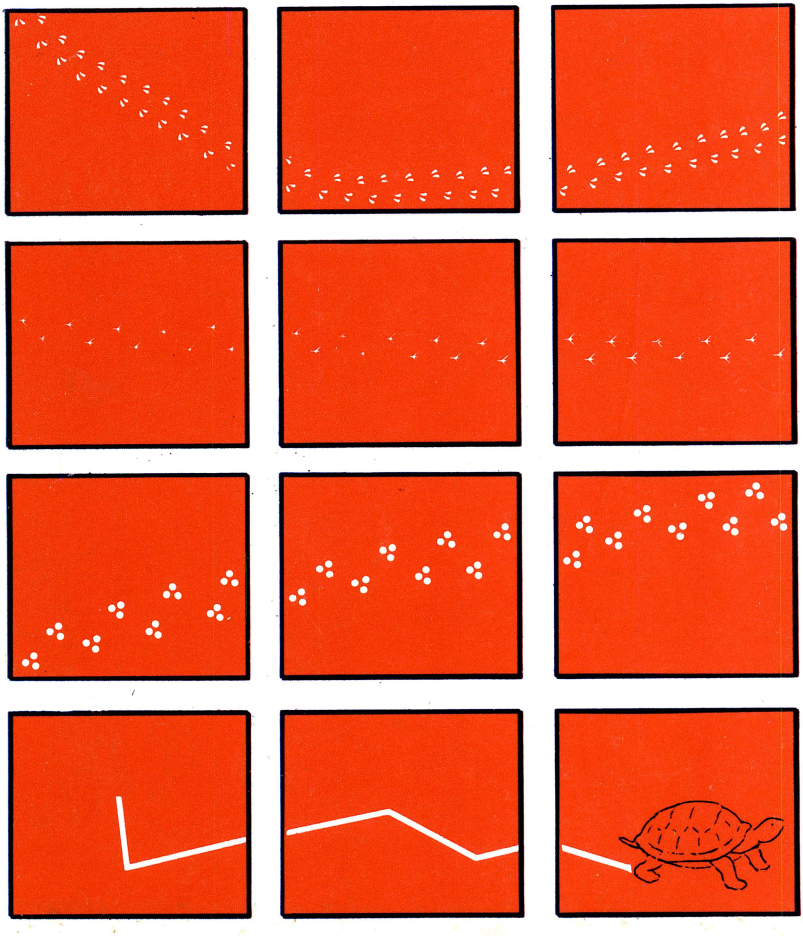
Apple II

Apple Logo

Reference Manual



A Product of Logo Computer Systems, Inc.

Apple Logo

Reference Manual

Author: Laurence J. Davidson

Editorial Board: Alison Birch, Edward Hardebeck, Brian Harvey, Margaret Minsky, Seymour Papert, Cynthia Solomon, and the staff of LCSI.

Technical Editors: Glenn Forrester and William Weinreb

Designers: Judith Richland and Nancy Gardner

Plotter Images: Peter Cann

Photographs: Ralph Mercer

Compositor: County Photo Compositing Corporation

Printer: RBT Printing Ltd.

This Guide is set in Century Textbook and OCR-B on a Quadex/ Compugraphic 8600 typsetting system.

Logo Computer Systems, Inc.
222 Brunswick Blvd.
Pt. Claire, Quebec
Canada H9R 1A6
(514) 694-2885

989 Avenue of the Americas
New York, New York 10018

368 Congress St.
Boston, Ma. 02210

Printed in Canada.

Table of Contents

How to Use This Manual

This manual is not a primer. We assume you know some things about Logo or about languages like Logo.

If you are approaching Logo for the first time, you should begin by working through the *Introduction to Programming through Turtle Graphics* that comes with Apple Logo. Experienced programmers and computer scientists should be able to pick up enough about Logo by reading this manual directly.

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

Procedure definitions and sample dialogs between the user and Logo are printed in a different type font from the rest of the manual. We use this font to represent more closely what you see on your text screen. Sometimes we further distinguish by using one color ink to represent what the computer types and another color to represent what you type. There is one liberty we have taken: your Apple screen can show 40 characters on a line; in this guide we can show only 37 characters across the page.

There are several ways to use this manual. If you want to know what a particular primitive does, look it up in the index, or for a quick reference look at Appendix A or B. If you want to find a primitive to perform a particular task, look at the categories listed as chapter headings. You might also look for references in the index.

Sample entry
Here is a typical procedure entry:

MAKE

MAKE *name object* (command)
MAKE expects two inputs: the first is a Logo name, the second is any Logo object. MAKE is a *command* (as opposed to an *operation*). Notice that the procedure name is printed in capital letters. Its inputs, if any, are printed in italics. For convenience the procedure name is also printed in the left margin.

A procedure entry is followed by a description of what the procedure does, and then some examples of its use are given. The examples are set in a different type to more closely resemble what you see on your Apple.

Here is a slightly more complicated entry:

PRINT
PR

PRINT *object* (command) short form: PR
(PRINT *object1 object2* . . .)
The *short form*, PR, is an abbreviated synonym for the command PRINT. Ordinarily, PRINT expects one input, any Logo object. But PRINT can also be given any number of inputs at once, provided that the entire expression is enclosed in parentheses. We show this variation underneath the visual form.

An Introduction to Logo for Experienced Programmers

Logo Programs

A Logo program is a collection of procedures.

For example, a program to draw a house consists of these procedures: HOUSE, BOX, TRI, RIGHT, FORWARD, and REPEAT. Of these, the last three are *primitives.* The first three are user-defined procedures, built out of Logo primitives.

```
TO HOUSE
BOX
FORWARD 50
RIGHT 30
TRI 50
END

TO BOX
REPEAT 4 [FORWARD 50 RIGHT 90]
END

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

We are assuming that you have had some experience with Logo and have built up an intuitive model of what Logo is about. Here we give a more formal model. This model is just another way to think about Logo. It is not meant to replace your current way of thinking, but rather to enhance it.

Formal Logo

The actual Logo system implemented for your Apple computer is very complicated. It will help if we consider a formal description of Logo, and then Logo as it really is. We begin our formal description by restricting ourselves to a part of Logo, which we'll call *formal Logo;* we then relax the restrictions in order to describe the language you actually use, which we'll call *relaxed Logo.*

Every instruction in formal Logo works without change in relaxed Logo. Conversely, anything that can be done in Logo can be done in formal Logo, but you will immediately recognize situations where formal Logo forces idioms that no one would actually use. For example, if you are experienced in Logo you will notice that numbers look peculiar in formal Logo: they are quoted. Since numbers are words, you can quote them in relaxed Logo as well. But we do so very rarely; in relaxed Logo, numbers are self-quoting.

Procedures and inputs: We assume that you know what procedures are, at least on an intuitive level. Here we develop some more precise ways to talk about them.

In formal Logo, every procedure requires a definite number of inputs. These inputs are always one of two kinds: they may be *words* or they may be *lists*. They may be given directly, as in FORWARD "100, or indirectly through the mediation of another procedure, as in FORWARD SUM "60 "40. Both of these examples have the same effect. If an input is given directly as a word, it must be preceded by a quote mark, as in PRINT "FORWARD. If it is given directly as a list, it is surrounded by square brackets, as in PRINT [HOW ARE YOU?].

Words and lists: A word is made up of characters. A list is made up of elements enclosed in square brackets, with spaces between elements; an element is either a word (without quotes) or a list.

Expressions: A procedure name followed by the required number of inputs is an *expression*. More generally, an *expression* is

- a quoted word,
- a list, or
- an (unquoted) procedure name followed by as many expressions as the procedure requires.

If this seems complicated, look at an example. REPEAT is a procedure that requires two inputs.

This is an expression:

```
REPEAT "3 [FD "10]
```

The following is also an expression:

```
REPEAT SUM "2 "1 [FD "10]
```

In this case the first input is not a quoted word, but an expression involving another procedure, SUM.

The following is also an expression:

```
REPEAT SUM "3 "1 SENTENCE "FD "10
```

Commands and operations: There are two kinds of procedures in Logo. Those (like SUM) that produce a Logo object are called *operations*. The others (like PRINT) are called *commands*.

With these definitions we can define a Logo instruction.

A Logo *instruction* is a particular kind of expression. It starts with a procedure name, and that procedure must be a command. All other procedures in the expression must be operations.

Why does it matter whether an expression is an instruction or not? Consider

```
SUM "3 "4
```

This is an expression, but it is not a Logo instruction.

SUM will produce a number; but simply writing SUM "3 "4 does not state what is to be done with the number. In fact, Logo will give an error message saying I DON'T KNOW WHAT TO DO WITH 7. Some programming languages would allow this and simply print the 7. In designing Logo we preferred to make every important act explicit; if you want something to be printed you should say so.

Relaxed Logo

Everything that can be done in Logo can be done using formal Logo. But we have allowed some other idioms, either to make the language feel more natural or to eliminate large amounts of typing.

Here we list some important relaxations of Logo. Others are mentioned in the body of the manual. See also Appendix I (Parsing).

Numbers: In formal Logo, all words used as direct inputs, including numbers, must be quoted. In relaxed Logo, numbers are self-quoting: the quote marks are unnecessary. For example,

```
SUM 3 4
```

A *number* is a word made up of digits; it may also contain a minus sign, a period, and an E or an N. See Chapter 4 (Arithmetic Operations).

Infix procedures: Formal Logo uses only prefix procedures. Thus addition is expressed by SUM "3 "4. In relaxed Logo you can also use the infix form 3 + 4. Similarly with multiplication, subtraction, and division. See Chapter 4 (Arithmetic Operations).

Variable number of inputs: In formal Logo, every procedure has a fixed number of inputs. In relaxed Logo, several procedures can have variable numbers of inputs. When you use other than the expected number of inputs you must enclose the expression in parentheses:

```
(SUM 3 4 5 6 7 8)
```

Such procedures do not require parentheses when you use fewer than the standard number of inputs, provided that nothing else follows the expression on the same line.

Dots: A familiar feature of relaxed Logo is the use of dots (:). In formal Logo, :X is written as THING "X.

Parentheses for grouping: Logo has parentheses so that you can explicitly tell Logo how to group things. For example, consider

```
25 + 20 / 5
```

Should the addition be done first, producing 9, or should the division be done first, producing 29? Relaxed Logo follows the traditional mathematical hierarchy, in which multiplication and division are done before addition and subtraction; thus this operation first divides 20 by 5, and produces 29. To group the numbers so that addition is done first, parentheses must be used:

```
(25 + 20) / 5
```

Commands and operations: In formal Logo, a given procedure is either a command or an operation, not both. In relaxed Logo, a procedure can

be sometimes a command and sometimes an operation. RUN and IF are examples of this. See Chapter 7 (Flow of Control).

The command TO: In formal Logo, you define a procedure with the command DEFINE; it takes two inputs, a word (the procedure name) and a list (the definition). All the usual rules about quotes and brackets apply without exception:

```
DEFINE "SQUARE [[ ] [REPEAT 4 [FD 100 RT 90]]]
```

In relaxed Logo, you can define a procedure with EDIT and TO. The command TO is unusual in two ways. It automatically quotes its inputs, and it allows you to type in the lines of the procedure interactively:

```
?TO SQUARE
>REPEAT 4 [FD 100 RT 90]
>END
```

A Further Note on Operations

In talking about formal Logo we have been using the word *produce:* a procedure *produces* an object (called the *output* of the procedure). In traditional Logo terminology we say that the procedure *outputs* an object.

Logo Objects

There are two types of Logo *things* (or *objects*): *words* and *lists.*

A word, such as TELEVISION or 617, is made up of characters (letters, digits, punctuation). A number is a kind of word.

A list, such as [RABBITS TELEVISION 7 [EARS FEET]], is made up of Logo objects, each of which can be either a word or a list. See Chapter 2 (Words and Lists).

How You Might Think about Quotes

The role of quotes is best understood through the following example:

```
?PRINT "HEADING
HEADING
?PRINT HEADING
180.
```

In the first case the quotation mark (") indicates that the word HEADING itself is the input to PRINT.

In the second case the input is not quoted. It is therefore interpreted as a procedure, which provides the input to PRINT.

How You Might Think about MAKE

The effect of the command

```
MAKE "BIRD "PIGEON
```

can be thought of as follows. A container is given a name: BIRD. The word PIGEON is put into the container.

Once this is done the operation THING has the following effect:

```
THING "BIRD
  produces PIGEON
```

Dots are similar to THING; for example,

```
:BIRD
  produces PIGEON
```

We can talk about the same situation in several ways:

BIRD is a *variable*; PIGEON is its *value*.
BIRD is a *name* of PIGEON.
BIRD is *bound* to PIGEON.
PIGEON is the *thing* of BIRD.

You can change what it is in the container; if you type

```
MAKE "BIRD "SPARROW
```

then the container BIRD contains the word SPARROW instead of the word PIGEON.

How to Think about Procedures You Define and their Inputs

Procedures you define can have inputs. When you define a procedure, its title line specifies how many inputs it has and a name for each one. The inputs are put into the containers designated on the title line of the procedure definition in the order in which they occur. For example, the following procedure makes the turtle draw various polygons, depending upon what inputs are used.

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

Whatever inputs you choose—for example, 50 and 90—are put into the specified containers. The first input is always put in the container STEP; the second input is always put in the container ANGLE. The first input variable, STEP, controls the size of the figure. The second input variable, ANGLE, controls the shape of the figure. POLY 50 90 makes the turtle draw a square; POLY 50 144 makes the turtle draw a pentagram; POLY 50 120 makes the turtle draw a triangle.

If the input containers already have objects in them when the procedure is run, the objects are removed but are saved. They are restored when

the procedure is done. In other words, the procedure *borrows* the container and leaves it in its original condition when done with it. This is what is meant by saying that a variable is *local* to the procedure within which it occurs.

For example:

```
?MAKE "SOUND "CRASH
?PRINT :SOUND
CRASH
```

Now we run a procedure called ANIMAL; its input variable is SOUND.

