish to assemble in the format of a Logo procedure, using the Logo editor. For example, the example above would be written as the procedure:

```
TO HI.CODE
HI: LDA # "H
  JSR TPCHR
  LDA # "I
  JSR TPCHR
  LDA # [$ "0D]
  JSR TPCHR
  RTS
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 in section 3.3.1.

Once you have defined the procedure you may now assemble it by typing

```
READ "ADDRESSES
ASSEMBLE "HI.CODE
```

ASSEMBLE will now assemble the instructions and place them in the default location ($0C40, but you can change it by changing the value of the Logo variable ORG). Any labels in the code (such as HI, above) will now be defined as Logo symbols. To call the routine, type

```
.CALL :HI 0
```

3.3.1 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. Example: LOOP: CMP # 0

- References to page-zero memory locations (other than indirect-indexed and indexed-indirect) 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. Examples:

```
LDA ! NARG2
STA ! TABLE ,X
```

- Immediate mode references are standard, using the # character. Example:

```
LDA # TABLE.LENGTH
```

- There must be a space following every exclamation point and number sign ("#"), and after every label or reference to a label. Spaces around parentheses and brackets are supplied automatically by Logo. In the indexed instructions ",X" and ",Y" are to be written without spaces.

- 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, PRINTing, etc.) 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. Since the contents of the bracketed list is not processed by the assembler, but rather by Logo, references to the values of labels inside square brackets must have dots (:) before them, spaces have their normal significance, and in general all normal Logo rules must be followed.

- 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 $ Logo procedure defined in the assembler file 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 (see the example in section 3.3.) 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. Example:

```
BIT.TABLE: 1
           2
           4
           8
```

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.

3.3.2 Saving Assembled Routines on Disk

With the BSAVE primitive, you can save the actual machine code that the assembler generates. BSAVE takes three inputs: the file name, the start address, and the end address. The following will save the entire machine-language area in a file called ROUTINES.

```
BSAVE "ROUTINES.BIN $ "0C40 $ "0DFF
```

A better idea would be to save only the region of memory containing your program. Immediately after assembling a routine, the Logo variable :ORG contains the first address it occupies, and the variable :END contains the last. So, an optimal disk save would be

```
BSAVE "ROUTINES.BIN :ORG :END+1
```

To load the routines into Logo, type

```
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. The easiest way to do this is to type EDIT NAMES, and put a TO INIT at the beginning of the edit buffer. Use the editing keys to move down to the last name associated with your routine, put on the next two lines

```
BLOAD "ROUTINES
END
```

Then type CTRL-C and type

```
(SAVE "ROUTINES [INIT])
```

Later, to reload your routine, type READ "ROUTINES and type INIT.

3.4 Example: Changing Colors

This section presents an example of an assembly language extension to Logo.

The Commodore 64 uses a clever mechanism to allow high resolution graphics in 16 colors. If each of the 64000 (320*200) points on the screen (pixels) were able to be in any of the 16 colors independently, there would have to be 32000 bytes of memory devoted just for the colors! That wouldn't leave much room for programs. So, in SINGLECOLOR mode the Commodore 64 groups pixels into 8×8 blocks and allows each block to have only one color. This reduces the memory requirement by a factor of 64 (8*8), to 500 bytes. In actuality, it's stored as 1000 bytes, with only four bits of each byte in use. (In multicolor mode, all eight bits are used.)

All the information about the colors of lines and points on the screen is stored in this 1000 byte area, located at $0800. We can take advantage of this fact and write a procedure which changes the color of designs after they've been drawn. Let's write a procedure to change all the lines of color :OLD to color :NEW. The idea is to go through each of the 1000 bytes and check the lower and upper nybbles (four bits) against the old color, changing it to the new color if necessary. Below is a simplified version of this algorithm. It checks only the upper nybble, and so works only in SINGLECOLOR mode.

```
TO ALTER.COLOR :OLD :NEW
  ALTER.NYBBLES :OLD :NEW 2048 2048+1000
END

TO ALTER.NYBBLES :OLD :NEW :START :END
  IF :START>:END STOP
  IF UPPER.EQUAL? :OLD .EXAMINE :ADDR
    UPPER.CHANGE :NEW :ADDR
  ALTER.NYBBLES :OLD :NEW :START+1 :END
END

TO UPPER.EQUAL? :NYBBLE :BYTE
  OP :NYBBLE=QUOTIENT :BYTE 16
END

TO UPPER.CHANGE :VALUE :LOC
  .DEPOSIT :LOC
    (BITOR :VALUE*16 (BITAND .EXAMINE :LOC 15))
END
```

Although these procedures work, they are impractical since execution takes about one minute. A machine language implementation of the same algorithm would be much faster. Here is an implementation of the routine in 6502 assembly language, in the Logo assembler format. Additionally, it works in DOUBLECOLOR mode by checking the location COLMOD (which tells which color mode Logo is in) and altering the lower nybble if necessary.

```
TO CODE
[MAKE "COLORS $ "0800]
[MAKE "PTREND :COLORS+1000]
[MAKE "OCOLOR :NARG2]
[MAKE "NCOLOR :NARG2+1]
[MAKE "OCOLOR.H :NARG2+2]
[MAKE "NCOLOR.H :NARG2+3]
[MAKE "PTR :FREEPZ]
CCHANGE: LDA # [LO8 :COLORS]
         STA ! PTR
         LDA # [HI8 :COLORS]
```

```
         STA ! [:PTR+1]
         LDA ! OCOLOR
         ASL A
         ASL A
         ASL A
         ASL A
         STA ! OCOLOR.H
         LDA ! NCOLOR
         ASL A
         ASL A
         ASL A
         ASL A
         STA ! NCOLOR.H
LOOP:    LDY # 0
         LDA ( PTR ) ,Y
         AND # [$ "0F]
         CMP ! OCOLOR
         BNE CHKHI
         JSR DOLOW
CHKHI:   LDA ( PTR ) ,Y
         AND # [$ "F0]
         CMP ! OCOLOR.H
         BNE NXLOOP
         JSR DOHI
NXLOOP:  INC ! PTR
         BNE NXL1
         INC [:PTR+1]
NXL1:    LDA [:PTR+1]
         CMP # [HI8 :PTREND]
         BNE LOOP
         LDA PTR
         CMP # [LO8 :PTREND]
         BNE LOOP
         RTS
DOHI:    LDA ( PTR ) ,Y
         AND # [$ "0F]
         ORA ! NCOLOR.H
         STA ( PTR ) ,Y
         RTS
DOLOW:   LDA COLMOD
         BEQ DRTS
         LDA ( PTR ) ,Y
```

```
          AND # [$ "F0]
          ORA ! NCOLOR
          STA ( PTR ),Y
DRTS:     RTS
```

Some method of interfacing the machine-language routine to Logo 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 old color (OCOLOR) and the new color (NCOLOR). Since we arranged memory locations so that OCOLOR is the low byte of the second input to .CALL, and NCOLOR the next byte, the following procedure will give the two inputs to machine-language routines:

```
TO .CALL.2 :ADDR :INPUT1 :INPUT2
  .CALL :ADDR :INPUT1 + :INPUT2*256
END
```

If there were many inputs needed, it would be better to .DEPOSIT them in memory locations reserved by your assembly program before calling the routine. Below is the new ALTER.COLOR procedure, which uses the assembly-language program above, together with the .CALL.2 procedure:

```
TO CCHANGE :OLD :NEW
  .CALL.2 3136 :OLD :NEW
END
```

The Utilities Disk contains three files pertaining to the CCHANGE routine. One is the source for the assembly-language program, and the other is the saved machine-language code. The last is the Logo procedure above together with startup information. To try out the routine, type

```
READ "CCHANGE
```

Once in DRAW mode (SINGLECOLOR or DOUBLECOLOR), to change all the red items to green, type

```
CCHANGE 2 5
```

You can get interesting effects by drawing "invisible" objects in same pencolor as the background color, and then issuing a CCHANGE command to change all background colored objects to some other color, making them instantly appear.

3.5 Useful Memory Addresses

This section contains brief descriptions of addresses in the Logo program for interfacing assembly language programs to Logo as described in section 3.2. 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. When using an address in a Logo procedure, it should be preceded with the character : (dots) as in .EXAMINE :EPOINT. In the assembler, you may refer to the variable as a label without the dots, but inside a list of commands to execute the dots must be present.

Page zero locations:

NARG2 — Second input to .CALL. 4 bytes. See section 3.2.

FREEPZ — First user-available page zero location. All memory from here to FREEPZ+32 is available for user assembly-language routines. These locations are for temporary use only, and will not be saved between .CALLs. See section 3.2.

Vectors and Flags

TOPIRQ — This vector normally points to the Logo internal routine which is executed every 60th of a second while the raster scan is at line 16. You may insert the address of your own interrupt handling routine here, provided it exits by jumping to the address formerly specified by TOPIRQ.

