



*Apple® Logo II
Reference Manual*

The Apple II



*Compatible With
Ile (128K), Ilc*

Free Replacement

If you purchase Apple software or manuals that have physical defects, or even if you damage them yourself, Apple will replace them at no cost to you. This offer is good for two years after the last date that Apple continues to sell that product to its dealers. See your local authorized Apple dealer for details about the Apple Media Exchange Program.

Limitation on Warranties and Liability

Even though Apple has tested this manual and software and has reviewed their contents, neither Apple nor its suppliers make any warranty or representation, either express or implied, with respect to this manual or software, their quality, performance, merchantability, or fitness for any particular purpose. As a result, this manual and software are sold "as is," and you the purchaser are assuming the entire risk as to their quality and performance. In no event will Apple or its suppliers be liable for direct, indirect, incidental, or consequential damages resulting from any defect in the manual, even if they have been advised of the possibility of such damages. In particular, they shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering or reproducing these programs or data.

Copyright

This manual and the accompanying software (computer programs) are copyrighted by Apple or by Apple's suppliers, with all rights reserved. Under the copyright laws, this manual and software may not be copied, in whole or in part, without the written consent of Apple, except in the normal use of the software or to make a backup copy. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or lent to another person. Under the law, copying includes translating into another language.

You may use the software on any computer owned by you, but extra copies cannot be made for this purpose. For some products, a multi-use license may be purchased to allow the software to be used on more than one computer owned by the purchaser, including a shared-disk system. Contact your authorized Apple dealer for information on multi-use licenses.

© Logo Computer Systems Inc., 1982, 1984
9960 Cote de Liesse
Lachine, Quebec
H8T-1A1

© Apple Computer, Inc., 1984
20525 Mariani Avenue
Cupertino, California 95014

Apple, the Apple logo, and ProDOS are registered trademarks of Apple Computer, Inc. Simultaneously published in the United States and Canada. All rights reserved.
Reorder Apple Product #A2L4033



The Apple II

Apple Logo II Reference Manual

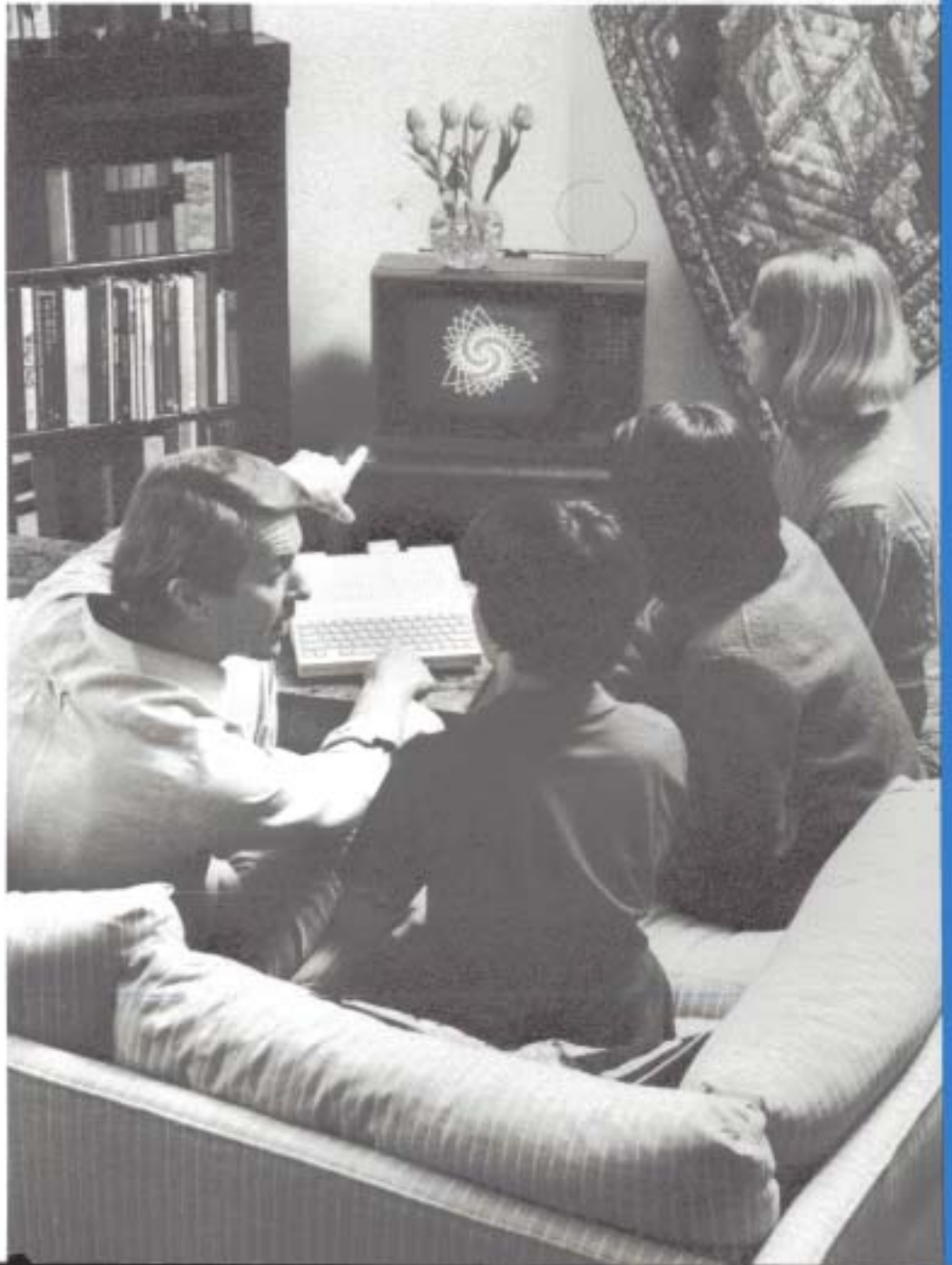


Table of Contents



	List of Figures and Tables	xv
Preface	About This Manual	xix
	xix How to Use This Manual	
	xxi Visual Cues	
Chapter 1	Introduction	3
	3 What You Need	
	4 Getting Help From Logo	
	5 Typing Logo Instructions	
	6 How Primitives Are Described	
Chapter 2	Logo Grammar	11
	11 Procedures	
	13 Punctuation and Inputs to Procedures	
	14 Commands and Operations	
	15 Variables	
	16 Global and Local Variables	
	17 Understanding a Logo Line	
Chapter 3	Defining Procedures With TO	21
	21 TO	
	22 END	

Chapter 4 Using the Logo Editor 26

- 26 How the Editor Works
- 28 Editing Procedures With EDIT
- 29 Typing and Editing in the Editor
- 29 Moving the Cursor
- 30 Inserting and Deleting Text
- 31 Getting Out of the Editor
- 31 Other Ways to Start Up the Editor

Chapter 5 Turtle Graphics 35

- 36 Changing the Turtle's State
- 36 BACK
- 37 CLEARSCREEN
- 37 FORWARD
- 38 HIDETURTLE
- 38 HOME
- 38 LEFT
- 39 RIGHT
- 40 SETHEADING
- 40 SETPOS
- 41 SETX
- 41 SETY
- 42 SHOWTURTLE
- 42 Getting Information About the Turtle's State
- 43 HEADING
- 43 POS
- 44 SHOWNP
- 45 TOWARDS
- 45 XCOR
- 46 YCOR
- 47 Using the Pen and Screen
- 47 CLEAN
- 47 DOT
- 48 FENCE
- 48 FILL
- 49 PENDOWN
- 50 PENERASE
- 50 PENREVERSE
- 51 PENUP
- 52 SETBG
- 52 SETPC
- 53 WINDOW

53	WRAP
54	Getting Information About the Pen and Screen
54	BACKGROUND
54	DOTP
54	PEN
55	PENCOLOR

Chapter 6 **Text and Screen Commands** **59**

60	Primitives Affecting Text on the Screen
60	CLEARTEXT
60	CURSOR
61	FULLSCREEN
61	SETCURSOR
62	SETWIDTH
63	SPLITSCREEN
63	TEXTSCREEN
63	WIDTH
63	Special Control Characters That Change Screen Use
63	CONTROL-L
64	CONTROL-S
64	CONTROL-T

Chapter 7 **Words and Lists** **67**

67	Words: Some General Information
68	Lists: Some General Information
69	Breaking Words and Lists Into Pieces
70	BUTFIRST
71	BUTLAST
71	FIRST
73	ITEM
73	LAST
74	MEMBER
75	Putting Words and Lists Together
76	FPUT
77	LIST
77	LPUT
78	PARSE
78	SENTENCE
80	WORD
81	Examining Words and Lists

81	ASCII
82	BEFOREP
83	CHAR
85	COUNT
85	EMPTYP
87	EQUALP
88	LISTP
88	MEMBERP
89	NUMBERP
90	WORDP
90	Changing the Case of Words
90	LOWERCASE
91	UPPERCASE

Chapter 8	Variables	95
------------------	------------------	-----------

95	Variables: Some General Information
96	EDN
97	EDNS
98	LOCAL
99	MAKE
100	NAME
101	NAMEP
101	THING

Chapter 9	Arithmetic Operations	105
------------------	------------------------------	------------

105	Arithmetic Operations: Some General Information
107	How Logo Evaluates Math Operations
108	Prefix-Form Operations
108	ARCTAN
109	COS
109	DIFFERENCE
110	FORM
111	INT
112	INTQUOTIENT
112	PRODUCT
113	QUOTIENT
113	RANDOM
114	REMAINDER
115	RERANDOM
116	ROUND

116	SIN
117	SQRT
117	SUM
118	Infix-Form Operations
119	Plus Sign
119	Minus Sign
120	Multiplication Sign
121	Division Sign
121	Less Than Sign
122	Equal Sign
122	Greater Than Sign

Chapter 10 **Conditionals and Flow of Control** **125**

125	Flow of Control: Some General Information
126	Using Conditionals
126	IF
127	IFFALSE
128	IFTRUE
128	TEST
129	Interrupting Procedures
130	CO
130	OUTPUT
131	PAUSE
132	STOP
132	WAIT
133	Transferring Control and Repeating Instructions
133	CATCH
135	ERROR
136	GO
137	LABEL
137	REPEAT
138	RUN
140	THROW
140	Debugging Programs
141	STEP
141	TRACE
143	UNSTEP
143	UNTRACE
144	Special Control Characters
144	OPEN APPLE-ESC
144	CONTROL-W
144	CONTROL-Z

Chapter 11	<i>Modifying Procedures Under Program Control</i>	147
	148 COPYDEF	
	148 DEFINE	
	150 DEFINEDP	
	150 PRIMITIVEP	
	151 TEXT	
<hr/>		
Chapter 12	<i>Logical Operations</i>	157
	158 AND	
	159 NOT	
	160 OR	
<hr/>		
Chapter 13	<i>The Outside World</i>	163
	163 Using Paddles	
	163 BUTTONP	
	164 PADDLE	
	164 Making Logo Read Information	
	164 KEYP	
	165 READCHAR	
	166 READCHARS	
	167 READLIST	
	167 READWORD	
	168 Making Logo Write Information	
	169 PRINT	
	170 SHOW	
	170 TYPE	
	171 Making Sounds With TOOT	
<hr/>		
Chapter 14	<i>Managing Your Workspace</i>	175
	176 Sizing Up Your Workspace	
	176 NODES	
	177 RECYCLE	
	177 Printing From the Workspace	
	177 PO	
	178 POALL	
	178 PON	
	179 PONS	

179	POPS
180	POT
180	POTS
180	Erasing From the Workspace
181	ERALL
181	ERASE
181	ERN
181	ERNS
182	ERPS
182	Cleaning and Organizing the Workspace
182	BURY
182	BURYALL
183	BURYNAME
184	UNBURY
184	UNBURYALL
185	UNBURYNAME

Chapter 15 **General File Management** **189**

189	Logo's File System: Some General Information
189	What Is a File?
190	Disk Formatting and Volume Names
190	Disk Organization
192	Accessing Files
194	General File System Primitives
194	CATALOG
195	CREATEDIR
196	EDITFILE
196	ERASEFILE
196	FILEP
197	LOADHELP
197	ONLINE
198	POFILE
198	PREFIX
199	RENAME
199	SETPREFIX

Chapter 16 **Managing Various Files** **205**

206	Working With Program Files
206	LOAD
206	SAVE
207	SAVEL

207	Working With Picture Files
208	LOADPIC
208	PRINTPIC
208	SAVEPIC
209	Working With Dribble Files
209	DRIBBLE
210	NODRIBBLE
209	Working With Data Files
211	Reading and Writing Information
211	Opening Files
212	ALLOPEN
212	CLOSE
213	CLOSEALL
214	FILELEN
215	OPEN
216	READER
217	READPOS
218	SETREAD
219	SETREADPOS
219	SETWRITE
220	SETWRITEPOS
221	WRITEPOS
221	WRITER
221	A Sample Project Using the Data File System
222	Step 1: Creating a Data File
224	Step 2: Retrieving Information
225	Step 3: Changing Information

Chapter 17 **Property Lists** **229**

229	Using Property Lists to Keep Records
230	ERPROPS
230	GPROP
231	PLIST
232	PPROP
232	PPS
233	REMPROP

Chapter 18 **Special Primitives** **237**

238	Assembly-Language Primitives
238	Some Specifics About the Apple's Memory
241	.AUXDEPOSIT

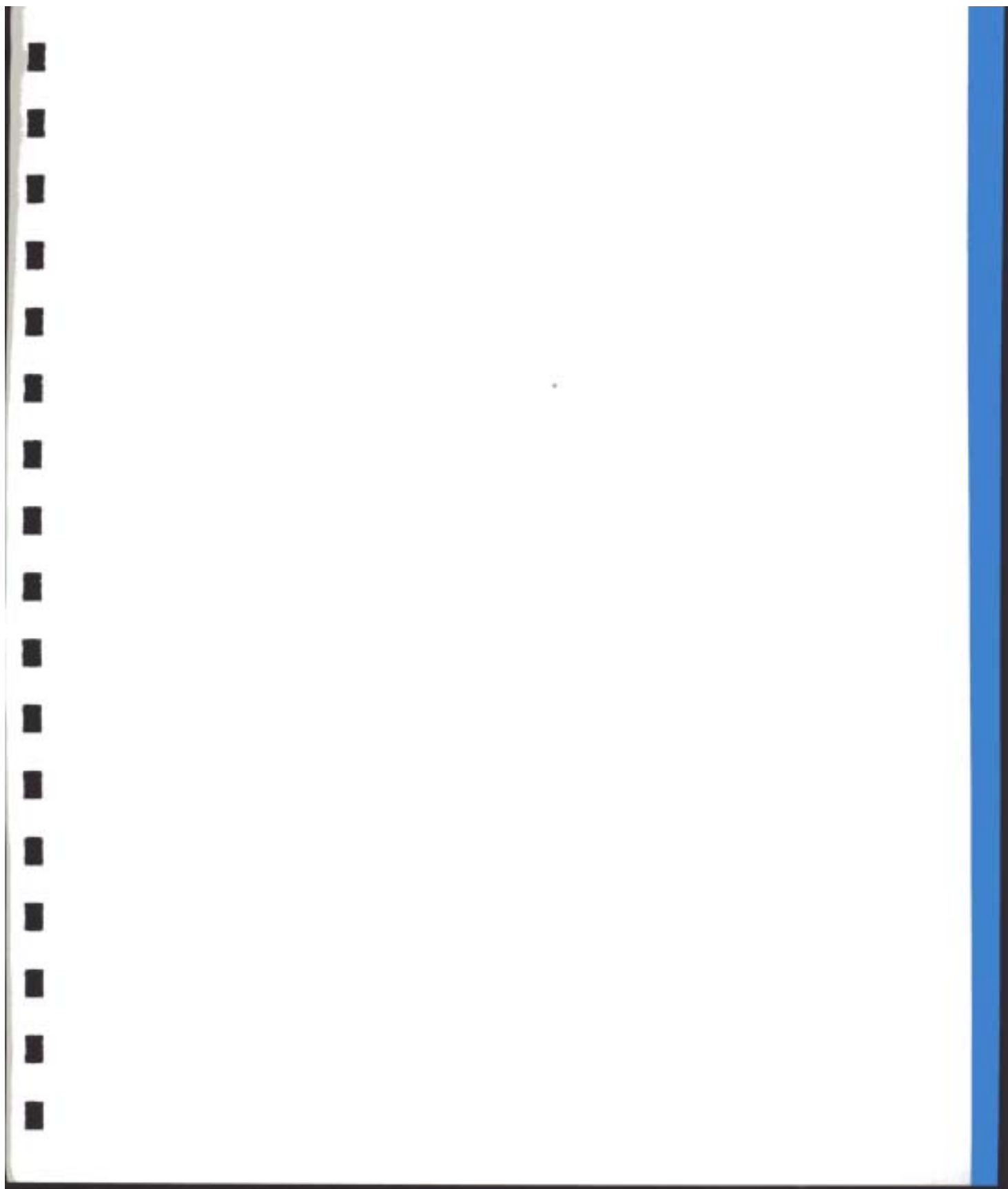
242	.AUXEXAMINE
242	.BLOAD
242	.BSAVE
242	.CALL
242	.DEPOSIT
243	.EXAMINE
243	Special Graphics Primitives
243	.SCRUNCH
243	.SETSCRUNCH
245	Miscellaneous Primitives
245	.CONTENTS
245	.QUIT

Appendix A	Messages	251
-------------------	-----------------	------------

Appendix B	Useful Tools	255
-------------------	---------------------	------------

255	Graphics Tools
255	ARCR and ARCL
256	CIRCLER and CIRCLEL
256	POLY
257	Math Tools
257	ABS
257	CONVERT
258	DIVISORP
258	LOG
258	LN
259	PWR
260	EXP
260	Program Logic or Debugging Tools
261	COMMENT
261	FOREVER
261	MAP
262	SORT and SUPERSORT
262	WHILE
262	Tools for the Young Logo User
262	DRIVE
263	TEACH

Appendix C	Startup Files	267
	267	Creating a STARTUP File
	268	A Note of Caution Before You Start
	268	The STARTUP Variable
Appendix D	Memory Space	271
	271	How Space Is Allocated
	272	Some Hints for Saving Space
Appendix E	Parsing	275
	275	Delimiters and Spacing
	276	Infix Procedures
	277	Brackets and Parentheses
	277	Quotation Marks and Delimiters
	278	The Minus Sign
Appendix F	ASCII Character Codes	281
Appendix G	Summary of Logo Primitives	285
Appendix H	Using a Printer With Logo	299
	300	The Software
	301	The Computer
	303	The Printer
	Glossary	307
	Index	315



List of Figures and Tables



Preface	About This Manual	xxii
	xxii Figure P-1. Sample Logo Screen	
Chapter 1	Introduction	3
	5 Table 1-1. Keystrokes for Typing and Editing	
	7 Table 1-2. Input Words	
Chapter 13	The Outside World	163
	172 Table 13-1. Note Frequencies for TOOT	
Chapter 15	General File Management	189
	192 Figure 15-1. Files and Subdirectories on a Volume	
Chapter 18	Special Primitives	237
	239 Figure 18-1. Map of Main Memory Bank	
	240 Figure 18-2. Map of Auxiliary Memory Bank	
	240 Table 18-1. Special Memory Locations	
Appendix A	Messages	251
	251 Table A-1. Logo Messages	

Appendix F *ASCII Character Codes* **281**

282 Table F-1. ASCII Codes for Normal Characters

283 Table F-2. ASCII Codes for Inverse Characters

Appendix H *Using a Printer With Logo* **299**

299 Table H-1. Printer Problems and Causes



About This Manual



This manual describes in detail how to use Apple Logo II and is intended for reference purposes. The accompanying manual, *Apple Logo II: An Introduction to Programming*, introduces you to the more fundamental features of Logo and is intended as a guide to becoming familiar with Logo.

This reference manual offers concise descriptions of each of the primitives in the Logo language, along with many sample programs (procedures). The chapter headings listed in the Table of Contents provide a handy reference to how the primitives are organized.

How to Use This Manual

Here are some suggestions on how to proceed.

The intended audience	This manual is written for people who already know something about Logo or languages like Logo.
To learn the basics	Work through the accompanying manual, <i>Apple Logo II: An Introduction to Programming</i> .
To get an overview of the rules of Logo's grammar	Read Chapter 2, Logo Grammar. You should read the overview before using this manual.

To find a primitive to perform a particular task

Look at the chapter headings in the Table of Contents or at the Apple Logo II Reference Card. Both list categories of primitives so you can locate a relevant chapter.

To find out what a particular primitive does

Look it up in Appendix G or in the Index.

To find out more about Logo

For a definition of a word, or an explanation of a new term, refer to the Glossary at the end of this manual. The Index is also a handy means of finding information quickly.

To get quick general help from Logo itself

Hold down either ⌘ or ⌘ and press ? at any time, except when a procedure is executing. You'll see a display with lots of helpful information.

To get quick help from Logo about a specific primitive

Type HELP followed by the name of the primitive you want to know about and press RETURN . (Remember to put a double quotation mark before the name of the primitive.) You'll see a display listing the inputs to that primitive.

To find out more about the Apple IIe or the Apple IIc

Read the appropriate owner's manual.

To help us improve future Apple products

Please fill out the *Tell Apple* form at the end of this manual. Your experience with Logo will help us in planning new products and manuals.

Visual Cues

Procedure definitions and sample interactions between you and Logo appear in a different type font from the rest of the manual. This font represents more closely what you see on your screen display.

Look for the following additional visual aids throughout the manual.

When you see a hyphen joining two keys, it means that you press the keys simultaneously. For instance, (⌘)-⌘ means you should press (⌘) and ⌘ at the same time. In actual practice, you probably will press (⌘) first and then, while still holding down (⌘), press ⌘.

Notes in the margin reinforce new terms or point to useful information elsewhere in this manual.

Note: Gray boxes like this provide helpful hints or interesting pieces of information.

Warning

Boxes like this indicate potential problems or disasters.

Figure P-1 shows the Logo opening screen display.

Figure P-1. Sample Logo Screen

Logo screens are shown like this.



A Note for Apple IIe Owners: If you are using an Apple IIe, the \square shown in the message above may appear on your screen as a black letter A on a light-colored rectangle. Whenever you see this on your screen, it stands for \square .

Introduction

- 3** What You Need
- 4** Getting Help From Logo
- 5** Typing Logo Instructions
- 6** How Primitives Are Described

Introduction



Logo is a computer language that offers features for both the novice and the experienced programmer. Logo's features range from turtle graphics, which lets you create interesting pictures quickly and easily, to features for list processing and file management.

This chapter gives you

- a list of the equipment you need to use Apple Logo II
- methods of getting help from Logo
- rules for typing Logo instructions
- an explanation of how Logo primitives are described throughout this manual.

What You Need

To use Apple Logo II, you must have

- either an Apple IIc computer or an Apple IIe computer with an Extended 80-Column Text Card and a disk drive (The Apple IIc has a built-in disk drive; you can use an additional drive if you wish, but it isn't necessary.)
- a video display device, either a video monitor or a television set

- the Logo disk, which has the name *APPLE LOGO II*
- a compatible printer (optional)

For the Apple IIc, you can use the Apple Imagewriter to print text and graphics. You connect the printer to port 1 on the back panel of the computer.

For the Apple IIe, you can use the Apple Imagewriter or the Apple Dot Matrix Printer. You can print text on both printers, but you cannot print graphics on the Dot Matrix Printer.

For information on printers, see the owner's manual that came with your Apple.

Other compatible printers may work for text, but not for graphics.

Getting Help From Logo

Logo provides two ways for you to get help while using it; one gives general help information and the other gives information about a specific Logo primitive.

To get general information about Logo, press $\langle \text{G} \rangle \langle \text{?} \rangle$. Logo displays one of two possible screens, depending on where you are when you request help:

- If you are at top level, the help screen has information about turtle graphics commands, using the Editor, defining a procedure, and special keystrokes.
- If you are in the Logo Editor, the help screen has information about the Editor keystrokes.

Before Logo displays the help screen, Logo saves the contents of the current screen. Then Logo displays the help screen in 40 columns. You can scroll through the screen using $\langle \text{Y} \rangle$ and $\langle \text{D} \rangle$, or, by pressing $\langle \text{G} \rangle \langle \text{ESC} \rangle$, you can return to the place from which you asked for help.

To get information about a specific Logo primitive, type `HELP` and the primitive name, with a quotation mark before the name. Logo displays the inputs required for that primitive.

Typing Logo Instructions

This section describes the guidelines for typing in uppercase and lowercase letters and the keystrokes for communicating with Logo from the keyboard.

Logo does not distinguish between uppercase and lowercase letters in any words you type. Thus, when typing anything into the computer, you need not pay attention to which case you are using. For example, if you define a procedure with the name SQUARE, then ask Logo to execute it, Logo will execute it regardless of what case you use for the letters. So, SQUARE is the same as Square or square.

Table 1-1 lists the keystrokes to use with Logo at top level and what they do.

For a list of the keystrokes used with the Logo Editor, see Chapter 4, Using the Logo Editor.

Table 1-1. Keystrokes for Typing and Editing









Keystroke	What It Does
	Moves the cursor left one character position.
	Moves the cursor right one character position.
	Moves the cursor left one word.
	Moves the cursor right one word.
 or 	Moves the cursor to the beginning of the current line.
 or 	Moves the cursor to the end of the current line.

Table 1-1. Keystrokes for Typing and Editing (continued)

Keystroke	What It Does
<code>CONTROL-D</code> or <code>DELETE</code>	Erases the character to the left of the cursor.
<code>CONTROL-F</code>	Erases the character under the cursor.
<code>CONTROL-X</code>	Erases all the characters on the current line.
<code>CONTROL-Y</code>	Erases all the characters from the present cursor position to the end of the line.
<code>CONTROL-R</code>	Retrieves the last line you typed or erased using <code>CONTROL-Y</code> .
<code>Q-?</code> or <code>Q-7</code>	Displays a screen of helpful information.
<code>RETURN</code>	From anywhere in the line, tells Logo to do what you just typed.

How Primitives Are Described

At the beginning of each primitive description, you will find

- the format of the primitive and its inputs: the name of the primitive, the number of inputs to the primitive, and the type of input required. All of the input words used are listed at the end of this chapter.
- the short form of the primitive, if there is one, in parentheses.
- an indication of what kind of primitive it is: command, operation, or infix operation.

Some primitives (such as SUM) have an optional format, which is enclosed in parentheses. This indicates that the primitive will accept as many inputs as you wish. When using more than two inputs with such a primitive (or, in some cases, one input), you must always put a left parenthesis before its name and a right parenthesis after the last input.

Table 1-2 lists the words used in the syntax of Logo primitives. The words represent the kind of input a primitive needs.

Table 1-2. *Input Words*

Input Word	Description
byte	A unit of data used by the computer. An integer from 0 through 255.
character	Letters of the alphabet, numbers, and punctuation marks.
colornumber	An integer from 0 through 5 giving the color of the pen or background.
[columnnumber linenumber]	A list of two integers giving the position of the cursor.
degrees	Degrees of an angle, a number.
distance	A number.
duration	An integer from 0 through 65,535.
field	An integer giving the number of elements in a number.
file	A pathname or a slot or port number.
frequency	An integer from 3 through 65,535.
inputs	Words with colons in front. Used in conjunction with TO.
integer	An integer. If you substitute a decimal number for an integer, Logo truncates the number and continues processing.
list	A list of words or lists.
loc	A location (region) of memory.
name(list)	A word naming a procedure or a variable, or a list of names.

Table 1-2. *Input Words (continued)*

Input Word	Description
number	A real number or an integer.
object (obj)	A Logo object—a word, a list, or a number.
paddlenumber	An integer (0, 1, 2 or 3) specifying the paddle.
pathname	A name that indicates the path to a file on a disk.
precision	An integer from 0 through 6, giving the number of digits after the decimal point in a real number.
predicate	An operation that gives either the word TRUE or the word FALSE.
prefix	A name for a ProDOS prefix of a file on disk.
property	A word.
width	An integer, either 40 or 80.
word	A sequence of characters.
[xcor ycor]	A list of two numbers giving the coordinates of the turtle.

Logo Grammar.

- 11** Procedures
- 13** Punctuation and Inputs to Procedures
- 14** Commands and Operations
- 15** Variables
- 16** Global and Local Variables
- 17** Understanding a Logo Line

Logo Grammar



Logo is a powerful and flexible programming language made up of *building blocks* called **procedures**. Some procedures are already built into the Logo system; these are called **primitives**. Others are defined by you. Other than the fact that primitives are built in, there is no difference between primitives and the procedures you define.

Procedures can construct, modify, and run other procedures. They obey the rules of Logo grammar. The following sections briefly describe these rules.

Procedures

Here is the definition of a procedure called WELCOME:

```
TO WELCOME                (title line)
PRINT "HI
END
```

The title line always begins with TO followed by the name of the procedure. The last line contains only the word END. For WELCOME, the main body is a request to run the primitive PRINT.

There are three ways of defining a procedure:

- By typing in its definition at top level (indicated by the question mark prompt character)
- By using the Logo Editor
- By using the primitive DEFINE.

Once a procedure is defined, one way of executing it is to type its name at top level:

```
?WELCOME          (procedure call)
HI                 (result)
```

Another way is to call the procedure inside the definition of another procedure. Suppose WARMWELCOME is defined like this:

```
TO WARMWELCOME
WELCOME
WELCOME
WELCOME
WELCOME
WELCOME
END
```

When it's called, WARMWELCOME executes WELCOME five times.

```
?WARMWELCOME
HI
HI
HI
HI
HI
```

WARMWELCOME is the **superprocedure** that contains the **subprocedure** WELCOME. Using superprocedures and subprocedures, you can build structures of great complexity.

A procedure can also be a subprocedure of itself. This is called **recursion**. You'll find many examples of this powerful Logo feature throughout this manual.

If you ask Logo to run an undefined procedure, an error message appears.

```
?TALK
I DON'T KNOW HOW TO TALK
```

Punctuation and Inputs to Procedures

Logo interprets every word as a request to run a procedure. You must use special characters to indicate when this is not the case.

A word beginning with a quotation mark—for example, "HI—tells Logo that the word must be treated literally, not as a procedure call. Here, "HI is an input to the procedure PRINT.

```
?PRINT "HI  
HI
```

Numbers are like literal words, but don't need quotation marks.

```
?PRINT 5  
5
```

A sequence of words surrounded by square brackets indicates a **list**. Lists can be inputs to procedures.

```
?PRINT [ARE WE HAVING FUN?]  
ARE WE HAVING FUN?
```

The list [ARE WE HAVING FUN] is a literal list; Logo does not try to execute it. The following example illustrates this more clearly.

```
?PRINT [2 + 2]  
2 + 2
```

Without the brackets, Logo will attempt to execute the sequence of words.

```
?PRINT ARE WE HAVING FUN?  
I DON'T KNOW HOW TO ARE
```

or

```
?PRINT 2 + 2  
4
```

Your procedures can also have inputs. For example:

```
TO GREET :NAME (title line)  
PR "HI  
PR :NAME  
PR [HAVE A NICE DAY]  
END
```

A word beginning with a colon (:) tells Logo that the word is a variable. Variables that hold the inputs to procedures are written on the title line after the name of the procedure. NAME is a variable whose value is determined when GREET is called. The main body of GREET contains three calls of the procedure PRINT (PR is the short form of PRINT). The second of these calls uses the current value of NAME.

Here's an example of a request to execute GREET at top level.

```
?GREET "GUY
HI
GUY
HAVE A NICE DAY
```

In this case, the input is the literal word GUY; Logo makes this the value of NAME when it executes GREET.

■ **Commands and Operations**

There are two kinds of procedures in Logo: operations and commands. **Operations** output a value to another procedure; **commands** (such as PRINT) do not.

The primitive SUM is an operation that outputs the sum of two numeric inputs. In this example, the output of SUM is sent to the primitive command PRINT:

```
?PRINT SUM 31 28
59
```

Because an operation can be used only as an input to another procedure, every Logo line must begin with a command. Otherwise, you get an error message. For example:

```
?SUM 31 28
YOU DON'T SAY WHAT TO DO WITH 59
```

Your procedures can be commands or operations. The procedure GREET is a command. To construct operations, you must use the primitive OUTPUT. The procedure FLIP, for example, is an operation:

```
TO FLIP
IF (RANDOM 2) = 0 [OUTPUT "HEADS]
OUTPUT "TAILS
END
```


FLIP outputs the literal word HEADS if RANDOM 2 outputs 0, or TAILS if RANDOM 2 outputs 1. You can pass the output from FLIP to PRINT:

```
?PR FLIP  
HEADS
```

Variables

You can think of a Logo **variable** as a container with a name on the outside and an **object** (a word, list, or number) inside. A colon in front of a word tells Logo it is a variable and makes its current value available to a procedure. For example:

```
PRINT :JOHN
```

tells Logo to look for a container named JOHN. If it finds one, it looks inside the container and makes whatever it finds available to PRINT. PRINT then displays the contents of JOHN on the screen.

If no variable JOHN exists, Logo prints the error message:

```
JOHN HAS NO VALUE
```

You can assign a value to a variable in two ways:

- By defining a procedure with inputs and then calling the procedure with specified values.
- By using the primitive MAKE or NAME.

MAKE requires two inputs: a word and a value.

```
?MAKE "JOHN 25  
?PRINT :JOHN  
25
```

In this case, the value is a number (25). However, it can be a word or a list as well. Consider this example:

```
MAKE "X "JOHN
```

See section "Punctuation and Inputs to Procedures."

Here, MAKE has two quoted words as inputs. It puts the literal word JOHN inside the container X. The contents of the variable name JOHN from the previous example are undisturbed. So,

```
?PRINT :X  
JOHN  
?PRINT :JOHN  
25
```

Global and Local Variables

Variables created with MAKE remain in the workspace until erased. These variables are called **global variables**. There are also variables that remain in the workspace only as long as a procedure is being executed. These variables are called **local variables**. Variables that are defined as inputs to procedures are local variables.

The procedure GREET can be modified to print the date.

```
TO GREET :NAME  
PR :DATE  
PR "HI  
PR :NAME  
PR [HAVE A NICE DAY]  
END
```

DATE does not appear on the title line of GREET, so it is a global variable. You can define the value of DATE at top level.

```
?MAKE "DATE [MARCH 14 1984]  
?GREET "BRIAN  
MARCH 14 1984  
HI  
BRIAN  
HAVE A NICE DAY
```

The variable NAME is not global. After GREET stops executing, NAME no longer has any value. (But DATE is still in the workspace.)

You could also use MAKE to define DATE inside the procedure GREET. It would still remain as a global variable after GREET executes. (The primitive LOCAL, however, lets you create local variables inside a procedure.)

Understanding a Logo Line

A Logo line can be longer than the line you see on the screen. For example:

```
MAKE "MANYNAMES [MIKE BARBARA GUY JUDY !  
SHARNEE EFFIE CHERYL]
```

The exclamation mark (!) indicates that the next screen line is a continuation of the previous screen line. A Logo line typed from top level can contain a maximum of 125 characters (including spaces). You end a Logo line by pressing **RETURN**.

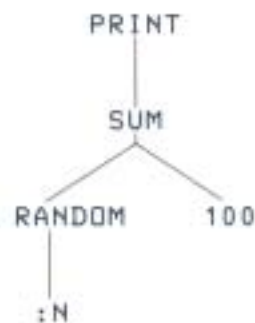
Here are some guidelines to help you interpret a complex Logo line:

- The first word of a Logo line must always be a command.
- An operation is always the input to another procedure.
- Every input to a procedure must be accounted for.
- When the inputs to a command have been accounted for, the next procedure must be another command.

Here is an example of a complex Logo line:

```
PRINT SUM RANDOM :N 100
```

PRINT is a command with one input, in this case the output of SUM. SUM requires two inputs. The first is the output of RANDOM, which itself requires one input (the current value of N). The second input to SUM is 100.



If N has been assigned the value 10,

```
?MAKE "N 10
```

then the line will print a number in the range 100..109:

```
?PRINT SUM RANDOM :N 100  
101
```

Defining Procedures With TO

```
21 TO  
22 END
```

Defining Procedures With T0



With the TO primitive, you can define your own procedures at top level without disturbing what's on the screen. This is advantageous if you need to look at instructions you have just used while entering a procedure definition.

TO

TO *name* (*input1 input2...*) (command)

TO tells Logo that you are defining a procedure called *name*, with inputs (if any) as indicated. From top level, the prompt character changes from ? to > to remind you that you are defining a procedure. While you are defining a procedure, Logo does not carry out the instructions you type; it makes them part of the procedure definition.

Note: You need not put a quotation mark before *name* because TO puts one there automatically.

To complete the procedure and return Logo to top level, type the word END as the last line of the procedure. The special word END must be used alone on the last line.

Example:

```
?TO GREET
>PRINT [HI THERE]
>END
GREET DEFINED
?
Input to Procedure —————>
Procedure Name —————> ?TO SQUARE :SIDE
Prompt Character —————> >FD :SIDE
                        >RT 90
                        >FD :SIDE
                        >RT 90
                        >FD :SIDE
                        >RT 90
                        >FD :SIDE
                        >RT 90
End of Procedure Definition —————> >END
Logo's Response —————> SQUARE DEFINED
?_
```

If you change your mind while defining a procedure with TO, press **[ESC]** to stop the definition. If a procedure is already defined, you can't change the definition with TO at top level. You must use EDIT or erase the old definition first with ERASE (ER).

END

END (special word)

END is necessary, when you are using TO, to tell Logo that you have finished defining the procedure. It must be on a line by itself. You must also use END to separate procedures when defining multiple procedures in the Logo Editor.

Using the Logo Editor

- 26** How the Editor Works
- 28** Editing Procedures With EDIT
- 29** Typing and Editing in the Editor
 - 29** Moving the Cursor
- 30** Inserting and Deleting Text
- 31** Getting Out of the Editor
- 31** Other Ways to Start Up the Editor

Using the Logo Editor



The Logo Editor is an interactive screen-oriented text editor, which provides a flexible way to define and change Logo instructions. The main command for starting up the Logo Editor is EDIT.

This chapter gives you

- information on how the Editor works
- the specifics of the EDIT command
- the rules for typing and editing in the Editor
- a brief description of other ways to start up the Editor.

How the Editor Works

When you call the Editor, Logo changes the screen. For example

```
?EDIT "POLY
```

```
                                LOGO EDITOR
-----
TO POLY :SIDE :ANGLE
FD :SIDE
RT :ANGLE
POLY :SIDE :ANGLE
END
-

-----
␣-A accept, ␣-? help, ␣-ESC cancel
```

There is no prompt character, but the cursor shows you where you are typing.