```
TO ANIMAL :SOUND
IF :SOUND = "MEOW [PR "CAT STOP]
IF :SOUND = "WOOF [PR "DOG STOP]
PR [I DON'T KNOW]
END
```

If we type ANIMAL "WOOF, then WOOF is put into the SOUND container for the duration of the ANIMAL procedure. Afterwards, CRASH is restored:

```
?MAKE "SOUND "CRASH
?PRINT :SOUND
CRASH
?ANIMAL "WOOF
DOG
?PRINT :SOUND
CRASH
```

Another Way to Talk about Procedures

When you use a procedure, we also say that you *call* that procedure. Procedures you define call other procedures. For example, the POLY procedure above calls other procedures (FORWARD and RIGHT).

POLY also calls itself. This is called a *recursive* call.

Review of Special Characters

Quotation marks, or *quotes,* ", used immediately before a word, indicate that it is being used *as a word,* not as the name of a procedure or the value of a variable.

A *colon,* known as *dots,* :, used immediately before a word, indicates that the word is to be taken as the name of a variable and produces the value of that variable.

Brackets, [ ], are used to surround a list. The left bracket ([) is typed with SHIFT-N, the right bracket (]) with SHIFT-M.

Parentheses, ( ),are necessary to group things in ways that Logo ordinarily would not, and to vary the number of inputs for certain procedures. Both of these uses are described above in this Overview.

A *backslash,* \, tells Logo to interpret the character that follows it literally *as a character,* rather than keeping some special meaning it might have. For instance, suppose you wanted to use 3[A]B as a single *word.* You need to type 3\[A\]B in order to avoid Logo's usual interpretation of the brackets as the envelope around a list. You have to backslash [, (,],), +, −, *, /, =, <, >, and \ itself.

Since the Apple keyboard does not contain a backslash, you type CTRL-Q. You can remember CTRL-Q by thinking of it as *quoting* the character that follows.

You need a 48K Apple II computer system, including a language card (16K RAM card) in slot 0, a 16-sector disk controller card in slot 6, at least one disk drive connected to the controller card, and a color or black-and-white TV. Setting up the Apple hardware is discussed in the Apple II manuals.

Your Logo kit includes a Logo diskette, which you put into the disk drive before you can use Logo. To put in the diskette, open the door of the disk drive by lifting it outwards; a hinge is on top. You should see a slot for the diskette. Hold the diskette so that the label is face up and slide it into the slot of the disk drive, then *close the door.* The disk drive is now ready to load information from the diskette into the Apple.

Turn the TV on. Then, to start Logo, turn on the Apple power switch, which is located on the back side of the computer. You can reach it from your left.

The disk drive light will go on. After a moment, the light will go off and if all is going well you should soon see a message on the screen saying

```
PRESS THE RETURN KEY TO BEGIN

IF YOU HAVE YOUR OWN FILE DISKETTE,
INSERT IT NOW, THEN PRESS RETURN
```

You do not need a file diskette to use Logo. If you do not have one and you want to continue without one, just press the RETURN key and go on. If you do not have a file diskette and want one now, see the section below titled "How to format a diskette".

After you press the RETURN key, Logo loads the file named STARTUP, if there is one*. You will then see a message on the screen saying

```
WELCOME TO LOGO
?
```

Below the message you will see a ? (question mark) followed by a flashing ▮ (rectangle).

The question mark is the *prompt* symbol, indicating that Logo is waiting for you to type something. Whenever you are typing in response to the prompt symbol, we say that you are at *toplevel.*

The flashing rectangle is the *cursor.* It appears when Logo wants you to type something and shows where the next character you type will appear.

Special Keys on the Keyboard

The ← Key

The ← or DELETE key causes Logo to erase the character before the cursor. Your Logo kit should have a DELETE sticker for you to put on the front of this key.

The RETURN Key

The RETURN key tells Logo: "Now do what I just typed."

The SHIFT Key

The SHIFT key can change character keys. For example, if while holding the SHIFT key down you press N, Logo prints [on the screen. We represent this two-key combination as SHIFT-N. SHIFT-M is]. Your Logo kit should have stickers to mark

*If you press CTRL-G instead of RETURN, Logo does not load the STARTUP file.

SHIFT-M and SHIFT-N as] and [on the fronts of the M and N keys.

The Space Bar
The space bar prints an invisible character called *space.* Logo uses spaces as word separators.

The CTRL Key
The CTRL key can change character keys into action keys. You use it like the SHIFT key: press it alone and nothing happens; hold it down and press a certain character key, and something happens. For example, if you type CTRL-G, Logo stops whatever it is doing.

How to Format a Diskette
To save your procedures and other Logo things you need a formatted Apple diskette on which you can write. If you do not have one you can make one.

Using the Apple DOS 3.3 SYSTEM MASTER you can format a diskette. Put the diskette in the disk drive and start it up by repowering your Apple. Then when the disk drive light goes off, remove the SYSTEM MASTER diskette and replace it with the blank diskette you want to format.

Now type

```
INIT HELLO
```

When the disk drive light goes off, the diskette will be formatted. You can test this out by typing

```
CATALOG
```

The file name HELLO should be displayed.

Now put the Logo diskette in the disk drive. Repower (reboot) the Apple. When Logo prints

```
PRESS THE RETURN KEY TO BEGIN

IF YOU HAVE YOUR OWN FILE DISKETTE,
INSERT IT NOW, THEN PRESS RETURN
```

press the RETURN key. Do not take the Logo diskette out of the drive yet. Let Logo load in the STARTUP file from the diskette.

After Logo prints

```
WELCOME TO LOGO
```

remove the Logo diskette and insert your file diskette. Now type

```
SAVE "STARTUP "AIDS
```

Now you have your own file diskette with its STARTUP file.

Chapter 1

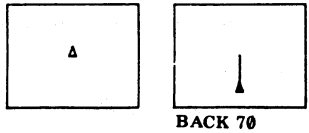
Turtle Graphics

When you use any primitive which refers to the turtle, Logo shows you the turtle graphics screen.*

In addition to those described in this chapter, the following primitives are related to turtle graphics: FULLSCREEN, SPLITSCREEN, TEXTSCREEN, which are described in Chapter 10.

BACK
BK

BACK *distance* (command) short form: BK
Moves the turtle *distance* steps back. Its heading does not change.



BACK 70

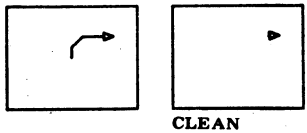
BACKGROUND
BG

BACKGROUND (operation) short form: BG
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)

CLEAN

CLEAN (command)
Erases the graphics screen but doesn't affect the turtle.



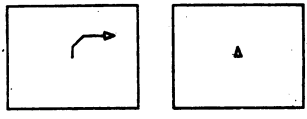
CLEAN

*If you use the Logo Editor or any file command, the screen is erased.

CLEARSCREEN
CS

CLEARSCREEN (command) short form: CS

Erases the graphics screen, puts the turtle at position [0 0] (which is the center of the screen and is called the *home position*), and sets the turtle's heading to 0 (north).



CLEARSCREEN

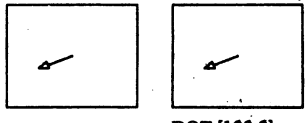
DOT

DOT *position* (command)

Puts a dot of the current pen color at the specified *position*, without moving the turtle. It does not draw a line, even if the pen is down.

Example:

DOT [100 0] puts a dot halfway down the right edge of the screen.



DOT [100 0]

FENCE

FENCE (command)

Fences in the turtle within the edges of the screen. If an attempt is made to move the turtle beyond the edges of the screen it will not move, and the error message "TURTLE OUT OF BOUNDS" is typed. An error message is also typed if you say FENCE when the turtle is already off the screen. See also WINDOW and WRAP.

Example:

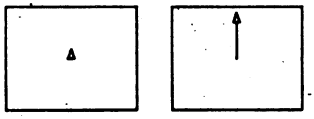
```
FENCE
CS
RT 5
FD 500
```

gives the error message "TURTLE OUT OF BOUNDS"

FORWARD
FD

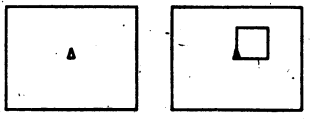
FORWARD *distance* (command) short form: FD
Moves the turtle forward *distance* steps in the direction in which it is heading.

Examples:



FORWARD 70

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



SQUARE 30

HEADING

HEADING (operation)
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).

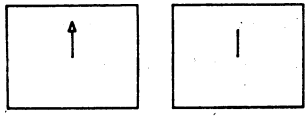
Example:

```
IF HEADING = 180 [PR [YOU ARE HEADED DUE SOUTH]]
```

HIDETURTLE
HT

HIDETURTLE (command) short form: HT

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

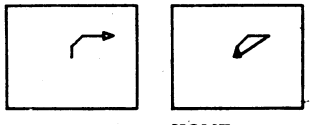


HIDETURTLE

HOME

HOME (command)

Moves the turtle to the center of the screen and sets its heading to 0. This command is equivalent to SETPOS [0 0] SETHEADING 0.



HOME

LEFT
LT

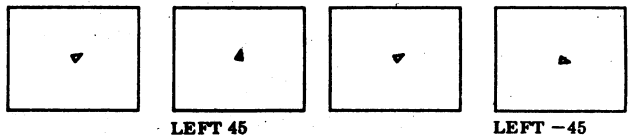
LEFT *degrees* (command) short form: LT

Turns the turtle left (counterclockwise) the specified number of *degrees*. It is an error if *degrees* > 4.19E6.

Examples:

LEFT 45 (turns the turtle 45 degrees left)

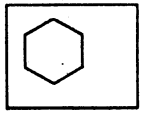
LEFT −45 (turns the turtle 45 degrees right)



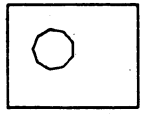
LEFT 45 LEFT −45

The procedure POLY draws figures like those illustrated:

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



POLY 70 60



POLY 30 40

POLY 80 144

PEN

PEN (operation)

Outputs a two-word list describing the current state of the turtle's pen. The first word is PENDOWN, PENERASE, PENUP, or PENREVERSE. The second word is the number representing the current pen color. When the turtle first starts up, PEN outputs [PENDOWN 1].

The form of the output is suitable for input to SETPEN. See example in entry for SETPEN.

PENCOLOR
PC

PENCOLOR (operation) short form: PC

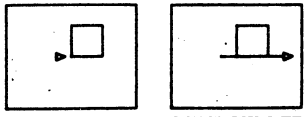
Outputs a number representing the current color of the pen:

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

When the turtle first starts up, PENCOLOR is 1.

PENDOWN
PD

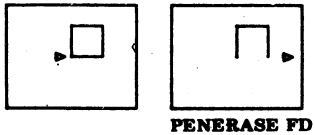
PENDOWN (command) short form: **PD**
Puts the turtle's pen down: when the turtle moves, it draws lines in the current pen color. The turtle begins with its pen down.



PENDOWN FD 100

PENERASE
PE

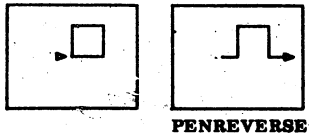
PENERASE (command) short form: **PE**
Puts the turtle's eraser down: when the turtle moves, it erases lines it passes over. To lift the eraser, run either **PENDOWN** or **PENUP**.



PENERASE FD 100

PENREVERSE
PX

PENREVERSE (command) short form: **PX**
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 pen color, background color, and whether lines are horizontal or vertical. The best results are on a black background.

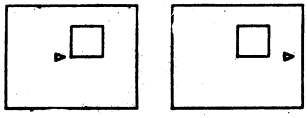


PENREVERSE FD 100

PENUP
PU

PENUP (command) short form: PU

Lifts the pen up: when the turtle moves, it does not draw lines.



PENUP FD 100

POS

POS (operation)

Outputs the coordinates of the current position of the turtle in the form of a list [x y]. 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
END

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

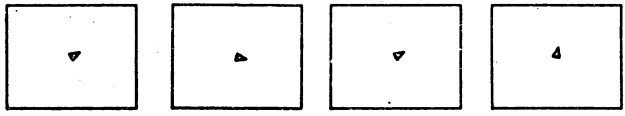
GOODVEE

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

RIGHT
RT

RIGHT *degrees* (command) short form: RT
Turns the turtle right (clockwise) the specified number of *degrees*. It is an error if *degrees* > 4.19E6.

Examples:
RIGHT 45 (turns the turtle 45 degrees right)
RIGHT −45 (turns the turtle 45 degrees left)



RIGHT 45 RIGHT −45

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

SPI 5 144 3

SCRUNCH

SCRUNCH (operation)
Outputs the *aspect ratio*, a decimal number which is the ratio of the size of a vertical turtle step to the size of a horizontal one. The aspect ratio is .8 when Logo starts up. See also SETSCRUNCH.

SETBG

SETBG *colornumber* (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)

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. See example in entry for 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

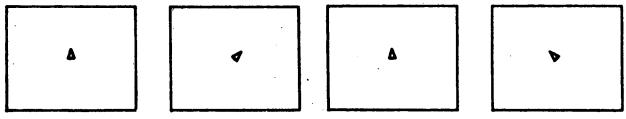
green or violet background:
orange becomes green
blue becomes violet

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

SETHEADING
SETH

SETHEADING *degrees* (command)
short form: SETH
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)



SETHEADING 45 SETHEADING −45

SETPC

SETPC *colornumber* (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

If the pen color does not look right on your screen, try adjusting your TV's 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.

For information on the interaction between pen and background colors, see entry for SETBG.

SETPEN

SETPEN *pair* (command)

Sets the pen state to the two-word list *pair*. The form of the input is the same as that of the output from PEN (see PEN). The first word must be PENDOWN, PENERASE, PENUP, or PENREVERSE. The second word is the number representing the current pen *color*.

Examples:

SETPEN [PENUP 3] is equivalent to

```
PENUP
SETPC 3
```

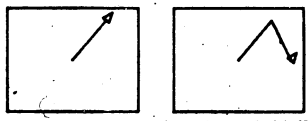
In the following program, the procedure DESIGN draws three sides of a square in three different colors; it happens to leave the pen up, with the pen color blue. The procedure RESTORE calls a procedure and ensures that the pen state is restored to whatever it was previously:

```
TO RESTORE :PROCEDURE
MAKE "SAVEPEN PEN
RUN SE :PROCEDURE
SETPEN :SAVEPEN
END

TO DESIGN
PIECE 2
PIECE 3
PIECE 5
PENUP FD 50
END

TO PIECE :COLOR
SETPC :COLOR
FD 50 RT 90
END
```

SETPOS

SETPOS *position* (command)
Moves the turtle to *position* (see POS).

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



SETPOS [100 0]

SETSCRUNCH

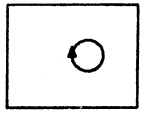
SETSCRUNCH *n* (command)
Sets the aspect ratio (the ratio of the size of a vertical turtle step to the size of a horizontal one) to *n*. YCOR is changed accordingly.

Example:
SETSCRUNCH .5 makes each vertical turtle step half the length of a horizontal one.

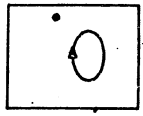
SETSCRUNCH has two uses. First, if "squares" turn out to be rectangles and "circles" turn out to be ellipses on your screen, you can correct this (for most screens an aspect ratio of .8 is correct). Second, if you want turtle drawings to come out squashed or extended, you can use SETSCRUNCH; for example, you can use a circle procedure to draw an ellipse:

```
TO CIRCLE :RADIUS
REPEAT 60 [FD :RADIUS * 3.14159 / 30
 RT 6]
END
```

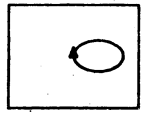
```
TO ELLIPSE :HORIZ :VERT
SETSCRUNCH .8 * :VERT / :HORIZ CIRC!
LE :HORIZ
END
```



CIRCLE 25



ELLIPSE 25 40



ELLIPSE 40 25

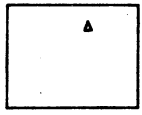
SETX

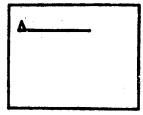
SETX *x* (command)

Moves the turtle horizontally to a point with x-coordinate *x* (y-coordinate is unchanged).

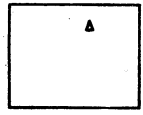
Example:

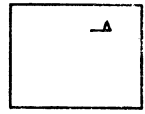
SETX −100 moves the turtle horizontally over to the left edge of the screen.





SETX −100





SETX 2 * XCOR

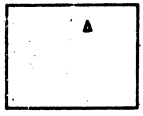
SETY

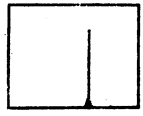
SETY *y* (command)

Moves the turtle vertically to a point with y-coordinate *y* (x-coordinate is unchanged).

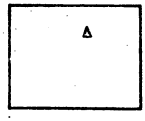
Example:

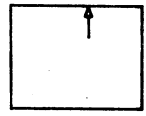
SETY −100 moves the turtle vertically down to the lower edge of the screen.





SETY −100





SETY 2 * YCOR

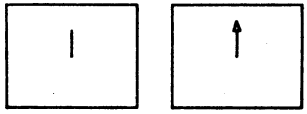
SHOWNP

SHOWNP (operation)

Outputs TRUE if the turtle is shown, FALSE otherwise.

SHOWTURTLE
ST

SHOWTURTLE (command) short form: ST
Makes the turtle visible. See also HIDETURTLE.



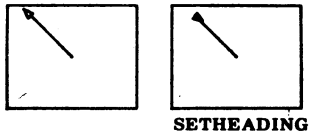
SHOWTURTLE

TOWARDS

TOWARDS *position* (operation)
Outputs a heading that would make the turtle face *position.*

Example:

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



SETHEADING TOWARDS [20 10]

WINDOW

WINDOW (command)
Make 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 40960 steps high* and 32768 steps wide. See also FENCE and WRAP.

Example:

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

*Depending on the value of SCRUNCH

WRAP

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. See also FENCE and WINDOW.

Example:

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

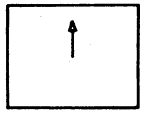
XCOR

XCOR (operation)

Outputs the x-coordinate of the current position of the turtle.

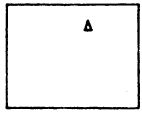
Examples:

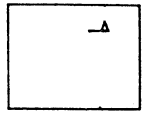
```
?PRINT XCOR
0
```



PRINT XCOR
0

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





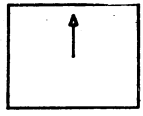
SETX 2 * XCOR

YCOR

YCOR (operation)
Outputs the y-coordinate of the current position of the turtle.

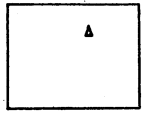
Examples:

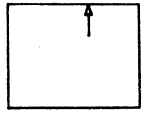
```
?PRINT YCOR
100
```



PRINT YCOR
100

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





SETY 2 * YCOR

Chapter 2

Words and Lists

Chapter 2 Words and Lists

There are two types of *objects* in Logo: *words* and *lists*. There are primitives to put them together, take them apart, and examine them.

A *word* is made up of characters.

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

These are all *words*. 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. That is, there is a space before the word (unless it is preceded by : or ") and a space after the word; they set the word off from the rest of the line. There are a few other delimiting characters:

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

To treat any of these characters as a normal alphabetic character, put a backslash "\" (typed with CTRL-Q) before it.

Example:

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

See Appendix I (Parsing) for more detail on this subject.

*Lehrer, Thomas. "An Evening Wasted With Tom Lehrer" (Reprise Records, 1960). The 3 is silent.

Note that " is not a word delimiter.

A *list* is made up of Logo *objects,* each of which is a word or another list. We indicate that something is a list by enclosing it in square brackets. The following are all *lists:*

[HELLO THERE, OLD CHAP]
[X Y Z]
[HELLO]
[[HOUSE MAISON] [WINDOW FENETRE] [DOG CHIEN]]
[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:

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

The list [ ], a list with no elements, is *the empty list.* There also exists an *empty word,* which is a *word* with no elements. You type in the empty word by typing ". See entry for EMPTYP for examples of both the empty list and the empty word.

ASCII

ASCII *character* (operation)
Outputs the ASCII code for *character.* (Appendix K is a chart of all ASCII codes.) If the input word

contains more than one character, ASCII uses only its first character. See also CHAR.

Examples:

```
ASCII "B
  outputs 66
```

The procedure SECRETCODE makes a new word by using the Caesar cipher (adding 3 to each letter):

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

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

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

BUTFIRST
BF

BUTFIRST *object* (operation) short form: BF

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

Examples:

```
BUTFIRST [LAURENCE J. DAVIDSON]
  outputs [J. DAVIDSON]

BUTFIRST "DOGS
  outputs OGS

BUTFIRST [DOGS]
  outputs [ ] (the empty list)

BUTFIRST [THE DOGS]
  outputs [DOGS]
```

BUTFIRST [[THE A AN] [DOG CAT MOUSE] [BARKS MEOWS]]
  outputs [[DOG CAT MOUSE] [BARKS MEOWS]]

BUTFIRST "
  is an error.

BUTFIRST [ ]
  is also an error.

```
TO TRIANGLE :OBJECT
IF EMPTYP :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
BL

BUTLAST *object* (operation) short for BL
Outputs all but the last element of *object*.

Examples:

BUTLAST [ROBERT J. LURTSEMA]
  outputs [ROBERT J.]

BUTLAST "FLOWER
  outputs FLOWE

BUTLAST [FLOWER]
  outputs [ ]

BUTLAST [[THE A AN] [BIRD BEE FLOWER]]
  outputs [[THE A AN]]

BUTLAST "
  is an error.