KERVER — This location is initialized to a copy of the Commodore Kernel version number byte, located in Kernel ROM at location $FF80. If this location contains $64, then Logo will operate in "black and white" mode. The B&W file on the Utilities Disk uses this feature.

COLMOD — This location contains 0 in SINGLECOLOR mode and 1 in DOUBLECOLOR mode. Depositing in this location will produce strange visual effects. It is documented for the use of the CCHANGE assembly language example.

VERSION — This location contains the version number of the Logo currently running. Each new release of Logo will have a different version number.

RPTFLG — If this location contains 0 (normally), only selected cursor motion keys will repeat if held down. Depositing 128 will cause all keys to repeat, and depositing 255 will disallow all repeating keys. Note that the SHIFT/COMMODORE combination to enter upper-/lowercase display mode is difficult to type in all-keys-repeat mode; use PRINT1 CHAR 14 and PRINT1 CHAR 142 instead.

Routines and Entry/Exit Points

OTPFX2 — Jumping to this address will cause the .CALL to output the integer stored in NARG2 through NARG2+3. See section 3.2.

OTPFIX	Like OTPFX2, but returns to Logo with the integer value stored in the four successive bytes starting with the page-zero location pointed to by the Y register.
PPTTP	An alternate exit for user machine-language routines. Jumping to this address runs the Logo primitive TOPLEVEL (section 4.7). It is useful to return to Logo in this manner when some error condition has occurred, making it inappropriate to continue executing .CALLing procedure.
COUT	Logo's normal screen character-output routine. Prints the character in the accumulator on the screen.

4. Logo System Primitives Glossary

Logo is a full-scale, powerful computer language. It includes commands for graphics, arithmetic and logical operations and list processing. It also incorporates a real-time screen editor that can be used both for editing command lines as they are typed, for editing procedure definitions, or for text editing.

Following is a list of sections and the commands within each section.

Graphics Commands

BACK BK
BACKGROUND BG
CLEARSCREEN CS
DOUBLECOLOR
DRAW
DRAWSTATE
FORWARD FD
FULLSCREEN <f5>
HEADING
HIDETURTLE HT
HOME
LEFT LT
NODRAW ND
NOWRAP
PENCOLOR PC
PENDOWN PD
PENUP PU
RIGHT RT
SETHEADING SETH
SETSHAPE
SETX
SETXY
SETY
SHAPE
SHOWTURTLE ST
SINGLECOLOR
SPLITSCREEN <f3>
STAMPCHAR
TELL
TEXTBG
TEXTCOLOR
TEXTSCREEN <f1>
TOWARDS
WHO
WRAP
XCOR
YCOR

Numeric Operations

+
-
*
/
>
<
ATAN
BITAND
BITOR
BITXOR
COS
INTEGER
NUMBER?
QUOTIENT
RANDOM
RANDOMIZE
REMAINDER
ROUND
SIN
SQRT

Word and List Operations
- =
- BUTFIRST BF
- BUTLAST BL
- COUNT
- EMPTY?
- FIRST
- FPUT
- ITEM
- LAST
- LIST
- LIST?
- LPUT
- MEMBER?
- SENTENCE SE
- WORD
- WORD?

Defining and Editing Procedures
- DEFINE
- EDIT ED
- END
- ERASE ER
- TEXT
- TO

Naming
- LOCAL
- MAKE
- THING
- THING?
- ALLOF
- ANYOF
- ELSE
- IF
- IFFALSE IFF
- IFTRUE IFT
- NOT
- TEST
- THEN

Debugging
- CONTINUE CO
- PAUSE
- NOTRACE
- TRACE

Input and Output
- .CTYO
- ASCII
- CHAR
- CLEARTEXT
- CLEARINPUT
- CURSOR
- CURSORPOS
- FPRINT
- JOYSTICK
- JOYBUTTON
- NOPRINTER
- PADDLE
- PADDLEBUTTON
- PRINT PR
- PRINT1
- PRINTER
- RC?
- READCHARACTER RC
- REQUEST RQ
- SETDISK

Filing and Managing Workspace
- BLOAD
- BSAVE
- CATALOG
- DOS
- ERASEFILE
- ERASEPICT
- ERNAME
- PRINTOUT PO
- POTS
- READ
- READPICT
- SAVE
- SAVEPICT

Control
- GO
- GOODBYE
- OUTPUT OP
- REPEAT
- RUN
- STOP
- TOPLEVEL

Miscellaneous Commands
.ASPECT
.CALL
.CONTENTS
.DEPOSIT
.EXAMINE
.GCOLL
.NODES
.OPTION
.SPRINT
;

4.1 Graphics Commands

BACK — Moves the turtle in the opposite direction from which it is pointing by the amount specified, drawing in the current pencolor (or erasing). Abbreviated: BK.

BACKGROUND — Takes a number 0 through 15 as input and sets the color of the graphics screen background as described in section 1.4. To find out the current background color, use ITEM 3 DRAWSTATE. Abbreviated: BG.

CLEARSCREEN — Clears the graphics screen without moving the turtle from its current position. Abbreviated: CS.

DOUBLECOLOR — Allows two colors per 8 × 8 pixel region, instead of just one. The resulting colors are much richer and easier to see than in SINGLECOLOR (the default) mode, but drawings will look less precise because lines are thicker horizontally. All sixteen colors can be used. Since the horizontal resolution is one half that of SINGLECOLOR mode, STAMPCHAR graphics will not be legible in DOUBLECOLOR mode. Executing DOUBLECOLOR will cause all graphics parameters to be reset to default.

DRAW — Clears the turtle graphics screen and enters DRAW mode in background 11 (grey) and pencolor 1 (white) with wrapping allowed (See WRAP, NOWRAP.) If Logo is already in DRAW mode, then it 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. To change the default background or pen color, see the .OPTION command in the Miscellaneous Commands section below. Use ITEM 6 DRAWSTATE to find out whether Logo is in DRAW mode or NODRAW mode.

DRAWSTATE — Takes no inputs. Outputs a list of nine items giving information about the state of the turtle and the graphics screen. You can use the ITEM primitive (or FIRST, LAST, BUTFIRST, etc.) to access the information in this list. Here is a typical output:

[TRUE TRUE 11 1 DRAW SINGLECOLOR SPLITSCREEN 14 6]

ITEM 1 TRUE if the pen is down, FALSE if it is up.

ITEM 2 TRUE if the turtle is shown, FALSE if it is hidden.

ITEM 3 Background color, 0...15.

ITEM 4 Pen color 0...15, or −1 for PENERASE.

ITEM 5 Screen mode, NODRAW or DRAW

ITEM 6 Color mode, SINGLECOLOR or DOUBLECOLOR.

ITEM 7 TEXTSCREEN, FULLSCREEN or SPLITSCREEN

ITEM 8 Text background color (TEXTBG).

ITEM 9 Color of text characters (TEXTCOLOR)

FORWARD — Moves the turtle in the direction in which it is pointing by the amount specified. If the pen is down, the turtle leaves a trail. (Or erases, with PENERASE.) Abbreviated: FD.

FULLSCREEN — In graphics mode, gives full graphics screen. Complementary to SPLITSCREEN. Equivalent to the function key F5. Use ITEM 7 DRAWSTATE to find out the current screen mode. See the Miscellaneous Commands section below for the .OPTION which applies to FULLSCREEN.

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 disappear. Does not affect drawing. Abbreviated: HT.

HOME — Moves the turtle to the center of the screen, pointing straight up. Note that if the pen is down the turtle will draw a line.

LEFT — Rotates the turtle. Takes an input that specifies the number of degrees to rotate. Zero, negative, and fractional inputs are allowed. Abbreviated: LT.

NODRAW — Exits graphics mode, giving a clear text page with the cursor homed in the upper left-hand corner of the screen. DRAW will return to draw mode with a clear turtle graphics screen. To view the full text screen temporarily without erasing your drawing, use function key F1 or the

TEXTSCREEN command. Use ITEM 6 DRAWSTATE to find out the current screen mode. Abbreviated: ND.

NOWRAP — 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. Logo remains in NOWRAP mode until the next WRAP or NODRAW command. DOUBLECOLOR and SINGLECOLOR reset wrapping to WRAP.

PENCOLOR — Takes a number from −1 through 15 and sets the color of the lines that the turtle will draw as described in section 1.4. See also PENERASE which is equal to a color of −1. To find out the pencolor, use ITEM 4 DRAWSTATE. Abbreviated: PC.

PENDOWN — Causes the turtle to leave a trail in the current pen color when it moves. This is the default state, and is changed by PENUP. Abbreviated: PD.

PENUP — Causes the turtle to move without leaving a trail or erasing. Abbreviated: PU.

RIGHT — Rotates the turtle. Takes an input that specifies the number of degrees to rotate. Abbreviated: RT.

SETHEADING — Rotates the turtle to the direction specified. Input determines number of degrees. Zero is straight up, with heading increasing clockwise. Abbreviated: SETH.

SETSHAPE — Takes a number 0-7 as input and sets the current sprite's shape to be the shape with that number. Normally sprite 1 has shape 1, sprite 2 has shape 2, etc. Only sprite 0 (the main turtle) can carry the rotating turtle shape, shape 0. See SHAPE.