Note: The POLY procedure continues executing until you press `␣-ESC` to stop it.

The text that you edit is in an area of memory called a **buffer**. When you enter the Editor, Logo displays the text from the edit buffer, up to 20 lines per screen.

You can move the cursor anywhere in the text using the cursor control keys described later in this chapter. You can also delete and insert characters using the appropriate keys.

Each key that you type makes the Editor take some action. Most typewriter characters (letters, numbers, punctuation, and `RETURN`) are simply inserted into the buffer at the place marked on the screen by the cursor.

When you press `RETURN`, the cursor (and any text that comes after it) moves to the next line, ready for you to continue typing.

You can have more characters on a line of text than fit across the screen. When you get to the end of the line on the screen, just continue typing without pressing `RETURN`. An exclamation mark (!) appears in the rightmost character position on the line and the cursor moves to the next line.

Logo does the same thing outside of the Editor. Here is what the screen might look like:

```
?TO PRINTMESSAGES :PERSON
>PRINT SENTENCE :PERSON [ , I AM GOING T!
0 TYPE A VERY LONG MESSAGE FOR YOU]
>PRINT SENTENCE [SO LONG, ] :PERSON
>END
?
```

The Editor has an auxiliary line buffer called the **kill buffer**. You can use it to move lines in a procedure or to repeat them in different places. The buffer can hold a maximum of 125 characters. While this is true for the kill buffer, the length of a line is limited only by the length of the edit buffer (6144).

You can use `CONTROL-X` and `CONTROL-Y` to delete a whole line and a partial line of text, respectively, and put them in the kill buffer. `CONTROL-R` inserts the same line of text later at the place marked by the cursor.

`CONTROL-L` lets you see temporarily the graphics screen and its most recent contents. `CONTROL-T` restores the screen back to the Editor so you can pick up where you left off.

When you exit from the Editor using `CONTROL-A`, Logo reads each line in the edit buffer as if you had typed it directly from top level.

If the instructions in the edit buffer define a procedure (that is, if there is a title line `TO ...` that starts the definition), Logo behaves as though you had typed the definition of the procedure using `TO`. If the buffer contains a procedure definition, but there is no `END` instruction at the end of the buffer, Logo helps out by ending the definition for you.

If there are Logo instructions on lines in the edit buffer that are not part of the definition of a procedure, Logo carries them out when you exit the Editor.

In the Editor, you may define more than one procedure at a time. When you exit the Editor, you can go back to your original graphics screen.

Editing Procedures With **EDIT**

EDIT (ED) (command)
EDIT *name(list)*

The EDIT command starts up the Logo Editor. If you give an input, the Editor starts up with the definition(s) of the given procedure(s) in the edit buffer. The input to EDIT can be a list of procedure names instead of a single name. In this case, all the procedure definitions will be brought into the Editor.

If the procedure *name* has not been previously defined, the edit buffer contains only the title line: TO *name*. If no input is given, the edit buffer has whatever it had the last time you used the Editor, or is empty if it is the first time you have used the Editor.

Press **(D)-(A)** to exit from the Editor and to have Logo read all the lines from the edit buffer as though it were typed at top level. If the end of the buffer is reached while there is a procedure definition in the Editor, Logo completes the procedure definition by inserting END.

Press **(D)-(ESC)** to stop editing without completing the definition. Use this key if you don't like the changes you are making or if you decide not to make any changes.

Typing and Editing in the Editor

This section presents the keystrokes you use when typing in the Editor. Note that some keystrokes work both inside and outside the Editor. These are indicated by an asterisk (*) to the left of the keystroke.

Note: Remember that pressing (F1)-(?) while in the Editor gives you a screen of information about the editing keystrokes.

Moving the Cursor

These keystrokes move the cursor around in the indicated ways.

*←	Moves the cursor left one character position.
*→	Moves the cursor right one character position.
↓	Moves the cursor down one line to the next line. The cursor tries to go to the character position directly underneath its position on the current line. If the next line is shorter than the cursor position on the current line, the cursor goes to the end of the next line. If the cursor is on the last line of the edit buffer, it does not move.

Example:

THIS IS A TEXT <u>L</u> INE	Cursor on <i>L</i> in <i>LINE</i>
THIS IS ANOTHER <u> </u> TEXT LINE	Cursor on space before <i>TEXT</i>
A SHORTER ONE <u> </u>	Cursor at end of line
THIS IS A LONGER <u>R</u> ONE THAN CAN FIT ON TH!	Cursor on <i>R</i> in <i>LONGER</i>
E SCREEN <u> </u>	Cursor at end of line
THIS IS THE NEXT <u>T</u> LINE	Cursor on <i>T</i> in <i>NEXT</i>

Ⓜ	Moves the cursor up one line to the previous line. The cursor tries to go to the character position directly above its position on the current line. If the previous line is shorter than the cursor position on the current line, the cursor moves to the end of the previous line.
*Ⓜ-Ⓛ	Moves the cursor to the left one word.
*Ⓜ-Ⓡ	Moves the cursor to the right one word.
*Ⓜ-Ⓛ or *Ⓜ-Ⓡ	Moves the cursor to the beginning of the current line.
*Ⓜ-Ⓡ or *Ⓜ-Ⓛ	Moves the cursor to the end of the current line.
Ⓜ-Ⓜ	Moves the cursor to the top of the page. If the cursor is already at the top of the page, it moves the cursor to the top of the previous page and displays the new page.
Ⓜ-Ⓜ	Moves the cursor to the bottom of the page. If the cursor is already at the bottom of the page, it moves the cursor to the top of the next page and displays the new page.
Ⓜ-Ⓜ through Ⓜ-Ⓜ	Moves the cursor to the beginning of a line at a point in the edit buffer. Ⓜ-Ⓜ moves to the start of the buffer, Ⓜ-Ⓜ moves to the end of the buffer, and the others move proportionately throughout the buffer.

Inserting and Deleting Text

These keystrokes insert and delete text in the indicated ways.

Ⓜ	From anywhere in the line, accepts the line as it is displayed and moves the cursor and the rest of the line to the beginning of a new line.
*Ⓜ-Ⓜ or *Ⓜ-Ⓜ	Erases the character to the left of the cursor.
*Ⓜ-Ⓜ	Deletes the character under the cursor.

- * **CONTROL-X** Deletes all the characters on the current line, up to 125 characters. Logo puts this text in the kill buffer.
- * **CONTROL-Y** Deletes all the characters from the present cursor position to the end of the current line. Logo puts this text in the kill buffer.
- * **CONTROL-R** When you are inside the Editor, **CONTROL-R** inserts a copy of the text that is in the kill buffer at the current cursor position. When you are outside the Editor, it retrieves the last line you typed, or whatever has been deleted with **CONTROL-X** or **CONTROL-Y**.
- CONTROL-O** Opens a line at the present cursor position.

Getting Out of the Editor

Use these keystrokes to get out of the Editor.

- CONTROL-A** Accepts your work and causes Logo to read the contents of the edit buffer as if you typed them at top level.
- CONTROL-ESC** Discards your work. Any changes you've made are left in the edit buffer. Use it if you don't like the changes you are making or you decide not to make changes. If you were defining a procedure, the definition will be the same as before you started editing. If you press **CONTROL-ESC** by accident, you can retrieve the contents of the edit buffer with the EDIT command and no inputs.

Other Ways to Start Up the Editor

You can use three other Logo primitives besides EDIT to start up the Logo Editor: EDN, EDNS, and EDITFILE.

You use EDN and EDNS for editing variables. EDN starts up the Editor with the variables you indicate and their corresponding values. You can then edit these variable names and values. EDNS starts up the Editor with all variable names and their values in it. EDITFILE starts up the Logo Editor with the contents of the file you indicate. You can then edit the file, and it will be saved with the same filename.

EDN and EDNS are described in Chapter 8, Variables.

You can read more about EDITFILE in Chapter 15, General File Management.

Turtle Graphics

- 36** Changing the Turtle's State
- 36** BACK
- 37** CLEARSCREEN
- 37** FORWARD
- 38** HIDE TURTLE
- 38** HOME
- 38** LEFT
- 39** RIGHT
- 40** SETHEADING
- 40** SETPOS
- 41** SETX
- 41** SETY
- 42** SHOW TURTLE
- 43** Getting Information About the Turtle's State
- 43** HEADING
- 43** POS
- 44** SHOWNP
- 45** TOWARDS
- 45** XCOR
- 46** YCOR
- 47** Using the Pen and Screen
- 47** CLEAN
- 47** DOT
- 48** FENCE
- 48** FILL
- 49** PENDOWN

50	PENERASE
50	PENREVERSE
51	PENUP
51	SETBG
52	SETPC
53	WINDOW
53	WRAP
54	Getting Information About the Pen and Screen
54	BACKGROUND
54	DOTP
54	PEN
55	PENCOLOR

Turtle Graphics



FULLSCREEN, SPLITSCREEN, and TEXTSCREEN are described in Chapter 6.

Apple Logo has two kinds of screens: the graphics screen and the text screen. When you use any primitive or procedure that refers to the turtle, Logo shows you the graphics screen. The commands FULLSCREEN, SPLITSCREEN, and TEXTSCREEN allow you to switch between the two kinds of screens.

This chapter presents a complete list of the commands that change what you see on the graphics screen. It also includes a number of operations that give you information about the state of the turtle, the pen, and the screen. The primitives appear in four groups:

- primitives that change the turtle's state
- primitives that give you information about the turtle's state
- primitives that tell Logo to do something with the pen or screen
- primitives that tell you about the state of the pen or the screen.

Many of these commands are discussed in the *Apple Logo II: An Introduction to Programming* manual. This chapter assumes that you have already read that manual.

Changing the Turtle's State

This section explains all the commands that tell the turtle to do something. The commands appear in this order:

BACK	RIGHT
CLEARSCREEN	SETHEADING
FORWARD	SETPOS
HIDETURTLE	SETX
HOME	SETY
LEFT	SHOWTURTLE

The screen limits are 240 turtle steps high and 280 steps wide. Hence, when using Cartesian coordinates (as in SETPOS), you reach the edge of the screen when the *y*-coordinate is 119 (top) or -120 (bottom) and the *x*-coordinate is -140 (left edge) or 139 (right edge). (This is true when the aspect ratio is .8.) Note that you need not worry about these coordinates when using FORWARD and BACK.

BACK

BACK *distance* (BK) (command)

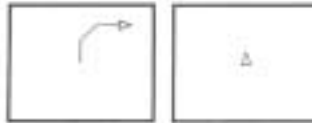
The BACK command moves the turtle *distance* steps back. Its heading does not change. If the pen is down, Logo draws a line the specified distance.



CLEARSCREEN

CLEARSCREEN (CS) (command)

CLEARSCREEN erases the graphics screen, puts the turtle in the center of the screen, and sets the turtle's heading to 0 (north). The center of the screen is position [0 0] and is called the **home position**.



FORWARD

FORWARD *distance* (FD) (command)

FORWARD moves the turtle forward *distance* steps in the direction in which it is heading. If the pen is down, Logo draws a line the specified distance.

Examples:



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



HIDETURTLE

HIDETURTLE (HT) (command)

HIDETURTLE makes the turtle invisible. (The turtle draws faster when it is hidden.)



HOME

HOME (command)

The HOME command moves the turtle to the center of the screen and sets its heading to 0. If the pen is down, Logo draws a line to the new position. The HOME command is equivalent to

```
SETPOS [0 0]  
SETHEADING 0
```



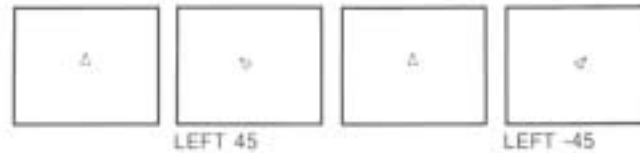
LEFT

LEFT *degrees* (LT) (command)

The LEFT command turns the turtle left (counterclockwise) the specified number of degrees. The number of degrees must not be greater than 4.19E6.

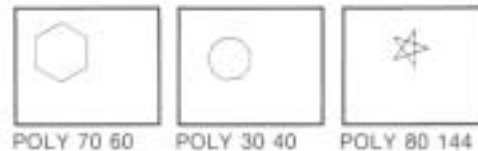
Examples:

LEFT 45 turns the turtle 45 degrees left
 LEFT -45 turns the turtle 45 degrees right



The procedure POLY draws figures like those illustrated:

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

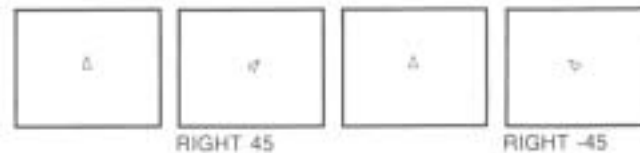
**RIGHT**

RIGHT *degrees* (RT) (command)

The RIGHT command turns the turtle right (clockwise) the specified number of degrees. The number of degrees must not be greater than 4.19E6.

Examples:

RIGHT 45 turns the turtle 45 degrees right
 RIGHT -45 turns the turtle 45 degrees left



```
TO SPI :SIDE :ANGLE :INC
FD :SIDE RT :ANGLE
SPI :SIDE + :INC :ANGLE :INC
END
```



SPI 5 144 3

SETHEADING

SETHEADING *degrees* (SETH) (command)

SETHEADING turns the turtle so that it is heading in the direction *degrees*, which can be any decimal number less than 4.19E6. Positive numbers are clockwise from north, negative numbers are counterclockwise from north. Note that RIGHT and LEFT do relative motion, but SETHEADING does absolute motion.

Examples:

SETHEADING 45 heads the turtle northeast
SETHEADING -45 heads the turtle northwest



SETPOS

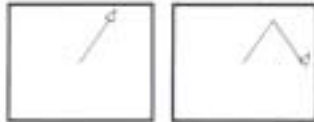
SETPOS [*xcor* *ycor*] (command)

The SETPOS (for set position) command moves the turtle to the indicated coordinates. If the pen is down, Logo draws a line to the new position.

See also section "POS."

Example:

SETPOS [100 0] moves the turtle to a point halfway down the right edge of the screen.



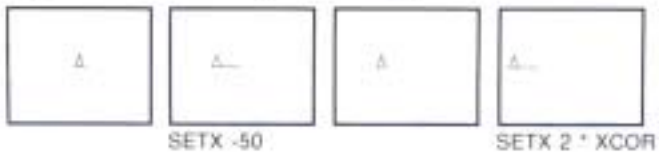
SETX

SETX *xcor* (command)

SETX moves the turtle horizontally to a point with x-coordinate *xcor*. The y-coordinate is unchanged. If the pen is down, Logo draws a line to the new position.

Example:

SETX -50 moves the turtle horizontally over towards the left edge of the screen. (The left edge of the screen is -140.)



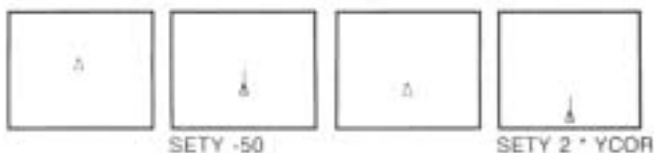
SETY

SETY *ycor* (command)

SETY moves the turtle vertically to a point with y-coordinate *ycor*. The x-coordinate is unchanged. If the pen is down, Logo draws a line to the new position.

Example:

SETY -50 moves the turtle vertically towards the lower edge of the screen. (The lower edge of the screen is -120 when the aspect ratio is .8 .)



SHOWTURTLE

SHOWTURTLE (ST) (command)

See also section "HIDETURTLE."

SHOWTURTLE makes the turtle visible.



Getting Information About the Turtle's State

This section explains all the operations that inform you about the turtle's state. The primitives appear in this order:

HEADING
POS
SHOWNP
TOWARDS
XCOR
YCOR

HEADING

HEADING (operation)

HEADING outputs the turtle's heading, a decimal number greater than or equal to 0 and less than 360. Logo follows the compass system where north is a heading of 0 degrees, east 90, south 180, and west 270. When you start up Logo, the turtle has a heading of 0 (straight up).

Example:

```
IF HEADING = 180 [PR [YOU ARE HEADED DU!  
E SOUTH]]
```

POS

POS (operation)

POS (for position) outputs the coordinates of the current position of the turtle in the form of a list [xcor ycor]. When you start up Logo, the turtle is at [0 0], the center of the turtle field.

Example:

```
TO GOODVEE
MAKE "SAVEPOS POS
VEE
PENUP
SETPOS :SAVEPOS
PENDOWN
END
```

```
TO VEE
RT 135 FD 20
LT 90 FD 20
LT 45
END
```



GOODVEE

GOODVEE calls the procedure VEE and then restores the turtle's position to wherever it was before GOODVEE was called.

SHOWNP

SHOWNP

(operation)

SHOWNP outputs TRUE if the turtle is not hidden, FALSE otherwise.

TOWARDS

TOWARDS [*xcor* *ycor*] (operation)

TOWARDS outputs a heading that would make the turtle face in the direction indicated by [*xcor* *ycor*].

Example:

SETHEADING TOWARDS [20 10] heads the turtle in the direction of the position [20 10].



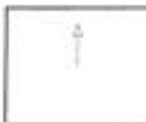
XCOR

XCOR (operation)

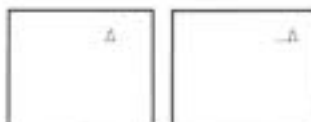
XCOR outputs the x-coordinate of the current position of the turtle.

Examples:

```
?PRINT XCOR  
10.0
```



SETX 2 * XCOR moves the turtle horizontally to a position twice as far from the y-axis as it used to be.



YCOR

YCOR

(operation)

YCOR outputs the y-coordinate of the current position of the turtle.

Examples:

```
?PRINT YCOR  
50.0
```



SETY 2 * YCOR moves the turtle vertically to a position twice as far from the x-axis as it used to be.



Using the Pen and Screen

This section explains all the commands that direct Logo to do something with the pen or screen. The commands appear in this order:

CLEAN	PENREVERSE
DOT	PENUP
FENCE	SETBG
FILL	SETPC
PENDOWN	WINDOW
PENERASE	WRAP

CLEAN

CLEAN (command)

The CLEAN command erases the graphics screen but doesn't affect the turtle.



DOT

DOT [*xcor* *ycor*] (command)

The DOT command puts a dot of the current pen color at the specified coordinates, without moving the turtle. It does not draw a line, even if the pen is down.

Example:

`DOT [120 0]` puts a dot near the right edge of the screen.



FENCE

`FENCE` (command)

See also sections "WINDOW" and "WRAP."

The `FENCE` command fences in the turtle within the edges of the screen. If you try to move the turtle beyond the edges of the screen, an error occurs and the turtle does not move. If the turtle is already out of bounds, Logo repositions it at its home position [0 0].

Example:

```
FENCE
CS
RT 5
FD 500
```

gives the error message `TURTLE OUT OF BOUNDS`.

FILL

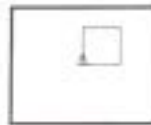
`FILL` (command)

The `FILL` command fills the shape outlined by the current pen color with the current pen color. If the turtle is not enclosed, the background is filled with the current pen color. Logo ignores lines of colors other than the current pen color when determining what to fill.

Example:

```
TO FILLAT :POS  
LOCAL "POSITION  
MAKE "POSITION POS  
PU SETPOS :POS PD FILL  
PU SETPOS :POSITION PD  
END
```

This procedure moves the turtle to a specified position, fills, and returns the turtle to its original position.



REPEAT 4
[FD 50 RT 90]



PU RT 45 FD 20

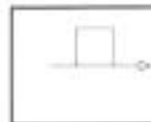


PD FILL

PENDOWN

PENDOWN (PD) (command)

The PENDOWN command puts the turtle's pen down. When the turtle moves, it draws lines in the current pen color. When you start up Logo, the pen is down.



PENDOWN FD 100

PENERASE

PENERASE (PE) (command)

PENERASE puts the turtle's eraser down. When the turtle moves, it erases lines it passes over. To take away the eraser, use either PENDOWN or PENUP.

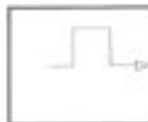


PENERASE FD 100

PENREVERSE

PENREVERSE (PX) (command)

PENREVERSE puts the reversing pen down. When the turtle moves, it tries to interchange the pen color and background color, drawing where there aren't lines and erasing where there are. The exact effect of this reversal is complex; what it looks like on the screen depends on the pen color, background color, and whether lines are horizontal or vertical. The best results are on a black background.



PENREVERSE FD 100

PENUP

PENUP (PU) (command)

The PENUP command lifts the pen up: when the turtle moves, it does not draw lines. The turtle cannot draw until the pen is put down again.



PENUP FD 100

SETBG

SETBG *colornumber* (command)

The SETBG (for set background) command sets the background color to the color represented by *colornumber*, where *colornumber* is one of the following numbers:

0	black
1	white
2	green
3	violet
4	orange
5	blue
6	black (for black-and-white TV)

See the example included with the BACKGROUND command.

Note that background colors 0 and 6 are both black; 6 is the recommended background for a black-and-white screen, since the pen draws thinner lines with a 6 background.

There are certain unavoidable limitations when you draw with a colored pen on a colored background. Black and white pens draw successfully on any background; any colored pen draws successfully on a black or white background. If you try to draw a green or violet line on an orange or blue background, or an orange or blue line on a green or violet background, the following will happen:

orange or blue background:	green becomes orange violet becomes blue
green or violet background:	orange becomes green blue becomes violet

If you change the background after you've already drawn with a colored pen, the results may be blotchy.

SETPC

SETPC *colornumber* (command)

The SETPC (for set pencolor) command sets the color of the pen to *colornumber*, where *colornumber* is one of the following numbers:

0	black
1	white
2	green
3	violet
4	orange
5	blue

For information on the interaction between pen and background colors, see section "SETBG" in this chapter.

If the pen color does not look right on your screen, try adjusting the tint control. However, when two lines of different colors are horizontally close to each other, one of them may be the wrong color, no matter what you do.

See also sections "FENCE" and "WRAP."

See also sections "FENCE" and "WINDOW."

WINDOW

WINDOW (command)

The WINDOW command makes the turtle field unbounded; what you see is a portion of the turtle field as if looking through a small window around the center of the screen. When the turtle moves beyond the visible bounds of the screen, it continues to move but can't be seen: The screen is 240 turtle steps high (only if the scrunch factor is .8) and 280 steps wide. The entire turtle field is 40,960 steps high and 32,768 steps wide. Changing WINDOW to FENCE or WRAP when the turtle is off the screen sends the turtle to its home position [0 0].

Example:

```
?WINDOW
?CS RT 5
?FD 500
?PRINT POS
43.5779 498.097
```

WRAP

WRAP (command)

The WRAP command makes the turtle field wrap around the edges of the screen: if the turtle moves beyond one edge of the screen, it continues from the opposite edge. The turtle never leaves the visible bounds of the screen; when it tries to, it wraps around to the other side.

Example:

```
?WRAP
?CS RT 5
?FD 500
?PRINT POS
43.5779 18.0973
```

Getting Information About the Pen and Screen

This section explains all the operations that inform you about the state of the pen or screen. The primitives appear in this order:

BACKGROUND
DOTP
PEN
PENCOLOR

BACKGROUND

BACKGROUND . (BG) (operation)

BACKGROUND outputs a number representing the color of the background:

0	black
1	white
2	green
3	violet
4	orange
5	blue
6	black (for black-and-white TV)

When Logo first starts up, BACKGROUND outputs 0.

DOTP

DOTP [*xcor* *ycor*] (operation)

The DOTP operation outputs TRUE if there is a dot on the screen at the indicated coordinates. If there is no dot, DOTP outputs FALSE.

PEN

PEN (operation)

PEN outputs the current state of the turtle's pen. The states are PENDOWN, PENERASE, PENUP, and PENREVERSE. When the turtle first starts up, PEN outputs PENDOWN.

PENCOLOR

PENCOLOR (PC) (operation)

PENCOLOR outputs a number representing the current color of the pen:

0	black
1	white
2	green
3	violet
4	orange
5	blue

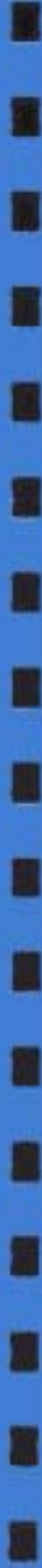
When Logo first starts up, PENCOLOR outputs 1.



Text and Screen Commands

- 60** Primitives Affecting Text on the Screen
- 60** `CLEARTEXT`
- 60** `CURSOR`
- 61** `FULLSCREEN`
- 61** `SETCURSOR`
- 62** `SETWIDTH`
- 63** `SPLITSCREEN`
- 63** `TEXTSCREEN`
- 63** `WIDTH`
- 63** Special Control Characters That Change Screen Use
- 63** `CONTROL-L`
- 64** `CONTROL-S`
- 64** `CONTROL-T`

Text and Screen Commands



Your Apple computer has 24 lines of text on the screen, with 40 or 80 characters on each line, depending on the current screen width setting. You can use the screen entirely for text or entirely for graphics. The Apple also lets you use the top 20 lines for graphics and the bottom four for text at the same time. When you start up Logo, the entire screen is available for text.

Your screen can be either 40 or 80 characters wide. You can switch between the two settings with the SETWIDTH primitive.

Note: If you have an Apple IIe, Logo will be in 40-column mode when you start up.

If you have an Apple IIc, Logo will read the state of the 80/40-column switch to determine which mode to start in.

There are two ways to change the use of your screen:

- With regular Logo commands, which you can type at top level or insert within procedures (FULLSCREEN, SPLITSCREEN, TEXTSCREEN, and SETWIDTH)
- With special control characters, which are read from the keyboard and obeyed almost immediately (while a procedure continues running); these cannot be placed within procedures ((CONTROL)-L, (CONTROL)-S, and (CONTROL)-T).

In addition to those described in this chapter, the primitives .SCRUNCH and .SETSCRUNCH are related to screen commands.

.SCRUNCH and .SETSCRUNCH are described in Chapter 18.

Primitives Affecting Text on the Screen

This section presents the commands that affect the screen. The commands are

CLEARTEXT
CURSOR
FULLSCREEN
SETCURSOR
SETWIDTH
SPLITSCREEN
TEXTSCREEN
WIDTH

CLEARTEXT

CLEARTEXT (CT) (command)

CLEARTEXT clears the entire screen and puts the cursor at the upper-left corner of the text part of the screen. If you have been using the split screen, the cursor is on the fourth line from the bottom.

CURSOR

CURSOR (operation)

CURSOR outputs a list of the column and line numbers of the cursor position. The upper-left corner of the screen is [0 0]. The upper-right is [39 0] if the screen width is 40, and [79 0] if the screen width is 80.

Example:

The procedure TAB tabs over to the next tab stop after something is typed. Tab stops are located in every eighth column.

```
TO TAB
TYPE CHAR 32
IF (REMAINDER FIRST CURSOR 8) > 0 [TAB]
END
```

See section "SETCURSOR."

```

TO FLAVORCHART
TYPE "FLAVOR TAB TAB PR "RATING PR [!
]
TYPE "CHOCOLATE TAB PR 97
TYPE "STRAWBERRY TAB PR 73
TYPE "BANANA TAB TAB PR 19
END

```

```

?FLAVORCHART
FLAVOR                RATING

CHOCOLATE             97
STRAWBERRY            73
BANANA                 19

```

FULLSCREEN

FULLSCREEN (FS) (command)

The FULLSCREEN command devotes the entire screen to graphics. Only the turtle field shows; any text you type will be invisible to you, although Logo will still carry out your instructions.

If Logo needs to display an error message while you are using the full graphics screen, Logo splits the screen.

SETCURSOR

SETCURSOR [*columnnumber linenumber*] (command)

SETCURSOR sets the cursor to the position indicated by *columnnumber* and *linenumber*. Lines on the screen are numbered from 0 to 23. Character positions (columns) are

numbered from 0 to 39 if the screen width is 40 and 0 to 79 if the screen width is 80.



An error occurs if the line number is not between 0 and 23, or if the column number is not between 0 and 38 (78 if the screen width is 80). If *columnnumber* or *linenumber* is a decimal number, Logo truncates it to an integer.

Examples:

SETCURSOR [20 12] puts the cursor near the middle of the screen.

```
TO MOVECURSOR :X :Y
SETCURSOR LIST (:X + FIRST CURSOR) (:Y !
+ LAST CURSOR)
END
?CLEARTEXT
?PRINT "A MOVECURSOR 2 5 PRINT "B
```

SETWIDTH

SETWIDTH *width* (command)

The SETWIDTH command sets the width of the screen to *width* characters per line. The *width* input must have a value of either 40 or 80. The default setting for the screen width depends on which computer you're using. If you're using an Apple IIe, the default setting for the screen width is 40. If you're using an Apple IIc, the default setting is whatever the 80/40-column switch is set to.

See section "WIDTH."

Example:

?SETWIDTH 80 changes the screen width to 80 characters per line.

SPLITSCREEN

SPLITSCREEN (SS) (command)

SPLITSCREEN devotes the top 20 lines of the screen to graphics and the bottom four lines to text.

TEXTSCREEN

TEXTSCREEN (TS) (command)

TEXTSCREEN devotes the entire screen to text; the graphics screen is invisible to you until a graphics procedure is run.

WIDTH

WIDTH (operation)

WIDTH outputs the current width of the screen, either 40 or 80. When you start up Logo, WIDTH outputs either 40, if you're using an Apple IIe, or whatever the 80/40-column switch is set to, if you're using an Apple IIc.

See section "SETWIDTH" for changing the screen width.

Special Control Characters That Change Screen Use

This section covers the special control characters that you can use to change the screen use. These control characters are

CONTROL-L
CONTROL-S
CONTROL-T

CONTROL-L

CONTROL-L (special character)

CONTROL-L is similar in effect to FULLSCREEN. You can use it at any time.

If you press **CONTROL-L** while in the Logo Editor, the graphics screen appears. (Use **CONTROL-T** to restore the Editor text screen.)

CONTROL-S

CONTROL-S (special character)

CONTROL-S is similar in effect to **SPLITSCREEN**. You can use it at any time.

CONTROL-T

CONTROL-T (special character)

CONTROL-T is similar in effect to **TEXTSCREEN**: it devotes the entire screen to text. You can use it at any time. **CONTROL-T** restores the Editor text screen if you have just used **CONTROL-L** from the Editor.

Words and Lists

- 67** Words: Some General Information
- 68** Lists: Some General Information
- 69** Breaking Words and Lists Into Pieces
- 70** BUTFIRST
- 71** BUTLAST
- 71** FIRST
- 73** ITEM
- 73** LAST
- 74** MEMBER
- 75** Putting Words and Lists Together
- 76** FPUT
- 76** LIST
- 77** LPUT
- 78** PARSE
- 78** SENTENCE
- 80** WORD
- 81** Examining Words and Lists
- 81** ASCII
- 82** BEFOREP
- 83** CHAR
- 85** COUNT
- 85** EMPTY
- 87** EQUALP
- 88** LISTP
- 88** MEMBERP
- 89** NUMBERP
- 90** WORDP
- 90** Changing the Case of Words
- 90** LOWERCASE
- 91** UPPERCASE

Words and Lists



This chapter describes the primitives that work on two types of objects in Logo: **words** and **lists**. With the primitives described in this chapter, you can

- break words and lists into pieces
- put words and lists together
- examine words and lists
- change the case of words and lists.

Words: Some General Information

A **word** is made up of characters. Here are some examples of words:

HELLO
X
314
3.14
R2D2
PIGLATIN
PIG.LATIN
PIG-LATIN (typed as PIG\LATIN)
HEN3RY
WHO?
!NOW!

Each character is an **element** of the word. The word HEN3RY contains six elements:

H E N 3 R Y

A word is usually delimited by spaces, which means that there is a space before the word (unless it is preceded by : or ") and a space after the word. The spaces set the word off from the rest of the line. In addition to spaces, these characters delimit words:

```
[ ] ( ) = < > + - *
```

To treat any of these characters or the space as a normal alphabetic character, put a backslash (\) before it.

Example:

```
?PR "PIG\L-LATIN  
PIG-LATIN
```

Note that the quotation mark character (") and the colon (:) are not word delimiters.

You can also have an **empty word**, which is a word with no elements. You type in the empty word by typing

```
"
```

See Appendix E, Parsing, for more details on how Logo treats special characters.

See the section of this chapter on the EMPTYP primitive for examples of the empty word.

Lists: Some General Information

A **list** is made up of Logo objects, each of which is a word or another list. You indicate that something is a list by enclosing it in square brackets ([]). Here are some examples of lists:

```
[HELLO THERE, OLD CHAP]  
[X Y Z]  
[HELLO]  
[[HOUSE MAISON] [WINDOW FENETRE] [DOG C]  
HIEN]]  
[HAL [C3PO R2D2] [QRZ] [ROBBIE SHAKEY]]  
[1 [1 2] [17 [17 2]]]  
[]
```

The list [HELLO THERE, OLD CHAP] contains four elements:

```
HELLO  
THERE,  
OLD  
CHAP
```

Note that the list [1 [1 2] [17 [17 2]]] contains only three elements, not six; the second and third elements are themselves lists:

See the section of this chapter on the EMPTYP primitive for examples of the empty list.

Element 1: 1
Element 2: [1 2]
Element 3: [17 [17 2]]

The list [], a list with no elements, is the **empty list**.

Breaking Words and Lists Into Pieces

The operations that break words and lists into pieces are

BUTFIRST (BF)
BUTLAST (BL)
FIRST
ITEM
LAST
MEMBER

The following chart shows how FIRST and BUTFIRST (BF) work. If you want to try out these operations, use the SHOW command.

FIRST	"JOHN	J
BF	"JOHN	OHN
FIRST	[MARY JOHN BILL]	MARY
BF	[MARY JOHN BILL]	[JOHN BILL]
FIRST	[[MARY JOHN] BILL]	[MARY JOHN]
BF	[[MARY JOHN] BILL]	[BILL]
FIRST	[MARY [JOHN BILL]]	MARY
BF	[MARY [JOHN BILL]]	[[JOHN BILL]]
FIRST	[] or "	Error
BF	[] or "	Error

LAST and BUTLAST (BL) work in the same way except that they work on the last element.

BUTFIRST

BUTFIRST *object* (BF) (operation)

BUTFIRST outputs all but the first element of *object*. BUTFIRST of the empty word or the empty list is an error.

Examples:

Operation	Output
BUTFIRST [EFFIE MANIATIS]	[MANIATIS]
BUTFIRST "DOGS	OGS
BUTFIRST [DOGS]	[] (the empty list)
BUTFIRST [THE DOGS]	[DOGS]
BUTFIRST [[THE A AN] [DOG CAT MOUSE] [BARKS MEOWS]]	[[DOG CAT MOUSE] [BARKS MEOWS]]
BUTFIRST "	Error
BUTFIRST []	Error

```
TO TRIANGLE :OBJECT
IF EMPTY? :OBJECT [STOP]
PR :OBJECT
TRIANGLE BUTFIRST :OBJECT
END
```

```
?TRIANGLE "STROLL
STROLL
TROLL
ROLL
OLL
LL
L
```

```
?TRIANGLE [KANGAROOS JUMP GRACEFULLY]
KANGAROOS JUMP GRACEFULLY
JUMP GRACEFULLY
GRACEFULLY
```

BUTLAST

BUTLAST *object* (BL) (operation)

BUTLAST outputs all but the last element of *object*.

Examples:

Operation	Output
BUTLAST [BARB G. MINGIE]	[BARB G.]
BUTLAST "FLOWER	FLOWE
BUTLAST [FLOWER]	
BUTLAST [[THE A AN] [BIRD BEE FLOWER]]	[[THE A AN]]
BUTLAST "	Error
BUTLAST []	Error

The input to the following procedure should be an adjective ending in Y:

```
TO COMMENT :WORD
PR SE [YOU ARE] :WORD
PR SE [I AM] WORD BUTLAST :WORD "IER
END

?COMMENT "FUNNY
YOU ARE FUNNY
I AM FUNNIER
```

FIRST

FIRST *object* (operation)

FIRST outputs the first element of *object*. FIRST of the empty word or the empty list is an error. Note that FIRST of a word is a single character; FIRST of a list can be a word or a list.

Examples:

Operation	Output
FIRST [HOUSE MOUSE LOUSE]	HOUSE
FIRST "HOUSE	H
FIRST [HOUSE]	HOUSE

Operation	Output
FIRST [[THE A AN] [UNICORN RHINO] [SWIMS FLIES GROWLS RUNS]]	[THE A AN]
FIRST "	Error
FIRST []	Error

```
TO PRINTDOWN :INPUT
IF EMPTY? :INPUT [STOP]
PR FIRST :INPUT
PRINTDOWN BF :INPUT
END

?PRINTDOWN "MOUSE
M
O
U
S
E
?PRINTDOWN [A STRAWBERRY SUNDAE]
A
STRAWBERRY
SUNDAE
```

ITEM

ITEM *integer object* (operation)

ITEM outputs the element of *object* whose position within *object* corresponds to *integer*. For example, if *integer* is 3, ITEM outputs the third element in the object. *Object* is a word or a list. An error occurs if *integer* is greater than the length of *object* or if *object* is the empty word or list.

Examples:

```
?MAKE "PETS [DOG CAT HAMSTER CANARY]
?PR ITEM 3 :PETS
HAMSTER
?PR ITEM 1 "APPLE
A
```

LAST

LAST *object* (operation)

LAST outputs the last element of *object*. LAST of the empty word or the empty list is an error.

Examples:

Operation	Output
LAST [SHARNEE MARIO RENAUD]	RENAUD
LAST "VANILLA	A
LAST [VANILLA]	VANILLA
LAST [[THE A] FLAVOR IS [VANILLA CHOCOLATE STRAWBERRY]]	[VANILLA CHOCOLATE STRAWBERRY]
LAST "	Error
LAST []	Error

```
TO PRINTBACK :INPUT
IF EMPTY? :INPUT [STOP]
PR LAST :INPUT
PRINTBACK BL :INPUT
END

?PRINTBACK "GANDALF
F
L
A
D
N
A
G
```

MEMBER

MEMBER *object1 object2* (operation)

MEMBER outputs the part of *object2* in which *object1* is the first element. If *object1* is not an element of *object2*, MEMBER outputs the empty list or the empty word. This operation is useful for accessing information in a file or for sorting long lists.