BUTLAST [ ]
is also an 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
```

CHAR

CHAR *n* (operation)
Outputs the character whose ASCII code is *n*. (Appendix K is a chart of all ASCII codes.) It is an error if *n* is not the ASCII code for any character.

Characters can be *normal* (white characters on black background), *inverse video* (black characters on white background), or *flashing* (rapidly alternating between normal and inverse). The ASCII codes are organized as follows:

0- 31	CTRL characters
32- 47	punctuation
48- 57	digits
58- 63	punctuation
64- 90	upper-case letters
91- 96	punctuation
97-122	lower-case letters (upper-case on the Apple)
123-127	punctuation
128-154	inverse-video upper-case letters
155-191	inverse-video digits and punctuation
192-218	flashing upper-case letters
219-255	flashing digits and punctuation

To change a normal character to inverse video, use the expression

```
CHAR 128 + REMAINDER ASCII :CHARACTE!
R 64
```

To change a normal character to flashing, use the expression

```
CHAR 192 + REMAINDER ASCII :CHARACTE!
R 64
```

Examples:

INVERSE and FLASH display a word in either inverse video or flashing:

```
TO INVERSE :WORD
CONVERT :WORD 128
END

TO FLASH :WORD
CONVERT :WORD 192
END

TO CONVERT :WORD :NUM
IF EMPTYP :WORD [STOP]
TYPE CHAR :NUM + REMAINDER ASCII FIR!
ST :WORD 64
CONVERT BF :WORD :NUM
END

?INVERSE "YOGURT
YOGURT?
```

COUNT

COUNT *list* (operation)

Outputs the number of elements in *list*.

Examples:

COUNT [A QUICK BROWN FOX]
  outputs 4

COUNT [A [QUICK BROWN] FOX]
  outputs 3

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

The following procedure prints a random element of a list:

```
TO RANPICK :LIST
PR ITEM (1 + RANDOM COUNT :LIST) :LI!
ST
END

?RANPICK :CLASS                    (see list CLASS above)
BRIAN
```

EMPTYP

EMPTYP *object* (operation)
Outputs TRUE if *object* is the empty word or the empty list; otherwise FALSE.

Examples:

EMPTYP 3
  outputs FALSE

EMPTYP BUTFIRST "UNICORN
  outputs FALSE

EMPTYP BUTLAST "U
  outputs TRUE

EMPTYP BUTFIRST [UNICORN]
  outputs TRUE

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

?TALK [DOGS BIRDS PIGS] [BARK CHIRP O!
INK]
DOGS BARK
BIRDS CHIRP
PIGS OINK
THAT'S ALL THERE IS!
```

```
TO REVPRINT :THING
IF EMPTYP :THING [PR [] STOP]
TYPE LAST :THING
IF LISTP :THING [TYPE CHAR 32]      (space)
REVPRINT BL :THING
END

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

EQUALP

EQUALP *object1 object2* (operation)
Outputs TRUE if *object1* and *object2* are equal numbers, identical words, or identical lists; otherwise outputs FALSE. Equivalent to = (see list of infix-form operations at end of Chapter 4).

Examples:

EQUALP "RED FIRST [RED YELLOW]
  outputs TRUE

EQUALP 100 50 * 2
  outputs TRUE

EQUALP [THE A AN] [THE A]
  outputs FALSE

EQUALP " []
  outputs FALSE (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 EMPTYP :WORD [OUTPUT "FALSE]
IF EQUALP :CHAR FIRST :WORD [OUTPUT "!
TRUE]
OUTPUT INP :CHAR BUTFIRST :WORD
END

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

FIRST

FIRST *object* (operation)

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:

FIRST [HOUSE MOUSE LOUSE]
  outputs HOUSE

FIRST "HOUSE
  outputs H

FIRST [HOUSE]
  outputs HOUSE

FIRST [[THE A AN] [UNICORN RHINO] [SWIMS FLIES GROWLS RUNS]]
  outputs [THE A AN]

FIRST "
  is an error.

FIRST [ ]
  is also an error.

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

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

FPUT

FPUT *object list* (operation)

(Stands for "First PUT".) Outputs a new list formed by putting *object* at the beginning of *list*. See entry under LIST for chart comparing FPUT with other primitives that combine words and lists.

Examples:

FPUT "HAMSTER [DOG CAT]
  outputs [HAMSTER DOG CAT]

FPUT [THE A AN] [CUP GLASS]
  outputs [[THE A AN] CUP GLASS]

FPUT "A [ ]
  outputs [A]

ITEM

ITEM *n list* (operation)

Outputs the *n*th element of *list*. It is an error if *n* is greater than the length of *list* or if *list* is the empty list.

Examples:

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

LAST

LAST *object* (operation)

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

Examples:

LAST [ALISON LARRY PHIL]
outputs PHIL

LAST "VANILLA
outputs A

LAST [VANILLA]
outputs VANILLA

LAST [[THE A] FLAVOR IS [VANILLA CHOCOLATE STRAWBERRY]]
outputs [VANILLA CHOCOLATE STRAWBERRY]

LAST "
is an error.

LAST [ ]
is also an error.

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

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

LIST

LIST *object1 object2* (operation)
(LIST *object1 object2 object3 . . .*)
Outputs a list whose elements are *object1, object2,* etc.

Examples:

LIST "ROSE [TULIP CHRYSANTHEMUM]
outputs [ROSE [TULIP CHRYSANTHEMUM]]

(LIST "ROSE "TULIP "CHRYSANTHEMUM)
outputs [ROSE TULIP CHRYSANTHEMUM]

```
LIST [A QUICK BROWN FOX] [LOOKS AT THE LAZY
FROG]
  outputs [[A QUICK BROWN FOX] [LOOKS AT THE
  LAZY FROG]]

LIST "A [ ]
  outputs [A [ ]]
```

When LIST is used with a single input, parentheses around the expression are needed only if there is something else on the line following the expression:

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

The following chart compares four primitives that combine words and lists:

operation	input #1	input #2	output
FPUT	"COW	"HORSE	*error*
LIST	"COW	"HORSE	[COW HORSE]
LPUT	"COW	"HORSE	*error*
SENTENCE	"COW	HORSE	[COW HORSE]
FPUT	"LOGO	[IS WONDERFUL]	[LOGO IS WONDERFUL]
LIST	"LOGO	[IS WONDERFUL]	[LOGO [IS WONDERFUL]]
LPUT	"LOGO	[IS WONDERFUL]	[IS WONDERFUL LOGO]
SENTENCE	"LOGO	[IS WONDERFUL]	[LOGO IS WONDERFUL]
FPUT	[THE FOX]	[LOOKS AT FIDO]	[[THE FOX] LOOKS AT FIDO]
LIST	[THE FOX]	[LOOKS AT FIDO]	[[THE FOX] [LOOKS AT FIDO]]
LPUT	[THE FOX]	[LOOKS AT FIDO]	[LOOKS AT FIDO [THE FOX]]
SENTENCE	[THE FOX]	[LOOKS AT FIDO]	[THE FOX LOOKS AT FIDO]
FPUT	"COMPUTERS	[ ]	[COMPUTERS]
LIST	"COMPUTERS	[ ]	[COMPUTERS [ ]]
LPUT	"COMPUTERS	[ ]	[COMPUTERS]
SENTENCE	"COMPUTERS	[ ]	[COMPUTERS]

LISTP

LISTP *object* (operation)

Outputs TRUE if *object* is a list; otherwise FALSE.

Examples:

```
LISTP 3
  outputs FALSE

LISTP [3]
  outputs TRUE

LISTP [ ]
  outputs TRUE

LISTP "
  outputs FALSE

LISTP [A B C [D E] [F [G]]]
  outputs TRUE

LISTP BUTFIRST "CHOCOLATE
  outputs FALSE

LISTP BUTFIRST [CHOCOLATE]
  outputs TRUE
```

LPUT

LPUT *object list* (operation)

(Stands for "LAST PUT".) Outputs a new list formed by putting *object* at the end of *list*. See entry under LIST for chart comparing LPUT with other primitives that combine words and lists.

Examples:

```
LPUT "GERBIL [HAMSTER GUINEA.PIG]
  outputs [HAMSTER GUINEA.PIG GERBIL]

LPUT [THE A AN] [CAT ELEPHANT]
  outputs [CAT ELEPHANT [THE A AN]]

LPUT "A [ ]
  outputs [A]

LAST LPUT "GERBIL [HAMSTER GUINEA.PIG]
  outputs GERBIL
```

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

```
TO NEWENTRY :ENTRY
MAKE "DICTIONARY LPUT :ENTRY :DICTION!
ARY
END

?MAKE "DICTIONARY [[HOUSE CASA] [SPAN!
ISH ESPANOL] [HOW COMO]]
?SHOW :DICTIONARY
[[HOUSE CASA] [SPANISH ESPANOL] [HOW !
COMO]]
?NEWENTRY [TABLE MESA]
?SHOW :DICTIONARY
[[HOUSE CASA] [SPANISH ESPANOL] [HOW !
COMO] [TABLE MESA]]
```

MEMBERP

MEMBERP *object list* (operation)
Outputs TRUE if *object* is an element of *list*; otherwise outputs FALSE.

Examples:

MEMBERP 3 [2 5 [3] 6]
outputs FALSE

MEMBERP 3 [2 5 3 6]
outputs TRUE

MEMBERP [2 5] [2 5 3 6]
outputs FALSE

MEMBERP "BIT "RABBIT
is an error.

MEMBERP [FLORIDA GEORGIA] [[FLORIDA GEORGIA] IOWA]
outputs TRUE

MEMBERP [FLORIDA GEORGIA] [FLORIDA GEORGIA IOWA]
outputs FALSE

MEMBERP BUTFIRST "FOG [OE OF OG OH]
outputs 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)
Outputs **TRUE** if *object* is a number; otherwise **FALSE.**

Examples:

NUMBERP 3
outputs **TRUE**

NUMBERP [3]
outputs **FALSE**

NUMBERP 3.14E23
outputs **TRUE**

NUMBERP [ ]
outputs **FALSE**

NUMBERP "
outputs **FALSE**

NUMBERP BUTFIRST 3165.2
outputs **TRUE**

NUMBERP BUTFIRST [ELEPHANT]
outputs **FALSE**

SENTENCE
SE

SENTENCE *object1 object2* (operation)
(**SENTENCE** *object1 object2 object3* ...)
short form: **SE**
Outputs a list made up of the words in its inputs.

Examples:

SENTENCE "PAPER "BOOKS
  outputs [PAPER BOOKS]

SENTENCE [PAPER] [BOOKS]
  also outputs [PAPER BOOKS]

SENTENCE "APPLE [PEAR PLUM BANANA]
  outputs [APPLE PEAR PLUM BANANA]

SENTENCE [A QUICK BROWN FOX] [LOOKS AT THE LAZY FROG]
  outputs [A QUICK BROWN FOX LOOKS AT THE LAZY FROG]

```
TO FLATTER :WHO
PR SE [HELLO THERE,] :WHO
PR []
PR SE :WHO [IS A VERY NICE PERSON.]
PR SE WORD :WHO :WHO [IS DOUBLY NICE]
END
?FLATTER "BARBARA
HELLO THERE, BARBARA
BARBARA IS A VERY NICE PERSON.
BARBARABARBARA IS DOUBLY NICE
```

See entry under LIST for chart comparing SENTENCE with other primitives that combine words and lists.

Further examples:

(SENTENCE "APPLE "PEAR "BANANA)
  outputs [APPLE PEAR BANANA]

(SENTENCE "MONET)
  outputs [MONET]

SENTENCE "MONET [ ]
  also outputs [MONET]

When SENTENCE is used with a single input, parentheses around the expression are needed only if there is something else on the line following the expression:

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

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

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

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

WORD

WORD *word1 word2* (operation)
(WORD *word1 word2 word3* ...)
Outputs a word made up of its inputs.

Examples:

WORD "SUN "SHINE
  outputs SUNSHINE

(WORD "CHEESE "BURG "ER)
  outputs CHEESEBURGER

WORD "BURG [ER]
  is an error.

WORD "S "MILES
  outputs 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 into a dialect of Pig Latin:

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

TO PIG :WORD
IF MEMBERP FIRST :WORD [A E I O U] [!
OP 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 I!
GPAY ATINLAY AMONGAY UMANSHAY
?
```

WORDP

WORDP *object* (operation)

Outputs **TRUE** if *object* is a word; otherwise **FALSE**.

Examples:

WORDP 3
outputs **TRUE**

WORDP [3]
outputs **FALSE**

WORDP "ZAM
outputs **TRUE**

WORDP [E GRESS]
outputs **FALSE**

WORDP []
outputs **FALSE**

WORDP "
outputs **TRUE**

WORDP BUTFIRST "BURG
outputs **TRUE**

WORDP BUTFIRST [BURG]
outputs **FALSE**

Chapter 3

Variables

Chapter 3 Variables

A Logo word can be used as a *variable*; a variable is a "container" which holds a Logo object. This object is called the word's *value*. This can be accomplished either by using MAKE or NAME, or by using procedure *inputs*. See the Overview and the Index for a discussion of inputs.

LOCAL

LOCAL *name* (command)
(LOCAL *name1 name2* . . .)
Makes its input(s) *local* to the procedure within which the LOCAL occurs. This means that each of them is accessible only to that procedure and to procedures it calls; in this regard they resemble 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 [NI.. TO MEET YOU,\ ] :A!
NSWER
END

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

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

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

MAKE

MAKE *name object* (command)

Puts *object* in *name's* container, i.e. 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 [JANE DOE]
```

At this point :JOB is WELDER, and THING :JOB is [JANE DOE].