SETX — Moves the turtle horizontally to the specified coordinate, drawing a line if the pen is down.

SETXY — Takes two numeric inputs. Moves the turtle to the specified point. 0,0 is screen center. As with all primitives or procedures of two or more inputs, when the second input (y-coordinate) is a negative number typed in directly, it must be enclosed by parenthesis.

SETY — Moves the turtle vertically to the specified coordinate, drawing a line if the pen is down.

SHAPE — Outputs the shape number of the current sprite. Shape 0 is stored at VIC sprite location 48, so to find the actual address of a sprite bitmap, use 64*(SHAPE+48). See SETSHAPE.

SHOWTURTLE	Makes the turtle appear. This is the default state, and is changed by HIDETURTLE. Abbreviated: ST.
SINGLECOLOR	Allows the turtle to draw in only one color per 8×8 pixel region. If you draw over a line in a different color, a small section of the line will change color. All 16 colors can be used. Lines drawn in SINGLECOLOR mode are more precise, but the colors are harder to see than in DOUBLE-COLOR mode. SINGLECOLOR is the default mode, remaining in effect until a DOUBLECOLOR command is executed. STAMPCHAR works best in SINGLECOLOR mode. Executing SINGLECOLOR will clear the screen and restore all graphics parameters to default. See DOUBLE-COLOR.
SPLITSCREEN	In graphics mode, gives mixed text/graphics screen, which is the default state. Complementary to FULLSCREEN. Equivalent to function key F3. SPLITSCREEN can take an optional argument specifying how many lines of text to show. Normally five lines of text are shown. To change the lines of text, use the following format: (SPLITSCREEN 10). Parentheses must be used when an input is used with this command. The number of lines of text will remain the same until the number is changed with SPLITSCREEN. The number can be from 0 to 24. Use ITEM 7 DRAWSTATE to find out the current screen mode.
STAMPCHAR	Takes one input, a word containing one letter. Makes the turtle stamp the appearance of that character at its current location in the current pencolor. STAMPCHAR has several .OPTIONs. See .OPTION under Miscellaneous Commands at the end of this section.
TELL	Takes one input, a number 0-7; and tells that sprite to receive all graphics commands until the next TELL command. Sprite 0 is the default, and normally has the rotating turtle shape. By default, sprites other than 0 are hidden (see HIDETURTLE/SHOWTURTLE) and have pens up (see PENUP/PENDOWN). See WHO.
TEXTBG	Takes one input which specifies a new default color for the text screen in the draw and nodraw modes. The change remains in effect until the next TEXTBG command. To find out the current text background color, use ITEM 8 DRAWSTATE. See the section on .OPTION to find out how to change the procedure editor background color. (Note that there is no long name for this command.)
TEXTCOLOR	Takes one input which specifies a new default color for printing characters to the screen. This command does not affect STAMPCHAR; PEN-COLOR must be used instead. Use ITEM 9 DRAWSTATE to find out the current color used for printing characters. See the section on .OPTION to find out how to change the procedure editor character color.

TEXTSCREEN — In graphics mode, gives full text screen. See SPLITSCREEN, FULLSCREEN. Equivalent to the function key F1. Use ITEM 7 DRAWSTATE to find out the current screen mode. TEXTSCREEN has one .OPTION. For a description, see the Miscellaneous Commands at the end of this section.

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.

WHO — Outputs the last input given to TELL. All graphics commands are directed to the sprite with this number. When Logo starts up, sprite 0 carrying the rotating turtle shape is the current receiver of graphics commands.

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 only 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.

4.2 Numeric Operations

+ — Addition

– — Subtraction (two inputs) and negation (one input). Note that negative numbers typed in directly must sometimes have parentheses around them. For example,

```
SETXY 50 -30
```

will be treated by Logo as SETXY 20, since 20 is 50 – 30. To get the correct behavior with procedures or primitives of two or more inputs, use one of the following:

```
SETXY 50 (-30)
SETXY 50 0-30
```

* — Multiplication

/ — Division (always outputs a decimal value).

> — Outputs TRUE if its first input is greater than its second, FALSE otherwise.

<	Outputs TRUE if its first input is less than its second, FALSE otherwise.
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.
BITAND	Takes two inputs, both integers, and outputs their boolean AND. For example, BITAND :N 7 gives the three least significant bits of N.
BITOR	Takes two inputs, both integers, and outputs their boolean OR. BITOR 5 9, for example, outputs 12.
BITXOR	Takes two inputs, both integers, and outputs their boolean eXclusive OR. BITXOR 65535 :N will invert the bottom 16 bits of N. BITXOR -1 :N will invert all 32 bits of N.
COS	Outputs the cosine of its input (an angle in degrees).
INTEGER	Takes one numeric input and outputs the integer part, ignoring the fractional part.
NUMBER?	Outputs TRUE if its input is a number. See also WORD? and LIST?.
QUOTIENT	Outputs the integer quotient of its two inputs. If the inputs are not integers, it first rounds them to the nearest integer. (Note that if the second input is preceded by a minus sign 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) above.
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. Note that if the second input is preceded by a minus sign, it must be enclosed by parentheses.)
ROUND	Outputs the nearest integer to its input.

SIN — Outputs the sine of its input (an angle in degrees).

SQRT — Takes a positive number as input and outputs the square root of that number.

4.3 Word and List Operations

= — 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. (Note that a quoted word must have a space before the =.) If both inputs are lists, compares them to see if their corresponding elements are equal. Outputs TRUE or FALSE accordingly.

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. Note that numbers are kinds of Logo words. Abbreviation: BF. Gives an error when called with the empty word or the empty list.

BUTLAST — If input is a list, outputs a list containing all but the last element. If input is a word, outputs a word containing all but the last character. Note that numbers are kinds of Logo words. Abbreviation: BL. Gives an error when called with the empty word or the empty list.

COUNT — Takes one input. If the input is a word (or number), COUNT outputs the number of characters it contains. If the input is a list, COUNT outputs the number of elements in the list. Note that COUNT counts only the top-level elements of a list. Any list inside the list being counted is regarded only as one element. COUNT could have been written as the following procedure:

```
TO COUNT :THING
  IF EMPTY? :THING OUTPUT 0
  OUTPUT 1 + COUNT BUTFIRST :THING
END
```

EMPTY? — Takes one input and outputs TRUE if the input is the empty word (") or the empty list ([ ]). It outputs FALSE otherwise. EMPTY? could have been written as the following procedure:

```
TO EMPTY? :THING
  OUTPUT ANYOF :THING=[ ] :THING="
END
```

FIRST — If input is a list, outputs the first element. If input is a word, outputs the first character. Note that numbers are kinds of Logo words. Gives an error when called with the empty word or the empty list.

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. If the first input is a list, for example FPUT [A B] [C D], the result will be [[A B] C D]. Note that FPUT is the complementary operation to FIRST and BUTFIRST. FIRST FPUT :A :B will output :A, and BUTFIRST FPUT :A :B will output :B. See also LPUT, LIST, and SENTENCE.

ITEM

Takes two inputs, a number and a word or list, and outputs the Nth element of the list. N is the number given as the first input. For instance, ITEM 1 outputs the first item of a list or the first character of a word specified as the second input to ITEM. An error will occur if the list or word is too short. ITEM could be written as the following procedure.

```
TO ITEM :N :STUFF
  IF EMPTY? :STUFF PR [TOO FEW ITEMS] TOPLEVEL
  IF :N=1 OUTPUT FIRST :STUFF
  OUTPUT ITEM :N-1 BUTFIRST :STUFF
END
```

Following are two procedures suggesting a way to use ITEM.

```
TO PICKRANDOM :STUFF
  OUTPUT ITEM (1+RANDOM COUNT :STUFF) :STUFF
END

TO CLOSING
  OUTPUT PICKRANDOM [[SINCERELY YOURS] [YOURS TRULY]
    [YOUR HUMBLE SERVANT] [CORDIALLY]]
END
```

LAST

If input is a list outputs the last element. If input is a word, outputs the last character. Note that numbers are kinds of Logo words. Gives an error when called with the empty word or the empty list.

LIST

Takes a variable number of inputs (two by default) and outputs a list of the inputs. If the first and second inputs are lists, for example LIST [A B] [C D], the result will be [[A B] [C D]]. See also FPUT, LPUT, and SENTENCE. If there are more than two inputs, there must be an opening parenthesis before LIST, and a closing parenthesis after the last input.

LIST?

Outputs TRUE if its input is a list. See also WORD? and NUMBER?.

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. 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.

MEMBER?