Examples:

```
?SHOW MEMBER "A [A B C]
[A B C]

?SHOW MEMBER "Bugs [Learn Bugs Logo]
[Bugs Logo]

?SHOW MEMBER [Piaget Papert] [Children !
Computers [Teach Activity] [Piaget Pap!
ert]]
[[Piaget Papert]]

?PR MEMBER "ABC "XYZABCDEF
ABCDEF
```

Putting Words and Lists Together

The operations that put words and lists together are

FPUT
LIST
LPUT
PARSE
SENTENCE (SE)
WORD

The following chart compares FPUT, LIST, LPUT, SENTENCE (SE), and WORD.

Operation	Input 1	Input 2	Output
FPUT	"LOGO	"TIME	Error
LIST	"LOGO	"TIME	[LOGO TIME]
LPUT	"LOGO	"TIME	Error
SE	"LOGO	"TIME	[LOGO TIME]
WORD	"LOGO	"TIME	LOGOTIME
FPUT	"TURTLE	[IS FUN]	[TURTLE IS FUN]
LIST	"TURTLE	[IS FUN]	[TURTLE [IS FUN]]
LPUT	"TURTLE	[IS FUN]	[IS FUN TURTLE]
SE	"TURTLE	[IS FUN]	[TURTLE IS FUN]
WORD	"TURTLE	[IS FUN]	Error
FPUT	[AND MORE]	[TO COME]	[[AND MORE] TO COME]
LIST	[AND MORE]	[TO COME]	[[AND MORE] [TO COME]]
LPUT	[AND MORE]	[TO COME]	[TO COME [AND MORE]]
SE	[AND MORE]	[TO COME]	[AND MORE TO COME]
WORD	[AND MORE]	[TO COME]	Error

Operation	Input 1	Input 2	Output
FPUT	"COMPUTERS	[]	[COMPUTERS]
LIST	"COMPUTERS	[]	[COMPUTERS []]
LPUT	"COMPUTERS	[]	[COMPUTERS]
SE	"COMPUTERS	[]	[COMPUTERS]
WORD	"COMPUTERS	[]	Error

FPUT

FPUT *object list* (operation)

The FPUT (for first put) operation outputs a new list formed by putting *object* at the beginning of *list*.

Examples:

Operation	Output
FPUT "HAMSTER [DOG CAT]	[HAMSTER DOG CAT]
FPUT [THE A AN] [CUP GLASS]	[[THE A AN] CUP GLASS]
FPUT "A []	[A]

LIST

LIST *object1 object2* (operation)
(LIST *object1 object2 object3 object4...*)

The LIST operation outputs a list whose elements are *object1*, *object2*, and so on.

Examples:

Operation	Output
LIST "ROSE [TULIP CHRYSANTHEMUM]	[ROSE [TULIP CHRYSANTHEMUM]]
(LIST "ROSE "TULIP "CHRYSANTHEMUM)	[ROSE TULIP CHRYSANTHEMUM]
LIST [A QUICK BROWN FOX] [LOOKS AT THE LAZY FROG]	[[A QUICK BROWN FOX] [LOOKS AT THE LAZY FROG]]
LIST "A []	[A []]

When LIST is used with a single input, parentheses are needed around the expression. For example:

```
?MAKE "ANIMALS "TOADS  
?SHOW (LIST :ANIMALS)  
[TOADS]
```

LPUT

LPUT *object list* (operation)

The LPUT (for last put) operation outputs a new list formed by putting *object* at the end of *list*.

Examples:

Operation	Output
LPUT "GERBIL [HAMSTER GUINEA.PIG]	[HAMSTER GUINEA.PIG GERBIL]
LPUT [THE A AN] [CAT ELEPHANT]	[CAT ELEPHANT [THE A AN]]
LPUT "A []	[A]
LAST LPUT "GERBIL [HAMSTER GUINEA.PIG]	GERBIL

The following procedure adds a new entry to an English-Spanish dictionary.

```
TO NEWENTRY :ENTRY
MAKE "DICTIONARY LPUT :ENTRY :DICTIONAR!
Y
END
```

```
?MAKE "DICTIONARY [[HOUSE CASA] [SPANIS!
H ESPANOL] [HOW COMO]]
?SHOW :DICTIONARY
[[HOUSE CASA] [SPANISH ESPANOL] [HOW CO!
MO]]
?NEWENTRY [TABLE MESA]
?SHOW :DICTIONARY
[[HOUSE CASA] [SPANISH ESPANOL] [HOW CO!
MO] [TABLE MESA]]
```

PARSE

PARSE *word* (operation)

PARSE outputs a list that is obtained from parsing *word*.
PARSE is useful for converting the output of READWORD into
a list.

Examples:

```
?SHOW PARSE "word
[word]
?MAKE "Input READWORD
dogs cats hamsters
?SHOW :Input
dogs cats hamsters
?SHOW PARSE :Input
[dogs cats hamsters]
```

SENTENCE

SENTENCE *object1 object2* (SE) (operation)
(SENTENCE *object1 object2 object3 ...*)

SENTENCE outputs a list made up of the contents in its inputs.

Examples:

Operation	Output
SENTENCE "PAPER "BOOKS	[PAPER BOOKS]
SENTENCE [PAPER] [BOOKS]	[PAPER BOOKS]
SENTENCE "APPLE [PEAR PLUM BANANA]	[APPLE PEAR PLUM BANANA]
SENTENCE [A QUICK BROWN FOX] [LOOKS AT THE LAZY FROG]	[A QUICK BROWN FOX LOOKS AT THE LAZY FROG]

The following procedure prints a birth announcement:

```
TO ANNOUNCE :FIRSTNAME :LASTNAME
PR [WE'RE HAPPY TO ANNOUNCE THE BIRTH OF
F]
PR (SE :FIRSTNAME "X. :LASTNAME)
PR [11 POUNDS 11 OZ]
END
```

```
?ANNOUNCE "ERIC "GEE\SILVERMAN
WE'RE HAPPY TO ANNOUNCE THE BIRTH OF
ERIC X. GEE-SILVERMAN
11 POUNDS 11 OZ
```

Further Examples:

Operation	Output
(SENTENCE "APPLE "PEAR "BANANA)	[APPLE PEAR BANANA]
(SENTENCE "MONET)	[MONET]
SENTENCE "MONET []	[MONET]

When you give SENTENCE a single input, you need to put parentheses around the expression. For example:

```
?MAKE "ANIMALS "KITTENS
?SHOW (SENTENCE :ANIMALS)
[KITTENS]
```

Compare the outputs when SENTENCE and LIST are applied to lists that contain other lists:

Operation

SENTENCE [THE DOG]
[LIKES [GREEN MICE]]

LIST [THE DOG] [LIKES
[GREEN MICE]]

Output

[THE DOG LIKES [GREEN
MICE]]

[[THE DOG] [LIKES [GREEN
MICE]]]

WORD

WORD *word1 word2* (operation)
(WORD *word1 word2 word3 ...*)

WORD outputs a word made up of its inputs.

Examples:**Operation**

WORD "SUN "SHINE
(WORD "CHEESE "BURG "ER)
WORD "BURG [ER]
WORD "S "MILES

Output

SUNSHINE
CHEESEBURGER
Error
SMILES

The procedure SUFFIX puts AY at the end of its input:

```
TO SUFFIX :WD
OUTPUT WORD :WD "AY
END
```

```
?PR SUFFIX "ANTEATER
ANTEATERAY
```

The essence of the procedure SUFFIX is incorporated into PIG and LATIN, which translate words and lists into a dialect of Pig Latin:

```
TO LATIN :SENT
IF EMPTY? :SENT [OP []]
OP SE PIG FIRST :SENT LATIN BF :SENT
END
```

```

TO PIG :WORD
IF MEMBERP FIRST :WORD [A E I O U Y] [O!
P WORD :WORD "AY]
OP PIG WORD BF :WORD FIRST :WORD
END

?PR LATIN [NO PIGS HAVE EVER SPOKEN PIG!
LATIN AMONG HUMANS]
ONAY IGSPAY AVEHAY EVERAY OKENSPAY IGPA!
Y ATINLAY AMONGAY UMANSWAY
?

```

Examining Words and Lists

The operations that you use in checking words and lists are

ASCII	EQUALP
BEFOREP	LISTP
CHAR	MEMBERP
COUNT	NUMBERP
EMPTYP	WORDP

ASCII

ASCII *character* (operation)

See also section "CHAR." Refer to Appendix F for a complete list of the ASCII codes.

ASCII outputs the American Standard Code for Information Interchange (ASCII) code for *character*. If the input word contains more than one character, ASCII uses only its first character.

Examples:

ASCII "B" outputs 66.

The procedure SECRETCODE makes a new word by using the Caesar cipher (adding 3 to each letter). Note that this example does not work with lowercase letters.

```

TO SECRETCODE :WD
IF EMPTYP :WD [OUTPUT " ]
OUTPUT WORD SECRETCODELET FIRST :WD SEC!
RETCODE BF :WD
END

```

```

TO SECRETCODELET :LET
MAKE "LETNUM (ASCII :LET) + 3
IF :LETNUM > ASCII "Z (MAKE "LETNUM :LET
TNUM - 26)
OUTPUT CHAR :LETNUM
END

?PR SECRETCODE "CAT
FDW
?PR SECRETCODE "CRAYON
FUDBRQ

```

BEFOREP

BEFOREP *word1 word2* (operation)

See Appendix F for a list of all ASCII codes and their meanings.

BEFOREP outputs TRUE if *word1* comes before *word2*. To make the comparison, Logo uses the ASCII codes of the characters in the words. Note that all uppercase letters come before all lowercase letters.

Examples:

Operation	Output
BEFOREP "A "a	TRUE
BEFOREP "apple "Zoo	FALSE
BEFOREP UPPERCASE "apple UPPERCASE "Zoo	TRUE

The following SORT procedure takes a list of words and outputs them alphabetically.

```

TO SORT :ARG :LIST
IF EMPTY? :ARG [OP :LIST]
MAKE "LIST INSERT FIRST :ARG :LIST
OP SORT BF :ARG :LIST
END

TO INSERT :A :L
IF EMPTY? :L [OP ( LIST :A )]
IF BEFOREP :A FIRST :L [OP FPUT :A :L]
OP FPUT FIRST :L INSERT :A BF :L
END

```


Try this:

```
MAKE "SORTLIST SORT [A D E F T C Z] []  
PR :SORTLIST  
A C D E F T Z
```

Then type

```
MAKE "SORTLIST SORT [FOO BAR BAZ] :SOR!  
TLIST  
PR :SORTLIST  
A BAR BAZ C D E F FOO T Z
```

CHAR

CHAR *integer* (operation)

Refer to Appendix F for a complete list of the ASCII codes.

The CHAR operation outputs the character whose ASCII code is *integer*. An error occurs if *integer* is not the ASCII code for any character.

Characters can be normal (white characters on black background) or inverse video (black characters on white background). The ASCII codes are organized as follows:

0 - 31	uppercase letters
32 - 47	punctuation
48 - 57	digits
58 - 63	punctuation
64 - 90	uppercase letters
91 - 96	punctuation
97 - 122	lowercase letters
123 - 127	punctuation
128 - 154	inverse-video uppercase letters
155 - 191	inverse-video digits and punctuation
192 - 218	special graphics characters
219 - 255	inverse-video lowercase letters

To change a normal character to inverse video, use the following procedure:

Examples:

```
TO CONVERT :CHAR
IF (ASCII :CHAR) > 127 [OP :CHAR]
IF OR (ASCII :CHAR) < 64 AND (ASCII :CHAR) > 96 (ASCII :CHAR) < 128 [OP CHAR 1!
28 + ASCII :CHAR] [OP CHAR 64 + ASCII :CHAR]
END
```

INVERSE displays a word in inverse video:

```
TO INVERSE :WORD
IF EMPTY :WORD [OP " ]
OP WORD CONVERT FIRST :WORD INVERSE BF !
:WORD
END

?PRINT INVERSE "YOGURT
YOGURT
?
```

COUNT

COUNT *object* (operation)

COUNT outputs the number of elements in *object*, which is a word or a list.

Examples:

Operation	Output
COUNT [A QUICK BROWN	4
COUNT [A QUICK BROWN FOX]	4
COUNT [A [QUICK BROWN] FOX]	3
COUNT "COMPUTER	8

```
?MAKE "CLASS [JOSE ANGELA WINIFRED LIN!  
G NORBERT BRIAN MARIA]  
?PR COUNT :CLASS  
7
```

The following procedure prints a random element of a word or a list:

```
TO RANPICK :OBJECT  
PR ITEM (1 + RANDOM COUNT :OBJECT) :OB!  
JECT  
END  
  
?RANPICK :CLASS  
BRIAN
```

EMPTYP

EMPTYP *object* (operation)

EMPTYP outputs TRUE if *object* is the empty word or the empty list; otherwise it outputs FALSE.

Examples:

Operation	Output
EMPTY 3	FALSE
EMPTY BUTFIRST "UNICORN	FALSE
EMPTY BUTLAST "U	TRUE
EMPTY BUTFIRST [UNICORN]	TRUE

This procedure, TALK, matches animal sounds to animals:

```
TO TALK :ANIMALS :SOUNDS
IF OR EMPTY :SOUNDS EMPTY :ANIMALS [P!
R [THAT'S ALL THERE IS!] STOP]
PR SE FIRST :ANIMALS FIRST :SOUNDS
TALK BF :ANIMALS BF :SOUNDS
END
```

```
?TALK [DGS BIRDS PIGS] [BARK CHIRP DIN!
K]
DGS BARK
BIRDS CHIRP
PIGS DINK
THAT'S ALL THERE IS!
```

The REVPRINT procedure reverses elements in a word or list.

```
TO REVPRINT :THING
IF EMPTY :THING [PR [] STOP]
TYPE LAST :THING
IF LISTP :THING [TYPE CHAR 32]
REVPRINT BL :THING
END
```

```
?REVPRINT "ELEPHANT
TNAHPELE
?REVPRINT "PUMPERNICKEL
LEKCINREPMUP
?REVPRINT [ALISON LOVES MATTHEW]
MATTHEW LOVES ALISON
?REVPRINT "OTTO
OTTO
```

See the list of infix-form operations in Chapter 9.

EQUALP

`EQUALP object1 object2` (operation)

`EQUALP` outputs `TRUE` if *object1* and *object2* are equal numbers, identical words, or identical lists; otherwise `EQUALP` outputs `FALSE`. This operation is equivalent to the equal sign (=).

Examples:

Operation	Output
<code>EQUALP "RED FIRST [RED YELLOW]</code>	<code>TRUE</code>
<code>EQUALP 100 50 * 2</code>	<code>TRUE</code>
<code>EQUALP [THE A AN] [THE A]</code>	<code>FALSE</code>
<code>EQUALP " []</code>	<code>FALSE</code> (the empty word and the empty list are not identical)

The following operation tells whether its first input (a character) is an element of its second input (a word).

```
TO INP :CHAR :WORD
IF EMPTY? :WORD [OUTPUT "FALSE]
IF EQUALP :CHAR FIRST :WORD [OUTPUT "TRUE]
OUTPUT INP :CHAR BUTFIRST :WORD
END

?PR INP "A "TEACUP
TRUE
?PR INP "I "SAUCER
FALSE
```

LISTP

LISTP *object* (operation)

LISTP outputs TRUE if *object* is a list; otherwise it outputs FALSE.

Examples:

Operation	Output
LISTP 3	FALSE
LISTP [3]	TRUE
LISTP []	TRUE
LISTP "	FALSE
LISTP [A B C [D E] [F [G]]]	TRUE
LISTP BUTFIRST "CHOCOLATE	FALSE
LISTP BUTFIRST [CHOCOLATE]	TRUE

MEMBERP

MEMBERP *object1 object2* (operation)

MEMBERP outputs TRUE if *object1* is an element of *object2*, otherwise it outputs FALSE.

Examples:

Operation	Output
MEMBERP 3 [2 5 [3] 6]	FALSE
MEMBERP 3 [2 5 3 6]	TRUE
MEMBERP [2 5] [2 5 3 6]	FALSE
MEMBERP "BIT "RABBIT	TRUE
MEMBERP [FLORIDA GEORGIA] [[FLORIDA GEORGIA] IOWA]	TRUE

Operation	Output
MEMBERP [FLORIDA GEORGIA] [FLORIDA GEORGIA IOWA]	FALSE
MEMBERP BUTFIRST "FOG [OE OF OG OH]	TRUE

The following procedure determines whether its input is a vowel:

```

TO VOWELP :LETTER
OUTPUT MEMBERP :LETTER [A E I O U]
END

?PR VOWELP "F
FALSE
?PR VOWELP "A
TRUE

```

NUMBERP

NUMBERP *object* (operation)

NUMBERP outputs TRUE if *object* is a number; otherwise it outputs FALSE.

Examples:

Operation	Output
NUMBERP 3	TRUE
NUMBERP [3]	FALSE
NUMBERP 3.14E23	TRUE
NUMBERP []	FALSE
NUMBERP "	FALSE
NUMBERP BUTFIRST 3165.2	TRUE
NUMBERP BUTFIRST [ELEPHANT]	FALSE

WORDP

WORDP *object* (operation)

WORDP outputs TRUE if *object* is a word; otherwise it outputs FALSE.

Note: In Logo, numbers are considered words.

Examples:

Operation	Output
WORDP 'ZAM	TRUE
WORDP [E GRESS]	FALSE
WORDP 3	TRUE
WORDP [3]	FALSE
WORDP []	FALSE
WORDP ''	TRUE
WORDP BUTFIRST 'BURG	TRUE
WORDP BUTFIRST [BURG]	FALSE

Changing the Case of Words

The operations that change the case of words are LOWERCASE and UPPERCASE.

LOWERCASE

LOWERCASE *word* (operation)

LOWERCASE outputs *word* in all lowercase letters.

Examples:

Operation	Output
LOWERCASE "Hello	hello
LOWERCASE "BIG	big

```
TO YESP :WORD
IF EQUALP LOWERCASE :WORD "yes [OP "TRUE]
E] [OP "FALSE]
END
?PR YESP "YES
TRUE
?PR YESP "SEVEN .
FALSE
```

UPPERCASE

UPPERCASE *word* (operation)

UPPERCASE outputs *word* in all uppercase letters.

Examples:

Operation	Output
UPPERCASE "Hello	HELLO
UPPERCASE "little	LITTLE

```
TO PRIMARYP :WORD
IF MEMBERP UPPERCASE :WORD [RED BLUE YE!
LLOW] [OP "TRUE] [OP "FALSE]
END
?PR PRIMARYP "red
TRUE
?PR PRIMARYP "green
FALSE
```



Variables

- 95** Variables: Some General Information
- 96** EDN
- 97** EDNS
- 98** LOCAL
- 99** MAKE
- 100** NAME
- 101** NAMEP
- 101** THING

Variables



This chapter gives you some general information about how Logo uses variables and then provides descriptions of the primitives that you use with variables. The primitives are

EDN
EDNS
LOCAL
MAKE
NAME
NAMEP
THING

Variables: Some General Information

For more information on variables, see Chapter 2.

A **variable** is a container that *holds* a Logo object. The container has a name and a value. The object held in the container is called the variable's **value**. You create a variable in one of two ways: either by using the MAKE or NAME command, or by using procedure inputs.

Logo has two kinds of variables: local variables and global variables. Variables used as procedure inputs are local to that procedure. They exist only as long as the procedure is running, and will disappear from your workspace after the procedure stops running.

Normally a variable created by MAKE is a global variable. The LOCAL command lets you change those variables into local variables. This can be very useful if you want to avoid cluttering up your workspace with unwanted variables.

EDN *name(list)* (command)

The EDN (for edit name) command starts up the Logo Editor with the named variable(s) and corresponding value(s). You can then edit these variable name(s) and value(s). When you exit the Editor, Logo reads the contents of the edit buffer as if you had typed each line from top level. Whatever variables and values have been changed in the Editor are changed in Logo.

Example:

?EDN "LANGUAGE,

The screen now looks like:

```
LOGO EDITOR
-----
MAKE "LANGUAGE [ENGLISH FRENCH SPANISH]
-----
^A accept, ^? help, ^-ESC cancel
```

You can now edit this variable as you wish and then press **(^)-(A)** to exit the Editor.

EDNS

EDNS

(command)

EDNS (for edit names) starts up the Logo Editor with all variable names and their values in it. You can then edit these variables' names and values. When you exit the Editor, Logo reads the contents of the edit buffer as if you had typed each line from top level. Whatever variables and values have been changed in the Editor are changed in Logo.

Example:

```
?PDNS  
MAKE "ANIMAL "GIBBON  
MAKE "SPEED 55  
MAKE "AIRCRAFT [JET HELICOPTER]  
?EDNS
```

The display now looks like:



```
LOGO EDITOR  
-----  
MAKE "ANIMAL "GIBBON  
MAKE "SPEED 55  
MAKE "AIRCRAFT [JET HELICOPTER]  
  
-----  
ó-A accept, ó-? help, ó-ESC cancel
```

EDNS

You can then edit the names so they look like this list:

```
MAKE "ANIMAL "GRYFFIN
MAKE "SPEED 55
MAKE "AIRCRAFT [JET HELICOPTER BLIMP]
```

Then,

```
?PONS
MAKE "ANIMAL "GRYFFIN
MAKE "SPEED 55
MAKE "AIRCRAFT [JET HELICOPTER BLIMP]
```

LOCAL

LOCAL *name(list)* (command)

The LOCAL command makes its input(s) local to the procedure within which the LOCAL occurs. A local variable is accessible only to that procedure and to procedures it calls; in this regard it resembles inputs to the procedure.

Example:

```
TO YESNO :QUESTION
LOCAL "ANSWER
PR :QUESTION
MAKE "ANSWER FIRST READLIST
IF EQUALP :ANSWER "YES [OUTPUT "TRUE]
OUTPUT "FALSE
END

TO GREET
PR [WHAT IS YOUR FULL NAME?]
MAKE "ANSWER READLIST
IF YESNO [DO YOU LIKE YOUR NAME?] [PR [!
THAT'S GOOD]] [PR [TOO BAD]]
PR SENTENCE [NICE TO MEET YOU, ] :ANSWE!
R
END

?GREET
WHAT IS YOUR FULL NAME?
ROBIN GLASS
DO YOU LIKE YOUR NAME?
```

```
NO
TOO BAD
NICE TO MEET YOU, ROBIN GLASS
```

Imagine what happens if the LOCAL command is omitted from YESNO. Each procedure uses a variable named ANSWER to hold the user's answer to a question. Because the variables are not local, the procedure YESNO destroys the value that GREET expects to have in that variable:

```
?GREET
WHAT IS YOUR FULL NAME?
ROBIN GLASS
DO YOU LIKE YOUR NAME?
NO
TOO BAD
NICE TO MEET YOU, NO
```

MAKE

MAKE *name object* (command)

The MAKE command puts *object* in *name*'s container, that is, it gives the variable *name* the value *object*.

Examples:

```
MAKE "JOB 259
?PR :JOB
259
?MAKE "JOB "WELDER
?PR :JOB
WELDER
?MAKE "WELDER 32
?PR :WELDER
32
?PR THING :JOB
32
?MAKE :JOB [SHARNEE CHAIT]
```

At this point :JOB is WELDER, and THING :JOB is [SHARNEE CHAIT].

```
?PRINT "JOB
JOB
?PRINT :JOB
```

```

WELDER
?PRINT THING "JOB
WELDER
?PRINT THING :JOB
SHARNEE CHAIT

TO WEATHER
PR [WHAT'S THE WEATHER LIKE TODAY?]
MAKE "ANSWER READLIST
IF :ANSWER = [RAINING] [PR [I WISH IT W!
OULD STOP RAINING] STOP]
IF :ANSWER = [SUNNY] [PR [I HOPE IT STA!
YS SUNNY] STOP]
PR (SE [I WONDER IF IT WILL BE] :ANSWER!
"Tomorrow.) .
END

?WEATHER
WHAT'S THE WEATHER LIKE TODAY?
SUNNY
I HOPE IT STAYS SUNNY
?WEATHER
WHAT'S THE WEATHER LIKE TODAY?
CLOUDY
I WONDER IF IT WILL BE CLOUDY TOMORROW.
?WEATHER
WHAT'S THE WEATHER LIKE TODAY?
RAINING
I WISH IT WOULD STOP RAINING

```

NAME

NAME *object name* (command)

The NAME command puts *object* in *name*'s container, that is, it gives the variable *name* the value *object*.

Examples:

```
?NAME 259 "JOB
?PR :JOB
259
?NAME "WELDER "JOB
?PR :JOB
WELDER
```

NAME is equivalent to MAKE with the order of the inputs reversed. Thus NAME "WELDER "JOB has the same effect as MAKE "JOB "WELDER.

NAMEP

NAMEP *word* (operation)

NAMEP outputs TRUE if *word* has a value, that is, if *word* exists; it outputs FALSE otherwise.

Examples:

```
?PR NAMEP "ANIMAL
FALSE
?MAKE "ANIMAL "AARDVARK
?PR :ANIMAL
AARDVARK
?PR NAMEP "ANIMAL
TRUE
```

The procedure INC, listed with the THING operation that follows, shows a use of NAMEP.

THING

THING *name* (operation)

THING outputs the thing in the container *name*, that is, the value of the variable *name*. THING "ANY is equivalent to :ANY.

Example:

This procedure increments (adds 1 to) the value of a variable:

```
TO INC :X
IF NOT NAMEP :X [STOP]
IF NUMBERP THING :X [MAKE :X 1 + THING !
:X]
END
```

Note the use of MAKE :X rather than MAKE "X. It is not X that's being incremented. The value of X is not a number, but the name of another variable. It is that second variable that is incremented.

For other examples, see section "MAKE."

```
?MAKE "TOTAL 7
?PR :TOTAL
7
?INC "TOTAL
?PR :TOTAL
8
?INC "TOTAL
?PR :TOTAL
9
```

Arithmetic Operations

- 105** Arithmetic Operations: Some General Information
- 107** How Logo Evaluates Math Operations
- 108** Prefix-Form Operations
 - 108** ARCTAN
 - 109** COS
 - 109** DIFFERENCE
 - 110** FORM
 - 111** INT
 - 112** INTQUOTIENT
 - 112** PRODUCT
 - 113** QUOTIENT
 - 113** RANDOM
 - 114** REMAINDER
 - 115** RERANDOM
 - 116** ROUND
 - 116** SIN
 - 117** SQRT
 - 117** SUM
- 118** Infix-Form Operations
 - 119** Plus Sign
 - 119** Minus Sign
 - 120** Multiplication Sign
 - 121** Division Sign
 - 121** Less Than Sign
 - 122** Equal Sign
 - 122** Greater Than Sign

Arithmetic Operations



This chapter presents all the Logo operations that manipulate numbers. Logo has two kinds of notation for expressing arithmetic operations: prefix notation and infix notation. **Prefix notation** means that the name of the procedure comes before its inputs. With **infix notation**, the name of the procedure goes between its inputs, not before them.

This chapter contains

- a general introduction to Logo's arithmetic operations
- descriptions of the prefix-form operations
- descriptions of the infix-form operations.

Arithmetic Operations: Some General Information

Logo has two kinds of numbers: integers and decimals.

3 is an integer

3.14 and 3. are decimal numbers

Logo provides primitives that let you add, subtract, multiply, and divide numbers. You can find sines, cosines, arctangents, and square roots; and you can test whether a number is equal to, less than, or greater than another number.

Scientific notation is a way of expressing a number with an exponent.

Chapter 7, Words and Lists, describes the EQUALP primitive.

The result of an arithmetic operation can be either an integer or a decimal, depending on the operation:

- INT, INTQUOTIENT, RANDOM, REMAINDER, and ROUND always output integers.
- ARCTAN, COS, SIN, SQRT, QUOTIENT, and / always output decimal numbers.
- The rest output integers if all their inputs are integers, and decimal numbers if one or more of their inputs are decimal numbers (+, -, *).

Thus $7 / 2$ is 3.5 (a decimal number), but INTQUOTIENT $7 / 2$ is 3 (an integer).

Further, $3.5 + 6.5$ is 10.0 (a decimal number), but $3 + 7$ is 10 (an integer). Note that $3 + 7.0$ is 10.0 (a decimal number).

The largest possible integer in Logo is 2147483647, which is $2^{31}-1$; the smallest is -2147483647, which is $-(2^{31}-1)$.

Decimal numbers have six digits of accuracy and can include an exponent that ranges from 38 to -38. Logo uses exponential form (scientific notation) to represent numbers that cannot be written as just six digits. Here are some examples:

1.0E10 means 10^{10} , or 10,000,000,000

1.0N10 means 10^{-10} , or 0.00000000001

Notice that the *N* indicates a negative exponent.

Logo rounds off a decimal number if it contains more than six digits. For example, the number 2718281828459.045 is converted to 2.71828E12.

Addition, subtraction, multiplication, and division are available in infix notation. The name of an infix procedure goes between its inputs, not before them. Logo also provides addition and multiplication in prefix form as operations taking two or more inputs. For example, the following expressions are equivalent:

$2 + 1$
SUM 2 1

In addition to those primitives listed here, the primitive EQUALP is often used in conjunction with arithmetic operations. EQUALP is equivalent to the infix operation equal sign (=), described in this chapter.

How Logo Evaluates Math Operations

When a Logo line has several math operations, Logo evaluates them according to the operations' precedence. The order of precedence from highest to lowest is as follows:

-	Unary minus. Indicates a negative number (-3) or the additive inverse of the input (-XCOR).
*, /	Multiplication and division.
+, -	Addition and subtraction.
>, <, =	Greater than, less than, equals.
Other math operations	This group includes user-defined operations, as well as primitive operations such as SIN, DIFFERENCE, and SUM.

Thus,

```
COS 25 + 10
```

is read as

```
COS (25 + 10)
```

You can change the order of precedence just listed by using parentheses. Logo follows the standard mathematical practice of performing operations enclosed in parentheses before others. If there are several operations within one set of parentheses, Logo uses the order of precedence just given.

Example:

```
?PR 2 * 4 + 8 / 4  
10.0  
?PR 2 * (4 + 8 / 4)  
12.0  
?PR (2 * 4 + 8) / 4  
4.0
```

Prefix-Form Operations

This section explains the prefix-form operations, which appear in this order:

ARCTAN	RANDOM
COS	REMAINDER

DIFFERENCE	RERANDOM
FORM	ROUND
INT	SIN
INTQUOTIENT	SQRT
PRODUCT	SUM
QUOTIENT	

ARCTAN

ARCTAN *number* (operation)

The **arctangent** of a number is an angle whose tangent is that number.

ARCTAN outputs the arctangent (inverse tangent) of *number*. The output is a decimal number and is in degrees, not radians. The output of ARCTAN is always a number between -90 and 90. If *number* is close to -1, the output may be unreliable.

Examples:

Operation	Output
ARCTAN 2	63.4348
ARCTAN 444	89.871

The following procedures define ARCSIN and ARCCOS:

```
TO ARCSIN :X
OUTPUT ARCTAN :X / (SQRT 1 - :X * :X)
END
```

```
TO ARCCOS :X
OUTPUT ARCTAN (SQRT 1 - :X * :X) / :X
END
```

COS

COS *degrees* (operation)

The COS operation outputs the cosine of *degrees*. The output is a decimal number. *Degrees* cannot be greater than 4.19E6. If it is, an error occurs.

Examples:

Operation	Output
COS 60	0.5
COS 30	0.866026

Here is a definition of the tangent function:

```
TO TAN :ANGLE
OUTPUT (SIN :ANGLE) / COS :ANGLE
END

?PR TAN 45
1.0
```

DIFFERENCE

DIFFERENCE *number1 number2* (operation)

DIFFERENCE outputs the result of subtracting *number2* from *number1*.

Examples:

Operation	Output
DIFFERENCE 7 1	6
DIFFERENCE (5+6) (3*7)	-10
DIFFERENCE 10 5	5
DIFFERENCE 6.3 107.4	-101.1

FORM

FORM *number field precision* (operation)

FORM outputs *number* as a word in the number of spaces indicated by *field*, with *precision* digits after the decimal point. The input for *field* must be an integer from 1 through 128. The input for *precision* must be an integer from 0 through 6.

If *number* is too small to use the full *field* spaces, Logo adds blank space before the number. Note that the decimal point (.) and the minus sign (-) both count as an element in *field*.

FORM works with all integers, but only some decimal numbers. These are

-999999.0 through -0.000001
0.000001 through 999999.0

Logo prints all other decimal numbers in scientific notation, and these cannot be handled by FORM. Instead, FORM outputs the number right justified in a word with *field* characters.

Note: Decimal numbers have only six significant digits no matter how many you enter. Even when numbers are used in conjunction with FORM, they are reduced to six significant digits before being passed to FORM.

An error occurs if *field* is 0 or is less than the number of digits before the decimal point in *number*. If *precision* is 0, FORM outputs *number* as an integer. Trailing zeros are added if *precision* is greater than the number of digits after the decimal point in *number*.

If FORM outputs a number with fewer digits after the decimal point than the input number, the last digit is the result of truncating the missing digits.

FORM is useful when you are trying to print columns of numbers in an unvarying format.

Examples:

Operation	Output
FORM 27.33 10 1	27.3
FORM 27.33 10 3	27.330
FORM 2.7E20 15 2	2.7E20
?MAKE "A -8.8888	
?PR FORM :A 9 3	
	8.888

INT

INT *number* (operation)

The INT operation outputs the integer portion of *number*. Logo removes the decimal portion of the number, if one exists. The maximum integer is 2,147,483,647.

Examples:

Operation	Output
INT 5.2129	5
INT 5.5129	5
INT 5	5
INT -5.8	-5
INT -12.3	-12

The procedure INTP tells whether its input is an integer:

```
TO INTP :N
  IF NOT NUMBERP :N [OP [NOT A NUMBER]]
  OP (COUNT :N) = (COUNT INT :N)
END

?PRINT INTP 17
TRUE
?PRINT INTP 100 / 8
FALSE
?PRINT INTP "ONE
```

See also section "ROUND."


```
NOT A NUMBER
?PRINT INTP SQRT 50
FALSE
```

INTQUOTIENT

INTQUOTIENT *integer1 integer2* (operation)

INTQUOTIENT outputs the result of dividing *integer1* by *integer2*, truncated to an integer. An error occurs if *integer2* is 0. If either input is a decimal number, it is truncated.

Examples:

Operation	Output
INTQUOTIENT 12 5	2
INTQUOTIENT -12 5	-2
INTQUOTIENT 9 2	4
INTQUOTIENT 3 0	Error

PRODUCT

PRODUCT *number1 number2* (operation)
(PRODUCT *number1 number2 number3 ...*)

PRODUCT outputs the product of its inputs. It is equivalent to the * infix-form operation. With one input, PRODUCT outputs its input.

Examples:

Operation	Output
PRODUCT 6 2	12
(PRODUCT 2 3 4)	24
PRODUCT 2.5 4	10.0

```
TO CUBE :NUM
OP (PRODUCT :NUM :NUM :NUM)
END
```

```
?PR CUBE 2
8
```

QUOTIENT

QUOTIENT *number1 number2* (operation)

QUOTIENT outputs the result of dividing *number1* by *number2*. It is equivalent to the / infix-form operation. *Number2* must not be 0. If it is, an error occurs.

Examples:

Operation	Output
QUOTIENT 12 5	2.4
QUOTIENT -12 5	-2.4
QUOTIENT 6 2.5	2.4
QUOTIENT 3.2 0	Error

RANDOM

RANDOM *integer* (operation)

RANDOM outputs a random non-negative integer less than *integer*.

Example:

RANDOM 6 can output 0, 1, 2, 3, 4, or 5. The following program simulates a roll of a six-sided die:

```
TD D6
OUTPUT 1 + RANDOM 6
END
?PR D6
3
?PR D6
5
?PR D6
3
```

REMAINDER

REMAINDER *integer1 integer2* (operation)

REMAINDER outputs the remainder obtained when *integer1* is divided by *integer2*. The remainder is always an integer. If *integer1* and *integer2* are integers, this is *integer1 mod integer2*. If *integer1* and *integer2* are not integers, they are truncated. *integer2* must not be 0. If it is, an error occurs.

Examples:

Operation	Output
REMAINDER 12 10	2
REMAINDER 12 5	2
REMAINDER 12 15	12
REMAINDER -12 5	-2

The following procedure tells whether its input is even:

```
TO EVENP :NUMBER
OP 0 = REMAINDER :NUMBER 2
END

?PR EVENP 5
FALSE
?PR EVENP 12462
TRUE
```

The following more general procedure tells whether its first input is a divisor of its second input:

```
TO DIVISORP :A :B
OP 0 = REMAINDER :B :A
END

?PR DIVISORP 3 15
TRUE
?PR DIVISORP 4 15
FALSE
```

RERANDOM

RERANDOM

(command)

RERANDOM makes RANDOM behave reproducibly: after you run RERANDOM, calls to RANDOM generate the same sequences of numbers from the beginning each time.

Example:

```
TO DICE :THROWS
IF :THROWS = 0 [STOP]
PR 1 + RANDOM 6
DICE :THROWS - 1
END
```

```
?DICE 6
```

```
3
```

```
2
```

```
6
```

```
6
```

```
3
```

```
1
```

```
?DICE 6
```

```
5
```

```
5
```

```
5
```

```
1
```

```
3
```

```
1
```

```
?RERANDOM
```

```
?DICE 6
```

```
3
```

```
2
```

```
6
```

```
6
```

```
3
```

```
1
```

```
?RERANDOM
?DICE 6
3
2
6
6
3
1
```

ROUND

ROUND *number* (operation)

See also the examples in section "INT."

The ROUND operation outputs *number* rounded off to the nearest integer. The maximum integer is 2,147,483,647.

Examples:

Operation	Output
ROUND 5.2129	5
ROUND 5.5129	6
ROUND .5	1
ROUND -5.8	-6
ROUND -12.3	-12

SIN

SIN *degrees* (operation)

See also section "COS."

The SIN operation outputs the sine of *degrees*. *Degrees* cannot be greater than 4.19E6. If it is, an error occurs.

Example:

SIN 30 outputs 0.5

SQRT

SQRT *number* (operation)

The SQRT operation outputs the square root of *number*. The value *number* must not be negative or an error will occur.

Examples:

Operation	Output
SQRT 25	5.0
SQRT 259	16.0935

The following procedure outputs the distance from the turtle's position to HOME.

```
TO FROM.HOME
OP SQRT SUM XCOR * XCOR YCOR * YCOR
END
```

The procedure DISTANCE takes any two positions as inputs, and outputs the distance between them:

```
TO DISTANCE :POS1 :POS2
OP SQRT SUM SQ ((FIRST :POS1) - FIRST :!
POS2) SQ ((LAST :POS1) - LAST :POS2
END
```

```
TO SQ :N
OP :N * :N
END
```

```
?PR DISTANCE [-70 10] [50 60]
130.0
```

SUM

SUM *number1 number2* (operation)
(SUM *number1 number2 number3 ...*)

The SUM operation outputs the sum of its inputs. SUM is equivalent to the + infix-form operation.