```
?PRINT "JOB
JOB
?PRINT :JOB
WELDER
?PRINT THING "JOB
WELDER
?PRINT THING :JOB
[JANE DOE]
```

```
TO WEATHER
PR [WHAT'S THE WEATHER LIKE TODAY?]
MAKE "ANSWER READLIST
IF :ANSWER = [RAINING] [PR [I WISH I!
T WOULD STOP RAINING] STOP]
IF :ANSWER = [SUNNY] [PR [I HOPE IT !
STAYS SUNNY] STOP]
PR (SE [I WONDER IF IT WILL BE] :ANS!
WER "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 TOMORR!
OW.
?WEATHER
WHAT'S THE WEATHER LIKE TODAY?
RAINING
I WISH IT WOULD STOP RAINING
```

NAME

NAME *object name* (command)
Puts *object* in *name's* container, i.e. 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 *name* (operation)

Outputs, TRUE if *name* has a value (i.e., if *:name* exists), FALSE otherwise.

Examples:

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

The procedure INC, listed below, shows a use of NAMEP.

THING

THING *name* (operation)

Outputs the object in the container *name*, i.e. 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 + THI!
NG :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 which is incremented.)

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

For other examples, see entry for MAKE above.

Chapter 4

Arithmetic Operations

Chapter 4 Arithmetic Operations

Logo has *integer* and *decimal* numbers:

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.

Some arithmetic operations (INT, QUOTIENT, RANDOM, REMAINDER, ROUND) always output integers; some always output decimal numbers (ARCTAN, COS, SIN, SQRT, /); some output integers if all their inputs are integers, decimal numbers otherwise (+,−,*).

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

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

The largest possible integer in Logo is 2147483647, which is $2^{31}-1$; the smallest is −2147483648, which is -2^{31}. Decimal numbers with more than six digits are converted into *exponential* form (scientific notation). For example:

2E6 means 2 times 10^6, or 2,000,000;
2.59N4 means 2.59 times 10^{-4}, or .000259
(the N indicates a negative exponent).

Exponents range from −38 to 37.

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. Addition and multiplication are also provided in *prefix* form as Logo operations taking two or more inputs. For example, the following expressions are equivalent:

```
2 + 1
SUM 2 1
```

In addition to those listed here, the primitive EQUALP is often used in conjunction with arithmetic operations. It is described in Chapter 2 (Words and Lists). The infix operation =, described in this chapter, is equivalent to EQUALP.

ARCTAN

Prefix-Form

ARCTAN *n* (operation)

Outputs the arctangent (inverse tangent) of *n*.

Examples:

```
ARCTAN 2
  outputs 63.4348

ARCTAN 444
  outputs 89.8710
```

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 *n* (operation)

Outputs the cosine of *n* degrees. It is an error if *n* is greater than 4.19E6.

Examples:

```
COS 60
  outputs .5

COS 30
  outputs .866025
```

Here is a definition of the tangent function:

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

?PR TAN 45
1.
```

INT

INT *n* (operation)

Outputs the integer portion of *n* (by removing the decimal portion, if any). See also ROUND.

Examples:

```
INT 5.2129
  outputs 5

INT 5.5129
  outputs 5

INT 5
  outputs 5

INT -5.8
  outputs -5

INT -12.3
  outputs -12
```

The procedure INTP tells whether its input is an integer:

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

```
?PRINT INTP 17
TRUE
?PRINT INTP 100 / 8
FALSE
?PRINT INTP "ONE
NOT A NUMBER
?PRINT INTP SQRT 50
FALSE
```

PRODUCT

PRODUCT *a b* (operation)
(PRODUCT *a b c . . .*)
Outputs the product of its inputs. Equivalent to * (see list of infix-form operations below).

Examples:

PRODUCT 6 2
outputs 12

(PRODUCT 2 3 4)
outputs 24

PRODUCT 2.5 4
outputs 10.

PRODUCT 2.5 2.5
outputs 6.25

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

?PR CUBE 2
8
```

With one input, PRODUCT outputs its input.

QUOTIENT

QUOTIENT *a b* (operation)
Outputs the result of dividing *a* by *b*, truncated to an integer. It is an error if *b* is 0.

Examples:

QUOTIENT 12 5
outputs 2

QUOTIENT −12 5
outputs −2

QUOTIENT 6 2.5
outputs 2

QUOTIENT 3.2 0
is an error.

RANDOM

RANDOM *n* (operation)
Outputs a random non-negative integer less than *n*.

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

```
TO D6
OUTPUT 1 + RANDOM 6
END

?PR D6
3
?PR D6
5
?PR D6
3
```

REMAINDER

REMAINDER *a b* (operation)
Outputs the remainder obtained when *a* is divided by *b*. (If *a* and *b* are integers, this is a mod *b*.) It is an error if *b* is 0.

Examples:
REMAINDER 12 10
outputs 2

REMAINDER 12 5
outputs 2

REMAINDER 12 15
outputs 12

REMAINDER −12 5
outputs −2

The following procedure tells whether its input is even:

```
TO EVENP :NUMBER
OP Ø = 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 Ø = REMAINDER :B :A
END

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

RERANDOM

RERANDOM (command)

Makes RANDOM behave reproducibly: after you run RERANDOM, calls to RANDOM output the same sequences of numbers each time.

Example:

```
TO DICE :THROWS
IF :THROWS = Ø [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
4
3
6
6
1
2
?RERANDOM
?DICE 6
4
3
6
6
1
2
```

ROUND

ROUND *n* (operation)
Outputs *n* rounded off to the nearest integer. Compare with examples under INT.

Examples:

ROUND 5.2129
outputs 5

ROUND 5.5129
outputs 6

ROUND .5
outputs 1

ROUND −5.8
outputs −6

ROUND −12.3
outputs −12

SIN

SIN *n* (operation)

Outputs the sine of *n* degrees. It is an error if *n* is greater than 4.19E6. See also COS.

Example:

```
SIN 30
  outputs .5
```

SQRT

SQRT *a* (operation)

Outputs the square root of *a*. It is an error if *a* is negative.

Examples:

```
SQRT 25
  outputs 5.

SQRT 259
  outputs 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
```

And 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) - FIRS!
T :POS2) SQ ((LAST :POS1) - LAST :PO!
S2)
END

TO SQ :N
OP :N * :N
END

?PR DISTANCE [-70 10] [50 60]
130.
```

SUM

SUM *a b* (operation)
(SUM *a b c . . .*)
Outputs the sum of its inputs. Equivalent to + (see list of infix-form operations below).

Examples:

SUM 5 2
  outputs 7

(SUM 1 3 2 −1)
  outputs 5

SUM 2.3 2.561
  outputs 4.861

With one input, SUM outputs its input.

Infix-Form

Note: Since the symbols for these operations are word-separators, spaces are optional before and after them. Thus the following are equivalent:

2 + 5
2+5

\+

a + *b* (infix-form operation)
Outputs the sum of its inputs. Equivalent to SUM (see list of prefix-form operations above).

Examples:

5 + 2
  outputs 7

1 + 3 + 2 + 1
  outputs 7

2.54 + 12.3
  outputs 14.84

a − *b* (infix-form operation)
Outputs the result of subtracting *b* from *a*. If *a* is missing and there is no space after the minus sign, it outputs the opposite of *b* (0-*b*).

Examples:

```
?PR 7 - 1
6
?PR 7-1
6
?PR PRODUCT 7 -1
-7
?PR -3
-3
?PR - 3
NOT ENOUGH INPUTS TO
?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":

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

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

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).

For more detailed information, see Appendix I (Parsing).

a * *b* (infix-form operation)

Outputs the product of its inputs. Equivalent to PRODUCT (see list of prefix-form operations above).

Examples:

6 * 2
  outputs 12

2 * 3 * 4
  outputs 24

1.3 * 1.3
  outputs 1.69

48 * .5
  outputs 24.

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

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

a / *b* (infix-form operation)

Outputs *a* divided by *b*. It is an error if *b* is 0.

Examples:

6 / 3
  outputs 2.

8 / 3
  outputs 2.66667

2.5/3.8
  outputs .657895

```
0 / 7
```
outputs 0.

```
7 / 0
```
is an error.

<

a < *b* (infix-form operation)

Outputs TRUE if *a* is less than *b;* otherwise outputs FALSE.

Examples:

```
2 < 3
```
outputs TRUE

```
-7 < -10
```
outputs FALSE

=

a = *b* (infix-form operation)

Outputs TRUE if *a* and *b* are equal numbers, identical words, or identical lists; otherwise outputs FALSE. Equivalent to EQUALP (described in Chapter 2).

Examples:

```
100 = 50*2
```
outputs TRUE

```
3 = FIRST "3.1416
```
outputs TRUE

```
[THE A AN] = [THE A]
```
outputs FALSE

```
7. = 7
```
outputs TRUE (a decimal number is equivalent to the corresponding integer)

```
" = [ ]
```
outputs FALSE (the empty word and the empty list are not identical)

`>`

a > *b* (infix-form operation)

Outputs TRUE if *a* is greater than *b;* otherwise outputs FALSE.

Examples:

```
4 > 3
```
outputs TRUE

```
-10 > -7
```
outputs FALSE

Chapter 5

Defining Procedures with TO

There are several ways to define Logo procedures. One way is with TO. See EDIT (in Chapter 6) and DEFINE (in Chapter 13) for the other ways.

The primitives COPYDEF and TEXT are also related to procedure definition. They are described in Chapter 13.

TO

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

```
?TO SQUARE :SIDE
>FD :SIDE
>RT 90
>FD :SIDE
>RT 90
>FD :SIDE
>RT 90
>FD :SIDE
>RT 90
>END
SQUARE DEFINED
?

?TO ANNOUNCE :FIRSTNAME :LASTNAME
>PRINT [WE'RE HAPPY TO ANNOUNCE THE
 BIRTH OF]
>PRINT (SE :FIRSTNAME "Q. :LASTNAME)
>PRINT [9 POUNDS 16 OZ]
>END
ANNOUNCE DEFINED
?
```

(The prompts are shown in these examples to emphasize the actions of TO and END.)

TO tells Logo that you are defining a procedure called *name,* with inputs (if any) as indicated. (Do not quote *name,* nor *input1 input2* . . . , since TO quotes them automatically.) The prompt changes from "?" to ">" to remind you that you are defining a procedure. Logo does not carry out the actions that you type; it makes them part of the procedure definition. The special word END must

be used alone on a line to stop defining the procedure and to return to Logo toplevel.

END

END (special word)
END is *necessary*, when you are using TO, to tell Logo that you are done defining the procedure. It must be on a line by itself.

Chapter 6

Defining and Editing with the Logo Editor

Chapter 6 Defining and Editing with the Logo Editor

This chapter tells you how to define and change Logo procedures using EDIT.

Logo has an interactive screen-oriented text editor; it provides a flexible way to define and change procedures. You may define more than one procedure at a time in the editor.

If you are defining a procedure for the first time rather than changing an existing definition, you can use TO, but you may still prefer EDIT.

EDIT
ED

EDIT (command) short form: ED
EDIT *name(list)*
Starts up the Logo editor. If an input is given, the editor starts up with the definition(s) of procedures *name(list)* in the edit buffer. If any procedure *name* is undefined, the edit buffer contains just the title line: TO *name*. If no input is given, the edit buffer has the same contents as the last time you used the editor, unless you have used turtle graphics or the file system since then, in which case the buffer is empty.

CTRL-C is the standard way to exit from the editor. Logo reads every line from the edit buffer as though you had typed it outside the editor. If there is a procedure definition in the editor, and the end of the buffer is reached, Logo ENDS the procedure definition.

CTRL-G aborts editing. Use it if you don't like the changes you are making or you decide not to make changes. Any changes you make in the edit buffer will be ignored. If you were defining a procedure, the definition will be the same as before

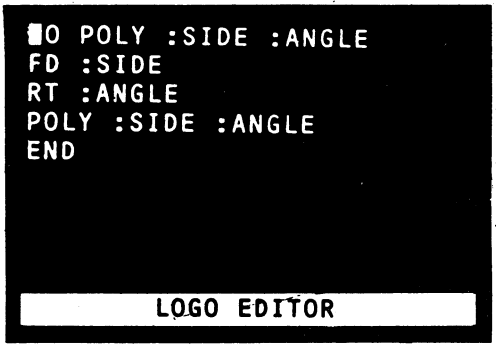
you started editing. If you type CTRL-G when you don't mean to, you can get back to your edit buffer with EDIT (no input).