Takes two inputs which may be words or lists. MEMBER? is a predicate (meaning that it returns TRUE or FALSE) telling whether the first input is present in the second input. If the second input is a word, the first input must be a single character word or an error will result. MEMBER? will output TRUE if the single character is contained in the word, and FALSE if it is not. If the second input is a list, MEMBER? will output TRUE if the first input is equal to an element of that list. MEMBER? (without the error checking) could have been defined as the following procedure.

```
TO MEMBER? :THING :PLACE
  IF EMPTY? :PLACE OUTPUT "FALSE
  IF :THING = FIRST :PLACE OUTPUT "TRUE
  OUTPUT MEMBER? :THING BUTFIRST :PLACE
END
```

Example:

```
MEMBER? 3 12345
  RESULT: TRUE
MEMBER? ". 3/2
  RESULT: TRUE
MEMBER? "ALEC [DORIS MARK SEAN RUTH]
  RESULT: FALSE
MEMBER? [HI] [[BYE] [PIE] [SLY] 5 MY [HI]]
  RESULT: TRUE
```

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.

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 closing parenthesis after the last input. (Note that quoted words must be followed by a space, so you must put a space before the final close parenthesis if the last input is a quoted word.)

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?.

4.4 Defining and Editing Procedures

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 of the list is a 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]]]. Due to limitations of the DEFINE command, each sublist in the second input (each procedure line) must be less than 255 characters long. This does not apply to procedures defined with the editor. See TEXT. Note that one normally uses TO rather than DEFINE in order to define procedures.

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. If EDIT is followed by a list of procedures, the procedures in the list will all be available in edit mode. Example: EDIT [SQ TRI BOX] Note that since EDIT does not evaluate its input, a list that is input to EDIT as the value of a variable must be typed as RUN LIST "EDIT :LISTNAME. Edit can also take auxiliary words: ALL, NAMES, PROCEDURES. See section 1.2.2 for a description of keystroke commands inside the editor. Abbreviated: ED.

END

Terminates a procedure definition that is typed in to 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. To actually exit the editor, you must type CTRL-C. Do not include END in arguments to the DEFINE primitive.

ERASE

Erases designated procedure from workspace. Can also take qualifiers ALL, NAMES, PROCEDURES. Signals an error if there is no procedure with the given name. To keep you from accidently running a procedure you intended to erase, the input to erase is not evaluated; to erase a procedure called LOOKUP, type ERASE LOOKUP, not ERASE "LOOKUP. Abbreviated: ER. ERASE can take a list of procedures to erase as input, such as ERASE [SQ TRI]. To erase a list which is a variable, use RUN LIST "ERASE :LISTNAME.

TEXT

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.

TO

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 procedure named by the first input. With no inputs at all, TO enters edit mode with an empty edit buffer.

4.5 Naming

LOCAL

LOCAL takes the name of a variable (a word) as input and makes the variable be local to the current procedure. If the procedure FOO executes LOCAL "X, subsequent MAKE commands in FOO will only affect the value of X within FOO. Normally, MAKE would change the value of X in any calling procedures as well as in FOO, or, if there were no procedure inputs with the name X, it would change the value of the global variable. Just as procedure inputs have no value outside of the procedure, a local variable has no value outside its procedure. In fact, local variables are exactly like procedure inputs except that they have to be given a value with MAKE instead of when the procedure is called. LOCAL is most useful in procedures which require temporary variables, particularly when REQUEST and READCHARACTER are used. For example, the following procedure asks a question and guarantees a response.

```
TO ASK :QUESTION
  LOCAL "ANSWER
  PRINT :QUESTION
  MAKE "ANSWER REQUEST
  IF EMPTY? :ANSWER OUTPUT ASK :QUESTION
    ELSE OUTPUT :ANSWER
END
```

MAKE

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 (thing) of the variable. Here is a simple example:

```
MAKE "MYAGE 21
```

Since MAKE evaluates its first input, you can use it to alter a variable whose name is the result of some expression:

```
TO ARSET :ARRAYNAME :INDEX :VALUE
  MAKE (WORD :ARRAYNAME :INDEX) :VALUE
END
```

THING

Outputs the value of its input, which must be a word. Note that this gives an "extra level" of evaluation. THING "XXX is equivalent to :XXX. Sample use:

```
TO ARGET :ARRAYNAME :INDEX
  OUTPUT THING (WORD :ARRAYNAME :INDEX)
END
```

THING?

Outputs TRUE if its input has a value associated to it. Example:

```
THING? "NUM
FALSE

MAKE "NUM 5
THING? "NUM
TRUE
```

4.6 Conditionals

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.

ELSE

Used in IF ... THEN ... ELSE.

IF

Used in the basic conditional form IF condition THEN action. The condition should be a Logo expression which outputs the word TRUE or FALSE. A Logo variable whose value is TRUE or FALSE or a user procedure which outputs TRUE or FALSE satisfies this condition, as do the various predicates (testing functions) such as <, >, =, LIST?, ALLOF, etc. The action may be any number of Logo commands. If the condition outputs TRUE, then the rest of the commands from the THEN until the end of the line (or an ELSE, see above) are performed. The "THEN" may always be omitted. Here are some sample IF statements:

```
IF HEADING=:OHEADING THEN STOP

IF ALLOF :X=0 :Y=0 PRINT [YOU WON!] STOP

IF WORD? :ITEM PRINT1 FIRST :ITEM PRINT BF :ITEM
```

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
NOT	Outputs TRUE if its input is FALSE, FALSE if its input is TRUE.
TEST	Tests a condition to be used in conjunction with IFTRUE and 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.
THEN	Used with IF ... THEN ... ELSE ...

4.7 Control

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 rarely used in Logo programming.) To break up IF statements into shorter lines, use the TEST and IFTRUE/IFFALSE primitives.
GOODBYE	Clears workspace and restarts Logo. (But it does not clear the user machine-language area.)
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.
REPEAT	Takes a number and a list as input. RUNs the list the designated number of times.
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 top-level command line, 255. Otherwise, an error is signalled.
STOP	Causes the current procedure to stop and return control to the calling procedure. 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 and is never executed.

TOPLEVEL — Aborts the current procedure and all calling procedures and returns control to toplevel (i.e. 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 and gives control back to the user. It is not used very often in Logo programming.

4.8 Input and Output

.CTYO — Takes one input, a number from 0 to 255. .CTYO prints the character with that ASCII number. The command stands for "Character TYpe-Out." .CTYO is most useful for printing character code 0, which normally may not be part of a Logo word. Except for this problem, .CTYO could have been defined as:

```
TO .CTYO :N
  PRINT1 CHAR :N
END
```

ASCII — Takes a character as input and outputs the number that is the ASCII code of that character.

CHAR — Takes an integer as input and outputs the character whose ASCII code is that integer.

CLEARTEXT — Clears the text screen and homes the cursor.

CLEARINPUT — Clears the character input buffer of any typed-ahead characters.

CURSOR — Takes two inputs, column and row, and positions the cursor there. Columns are 0-39, rows are 0-24. 0,0 is upper left. See the CURSORPOS primitive to find out how to determine the cursor's current position.

CURSORPOS — Takes no inputs. It outputs a list containing the horizontal and vertical position of the text cursor. Note that CURSORPOS outputs a list, but the primitive for setting cursor position, CURSOR, takes two inputs instead of a list with two elements.

FPRINT — Operates exactly like PRINT except that it does NOT strip single quotes when printing words or the outer level of brackets when printing lists. FPRINT is useful when the contents of a list is not a sentence, but is simply a list, which is more readable if the list is printed with the brackets.

Examples:

```
MAKE "COLORS [WHITE BLACK]
PRINT :COLORS
WHITE BLACK

FPRINT :COLORS
[WHITE BLACK]

(PRINT [WHITE BLACK] [ORANGE RED])
WHITE BLACK ORANGE RED

(FPRINT [WHITE BLACK] [ORANGE RED])
[WHITE BLACK] [ORANGE RED]
```

JOYSTICK

Takes a number 0 or 1 as input, which specifies the Control Port. The JOYSTICK primitive outputs the current state of the joystick, which should be plugged into Control Port 2 for JOYSTICK 1 and Port 1 for JOYSTICK 0. (Using port 1 is not recommended because it interferes with the keyboard.) The output is -1 if the stick is centered, or ranges from 0 to 7 (inclusive) if the joystick is pressed in some direction. 0 is forward, 1 is forward and right, and so on clockwise. If the output of JOYSTICK is positive, multiplying it by 45 will give an input suitable for SETHEADING. The JOYSTICK primitive has .OPTIONS. See the Miscellaneous Commands section below. Example:

```
TO PLAY
  MOVEJOY JOYSTICK 1
  IF JOYBUTTON 1 PENDOWN ELSE PENUP
  IF RC? RUN REQUEST ;Run any typed commands.
  PLAY
END

TO MOVEJOY :DIR
  IF :DIR < 0 STOP
  SETHEADING :DIR*45 FD 8
END
```

JOYBUTTON

Takes one input (0 or 1) describing which joystick to check. Outputs TRUE if the joystick button is depressed, and FALSE otherwise.

NOPRINTER

Terminates PRINTER mode. See PRINTER.