With one input, SUM outputs its input.

Examples:

Operation	Output
SUM 5 2	7
(SUM 1 3 2 -1)	5
SUM 2.3 2.561	4.861

Infix-Form Operations

This section explains the infix-form operations, which appear in this order:

+
-
*
/
<
=
>

Note that because the symbols for these operations are word-separators, spaces are optional before and after all of them except the slash (see the following explanation). Thus the following are equivalent:

2 + 5
2+5

The only exception is the slash (/), which indicates division. You must always put spaces before and after the slash character.

4 / 8
3 / 9

The reason for this is that the / sign is used in pathnames.

Plus Sign

number1 + *number2*

(infix-form operation)

The plus sign (+) outputs the sum of its inputs. It is equivalent to SUM, which is a prefix-form operation.

Examples:

Operation	Output
5 + 2	7
1 + 3 + 2 + 1	7
2.54 + 12.3	14.84

Minus Sign

number1 - *number2*

(infix-form operation)

The minus sign (-) outputs the result of subtracting *number2* from *number1*. If *number1* is missing and there is no space after the minus sign, it outputs the opposite of *number2* (0-*number2*).

Examples:

```
?PR 7 - 1
6
?PR 7-1
6
?PR PRODUCT 7 -1
-7
?PR -3
-3
?PR - 3
-3
?PR -3 - -2
-1
```

The procedure ABS outputs the absolute value of its input:

```
TO ABS :NUM
OP IF :NUM < 0 [ -:NUM ] [ :NUM ]
END
```

```
?PR ABS -35
35
?PR ABS 35
35
```

NEAR tells whether two numbers are close in value:

```
TO NEAR :A :B
OP (ABS :A - :B) < .01
END
```

```
?PR NEAR XCOR 100
TRUE
?PR XCOR
99.9934
```

For more detailed information on how Logo treats the minus sign, see Appendix E, Parsing.

Note that there is a potential ambiguity between the minus sign with one input and the minus sign with two inputs. Logo resolves this ambiguity as follows:

```
7-1 is 6
7 - 1 is also 6
7- 1 is also 6
But 7 -1 is a pair of numbers (7 and -1).
```

Multiplication Sign

*number1 * number2* (infix-form operation)

The asterisk (*) outputs the product of its inputs. It is equivalent to PRODUCT, which is a prefix-form operation.

Examples:

Operation	Output
6 * 2	12
2 * 3 * 4	24
1.3 * 1.3	1.69

The procedure FACTORIAL outputs the factorial of its input. For example, FACTORIAL 5 outputs the product of $5 * 4 * 3 * 2 * 1$.

```
TO FACTORIAL :N
IF :N = 0 [OP 1] [OP :N * FACTORIAL :N-1]
END

?PR FACTORIAL 4
24
?PR FACTORIAL 1
1
```

Division Sign

number1 / *number2* (infix-form operation)

The slash (/) outputs *number1* divided by *number2*. It is the same as the QUOTIENT operation. *Number2* must not be 0.

Examples:

Operation	Output
6 / 3	2.0
8 / 3	2.66667
2.5 / 3.8	0.657895
0 / 7	0.0
7 / 0	Error

Less Than Sign

number1 < *number2* (infix-form operation)

The less than sign (<) outputs TRUE if *number1* is less than *number2*; otherwise it outputs FALSE. It is similar to the BEFOREP operation but takes only numbers as inputs.

Examples:

Operation	Output
2 < 3	TRUE
-7 < -10	FALSE

The BEFOREP operation is described in Chapter 7.

The equal sign is equivalent to EQUALP, which is described in Chapter 7.

Equal Sign

object1 = *object2* (infix-form operation)

The equal sign (=) outputs TRUE if *object1* and *object2* are equal numbers, identical words, or identical lists; otherwise it outputs FALSE.

Note that the use of parentheses affects how Logo evaluates the equal sign, as shown in this example:

FIRST "3.1416 = 3 outputs F.
(FIRST "3.1416) = 3 outputs TRUE.

In the first of these examples, Logo evaluates whether 3.1416 equals 3 before it executes FIRST.

Examples:

Operation	Output
100 = 50*2	TRUE
3 = FIRST "3.1416	TRUE
[THE A AN] = [THE A]	FALSE
7. = 7	TRUE (a decimal number is equivalent to the corresponding integer)
" = []	FALSE (the empty word and the empty list are not identical)

Greater Than Sign

number1 > *number2* (infix-form operation)

The greater than sign (>) outputs TRUE if *number1* is greater than *number2*; otherwise it outputs FALSE.

Examples:

Operation	Output
4 > 3	TRUE
-10 > -7	FALSE

Conditionals and Flow of Control

- 125** Flow of Control: Some General Information
- 126** Using Conditionals
 - 126** IF
 - 127** IFFALSE
 - 128** IFTRUE
 - 128** TEST
- 129** Interrupting Procedures
 - 130** CO
 - 130** OUTPUT
 - 131** PAUSE
 - 132** STOP
 - 132** WAIT
- 133** Transferring Control and Repeating Instructions
 - 133** CATCH
 - 135** ERROR
 - 136** GO
 - 137** LABEL
 - 137** REPEAT
 - 138** RUN
 - 140** THROW
- 140** Debugging Programs
 - 141** STEP
 - 141** TRACE
 - 143** UNSTEP
 - 143** UNTRACE
- 144** Special Control Characters
 - 144** OPEN APPLE-ESC
 - 144** CONTROL-W
 - 144** CONTROL-Z

Conditionals and Flow of Control



This chapter presents the primitives and special control characters that you use to change Logo's normal way of executing a procedure. The primitives and special characters appear in five groups:

- primitives called **conditionals** that tell Logo to carry out different instructions, depending on whether a condition is met
- primitives that interrupt a procedure before it has finished executing
- primitives that tell Logo to repeat instructions a certain number of times or to jump or transfer control to some other instruction
- primitives for debugging programs
- special control characters that interrupt Logo's flow of control, either temporarily or permanently.

Flow of Control: Some General Information

Logo reads procedure definitions line by line, following the instructions given in each line. If a procedure contains a subprocedure, Logo reads the lines of the subprocedure before continuing in the superprocedure. **Flow of control** refers to the order in which Logo follows instructions. There are times when you want to alter Logo's normal flow of control. You can do so with any of these methods:

Conditionals tell Logo to do one thing if such-and-such is true; otherwise, do something else.

Repetition	tells Logo to run a list of instructions one or more times.
Halting	tells Logo to stop this procedure before it reaches the end.
Pausing	tells Logo to interrupt this procedure while it's running, but let it resume afterwards.

Using Conditionals

Conditionals allow Logo to carry out different instructions, depending on whether a condition is met. You use operations that output TRUE or FALSE, called **predicates**, to create this condition. The result of the operation is the first input to one of the IF primitives. The primitives for writing conditionals are

IF
IFFALSE
IFTRUE
TEST

The three primitives TEST, IFTRUE, and IFFALSE perform exactly the same function as the single primitive IF. Which you use is a matter of convenience and personal taste.

IF

IF *predicate list1* (command or operation)
IF *predicate list1 list2*

If *predicate* is TRUE, Logo runs *list1*. If *predicate* is FALSE, Logo runs *list2* (if present). In either case, if the selected list outputs something, the IF is an operation. If the list outputs nothing, the IF is a command.

Examples:

The procedure DECIDE appears in three equivalent ways. The first two use IF as a command—one version with two inputs to IF, one with three inputs. The third version of DECIDE uses IF (with three inputs) as an operation.

IF as a command:

```
TO DECIDE
IF 0 = RANDOM 2 [OP "YES]
OP "NO
END
```

```
TO DECIDE
IF 0 = RANDOM 2 [OP "YES] [OP "NO]
END
```

IF as an operation:

```
TO DECIDE
OUTPUT IF 0 = RANDOM 2 ["YES] ["NO]
END
```

IFFALSE

IFFALSE *list* (IFF) (command)

IFFALSE runs *list* if the result of the most recent TEST was FALSE, otherwise it does nothing. Note that if TEST has not been run in the same procedure or a superprocedure, or from top level, IFFALSE does nothing.

Example:

```
TO QUIZ
PRINT [WHAT IS THE CAPITAL OF NEW JE!
RSEY?]
TEST "TRENTON = UPPERCASE READWORD
IFTRUE [PRINT "CORRECT!]
IFFALSE [PRINT "WRONG]
END

?QUIZ
WHAT IS THE CAPITAL OF NEW JERSEY?
NEWARK
WRONG
```

See section "TEST."

See section "TEST."

IFTRUE

IFTRUE *list* (IFT) (command)

IFTRUE runs *list* if the result of the most recent TEST was TRUE, otherwise it does nothing. Note that if TEST has not been run in the same procedure or a superprocedure, or from top level, IFTTRUE does nothing.

Example:

```
TO QUIZZ
PR [WHO IS THE GREATEST?]
TEST "ME = UPPERCASE READWORD
IFTRUE [PR [RIGHT ON] STOP]
PR [NO, TRY AGAIN]
QUIZZ
END

?QUIZZ
WHO IS THE GREATEST?
GEORGE
NO, TRY AGAIN
WHO IS THE GREATEST?
ME
RIGHT ON
```

TEST

TEST *predicate* (command)

TEST remembers whether *predicate* is TRUE or FALSE for subsequent use by IFTTRUE or IFFALSE. Each TEST is local to the procedure in which it occurs.

Example:

```
TO SHORTQUIZ
PR [HOW ARE YOU?]
TEST "FINE = UPPERCASE READWORD
IFTRUE [PR [I'M GLAD TO HEAR IT]]
END

?SHORTQUIZ
HOW ARE YDU?
LOUSY

?SHORTQUIZ
HOW ARE YOU?
FINE
I'M GLAD TO HEAR IT
```

Interrupting Procedures

The commands for stopping a procedure, either temporarily or permanently, are

```
CO
OUTPUT
PAUSE
STOP
WAIT
```

To **halt** a procedure before it reaches an END statement, use the STOP and OUTPUT commands. Logo then transfers control back to the calling procedure (the procedure using it) or to top level. OUTPUT can communicate information to the calling procedure. Note that these commands (STOP and OUTPUT) halt only the procedure they appear in.

To interrupt a procedure without permanently stopping it, use the PAUSE and WAIT commands. PAUSE applies mainly to debugging. You can use WAIT for time-critical code like animated graphics.

READCHAR, READCHARS, READLIST, and READWORD are described in Chapter 13.

Note: Other primitives such as READCHAR, READCHARS, READLIST, and READWORD also temporarily interrupt procedures.

CO

CO (command)

The CO (for continue) command resumes running of a procedure after a PAUSE or `(CONTROL)-Z`, continuing from wherever the procedure paused.

OUTPUTOUTPUT *object* (OP) (command)

The OUTPUT command is meaningful only when it is within a procedure, not at top level. It makes *object* the output of your procedure and returns control to the caller. Note that although OUTPUT is itself a command, the procedure containing it is an operation because it has an output. Compare with STOP.

Examples:

```
TO MARK.TWAIN
OUTPUT [SAMUEL CLEMENS]
END
```

```
?PR SE MARK.TWAIN [IS A GREAT AUTHOR]
SAMUEL CLEMENS IS A GREAT AUTHOR
```

WHICH outputs the position of an element in a list:

```
TO WHICH :MEMBER :LIST
IF NOT MEMBERP :MEMBER :LIST [OUTPUT 0]
IF :MEMBER = FIRST :LIST [OUTPUT 1]
OUTPUT 1 + WHICH :MEMBER BF :LIST
END
```

```
?MAKE "VOWELS [A E I O U]>
?PR WHICH "E :VOWELS
2
?PR WHICH "U :VOWELS
5
?PR WHICH "W :VOWELS
0
```

An alternate version of the absolute value operation appears in the discussion of the minus sign (-) operation in Chapter 9.

Here is one definition of the absolute-value operation:

```
TO ABS :N
IF :N < 0 [OUTPUT -:N] [OUTPUT :N]
END
```

PAUSE

PAUSE (command or operation)

The PAUSE command is meaningful only when it is within a procedure, not at top level. It suspends running of the procedure and tells you that you are pausing; you can then type instructions interactively. To indicate that you are in a pause and not at top level, the prompt character changes to the name of the procedure you were in, followed by a question mark. During a pause, (⏏)(ESC) does not work; the only way to return to top level during a pause is to run THROW TOPLEVEL.

All local variables are accessible during a pause. See PR :MAX in the following example.

The procedure may be resumed by typing CD.

Examples:

```
TO WALK :MAX
RT RANDOM 360
FD RANDOM :MAX
PR POS
PAUSE
WALK :MAX
END

?WALK 100
60.4109 -13.947
PAUSING...
WALK?PR HEADING
103
WALK?PR :MAX
100
WALK?CD
68.4381 2.1059
```

STOP

STOP

(command)

The STOP command stops the procedure that is running and returns control to the caller. This command is meaningful only when it is within a procedure—not at top level. Note that a procedure containing STOP is a command. Compare STOP with OUTPUT.

Examples:

```
TO COUNTDOWN :NUM
PR :NUM
IF :NUM = 0 [PR [BLAST OFF!] STOP]
COUNTDOWN :NUM - 1
END

?COUNTDOWN 4
4
3
2
1
0
BLAST OFF!
```

WAIT

WAIT *integer*

(command)

WAIT tells Logo to wait for *integer* 60ths of a second.

Example:

The procedure REPORT keeps printing the turtle's position as it moves randomly. It uses WAIT to give you time to read the position.

```
TO REPORT
RT 10 * RANDOM 36
FD 10 * RANDOM 10
PR POS
WAIT 100
REPORT
END
```



```
?CS HT
?REPORT
0. 90.
-46.9846 72.889
-41.7752 43.3547
.
```

Transferring Control and Repeating Instructions

This section describes the primitives you use to repeat instructions and to transfer control to some other instruction. The primitives in this section are

CATCH
ERROR
GO
LABEL
REPEAT
RUN
THROW

Two pairs of primitives tell Logo to jump or transfer control to some other instruction. To transfer control to an instruction in the same procedure, use GO and LABEL. To transfer control to another procedure, use CATCH and THROW. You can use CATCH and THROW to stop an entire program.

See this chapter's section "RUN" for examples of some complex repetitive procedures.

Repetition can be done by using REPEAT or a recursive procedure. There are many examples of such procedures throughout this manual.

CATCH

CATCH *name list* (command)

CATCH runs *list*. If a THROW *name* command is called while *list* is run, control returns to the first statement after the CATCH. The *name* is used to match up a THROW with a CATCH. For instance, CATCH "CHAIR [whatever] catches a THROW "CHAIR but not a THROW "TABLE.

There is one special case. CATCH "ERROR catches an error that would otherwise print an error message and return to top level. If an error is caught, the message that Logo would normally print isn't printed. See the explanation of ERROR in this chapter to find out how to tell what the error was.

Examples:

The procedure SNAKE reads *numbers* typed in by you, and uses them as distances to move the turtle. It turns the turtle between moves. If you type something other than a number, the program (using its READNUM subprocedure) prints an appropriate message and continues working.

```
TO SNAKE (superprocedure)
CATCH "NOTNUM [SLITHER]
SNAKE
END
```

```
TO SLITHER (subprocedure)
PR [TYPE A NUMBER, PLEASE.]
FD READNUM
RT 10
END
```

```
TO READNUM (subprocedure)
LOCAL "LINE
MAKE "LINE READLIST
IF NOT NUMBERP FIRST :LINE [PR [THAT'S !
NOT A NUMBER.] THROW "NOTNUM]
IF NOT EMPTY? BF :LINE [PR [ONLY ONE NU!
MBER, PLEASE!] THROW "NOTNUM]
OUTPUT FIRST :LINE
END
```

Notice that STOP in place of THROW "NOTNUM would have returned to SLITHER, not to SNAKE.

The procedure DOIT runs *instructions* typed in by you. When an error occurs, Logo does not display the standard error message and does not return to top level; instead, it displays THAT STATEMENT IS INCORRECT and lets you continue typing instructions.

```
TO DOIT
CATCH "ERROR [DOIT1]
PR [THAT STATEMENT IS INCORRECT]
DOIT
END
```

```
TO DOIT1
RUN READLIST
DOIT1
END
```

```
?DOIT
PR 3 + 5
8
PR 12 - 7
THAT STATEMENT IS INCORRECT
PR 12 - 7
5
THROW "TOPLEVEL
```

ERROR

ERROR (operation)

ERROR outputs a four-element list containing information about the most recent error that has not had a message printed or output by ERROR. If there was no such error, ERROR outputs the empty list. The elements in the list are

- a unique number identifying the error
- a message explaining the error
- the name of the primitive causing the error, if any
- the name of the procedure within which the error occurred (the empty list, if top level).

Appendix A has a complete list of error numbers and their meanings.

Logo runs THROW "ERROR whenever an error occurs during the execution of a procedure. Control passes to top level unless a CATCH "ERROR has been run. When an error is caught in this way, no error message is printed, and you can design your own.

Example:

```
TO SAFESQUARE :SIDE
CATCH "ERROR [REPEAT 4 [FD :SIDE RT 90]!
STOP]
PR ERROR
END
```

```
?SAFESQUARE "SIXINCHES
41 [FORWARD DOESN'T LIKE SIXINCHES AS I!
NPUT] FD SAFESQUARE
```

SAFESQUARE runs CATCH "ERROR and prints ERROR if an error occurs. You can modify the procedure to print your own error message.

```
TO SAFESQUARE :SIDE
CATCH "ERROR [REPEAT 4 [FD :SIDE RT 90]!
STOP]
PR [DOOPS, A BUG!]
END
```

```
?SAFESQUARE "SIX
DOOPS, A BUG!
```

GO

GO *word* (command)

The GO command transfers control to the instruction following LABEL *word* in the same procedure.

Example:

```
TO COUNTDOWN :N
  LABEL "LOOP
  IF :N < 0 [STOP]
  PRINT :N
  MAKE "N :N - 1
  GO "LOOP
END
```

LABELLABEL *word* (command)

See section "GO."

The LABEL command itself does nothing. However, a GO *word* passes control to the instruction following it. Note that *word* must always be a literal word (that is, it must be preceded by a quotation mark).

REPEATREPEAT *integer list* (command)

REPEAT runs *list* *integer* times. An error occurs if *integer* is negative.

Examples:

REPEAT 4 [FD 100 RT 90] draws a square 100 turtle steps on a side.

REPEAT 3 [FD 100 RT 90] draws three quarters of a square.



RUN

`RUN list` (command or operation)

The `RUN` command runs *list* as if typed in directly. If *list* is an operation, then `RUN` outputs whatever *list* outputs.

Examples:

```
TO CALCULATOR
PR RUN READLIST
PR []
CALCULATOR
END
```

```
?CALCULATOR
2 + 3
5
```

```
17.5 * 3
52.5
```

```
42 = 8 * 7
FALSE
```

```
REMAINDER 12 5
2
```

The `WHILE` procedure runs a list of instructions while a specified condition is true:

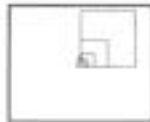
```
TO WHILE :CONDITION :LIST
TEST RUN :CONDITION
IFFALSE [STOP]
RUN :LIST
WHILE :CONDITION :LIST
END
```

```
?RT 10
?WHILE [XCOR < 100] [FD 25 PR POS]
```

The following procedure applies a command to each element of a list in turn:

```
TO MAP :CMD :LIST
IF EMPTY :LIST [STOP]
RUN LIST :CMD WORD "" FIRST :LIST
MAP :CMD BF :LIST
END
```

```
TO SQUARE :SIDE
REPEAT 4 [FD :SIDE RT 90]
END
?MAP "SQUARE [10 20 40 80]
```



```
?MAKE "NEW.ENGLAND [ME NH VT MA RI CT]
?MAP "PRINT :NEW.ENGLAND
ME
NH
VT
MA
RI
CT
```

The following procedure, FOREVER, repeats its input forever (unless it hits an error or is stopped with ⏏ -(ESC)):

```
TO FOREVER :LIST
RUN :LIST
FOREVER :LIST
END
```

The command FOREVER [FD 1 RT 1] tells the turtle to draw a circle.



The command FOREVER [PR RUN READLIST PR []] is equivalent to the CALCULATOR procedure defined above.

The procedure SAFE.SQUARE draws a square and then restores the pen type to whatever it was previously:

```
TO SAFE.SQUARE
MAKE "SAVETYPE PEN
PENDOWN
```



```
SQUARE 100
RUN (SE :SAVETYPE)
END

TO SQUARE :LEN
REPEAT 4 [FD :LEN RT 90]
END

?SHOW PEN
PENUP
?SAFE .SQUARE
?SHOW PEN
PENUP
```

RUN READLIST runs any commands you type in.
PRINT RUN READLIST prints the output from any
expression you typed in.

THROW

THROW *name* (command)

See section "CATCH."

The THROW command is meaningful only within the range of the CATCH command. An error occurs if no corresponding CATCH *name* is found.

THROW **TOPLEVEL returns control to top level. Contrast with STOP.

Debugging Programs

You use the primitives in this section to analyze and debug programs. The primitives are

```
STEP
TRACE
UNSTEP
UNTRACE
```

STEP

STEP *name(list)* (command)

The STEP command takes the procedure indicated by *name(list)* as input and lets you run them line by line. STEP pauses at each line of execution and continues only when you press any key on the keyboard.

Examples:

```
TO TRIANGLE :WORD
IF EMPTY? :WORD [STOP]
PR :WORD
TRIANGLE BL :WORD
END
```

```
?STEP "TRIANGLE
?TRIANGLE "IT
IF EMPTY? :WORD      You press any key.
[STOP]
```

```
PR :WORD              You press any key.
IT
TRIANGLE BL :WORD    You press any key.
IF EMPTY? :WORD      You press any key.
[STOP]
```

```
PR :WORD              You press any key.
1
TRIANGLE BL :WORD    You press any key.
IF EMPTY? :WORD      You press any key.
[STOP]
?
```

TRACE

TRACE *name(list)* (command)

The TRACE command takes the procedures indicated by *name(list)* as input and causes them to print tracing information when executed. It does not interrupt the execution of the procedure, but allows you to see the depth of the procedure

stack during execution. TRACE is useful in understanding recursive procedures or complex programs with many subprocedures.

Examples:

```
?POPS
TO COUNTUP :N
IF :N = 10 [STOP]
COUNTUP :N + 1
PR :N
END
?TRACE "COUNTUP
?COUNTUP 5
COUNTUP 5
COUNTUP 6
COUNTUP 7
COUNTUP 8
COUNTUP 9
COUNTUP 10
COUNTUP stopped
9
COUNTUP stopped
8
COUNTUP stopped
7
COUNTUP stopped
6
COUNTUP stopped
5
COUNTUP stopped
?
```

UNSTEP

UNSTEP *name(list)* (command)

UNSTEP restores the procedure(s) indicated by *name(list)* back to their original states. After you step through a procedure (with STEP), you must use UNSTEP so that it will execute normally again.

Examples:

```
?UNSTEP "TRIANGLE
?TRIANGLE "IT
IT
1
?
```

UNTRACE

UNTRACE *name(list)* (command)

UNTRACE stops the tracing of procedure *name* and causes it to execute normally again.

Examples:

```
?UNTRACE "COUNTUP
?COUNTUP 5
9
8
7
6
5
?
```

Special Control Characters

The special characters in this section interrupt Logo's flow of control, either temporarily or permanently.

OPEN APPLE-ESC

(**⌘**)-ESC (special character)

Pressing (**⌘**)-ESC immediately stops whatever is running, returning Logo to top level, unless in a pause mode.

CONTROL-W

CONTROL-W (special character)

Pressing CONTROL-W interrupts whatever is running. Typing any character resumes normal execution. This special character is particularly useful in giving yourself time to read when Logo is displaying more than one screenful of information.

CONTROL-Z

CONTROL-Z (special character)

Pressing CONTROL-Z interrupts whatever is running, causing a pause. CONTROL-Z is equivalent in effect to PAUSE, but different in its use: you press CONTROL-Z at the keyboard during the running of a procedure, while PAUSE is part of the definition of a procedure.

Modifying Procedures Under Program Control

- 148 COPYDEF
- 148 DEFINE
- 150 DEFINEDP
- 150 PRIMITIVEP
- 151 TEXT

Modifying Procedures Under Program Control

This chapter explains the feature of Logo that allows you to write procedures that define and modify other procedures. The primitives for this feature are

COPYDEF
DEFINE
DEFINEDP
PRIMITIVEP
TEXT

You use the DEFINE and TEXT primitives to define and modify procedures within other procedures. DEFINE changes a list of instructions into a procedure. TEXT works the other way around, changing a procedure into a list. The list can be modified, using the list manipulation techniques described in Chapter 7.

You can use the same list manipulation techniques to create a completely new list. DEFINE then stores it as a procedure in your workspace. Note that if you want to execute this list but don't want to keep it in your workspace, you should use RUN instead of DEFINE.

An explanation of RUN appears in Chapter 10.

PRIMITIVEP and DEFINEDP tell you if a procedure name already exists. They can be useful in writing debugging programs and in avoiding certain error conditions.

COPYDEF creates a copy of a procedure under a new name. You might want to use COPYDEF to create a backup copy of a procedure, because DEFINE can accidentally destroy an existing procedure.

COPYDEF

COPYDEF *name newname* (command)

COPYDEF copies the definition of *name*, making it the definition of *newname* as well.

Examples:

COPYDEF "SQUARE "NEWSQUARE gives NEWSQUARE the same definition as SQUARE.

COPYDEF "FORWARD "F gives F the same definition as FORWARD.

DEFINE

DEFINE *name list* (command)

DEFINE makes *list* the definition of the procedure *name*. The first element of *list* is a list of the inputs to *name*, with no colon (:) before the names.

If *name* has no inputs, this must be the empty list. Each subsequent element is a list consisting of one line of the procedure definition. (This list does not contain END, because END is not part of the procedure definition.)

The second input to DEFINE has the same form as the output from TEXT. DEFINE can redefine an existing procedure.

Examples:

```
DEFINE "SQUARE [[SIDE] [REPEAT 4 [FD :SIDE RT 90]]]
```

defines the same procedure as

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

LEARN is a program that lets you type successive lines defining a procedure that has no inputs. Each time you press **RETURN**, Logo runs the instruction as well as making it part of the procedure definition. By typing ERASE, you can erase the previous line.

```

TO LEARN
MAKE "PRO [[]]
READLINES
PR [DO YOU WANT TO SAVE THIS AS THE DEFINITION OF A PROCEDURE?]
TEST(FIRST FIRST READLIST)= "Y
IFT [TYPE [PROCEDURE NAME?] DEFINE FIRST READLIST :PRO]
END

TO READLINES
MAKE "NEXTLINE READLIST
IF :NEXTLINE = [END] [STOP]
TEST :NEXTLINE = [ERASE]
IFTRUE [CANCEL] .
IFFALSE [RUN :NEXTLINE MAKE "PRO LPUT :NEXTLINE :PRO]
READLINES
END

TO CANCEL
PR SE [I WILL ERASE LINE] LAST :PRO
MAKE "PRO BL :PRO
END

?LEARN
FD 20
RT 36
ERASE
I WILL ERASE LINE RT 36
RT 72
END
DO YOU WANT TO SAVE THIS AS THE DEFINITION OF A PROCEDURE?
YES
PROCEDURE NAME?LEG

```

```
?PD "LEG
TO LEG
FD 20
RT 72
END
```



DEFINEDP

DEFINEDP *word* (operation)

DEFINEDP outputs TRUE if *word* is the name of a user-defined procedure, FALSE otherwise.

PRIMITIVEP

PRIMITIVEP *name* (operation)

PRIMITIVEP outputs TRUE if *name* is the name of a primitive, FALSE otherwise.

Examples:

Operation	Output
PRIMITIVEP "FORWARD	TRUE
PRIMITIVEP "SQUARE	FALSE

TEXT

TEXT *name* (operation)

The TEXT primitive outputs the definition of *name* as a list of lists, suitable for input to DEFINE.

Example:

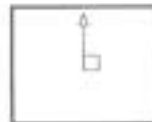
```
?SHOW TEXT "POLY
[[SIDE ANGLE] [FD :SIDE RT :ANGLE] [POL!
Y :SIDE :ANGLE]]
```

The first element of the output is a list of the names of the procedure's inputs. The rest of the elements are lists; each one is a line in the procedure definition. (If the procedure name is undefined, TEXT outputs the empty list.) The previous example corresponds to:

```
?PD "POLY
TO POLY :SIDE :ANGLE
FD :SIDE RT :ANGLE
POLY :SIDE :ANGLE
END
```

You can use TEXT in conjunction with DEFINE to create procedures that modify other procedures. Here is a simple example:

```
?PD "SQUARE
TO SQUARE
REPEAT 4 [FD 30 RT 90]
END
?DEFINE "SQUARE.WITH.TAIL LPUT [FD 100]!
TEXT "SQUARE
?PD "SQUARE.WITH.TAIL
TO SQUARE.WITH.TAIL
REPEAT 4 [FD 30 RT 90]
FD 100
END
```



The primitive STEP is described in Chapter 10.

Complex Example:

The procedure \$STEP in this example modifies the definition of a procedure to make it run one line at a time. The procedure \$STEP is similar to the primitive STEP. The example is included to show you how to modify a procedure definition.

After each line is run, Logo waits for you to press before it proceeds. \$UNSTEP restores the original procedure definition.

The Program:

```
TO $STEP :PRO
COPYDEF :PRO WORD ". :PRO
MAKE "OLDDEF TEXT :PRO
MAKE "NEWDEF (LIST FIRST :OLDDEF)
MAKE "NEWDEF LPUT (LIST "PRINT (LIST "E!
ENTERING :PRO)) :NEWDEF
SHOWINPUTS FIRST :OLDDEF
SHOWLINES BF :OLDDEF
DEFINE :PRO :NEWDEF
END

TO IGNORE :INPUTT
END

TO STEPPER
TYPE "
IGNORE READLIST
END
```

```
TO SHOWLINES :INSTRUCTIONS
IF EMPTY :INSTRUCTIONS [STOP]
MAKE "NEWDEF LPUT (LIST "TYPE FIRST :IN!
STRUCTIONS) :NEWDEF
MAKE "NEWDEF LPUT [STEPPER] :NEWDEF
MAKE "NEWDEF LPUT FIRST :INSTRUCTIONS :!
NEWDEF
SHOWLINES BF :INSTRUCTIONS
END
```

```
TO SHOWINPUTS :ARGLIST
IF EMPTY :ARGLIST [STOP]
MAKE "NEWDEF LPUT (LIST "PRINT "SENTENC!
E (LIST (FIRST :ARGLIST) "IS) (WORD "!!
FIRST :ARGLIST)) :NEWDEF
SHOWINPUTS BF :ARGLIST
END
```

```
TO $UNSTEP :PRO
COPYDEF WORD ". :PRO :PRO
ERASE WORD ". :PRO
END
```


Using the Program:

```
TO TRIANGLE :WORD  
IF EMPTY? :WORD [STOP]  
PR :WORD  
TRIANGLE BL :WORD  
END
```

```
?#STEP "TRIANGLE  
?TRIANGLE "IT  
ENTERING TRIANGLE  
WORD IS IT  
IF EMPTY? :WORD [STOP]  
PR :WORD  
IT  
TRIANGLE BL :WORD  
ENTERING TRIANGLE  
WORD IS I  
IF EMPTY? :WORD [STOP]  
PR :WORD  
I  
TRIANGLE BL :WORD  
ENTERING TRIANGLE  
WORD IS  
IF EMPTY? :WORD [STOP]  
?
```

You press

You press

You press

You press

You press

You press

Logical Operations

- 158 AND
- 159 NOT
- 160 OR

Logical Operations



Predicates are operations that output only TRUE or FALSE. Most of their names end in *P*.

This chapter describes the logical operations AND, NOT, and OR. A **logical operation** is a predicate whose input must be either TRUE or FALSE.

The inputs to logical operations are usually other predicates. Predicates are found throughout the other chapters of this manual:

Predicate	Chapter
BEFOREP	7
BUTTONP	13
DEFINEDP	11
DOTP	5
EMPTYP	7
EQUALP	7
FILEP	15
KEYP	13
LISTP	7
MEMBERP	7
NAMEP	8
NUMBERP	7
PRIMITIVEP	11
SHOWNP	5
WORDP	7
<	9
=	9
>	9

AND

AND *predicate1 predicate2* (operation)
(AND *predicate1 predicate2 predicate3 ...*)

AND outputs TRUE if all its inputs are true, FALSE otherwise.

Examples:

Operation	Output
AND "TRUE "TRUE	TRUE
AND "TRUE "FALSE	FALSE
AND "FALSE "FALSE	FALSE
(AND "TRUE "TRUE "FALSE "TRUE)	FALSE
AND 5 7	Error
AND PENCOLOR=1 BACKGROUND=0 (when you start up Logo)	FALSE

The following procedure, DECIMALP, tells whether its input is a decimal number:

```
TO DECIMALP :OBJ
  OUTPUT AND NUMBERP :OBJ MEMBERP ". :OBJ
END

?PR DECIMALP 17
FALSE
?PR DECIMALP 17.
TRUE
?PR DECIMALP "STOP.
FALSE
```

The following procedure tells you whether the temperature is comfortable (between 50 and 90 degrees F):

```
TO COMFORT
  IF AND :TEMPERATURE > 50 :TEMPERATURE <!  
    90 [PR "DELIGHTFUL] [PR "UNPLEASANT]
END
```

```
?MAKE "TEMPERATURE 68
?COMFORT
DELIGHTFUL
```

NOT

NOT *predicate* (operation)

NOT outputs TRUE if *predicate* is FALSE; if *predicate* is TRUE, NOT outputs FALSE.

Examples:

Operation	Output
NOT EQUALP "A "B	TRUE
NOT EQUALP "A "A	FALSE
NOT "A = FIRST "DOG	TRUE
NOT "A	Error

If WORDP were not a primitive, it could be defined as follows:

```
TO WORDP :OBJ
  OUTPUT NOT LISTP :OBJ
END
```

The following procedure tells whether its input is a word that isn't a number:

```
TO REALWORDP :OBJ
  OUTPUT AND WORDP :OBJ NOT NUMBERP :OBJ
END
```

```
?PR REALWORDP HEADING
FALSE
?PR REALWORDP POS
FALSE
?PR REALWORDP "KANGAROO
TRUE
?PR REALWORDP PEN
TRUE
```

OR

OR *predicate1 predicate2* (operation)
(OR *predicate1 predicate2 predicate3 ...*)

OR outputs FALSE if all its inputs are false; otherwise it outputs TRUE.

Examples:

Operation	Output
OR "TRUE" "TRUE"	TRUE
OR "TRUE" "FALSE"	TRUE
OR "FALSE" "FALSE"	FALSE
(OR "FALSE" "FALSE" "FALSE" "TRUE")	TRUE
OR 5 7	Error

The procedure MOUNTAINS draws mountains:

```
TO MOUNTAINS
  SETPC 5
  RT 45
  FD 5
  SUBMOUNTAIN
END

TO SUBMOUNTAIN
  FD 5 + RANDOM 10
  IF OR YCOR > 50 YCOR < 0 [SETHEADING 1!
  80-HEADING]
  SUBMOUNTAIN
END
```



The Outside World

- 163** Using Paddles
- 163** **BUTTONP**
- 164** **PADDLE**
- 164** Making Logo Read Information
- 164** **KEYP**
- 165** **READCHAR**
- 166** **READCHARS**
- 167** **READLIST**
- 167** **READWORD**
- 168** Making Logo Write Information
- 169** **PRINT**
- 170** **SHOW**
- 170** **TYPE**
- 171** Making Sounds With **TOOT**

The Outside World



This chapter describes primitives for communicating with various devices through the computer. The devices include the keyboard, the television set, and the game paddles. The primitives are divided into four groups:

- primitives for using paddles
- primitives for making Logo read information
- primitives for making Logo write information
- a primitive for making sounds.

Using Paddles

This section describes the `BUTTONP` and `PADDLE` primitives, which communicate information from the paddle, or hand control.

`BUTTONP`

`BUTTONP` *paddlenumber* (operation)

`BUTTONP` outputs `TRUE` if the button on the specified paddle is down and `FALSE` if the button is up. The *paddlenumber* must be 0, 1, 2 OR 3. (◁) is button 0 and (▶) is button 1.

PADDLE

PADDLE *paddlenumber* (operation)

PADDLE outputs a number between 0 and 255, representing the rotation of the dial on the specified paddle.

Example:

```
TO PDRAW
  RIGHT (PADDLE 0) / 25.6
  FORWARD (PADDLE 1) / 25.6
PDRAW
END
```

Making Logo Read Information

This section presents the primitives that you use to make Logo read information from a device or a file. Normally, this device is the keyboard. The primitives are

KEYP
READCHAR
READCHARS
READLIST
READWORD

The operations READCHAR, READCHARS, READLIST, and READWORD let Logo read text that has been typed into the keyboard. KEYP is a keyboard predicate mainly useful in game situations.

READCHAR, READCHARS, READLIST, and READWORD are also used in connection with the file-handling system described in Chapters 15 and 16.

KEYP

KEYP (operation)

KEYP outputs TRUE if there is at least one character waiting to be read—that is, one that has been typed on the keyboard and not yet picked up by READCHAR or READLIST. KEYP outputs FALSE if there are no such characters.

Example:

```

TO STEER
FD 2
IF KEYP [TURN READCHAR]
STEER
END

TO TURN :DIR
IF :DIR = "R [RT 10]
IF :DIR = "L [LT 10]
END

```

READCHAR

READCHAR (RC) (operation)

READCHAR outputs the first character typed at the keyboard or read from the current file. If you are reading from the keyboard and no character is waiting to be read, READCHAR waits until you type something.

See also section "KEYP."

READCHAR does not output a character if you are reading from a file and the end-of-file position is reached. In this case, READCHAR outputs an empty list. Note that READCHAR from the keyboard does not echo what you type on the screen.

If you are reading from the keyboard, you can set the high bit of the character being read by holding down either Apple key as you type the character. Setting the high bit adds 128 to the character.

The following procedure, XYZZY, lets you run certain commands with a single keystroke: (F) does FORWARD 5, and (R) does RIGHT 10. (You can add to the list.) You need not press (RETURN) after the keystroke.

```

TO XYZZY
INTERPRET READCHAR
XYZZY
END

TO INTERPRET :CHAR
IF :CHAR = "F [FD 5]
IF :CHAR = "R [RT 10]
IF :CHAR = "S [THROW "TOPLEVEL]
END

```

READCHARS

READCHARS *integer* (RCS) (operation)

The READCHARS operation outputs the first *integer* number of characters typed at the keyboard or read from the current file. If you are reading from the keyboard and no characters are waiting to be read, READCHARS waits for you to type something.