When you are in the editor, you may use all the editing actions described below. Appendix B contains a quick reference table of editor actions.

An Overview of the Logo Editor

When the editor is called, the screen changes. For example:

```
?EDIT "POLY
```



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

The text that you edit is in a *buffer*. You see the text on your screen.

You can move the cursor anywhere in the edit buffer; you can change the text around, insert any text you want, and delete any text.

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

The arrow keys and many CTRL characters have special meanings that help you edit. (The Logo Editor actions are based on those of EMACS, an editor in widespread use on other computer systems.)

When you type RETURN, the cursor moves to the next line, ready for you to type more.

You can have more characters on a line of text than fit across the screen. An exclamation mark (!) is put in the rightmost character position on the line (column 39) 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!
 TO TYPE A VERY LONG MESSAGE FOR YOU]
PRINT SENTENCE [SO LONG,] :PERSON
```

The editor has an auxiliary line buffer called the kill buffer. CTRL-K deletes a whole line of text and puts it in this buffer. CTRL-Y inserts this line of text later at the place marked by the cursor.

When you exit from the editor (by typing CTRL-C) Logo reads each line in the edit buffer as if you had typed it directly to Logo.

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, they will be carried out when you exit the editor.

Editing Actions

Editing Logo Instructions Outside the Editor

When you are outside the editor you can change the line of text that you are typing (the one with the cursor on it) by using the starred editor commands below.

Cursor Motion

→
CTRL-F

* → or CTRL-F (cursor Forward) moves the cursor right (forward) one character position. (CTRL-U also has the same action.)

CTRL-B

*CTRL-B (cursor Back) moves the cursor left (back) one character position.

CTRL-N

CTRL-N (Next line) 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 at the end of the buffer, it does not move.

Example:

```
THIS IS A TEXT ▮INE          Cursor on L
THIS IS ANOTHER▮TEXT LINE    Cursor on space
A SHORTER ONE▮               Cursor at end
THIS IS A LONGE▮ ONE THAN CAN FIT ON!
THE SCREEN                   Cursor on R
THIS IS THE NEX▮ LINE        Cursor on T
```

CTRL-P

CTRL-P (Previous line) 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. If the cursor is on the first line in the buffer, it does not move.

CTRL-A

• CTRL-A (beginning of line) moves the cursor to the beginning of the current line.

CTRL-E

• CTRL-E (End of line) moves the cursor to the end of the current line (to the right of the last visible character on the line).

The following are sequences of two keystrokes. First type ESC, then the next key as specified.

ESC >

ESC > (end of buffer) moves the cursor to the end of the buffer.

ESC <

ESC < (beginning of buffer) moves the cursor to the beginning of the buffer.

Inserting and Deleting

CTRL-M

• CTRL-M is the same as RETURN.

CTRL-Q

• CTRL-Q inserts a quoting character (printed as \) in the buffer. This allows you to quote characters like space, [,], (, etc. which ordinarily have special meaning to Logo.

CTRL-O

CTRL-O (Open line) opens a new line at the position of the cursor. It is equivalent to RETURN CTRL-B.

DELETE

• The DELETE key erases the character *to the left* of the cursor. The cursor backs up one position. The ← key is the DELETE key. You should have a label marked DELETE stuck on the front of this key. (CTRL-H has exactly the same action.)

CTRL-D

• CTRL-D erases the character *at* the cursor position. Compare with DELETE.

CTRL-K

• CTRL-K (Kill to end of line) deletes text from the cursor position to the end of the current line. This text is placed in the kill buffer, which can hold up to 256 characters.

CTRL-Y

• CTRL-Y 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.

Getting Out of the Editor

CTRL-C

CTRL-C is the standard way to exit from the editor.

CTRL-G

CTRL-G aborts editing. Use it if you don't like the changes you are making or you decide not to make changes. Any changes you make in the edit buffer will be ignored. If you were defining a procedure, the definition will be the same as before you started editing. If you type CTRL-G when you don't mean to, you can get back to your edit buffer with EDIT (no input).

Scrolling the Screen

If the text in your edit buffer has too many lines to fit on the screen, you can use these commands to control what is shown on the screen. Changing what portion of the buffer is shown is called

"scrolling" because you can imagine the buffer being rolled up and down past the screen.

CTRL-V

CTRL-V (next page). Your "next page" of text is brought to the screen. The cursor is moved to the beginning of the last line on the screen, and then that line becomes the first line on the screen. If there is no new page of text (i.e. no text beyond the screen), the editor beeps and does nothing.

ESC V

ESC V (previous page) scrolls back one page. This has the opposite action from CTRL-V.

CTRL-L

CTRL-L (redisplay) scrolls the screen so that the current line is in the center of the screen. If there is less than half a screen of text in the buffer, the editor beeps and does nothing.

Note that CTRL-L has a different meaning outside of the editor. See Chapter 10 (text and screen commands).

EDNS

EDNS (command)
EDNS *package*
EDNS *package list*
(Stands for EDit NameS.) With no input, puts all variables and their values (except those in buried packages) in the edit buffer. EDNS with an input puts all the variables in the named *package(s)* in the edit buffer (even if those packages are buried). The edit buffer has the following form:

```
MAKE "VAR1 VAL1
MAKE "VAR2 VAL2
MAKE "VAR3 VAL3
.
.
.
```

These variables' names and values can then be edited. When you exit from the editor the MAKE commands are run, so whatever variables and values have been changed in the edit buffer are changed in Logo.

Example:

```
?PONS
ANIMAL IS GIBBON
SPEED IS 55
AIRCRAFT IS [JET HELICOPTER]
?EDNS
```

The screen looks like:

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

If you edit to make the following changes and exit the editor with CTRL-C:

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

Then

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

Chapter 7

Conditionals and Flow of Control

Chapter 7 Conditionals and Flow of Control

This chapter describes how you can control the order in which Logo follows instructions. It discusses *conditionals* ("if such-and-such is true, do one thing; otherwise, do something else"), *iteration* ("run a list of instructions one or more times"), *halting* ("stop this procedure"), and *pausing* ("interrupt this procedure while it's running, but let it resume afterwards").

General example of flow of control among Logo procedures:

```
TO ANSWER :QUESTION
PR DECIDE                          (calls DECIDE)
END

TO DECIDE
IF FLIP [OP "YES]                    (calls FLIP)
OP "NO
END

TO FLIP
OP 0 = RANDOM 2
END

?ANSWER [WILL MY NEXT FORTUNE COOKIE
TELL THE TRUTH?]
YES
```

In addition to those described in this chapter, the following primitives are related to conditionals and flow of control: CATCH, ERROR, GO, LABEL, THROW, which are described in Chapter 13.

CO

CO (command)
CO *object*
(Stands for "continue".) Resumes running of a procedure after a PAUSE or CTRL-Z, continuing from wherever the procedure paused. If CO has an

input, it becomes the output from PAUSE; this is particularly useful when you've interrupted a READLIST or READCHAR with a CTRL-Z, since CO's input is then read by the READLIST or READCHAR.

IF

IF *pred instructionlist1* (command or operation)
IF *pred instructionlist1 instructionlist2*
If *pred* is TRUE, runs *instructionlist1*. If *pred* is FALSE, runs *instructionlist2* (if present).

In either case, if the selected *instructionlist* outputs, then IF outputs what *instructionlist* outputs. If the list does not output, neither does IF.

Examples:
The procedure DECIDE is written 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
IFF

IFFALSE *instructionlist* (command) short form: IFF
Runs *instructionlist* if the result of the most recent TEST was FALSE, otherwise does nothing. See TEST.

Example:

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

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

IFTRUE
IFT

IFTRUE *instructionlist* (command) short form: **IFT**

Runs *instructionlist* if the result of the most recent **TEST** was **TRUE**, otherwise does nothing. See **TEST**.

Example:

```
TO QUIZ
PR [WHO IS THE GREATEST?]
TEST READLIST = [ME]
IFTRUE [PR [RIGHT ON] STOP]
PR [NO, TRY AGAIN]
QUIZ
END

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

OUTPUT
OP

OUTPUT *object* (command) short form: OP

This command is meaningful only when it is within a procedure, not at toplevel. 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 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!
Ø]

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
Ø
```

Here is one definition of the absolute-value operation:

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

(see entry for *minus* for alternate version)

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

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

?PRINT INP "A "TEACUP
TRUE
?PRINT INP "TEA "TEACUP
FALSE
?PRINT INP "I "SAUCER
FALSE
```

PAUSE

PAUSE (command or operation)

This command is meaningful only when it is within a procedure, not at toplevel. 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 toplevel, the prompt changes to the name of procedure(s) followed by a question-mark. During a PAUSE, CTRL-G does not work; the only way to return to toplevel during a pause is to run THROW "TOPLEVEL.

All local variables are accessible during a PAUSE (see PR :MAX in example below).

The procedure may be resumed by typing CO. If CO has an input, then PAUSE is an operation. CO's input becomes PAUSE's output.

Examples:

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

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

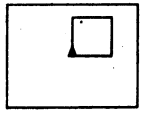
REPEAT

REPEAT *n instructionlist* (command)
Runs *instructionlist n* times. It is an error if *n* is negative. If *n* is not an integer it is truncated to an integer.

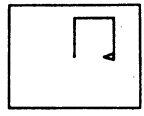
Examples:

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.



REPEAT 4
[FD 100 RT 90]



REPEAT 3
[FD 100 RT 90]

RUN

RUN *instructionlist* (command or operation)
Runs *instructionlist* as if typed in directly. If *instructionlist* is an operation, then RUN outputs whatever *instructionlist* outputs.

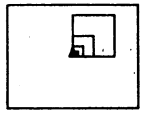
Examples:

```
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

TO WHILE :CONDITION :INSTRUCTIONLIST
TEST RUN :CONDITION
IFFALSE [STOP]
RUN :INSTRUCTIONLIST
WHILE :CONDITION :INSTRUCTIONLIST
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 EMPTYP :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]
```



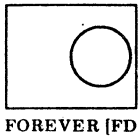
MAP "SQUARE [10 20 40 80]

```
?MAKE "NEW.ENGLAND [ME NH VT MA RI C!
T]
?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 CTRL-G):

```
TO FOREVER :INSTRUCTIONLIST
RUN :INSTRUCTIONLIST
FOREVER :INSTRUCTIONLIST
END
```

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



FOREVER [FD 1 RT 1]

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 FIRST PEN
PENDOWN
SQUARE 100
RUN SE :SAVETYPE
END

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

?SHOW PEN
[PENUP 1]
?SAFE.SQUARE
?SHOW PEN
[PENUP 1]
```

RUN READLIST runs any commands typed in by the user.

PRINT RUN READLIST prints the output from any expression typed in by the user.

STOP

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 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!
```

TEST

TEST *pred* (command)
Remembers whether *pred* is TRUE or FALSE for subsequent use by IFTRUE or IFFALSE. Each TEST is *local* to the procedure in which it occurs.

Example:

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

?SHORTQUIZ
HOW ARE YOU?
LOUSY

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

Special CTRL characters

CTRL-G

CTRL-G (special character)
Immediately stops whatever is running, returns Logo to top level, and types a question-mark. May be typed at any time.

CTRL-W

CTRL-W (special character)
Interrupts whatever is running. Typing any character other than CTRL-W resumes normal running. This character is particularly useful to give yourself time to read when more than one screenful of information is being printed out.

CTRL-Z

CTRL-Z (special character)
Interrupts whatever is running, causing a PAUSE. Equivalent in effect to PAUSE, but different in how it is used: CTRL-Z is typed at the keyboard during the running of a procedure; PAUSE is part of the definition of a procedure.

Chapter 8

Logical Operations

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

There are some Logo predicates whose *inputs* must be TRUE or FALSE. These are called *logical operations*. Their names do not end in P. The designers of Apple Logo have chosen to retain the traditional names AND, OR, and NOT.

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

Predicate	*Chapter*
BUTTONP	9
DEFINEDP	13
EMPTYP	2
EQUALP	2
KEYP	9
LISTP	2
MEMBERP	2
NAMEP	3
NUMBERP	2
PRIMITIVEP	13
SHOWNP	1
WORDP	2
<	4
=	4
>	4

AND

AND *pred1 pred2* (operation)
(AND *pred1 pred2 pred3* . . .)
Outputs TRUE if all its inputs are true, FALSE otherwise.

Examples:

```
AND "TRUE "TRUE
  outputs TRUE
```

```
AND "TRUE "FALSE
  outputs FALSE

AND "FALSE "FALSE
  outputs FALSE

(AND "TRUE "TRUE "FALSE "TRUE)
  outputs FALSE

AND 5 7
  is an error.