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. In practice, some paddles will return a slightly different range. Only the upper limit will vary in such cases; for example, your paddles might return a range from 0 to 170.
PADDLEBUTTON	Takes a number 0 through 3 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 THEN CLEARSCREEN.
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 PRINTOUT) Abbreviated: PR.
PRINT1	Like PRINT, but does not terminate output line with a return. With multiple inputs, does not print spaces between elements.
PRINTER	PRINTER selects the Commodore Serial Graphics printer for output. The inputs to PRINT, PR, PRINT1, and FPRINT will be printed on the printer instead of on the screen, as will error messages. NOPRINTER terminates this mode. The PRINTER command has various .OPTIONS. See the Miscellaneous Commands section below.
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.
READCHARACTER	Outputs the least recent character in the character buffer, or if empty, waits for an input character. See CLEARINPUT, and section 2.3. See the explanation of the INSTANT program on the utilities disk for an example of its use. Abbreviated RC.
REQUEST	Waits for an input line to be typed by the user and terminated with RETURN. Outputs the line (as a list). Abbreviated: RQ.

SETDISK — Takes one input which must be a number, and uses the disk with that device number. The default is 8. SETDISK 9 is popular in two-disk systems.

4.9 Filing and Managing Workspace

BLOAD — Takes the name of a file as input and loads it into memory at the start address specified when the file was saved. The exact filename given as input is used; nothing is added as with READ and READPICT. This primitive is useful for loading files of sprite shapes and assembly language routines.

BSAVE — Takes three inputs: the file name, the address of the start of the region to be saved, and the address of the end. The end address must be greater than the start address. The following procedure will save the indicated range of sprite shape numbers (inclusive):

```
TO SAVESHAPES :FILE :START :END
  BSAVE WORD :FILE ".SHAPES
    3072+:START*64 3072+:END*64+63
END
```

The following line will save the definitions of sprites 6 and 7 in a file called FROGS.SHAPES, suitable for reloading with the READSHAPES procedure on the Utilities Disk.

```
SAVESHAPES "FROGS 6 7
```

CATALOG — Prints the names of files on the disk currently in the disk drive.

DOS — Takes one input (a list), and interprets it as commands to DOS. All Commodore disk commands which (in BASIC) start with PRINT#15 may be used with the DOS command. The differences are that the OPEN command is unnecessary and the PRINT#15 is left off of the command. Following are examples of the commands.

```
DOS [N0:LOGO FILES,01]
DOS [C0:GAMEBACKUP.LOGO=0:GAME.LOGO]
DOS [R0:TEST.LOGO=TSET.LOGO]
DOS [S0:JOESFILE.LOGO]
```

The format is that the letter representing the command comes first, followed by a zero and a colon. Next comes a file name, and what follows this varies for each command. Following is a short description of each command.

N for NEW formats a disk. The name in this case, LOGO FILES, is not a file name but the name of the disk. It is followed by a comma and any 2 digit number.

C for COPY makes a copy of a file on the same disk. First give the new file's name and then the old file's name using the format shown above.

R for RENAME renames a file on the disk. Give the new name then the old name of the file as in the example above, where TSET.LOGO is the old name.

S for SCRATCH which deletes a file. For a more detailed explanation of these commands, see the Disk Commands section in chapter 4 of the 1541 manual.

ERASEFILE — Removes from the disk a file saved with SAVE. Takes file name as input, which must begin with a " mark.

ERASEPICT — Removes a picture that has been stored on the disk using SAVEPICT. Takes 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 ERNAME TEMP is typed, TEMP is assumed to be a procedure and Logo tries to run it. See LOCAL.

POTS — Prints out the titles (names) of all the procedures in the workspace. It is a short form for PRINTOUT TITLES.

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. If given a list of procedure names as input, prints out the text of the procedures in the list. 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. To print out a list which is a variable, use:

```
RUN LIST "PO :LISTNAME
```

Abbreviated: PO. POTS is an abbreviation for PRINTOUT TITLES. (See also PRINT.)

READ

Takes a file name as input and reads a file of Logo procedures from disk. Destroys any graphics display. If the file makes a variable called START-UP which is a list, the contents of the list are RUN when the file is finished loading. Note that READ (like most Logo primitives) evaluates its input, so you must put a quotation mark before the filename:

```
READ "SPACEGAME
```

See the Miscellaneous Commands section below for the .OPTION associated with READ.

READPICT

Reads a picture that has been stored on disk and displays it on the graphics screen. Takes picture name as input.

SAVE

Saves the contents of the workspace on disk. Destroys any graphics display. Takes a file name as input. You may give SAVE a second input, which is a list of procedures to save. Note that in the two-input case, the whole command and inputs must be enclosed in parentheses.

```
SAVE "MYFILE
(SAVE "NEWFILE [SQ TRI PENT])
```

See the Miscellaneous Commands section below for the .OPTION associated with SAVE.

SAVEPICT

Save on disk the picture on the screen. Takes picture name as input. Note that since the input is evaluated, you must begin the filename with a quotation mark. Example:

```
SAVEPICT "HOUSE
```

4.10 Debugging

CONTINUE

Resumes execution after a PAUSE or CTRL-Z. Abbreviated: CO.

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, or aborted with TOP-LEVEL or CTRL-G.

NOTRACE

Turns off tracing.

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 space will continue execution. See CTRL-W, section 1.1.3.

4.11 Miscellaneous Commands

.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 0.768. 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. (There is a problem with using values of .ASPECT that are too different from 0.768: Although lines will be drawn at the correct angle, the turtle pointer may not always appear to be pointing exactly along the line.)
.CALL	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 section 3.2.
.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". (Actually, no version of Logo for any microcomputer yet implements GCTWA, so the storage is never recovered.) 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.
.DEPOSIT	Takes two numeric inputs, an address and a value, and deposits a byte of data at the designated memory location. See section 3.1. See below for an explanation of the .OPTION associated with .DEPOSIT.
.EXAMINE	Takes one input. Outputs the value of the byte at the specified address. See section 3.1. See below for an explanation of the .OPTION associated with .EXAMINE.
.GCOLL	Forces a garbage collection.
.NODES	Outputs the number of currently free nodes. To obtain a true count of free memory, type .GCOLL before typing .NODES.

.OPTION

The .OPTION primitive is provided to alter the behavior of the built-in Logo primitives. It takes three inputs. The first input is the name of the primitive to alter, the second is which behavior to alter, and the third is the new value for the option. Sometimes the options are only loosely associated with the primitives, but there they are. See section 4.12 for a full discussion of the various options.

.SPRINT

Takes one input, a shape number, and prints the definition of that shape on the textscreen using balls and dots. This primitive is used mainly by the sprite shape editor.

Causes the rest of the line not to be evaluated. Useful for including comments in procedures and procedure titles.

4.12 The .OPTION primitive

The .OPTION primitive allows you to alter the operations of some primitives.
Here is a full list of the primitives and their various options:

DRAW

.OPTION "DRAW 0 :N

Sets the default background color for turtle graphics to N. This color, which normally defaults to Medium Gray (11), will be used when Logo leaves NODRAW mode and enters DRAW mode, and when switching between SINGLECOLOR and DOUBLECOLOR.

.OPTION "DRAW 0 2

will set the default graphics screen color to red. To change the background color while turtle graphics is in use, use the BACKGROUND command as normal. This option is provided for users who decide a particular color is better for their television than grey 11. Blue 6 is a frequent choice for color TVs, but looks abominable on Black&White TVs. The SINGLECOLOR and DOUBLECOLOR primitives will reset the background color to default color specified with this .OPTION.

.OPTION "DRAW 1 :N

sets the default pen color used by DRAW. DRAW will always restore the pencolor to this value when coming from NODRAW mode or switching between SINGLECOLOR and DOUBLECOLOR. It defaults to 1 (white).

.DEPOSIT, .EXAMINE

.OPTION ".DEPOSIT 0 :N, .OPTION ".EXAMINE 0 :N

N=2 Kernel and character set 4K ROM instead of I/O.
N=4 Logo stack and workspace — 64K Ram. No I/O space
N=6 Kernel and I/O. This is the default. See section 3.1 for a fuller description of these various memory map modes.

EDIT

The .OPTIONs for EDIT correspond to those for DRAW:

.OPTION "EDIT 0 :N — sets the background color for the editor (default setting is 11)

.OPTION "EDIT 1 :N — sets the text color for the editor (default setting is 1)

JOYSTICK

.OPTION "JOYSTICK 0 :N

If N is 1, JOYSTICK will output a number which is the sum of the switch values. This mode is documented in the Commodore Programmer's Reference Guide. The default for N is 0.

PRINTER

Changes to these options take effect at the PRINTER or NOPRINTER commands, and at the end of each line sent to the printer.

.OPTION "PRINTER 0 :N

sets the printer serial device number (normally 4) to :N. Other device numbers are useful for different peripherals.

.OPTION "PRINTER 1 :N

sets the secondary address for PRINTER. This is normally 0, which means that the Commodore VIC-1525E printer should use the uppercase/graphics character set. If you call this primitive with N being 7, the printer will be set up in upper/lowercase mode. Other peripheral devices may have different uses for different secondary addresses.