If you are reading from a file and the end-of-file position is reached before *integer* characters are read, READCHARS outputs the characters read up to that point. If the end-of-file position was reached before READCHARS was called, READCHARS outputs an empty list.

Note that READCHARS from the keyboard does not echo what you type on the screen.

Remember that a carriage return is read as a character.

If you are reading from the keyboard, you can set the high bit of the character being read by holding down either Apple key as you type the character. Setting the high bit adds 128 to the character.

Example:

```
?PRINT READCHARS 4
```

Type the following letters:

ABC (Don't press `RETURN`.)

Nothing happens. Now type

D

The following appears on the screen:

ABCD

READLIST

READLIST (RL) (operation)

The READLIST operation reads a line of information from the current file and outputs the information in the form of a list. Normally, the source is the keyboard, where you type in information followed by a carriage return. This information is echoed on the screen. The command SETREAD allows you to read from other files.

If you are reading from a file where the end-of-file position has already been reached, READLIST outputs the empty word.

Examples:

```
?PRINT COUNT READLIST
I HOPE THIS REALLY WORKS
5

TO GET.USER
PRINT [WHAT IS YOUR NAME?]
MAKE "USER READLIST
PRINT SE [WELCOME TO LOGO,] :USER
END

?GET.USER
WHAT IS YOUR NAME?
EFFIE
WELCOME TO LOGO, EFFIE
?GET.USER
WHAT IS YOUR NAME?
EFFIE MANIATIS
WELCOME TO LOGO, EFFIE MANIATIS
```

READWORD

READWORD (RW) (operation)

READWORD reads a line of information from the current file and outputs it as a word. Normally, the source is the keyboard, and READWORD waits for you to type and press `(RETURN)`. What you type is echoed on the screen. If you press `(RETURN)` before typing a word, READWORD outputs an empty word.

See sections "READLIST, READCHAR, READCHARS," and "SETREAD."

If you use READWORD from a file, READWORD reads characters until it reaches a carriage return, and outputs those characters as a word. The next character to be read is the one after the carriage return. When the end-of-file position is reached, READWORD outputs an empty list.

Examples:

```
?SHOW READWORD
LONDON ONTARIO
LONDON ONTARIO

?PRINT COUNT READWORD
THERE IS SOME VALUE IN COUNTING WORDS
37
```

The following procedure asks your age and then prints how old you will be next year.

```
TO AGE
PRINT [HOW OLD ARE YOU?]
PRINT MESSAGE READWORD
END

TO MESSAGE :AGE
OP SE [NEXT YEAR YOU WILL BE] :AGE + 1
END

?AGE
HOW OLD ARE YOU?
11
NEXT YEAR YOU WILL BE 12

?AGE
HOW OLD ARE YOU?
35
NEXT YEAR YOU WILL BE 36
```

Making Logo Write Information

This section presents the primitives that you use to make Logo write information to a destination such as the screen. The primitives are

```
PRINT
SHOW
TYPE
```

PRINT

PRINT *object* (PR) (command)
(PRINT *object1 object2 ...*)

The PRINT command prints its inputs followed by a carriage return on the screen, unless the destination has been changed by SETWRITE. The outermost brackets of lists are not printed. Compare with TYPE and SHOW.

Examples:

```
?PRINT "A
A
?PRINT "A PRINT [A B C]
A
A B C
?(PRINT "A [A B C])
A A B C
?PRINT []
?
TO REPRINT :MESSAGE :HOWMANY
IF :HOWMANY < 1 [STOP]
PR :MESSAGE
PR
REPRINT :MESSAGE :HOWMANY-1
END
?REPRINT [TODAY IS FRIDAY!] 4
TODAY IS FRIDAY!
TODAY IS FRIDAY!
TODAY IS FRIDAY!
TODAY IS FRIDAY!
?
```

SHOW

SHOW *object* (command)

The SHOW command prints *object* followed by a carriage return on the screen, unless the destination has been changed by SETWRITE. If *object* is a list, Logo leaves brackets around it. Compare with TYPE and PRINT.

Examples:

```
?SHOW "A
A
?SHOW "A SHOW [A B C]
A
[A B C]
```

TYPE

TYPE *object* (command)
(TYPE *object1 object2 ...*)

The TYPE command prints its inputs without a carriage return on the screen, unless the destination has been changed by SETWRITE. The outermost brackets of lists are not printed. Compare with PRINT and SHOW.

Examples:

```
?TYPE "A
A?TYPE "A TYPE [A B C]
AA B C?(TYPE "A [A B C])
AA B C?
```

The procedure PROMPT types a message followed by a space:

```
TO PROMPT :MESSAGE
TYPE :MESSAGE
TYPE CHAR 32
END

TO MOVE
PROMPT [HOW MANY STEPS SHOULD I TAKE?]
FD FIRST READLIST
MOVE
END

?MOVE
HOW MANY STEPS SHOULD I TAKE? 50
HOW MANY STEPS SHOULD I TAKE? 37
HOW MANY STEPS SHOULD I TAKE? 2
HOW MANY STEPS SHOULD I TAKE? 108
```

Making Sounds With TOOT

TOOT *frequency duration* (command)

TOOT generates a tone via a loudspeaker. The frequency is specified in Hertz (cycles per second). The tuning note A is 440. The duration can range from 0 to 65,535. It is measured in units of 1/60 of a second.

Example:

```
TO SIREN :FREQ
IF :FREQ > 440 [STOP]
TOOT :FREQ 3
SIREN :FREQ + 5
TOOT :FREQ 3
END
```

SIREN produces a siren sound of ascending and descending notes.

Table 13-1 provides the frequencies of approximately seven octaves of notes.

Table 13-1. Note Frequencies for TOOT

Note	Frequency, by Octave							
B	62	123	247	494	988	1973	3946	
A#	58	117	233	466	932	1864	3743	
A	55	110	220	440	881	1761	3510	
G#	52	104	208	415	830	1663	3327	
G	49	98	196	392	784	1566	3142	
F#	46	92	185	370	740	1480	2959	
F	44	87	175	349	698	1398	2797	
E	41	82	165	330	659	1319	2637	
D#	39	78	156	311	622	1244	2495	4990
D	37	73	147	294	587	1176	2346	4713
C#	35	69	139	277	554	1109	2213	4426
C	33	65	131	262	523	1047	2095	4172

Middle C

Managing Your Workspace

- 176** Sizing Up Your Workspace
- 176** NODES
- 177** RECYCLE
- 177** Printing From the Workspace
- 177** PO
- 178** POALL
- 178** PON
- 179** PONS
- 179** POPS
- 180** POT
- 180** POTS
- 180** Erasing From the Workspace
- 181** ERALL
- 181** ERASE
- 181** ERN
- 181** ERNS
- 182** ERPS
- 182** Cleaning and Organizing the Workspace
- 182** BURY
- 183** BURYALL
- 183** BURYNAME
- 184** UNBURY
- 184** UNBURYALL
- 185** UNBURYNAME

Managing Your Workspace



This chapter tells you how to manage the workspace in your Apple computer. **Workspace** is an area of the Apple's memory where Logo keeps your procedures, variables, and properties that it knows about right now. It does not include primitives.

Logo provides primitives to let you

- examine the size of your workspace and free up additional space there
- see what you have in your workspace
- selectively erase variables and procedures from your workspace
- clean up and organize your workspace.

The workspace is a temporary storage space. Your procedures, variables, and properties will be erased when you turn off the power of the computer. If you want to keep them for future use, you must store them on a disk in the form of files.

Procedures and names in the workspace can be buried, making them invisible to global commands such as ERALL, ERPS, POALL, POPS, POTS, and SAVE. A buried procedure or name still exists in the workspace. Therefore, you can run, edit, print out, or erase a buried procedure, as long as you specify its name.

The bury facility is useful for organizing your workspace. You can use it to selectively save procedures in different files. You can also use the bury facility to make procedures appear as primitives. For instance, you may want some of the procedures in Appendix B, Useful Tools, to be buried in the workspace.

See Chapters 15 and 16 for information on files.

Here is an example of how to organize your workspace.

```
?POTS
TO SENGEN :NOUNS :VERBS
TO PICK :OBJECT
TO SUPERSENGEN
TO POLY :SIDE :ANGLE
TO POLYSPI :SIDE :ANGLE :INC
TO SQ :SIDE
TO TRIANGULATE :WORD

?PONS
MAKE "NOUNS [COMPUTERS HOUSES BEDS CHAIRS TV STEREO]
MAKE "VERBS [PLAY COMPUTE LIE SIT [FALL! DOWN]]
MAKE "START HEADING
```

You can group the procedures and variables by giving them names.

```
?MAKE "LANGUAGE [SENGEN PICK SUPERSENGEN]
?MAKE "LANGNAMES [NOUNS VERBS]
```

Now use the bury feature to save them in a file.

```
?BURYALL
?UNBURY :LANGUAGE
?UNBURYNAME :LANGNAMES
?SAVE "LANGUAGE
```

Sizing Up Your Workspace

You use the primitives in this section to figure out how much free space you have in your workspace (NODES) and to free up as much workspace as possible (RECYCLE).

NODES

NODES (operation)

NODES outputs the number of free nodes. This gives you an idea of how much space you have in your workspace for procedures, variables, properties, and the running of procedures. NODES is most useful if run immediately after RECYCLE.

See Appendix D, Memory Space.

See section "NODES" and also Appendix D, Memory Space.

RECYCLE

RECYCLE (command)

The RECYCLE command frees up as many nodes as possible, performing what is called a *garbage collection*. When you don't use RECYCLE, garbage collections happen automatically whenever necessary, but each one takes at least one second. Running RECYCLE before a time-dependent activity prevents the automatic garbage collector from slowing things down at an awkward time.

Printing From the Workspace

This section tells you how to print the contents of your workspace. The primitives to use for this are

PO
POALL
PON
PONS
POPS
POT
POTS

PO

PO *name(list)* (command)

The PO (for print out) command prints the definition(s) of the named procedure(s).

Examples:

```
?PO "LENGTH
TO LENGTH :OBJ
IF EMPTYP :OBJ [OP 0] [OP 1 + LENGTH BF!
:OBJ]
END
?PO [LENGTH GREET]
TO LENGTH :OBJ
IF EMPTYP :OBJ [OP 0] [OP 1 + LENGTH BF!
:OBJ]
END
```

```
TO GREET
PR [GOOD MORNING. HOW ARE YOU TODAY?]
END
```

POALL

POALL (command)

The POALL (for print out all) command prints the definition of every procedure and the value of every variable in the workspace.

Example:

```
?POALL
TO POLY :SIDE :ANGLE
FD :SIDE
RT :ANGLE
POLY :SIDE :ANGLE
END

TO LENGTH :OBJ
IF EMPTY? :OBJ [OP 0] [OP 1 + LENGTH BF!
:OBJ]
END

TO GREET
PR [GOOD MORNING. HOW ARE YOU TODAY?]
END

TO SPI :SIDE :ANGLE :INC
FD :SIDE
RT :ANGLE
SPI :SIDE * :INC :ANGLE :INC
END

MAKE "ANIMAL "AARDVARK
MAKE "LENGTH 3.98
MAKE "MYNAME "STEVE
```

PON

PON *name(list)* (command)

PON (for print out name) prints the name and value of the named variable(s).

See section "BURY" for exceptions.

Examples:

```
?PON "LENGTH
MAKE "LENGTH 3.98
?PON :LANGNAMES
MAKE "NOUNS (COMPUTERS HOUSES BEDS CHAI!
RS TV STEREO)
MAKE "VERBS (PLAY COMPUTE LIE SIT (FALL!
DOWN))
```

PONS

PONS (command)

PONS (for print out names) prints the name and value of every variable in the workspace.

Example:

```
?PONS
MAKE "F 3
MAKE "LIST (A B C)
```

POPS

POPS (command)

POPS (for print out procedures) prints the definition of every procedure in the workspace.

Example:

```
?POPS
TO POLY :SIDE :ANGLE
FD :SIDE
RT :ANGLE
POLY :SIDE :ANGLE
END

TO SPI :SIDE :ANGLE :INC
FD :SIDE
RT :ANGLE
SPI :SIDE + :INC :ANGLE :INC
END
```

See section "BURY" for exceptions.

POT

POT *name(list)* (command)

The POT (for print out title) command prints the title line of the named procedure(s) in the workspace.

Example:

You may want to group some procedures by giving them a variable name:

```
?MAKE "LANGUAGE [SENGEN PICK]
```

To find out the titles in the LANGUAGE variable, use POT.

```
?POT :LANGUAGE  
TO SENGEN :NOUNS :VERBS  
TO PICK :OBJECT
```

POTS

POTS (command)

POTS (for print out titles) prints the title line of every procedure in the workspace.

Examples:

```
?POTS  
TO PDLY :SIDE :ANGLE  
TO LENGTH :OBJ  
TO GREET  
TO SPI :SIDE :ANGLE :INC
```

See section "BURY" for exceptions.

Erasing From the Workspace

This section tells you how to erase information from the workspace. The primitives for doing this are

```
ERALL  
ERASE  
ERN  
ERNS  
ERPS
```


See section "BURY" for exceptions.

ERALL

ERALL (command)

ERALL erases all procedures, variables, and properties from the workspace.

ERASE

ERASE *name(list)* (ER) (command)

The ERASE command erases the named procedure(s) from the workspace.

Examples:

ERASE *TRIANGLE erases the TRIANGLE procedure.
ERASE [TRIANGLE SQUARE] erases the TRIANGLE and SQUARE procedures.

ERN

ERN *name(list)* (command)

The ERN (for erase name) command erases the named variable(s) from the workspace.

Examples:

See the example using NOUNS and VERBS at the beginning of this chapter.

ERN *LENGTH erases the LENGTH variable.
ERN :LANGNAMES erases the NOUNS and VERBS variables.

ERNS

ERNS (command)

See section "BURY" for exceptions.

ERNS (for erase names) erases all variables from the workspace.

ERPS

ERPS (command)

See section "BURY" for exceptions.

The ERPS (for erase procedures) command erases all procedures from the workspace.

Cleaning and Organizing the Workspace

This section discusses the primitives that you use to manage your workspace effectively. The primitives for doing this are

BURY
BURYALL
BURYNAME
UNBURY
UNBURYALL
UNBURYNAME

BURY

BURY *name(list)* (command)

The BURY command buries the procedure(s) in its input. Certain commands (ERALL, ERPS, POALL, POPS, POTS, and SAVE) act on everything in the workspace except procedures and names that are buried.

Example:

SAVE "GOODSTUFF" saves the whole workspace in the file GOODSTUFF except procedures and names that are buried.

See section "UNBURYALL" for unburying everything in the workspace.

BURYALL

BURYALL (command)

The BURYALL command buries all the procedures and variable names in the workspace.

Example:

```
?POTS
TO POLY :SIDE :ANGLE
TO LENGTH :OBJ
TO GREET
TO SPI :SIDE :ANGLE :INC
?PONS
MAKE "ANIMAL "AARDVARK
MAKE "LENGTH 3.98
MAKE "MYNAME "STEVE
?BURYALL
?POTS
?PONS
?
```

Once BURYALL is run, there are no procedure titles or names visible.

BURYNAME

BURYNAME *name(list)* (command)

BURYNAME buries the variable name(s) in its input.

Example:

```
?PONS
MAKE "ANIMAL "AARDVARK
MAKE "LENGTH 3.98
MAKE "MYNAME "STEVE
?BURYNAME "MYNAME
?PONS
MAKE "ANIMAL "AARDVARK
MAKE #LENGTH 3.98
```

See section "UNBURYNAME" to unbury variables.

See section "BURY."

UNBURY

UNBURY *name(list)* (command)

The UNBURY command unburies the named procedure(s).

UNBURYALL

UNBURYALL (command)

UNBURYALL unburies all procedures and variable names that are currently buried in the workspace.

Example:

```
?PQTS  
?PQNS
```

There are no procedures or variable names printed.

```
?UNBURYALL  
?PQTS  
TO POLY :SIDE :ANGLE  
TO LENGTH :OBJ  
TO GREET  
TO SP1 :SIDE :ANGLE :INC  
?PQNS  
MAKE "ANIMAL "AARDVARK  
MAKE "LENGTH 3.98  
MAKE "MYNAME "STEVE
```

Once UNBURYALL is run, the procedures and variable names are visible.

UNBURYNAME

UNBURYNAME *name(list)* (command)

UNBURYNAME unburies the variable name(s) in its input.

Example:

```
?PONS  
?
```

There are no variables visible.

```
?UNBURYNAME [LENGTH NOUNS]  
?PONS  
MAKE "LENGTH 3.98  
MAKE "NOUNS [COMPUTERS HOUSES BEDS CHAIRS TV STEREO]
```



General File Management

- 189** Logo's File System: Some General Information
- 189** What Is a File?
- 190** Disk Formatting and Volume Names
- 190** Disk Organization
- 192** Accessing Files
- 194** General File System Primitives
- 194** CATALOG
- 195** CREATEDIR
- 196** EDITFILE
- 196** ERASEFILE
- 196** FILEP
- 197** LOADHELP
- 197** ONLINE
- 198** POFILE
- 198** PREFIX
- 199** RENAME
- 199** SETPREFIX

General File Management



Chapter 16 gives you the primitives for working with the specific types of files.

Logo uses four types of files in its file system: program files, picture files, dribble files, and data files. This chapter presents general information about Logo's file system, as well as the primitives that you use to manage all types of Logo files.

This chapter is divided into two main sections, which provide

- general information about the file system, including some terminology and rules you need to use it
- the primitives that deal with general file management.

Logo's File System: Some General Information

This section gives you the basics of Logo's file system and introduces you to the example that is used throughout the chapter to show the file-handling features.

What Is a File?

A **file** is a collection of information. Generally, this information is organized and stored on a disk. Logo creates different types of files on disk according to the nature of the information that is stored.

There are four types of files you work with in Logo: program files, picture files, dribble files, and data files. A **program file** is a file of Logo procedures that you want to keep and use again later. A **picture file** is a file containing a picture that you've created. A **dribble file** is a record of the text that is printed on the screen. A **data file** contains information that you want to keep track of, such as the addresses and telephone numbers of your friends.

Although the nature of the files may be different, they are all organized on the disk in a similar manner. The next section explains how files are organized by ProDOS—the operating system under which Logo runs.

Disk Formatting and Volume Names

Every disk must be formatted for use. The formatting process prepares a disk in three ways:

- It divides the disk surface into uniform areas, called **blocks**, where ProDOS stores information.
- It gives the disk a volume name that you select.
- It writes a volume directory and other information that ProDOS needs to locate files.

You must format all disks before using them to store any information.

A **volume** is a formatted disk on which you keep files of information. Every volume has a name. Here are some examples of volume names:

Volume Name	Might Be Used for
/LOGO/	The disk you use to start up Logo
/MYDISK/	A disk containing your work
/LOGO.SAMPLES/	The fictitious disk used for the examples in this chapter

You use volume names to tell Logo where to find the file you want to get or where to put the file you want to save.

Disk Organization

Files can be saved on disk in various ways. To get a listing of which files are on your disk, you use the CATALOG command. This listing of the names and sizes of files is called a **directory**. Whenever you try to open a file, ProDOS checks the volume directory to find the file on the disk.

The disk volume MYDISK includes the following directory.

```
?CATALOG
/MYDISK/
  PICTURES1      10
  POLYS          15
  SPIRALS        10
  GAME           10
  PHONELIST      20
  ADDRESS        10
Blocks Free: 255 Blocks Used: 75
?
```

This directory contains files saved at the root of the directory. After you have accumulated a large number of files, this way of storing them on your disk might become cumbersome.

ProDOS lets you classify your files on disk according to your own needs, using a subdirectory structure. LOGO.SAMPLES is organized using a system of subdirectories. **Subdirectories** are files that contain lists of other files.

Subdirectories are very useful in keeping your information organized. For example, on the disk /LOGO.SAMPLES/ there are three subdirectories. One (PROGRAMS) holds Logo programs; the second (PICTURES) holds graphics pictures; the third (DATA) contains data for your programs.

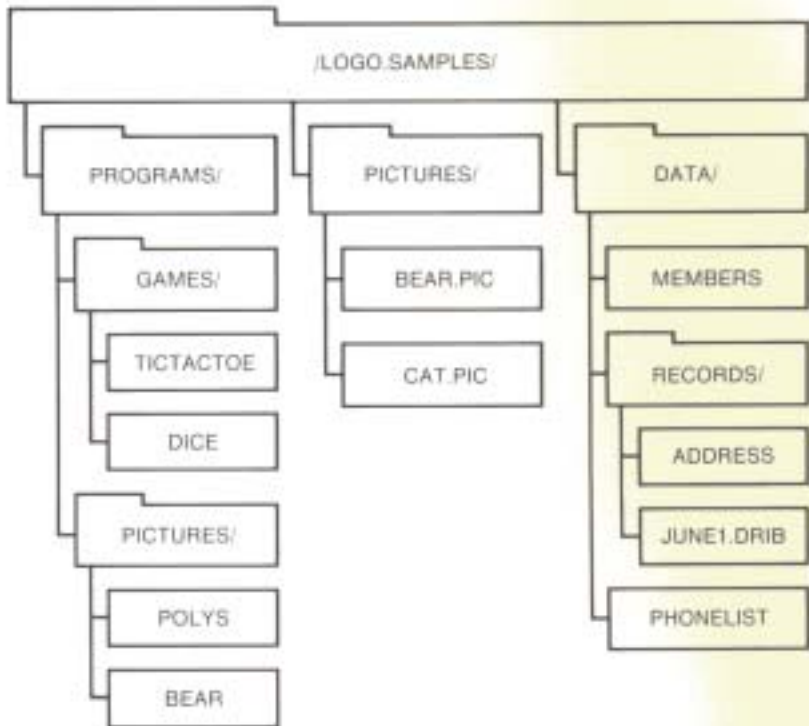
The disk volume /LOGO.SAMPLES/ has the following volume directory:

```
?CATALOG
/LOGO.SAMPLES/
  PROGRAMS/
  PICTURES/
  DATA/
Blocks Free: 138 Blocks Used: 142
?
```

Notice that the names of these files listed all end with a slash (/). The slash indicates that the files are subdirectories.

Figure 15-1 shows you a diagram of the directory structure of the fictitious disk /LOGO.SAMPLES/. The directory structure shown in this figure is used in most of the examples throughout this chapter and in Chapter 16.

Figure 15-1. Files and Subdirectories on a Volume



Notice that the subdirectories `/LOGO.SAMPLES/PROGRAMS/` and `/LOGO.SAMPLES/DATA/` contain additional subdirectories that further organize what is stored.

CREATEDIR and ERASEFILE are explained in this chapter.

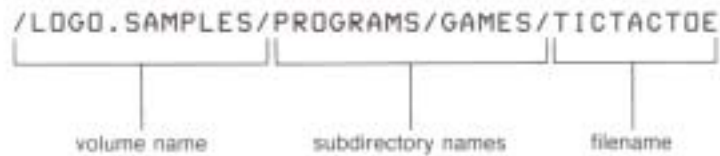
To create a subdirectory, use the CREATEDIR command. To erase a subdirectory, use the ERASEFILE (ERF) command.

Accessing Files

ProDOS checks through the various directory levels you've set up whenever it needs to access a file on the disk.

For example, to access the file TICTACTOE on the disk `/LOGO.SAMPLES/` you trace a *path* from `/LOGO.SAMPLES/` to `PROGRAMS/` to `GAMES/` and finally to TICTACTOE.

The file's full name or **pathname** is



Note: A filename can be from 1 to 15 characters long and must begin with a letter. The name can contain any letter from A through Z, any digit 0 through 9, and periods (.).

A **prefix** is a pathname of a directory or subdirectory, which is automatically placed in front of a filename that does not begin with a slash (/).

There are two ways to gain access to the file TICTACTOE:

- Use the full pathname. For example:

```
LOAD "/LOGO.SAMPLES/PROGRAMS/GAMES/TICTACTOE
```

- Set the prefix to the subdirectory containing TICTACTOE and then use only the filename. For example:

```
SETPREFIX "/LOGO.SAMPLES/PROGRAMS/GAMES  
LOAD "TICTACTOE
```

If you intend to use several files in the same subdirectory, the second method is easier.

CATALOG uses the prefix each time it lists a directory.

```
?CATALOG  
/LOGO.SAMPLES/PROGRAMS/GAMES/ (This is the prefix.)  
TICTACTOE          12  
DICE                5  
Blocks Free: 138  Blocks Used: 142  
?
```


General File System Primitives

The rest of this chapter describes the primitives that perform general file management tasks, such as creating a subdirectory, checking which volumes are on line, and so on. These primitives apply to all files, regardless of the information stored in the files. The primitives are

CATALOG	ONLINE
CREATEDIR	POFILE
EDITFILE	PREFIX
ERASEFILE	RENAME
FILEP	SETPREFIX
LOADHELP	

CATALOG

CATALOG (command)

CATALOG prints the names of the files in the current directory as well as the number of blocks used by each. The current directory is the directory pointed to by the current ProDOS prefix.

Example:

```
?CATALOG
/LDGD.SAMPLES/      (current ProDOS prefix)
PROGRAMS/           (subdirectory)
PICTURES/           (subdirectory)
DATA/               (subdirectory)
Blocks Free: 138  Blocks Used: 142
?
?SETPREFIX "PROGRAMS (sets prefix)
?CATALOG
/LDGD.SAMPLES/PROGR- (current ProDOS prefix)
AMS/
GAMES/              (subdirectory)
PICTURES/           (subdirectory)
Blocks Free: 138  Blocks Used: 142
?
```

To see what is in the subdirectory PICTURES:

```
?SETPREFIX "PICTURES
?CATALOG
/LOGO.SAMPLES/PROGRAMS/PICTURES/      (current ProDOS prefix)
  POLYS          2                      (filename)
  BEAR           3                      (filename)
Blocks Free: 138 Blocks Used: 142
?SETPREFIX "/LOGO.SAMPLES/DATA/RECORDS
?CATALOG
/LOGO.SAMPLES/DATA/RECORDS/
  ADDRESS        10                     (filename)
  PHONELIST      15                     (filename)
Blocks Free: 138 Blocks Used: 142
?
```

CREATEDIR

CREATEDIR *pathname* (command)

CREATEDIR creates the subdirectory indicated by *pathname*. The last file name in *pathname* is the subdirectory to be created, and preceding names indicate where it should be placed.

Examples:

```
?CREATEDIR "/LOGO.SAMPLES/PROGRAMS/TOOLS
```

creates the subdirectory TOOLS in the subdirectory PROGRAMS. If the ProDOS prefix is already set to /LOGO.SAMPLES/PROGRAMS/, then

```
?CREATEDIR "TOOLS
```

has the same effect.

For details on using the Editor, see Chapter 4.

EDITFILE

EDITFILE *pathname* (command)

EDITFILE loads the file indicated by *pathname* into the edit buffer and saves the edited contents under the same filename. The old contents will be lost.

You can use EDITFILE on any file, whether it exists or not. If it does not exist, Logo creates it when you save the contents of the edit buffer.

The edit buffer cannot hold more than 6144 characters. If the file you try to edit contains more than this, Logo displays an error message and does not let you edit the file.

ERASEFILE

ERASEFILE *pathname* (ERF) (command)

The ERASEFILE command erases the file indicated by *pathname* from the disk. If the input is a filename alone, the file must be located in the current directory. An error occurs if no file exists.

Example:

```
?ERASEFILE **/LOGO.SAMPLES/PROGRAMS/PICTURES  
/BEAR
```

erases the file called BEAR from the subdirectory PICTURES in the subdirectory PROGRAMS.

ERASEFILE will also erase subdirectories, but only if they contain no files. An error occurs if you try to erase a subdirectory with files in it.

FILEP

FILEP *pathname* (operation)

FILEP outputs TRUE if a file indicated by *pathname* exists on the disk; otherwise it outputs FALSE. An error occurs if you try to use FILEP on a device.

Examples:

```
?PRINT FILEP ""/LOGO.SAMPLES/PROGRAMS/HANOI
NOI
FALSE
```

The file called HANOI does not exist.

The REPLACEFILE procedure allows you to replace an old file with something new when saving on disk.

```
TO REPLACEFILE :FILE
IF FILEP :FILE [ERF :FILE]
SAVE :FILE
END
```

LOADHELP

LOADHELP *pathname* (command)

The LOADHELP primitive loads the file indicated by *pathname* into memory where the main help screen is stored. This primitive lets you write Logo programs that provide help to the user.

The help screen is displayed any time the user presses -

The file that you load must contain less than 1023 characters. Spaces and carriage returns count as characters. You can use the EDITFILE command to create the text for your help screen and the FILELEN operation to verify that the file is not too long.

Example:

```
?LOADHELP ""/LOGO.SAMPLES/NEWHHELP
```

ONLINE

ONLINE (operation)

ONLINE outputs the volume name of every disk on line. For example, if you have two disk drives connected, and a disk in each of them, ONLINE outputs the names of both those disks.

Example:

```
?SHOW ONLINE  
[/LOGO.SAMPLES/]
```

You might want to use ONLINE when you have a disk and you cannot remember the name you gave it. Just put it in a drive and type PR ONLINE. Logo displays the name of the disk.

POFILE

POFILE *pathname* (command)

POFILE (for print out file) prints out the contents of the file indicated by *pathname*. Logo prints the contents to the screen. An error occurs if you try to use POFILE on a file that is already open.

This procedure can be used to copy a file:

```
TO COPY :TO :FROM  
DRIBBLE :TO  
POFILE :FROM  
NODRIBBLE  
END
```

To copy a file POLYS to a file SHAPES:

```
?COPY "POLYS "SHAPES
```

PREFIX

PREFIX (operation)

PREFIX outputs the current ProDOS prefix. You use SETPREFIX to set the prefix.

```
?PR PREFIX  
/LOGO.SAMPLES/  
?SETPREFIX "PICTURES  
?PR PREFIX  
/LOGO.SAMPLES/PICTURES/
```

RENAME

RENAME *pathname newpathname* (command)

The RENAME command finds the file indicated by *pathname* on the disk and changes its name to *newpathname*. The file's contents are not affected. *Newpathname* must specify a file in the same directory as *pathname*.

Example:

```
?RENAME "/LOGD.SAMPLES/DATA/ADDRESS "/L!  
DGD.SAMPLES/DATA/ADDRESS.DLD
```

renames the file ADDRESS to ADDRESS.DLD.

SETPREFIX

SETPREFIX *prefix* (command)

SETPREFIX tells Logo to set the ProDOS prefix to *prefix*. This command lets you access a file in the subdirectory named by *prefix* without having to type its full pathname. It also affects what the CATALOG command prints.

Examples:

```
?SETPREFIX "/LOGD.SAMPLES/PROGRAMS  
?CATALOG  
/LOGD.SAMPLES/PROGRAMS/  
  GAMES/  
  PICTURES/  
Blocks Free: 138 Blocks Used: 142  
?
```

You can now access the files or subdirectories under the subdirectory PROGRAMS—in this case, GAMES and PICTURES—by the filenames alone.

```
To access files in the root directory,  
?SETPREFIX "/LOGO.SAMPLES  
?CATALOG  
/LOGO.SAMPLES/  
  PROGRAMS/  
  PICTURES/  
  DATA/  
Blocks Free: 138 Blocks Used: 142  
?
```



Managing Various Files

- 206** Working With Program Files
 - 206** LOAD
 - 206** SAVE
 - 207** SAVEL
- 207** Working With Picture Files
 - 208** LOADPIC
 - 208** PRINTPIC
 - 208** SAVEPIC
- 209** Working With Dribble Files
 - 209** DRIBBLE
 - 210** NODRIBBLE
- 211** Working With Data Files
 - 211** Reading and Writing Information
 - 211** Opening Files
 - 212** ALLOPEN
 - 212** CLOSE
 - 213** CLOSEALL
 - 214** FILELEN
 - 215** OPEN
 - 216** READER
 - 217** READPOS
 - 218** SETREAD
 - 218** SETREADPOS
 - 219** SETWRITE
 - 220** SETWRITEPOS
 - 221** WRITEPOS
 - 221** WRITER

- 221** A Sample Project Using the Data File System
- 222** Step 1: Creating a Data File
- 224** Step 2: Retrieving Information
- 225** Step 3: Changing Information

Managing Various Files



Chapter 15 describes general file management with Logo.

This chapter gives you information about the various types of files that Logo uses.

This chapter is divided into five main sections, which provide

- the primitives for working with program files
- the primitives for working with picture files
- the primitives for working with dribble files
- the primitives for working with data files
- a sample project using data files.

The examples in this chapter are based on the disk named LOGO.SAMPLES, which is used for illustration in the previous chapter. You may want to refer to that disk's overall directory structure (see Figure 15-1) when you are trying them out.

Logo reads information from three sources: files on a disk, the keyboard, and some devices that are attached to the computer. When you start up Logo, it reads information from the keyboard.

Likewise, Logo writes information to three destinations: files on a disk, the screen, and the devices attached to the computer. When you start up Logo, it writes information onto the screen.

Note: A **device** is a piece of hardware that is attached to the computer through a slot (on the Apple IIe) or a port (on the Apple IIc). It is important to note that Logo treats the keyboard, the screen, and other devices such as a printer, as files, just as it treats information on a disk as files.

Some file primitives work with both files on disks and devices like printers. In this chapter, the input term *file* represents inputs of this kind. The devices are accessed through the port or slot number to which they are attached. The most common device you will access this way is a printer. A printer attached to port 1 or slot 1 would be accessed by the number 1.

Working With Program Files

This section tells you how to save and load files containing Logo programs. The primitives you use to do this are

LOAD
SAVE
SAVEL

LOAD

LOAD *pathname* (command)

The LOAD command loads the contents of the file indicated by *pathname* into the workspace, as if you typed it directly from top level. An error occurs if the file does not exist. An error also occurs if you try to load to a device.

After Logo loads the contents of a file, it looks for a variable called STARTUP. If one exists, Logo executes its contents.

Examples:

```
?SETPREFIX "/PROGRAMS/PICTURES  
?LOAD "BEAR
```

Logo reads everything in the file BEAR into the workspace.

SAVE

SAVE *pathname* (command)

The SAVE command creates a file and saves in it all unburied procedures and variables and all properties in the workspace. An error occurs if the file you name already exists. In this case,

you should first either erase the existing file using ERASEFILE or rename it using RENAME. An error also occurs if you try to save to a device.

Examples:

```
?SAVE **/PROGRAMS/PICTURES/FACES
```

saves the contents of the workspace in the file called FACES.

SAVEL

SAVEL *name(list) pathname* (command)

The SAVEL command saves the procedures named in *name(list)*, and all the unburied variables and properties in the workspace to *pathname*. This command is useful for saving a portion of your workspace onto a disk. An error occurs if you try to save to a device using SAVEL. Compare it with SAVE.

Example:

```
?POTS  
TO TRI :OBJECT  
TO POLY :SIDE :ANGLE  
TO SPI :SIDE :ANGLE :INC  
TO INSPI :SIDE :ANGLE :INC  
TO WELCOME :NAME  
  
?SAVEL [POLY SPI INSPI] **/LOGD.SAMPLES/!  
PROGRAMS/PICTURES/POLYS
```

Working With Picture Files

This section describes the primitives you use to load, save, and print Logo pictures. The primitives are

```
LOADPIC  
PRINTPIC  
SAVEPIC
```

LOADPIC

LOADPIC *pathname* (command)

The LOADPIC command loads the picture named by *pathname* onto the graphics screen. Logo will load any file onto the graphics screen. If the file is not a picture, something will be put on the graphics screen, but you cannot be sure what it will be.

Example:

```
?LOADPIC "/LOGO.SAMPLES/PICTURES/CAT.PIC
```

loads the picture contained in the file CAT.PIC onto the graphics screen.

PRINTPIC

PRINTPIC *integer* (command)

PRINTPIC prints the contents of the graphics screen to the printer in the slot or port named. You can print pictures only to the Apple Imagewriter printer. If you try to use this primitive with other printers, the results are unpredictable.

Example:

```
?PRINTPIC 1
```

SAVEPIC

SAVEPIC *pathname* (command)

SAVEPIC saves the graphics screen into the file indicated by *pathname*. You can retrieve the screen later using LOADPIC.

Example:

```
?SAVEPIC "/LOGO.SAMPLES/PICTURES/CAT.PIC
```

Working With Dribble Files

This section describes the two primitives that you use to record the interactions between you and the Apple computer. The primitives are DRIBBLE and NODRIBBLE.

DRIBBLE

DRIBBLE *file* (command)

DRIBBLE starts the process of sending a copy of the characters displayed on the text screen to *file*. DRIBBLE records interactions between the Apple computer and the person at the keyboard. DRIBBLE automatically opens *file*. NODRIBBLE stops the process of dribbling.

You cannot use SETREAD or SETWRITE with a dribble file while still dribbling. However, once a dribble file on disk has been closed with NODRIBBLE, you can treat it like any other file. You can then open it, read from it, or write to it.

Note that only one dribble file can be open at one time.

Examples:

```
?DRIBBLE /DATA/RECORDS/JUNE1.DRIB
```

creates a file called JUNE1.DRIB and starts the dribbling process. Every line appearing after DRIBBLE is sent to this file.

```
?CS  
?FD 100  
?RT 80  
?FD 50  
?NODRIBBLE
```


DRIBBLE can be used to print the contents of a file to the printer.

```
TO DUMP :FILE
DRIBBLE 1
POFILE :FILE
NODRIBBLE
END
```

NODRIBBLE

NODRIBBLE (command)

NODRIBBLE turns off the dribble feature so a copy of the characters from the screen will no longer be sent to the file or device named previously by the DRIBBLE command.

Examples:

```
?DRIBBLE "/LOGO.SAMPLES/DATA/RECORDS/CLASS.DRIB
```

creates a file called CLASS.DRIB and starts the dribbling process.

```
?REPEAT 5 [PR RANDOM 10]
8
0
3
3
2
?NODRIBBLE
```

Everything put on the text screen after the DRIBBLE line is sent to the file CLASS.DRIB. Now, if you print out the file CLASS.DRIB, you will see what you just typed.

```
?POFILE "/LOGO.SAMPLES/DATA/RECORDS/CLASS.DRIB
?REPEAT 5 [PR RANDOM 10]
8
0
3
3
2
?NODRIBBLE
```

Working With Data Files

This section gives you information about

- reading and writing information in data files
- opening and closing data files
- the primitives that work with data files.