AND PENCOLOR=0 BACKGROUND=0 (when you start up the turtle)
  outputs FALSE
```

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

```
TO DECIMALP :OBJ
OUTPUT AND NUMBERP :OBJ HASDOTP :OBJ
END

TO HASDOTP :WORD
IF EMPTYP :WORD [OUTPUT "FALSE]
IF ". = FIRST :WORD [OUTPUT "TRUE]
OUTPUT HASDOTP BUTFIRST :WORD
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 Fahrenheit):

```
TO COMFORT
IF AND :TEMPERATURE > 50 :TEMPERATUR!
E < 90 [PR "DELIGHTFUL] [PR "UNPLEAS!
ANT]
END
```

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

NOT

NOT *pred* (operation)
Outputs TRUE if *pred* is FALSE; outputs FALSE if *pred* is TRUE.

Examples:

NOT EQUALP "A "B
  outputs TRUE

NOT EQUALP "A "A
  outputs FALSE

NOT "A = FIRST "DOG
  outputs TRUE

NOT "A
  is an 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 FIRST PEN
TRUE
```

OR

OR *pred1 pred2* (operation)
(OR *pred1 pred2 pred3* . . .)
Outputs FALSE if all its inputs are false, TRUE otherwise.

Examples:

OR "TRUE "TRUE
  outputs TRUE

OR "TRUE "FALSE
  outputs TRUE

OR "FALSE "FALSE
  outputs FALSE

(OR "FALSE "FALSE "FALSE "TRUE)
  outputs TRUE

OR 5 7
  is an 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!
180-HEADING]
SUBMOUNTAIN
END
```

MOUNTAINS

Chapter 9

The Outside World

Chapter 9 The Outside World

This chapter describes primitives for communicating to the computer through the keyboard, for communicating through special-purpose peripherals such as game paddles, and for telling the computer to display information on the TV screen.

In addition to those described in this chapter, the primitives CURSOR and SETCURSOR are related to communication with the outside world. They are described in Chapter 10.

BUTTONP

BUTTONP *paddlenumber* (operation)
Outputs TRUE if button on specified paddle is down, FALSE otherwise (*paddlenumber* must be 0, 1, or 2; BUTTONP 3 is an error, since paddle #3 has no button). If nothing is plugged into the game I/O socket, BUTTONP outputs TRUE.

KEYP

KEYP (operation)
Outputs TRUE if there is at least one character waiting to be read (i.e., one that has been typed on the keyboard and not yet picked up by READCHAR or READLIST), FALSE if there isn't any.

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
```

PADDLE

PADDLE *paddlenumber* (operation)
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
```

PRINT
PR

PRINT *object* (command) short form: PR
(PRINT *object1 object2* . . .)
Prints its input(s) on the screen, followed by a RETURN. 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!

?
```

READCHAR
RC

READCHAR (operation) short form: RC

Outputs the first character typed. This character can even be a CTRL character, except for CTRL-G and CTRL-Z. If no character is waiting to be read, READCHAR waits until the user types something. See also KEYP.

Example:

The following procedure, XYZZY, lets the user run certain commands with a single keystroke (F does FORWARD 5, and R does RIGHT 10—you can add to the list). No RETURN is needed.

```
TO XYZZY
INTERPRET READCHAR
XYZZY
END

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

READLIST
RL

READLIST (operation) short form: RL

Waits for the user to type a line, and outputs that line in the form of a list. If lines have already been typed, it outputs the first line that has been typed but not read.

Examples:

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

?GET.USER
WHAT IS YOUR NAME?
LARRY
WELCOME TO LOGO, LARRY
?GET.USER
WHAT IS YOUR NAME?
LARRY DAVIDSON
WELCOME TO LOGO, LARRY DAVIDSON

TO AGE
.PR [HOW OLD ARE YOU?]
PR MESSAGE FIRST RL
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
```

SHOW

SHOW*object* (command)

Prints *object* on the screen, followed by a RETURN. If *object* is a list it is printed with 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 . . .*)
Prints its input(s) on the screen, not followed by a RETURN. 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 "\                    (CTRL-Q followed by a space)
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? 100
```

WAIT

WAIT *n* (command)
Tells Logo to wait for *n* 60ths of a second.

Example:
The procedure REPORT keeps printing the turtle's position as it moves randomly. It uses WAIT to give the user 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
```

Chapter 10

Text and Screen Commands

Chapter 10 Text and Screen Commands

The Apple has 24 lines of text on the screen, with 40 characters on each line. Your screen can be used entirely for text or entirely for graphics; the Apple also lets you use the top 20 lines for graphics and the bottom 4 for text at the same time. When you start up Logo, the entire screen is available for text.

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

1. regular Logo commands, which can be typed at top level or inserted within procedures (FULLSCREEN, SPLITSCREEN, TEXTSCREEN);

2. special CTRL characters, which are read from the keyboard and obeyed almost immediately (while a procedure continues running); these may not be placed within procedures (CTRL-L, CTRL-S, CTRL-T).

In addition to those described in this chapter, the primitives SCRUNCH and SETSCRUNCH are related to text and screen commands. They are described in Chapter 1.

CLEARTEXT

CLEARTEXT (command)
Clears the entire text screen and puts the cursor at the upper left corner of the text part of the screen. If you have been using the turtle screen, the cursor is on the fourth line from the bottom.

CURSOR

CURSOR (operation)
Outputs a list of the column and line numbers of the cursor position. The upper-left corner of the screen is [0 0], and the upper right is [39 0]. See SETCURSOR.

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    CHAR 32 is space
IF (REMAINDER FIRST CURSOR 8)>Ø [TAB]
END

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
```

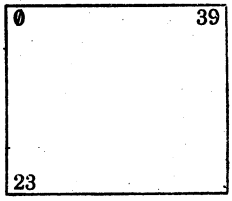
FULLSCREEN

FULLSCREEN (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 type an error message while you are in FULLSCREEN, it automatically runs a SPLITSCREEN.

SETCURSOR

SETCURSOR *position* (command)
Sets the cursor to *position*. The first element of position is the column number; the second, the line number. Lines on the screen are numbered from Ø to 23, character positions (columns) from Ø to 39.



It is an error if the line number is not between 0 and 23, or if the column number is not between 0 and 39. If an element of *position* is a decimal number, it is truncated to an integer.

Examples:

SETCURSOR [39 12] puts the cursor half-way down the right edge of the screen.

```
TO MOVECURSOR :X :Y
SETCURSOR LIST (:X + FIRST CURSOR) (:
:Y + LAST CURSOR)
END

?PRINT "A MOVECURSOR 2 5 PRINT "B
```

SPLITSCREEN

SPLITSCREEN (command)
Devotes the top 20 lines of the screen to the turtle field, with the bottom four lines reserved for text.

TEXTSCREEN

TEXTSCREEN (command)
Devotes the entire screen to text; the turtle field will be invisible to you until a graphics procedure is run.

Special CTRL Characters

CTRL-L

CTRL-L (special character)
Similar in effect to FULLSCREEN. May be typed at any time.

This has no effect if you have used the editor or the file system after your most recent graphics instruction, or if you haven't used graphics at all since starting up Logo.

Note that CTRL-L has a different meaning in the editor. See Chapter 6 (Defining and Editing with the Logo Editor).

CTRL-S

CTRL-S (special character)
Similar in effect to SPLITSCREEN. May be typed at any time.

This has no effect if you have used the editor or the file system after your most recent graphics instruction, or if you haven't used graphics at all since starting up Logo.

CTRL-T

CTRL-T (special character)
Similar in effect to TEXTSCREEN. May be typed at any time.

Chapter 11

Workspace Management

Chapter 11 Workspace Management

Your *workspace* comprises the variables, procedures, and properties that Logo knows about right now. It does not include primitives.

There are several primitives that let you see what you have in your workspace. You can also selectively erase variables and procedures from your workspace.

A *package* is a collection of procedures and variables. Many workspace management primitives can treat the entire package as a single unit.

In addition to those described in this chapter, the primitives NODES and PPS are related to workspace management. They are described in Chapter 13.

BURY

BURY *package* (command)
Buries all procedures and names in *package* (see PACKAGE). Certain commands (ERALL, ERNS, ERPS, POALL, PONS, POPS, POTS, SAVE), when used without a package name input, act on everything in the workspace except procedures and variables in buried packages. BURY works by putting the BURY property with value TRUE on the property list associated with the name of *package* (see property list section in Chapter 13).

Example:
SAVE "GOODSTUFF saves the whole workspace in the file GOODSTUFF.LOGO except procedures and variables in buried packages.

ERALL

ERALL (command)
ERALL *package*
ERALL *packagelist*
Erases all procedures and variables (in *package* or *packagelist*) from the workspace. See BURY for exceptions.

ERASE
ER

ERASE *name* (command) short form: ER
ERASE *namelist*
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* (command)
ERN *namelist*
(Stands for "ERase Name".) Erases the named variable(s) from the workspace.

Examples:
ERN "LENGTH erases the LENGTH variable.
ERN [LENGTH PI] erases the LENGTH and PI variables.

ERNS

ERNS (command)
ERNS *package*
ERNS *packagelist*
(Stands for "ERase NameS".) Erases all variables (in *package* or *packagelist*) from the workspace. See BURY for exceptions.

ERPS

ERPS (command)
ERPS *package*
ERPS *packagelist*
(Stands for "ERase Procedures".) Erases all procedures (in *package* or *packagelist*) from the workspace. See BURY for exceptions.

PACKAGE

PACKAGE *package name* (command)
PACKAGE *package namelist*
Puts each named procedure in *package*. It does so by putting the PROCPKG property with value *package* on the property list associated with the name of each procedure (see property list section in Chapter 13).

Example:

```
PACKAGE "SHAPES [TRIANGLE SQUARE]
```

puts TRIANGLE and SQUARE in the package SHAPES.

PKGALL

PKGALL *package* (command)
Puts into *package* all procedures and variables that are not already in packages. See PACKAGE.

PO

PO *name* (command)
PO *namelist*
(Stands for "Print Out".) Prints the definitions 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)

POALL *package*

POALL *packagelist*

(Stands for "Print out ALL".) Prints the definition of every procedure and the value of every variable (in *package* or *packagelist*). See BURY for exceptions.

Example:

```
?POALL
TO POLY :SIDE :ANGLE
FD :SIDE
RT :ANGLE
POLY :SIDE :ANGLE
END

TO LENGTH :OBJ
IF EMPTYP :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

ANIMAL IS AARDVARK
LENGTH IS 3.98
MYNAME IS LARRY
```

POLY, SPI, and LENGTH are in package SHAPES.

```
?POALL "SHAPES
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

LENGTH IS 3.98
```

PONS

PONS (command)

PONS *package*

PONS *packagelist*

(Stands for "Print Out Names".) Prints the name and value of every variable (in *package* or *packagelist*). See BURY for exceptions.

Example:

```
?PONS "LANG
F IS 3
LIST IS [A B C]
```

POPS

POPS (command)

POPS *package*

POPS *packagelist*

(Stands for "Print Out Procedures".) Prints the definition of every procedure (in *package* or *packagelist*). See BURY for exceptions.