.OPTION "PRINTER 2 :N

controls whether printout directed to the printer is also printed to the screen. The default value, 0, causes printout to be echoed to the screen; setting the option to 1 inhibits this behavior.

RC

.OPTION "RC 0 :N — directs whether "interrupt characters" will be processed immediately (as normal, :N = 0) or treated as normal characters, available for typing and readable by RC and READCHARACTER. The action of each interrupt character listed below is controlled by one bit in this option. If the bit is 0, then the interrupt character has its usual significance. If the bit is 1, then the character is treated like any other control character.

character	number
CTRL-G	1
CTRL-Z	2
CTRL-W	4
f1	8
f3	16
f5	32

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. For example, this option may be used to inhibit CTRL-G while output is directed to the printer or other special output device.

Note that this option affects only the action of interrupt keys, that is, the keys which perform actions even while a procedure is running. It does not alter the behavior of the editing keys.

Use the following procedures for disabling and re-enabling the interrupt characters. To enable or disable several keys at once, sum their values from the above table and use the result as input to the DISABLE or ENABLE procedures.

```
TO DISABLE :KEYBIT
  .OPTION "RC 0 BITOR (.OPTION "RC 0) :KEYBIT
END

TO ENABLE :KEYBIT
  .OPTION "RC 0 BITAND (.OPTION "RC 0) 255-:KEYBIT
END
```

READ, SAVE

.OPTION "READ 0 :N — N=0 is the default setting, for normal behavior. N=1 The file will be read into the editor but will not be evaluated. This setting is usually used when the editor is employed as a text editor. The TEXTEDIT utilities file utilizes this option.

.OPTION "SAVE 0 :N — N=0 is the default setting, for normal behavior. N=1 The edit buffer is saved instead of the workspace. This allows the editor to be used as a text editor. See the utilities file TEXTEDIT. Normally, SAVE will clear the edit buffer, load the contents of the workspace into the buffer, and then save it. One trick you can do with N set to 1 is to type EDIT ALL. This will bring the contents of the workspace into the buffer. Then commands can be inserted between procedures such as PRINT [THE FILE IS HALFWAY LOADED.]. Leave the editor with CTRL-G so it will not be evaluated and SAVE the file.

STAMPCHAR

.OPTION "STAMPCHAR 0 :N

sets the method used for putting the character on the screen. N=0 means write over the previous contents of the 8×8 area where the character goes. This means the old character will disappear and be replaced by the new character. This is the default. N=1 means combine by overwriting. If two letters are printed in the same 8×8 area, they will be illegible; however, this mode works best for combining turtle graphics and character graphics. N=2 means combine by reversing whichever points are on in the character shape. This allows you to write over graphics and then erase the printing and restore the graphics later by stamping the same character in the same place.

.OPTION "STAMPCHAR 1 :N

controls which character set is used. N=0 means use uppercase/graphics (the default). N=1 means use upper/lowercase.

.OPTION "STAMPCHAR 2 :N

controls whether characters are displayed in reverse video. N=0 means no (the default), and N=1 means yes.

TEXTSCREEN

.OPTION "TEXTSCREEN 0 :N

controls which sprites are displayed on both the full text screen and the text portion of the SPLITSCREEN display.

The option for TEXTSCREEN controls which sprite objects are displayed on the text screen. Sprites 0-7 are by default never displayed. Each sprite is represented by a bit in the number which is set by this option. The least significant bit is sprite 0, as described in Commodore 64 Programmer's Reference Guide.

Note that which sprites are displayed on the graphics screen is controlled by their individual HIDETURTLE/SHOWTURTLE states.

Index

" G-22, C-5, W&L-24
' W&L-43
: G-37, C-5, W&L-47
+ G-4, C-1, A-143
− C-1, A-143
* G-4, C-1, A-143
/ G-4, C-1, A-143
( ) C-2
< > B-8
> G-50, A-143
< G-50, A-144
= G-48 to G-49, A-145
? B-7, G-64
[ ] C-9, W&L-44
; G-61, W&L-16, A-159
! A-114

.ASPECT, A-118, A-158
.CALL, A-127, A-158
.CONTENTS, A-124, A-158
.CTYO, A-124, A-152
.DEPOSIT, A-126, A-158, A-159
.EXAMINE, A-126, A-158, A-159
.GCOLL, A-125, A-158
.NODES, A-158
.OPTION, A-159 to A-162
.SPRINT, A-159

Abbreviations, G-4, G-18
Abelson, Harold, B-1, B-3
ABS, C-19 to C-20
Absolute value, C-19 to C-20
Addition, G-4, C-1
ADDRESSES, A-109, A-129
Addresses, useful, A-125, A-134
ADVENTURE, A-104
ALLOF, W&L-55, W&L-84, A-150
AMODES, A-109, A-128
ANIMAL program, A-104
ANIMAL.INSPECTOR program, A-105
ANIMALS, S-2 to S-3, S-14
ANYOF, W&L-55, W&L-84, A-150
Arcs, G-43, A-60 to A-62, A-94 to A-95
ARCL, A-94
ARCR, A-94
Arithmetic, G-4, G-35, C-1
Arrow keys, B-9, G-4, G-12, A-11
ASCII, W&L-44, A-70, A-124, A-152
Aspect ratio, A-118
Assembler/Logo interfacing, A-126
ASSEMBLER, A-109, A-128
ASSORTED, S-2, S-14
ATAN, A-144
Attack/Decay, M-8

B&W, A-95
BACK, G-2, A-139
BACKGROUND, G-7 to G-8, A-118, A-139
BASE, A-95, A-95
BF, W&L-31
BG, G-7 to G-8, A-118, A-139
Binary files, A-131
Binary tree, G-54, A-49 to A-51
BITAND, A-144
BITOR, A-144
BITXOR, A-144
BK, G-2
BL, W&L-31
Blank disk, preparing for use, B-5
BLOAD, A-155
Brackets < >, B-8
BSAVE, A-155
Bugs, G-15, A-1
BUTFIRST, W&L-31, W&L-70, A-68, A-145
BUTLAST, W&L-31, W&L-70, A-145

CATALOG, G-21, G-23, A-120, A-155
CCHANGE, A-96, A-131 to A-134
Changing inputs, G-46

CHAR, W&L-44, A-70, A-124, A-152
CIRCLEL, A-94
CIRCLER, A-94
Circles, G-42 to G-43, A-94
Clear Key, G-6, W&L-23, A-115
Clearing the workspace, G-23 to G-25
CLEARINPUT, A-152
CLEARSCREEN, G-19, G-20, A-139
CLEARTEXT, W&L-23, A-152
<CLR> Key, G-6, W&L-23, A-115
CO, G-59, A-113
Color, G-7 to G-8, G-55 to G-57, A-118, A-119, A-131
COLORS, A-96
Comments, G-61, W&L-16
Commodore Key, S-19, A-122
Computation, B-4, C-1
Conditional, G-48 to G-50
CONTINUE, G-59, A-113, A-157
Control commands, See <CTRL>
Copying a procedure, G-34
COS, C-3, C-4, C-15, C-17, A-144
Cosine, C-15, C-17
COUNT, A-72, A-145
<CRSR> Key, B-9, S-13, A-115
CS, G-19 to G-20
<CTRL> key, B-8
<CTRL> A, G-30 to G-31, A-11, A-115
<CTRL> B, A-11, A-116
<CTRL> C, G-11, G-14, A-13, A-111, A-116, A-124
<CTRL> D, G-30 to G-31, A-11, A-116
<CTRL> F, A-11, A-116
<CTRL> G, B-9, B-10, G-11, G-14, G-46, A-13, A-111, A-113, A-116
<CTRL> K, G-30 to G-31, A-12, A-116
<CTRL> L, A-11, A-116
<CTRL> N, G-30 to G-31, A-11, A-116
<CTRL> O, G-30 to G-31, A-12, A-116
<CTRL> P, G-4 to G-5, G-30 to G-31, A-11, A-116
<CTRL> W, G-32 to G-33, A-113, A-124
<CTRL> Z, G-59, A-113, A-124
Current sprite, S-2
Cursor, B-7, G-14, A-11
CURSOR, A-152
Cursor Key, B-9, S-13, A-115
CURSORPOS, A-152
Curves, G-43

D, G-65
Debugging, G-15, G-57 to G-59, G-61
Decay, M-8
DEFINE, A-148
<DEL>, B-9, G-12 to G-14, A-12, A-115
Demonstration programs, sprite, S-15 to S-20
DINOSAURS, S-15 to S-17
diSessa, Andrea, B-1, B-3
Disk, backup of utilities, A-88
Disk files, A-119
Disk preparation, B-5, A-88, A-155
Division, G-4, C-1
DOS, A-155
Dots, G-37, C-5
DOUBLECOLOR, G-8, A-119, A-139
DRAW, G-1, G-3, G-5, C-15, C-16, A-139, A-159
DRAWSTATE, G-62, A-67 to A-68, A-139
Draw mode, G-1, A-112
Driving the turtle, G-2
DROVE, A-59
Duration, M-2 to M-3
DYNATRACK, A-106