Reading and Writing Information

With Logo's file system, there is always a current file open for reading, called the **reader**, and a current file open for writing, called the **writer**. When you start up Logo, Logo assumes that the current reader is the keyboard and the current writer is the screen. You can change the current reader and writer files with the SETREAD and SETWRITE commands, which are described later in this chapter.

When the current reader or writer is a file on disk, there are current positions in the file where Logo will start reading or writing. For example, when Logo opens a file, it is ready to read from the beginning of the file and write at the end. You can change the read and write positions with the SETREADPOS and SETWRITEPOS commands, which are described later in this chapter.

Opening Files

You must open a file or device with the OPEN command before you can read from it or write to it. Only one device can be open at a time although you can open as many as six files. So, if a device is currently open, you cannot use a primitive that automatically opens and closes devices. For example, you cannot use the DRIBBLE command for a printer in slot 1 or port 1 if slot 2 or port 2 is already open.

The data file primitives are

ALLOPEN	READER	SETWRITE
CLOSE	READPOS	SETWRITEPOS
CLOSEALL	SETREAD	WRITEPOS
FILELEN	SETREADPOS	WRITER
OPEN		

ALLOPEN

ALLOPEN

(operation)

ALLOPEN outputs a list of all files and devices currently open. The OPEN command opens a file or a device.

```
?PRINT ALLOPEN
```

```
?
```

No files or devices are open.

```
?PRINT ALLOPEN
```

```
1 PHONELIST
```

The device in slot 1 or port 1 (the printer) and the file called PHONELIST are open.

The procedure BYE makes sure all files are closed before you turn off the machine.

```
TO BYE
```

```
IF NOT EMPTY ALLOPEN [CLOSEALL]
```

```
PR [YOU CAN NOW TURN OFF THE POWER.]
```

```
END
```

CLOSE

CLOSE *file*

(command)

The CLOSE command closes the named file or device that is currently open. See OPEN to open a file or device. An error occurs if you try to use CLOSE with a file or device that is not open. An error also occurs if you try to use CLOSE with a file that is opened by the DRIBBLE command.



Warning

It is important that you never turn off your computer while files are open. This can damage the integrity of your disk.

Examples:

```
?CLOSE "/LOGO.SAMPLES/DATA/PHONELIST
```

closes the file called PHONELIST.

The STORE procedure opens a file, sends data to it, and closes the file.

```
TO STORE :FILE :DATA  
OPEN :FILE  
SETWRITE :FILE  
PRINT :DATA  
SETWRITE []  
CLOSE :FILE  
END
```

```
?STORE "/LOGO.SAMPLES/DATA/PHONELIST [B!  
ARBARA: 765-4201]
```

The name and telephone number are written to the file called PHONELIST.

CLOSEALL

CLOSEALL (command)

The CLOSEALL command closes all files and devices that are currently open. Dribble files are not closed with CLOSEALL.

Use the OPEN and CLOSE commands to open and close one file at a time. If you try to use CLOSEALL when no files or device are open, it is ignored.

```
?OPEN 1  
?OPEN "/LOGO.SAMPLES/DATA/PHONELIST
```

You opened the printer in slot 1 or port 1, and a file called PHONELIST. After sending data to the file and to the printer, you can close both by typing

```
?CLOSEALL
```

See section "NODRIBBLE" for closing dribble files.

FILELEN

FILELEN *pathname* (operation)

FILELEN outputs the length in bytes of the contents of the file indicated by *pathname*. The file must be open to use this primitive. An error occurs if the file is not open.

Example:

```
?OPEN "/LOGD.SAMPLES/DATA/RECORDS/ADDRESSES
?PRINT FILELEN "/LOGD.SAMPLES/DATA/RECORDS/ADDRESSES
128
```

The file called ADDRESSES already has 128 bytes of data.

```
TO FILLIN :FILE :LEN
OPEN :FILE
SETWRITE :FILE
MAKE "SPACE :LEN - FILELEN :FILE
IF :SPACE > 0 [REPEAT :SPACE [TYPE 0]]
SETWRITE []
CLOSE :FILE
END
```

The procedure FILLIN opens the file :FILE and fills it in with zeros so the file will be :LEN bytes long.

OPEN

OPEN *file* (command)

The OPEN command opens *file* so it can send or receive characters. You must open a data file before you can access it. Note that you can open only one device at a time.

You can open a maximum of six disk files at once. If the file named by *file* does not exist, then OPEN creates the file. When you finish using Logo, you must close all devices or files that are open.

See the CLOSE and CLOSEALL commands.

Example:

```
TO READFILE :FILE
SETREAD :FILE
IF EQUALP FILELEN :FILE READPOS [SETREA!
D [] CLOSE :FILE STOP]
PRINT READLIST
READFILE :FILE
END

?SETPREFIX "/LOGO.SAMPLES/DATA/RECORDS
?OPEN "ADDRESSES
?READFILE "ADDRESSES
ADDRESS LIST
MARIE: 55 CEDARWOOD
LOGO: 9960 COTE DE LIESSE
```

The READFILE procedure reads information from a file that is already open until the end-of-file position (EQUALP FILELEN :FILE READPOS) is reached. At that time, the file is closed and execution of the procedure stops.

READER

READER (operation)

READER outputs the current file that is open for reading. You can change the current read file with the SETREAD primitive. READER returns the name of the file or the empty list if the current reader is the keyboard.

Examples:

```
?PRINT READER
/LOGO.SAMPLES/DATA/RECORDS/ADDRESS
```

The file called ADDRESS in the subdirectory DATA/RECORDS is the current read file.


```
TO CHECKREAD :FILE
IF NOT EQUALP READER :FILE [OPEN :FILE !
SETREAD :FILE]
IF EQUALP FILELEN :FILE READPOS [CLOSE !
:FILE SETREAD [] STOP]
PRINT READLIST
CHECKREAD :FILE
END
```

```
?CHECKREAD "/LOGO.SAMPLES/DATA/CLASS.LI!
ST
ERIC BROWN
MICHAEL QUINN
CHERYL BARTLEY
JENNY SPARROW
```

The CHECKREAD procedure checks to see if the file it has as input is the current read file. If it is not, CHECKREAD opens the file, makes it the current read file, and then reads until reaching the end-of-file position.

READPOS

READPOS (operation)

READPOS (for READ POSition) outputs the position in the current reader. An error occurs if the current reader is the keyboard or a device. To set the position in the read file, see the SETREADPOS command.

Examples:

```
?SETPREFIX "/LOGO.SAMPLES/DATA/RECORDS
?OPEN "PHONELIST
?SETREAD "PHONELIST
?PRINT READPOS
0
```

If you have just opened a file, READPOS outputs 0.

The procedure LISTFILE lists the information stored in the read file, along with a number indicating where each line is stored.


```
TO LISTFILE :FILE
IF EQUALP FILELEN :FILE READPOS [STOP]
PRINT READPOS
PRINT READWORD
LISTFILE :FILE
END
```

```
?OPEN "PHONELIST
?SETREAD "PHONELIST
?LISTFILE "PHONELIST
0
PASCAL 545-2654
16
MARIO 631-2222
```

SETREAD

SETREAD *file* (command)

SETREAD sets the current reader to *file*. After you give this command, READLIST, READWORD, READCHAR, and READCHARS read information from this file.

Before you use SETREAD, you must open the file with the OPEN command. An error occurs if the file is not open. To set the current reader back to the keyboard, give SETREAD the empty list as input.

Examples:

```
?SETPREFIX "/LOGO.SAMPLES/DATA
?OPEN "PHONELIST
?SETREAD "PHONELIST
?PRINT READPOS
0
```

The reader is set to PHONELIST and the read position is at the beginning of the file.

```
?PRINT READWORD
PASCAL: 545-2654
```

READLIST reads from the current reader. To set the reader back to the keyboard:

```
?SETREAD [ ]
```

See section "READPOS" for more information about the SETREADPOS command.

SETREADPOS

SETREADPOS *integer* (command)

SETREADPOS sets the read position in the current reader. The *integer* should be a number between 0 and the current length of the file. An error occurs if it is not in this range. An error also occurs if the current reader is the keyboard or a device.

Examples:

```
?OPEN "PHONELIST
?SETREAD "PHONELIST
?SETREADPOS 2
?PRINT READCHAR
S
```

The file PHONELIST is opened and set up for reading. The read position is set to 2, and the character at that position is printed.

```
TO FILERL :POS
SETREADPOS :POS
OUTPUT READWORD
END
```

```
?PRINT FILERL 34
RENAUD 734-8374
```

The FILERL procedure outputs the list found at the file position you gave as input.

SETWRITE

SETWRITE *file* (command)

SETWRITE sets the current writer to the file you name. The primitives PRINT, TYPE, and SHOW all print to the current writer. You cannot use SETWRITE unless the file has previously been opened.

To restore the screen as the current writer, use the SETWRITE command with the empty list as input.

Note: The commands PO, POALL, PON, PONS, POPS, POT, POTS, and POFILE all print to the screen but not to the current writer.

Examples:

```
?OPEN 1
?SETWRITE 1
```

Now the various print commands will send information to the device in slot 1 or port 1.

```
?PRINT [LOGO TELEPHONE DIRECTORY]
```

If the device in slot 1 or port 1 is a printer, LOGO TELEPHONE DIRECTORY is printed there.

```
?SETWRITE []
```

The current writer is set back to the screen.

```
TO STORE :FILE :DATA
DPEN :FILE
SETWRITE :FILE
PRINT :DATA
SETWRITE []
CLOSE :FILE
END
```

```
?STORE "PHONELIST [BRIAN: 451-2513]
```

SETWRITEPOS

SETWRITEPOS *integer* (command)

SETWRITEPOS sets the write position in the current file. This command is useful when modifying information in a file. You must set the write position to a number that is between 0 and the end-of-file position. If you try to set it somewhere out of this range, an error occurs.

An error also occurs if you try to set the write position when the current writer is the screen or a device.

To check the current position, use the WRITEPOS command.

Examples:

```
?OPEN "PHONELIST
?SETWRITE "PHONELIST
?SETWRITEPOS 0
?PRINT [MARIE 935-3395]
?SETWRITE [ ]
```

The file PHONELIST is opened, selected for writing, and the write position is set to 0 (it was at the end of the file when the file was opened). The list that is printed replaces whatever was at the beginning of the file.

WRITEPOS

WRITEPOS (operation)

WRITEPOS (for write position) outputs where in the current write file the the next character will be written. An error occurs if the current writer is the screen or a device.

Examples:

```
?OPEN "PHONELIST
?SETWRITE "PHONELIST
?MAKE "POS WRITEPOS
?SETWRITE [ ]
?PR :POS
33
```

Notice that you can't use PRINT WRITEPOS directly because the write position will be printed into the file PHONELIST.

The CHECKPOS procedure prints the file position of the current writer.

```
TO CHECKPOS
MAKE "POS WRITEPOS
MAKE "FILE WRITER
SETWRITE [ ]
PR :POS
SETWRITE :FILE
END

?CHECKPOS "PHONELIST
33
```

WRITER

WRITER (operation)

WRITER outputs the current file or device that is open for writing. Compare this with the ALLOPEN operation.

Examples:

```
TO CHECKWRITE :FILE :DATA
IF NOT MEMBERP :FILE ALLOPEN [OPEN :FILE]
MAKE "OLDWRITER WRITER
SETWRITE :FILE
PRINT :DATA
SETWRITE :OLDWRITER
END

?CHECKWRITE "/LOGO.SAMPLES/DATA/CLASS.L!
1ST [KIYOKO OKUMURA]
```

The CHECKWRITE procedure first determines if a file is open. If it is not, CHECKWRITE opens the file, makes it the current writer, and sends data to it. CHECKWRITE maintains the original writer.

A Sample Project Using the Data File System

This section examines the data file system, using a telephone directory project as an example. You want to store the telephone numbers of the members of a social club. The objectives of the project are:

1. To store the members' names and their phone numbers.
2. To find a particular member's phone number.
3. To change a member's phone number.

Step 1: Creating a Data File

Here is a procedure that reads the name and phone number of someone from the keyboard.

```
TO ASKINFO
PRINT [Type in the member's name:]
MAKE "NAME READWORD
PRINT [Type in the phone number:]
MAKE "TEL READWORD
END
```

ASKINFO prints the message on the screen, takes the answer from the keyboard, and gives a name to this answer. When ASKINFO finishes its job, it creates two variables: one is called NAME and the other TEL. The next step is to write the information into a file.

Logo lets you write to files (or devices) as easily as you can write to the screen. In addition, Logo lets you read from a file as easily as you read from the keyboard.

The SETWRITE command is used to direct information to different files or devices.

```
TO WRITEINFO
SETWRITE "MEMBERS      (MEMBERS is the filename)
PRINT :NAME
PRINT :TEL
SETWRITE []           (directs output back to the
                       screen)
END
```

All that remains now is to write the superprocedure to open the data file called MEMBERS, run these subprocedures, and close the file.

```
TO SAVEINFO
OPEN "MEMBERS
ASKINFO
WRITEINFO
CLOSE "MEMBERS
END
```

Let's try the procedure now.

```
?SAVEINFO
Type in the member's name:
Mario Carriere
Type in the phone number:
423-5800
?
```

The program finished running, but you can't see what happened to the data file. To check the result, print out the file.

```
?PDFILE "MEMBERS
```

Logo displays everything written in the data file MEMBERS.

```
Mario Carriere
423-5800
```

What happens if we run the procedure again?

```
?SAVEINFO
Type in the member's name:
Renaud Nadeau
Type in the phone number:
392-1563
?
```

SAVEINFO worked just like it did the first time. Now look at the result.

```
?PDFILE "MEMBERS
Mario Carriere
423-5800
Renaud Nadeau
392-1563
?
```

The procedures work for adding more members as well as for creating the data file for the first time.

Step 2: Retrieving Information

After creating the data file containing names and phone numbers, the next step is to build a program to find a particular member's phone number.

```
TO FINDINFO
PRINT [Type in the member's name:]
MAKE "NAME READLIST
OPEN "MEMBERS
SETREAD "MEMBERS
FINDTEL :NAME
SETREAD []
CLOSE "MEMBERS
END

TO FINDTEL :NAME
IF READLIST = :NAME [PR SE [The phone n!
umber is:] READWORD STOP]
IF EQUALP FILELEN "MEMBERS READPOS [PR !
[Can't find this name.] STOP]
FINDTEL :NAME
END
```

FINDINFO is the superprocedure. First, it reads from the keyboard the name of the person whose phone number is wanted. Then, it opens the data file and tells Logo that it wants to read information from this data file.

The subprocedure FINDTEL starts reading line by line (using READLIST) from the beginning of this data file. Each time it reads a line, FINDTEL compares the line with the name it is looking for. If they are identical, it reads another line and prints the sentence

The phone number is:

If not, it checks to see if READLIST has reached the end-of-file position (EQUALP FILELEN "MEMBERS READPOS). If the end-of-file position has been reached, FINDTEL prints the message

Can't find this name.

Step 3: Changing Information

A member's phone number may change, so you must be able to update the data in the file. To modify part of the data, you must know the location of the information to be changed. The procedures to retrieve the information (FINDINFO and FINDTEL) can be used for this purpose. Once the location is found, you can write the procedure MODIFY, which rewrites the information at this location.

```
TO MODIFY :LOCATION
PRINT [Type in the new phone number:]
SETREAD []
SETWRITE "MEMBERS
SETWRITEPOS :LOCATION
PRINT READWORD
SETWRITE []
END
```

SETREAD [] tells Logo that you want to read the data from the keyboard. SETWRITE "MEMBERS tells Logo that you want to direct the next PRINT command to write the new data into the MEMBERS file. SETWRITEPOS :LOCATION makes sure that it is written at the current location.

Thus, the command PRINT READWORD picks up data from the keyboard and prints it into the file.

Now you must incorporate this procedure into a new FINDTEL procedure. FINDTEL2 will read the file line by line comparing each line to the name it is looking for. It will then call MODIFY with the LOCATION it gets from READPOS in the procedure FINDTEL. READPOS is the input to MODIFY. Let's change the name of the superprocedure FINDINFO to MODINFO.

```
TO MODINFO
PRINT [Type in the member's name:]
MAKE "NAME READLIST
OPEN "MEMBERS
SETREAD "MEMBERS
FINDTEL2 :NAME
SETREAD []
CLOSE "MEMBERS
END

TO FINDTEL2 :NAME
IF RL = :NAME [MODIFY READPOS STOP]
IF EQUALP FILELEN "MEMBERS READPOS [PR!
[Can't find the name.] STOP]
FINDTEL2 :NAME
END
```

Property Lists

- 229** Using Property Lists to Keep Records
- 230** ERPROPS
- 230** GPROP
- 231** PLIST
- 232** PPROP
- 232** PPS
- 233** REMPROP

Property Lists



Any Logo word can have a property list associated with it. A **property list** consists of an even number of elements. Each pair of elements consists of a property, and its value, a word or a list.

A property list has the form [prop1 val1 prop2 val2 ...]. You can manipulate property lists using the primitives in this section:

```
ERPROPS
GPROP
PLIST
PPROP
PPS
REMPROP
```

SAVE and SAVEL are described in Chapter 16.

The commands SAVE and SAVEL save property lists in files at the same time they save procedures and variable names.

Using Property Lists to Keep Records

Property lists can be very useful in keeping records or other structured data bases. The following example is used as a context for explaining the property list primitives.

Suppose you want to keep track of the telephone numbers and birthdays of your friends. Invent a Logo word, say F1, to act as a placekeeper for your first friend. Then write

```
PPROP "F1 "NAME [BRIAN SILVERMAN]
PPROP "F1 "PHONE [514 555 4123]
PPROP "F1 "BIRTHDAY [SEPT 23]
```

Do this for all your friends, giving your second friend the placekeeping word F2 and so on. For example:

```
PPROP "F2 "NAME [EFFIE MANIATIS]
PPROP "F2 "PHONE [514 631 6123]
PPROP "F2 "BIRTHDAY [MAY 20]

PPROP "F3 "NAME [MICHAEL QUINN]
PPROP "F3 "PHONE [619 742 5555]
PPROP "F3 "BIRTHDAY [DEC 3]
```

After you have finished, make a list of the placekeeping words like this:

```
MAKE "FRIENDS [F1 F2 F3 ]
```

You can then use GPROP to write procedures that search through the list FRIENDS to do such things as find a given friend's birthday or list all your friends with the same area code. Examples of such procedures appear with the primitive descriptions that follow.

ERPROPS

ERPROPS (command)

ERPROPS (for erase properties) erases all properties from the workspace. To check which property lists are currently in the workspace, use PPS. Use REMPROP to remove properties one at a time from the workspace.

GPROP

GPROP *name property* (operation)

GPROP (for get property) outputs the value of *property* of *name*. If there is no such property, GPROP outputs the empty list.

Examples:

```
?SHOW GPROP "F1 "NAME
[BRIAN SILVERMAN]
```


The phone list procedure lists your friends' names and phone numbers.

```
TO PHONELIST :FRIENDS
IF EMPTY :FRIENDS [STOP]
PR SE GPROP FIRST :FRIENDS "NAME GPROP !
FIRST :FRIENDS "PHONE
PHONELIST BF :FRIENDS
END
```

```
?PHONELIST :FRIENDS
BRIAN SILVERMAN 514 555 4123
EFFIE MANIATIS 514 631 6123
MICHAEL QUINN 619 742 5555
```

PLIST

PLIST *name* (operation)

PLIST outputs the property list associated with *name*. This is a list of property names paired with their values, in the form [prop1 val1 prop2 val2 ...].

Example:

```
?SHOW PLIST "F2
[NAME [EFFIE MANIATIS] BIRTHDAY [MAY 20!
]PHONE [514 631 6123]]
```

The FINDBIRTH procedure outputs the birthday of a given friend.

```
TO FINDBIRTH :FRIEND :FRIENDS
IF EMPTY :FRIENDS [OP [NONE] ]
IF EQUALP FIRST BF PLIST FIRST :FRIENDS!
:FRIEND [OP GPROP FIRST :FRIENDS "BIRT!
HDAY]
OP FINDBIRTH :FRIEND BF :FRIENDS
END

?PR FINDBIRTH [MICHAEL QUINN] :FRIENDS
DEC 3
```

PPROP

PPROP *name property object* (command)

The PPROP (for put property) command gives *name property* with value *object*. Note that ERALL erases procedures, variables, and properties. Use REMPROP to erase properties one at a time or ERPROPS to erase them all at once.

Example:

```
?SHOW PLIST "F3
[NAME [MICHAEL QUINN] PHONE [619 742 55!
55] BIRTHDAY [DEC 3]]
?PPROP "F3 "ADDRESS [55 OAKRIDGE]
?SHOW PLIST "F3
[NAME [MICHAEL QUINN] PHONE [619 742 55!
55] BIRTHDAY [DEC 3] ADDRESS [55 OAKRID!
GE]]
```

PPS

PPS (command)

The PPS (for print properties) command prints the property lists of everything in the workspace.

Example:

```
?PPS
PPROP "F3 "NAME [MICHAEL QUINN]
PPROP "F3 "PHONE [619 742 5555]
PPROP "F3 "BIRTHDAY [DEC 3]
PPROP "F3 "ADDRESS [55 OAKRIDGE]
PPROP "F2 "NAME [EFFIE MANIATIS]
PPROP "F2 "PHONE [514 631 6123]
PPROP "F2 "BIRTHDAY [MAY 20]
PPROP "F1 "NAME [BRIAN SILVERMAN]
PPROP "F1 "PHONE [514 555 4123]
PPROP "F1 "BIRTHDAY [SEPT 23]
```

REMPROP

REMPROP *name property* (command)

See also sections "PPROP" and "GPROP."

The REMPROP (for remove property) command removes *property* from the property list of *name*.

Example:

```
? SHOW PLIST "F1
[NAME [BRIAN SILVERMAN] BIRTHDAY [SEPT !
23] PHONE [514 555 4123]]
?REMPROP "F1 "PHONE
?SHOW PLIST "F1
[NAME [BRIAN SILVERMAN] BIRTHDAY [SEPT !
23]]
```



Special Primitives

- 238** Assembly-Language Primitives
- 238** Some Specifics About the Apple's Memory
- 241** Using Buffer Space
- 241** Using Node Space
- 241** .AUXDEPOSIT
- 242** .AUXEXAMINE
- 242** .BLOAD
- 242** .BSAVE
- 242** .CALL
- 242** .DEPOSIT
- 243** .EXAMINE
- 243** Special Graphics Primitives
- 243** .SCRUNCH
- 243** .SETSCRUNCH
- 245** Miscellaneous Primitives
- 245** .CONTENTS
- 245** .QUIT

Special Primitives



This chapter presents some special primitives that may affect the Logo system itself. These primitives give you the power of directly accessing the computer memory or modifying what's in it. At the same time they are dangerous primitives because you can destroy the contents of your workspace in Logo by using them carelessly. If that happens, you will need to restart Logo. The names of these primitives start with a period to warn you that they are dangerous. You should save your work before experimenting with them.

The special primitives appear in three groups:

- assembly-language and direct-memory-access primitives
- special graphics primitives
- miscellaneous primitives.

Assembly-Language Primitives

This section explains the special primitives that allow you to use assembly-language programs from Logo and to directly access memory. It also gives you some specific information about the Apple's memory that is useful for programming in assembly language.

The primitives appear in this order:

```
.AUXDEPOSIT  
.AUXEXAMINE  
.BLOAD  
.BSAVE  
.CALL  
.DEPOSIT  
.EXAMINE
```

See the technical reference manual for your computer for a more complete explanation of the memory layout.

Some Specifics About the Apple's Memory

The Apple II's memory is divided into two 64K banks: the main bank and the auxiliary bank. The following memory maps show you how Logo uses these two banks.

Figure 18-1. Map of Main Memory Bank

Main Memory	
FFFF	ProDOS
D000	I/O Space
C000	Free Space and ProDOS
BF00	Logo Code
6100	Logo Data
6000	File Buffer 5
5C00	File Buffer 4
5800	File Buffer 3
5400	File Buffer 2
5000	File Buffer 1
4C00	File Buffer 0
4800	Dribble Buffer
4400	Load/Save Buffer
4000	Hi-Res Graphics
2000	Edit Buffer
800	Text Screen 1
400	Logo Data
0	

Figure 18-2. Map of Auxiliary Memory Bank

Auxiliary Memory

FFFF	ProDOS
E000	Logo Code
D000	I/O Space
C000	ProDOS
BF00	Editor Help
BB00	Main Help
B700	Node Space
800	Text Screen 2
400	ProDOS
200	Logo Data
0	

There are some specific locations in the two banks that you need to know about if you're writing assembly-language programs. Table 18-1 presents these locations.

Table 18-1. Special Memory Locations

Information	Location		Normal Value	
	Hex	Decimal	Hex	Decimal
Maximum number data files (multiplied by 9)	300	768	36	54
Pointer to first page beyond node space	10	16	B7	183
Flag for invalid edit buffer	301	769	0	0

Using Buffer Space

You can use the edit buffer, graphics buffer, and file buffers for your programs if these buffers are not being used by Logo while your programs are running.

The edit buffer and graphics buffer should be used only for temporary storage, that is, storage that you need only while your assembly code is executing. If you use the edit buffer, make sure you mark the flag for indicating invalid contents of the edit buffer. If you use the graphics buffer, be sure to clear it out when you're finished to prevent unpredictable graphic displays.

The file buffers can also be used for assembly-language code. To make sure that Logo does not use the buffers you are using, you must change the number of files that Logo can use at the location indicated in Table 18-1. Note that the number stored is nine times the number of files Logo will handle.

If you need 2K bytes for your code, you can change the number of files Logo can have open from 54 to 36. Doing this frees file buffers 4 and 5 for your use.

Using Node Space

You can use node space for assembly-language programs. The only time you can reserve the node space is when Logo first starts up, no matter when you intend to actually use it. You reserve space by changing the address of the end of node space shown in Table 18-1.

When Logo first starts up, node space extends from \$800 to \$B6FF; the end being \$B6FF. To reserve 8K bytes of node space for your use, change the \$B7 at the location indicated in Table 18-1 to \$97. You must remember to free up nodes in multiples of five bytes (node length).

.AUXDEPOSIT

`.AUXDEPOSIT loc byte` (command)

The `.AUXDEPOSIT` command stores the value *byte* at address *loc* in the auxiliary bank.

.AUXEXAMINE

`.AUXEXAMINE loc` (operation)

The `.AUXEXAMINE` operation outputs the value stored at address *loc* in the auxiliary bank.

.BLOAD

`.BLOAD pathname loc` (command)

The `.BLOAD` command loads a binary-format file, consisting of data or assembly-language code, into address *loc* in the main bank of memory.

.BSAVE

`.BSAVE pathname loc integer` (command)

The `.BSAVE` command copies an area of the main bank of computer memory to the file indicated by *pathname*. The memory area transferred starts at *loc* for *integer* bytes.

.CALL

`.CALL loc` (command)

The `.CALL` command transfers control to the indicated machine-language subroutine starting at address *loc* (decimal) in the main bank. An RTS in your subroutine returns control back to Logo.

.DEPOSIT

`.DEPOSIT loc byte` (command)

The `.DEPOSIT` command writes *byte* into machine address *loc* (decimal) in main memory.

.EXAMINE

`.EXAMINE loc` (operation)

The `.EXAMINE` operation outputs the contents of machine address *loc* (decimal) in main memory.

Special Graphics Primitives

The special graphics primitives let you review and change the **aspect ratio**, the ratio of lengths of vertical turtle steps to horizontal turtle steps. This ratio is set to 0.8 when you start up Logo.

You will want to change this ratio if squares that you draw on the screen appear as rectangles, and circles that you draw appear as ellipses.

.SCRUNCH

`.SCRUNCH` (operation)

See also section “`.SETSCRUNCH`.”

The `.SCRUNCH` operation outputs the **aspect ratio**, a decimal number that is the ratio of the size of a vertical turtle step to the size of a horizontal one. The aspect ratio is 0.8 when Logo starts up.

.SETSCRUNCH

`.SETSCRUNCH number` (command)

`.SETSCRUNCH` sets the aspect ratio to *number*. The aspect ratio is the ratio of the size of a vertical turtle step to the size of a horizontal one. If you change the aspect ratio, the value of your YCOR is changed so the turtle appears in the same place on the screen.

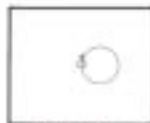
Example:

.SETSCRUNCH .5 makes each vertical turtle step half the length of a horizontal one.

.SETSCRUNCH has two uses. First, if squares turn out to be rectangles, and circles turn out to be ellipses on your screen, you can correct this; for most screens an aspect ratio of .8 is correct. Second, if you want turtle drawings to come out squashed or extended, you can use .SETSCRUNCH. For example, you can use a circle procedure to draw an ellipse:

```
TO CIRCLE :RADIUS
REPEAT 60 [FD :RADIUS * 3.14159 / 30 RT!
 6]
END

TO ELLIPSE :HORIZ :VERT
.SETSCRUNCH .8 * :VERT / :HORIZ
CIRCLE :HORIZ
END
```



CIRCLE 25



ELLIPSE 25 40



ELLIPSE 40 25

Miscellaneous Primitives

This section describes two miscellaneous primitives, `.CONTENTS` and `.QUIT`.

.CONTENTS

`.CONTENTS` (operation)

The `.CONTENTS` operation outputs a list of all objects that Logo knows about. This list includes your variables, procedures, and properties, the Logo primitives, most of the things you've typed in, and some other words. `.CONTENTS` can use up a lot of node space.

See Appendix D for more information on node space.

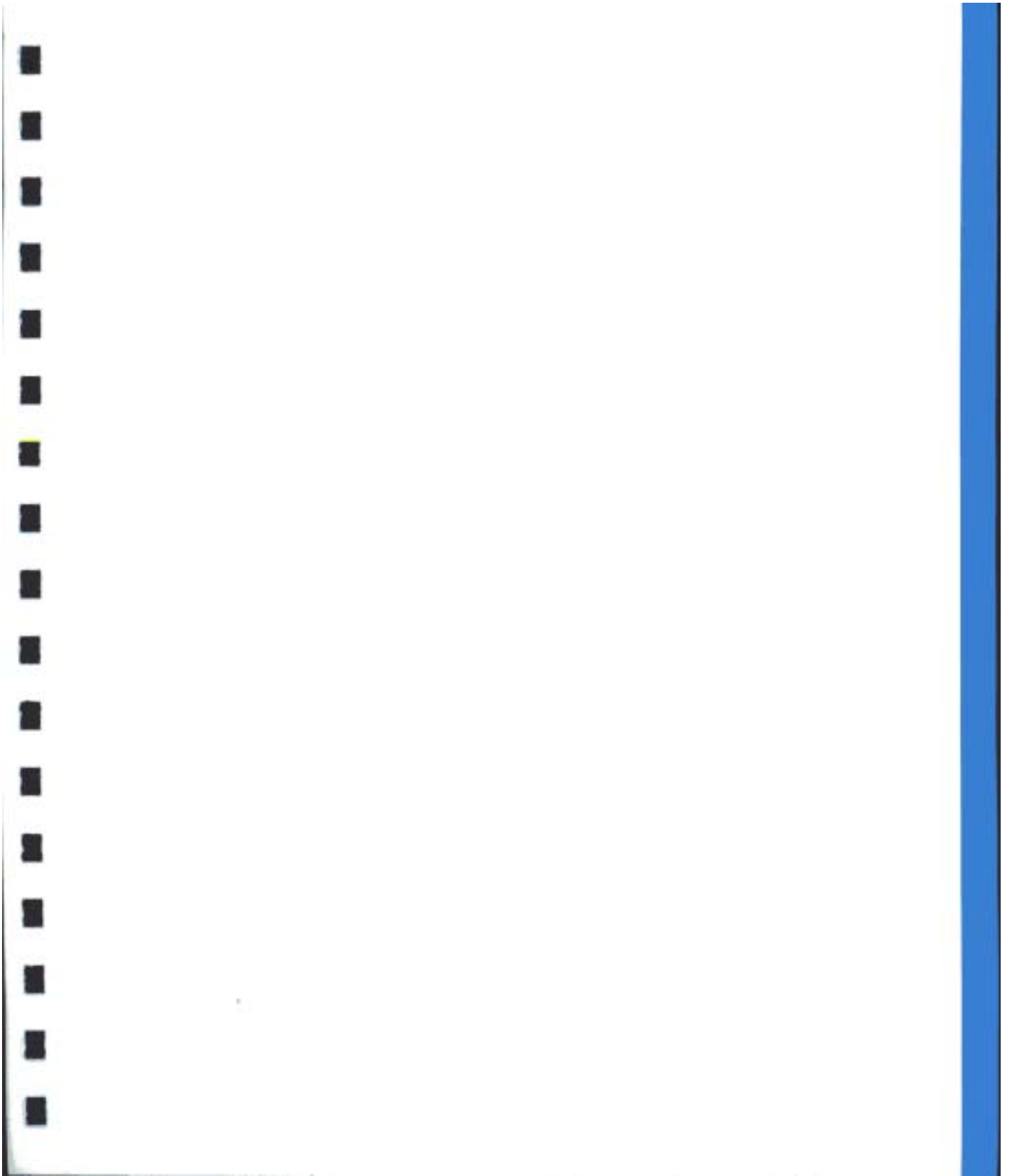
.QUIT

`.QUIT` (command)

The `.QUIT` command is a safe way to exit Logo. It ensures that all your files are closed and everything else is safe.

Appendixes





Appendix A	Messages	251
<hr/>		
Appendix B	Useful Tools	255
	255 Graphics Tools	
	255 ARCR and ARCL	
	256 CIRCLER and CIRCLEL	
	256 POLY	
	257 Math Tools	
	257 ABS	
	257 CONVERT	
	258 DIVISORP	
	258 LOG	
	258 LN	
	259 PWR	
	260 EXP	
	260 Program Logic or Debugging Tools	
	261 COMMENT	
	261 FOREVER	
	261 MAP	
	261 SORT and SUPERSORT	
	262 WHILE	
	262 Tools for the young Logo user	
	262 DRIVE	
	263 TEACH	

Appendix C	Startup Files	267
	267	Creating a Startup File
	268	A Note of Caution Before You Start
	268	The STARTUP Variable
Appendix D	Memory Space	271
	271	How Space Is Allocated
	272	Some Hints for Saving Space
Appendix E	Parsing	275
	275	Delimiters and Spacing
	276	Infix Procedures
	277	Brackets and Parentheses
	277	Quotation Marks and Delimiters
	278	The Minus Sign
Appendix F	ASCII Character Codes	281
Appendix G	Summary of Logo Primitives	285
Appendix H	Using a Printer With Logo	299
	300	The Software
	301	The Computer
	302	Serial Interfaces
	302	Parallel Interfaces
	303	The Printer

Messages



This appendix contains all the error messages you can get while using Logo. The words *file* and *name* (in lowercase letters) in this appendix are replaced with the specific word in question when the message is displayed.

Table A-1. Logo Messages

Number	Message
1	name IS ALREADY DEFINED
2	NUMBER TOO BIG
3	THE DISK WAS SWITCHED
6	name IS A PRIMITIVE
7	CAN'T FIND LABEL name
8	CAN'T name FROM THE EDITOR
9	name IS UNDEFINED
10	name DIDN'T OUTPUT TO name
11	I'M HAVING TROUBLE WITH THE DISK - number
12	DISK FULL
13	CAN'T DIVIDE BY ZERO
15	FILE file ALREADY EXISTS
16	FILE file PROTECTED
17	FILE file NOT FOUND
18	FILE file WRONG TYPE

Table A-1. Logo Messages (continued)

Number	Message
19	TOO FEW ITEMS IN name
20	TOO MANY FILES OPEN
21	CAN'T FIND CATCH FOR name
23	OUT OF SPACE
24	name CAN'T BE USED
25	name IS NOT TRUE OR FALSE
26	PAUSING...
27	YOU'RE AT TOPLEVEL
28	STOPPED!
29	NOT ENOUGH INPUTS TO name
30	TOO MANY INPUTS TO name
31	TOO MUCH INSIDE ()'s
33	CAN ONLY DO THAT IN A PROCEDURE
34	TURTLE OUT OF BOUNDS
35	I DON'T KNOW HOW TO name
36	name HAS NO VALUE
37	UNEXPECTED `)'
38	YOU DON'T SAY WHAT TO DO WITH name
40	DISK IS WRITE PROTECTED
41	name DOESN'T LIKE name AS INPUT
44	NO FILE SELECTED
45	FILE file NOT OPEN
46	FILE file ALREADY OPEN
47	FILE POSITION OUT OF RANGE
48	DEVICE UNAVAILABLE

Table A-1. Logo Messages (continued)

Number	Message
50	ALREADY DRIBBLING
52	DEVICE number IN USE
53	FILE file TOO BIG
54	VOLUME NOT FOUND FOR file
55	SUBDIRECTORY NOT FOUND FOR file
56	SUBDIRECTORY name NOT EMPTY
	!!! LOGO SYSTEM BUG !!!
	Should not occur. Please write to LCSI if it does.

Useful Tools



The procedures presented here are for your convenience when constructing your own procedures. Some of them were defined as examples for primitives and others appear here for the first time. These procedures are on the Logo disk in the file `TOOLS`.

Graphics Tools

See Appendix A of the Apple Logo II: An Introduction to Programming manual for other definitions of the procedures that draw arcs and circles.

You can use the procedures in this section to

- draw an arc that turns in a right or left direction (`ARCR` and `ARCL`)
- draw a circle that turns in a right or left direction (`CIRCLER` and `CIRCLEL`)
- draw a polygon (`POLY`).