EDIT (ED), A-101 to A-103, A-111, A-148, A-160

Edit mode, G-11 to G-14, A-11 to A-13, A-111
Editing, A-113
Editing commands, summary, G-31, A-11 to A-13
Editor, G-30 to G-31
Editor, sprite, S-10 to S-13
EDSH, S-10 to S-11
Elephant mascot, B-2, A-55 to A-56
Ellipse, C-15, C-20 to C-21
ELSE, W&L-55, A-150
Empty list, W&L-44 to W&L-45
Empty word, W&L-44
EMPTY?, W&L-17, A-145
END, G-11, G-16 to G-17, C-6, A-148
Envelopes, M-8
ER, G-23 to G-25
ERASE, G-23 to G-25, A-148
ERASE ALL, G-23 to G-24
ERASEFILE, G-23, A-120, A-156
ERASEPICT, G-25 to G-26, A-121, A-156
Erasing, G-8 to G-9
Erasing pictures, G-25 to G-26
ERNAME, A-156
Error messages, B-8, W&L-49, A-1
Errors, typing, B-9
Exclamation point, A-114
Executing a procedure, G-15
EXPONENT, C-12 to C-15
Exponentiation, C-12 to C-15

F, G-65
FALSE, G-48 to G-50, W&L-55, A-145
FD, G-2 to G-4
Files, G-21, G-24, A-119
FIRST, W&L-31, W&L-70, A-145
FPRINT, W&L-43, A-152
FOR, A-97
For-Next loop, A-97
FORWARD, G-2 to G-4, A-140
Formatting a disk, B-5
FPUT, W&L-18, W&L-68, A-146
Function Keys, G-19, A-112
Function Key <f1>, G-5 to G-6, A-112, A-113, A-124, A-143
Function Key <f3>, G-5 to G-6, A-112, A-113, A-124, A-142
Function Key <f5>, G-5 to G-6, A-112, A-113, A-124, A-140
Functions, C-3 to C-4
FULLSCREEN, G-5 to G-6, A-112, A-140

Global variables, C-5 to C-6, W&L-9 to W&L-12
Glossary of Logo primitives, A-137 to A-162
GO, A-151
GOODBYE, G-23, G-24, A-151
GRAMMAR, A-106
Graphics, B-3, G-1, A-14 to A-63
Graphics commands, summary, G-4
Graphics mode, G-1
Graphing functions, C-15 to C-21

Heading, G-33 to G-34
HEADING, G-62 to G-63, A-140
HIDETURTLE, G-26, C-15, C-16, A-140
Hierarchy of operations, C-1 to C-3
History lists, W&L-71 to W&L-75
HOME, G-19 to G-20, C-15, C-16, A-115, A-140
<HOME> Key, A-12
HT, G-26, C-15, C-16

IF, G-48 to G-50, W&L-55, A-150
IFFALSE (IFF), W&L-68, A-151
IFTRUE (IFF), W&L-68, A-151
IMMEDIATE mode, B-3, G-11 to G-12
Initializing a disk, B-5
Input, G-36, G-55
Input, changing, G-46

Inputs, negative, G-59
INSPI, A-107
INST Key, W&L-44, A-115, A-123
INSTANT, B-3, G-64 to G-67, W&L-3, W&L-73
INTEGER, C-3 to C-4, A-144
Integer, C-1
Integer operators, C-3 to C-4
Intelligent language interpretor, W&L-80 to W&L-87
Interpretive language, B-3
Interrupt characters, A-124
Invisible turtle, G-26
ITEM, W&L-35, A-146

JOY, A-97
JOYBUTTON, A-116, A-153
JOYSTICK, A-116, A-153, A-160

Keyboard, B-6

Language Disk, B-1
LAST, W&L-31, W&L-70, A-146
LEFT, G-2 to G-4, A-140
Levels of execution, W&L-49
Line length, A-124
LIST, W&L-68 to W&L-69, A-146
LIST?, W&L-56, W&L-67, A-146
Lists, C-9, W&L-42
Listing a procedure, G-32
Listing: Summary of commands, G-33
LOCAL, C-5, C-6, C-7 to C-10, A-75 to A-76, A-78, A-149
Local variables, C-5 to C-6, W&L-9 to W&L-12
LOG, A-97
LPUT, W&L-18, W&L-68, W&L-71, A-146
LT, G-2 to G-4

Mad-Libs, W&L-75 to W&L-79
Magic number, G-34
MAKE, C-5 to C-6, W&L-7, A-149
Manual, Tutorial, B-1
Mascots, B-2, A-55 to A-60
MEMBER?, W&L-17, W&L-56, A-147
Memory addresses, A-134
Memory organization chart, A-125
Messages, error, A-1
Mindstorms, Seymour Papert, B-1
Mode, DRAW, G-1, G-3, C-15, C-16, A-112
Mode, EDIT, G-11 to G-14, A-11 to A-13, A-111
Mode, IMMEDIATE, B-3, G-11 to G-12
Mode, NODRAW, G-5, G-6, A-111
Multiplication, G-4, C-1
Music, M-1 to M-10

N, G-65
Naming, G-10, G-21, G-22, G-37, G-39
Negative inputs, G-59
ND, G-5, G-6
NODRAW, G-5, G-6, A-111, A-140
Nodraw mode, A-111
NOPRINTER, A-153
NOT, W&L-55, A-64, A-68, A-151
Notation, standard music, M-4
NOTRACE, G-51, G-60, A-157
NOWRAP, G-46, G-47 to G-48, A-141
NUMBER?, W&L-12, A-64, A-144
Number limits, A-124
Numeric operations, C-1 to C-2

Object, W&L-27, W&L-65
OP, C-10 to C-14, W&L-26, W&L-31, W&L-50

OPCODES, A-109, A-128
Operations, C-1 to C-2
Operators, C-1, C-3 to C-4
Options, A-159 to A-162
Output, G-55, C-3 to C-4
OUTPUT, C-10 to C-14, W&L-26, W&L-31, W&L-50 to W&L-51, A-151
Overview, B-2

P, G-65
Package, Commodore Logo, B-1
PADDLE, A-116, A-154
PADDLEBUTTON, A-116, A-154
Parabola, C-15, C-18 to C-20
Papert, Seymour, B-1
Parentheses, B-8, W&L-41, W&L-53, W&L-66
PAUSE, G-59, A-157
PC, G-7, G-55, A-118
PD, G-19 to G-20
PENCOLOR, G-7, G-55, A-118, A-141
PENDOWN, G-19 to G-20, A-141
PENERASE, G-8, A-118
PENUP, G-19 to G-20, A-141
Peripherals, A-116
Pictures, printing, A-98, A-117
Pictures, saving on disk, G-25, A-120
PIG, A-108
Pitch, M-4 to M-5
Planning a procedure, G-16 to G-18, G-36 to G-37
PLAY, M-4
PLOTTER, A-98
PO, G-23, G-32, A-112
PO ALL, G-32
Pointed brackets, B-8
POLY, G-40 to G-42
POTS, G-21, G-23, A-156
PR, G-57 to G-59, W&L-7
Predicates, W&L-56 to W&L-57
Primitive, B-3, G-9 to G-10
PRINT, G-57 to G-59, W&L-7, W&L-66, A-154
PRINT1, W&L-66, A-64, A-154
PRINTER, A-117, A-154, A-160
Printers, A-116
PRINTFILE, A-103
Printing of Saved Pictures, A-98, A-117
PRINTOUT (PO), G-32, A-112, A-156
PRINTPICT, A-98, A-117
PRINTTEXT, A-103
Priority, sprite, S-17
Procedural language, B-2
Procedure, B-2, B-3, G-9 to G-10
Procedure copying, G-34
Procedure writing, G-9 to G-14
Procedure naming, G-10
Procedure saving on disk, G-21 to G-23
Procedures, A-16 to A-18
Procedures that take inputs, G-36
Projects: changing inputs, G-48, A-39 to A-43
Projects: curves, G-43, A-36 to A-38
Projects: history lists, W&L-75
Projects: language understanding, W&L-86 to W&L-87
Projects: Mad-Libs, W&L-79 to W&L-80
Projects: MAKE, W&L-12
Projects: more shapes, G-35, A-30 to A-31
Projects: PLURAL, W&L-59 to W&L-61
Projects: predicates, W&L-57
Projects: procedure, G-21, A-16 to A-19
Projects: RANDOM, G-57, A-53 to A-54
Projects: RC, W&L-6, W&L-16
Projects: recursion, G-54, A-45 to A-52

Projects: recursion, W&L-38
Projects: regular polygons, G-41, A-34 to A-35
Projects: REQUEST, W&L-63
Projects: shapes, G-31, A-19 to A-29
Projects: simple recursion, G-46, A-38 to A-39
Projects: sizeable shapes, G-39, A-32 to A-33
Projects: testing and stopping, G-51, A-43 to A-45
Projects: turtle driving, G-6, A-14 to A-16
Prompt, B-7
PU, G-19, G-20
PULSE, M-9