ARCR and ARCL

`ARCR` and `ARCL` draw right and left turn arcs, respectively. Their inputs are

<code>:RADIUS</code>	the radius of the circle from which the arc is taken
<code>:DEGREES</code>	the degrees of the arc (the length of the edge)

```

TO ARCR :RADIUS :DEGREES
LOCAL "STEP LOCAL "REM
MAKE "STEP 2 * :RADIUS * 3.1416 / 36
MAKE "REM REMAINDER :DEGREES 10
REPEAT :DEGREES / 10 [RT 5 FD :STEP RT !
5]
IF :REM > 0 [FD :STEP * :REM / 10 RT :R!
EM]
END

```

```

TO ARCL :RADIUS :DEGREES
LOCAL "STEP LOCAL "REM
MAKE "STEP 2 * :RADIUS * 3.1416 / 36
MAKE "REM REMAINDER :DEGREES 10
REPEAT :DEGREES / 10 [LT 5 FD :STEP LT !
5]
IF :REM > 0 [FD :STEP * :REM / 10 LT :R!
EM]
END

```

CIRCLER and CIRCLEL

CIRCLER and CIRCLEL draw right and left turn circles with a specified radius as input.

```

TO CIRCLER :RADIUS
LOCAL "STEP
MAKE "STEP 2 * :RADIUS * 3.1416 / 36
REPEAT 36 [RT 5 FD :STEP RT 5]
END

```

```

TO CIRCLEL :RADIUS
LOCAL "STEP
MAKE "STEP 2 * :RADIUS * 3.1416 / 36
REPEAT 36 [LT 5 FD :STEP LT 5]
END

```

POLY

POLY draws a polygon over and over.

```

TO POLY :SIDE :ANGLE
FD :SIDE
RT :ANGLE
POLY :SIDE :ANGLE
END

```

Math Tools

You can use the procedures in this section to

- find the absolute value of a number (ABS)
- change a number from one base to another (CONVERT)
- find out if one number divides evenly into a second number (DIVISORP)
- calculate the logarithm to the base 10 of a number (LOG)
- calculate the natural logarithm of a number (LN)
- find the value of a number to a given power (PWR)
- use the natural exponential function (EXP).

ABS

ABS outputs the absolute value of its input.

```
TO ABS :NUM
OP IF :NUM < 0 [- :NUM] [:NUM]
END
```

CONVERT

CONVERT converts *N*, a number, from a base value (:FRBASE) to another base value (:TOBASE).

```
TO CONVERT :N :FRBASE :TOBASE
OP DEC.TO.ANYBASE ANYBASE.TO.DEC :N :FR!
BASE 1 :TOBASE
END
```

```
TO ANYBASE.TO.DEC :N :BASE :POWER
IF EMPTY? :N [OP 0]
OP (:POWER * C.TO.N LAST :N) + ANYBASE.!
TO.DEC BL :N :BASE :POWER * :BASE
END
```

```
TO DEC.TO.ANYBASE :N :BASE
IF :N < :BASE [OP N.TO.C :N]
OP WORD DEC.TO.ANYBASE INT QUOTIENT :N !
:BASE :BASE N.TO.C REMAINDER :N :BASE
END
```

```
TO C.TO.N :N
IF NUMBERP :N [OP :N]
OP (ASCII :N) - 55
END
```

```
TO N.TO.C :N
IF :N < 10 [OP :N]
OP CHAR 55 + :N
END
```

You can then use CONVERT to convert decimal to hexadecimal or hexadecimal to decimal.

```
TO DECTOHEX :N
OP CONVERT :N 10 16
END
```

```
TO HEXTODEC :N
OP CONVERT :N 16 10
END
```

DIVISORP

DIVISORP indicates (TRUE or FALSE) whether its first input divides evenly into its second.

```
TO DIVISORP :A :B
OP 0 = REMAINDER :B :A
END
```

LOG

LOG returns the logarithm to the base 10 of an input number. It uses the LN procedure, which follows.

```
TO LOG :X
OP 0.434294 * LN :X
END
```

LN

LN calculates the natural logarithm of an input number using all the following math procedures as subprocedures.

```
TO LN :X
LOCAL "RLIST
IF :X < 0 [OP [CAN'T DO LOG OF NEGATIVE!
NUMBERS]]
```



```

IF :X = 1 [OP 0]
IF :X < 1 [MAKE "RLIST ROOT (1 / :X) 1 !
-1] [MAKE "RLIST ROOT :X 1 1]
OP (FIRST BF :RLIST) * (LN1 FIRST :RLIS!
T) / (LAST :RLIST)
END

TO ROOT :X :NPWR :CONST
IF :X < 1.2 [OP (LIST :X :NPWR :CONST)]
OP ROOT (SQRT :X) (2 * :NPWR) :CONST
END

TO LN1 :X
MAKE "X (:X - 1) / (:X + 1)
OP 2 * (:X + (PWR :X 3) / 3 + ( PWR :X !
5) / 5)
END

```

PWR

PWR returns the value of A to the X power. If X is a fraction and A is not equal to one, PWR uses the natural functions EXP and LN. If A is less than 0 and X is a fraction, the result should be a complex number.

```

TO PWR :A :X
IF AND (:A < 0) NOT (:X = INT :X) [PR (!
SE :A [TO POWER] :X [IS A COMPLEX NUMBE!
R!]) STOP]
IF OR :A = 1 :X = INT :X [OP INTPWR :A !
:X]
OP EXP ((LN :A) * :X)
END

TO INTPWR :A :INTP
IF OR :A = 1 :INTP = 0 [OP 1]
IF :INTP < 0 [OP 1 / PWRLOOP (:A) (-:IN!
TP)]
OP PWRLOOP :A :INTP
END

TO PWRLOOP :A :INTP
IF :INTP = 0 [OP 1]
OP :A * PWRLOOP :A :INTP - 1
END

```

EXP

EXP is the natural exponential function, calculated using a Taylor series. *E* is declared a local variable to ensure that it always contains the correct value.

```
TO EXP :X
LOCAL "E
MAKE "E 2.71828
IF (:X - INT :X) = 0 [OP INTPWR :E :X]
OP (INTPWR :E INT :X) * (1 + EFRAC (:X !
- INT :X) 1 1)
END

TO EFRAC :FRAC :COUNT :TERM
IF :COUNT > 9 [OP 0]
MAKE "TERM :TERM * :FRAC / :COUNT
OP :TERM + EFRAC :FRAC :COUNT + 1 :TERM
END
```

Try this:

```
?PR PWR 2 3
8
?PR PWR 3 2
9
?PR PWR 3 0
1
?PR LN 50
3.91201
?PR LN 2.71828
0.999998
```

Program Logic or Debugging Tools

The procedures in this section let you

- embed comments in a program (COMMENT)
- repeat a group of instructions until you halt them (FOREVER)
- apply a command to every element of a list (MAP)
- sort a list of words and arrange them in a flat list (SORT and SUPERSORT)
- repeat a group of instructions until a specific condition becomes false (WHILE).

COMMENT

COMMENT allows you to embed comments in your programs in the form:

```
; [THIS IS A COMMENT]
TO ; :COMMENT
END
```

FOREVER

FOREVER repeats a group of instructions until you press (⏏)-ESC or turn off the power.

```
TO FOREVER :INSTRUCTIONLIST
RUN :INSTRUCTIONLIST
FOREVER :INSTRUCTIONLIST
END
```

MAP

MAP applies a command to every element of a list.

```
TO MAP :CMD :LIST
IF EMPTY? :LIST [STOP]
RUN LIST :CMD WORD "" FIRST :LIST
MAP :CMD BF :LIST
END
```

SORT

SORT takes a list of words and outputs them alphabetically.

```
TO SORT :ARG :LIST
IF EMPTY? :ARG [OP :LIST]
MAKE "LIST INSERT FIRST :ARG :LIST
OP SORT BF :ARG :LIST
END

TO INSERT :A :L
IF EMPTY? :L [OP ( LIST :A )]
IF BEFOREP :A FIRST :L [OP FPUT :A :L]
OP FPUT FIRST :L INSERT :A BF :L
END
```

Try this:

```
MAKE "SORTLIST SORT [A D E F T C Z] []  
PR :SORTLIST  
A C D E F T Z
```

Then type

```
MAKE "SORTLIST SORT [FOO BAR BAZ] :SORT!  
LIST  
PR :SORTLIST  
A BAR BAZ C D E F FOO T Z
```

WHILE

WHILE repeats a group of instructions until :CONDITION becomes false.

```
TO WHILE :CONDITION :INSTRUCTIONLIST  
TEST RUN :CONDITION  
IFFALSE [STOP]  
RUN :INSTRUCTIONLIST  
WHILE :CONDITION :INSTRUCTIONLIST  
END
```

Tools for the Young Logo User

You can use the procedures in this section to

- drive the turtle around the screen with the touch of a key (DRIVE)
- define a procedure as you are running it line by line (TEACH).

DRIVE

DRIVE lets you drive the turtle around the screen with the touch of a key. This is an example of single-keypress interactive programming.

```
TO DRIVE  
IF KEYP [LISTEN]  
FD 1  
DRIVE  
END
```

```

TO LISTEN
MAKE "ANS RC
IF :ANS = "S [THROW "TOPLEVEL]
IF :ANS = "R [RT 10]
IF :ANS = "L [LT 10]
END

```

TEACH

TEACH lets you define a procedure as you are running it line by line. By typing END, you finish defining the procedure. Entering ERASE removes the previous line from the definition in progress. This is especially useful when working with young children.

```

TO TEACH
LOCAL "THISLINE
DEFINE "PROGRAM [[]]
CLEARSCREEN
GETLINES
NAMEIT
END
END

TO GETLINES
TYPE "??
MAKE "THISLINE READLIST
IF :THISLINE = [END] [STOP]
IF :THISLINE = [ERASE] [WIPEOUT] [IF (FI!
RST :THISLINE) = "TO [ ] [RUNSTORE]]
GETLINES
END

TO WIPEOUT
DEFINE "PROGRAM BUTLAST TEXT "PROGRAM
CLEARSCREEN
RUN [PROGRAM]
END

TO RUNSTORE
CATCH "ERROR [RUN :THISLINE STORE STOP]
PRINT FIRST BUTFIRST ERROR
END

```

```
TO STORE
DEFINE "PROGRAM LPUT :THISLINE TEXT "PR!
OGRAM
END

TO NAMEIT
LOCAL "NAME
PRINT [WHAT SHOULD I CALL THIS?]
MAKE "NAME READLIST
IF EMPTY :NAME [ERASE "PROGRAM STOP]
IF DEFINEDP FIRST :NAME [TRYAGAIN] [COPI]
Y]
END

TO TRYAGAIN
PRINT SENTENCE FIRST :NAME [IS ALREADY !
DEFINED.]
PRINT []
NAMEIT
END

TO COPY
DEFINE FIRST :NAME TEXT "PROGRAM
PRINT SENTENCE FIRST :NAME [DEFINED]
ERASE "PROGRAM
END
```



Startup Files



This appendix describes the feature of Logo that lets you automatically load a file into your workspace when you start up Logo. You must call the file `STARTUP`. There can be only one file with the name `STARTUP`, although it can include commands to load other files. The disk with the `STARTUP` file must be in drive 1 when you press `(RETURN)` from the title display.

Creating a Startup File

Before placing a procedure in the `STARTUP` file, you must first enter the procedure into your workspace. You do so either by typing procedures in or by loading them from another file. For instance, you might want to transfer something from the `TOOLS` file into the new `STARTUP` file. To check your workspace, type `POTS`.

You see the list of procedures that you just added, whether by keyboard entry or from another file, and the procedures that were previously in your workspace. At this point, you can save the new file with the name `STARTUP`.

However, if some procedures are buried when a file is loaded, `POTS` does not show you their names, and you can't save or erase them. The reason for this is that the global workspace commands `SAVE`, `ERALL`, and `ERPS` don't erase buried procedures (that's the reason for burying them!). To use the `ERALL`, `ERPS`, or `SAVE` command successfully on buried procedures, they must first be unburied.

To see all the procedure names, including any buried procedures, type

```
UNBURYALL  
POTS
```

Erasing these procedures is the same as erasing others: just specify the procedure names in a list following the ERASE command. Saving them individually onto a disk is similar: just put the names you want to save in a list for the SAVEL command. Only those procedures will be saved, regardless of whether they're buried or not. However, all the unburied names will also be saved, so check the names in your workspace with PONS before using SAVEL.

A Note of Caution Before You Start

If you already have a STARTUP file and you are about to create a new one to use in its place, you run the risk of losing useful procedures. Even if you want to do this, you might like the old procedures back some time (when a newcomer is trying Logo, for example).

So, before proceeding, you may want to save your old STARTUP file on a disk by giving it the name OLDSTARTUP or something like that. To change the name of any file, use the RENAME command. In this case, type

```
RENAME "STARTUP "OLDSTARTUP
```

Having done that, type

```
SAVE "STARTUP
```

All the procedures you just saved will be loaded in your computer and will be ready to use after you press **RETURN** from the title display.

The STARTUP Variable

Logo has a special variable named STARTUP. Any file, including the STARTUP file, can contain a STARTUP variable. The first thing Logo does after loading a STARTUP file is to look for the STARTUP variable. If one exists, Logo runs the contents of the variable. The contents of the STARTUP variable must be a list.

If you load your STARTUP file in your workspace, type

```
MAKE "STARTUP [PR [GOOD MORNING]]
```

Logo saves the STARTUP variable and its contents when you tell it to save your new STARTUP file. Then, whenever you start up Logo, your computer will greet you with GOOD MORNING before saying WELCOME TO LOGO.

It's easier to use the EDITFILE command to edit a file and add a variable such as STARTUP. To add a STARTUP variable to a STARTUP file this way, type

```
EDITFILE "STARTUP
```

The entire file contents will appear in the Logo Editor. Move to the bottom of the file (where the variables are stored) and add a line like this:

```
MAKE "STARTUP [WELCOME]
```

Then move the cursor back up into the area where procedures are stored, begin a new line, and type something like this:

```
TO WELCOME  
LOCAL "ANSWER  
PR [Hello again, Eric!]  
TYPE [How are you today?]  
MAKE "ANSWER RW  
IF MEMBERP :ANSWER [FINE OK GREAT] [PR !  
[I'm happy to hear that] STOP]  
PR [Well, let's hope Logo-ing will help!  
]  
END
```

To summarize, Logo looks for a file called STARTUP on the disk in drive 1. If Logo finds the file, Logo loads it and then looks for a variable called STARTUP. If the variable exists, Logo runs its contents.

Memory Space



Logo procedures and variables take up space; more space is used when the procedures are run. This appendix tells you how Logo allocates memory space and how you can use less of that space.

In general, you need not worry about saving space. Instead you should try to write procedures as clearly and elegantly as possible. However, we recognize that Apple Logo has only a finite memory. For this reason, you might want to know how Logo manages its memory space.

How Space Is Allocated

Logo allocates space in **nodes**, each of which is five bytes long. All Logo objects and procedures are built out of nodes. Every Logo word used is stored only once: all occurrences of that word are actually pointers to the word.

Logo allocates nodes in this way:

- A literal word takes up one node for every two characters.
- A variable name and a procedure name each take up three nodes plus the size of the name.
- A property list takes up three nodes plus two nodes for each property plus the size of the property list itself.
- A number, whether integer or decimal, takes up one node.
- A list takes up one node for each element plus the size of the element itself.

The internal workings of Logo also use nodes. The interpreter knows about certain free nodes that are available for use. When there are no more free nodes, a special part of Logo called the garbage collector looks through all the nodes and reclaims any nodes that are not being used.

Example:

```
MAKE "NUMBER 7  
MAKE "NUMBER 90
```

When Logo executes `MAKE "NUMBER 7`, it assigns `NUMBER` to one node, which holds the value 7. After executing `MAKE "NUMBER 90`, Logo can reuse the nodes containing the 7. Logo will reclaim those nodes as free nodes the next time the garbage collector runs. The garbage collector runs automatically when necessary, but you can make it run with the Logo command `RECYCLE`.

For a description of the `RECYCLE` command, see Chapter 14.

The operation `NODES` outputs the number of free nodes; however, if you really want to find out how much space you have, you should do something like the following:

```
RECYCLE PRINT NODES  
1259
```

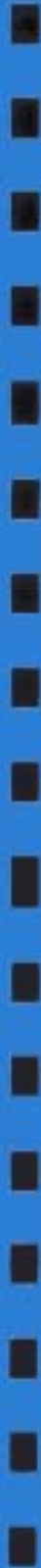
Some Hints for Saving Space

If you find that you are running out of space, you might want to rewrite your program so that it uses less space. Consider these programming tips:

- Use procedures to replace repetitive sections of the program.
- Avoid creating new words. To save space, you can use the names of inputs of one procedure as the names of inputs of other procedures. You can also use the names of procedures and primitives as variable names.
- Remember that it is bad form to try to save space by using short or obscure words in your procedures. Doing so may save space, but it makes the procedures less readable.



Parsing



This appendix will help you understand how Logo parses lines. **Parsing** works like this: when you type a line in Logo, Logo recognizes the characters as words and lists, and builds a list that is Logo's internal representation of the line. To see the parsing effect, type the line in a procedure definition with the command TO and use the Logo Editor to see the result.

Delimiters and Spacing

A word is usually delimited by spaces. This means that there is a space before the word and a space after the word; they set the word off from the rest of the line. There are more delimiting characters besides the space:

```
[ ] ( ) = < > + - *
```

You need not type a space between a word and any of these characters. For example, to find out how this line is parsed:

```
IF 1<2[PRINT(3+4)*5][PRINT :X+6]
type
?TO TESTIT
>IF 1<2[PRINT(3+4)*5][PRINT :X+6]
>END
?ED "TESTIT
```

The screen will look like this:

```
LOGO EDITOR
-----
TO TESTIT
IF 1 < 2 [PRINT ( 3 + 4 ) * 5] [PRINT :!
X + 6]
END
-----
ó-A accept, ó-? help, ó-ESC cancel
```

To treat any of the characters mentioned above as a normal alphabetic character, put a backslash (\) before it. For example:

```
?PRINT "GOOD\ -BYE
GOOD-BYE
?PRINT "SAN\ FRANCISCO
SAN FRANCISCO
```

Infix Procedures

The following characters are the names of infix procedures. You write the name between the two inputs, but Logo considers the procedures to have two inputs.

+ - * / = < >

Brackets and Parentheses

The left bracket ([]) and right bracket (]) characters indicate the start and end of a list or sublist.

Parentheses group things in ways Logo ordinarily would not, and vary the number of inputs for certain primitives.

If you reach the end of a Logo line—that is, you press **RETURN**—and brackets or parentheses are still open, Logo closes all sublists or expressions. For example:

```
?REPEAT 4 [PRINT [THIS [IS [A [TEST  
THIS [IS [A [TEST]  
THIS [IS [A [TEST]  
THIS [IS [A [TEST]  
THIS [IS [A [TEST]
```

If Logo finds a right bracket for which there was no corresponding left bracket, Logo stops execution of the rest of the line or procedure. For example:

```
?]PRINT "ABC  
?
```

Quotation Marks and Delimiters

Normally, you have to put a backslash (\) before the characters [,], (,), +, -, *, =, <, >, and \ itself. But the first character after a quotation mark (") does not need to have a backslash preceding it. For example:

```
?PRINT ""  
*
```

If a delimiter occupies any position but the first one after the quotation mark, it must have a backslash preceding it. For example:

```
?PRINT "*****
NOT ENOUGH INPUTS TO *
```

The only exception to the above general rule is brackets ([]). If you want to put a quotation mark before a bracket, you must always include a backslash between the quotation mark and the bracket. For example:

```
?PRINT "[
YOU DON'T SAY WHAT TO DO WITH [ ]
?PRINT "\[
[
```

The Minus Sign

The way in which Logo parses the minus sign (-) is an unusual case. The problem here is that the minus sign character is used to represent three different things:

- part of a number, to indicate that it is negative, as in -3
- a procedure of one input, called unary minus, which outputs the additive inverse of its input, as in -XCOR or -:DISTANCE
- a procedure of two inputs, which outputs the difference between its first input and its second, as in 7 - 3 and XCOR - YCOR.

The parser tries to be clever about this potential ambiguity and figures out which of the three uses is meant, using the following rules:

1. If the minus sign immediately precedes a number, and follows any delimiter (including a space) except right parenthesis, Logo parses the number as a negative number. This allows the following behavior:

```
PRINT 3 * -1           parses as 3 times negative 1
PRINT 3*-4             parses as 3 times negative 4
FIRST [- 3 4]         outputs -
FIRST [-3 4]          outputs -3
```

2. If the minus sign is preceded by a numeric expression, it works like an infix procedure. For example:

```
PR 3-4 is -1
```

```
PR XCOR - YCOR
```

The following are interpreted the same:

```
MAKE "A SE XCOR - YCOR 3
```

```
MAKE "A SE XCOR -YCOR 3
```

```
MAKE "A SE XCOR-YCOR 3
```

3. If the minus sign is not preceded by a numeric expression, it works like a unary minus. For example:

```
PR -XCOR
```

```
PR -(3+4)
```


ASCII Character Codes



This appendix contains a chart of American Standard Code for Information Interchange (ASCII) code values (in decimal) for all characters in Logo. Note that characters can be

- normal (white characters on black background)
- inverse video (black characters on white background).

Table F-1 shows the ASCII codes for normal characters; Table F-2 shows the ASCII codes for characters in inverse video.

To change a normal character to inverse, use the following procedure:

```
TO INVERSE :CHAR
  IF (ASCII :CHAR) > 127 [OP :CHAR]
  IF OR (ASCII :CHAR) < 64 AND (ASCII :CHAR!
) > 96 (ASCII :CHAR) < 128 [OP CHAR 128 +!
  ASCII :CHAR] [OP CHAR 64 + ASCII :CHAR]
END
```

Table F-1. ASCII Codes for Normal Characters

ASCII code	char	ASCII code	char	ASCII code	char	ASCII code	char
0	@	32	SPACE	64	@	96	`
1	A	33	!	65	A	97	a
2	B	34	"	66	B	98	b
3	C	35	#	67	C	99	c
4	D	36	\$	68	D	100	d
5	E	37	%	69	E	101	e
6	F	38	&	70	F	102	f
7	G	39	'	71	G	103	g
8	H	40	(72	H	104	h
9	I	41)	73	I	105	i
10	J	42	*	74	J	106	j
11	K	43	+	75	K	107	k
12	L	44	,	76	L	108	l
13	RETURN	45	-	77	M	109	m
14	N	46	.	78	N	110	n
15	O	47	/	79	O	111	o
16	P	48	0	80	P	112	p
17	Q	49	1	81	Q	113	q
18	R	50	2	82	R	114	r
19	S	51	3	83	S	115	s
20	T	52	4	84	T	116	t
21	U	53	5	85	U	117	u
22	V	54	6	86	V	118	v
23	W	55	7	87	W	119	w
24	X	56	8	88	X	120	x
25	Y	57	9	89	Y	121	y
26	Z	58	:	90	Z	122	z
27	[59	;	91	[123	{
28	\	60	{	92	\	124	.
29]	61	}	93]	125	}
30	^	62	?	94	^	126	~
31	_	63	?	95	_	127	Blot

Table F-2. ASCII Codes of Inverse Characters

ASCII code	char	ASCII code	char	ASCII code	Mouse Text	ASCII code	char
128	@	160	SPACE	192		224	'
129	A	161	!	193		225	a
130	B	162	"	194		226	b
131	C	163	#	195		227	c
132	D	164	\$	196		228	d
133	E	165	%	197		229	e
134	F	166	&	198		230	f
135	G	167	'	199		231	g
136	H	168	(200		232	h
137	I	169)	201		233	i
138	J	170	*	202		234	j
139	K	171	+	203		235	k
140	L	172	,	204		236	l
141	M	173	-	205		237	m
142	N	174	.	206		238	n
143	O	175	/	207		239	o
144	P	176	0	208		240	p
145	Q	177	1	209		241	q
146	R	178	2	210		242	r
147	S	179	3	211		243	s
148	T	180	4	212		244	t
149	U	181	5	213		245	u
150	V	182	6	214		246	v
151	W	183	7	215		247	w
152	X	184	8	216		248	x
153	Y	185	9	217		249	y
154	Z	186	:	218		250	z
155	[187	;	219		251	{
156	\	188	<	220		252	
157]	189	=	221		253	}
158	^	190	>	222		254	~
159	_	191	?	223		255	Blot

Summary of Logo Primitives



Parentheses around an input indicate that the input is optional. A number sign (#) indicates a procedure that can take any number of inputs; if you give it other than the number indicated, you must enclose the entire expression in parentheses.

ALLOPEN	Outputs a list of the files that are currently open.
#AND <i>pred1 pred2</i>	Outputs TRUE if all of its inputs are TRUE.
ARCTAN <i>number</i>	Outputs the arctangent of <i>number</i> in degrees.
ASCII <i>char</i>	Outputs the ASCII code for the character <i>char</i> .
.AUXDEPOSIT <i>loc byte</i>	Stores the value <i>byte</i> at address <i>loc</i> in the auxiliary bank.
.AUXEXAMINE <i>loc</i>	Outputs the value stored at <i>loc</i> in the auxiliary bank.
BACK, BK <i>distance</i>	Moves the turtle <i>distance</i> steps back.
BACKGROUND, BG	Outputs a number representing the background color.
BEFOREP <i>word1 word2</i>	Outputs TRUE if <i>word1</i> comes before <i>word2</i> according to the ASCII code.
.BLOAD <i>pathname loc</i>	Loads an assembly-language file into memory at <i>loc</i> .

<code>.BSAVE <i>pathname loc integer</i></code>	Saves memory region (starting at <i>loc</i> for <i>integer bytes</i>) into the file indicated by <i>pathname</i> .
<code>BURY <i>name(list)</i></code>	Buries all procedures contained in <i>name(list)</i> .
<code>BURYALL</code>	Buries all procedures and variables contained in the workspace.
<code>BURYNAME <i>name(list)</i></code>	Buries the variable name(s) contained in the <i>name(list)</i> .
<code>BUTFIRST, BF <i>obj</i></code>	Outputs all but the first element of its input.
<code>BUTLAST, BL <i>obj</i></code>	Outputs all but the last element of its input.
<code>BUTTONP <i>paddlenumber</i></code>	Outputs TRUE if the button on the indicated paddle is down, FALSE if it is up.
<code>.CALL <i>loc</i></code>	Calls the machine-language subroutine at address <i>loc</i> .
<code>CATALOG</code>	Displays the names of files in the current directory and the number of blocks used by each.
<code>CATCH <i>name list</i></code>	Runs <i>list</i> ; returns when <code>THROW <i>name</i></code> is run.
<code>CHAR <i>integer</i></code>	Outputs the character whose ASCII code is <i>integer</i> .
<code>CLEAN</code>	Erases the graphics screen without affecting the turtle.
<code>CLEARSCREEN, CS</code>	Erases the screen, moves the turtle to [0 0], and sets the heading to 0.
<code>CLEARTEXT, CT</code>	Clears the text portion of the screen.
<code>CLOSE <i>file</i></code>	Closes a currently opened file or device.

CLOSEALL	Closes all currently opened files and devices.
CO	Resumes a procedure after a pause.
.CONTENTS	Outputs a list of all names, procedure names, and other words in the workspace.
COPYDEF <i>name newname</i>	Copies the definition of <i>name</i> onto <i>newname</i> .
COS <i>degrees</i>	Outputs the cosine of <i>degrees</i> .
COUNT <i>obj</i>	Outputs the number of elements in its input.
CREATEDIR <i>pathname</i>	Creates a subdirectory named by the last element of <i>pathname</i> .
CURSOR	Outputs the position of the cursor.
DEFINE <i>name list</i>	Makes <i>list</i> the definition of <i>name</i> .
DEFINEDP <i>word</i>	Outputs TRUE if <i>word</i> is the name of a procedure.
.DEPOSIT <i>loc byte</i>	Stores the value <i>byte</i> at address <i>loc</i> .
DIFFERENCE <i>number1 number2</i>	Outputs <i>number2</i> subtracted from <i>number1</i> .
DOT [<i>xcor ycor</i>]	Puts a dot at the specified coordinates.
DOTP [<i>xcor ycor</i>]	Outputs TRUE if there is a dot on the screen at the specified coordinates.
DRIBBLE <i>file</i>	Sends a copy of whatever text is printed on the screen to the specified file or device.
EDIT, ED (<i>name(list)</i>)	Starts the Logo Editor (containing the named procedure(s)).

EDITFILE <i>pathname</i>	Starts the Logo Editor with the contents of the file indicated by <i>pathname</i> .
EDN <i>name(list)</i>	Starts the Logo Editor containing the named variable(s).
EDNS	Starts the Logo Editor containing all variables in the workspace.
EMPTY? <i>obj</i>	Outputs TRUE if <i>obj</i> is the empty list or the empty word.
EQUALP <i>obj1 obj2</i>	Outputs TRUE if its inputs are equal.
ERALL	Erases everything in the workspace.
ERASE, ER <i>name(list)</i>	Erases the named procedure(s).
ERASEFILE, ERF <i>pathname</i>	Erases the file indicated by <i>pathname</i> from the disk.
ERN <i>name(list)</i>	Erases the named variable(s).
ERNS	Erases the variables in the workspace.
ERPROPS	Erases all properties from the workspace.
ERPS	Erases all the procedures in the workspace.
ERROR	Outputs a four-element list of information about the most recent error.
.EXAMINE <i>loc</i>	Outputs the byte stored at address <i>loc</i> .
FENCE	Fences the turtle within the edges of the screen.
FILELEN <i>pathname</i>	Outputs the length in bytes of the file indicated by <i>pathname</i> .
FILEP <i>pathname</i>	Outputs TRUE if the file indicated exists.

FILL	Fills the shape enclosing the turtle with the current pen color. If the turtle is not enclosed, the background is filled.
FIRST <i>obj</i>	Outputs the first element of its input.
FORM <i>number field precision</i>	Outputs <i>number</i> in <i>field</i> spaces with <i>precision</i> digits after the decimal point.
FORWARD, FD <i>distance</i>	Moves the turtle <i>distance</i> steps forward.
FPUT <i>obj list</i>	Outputs a list formed by putting its first input in front of <i>list</i> .
FULLSCREEN, FS	Devotes the entire screen to graphics. Same as CONTROL-L .
GO <i>word</i>	Transfers control to LABEL <i>word</i> .
GPROP <i>name prop</i>	Outputs <i>prop</i> property of <i>name</i> .
HEADING	Outputs the turtle's heading (its direction) in degrees.
HELP <i>word</i>	Prints the inputs for the primitive or procedure indicated.
HIDETURTLE, HT	Makes the turtle invisible.
HOME	Moves the turtle to [0 0] and sets the heading to 0.
IF <i>pred list1 (list2)</i>	If <i>pred</i> is TRUE, runs <i>list1</i> ; otherwise, runs <i>list2</i> .
IFFALSE, IFF <i>list</i>	Runs <i>list</i> if the most recent TEST was FALSE. If no test has been made, the list is not run.

IFTRUE, IFT <i>list</i>	Runs <i>list</i> if the most recent TEST was TRUE. If no test has been made, the list is not run.
INT <i>number</i>	Outputs the integer portion of <i>number</i> .
INTQUOTIENT <i>integer1</i> <i>integer2</i>	Outputs <i>integer1</i> divided by <i>integer2</i> , truncated to an integer.
ITEM <i>integer obj</i>	Outputs the element whose position in <i>obj</i> is <i>integer</i> .
KEYP	Outputs TRUE if a key has been pressed but not yet read.
LABEL <i>word</i>	Creates a labeled line for use by GO.
LAST <i>obj</i>	Outputs the last element of its input.
LEFT, LT <i>degrees</i>	Turns the turtle <i>degrees</i> left (counterclockwise).
#LIST <i>obj1 obj2</i>	Outputs a list of its inputs, preserving their list structure.
LISTP <i>obj</i>	Outputs TRUE if <i>obj</i> is a list.
LOAD <i>pathname</i>	Loads the file indicated into the workspace.
LOADHELP <i>pathname</i>	Loads the file named into the helpscreen area of memory so it will appear when (G)-(?) is pressed.
LOADPIC <i>pathname</i>	Loads the screen image in the file indicated directly onto the screen.
LOCAL <i>name(list)</i>	Makes <i>name(list)</i> local.
LOWERCASE <i>word</i>	Outputs <i>word</i> in all lowercase letters.
LPUT <i>obj list</i>	Outputs a list formed by putting its first input after <i>list</i> .
MAKE <i>name obj</i>	Gives the value <i>obj</i> to the variable <i>name</i> .

MEMBER <i>obj1 obj2</i>	Outputs the part of <i>obj2</i> that starts with <i>obj1</i> .
MEMBERP <i>obj1 obj2</i>	Outputs TRUE if its first input is an element of its second input.
NAME <i>obj name</i>	Makes <i>obj</i> the value of <i>name</i> .
NAMEP <i>word</i>	Outputs TRUE if <i>word</i> has a value.
NODES	Outputs the number of free nodes.
NODRIBBLE	Closes a dribble file.
NOT <i>pred</i>	Outputs TRUE if <i>pred</i> is FALSE.
NUMBERP <i>obj</i>	Outputs TRUE if <i>obj</i> is a number.
ONLINE	Lists the disk volumes on line.
OPEN <i>file</i>	Opens <i>file</i> so it can send or receive characters.
#OR <i>pred1 pred2</i>	Outputs TRUE if any of its inputs are TRUE.
OUTPUT, OP <i>obj</i>	Returns control to the calling procedure, with <i>obj</i> as output.
PADDLE <i>paddlenumber</i>	Outputs the rotation of the dial on the indicated paddle.
PARSE <i>word</i>	Outputs a list obtained from parsing <i>word</i> .
PAUSE	Makes a procedure pause.
PEN	Outputs the pen state (PD, PU, PE, PX).
PENCOLOR, PC	Outputs a number representing the pen color.
PENDOWN, PD	Puts the pen down.
PENERASE, PE	Puts the eraser down.
PENREVERSE, PX	Puts the reversing pen down.

PENUP, PU	Raises the pen.
PLIST <i>name</i>	Outputs the property list of <i>name</i> .
PO <i>name(list)</i>	Prints definitions of the named procedure(s).
POALL	Prints definitions of all procedures and variables in the workspace.
POFILE <i>pathname</i>	Prints out the contents of the file indicated.
PON <i>name(list)</i>	Prints the name(s) and value(s) of the variable(s) listed.
PONS	Prints the names and values of all unburied variables in the workspace.
POPS	Prints definitions of all unburied procedures in the workspace.
POS	Outputs the position of the turtle in coordinates.
POT <i>name(list)</i>	Prints the title line(s) of the named procedure(s).
POTS	Prints the title lines of all unburied procedures in the workspace.
PPROP <i>name prop obj</i>	Gives <i>name</i> the property <i>prop</i> with the value <i>obj</i> .
PPS	Prints property list(s) of everything in the workspace.
PREFIX	Outputs the current ProDOS prefix, most recently set with SETPREFIX.
PRIMITIVEP <i>word</i>	Outputs TRUE if <i>word</i> is a primitive.

#PRINT, PR <i>obj</i>	Prints its input followed by a carriage return and linefeed (strips off the outer brackets of lists).
PRINTPIC <i>integer</i>	Prints the graphics screen to the printer in <i>integer</i> slot or port.
#PRODUCT <i>number1</i> <i>number2</i>	Outputs the product of its inputs.
.QUIT	Quits Logo and releases control to ProDOS.
QUOTIENT <i>number1 number2</i>	Outputs <i>number1</i> divided by <i>number2</i> . The result is a decimal number.
RANDOM <i>integer</i>	Outputs a random non-negative integer less than <i>integer</i> .
READCHAR, RC	Outputs the character read from the current file or device (default is the keyboard). Waits for input, if necessary.
READCHARS, RCS <i>integer</i>	Outputs <i>integer</i> characters read from the current file or device (default is the keyboard). Waits for input, if necessary.
READER	Outputs the current file opened for reading.
READLIST, RL	Outputs the line read from the current file or device (default is the keyboard). Waits for input, if necessary.
READPOS	Outputs the file position of the current file being read.
READWORD, RW	Outputs the line read by the current device (default is the keyboard) after a carriage return.
RECYCLE	Performs a garbage collection.

REMAINDER <i>integer1 integer2</i>	Outputs the remainder of <i>integer1</i> divided by <i>integer2</i> .
REMPROP <i>name prop</i>	Removes property <i>prop</i> from the property list of <i>name</i> .
RENAME <i>pathname newpathname</i>	Renames <i>pathname</i> to <i>newpathname</i> (both files must be closed).
REPEAT <i>integer list</i>	Runs <i>list</i> <i>integer</i> times.
RERANDOM	Makes RANDOM behave reproducibly.
RIGHT, RT <i>degrees</i>	Turns the turtle <i>degrees</i> right (clockwise).
ROUND <i>number</i>	Outputs <i>number</i> rounded off to the nearest integer.
RUN <i>list</i>	Runs <i>list</i> ; outputs what <i>list</i> outputs.
SAVE <i>pathname</i>	Writes the whole workspace onto the file indicated by <i>pathname</i> .
SAVEL <i>name(list) pathname</i>	Saves the named procedures and any unburied variables in the indicated file.
SAVEPIC <i>pathname</i>	Saves the picture on the screen in the file indicated.
.SCRUNCH	Outputs the current aspect ratio of the screen.
#SENTENCE, SE <i>obj1 obj2</i>	Outputs a list of its inputs.
SETBG <i>colornumber</i>	Sets the background to the color represented by <i>colornumber</i> .
SETCURSOR [<i>column linenum</i>]	Puts the cursor at the position specified by [<i>column linenum</i>].
SETHEADING, SETH <i>degrees</i>	Sets the turtle's heading to <i>degrees</i> .
SETPC <i>colornumber</i>	Sets the pen color to <i>colornumber</i> .

SETPOS [<i>xcor ycor</i>]	Moves the turtle to the coordinates specified.
SETPREFIX <i>pathname</i>	Sets the ProDOS prefix.
SETREAD <i>file</i>	Sets the file from which the output of RC, RCS, RL, and RW will be read.
SETREADPOS <i>integer</i>	Sets the file position for reading the current file.
.SETSCRUNCH <i>number</i>	Sets the aspect ratio of the screen to <i>number</i> .
SETWIDTH <i>width</i>	Sets the screen width to <i>width</i> , either 40 or 80 columns.
SETWRITE <i>file</i>	Sets the destination of inputs to PRINT, TYPE, SHOW.
SETWRITEPOS <i>integer</i>	Sets the file position for writing into the current file.
SETX <i>xcor</i>	Moves the turtle horizontally so that the x-coordinate is <i>xcor</i> .
SETY <i>ycor</i>	Moves the turtle vertically so that the y-coordinate is <i>ycor</i> .
SHOW <i>obj</i>	Prints its input followed by a carriage return (with brackets for lists).
SHOWNP	Outputs TRUE if the turtle is shown.
SHOWTURTLE, ST	Makes the turtle visible.
SIN <i>degrees</i>	Outputs the sine of <i>degrees</i> .
SPLITSCREEN, SS	Allows text and graphics on the same screen. Same as CONTROL-S .
SQRT <i>number</i>	Outputs the square root of <i>number</i> .
STEP <i>name(list)</i>	Causes the procedure(s) to execute one line at a time.