Quiz Programs, W&L-61 to W&L-65
Quotation marks, A-112
QUOTIENT, C-3 to C-4, A-144

R, G-65
Rabbit, B-2, A-55, A-56
Random numbers, G-55 to G-57
RANDOM, G-55 to G-57, C-3, A-144
RANDOMIZE, C-3, A-144
RC, W&L-3, M-5, A-63, A-124, A-160
RC?, W&L-13, W&L-56, A-154
READ, G-23 to G-24, A-120, A-157, A-161
READCHARACTER, W&L-3, M-5, A-63, A-124, A-154, A-160
READPICT, G-25 to G-26, A-120, A-157
READTEXT, A-103
Real numbers, C-1
Recalling lines, G-4
Recovery process, B-8
Recursion, G-45 to G-54, C-12 to C-14, W&L-35, W&L-74 to W&L-76, W&L-84
Recursion projects, G-46, G-54
Recursive designs, G-54, W&L-84, A-109
Release, M-9
REMAINDER, C-3, C-4, A-144
Remarks, G-61
REPEAT, G-19, A-151
Repeating with Up Arrow or <CTRL> P, G-4
Repeating keystrokes, A-135
REQUEST, C-7 to C-10, W&L-18, W&L-61 to W&L-63, A-18, A-61 to A-63
Restarting Logo, B-8
<RESTORE> Key, B-8
RESULT:, W&L-46
<RETURN> key, B-8, G-12, A-11
RIGHT, G-2 to G-4, A-141
ROUND, C-3, A-144
RQ, C-7 to C-10, W&L-18, W&L-61 to W&L-63, A-18, A-61 to A-63
RT, G-2 to G-4
RUN, A-58 to A-60, W&L-71 to W&L-75, A-151
<RUN/STOP> Key, B-8, A-13, A-111
RUNNER, S-2, S-19 to S-20
Running a procedure, G-15 to G-16

SAVE, G-21, G-22, A-119, A-157, A-161
SAVEPICT, G-25 to G-26, A-117, A-120, A-157
SAVETEXT, A-103
Saving pictures, G-25 to G-26, A-58 to A-60, A-120
Saving procedures, G-21 to G-23
Saving sprite shapes, S-14
Saving text, A-101 to A-103
Scales, M-4
Screen, G-5
Self-starting file, S-15, S-19, A-122

SENTENCE (SE), W&L-18, W&L-31, W&L-65 to W&L-66, A-147
SETDISK, A-155
SETHEADING (SETH), G-62 to G-63, A-141
SETSHAPE, S-8 to S-9, S-10, A-141
SETX, G-63 to G-64, A-141
SETXY, G-63 to G-64, C-15 to C-21, A-141
SETY, G-63 to G-64, A-141
SHAPE, S-9 to S-10, A-141
Shape names, S-14
Shape numbers, S-14
SHAPES, S-2, S-14
<SHIFT> Key, B-10
<SHIFT-CLR> Key, G-6, W&L-23, A-115
<SHIFT-INST> Key, A-115, A-123
SHOWFILE, A-103
SHOWTEXT, A-103
SHOWTURTLE, G-26, A-142
SIN, C-3, C-4, C-15 to C-17, A-145
Single quote, W&L-43
Sine, C-15 to C-17
SING, M-5
SINGLECOLOR, G-7, A-119, A-142
SMALLX, S-7 to S-8
SMALLY, S-7 to S-8
Snail, B-2, A-55, A-57
SNOW, A-108
SOUND, M-9 to M-10
Spaces in Logo lines, G-2, G-5, G-13
Special effects, G-55
Special Technology for Special Children, B-1
SPLITSCREEN, G-5, A-112, A-142
SPRED, S-10
Sprite demonstration programs, S-15
Sprite editor, S-10 to S-13
Sprite priority, S-17
Sprite shapes, S-7
SPRITEDEMOS, S-3, S-5, S-15
Sprites, B-3, S-1
Square, G-16 to G-18
SQRT, C-3, C-4, A-145
SSH, M-2
SSHER, M-2 to M-3
ST, G-26
STAMPCHAR, G-27, A-142, A-162
STAMPER, A-99
STAMPFD, A-99
Starting Logo, B-6 to B-7
Starting Logo summary, B-11
STARTUP, S-15, S-19, A-122
State, G-33
Status line, doublecolor, A-119
STOP, G-48 to G-51, W&L-3, A-151
STOP Key, B-8, A-13, A-111
STOPPED!, G-14
Storage, A-124
Structured programming, B-3
SUBMARINE, S-17 to S-19
Subprocedures, G-43 to G-45
Subtraction, G-4, C-1
Summary: commands with keyboard versions, G-19
Summary: editing commands, G-31
Summary: listing commands, G-33
Summary: Logo commands used so far, G-28 to G-29
Summary: sprite editing commands, S-13
Summary: starting Logo, B-11
Summary: turtle commands, G-4
Summary: Words and Lists Primitives, W&L-39, W&L-41
Superprocedure, G-43
Sustain/Release, M-9
System commands, B-3

Tangent, C-15, C-17 to C-18
TB?, S-5, S-21
TEACH, A-100 to A-101
TELL, S-1 to S-3, A-142
Templates, W&L-82 to W&L-87
TEMPO, M-3

TEST, W&L-67, A-151
Testing: IF-THEN-ELSE, G-48 to G-51, W&L-55
TET, G-54, A-109
TEXT, A-148
Text editor, using Logo as, A-101 to A-103
TEXTBG, G-26, A-142
TEXTCOLOR, G-26, A-142
TEXTEDIT, A-101 to A-103
TEXTSCREEN, G-5, A-113, A-143, A-162
THEN, G-48 to G-51, W&L-3, W&L-55 to W&L-56, A-151
THING, W&L-59, A-150
THING?, A-150
TO, G-11, C-6 to C-7, A-101 to A-103, A-111, A-149
TOPLEVEL, W&L-3, A-152
Total Turtle Trip Theorem, G-35
TOWARDS, G-62 to G-63, A-143
TRACE, G-51, G-61, A-113, A-157
Tree, G-54, A-49 to A-51
TRUE, G-48 to G-51, W&L-55, A-145
TS?, S-21
Tune blocks, M-6
Turtle, G-2
Turtle commands, G-4
Turtle driving projects, G-6, A-14
Turtle geometry, Abelson & diSessa, B-1, B-3
Tutorial Manual, B-1
TWINKLE, M-6 to M-7
Typing errors, correcting, B-9

U, G-65
UNTIL, A-104
Utilities Disk, B-1, A-88
Utilities Disk, backup, A-88
Utilities Disk files: summary, A-90
Utilities Disk files: explanation, A-94
Utilities Disk: use, A-88
Utilities: writing your own, G-55

Value, W&L-27
Variables, G-36 to G-39, C-5 to C-6, W&L-46 to W&L-48
Variables, global - See Global variables
Variables, local - See Local variables
VEHICLES, S-2, S-14

WAIT, S-21
Waveforms, M-8
WHILE, A-104
WHO, S-4, A-143
Wild card, W&L-82
WORD, W&L-31, A-147
WORD?, W&L-56, W&L-67, A-147
Word processor, using Logo as, A-101 to A-103
Words, W&L-42
Words and Lists, B-4,
Workspace, G-21, G-23
Workspace, clearing, G-23 to G-25
WRAP, G-46 to G-48, A-143
Writing a procedure, G-9 to G-14, C-6 to C-7

XCOR, G-63 to G-64, A-143
Xqpsnpfltk, W&L-81

YCOR, G-63 to G-64, A-143

Zero vs. letter O, G-13

If your diskette becomes damaged or worn out...

Commodore realizes that your original Commodore software diskette may become damaged or worn out through continued use... so we're making it possible to exchange your damaged diskette for a new one.

How to exchange your diskette

If your software diskette becomes damaged or worn out, or for any reason won't load your program, send the defective diskette back to Commodore with a check or money order for five dollars ($5.00) and we'll send you a replacement diskette by return mail within 10 days. NOTE THAT ONLY ORIGINAL COMMODORE PROGRAM DISKETTES SOLD WITH THE COMMODORE SOFTWARE PRODUCT MAY BE EXCHANGED UNDER THIS POLICY.

Return this form with your damaged diskette and check for $5.00 to:

Commodore Diskette Replacement
Commodore Software
1200 Wilson Drive
West Chester, Pennsylvania 19380

Diskette exchange form – PLEASE PRINT CLEARLY

Name of Program________________________________

Computer Model________________________________

Dealer's Name________________________________

City________________ State______ Zip______

Owner's Name________________________________

Address________________________________

City________________ State______ Zip______

Phone________________

Commodore Business Machines, Inc.
1200 Wilson Drive • West Chester, PA 19380

Commodore Business Machines, Limited
3370 Pharmacy Avenue • Agincourt, Ontario, M1W 2K4