STOP	Stops the procedure and returns control to the caller.
#SUM <i>number1 number2</i>	Outputs the sum of its inputs.
TEST <i>pred</i>	Determines whether <i>pred</i> is TRUE or FALSE.
TEXT <i>name</i>	Outputs the definition of procedure <i>name</i> as a list.
TEXTSCREEN, TS	Devotes the entire screen to text. Same as <code>(CONTROL)-T</code> .
THING <i>name</i>	Outputs the value of <i>name</i> .
THROW <i>name</i>	Transfers control to the corresponding CATCH.
TO <i>name (inputs)</i>	Begins the definition of <i>name</i> .
TOOT <i>frequency duration</i>	Produces a sound of <i>frequency</i> for <i>duration</i> .
TOWARDS [<i>xcor ycor</i>]	Outputs the heading the turtle would have if facing the coordinates specified.
TRACE <i>name(list)</i>	Causes tracing information to be printed for traced procedure(s).
#TYPE <i>obj</i>	Prints its input (strips off the outer brackets of lists).
UNBURY <i>name(list)</i>	Unburies the procedure(s) in <i>name(list)</i> .
UNBURYALL	Unburies all the procedures and variables buried in the workspace.
UNBURYNAME <i>name(list)</i>	Unburies the variable name(s) in <i>name(list)</i> .
UNSTEP <i>name(list)</i>	Ends the stepping of named procedure(s).
UNTRACE <i>name(list)</i>	Ends the tracing of named procedure(s).
UPPERCASE <i>word</i>	Outputs <i>word</i> in all uppercase letters.

WAIT <i>integer</i>	Pauses for approximately <i>integer</i> 60ths of a second.
WIDTH	Gives the current setting of the screen width, either 40 or 80 characters wide.
WINDOW	Makes the turtle field unbounded.
#WORD <i>word1 word2</i>	Outputs a word made up of its inputs.
WORDP <i>obj</i>	Outputs TRUE if <i>obj</i> is a word.
WRAP	Makes the turtle field wrap around the edges of the screen.
WRITEPOS	Outputs the file position of the current file being written to.
WRITER	Outputs the current file open for writing.
XCOR	Outputs the x-coordinate of the turtle.
YCOR	Outputs the y-coordinate of the turtle.
<i>number1 + number2</i>	Outputs <i>number1</i> plus <i>number2</i> .
<i>number1 - number2</i>	Outputs <i>number1</i> minus <i>number2</i> .
<i>number1 * number2</i>	Outputs <i>number1</i> times <i>number2</i> .
<i>number1 / number2</i>	Outputs <i>number1</i> divided by <i>number2</i> .
<i>number1 < number2</i>	Outputs TRUE if <i>number1</i> is less than <i>number2</i> .
<i>obj1 = obj2</i>	Outputs TRUE if <i>obj1</i> is equal to <i>obj2</i> .
<i>number1 > number2</i>	Outputs TRUE if <i>number1</i> is greater than <i>number2</i> .

Using a Printer With Logo



Here are some notes to help you get your printer working properly with Logo. If you are successfully using your printer from Logo, then you don't need to read any further.

If you are having printing problems, there are generally only three areas that you need to check to identify and correct the problem:

- the software—your program
- the computer's configuration, including its interface card or built-in port
- the printer's configuration, including its connecting cable.

Table H-1 gives common symptoms of printer problems and possible causes for each of them.

Table H-1. *Printer Problems and Causes*

Problem	Possible Cause (See Section)
No printing at all	<p>Software (programming) error (The Software)</p> <p>Computer or interface card incorrectly configured or installed (The Computer)</p> <p>Printer incorrectly set up or configured (The Printer)</p>

Table H-1. Printer Problems and Causes (Continued)

Problem	Possible Cause (See Section)
Incorrect printing	Computer or interface card incorrectly configured (The Computer)
	Wrong interface cable (The Computer)
	Printer incorrectly configured (The Printer)

Identify the type of error that you are observing, then go to the appropriate sections of this appendix to find more information and suggestions for fixing the problems.

If you follow all the suggestions and none of them turns out to be the cause of the problem, there may be something wrong with the equipment. In this case, take the printer and computer to your dealer to be thoroughly checked out and repaired, if necessary.

The Software

For more information, see Chapter 16.

If you can use your printer successfully with programs or languages other than Logo, it is likely that the problem lies with your Logo program. Logo treats all input and output operations as files. This means that before you can send information to the printer (referred to by the slot or port that it is connected to) you must open it for use and then select it as the current writer.

Assuming that your printer is connected to slot or port 1, this program will send text to the printer:

```
OPEN 1 SETWRITE 1
PR [THIS IS A TEST:]
PR [IF IT WORKS, SEND OUT FOR PIZZA!]
CLOSE 1 SETWRITE []
```

OPEN 1 opens slot or port 1 for use, while SETWRITE 1 selects slot or port 1 as the current writer. Any PRINT, TYPE, or SHOW statements after this prints to the current writer, now the printer. The last line of the program closes the printer file and resets the current writer back to the screen.

Note: If your printer is connected to a different slot, use that slot number instead of the 1's used in this program. When you finish printing, you must close the printer file and reset the current writer to the screen.

Remember that while up to six files can be open for use at one time, only *one* of these can be a slot or port.

The Computer

Refer to your serial card's manual for specific configuration information.

Start your hardware checks with the computer and the printer interface card.

Logo treats the printer interface in the same way that Apple II Pascal version 1.1 does. Any card that does not conform to the Apple II Pascal protocol, such as the Apple II Parallel Interface Card, cannot readily be made to work with Logo. If you have an Apple II Parallel Interface Card, see your dealer for help in making it work with Logo. If you have any questions about another interface card, refer them to that card's manufacturer.

If you're using an Apple IIe, make sure that the interface card is properly plugged into one of the computer's slots, usually slot 1. If you're using an Apple IIc, you must connect the printer to serial port 1.

If you have a serial printer such as the Apple Imagewriter, read the section "Serial Interfaces." If you have a parallel printer such as the Apple Dot Matrix Printer, skip to the section "Parallel Interfaces."

Refer to both the printer's and interface card's reference manuals to find out how to set them up and to set their respective configurations.

Serial Interfaces

A serial interface is primarily defined by the following characteristics:

- Data rate—how fast the information flows, measured in baud
- Data format—how the information is organized for transmission: the number of bits per character, parity scheme, and number of stop bits
- Other things affecting the printer's operation include whether or not: output is echoed to the screen, line feeds are appended to the ends of lines of text, and transmitted text is broken into lines of a given length.

When you turn on an Apple IIc, serial port 1 is automatically configured to match the factory-set configuration of the Apple Imagewriter printer:

- 9600 baud data rate
- 8-bit, no parity, two stop-bit data format
- No auto line feed

If you have an Apple IIe, you normally set your serial interface card to the same configuration as that of the Apple IIc's serial port 1.

If your interface card can't operate as fast as 9600 baud, set it to run at its fastest rate and change the printer's configuration to match the interface card's.

Now you can test your printer by running the program given in section "The Software." If your printer still doesn't work, skip to section "The Printer."

Parallel Interfaces

If you have an Apple IIc, this section doesn't apply.

Make sure that the interface card is correctly plugged in. Connect the printer interface cable to the card and then to the printer as described in the interface card manual. Check the interface card's switches, if any, and set them as described in its reference manual to match your printer's configuration.

The Printer

See Chapter 16 for more about PRINTPIC.

Make sure the printer is properly plugged into both the wall power socket and the printer interface cable. After setting any configuration switches as required to match the configuration of the interface being used, you are ready to test the printer.

Your printer may print text properly but not print graphics when using PRINTPIC. To print graphics, PRINTPIC needs an Apple Imagewriter, an Apple Dot Matrix Printer, or a compatible printer and an interface card such as the Super Serial Card, for example, whose firmware follows the conventions used by the Apple IIc's serial port 1. If you have an Apple Dot Matrix Printer and an Apple II Parallel Interface card, see your dealer to get the printer to work with Logo.

Now turn on your Apple II and the printer. Try to print some text using the test program in section "The Software." If nothing happens, check the following items:

- Has the printer run out of paper? Is the printer cover on correctly? Is the printer's ribbon installed correctly, or is the printer at the end of the ribbon?
- Is the printer on-line and selected? Some printers are set off-line, or deselected, when you replace paper or ribbons or advance the paper. After finishing one of these operations, the printer must be set back on-line, or selected (usually by pressing a button on the front panel), before you can continue printing.
- Are all interface and power connections properly set up? Is the printer's fuse blown?
- Are all configuration switches on the interface card and the printer set for the same values? Refer to the respective devices' reference manuals for the switch settings.
- Does the interface card have a configuration block? Is it the correct configuration block? Has it been installed correctly? Could the interface cable have been installed upside-down?
- Have you checked all the items listed above? If there is still no printing, see your dealer.

If the printer outputs gibberish or just "hiccoughs," check the data rate and data format settings of the interface and printer. Make sure that they match. Make sure that you have the proper interface cable.

If text is being over-printed, set the printer to generate a line feed character after each line. If text is always double-spaced, reset the printer to *not* generate a line feed after a carriage return.

Unexpected typefaces, such as double-width or very small characters, are probably caused by incorrect printer switch settings.

For any remaining problems, refer to the trouble-shooting section of your printer's reference manual.





Glossary



address: The location of a register, a particular part of memory, or some other data source or destination.

Americal Standard Code for Information Interchange (ASCII): The standard code used for exchanging information about data processing systems and associated equipment.

ASCII: See **Americal Standard Code for Information Interchange**.

ASCII file: A text file whose characters are represented in ASCII codes.

aspect ratio: A decimal number that is the ratio of the size of a vertical turtle step to the size of a horizontal one.

binary: Something that has two possible values or states. Also refers to the base 2 numbering system.

bit: A binary digit.

boot: The process of loading a language or application program into the computer's memory as in when you start up Logo.

buffer: An area of memory for temporary storage of data, used when transferring data from one device to another. Buffer usually refers to an area reserved for an input/output operation, into which data is read or from which data is written.

bug: An error in a program.

byte: Eight bits.

call: To bring a computer program, a procedure, or a subprocedure into effect.

character: A letter, digit, or other symbol that is used as part of the organization, control, or representation of data.

command: A Logo procedure, either a primitive or one that you define, that has no output. CLEARSCREEN, FORWARD, and PRINT are examples of commands. See **operation**.

conditional: A statement that causes Logo to carry out different instructions, depending on whether a condition is met.

cursor: A movable marker that is used to indicate a position on the display screen.

debug: To find and eliminate mistakes in a program.

default: A value or option that is provided by the program when none is specified.

device: Anything attached to the computer, such as a printer, video display, or disk drive.

directory: A table on a disk of the names of all the files on that disk, along with information that tells ProDOS where to find the files on the disk.

echo: To reflect received data to the sender. For example, keys pressed on the keyboard are usually echoed as characters displayed on the screen.

edit: To enter, modify, or delete data.

edit buffer: The portion of the computer's memory that contains all the text that is in the Logo Editor.

element: A member of a set; in particular, an item in a series.

empty list: A list that has no elements. You write the empty list as [].

empty word: A word that has no characters. You write the empty word as " ".

erase: To remove information permanently from either the workspace or a file.

execute: To perform an instruction or a computer program.

file: An organized collection of information that can be permanently stored for specific purposes.

format: The particular arrangement or layout of data on a data medium, such as the screen or a disk.

garbage collection: Cleaning the computer's memory to make more space available for storage.

global variable: A variable that is always in the workspace, such as a variable you create with the MAKE primitive. See **local variable**.

infix notation: A way of expressing an arithmetic operation where the operation symbol is placed between the two numerical inputs. See **prefix notation**.

input: The information that a Logo primitive or procedure needs to begin execution.

instruction: In a programming language, any meaningful expression that specifies one command and its inputs.

integer: A positive or negative number that does not contain any fractional parts.

interactive: A program that creates a dialogue between the computer and the user.

K: When referring to storage capacity, two to the tenth power or 1024 in decimal notation.

list: A collection of Logo objects, a sequence of words or lists that begins and ends with brackets.

literal word: An explicit representation of a value, especially the value of a word or list. A literal word is preceded by the quotation mark character (").

local variable: A variable that exists only when a procedure is being executed. See **global variable**.

location: Any place in which data may be stored.

logical operation: A predicate whose input must be either TRUE or FALSE.

name: A word used as a container for a value in the workspace.

node: A division of your workspace. Each node is five bytes long.

object: A word or a list.

operation: A Logo procedure, either a primitive or one that you define, that has some kind of output. SUM, ONLINE, POS are examples of operations. See **command**.

output: The information that a Logo primitive or procedure gives to another primitive or procedure.

parse: The process by which phrases are associated with the component names of the grammar that generated the string. In Logo, to make sense out of a Logo line.

pathname: The name that indicates the location of a file on a disk. A pathname consists of a device name, a subdirectory name or names, and the name of the file itself.

picture element (PIXEL): A graphics *point*. Also, the bits that contain the information for that point.

predicate: A procedure that outputs either TRUE or FALSE.

prefix: A pathname of a directory or subdirectory that is automatically placed in front of a filename that does not begin with a slash.

prefix notation: A way of expressing an arithmetic operation where the operation symbol or primitive is placed before the numerical inputs. See **infix notation**.

primitive: A procedure that is built into Logo.

procedure: A single instruction or a sequence of instructions to Logo, which has a name and can be permanently stored.

procedure call: A request to execute a named procedure. You call a procedure either from the top level or from within another procedure.

ProDOS: The Apple IIe and Apple IIc operating system under which Logo runs.

program: A set of procedures that work together.

prompt: A question the computer asks or a signal it displays when it wants you to supply information.

property list: A list consisting of an even number of elements. Each pair of elements consists of a property (such as I.D.) and its value, a word or list (such as Robin).

read: To input data into a device so that you can have access to it.

real number: Any positive or negative decimal number.

recursive procedure: A procedure that calls itself as a subprocedure. For example:

```
TO FLIP
:
:
:
FLIP
END
```

scientific notation: The expression of numbers using an exponent.

scroll: To move all or part of the display image vertically or horizontally so that new data appears at one edge as old data disappears at the opposite edge.

stack: A method of temporarily storing data so that the last item stored is the first item to be processed.

storage: A device, or part of a device, that can retain data.

string: A sequence of characters.

subdirectory: A group of logically related files on the same disk.

subprocedure: A procedure used in the definition of another procedure. For example:

```
TO A
  B
END
```

A calls B so B is a subprocedure of A.

superprocedure: A procedure that calls another procedure. For example:

```
TO A
  B
END
```

A calls B so A is a superprocedure of B.

syntax: The rules governing the structure of a language.

top level: The mode in which commands can be executed directly without being embedded in a program.

truncate: To remove the ending elements from a word. For a number, to remove the fractional part.

turtle: The shape on the screen that represents the pen Logo uses to draw lines.

value: The contents of a variable.

variable: A container that holds a value and has a name.

volume: A formatted disk. The volume name is also the name of the top level directory.

word: A series of characters treated as a unit.

workspace: The part of the computer's memory that holds variables, procedures, and properties only as long as the computer is turned on.

write: To record data on a data medium.



Index



BEFOREP operation 82
 BF (BUTFIRST) operation 69-70
 BG (BACKGROUND) operation 54
 BK (BACK) command 36
 BL (BUTLAST) operation 71
 BLOAD command 242
 blocks 190
 brackets ([]) 13, 68
 parsing of 277
 .BSAVE command 242
 buffer 26
 edit 28, 241
 file 241
 graphics 241
 kill 27
 BURY command 182
 bury facility 175
 BURYALL command 183
 BURYNAME command 183
 BUTFIRST (BF) operation 69-70
 BUTLAST (BL) operation 71
 BUTTONP operation 163
 BYE procedure 212

C

C.TO.N procedure 258
 CALCULATOR procedure 138
 .CALL command 242
 CANCEL procedure 149
 CATALOG command 194
 CATCH command 133, 136, 140
 CHAR operation 83
 characters
 ASCII codes for 281
 deleting 6
 reading 165, 166
 CHECKPOS procedure 220
 CHECKHEAD procedure 210
 CHECKWRITE procedure 221
 CIRCLE procedure 244
 CIRCLEL procedure 256
 CIRCLES procedure 256
 CLEAN command 47
 cleaning the workspace 182-185
 CLEARSCREEN (CS) command 37
 CLEARTEXT (CT) command 60
 CLOSE command 212
 CLOSEALL command 213
 CO command 130
 colon (:) 14, 15
 color
 background 51, 54
 pen 53, 55
 COMFORT procedure 158
 commands and operations 14
 COMMENT procedure 71, 261
 conditionals 125, 126-129
 .CONTENTS operation 245
 continuation lines 17, 27
 control characters 144
 interrupting procedures with 144
 changing screen use with 63
 CONTROL-L 63
 CONTROL-S 64
 CONTROL-T 64
 CONTROL-W 144
 CONTROL-Z 144
 CONVERT procedure 84, 257
 coordinates, x and y 41, 45
 COPY procedure 198, 264
 COPYDEF command 147-148
 COS operation 108
 cosine 108
 COUNT operation 85
 COUNTDOWN procedure 132, 137
 COUNTUP procedure 142
 CREATEDIR command 195

CS (CLEARSCREEN)
 command 37
CT (CLEARTEXT) command
 60
CUBE procedure 112
CURSOR operation 60
cursor movement 5, 29

D

D6 procedure 113
data files 189
 closing 212, 213
 opening 211, 214
 reading from 211
 sample project 221
 working with 211-221
 writing to 211
debugging programs 140-144
DEC.TO.ANYBASE procedure
 257
DECIDE procedure 126
DECIMALP procedure 158
decimals 105
DECTOHEX procedure 258
DEFINE command 147, 148
DEFINP operation 147, 150
defining procedure 11, 21-22
deleting
 characters 6
 lines 6
 text 30
delimiters, parsing of 275,
 277
.DEPOSIT command 242
device(s) 205
 closing 212, 213
 opening 211, 214
DICE procedure 115
DIFFERENCE operation 109
directory 190
 listing 194
 prefix 193

disk(s)
 formatting 190
 organization 190
 volume directory 190
 volume name 190
DISTANCE procedure 117
division 106
 with INTQUOTIENT
 operation 112
 with QUOTIENT operation
 113
division sign (/) 121
DIVISORP procedure 114, 258
DOIT procedure 135
DOIT1 procedure 135
\$STEP procedure 152
\$UNSTEP procedure 152
DOT command 47
DOTP operation 54
DRIBBLE command 209
dribble files 189
 working with 209-211
DRIVE procedure 262
DUMP procedure 210

E

ED (EDIT command 28
edit buffer 26, 28, 196, 241
EDIT command 28
EDITFILE command 31, 196
editing in the Editor 29
editing procedures 28
Editor 25
 editing in the 29
 getting out of 31
 help 4
 how it works 26
 keystrokes
 cursor movement 29
 deleting and inserting text
 30
 starting up 28, 31
 typing in the 29

EDN command 31, 96
 EDNS command 31, 97
 EFRAC procedure 260
 ELLIPSE procedure 244
 empty list 69
 empty word 68
 EMPTYP operation 85
 END (special word) 21, 22
 equal sign (=) 122
 EQUALP operation 87, 106
 equipment you must have 3
 ER (ERASE) command 181
 ERALL command 181
 ERASE (ER) command 181
 ERASEFILE (ERF) command 196
 erasing from the workspace 180-182
 ERF (ERASEFILE) command 196
 ERN command 181
 ERNS command 181
 ERPROPS command 230
 ERPS command 181
 error messages 135, 251
 ERROR operation 135
 EVENP procedure 114
 .EXAMINE operation 243
 examining words and lists 81
 exclamation mark (!) 17, 27
 executing procedures 12
 EXP procedure 260

F

FACTORIAL procedure 121
 FALSE 126
 FD (FORWARD) command 37
 FENCE command 48
 file(s) 189
 accessing 192
 closing 212, 213
 description of 189
 erasing 196
 listing 194
 opening 211, 214
 reading from 211
 saving 206
 with SAVEL 207
 startup 267
 types 189
 writing to 211
 file buffers 241
 FILELEN operation 214
 filename 193
 changing 199
 FILEP operation 196
 FILERL procedure 218
 FILL command 48
 FILLAT procedure 49
 FILLIN procedure 214
 FINDBIRTH procedure 231
 FINDINFO procedure 224
 FINDTEL procedure 224
 FIRST operation 69, 71
 FLAVORCHART procedure 61
 FLIP procedure 14
 flow of control 125-126
 FOREVER procedure 139, 261
 FORM operation 109
 formatting disks 190
 FORWARD (FD) command 37
 FPUT operation 75, 76
 frequencies, note 171
 FROM.HOME procedure 117
 FS (FULLSCREEN) command 61
 FULLSCREEN (FS) command 61

G

garbage collection 177
 GET.USER procedure 167
 GETLINES procedure 263

global variables 16, 95
GO command 136
GOODVEE procedure 44
GPROP operation 230
graphics, printing 4
graphics buffer 241
graphics screen 35, 59
 erasing with CLEAN 47
 erasing with CLEARSCREEN 37
 loading pictures into 208
 printing the 208
 saving the 208
greater than sign (>) 122
GREET 13, 16, 22, 98, 177, 178

H

halting procedures 126
HASDOTP procedure 158
HEADING operation 43
help feature 4, 6
help screen, loading 197
HEXTODEC procedure 258
HIDETURTLE (HT) command 38
HOME command 38
HT (HIDETURTLE) command 38

I

IF (command or operation) 126
IFFALSE (IFF) command 127
IFTRUE (IFT) command 128
IGNORE procedure 152
INC procedure 102
infix notation 105
infix procedures 276
infix-form operations 118-122
 parsing of 276

INP procedure 87
input word 7
inputs to procedures 13
INSERT procedure 82, 261
inserting text 30
instructions
 repeating 133
 transferring control 133
INT operation 111
integers 105
INTERPRET procedure 165
interrupting procedures 129-133
INTP procedure 111
INTPWR procedure 259
INTQUOTIENT operation 112
INVERSE procedure 281
inverse tangent 108
ITEM operation 73

J, K

KEYP operation 164
keystrokes used at top level 5
kill buffer 27

L

LABEL command 137
LAST operation 73
LATIN procedure 80
LEARN program 148
LEFT (LT) command 38
LENGTH procedure 177, 178
less than sign (<) 121
lines
 continuation 17
 deleting 6
 parsing 275
 reading 167
 retrieving 6

- list(s)
 - breaking into pieces 69
 - description of 68
 - empty 69
 - examining 81
 - property 229
 - putting together 75
 - reading 167
 - LIST operation 75, 76
 - LISTEN procedure 263
 - LISTFILE procedure 217
 - LISTP operation 88
 - LN procedure 258
 - LN1 procedure 259
 - LOAD command 206
 - LOADHELP command 197
 - LOADPIC command 208
 - LOCAL command 98
 - local variables 16, 95
 - LOG procedure 258
 - logical operations 157
 - Logo Editor See Editor
 - Logo line 17
 - lowercase letters 5
 - LOWERCASE operation 90
 - LPUT operation 75, 77
 - LT (LEFT) command 38
- M**
- main memory bank 238
 - MAKE command 15, 99
 - MAP procedure 138, 261
 - MARK.TWAIN procedure 130
 - math operations, evaluated by
 - Logo 107
 - MEMBER operation 74
 - MEMBERP operation 88
 - memory
 - auxiliary bank 238
 - main bank 238
 - memory space, how to save 272
 - MESSAGE procedure 168
 - messages, error 135, 251
 - minus sign (-) 119
 - parsing of 278
 - MODIFY procedure 225
 - MODINFO procedure 226
 - MOUNTAINS procedure 160
 - MOVE procedure 171
 - MOVECURSOR procedure 62
 - moving the cursor 5, 29-30
 - multiplication 106
 - with the asterisk (*) 120
 - with PRODUCT operation 112
 - music, making with TOOT 171
- N**
- N.TO.C procedure 258
 - NAME command 100
 - NAMEIT procedure 264
 - NAMEP operation 101
 - NEAR procedure 120
 - NEWENTRY procedure 78
 - node space 241
 - nodes, allocating 271
 - NODES operation 176
 - NODRIBBLE command 210
 - NOT operation 159
 - notation
 - infix 105
 - prefix 105
 - scientific 106, 110
 - note frequencies 171
 - number(s)
 - square root 117
 - types (decimal and integer) 105
 - NUMBRP operation 89

O

- object 15
- ONLINE operation 197
- OP (OUTPUT) command 130
- OPEN command 214
 - ESC 143
- opening files 211
- operations, logical 157
- operations and commands 14
- OR operation 160
- organizing the workspace 182-185
- OUTPUT (OP) command 130

P

- paddle 163
- PADDLE operation 163
- parentheses 107, 122
 - parsing of 277
- PARSE operation 78
- parsing 275-278
- pathname 193
 - changing 199
- PAUSE (command or operation) 131
- pausing in procedures 126
- PD (PENDOWN) command 49
- PE (PENERASE) command 50
- pen color 53, 55
- PEN operation 54
- pen state 47-53, 54-55
- PENCOLOR (PC) operation 55
- PENDOWN (PD) command 49
- PENERASE (PE) command 50
- PENREVERSE (PX) command 50
- PENUP (PU) command 51
- PHONELIST procedure 231
- picture file(s) 189
 - loading 208
 - printing 4, 208
 - saving the 208
 - working with 207-208
- PIG procedure 80, 81
- PLIST operation 231
- plus sign (+) 119
- PO command 177
- POALL command 178
- POFILE command 198
- POLY procedure 26, 39, 151, 178, 179, 256
- POLYSPI procedure 176
- PON command 178
- PONS command 179
- POPS command 179
- POS operation 43
- POT command 180
- POTS command 180
- PPROP command 232
- PPS command 232
- PR (PRINT) command 169
- predicate(s) 126, 157
- prefix 193
 - directory 193
 - notation 105
 - setting 199
- PREFIX operation 198
- prefix-form operations 107
- PRIMARYP procedure 91
- PRIMITIVEP procedure 147, 150
- primitives 4, 11
- PRINT (PR) command 169
- print text and graphics 4
- PRINTBACK procedure 74
- PRINTDOWN procedure 72
- printers 4
- printing variables 178
- printing with the DUMP procedure 210
- PRINTMESSAGES 27
- PRINTPIC command 208

- procedures 11
 - burying 182, 183
 - debugging 140-144
 - defining 11, 21
 - editing 28
 - erasing 181, 182
 - executing 12
 - halting 126
 - input to 13
 - interrupting 126, 129-133, 143
 - with CONTROL-W 144
 - with CONTROL-Z 144
 - pausing in 126
 - printing definitions of 177, 178
 - printing title lines of 180
 - punctuation in 13
 - saving with SAVE 206
 - saving with SAVEL 207
 - types 14
 - unburying 184
 - ProDOS 190
 - PRODUCT operation 112
 - program files 189
 - working with 206-207
 - programs, debugging 140-144
 - prompt character 21
 - PROMPT procedure 171
 - properties
 - erasing 230
 - printing 232
 - removing 233
 - saving with SAVE 206
 - saving with SAVEL 207
 - property list 229
 - erasing 230
 - printing 232
 - punctuation
 - brackets 13, 68
 - colon 14, 15
 - exclamation mark 17, 27
 - in procedures 13
 - parsing of 275-278
 - quotation marks 13
 - slash 191
 - PWR procedure 259
 - PWRLOOP procedure 259
 - PX (PENREVERSE) command 50
- Q**
- .QUIT command 245
 - QUIZ procedure 127
 - QUIZ2 procedure 128
 - quotation mark 13
 - parsing of 277
 - QUOTIENT operation 113
- R**
- RANDOM operation 113
 - RANPICK procedure 85
 - ratio, aspect 243
 - read position, setting 218
 - READCHAR operation 165
 - READCHARS operation 166
 - READER operation 215
 - reader, setting 217
 - READFILE procedure 215
 - READLINES procedure 149
 - READLIST (RL) operation 167
 - READNUM procedure 134
 - READPOS operation 216
 - READWORD (RW) operation 167
 - REALWORDP procedure 159

recursion 12
 RECYCLE command 177
 REMAINDER operation 114
 removing a character 6
 removing a line 6
 REMPROP command 233
 RENAME command 199
 REPEAT command 137
 repetition 126, 133
 REPORT procedure 132
 REPRINT procedure 169
 RERANDOM command 115
 retrieving a line 6
 REVPRINT procedure 86
 RIGHT (RT) command 39
 ROOT procedure 259
 ROUND operation 116
 RT (RIGHT) command 39
 RUN (command or operation)
 138
 RUNSTORE procedure 263

S

SAFE.SQUARE procedure 139
 SAFESQUARE procedure 136
 sample project using the data
 file 221
 SAVE command 206
 SAVEINFO procedure 222
 SAVEL command 207
 SAVEPIC command 208
 saving space 272
 scientific notation 106, 110
 screen
 changing use of 59
 dimensions 59
 graphics 35, 59
 text 35, 59
 .SCRUNCH operation 243
 SE (SENTENCE) operation 75
 SECRETCODE procedure 81
 SECRETCODELET procedure
 82
 SENGEN procedure 176

SENTENCE (SE) operation 75,
 78
 SETBG command 51
 .SETCRUNCH command 243
 SETCURSOR command 61
 SETH (SETHEADING)
 command 40
 SETHEADING (SETH)
 command 40
 SETPC command 53
 SETPOS command 40
 SETPREFIX command 199
 SETREAD command 217
 SETREADPOS command 218
 SETWIDTH command 62
 SETWRITE command 218
 SETWRITEPOS command 219
 SETX command 41
 SETY command 41
 SHORTQUIZ procedure 129
 SHOW command 170
 SHOWINPUTS procedure 153
 SHOWLINES procedure 153
 SHOWNP operation 44
 SHOWTURTLE (ST) command
 42
 SIN operation 116
 SIREN procedure 171
 slash (/) 191
 SLITHER subprocedure 134
 SNAKE procedure 134
 SORT 261, 261
 SORT procedure 82
 sounds, making with TOOT
 171
 space, saving 272
 SPI procedure 40, 178
 SPLITSCREEN (SS) command
 63
 SQ procedure 117, 176
 SQRT operation 117
 SQUARE procedure 37, 139,
 140, 148, 151
 square root 117

SQUARE.WITH.TAIL
 procedure 151
 SS (SPLITSCREEN) command
 63
 starting up the Editor 31
 STARTUP file, creating 267
 STARTUP variable 268
 STEER procedure 165
 \$STEP procedure 152
 STEP command 141
 STEPPER procedure 152
 STOP command 132
 STORE procedure 213, 216,
 264
 subdirectories 191
 creating 192, 195
 erasing 192
 listing 194
 prefix 193
 subprocedure 12, 125
 subtraction 106
 with DIFFERENCE operation
 109
 SUFFIX procedure 80
 SUM operation 117
 superprocedure 12, 125
 SUPERSORT procedure 82

T

TAB procedure 60
 TALK procedure 86
 TAN procedure 109
 tangent, inverse 108
 TEST command 128
 TESTIT procedure 275
 text 59
 deleting 30
 inserting 30
 printing 4
 TEXT operation 147, 151
 text screen 35, 59
 dribbling from 209

TEXTSCREEN (TS) command
 63
 THING operation 102
 THROW command 133, 136,
 140
 title line(s) 11
 printing 180, 180
 TO command 21
 TOOT command 171
 top level 11
 getting help at 4
 keystrokes for use at 5
 TOWARDS operation 44
 TRACE command 141
 transferring control 133
 TRIANGLE procedure 70, 141,
 154
 TRIANGULATE procedure 176
 TRUE (predicate) 126
 TRYAGAIN procedure 264
 TS (TEXTSCREEN) command
 63
 TURN procedure 165
 turtle graphics 35-55
 TYPE command 170
 typing in the Editor 29
 typing uppercase and
 lowercase letters 5

U

UNBURY command 184
 UNBURYALL command 184
 UNBURYNAME command 185
 \$UNSTEP procedure 152
 UNSTEP command 143
 UNTRACE command 143
 uppercase letters 5
 UPPERCASE operation 91

V

- value 95
 - of variable 178, 179
- variable(s) 14, 15-16
 - assigning values to 15
 - creating 95
 - with MAKE 99
 - with NAME 100
 - description of 95
 - editing with EDN 96
 - editing with EDNS 97
 - erasing 181
 - global 16, 95
 - local 16, 95
 - names
 - burying 183
 - printing 179
 - unburying 184, 185
 - saving with SAVE 206
 - saving with SAVEL 207
 - STARTUP 268
 - types 16, 95
 - value, printing 178, 179
- VEE procedure 44
- volume directory 190
- volume names 190
 - listing 197
- VOWELP procedure 89

W

- WAIT command 132
- WALK procedure 131
- WARMWELCOME procedure 12
- WEATHER procedure 100
- WELCOME procedure 269
- WHICH procedure 130
- WHILE procedure 138, 262
- WIDTH operation 63
- WINDOW command 53

- WIPEOUT procedure 263
- word, empty 68
- word delimiters 68
- WORD operation 75, 80
- WORDP operation 90, 159
- words
 - breaking into pieces 69
 - description of 67
 - changing the case of 90
 - examining 81
 - putting together 75
- workspace
 - cleaning 182-185
 - description of 175
 - erasing from 180-182
 - organizing 182-185
 - printing from 177-180
 - saving with SAVE 206
 - saving with SAVEL 207
- WRAP command 53
- write position, setting 219
- WRITEINFO procedure 222
- WRITEPOS operation 220
- writer, setting 218
- WRITER operation 221

X

- x-y coordinates 41, 45, 46
- XCOR operation 45
- XYZZY procedure 165

Y, Z

- y coordinate 41, 45, 46
- YCOR operation 46
- YESNO procedure 98
- YESP procedure 91



Apple Logo II Reference Card

Parentheses around an input indicate that the input is optional. A number sign (#) indicates a procedure that can take any number of inputs; if you give it other than the number indicated, you must enclose the entire expression in parentheses.

Defining and Editing

EDIT, ED (name(list))
EDN name(list)
EDNS
END
TO name (inputs)

Turtle Graphics

BACK, BK n
BACKGROUND, BG
CLEAN
CLEARSCREEN, CS
DOT [xcor ycor]
DOTP [xcor ycor]
FENCE
FILL

FORWARD, FD distance
HEADING
HIDETURTLE, HT
HOME
LEFT, LT degrees
PEN

PENCOLOR, PC
PENDOWN, PD
PENERASE, PE
PENREVERSE, PX
PENUP, PU
POS

RIGHT, RT degrees
SETBG color number
SETHEDGING, SETH degrees
SETPC color number
SETPOS [xcor ycor]
SETX xcor
SETY ycor

SHOWNP
SHOWTURTLE, ST
TOWARDS [xcor ycor]
WINDOW
WRAP
XCOR
YCOR

Text and Screen

CLEARTEXT, CT
CURSOR
FULLSCREEN, FS
SETCURSOR [column number
linenumber]
SETWIDTH width
SPLITSCREEN, SS
TEXTSCREEN, TS
WIDTH

Words and Lists

ASCII char
BEFOREP word1 word2
BUTFIRST, BF obj
BUTLAST, BL obj
CHAR integer
COUNT obj
EMPTYP obj
EQUALP obj1 obj2
FIRST obj
FPUT obj list
ITEM integer obj
LAST obj
#LIST obj1 obj2
LISTP obj
LOWERCASE word
LPUT obj list
MEMBER obj1 obj2
MEMBERP obj1 obj2
NUMBERP obj
PARSE word
#SENTENCE, SE obj1 obj2
UPPERCASE word
#WORD word1 word2
WORDP obj

Variables

EDN name(list)
EDNS
#LOCAL name(list)
MAKE name obj
NAME obj name
NAMEP word
THING name

Arithmetic Operations

ARCTAN number
COS degrees
DIFFERENCE number1
number2
FORM number field precision
INT number
INTQUOTIENT integer1 integer2
#PRODUCT number1 number2
QUOTIENT number1 number2
RANDOM integer
REMAINDER integer1 integer2
RANDOM
ROUND number
SIN degrees
SQRT number
#SUM number1 number2

Flow of Control

CATCH name list
CO
ERROR
GO word
IF pred list1 (list2)
IFFALSE, IFF list
IFTRUE, IFT list
LABEL word
OUTPUT, OP obj
PAUSE
REPEAT integer list
RUN list
STEP name(list)
STOP
TEST pred
THROW name
TRACE name(list)
UNSTEP name(list)
UNTRACE name(list)
WAIT integer

Defining Procedures Under Program Control

COPYDEF name newname
DEFINE name list
DEFINEDP word
PRIMITIVEP name
TEXT name

Logical Operations

#AND pred1 pred2
NOT pred
#OR pred1 pred2

The Outside World

BUTTONP paddlenumber
KEYP
PADDDLE paddlenumber
#PRINT, PR obj
READCHAR, RC
READCHARS, RCS integer
READJUST, RL
READWORD, RW
SHOW obj
TOOT frequency duration
#TYPE obj

Workspace

Management

BURY name(list)
BURYALL
BURYNAME name(list)
ERALL
ERASE, ER name(list)
ERN name(list)
ERNS
ERPS
NODES
PO name(list)
POALL
PON name(list)
PONS
POPS
POT name(list)
POTS
RECYCLE
UNBURY name(list)
UNBURYALL
UNBURYNAME name(list)

Files

ALLOPEN
CATALOG
CLOSE FILE
CLOSEALL
CREATEDIR pathname
DRIBBLE file
EDITFILE pathname
ERASEFILE, ERF pathname
FILELEN file
FILEP file
LOAD pathname
LOADPIC pathname
NOORIBBLE
ONLINE
OPEN pathname
POFILE pathname
PREFIX
PRINTPIC integer
READER
READPOS
RENAME pathname
newpathname
SAVE pathname
SAVEL name(list) pathname
SAVEPIC pathname
SETPREFIX prefix
SETREAD file
SETREADPOS integer
SETWRITE file
SETWRITEPOS integer
WRITER
WRITER

Property Lists

ERPROPS
GPROR name prop
PLIST name
PPROP name prop obj
pps
REMPROP name prop

Special Primitives

.AUXDEPOSIT loc byte
.AUXEXAMINE loc
.BLDAD pathname loc
.BSAVE pathname loc integer
.CALL loc
.CONTENTS
.DEPOSIT loc byte
.EXAMINE loc
.QUIT
.SCRUNCH
.SETSCRUNCH number

Primitives (Infix Form)

number1 + number2
number1 - number2
number1 * number2
number1 / number2
number1 < number2
obj1 - obj2
number1 > number2

Help Screen Primitives

HELP word
LOADHELP pathname

Special Words

ERROR
FALSE
STARTUP
TOPLEVEL
TRUE





20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010
TLX 171-576

Printed in U.S.A.

030-0866 C
A2L4033