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
```

POTS

POTS (command)
POTS *package*
POTS *packagelist*
(Stands for "Print Out Titles".) Prints the title line of every procedure (in *package* or *packagelist*). See BURY for exceptions.

Examples:

```
?POTS
TO POLY :SIDE :ANGLE
TO LENGTH :OBJ
TO GREET
TO SPI :SIDE :ANGLE :INC

?POTS "SHAPES
TO POLY :SIDE :ANGLE
TO SPI :SIDE :ANGLE :INC

?POTS [SHAPES LANG]
TO POLY :SIDE :ANGLE
TO SPI :SIDE :ANGLE :INC
TO LENGTH :OBJ
```

UNBURY

UNBURY *package* (command)
Unburies all procedures and names in *package*. (See BURY.)

Chapter 12

Files

You can store your workspace on a diskette. The information is organized in *files;* you decide what should go into each file*.

Note that SAVE and LOAD can take either one or two inputs. If you are using one of them with only one input, you should enclose the expression in parentheses if there is another command on the same line following it.

CATALOG

CATALOG (command)
Prints on the screen the names of all the files on the disk.

DISK

DISK (operation)
Outputs a list of three numbers: the disk drive number, the slot number of the disk drive, and the volume number of the disk which you have most recently used for CATALOG or set with SETDISK. For more information, see the Apple DOS Manual.

ERASEFILE

ERASEFILE *file* (command)
Erases the file named *file*.LOGO from the disk. It is an error if there is no such file.

Example:

```
?ERASEFILE "BEAR
```

erases file named BEAR.LOGO

LOAD

LOAD *file* (command)
LOAD *file package*
Loads the contents of *file*.LOGO into the workspace, as if typed in directly. It is an error if *file* doesn't exist. CTRL-G does not interrupt LOAD.

All variables and procedures go into the packages they came from, unless LOAD has a second input.

*When you use a file command, the graphics screen or edit buffer is erased.

If there is a second input, it specifies the *package* that everything goes into. If anything in the file is in a buried package, it is buried after being loaded into the workspace.

Examples:

?LOAD "BEAR
(everything in the file named BEAR.LOGO is read into the workspace; each variable or procedure is in the package it came from)

?LOAD "POLYS "SHAPES
(everything in the file named POLYS.LOGO is read into the workspace in package SHAPES)

SAVE

SAVE *file* (command)
SAVE *file package*
SAVE *file packagelist*
Creates a file named *file*.LOGO, and saves in it all procedures and variables in the named package(s), even if they are buried, including their properties. If there is no second input, the entire workspace is saved, except buried packages. If the file already exists, you must first erase it with ERASEFILE; then SAVE can re-use that filename.

File names longer than thirty characters are truncated to thirty characters.

SETDISK

SETDISK *drive* (command)
SETDISK *drive slot*
SETDISK *drive slot volume*
Tells Logo to set the disk drive, slot for the disk drive, and disk volume to the numbers given as inputs. The volume, an integer from 1 through 254, can be used to identify a diskette and to keep yourself from accidentally writing on the wrong one.

Chapter 13

Advanced or Rarely Used Primitives

Chapter 13 Advanced or Rarely Used Primitives

This chapter contains a wide variety of primitives that are either advanced or rarely used (or both). It is divided into four sections:

1. property lists
2. error handling, CATCH, and THROW
3. procedure definition and redefinition: further techniques
4. miscellaneous primitives

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 the name of a property (such as I.D.) and its value (such as FREDDY). For example:

```
[I.D. FREDDY COLOR GREEN LEGS 4]
```

Note that a property list has the form [PROP1 VAL1 PROP2 VAL2 ...]. You can manipulate property lists using the primitives in this section.

In addition, there are five other primitives which have effects or side effects that involve property lists: BURY, UNBURY, PACKAGE, PKGALL, LOAD.

BURY gives a package the BURY property with value TRUE. UNBURY removes the BURY property from the package. These primitives are described in Chapter 11 (Workspace Management).

PACKAGE *package name(s)* gives the named procedures the property PROCPKG with the value *package*. PKGALL *package* gives all unpackaged

procedures the property PROCPKG with the value *package;* it also gives all unpackaged variables the property VALPKG with the value *package.* These primitives also are described in Chapter 11 (Workspace Management).

LOAD *file package* gives the names of the procedures in the file the PROCPKG property with value *package;* it also gives the names of variables in the file the VALPKG property with value *package.* This primitive is described in Chapter 12 (Files).

GPROP

GPROP *name prop* (operation)
(Stands for "GET PROPerty".) Outputs the value of the *prop* property of *name;* outputs the empty list if there is no such property.

Example:

```
?SHOW GPROP "FROG "I.D.
FREDDY
?SHOW GPROP "AIDS "ANY
[]
?SHOW GPROP ".SYSTEM "BURY
TRUE
```

PLIST

PLIST *name* (operation)
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 . . .].

Examples:

```
?SHOW PLIST "FROG
[I.D. FREDDY COLOR GREEN LEGS 4]
?SHOW PLIST ".SYSTEM
[BURY TRUE]
```

PPROP

PPROP *name prop object* (command)
(Stands for "PUT PROPerty".) Gives *name* the property *prop* with value *object.* (Note that

ERALL, which erases procedure definitions and variables, does not reclaim the space used by properties created by PPROP. Use REMPROP to erase properties.)

Example:

```
?SHOW PLIST "BIRD
[I.D. PHOENIX LEGS 2]
?PPROP "BIRD "SIZE [VERY BIG]
?SHOW PLIST "BIRD
[SIZE [VERY BIG] I.D. PHOENIX LEGS 2]
```

PPS

PPS (command)
PPS *package*
PPS *packagelist*
(Stands for "Print Properties".) Prints the property list(s) of everything in the named package(s). With no input, prints the property lists of everything in the workspace.

Example:

```
?PPS "CHARACTERS
BIRD'S I.D. IS PHOENIX
BIRD'S COLOR IS YELLOW
BIRD'S LEGS IS 2
BIRD'S SIZE IS [VERY BIG]
FROG'S I.D. IS FREDDY
FROG'S COLOR IS GREEN
FROG'S LEGS IS 4
```

REMPROP

REMPROP *name prop* (command)
Removes property pair *prop* from the property list of *name.*

Example:

```
?SHOW PLIST "SHIP
[SPEED 50 HEADING 40 COLOR GREEN]
?REMPROP "SHIP "HEADING
?SHOW PLIST "SHIP
[SPEED 50 COLOR GREEN]
```

See also PPROP and GPROP.

Error Handling, CATCH, and THROW

CATCH

CATCH *name instructionlist* (command)

Runs *instructionlist.* If a THROW *name* is called while *instructionlist* is run, control returns to 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 are two special cases. CATCH "TRUE catches any THROW; CATCH "ERROR catches an error which would otherwise print an error message and return to toplevel. If an error is caught, the message which Logo would normally print isn't printed. See ERROR to find out how to tell what the error was. Note: CTRL-G does not work within a CATCH "ERROR, but THROW "TOPLEVEL does.

Examples:

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

```
TO SNAKE
CATCH "NOTNUM [SLITHER]
SNAKE
END

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

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

(Notice that STOP would have returned to SLITHER, not to SNAKE)

2. The procedure DOIT runs instructions typed in by the user. When an error occurs, Logo does not type the standard error message and does not return to toplevel; instead, it types THAT STATEMENT IS INCORRECT and lets the user continue to type in instructions.

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

TO DOIT1
RUN READLIST
DOIT1
END

?DOIT
PR 3 + 5
8
PR12 - 7
THAT STATEMENT IS INCORRECT
PR 12 - 7
5
THROW "TOPLEVEL
?
```

ERROR

ERROR (operation)

Outputs a six-element list containing information about the most recent error which has not had a message printed or output by ERROR (or the empty list if there was no such error):

- a unique number identifying the error (the numbers are listed in Appendix E);
- a message explaining the error;
- the name of the procedure within which the error occurred (the empty list, if top level);
- the line where the error occurred;
- the name of the primitive causing the error, if any;
- the object causing the error, if any.

Logo runs THROW "ERROR whenever an error occurs, unless :ERRACT is TRUE. Control passes to toplevel 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.

If :ERRACT is TRUE, an error causes a *pause* (see PAUSE).

Example:

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

?SAFESQUARE "SIXINCHES
41 [FORWARD DOESN'T LIKE SIXINCHES A!
S INPUT] SAFESQUARE [CATCH "ERROR [R!
EPEAT 4 [FD :SIDE RT 90]]] FORWARD S!
IXINCHES
```

THROW

THROW *name* (command)

This command is meaningful only within the range of a CATCH *name* command. See CATCH. It is an error if no corresponding CATCH *name* is found.

THROW "TOPLEVEL returns control to toplevel. (Contrast with STOP.)

Procedure definition and redefinition: further techniques

Procedures can be represented as lists as shown in the examples below. You can manipulate these list representations.

COPYDEF

COPYDEF *newname name* (command)

Copies the definition of *name*, making it the definition of *newname* as well. The redefinition is not part of your workspace, and cannot be SAVEd on a file.

Examples:

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

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

Neither *name* nor *newname* may be names of Logo primitives unless :REDEFP is TRUE. If :REDEFP is TRUE and *name* is a primitive, then *newname* is a primitive as well.

DEFINE

DEFINE *name list* (command)

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

defines the same procedure as

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

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

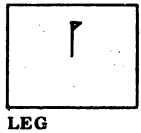
If *name* has no inputs, this must be the empty list. Each subsequent element is a list consisting of one line of the procedure definition. (This list does not contain END, since END is not part of the procedure definition.)

The second input to DEFINE has the same form as the output from TEXT. Note that *name* cannot be the name of a Logo primitive unless :REDEFP is TRUE (see Appendix J).

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 F!
IRST READLIST :PRO]
END

TO READLINES
MAKE "NEXTLINE READLIST
IF :NEXTLINE = [END] [STOP]
TEST :NEXTLINE = [ERASE]
IFTRUE [CANCEL]
IFFALSE [RUN :NEXTLINE MAKE "PRO LPU!
T :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 DEFI!
NITION OF A PROCEDURE?
YES
PROCEDURE NAME?LEG

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



LEG

PRIMITIVEP

PRIMITIVEP *name* (operation)
Outputs TRUE if *name* is the name of a primitive, FALSE otherwise.

Examples:
PRIMITIVEP "FORWARD outputs TRUE
PRIMITIVEP "SQUARE outputs FALSE

TEXT

TEXT *name* (operation)
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] [!
POLY :SIDE :ANGLE]]
```

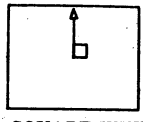
The first element of the output is a list of the names of the procedure's inputs. The rest of the elements are lists; each one is a line in the proce-

dure definition. (If the procedure *name* is undefined, TEXT outputs the empty list.) The example above corresponds to:

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

TEXT can be used in conjunction with DEFINE to create procedures that modify other procedures. For example:

```
?PO "SQUARE
TO SQUARE
REPEAT 4 [FD 30 RT 90]
END
?DEFINE "SQUARE.WITH.TAIL LPUT [FD 1!
00] TEXT "SQUARE


?PO "SQUARE.WITH.TAIL
TO SQUARE.WITH.TAIL
REPEAT 4 [FD 30 RT 90]
FD 100
END
```



SQUARE.WITH.TAIL

Complex example:
The procedure STEP modifies the definition of a procedure to make it run one line at a time; after each line is run, Logo waits for you to type RETURN before it proceeds. UNSTEP restores the original procedure definition.

The Program:

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

TO IGNORE :INPUT
END

TO STEPPER
TYPE "
IGNORE READLIST
END

TO SHOWLINES :INSTRUCTIONS
IF EMPTYP :INSTRUCTIONS [STOP]
MAKE "NEWDEF LPUT (LIST "TYPE FIRST !
:INSTRUCTIONS) :NEWDEF
MAKE "NEWDEF LPUT [STEPPER] :NEWDEF
MAKE "NEWDEF LPUT FIRST :INSTRUCTION!
S :NEWDEF
SHOWLINES BF :INSTRUCTIONS
END

TO SHOWINPUTS :ARGLIST
IF EMPTYP :ARGLIST [STOP]
MAKE "NEWDEF LPUT (LIST "PRINT "SENT!
ENCE (LIST (FIRST :ARGLIST) "IS) (WO!
RD ": FIRST :ARGLIST)) :NEWDEF
SHOWINPUTS BF :ARGLIST
END

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

Using the Program:

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

?STEP "TRIANGLE
?TRIANGLE "IT
ENTERING TRIANGLE
WORD IS IT
IF EMPTYP :WORD [STOP]
                              You press RETURN
PR :WORD                      You press RETURN
IT
TRIANGLE BL :WORD             You press RETURN
ENTERING TRIANGLE
WORD IS I
IF EMPTYP :WORD [STOP]
                              You press RETURN
PR :WORD                      You press RETURN
I
TRIANGLE BL :WORD             You press RETURN
ENTERING TRIANGLE
WORD IS
IF EMPTYP :WORD [STOP]
?
```

Miscellaneous primitives

DEFINEDP

DEFINEDP *word* (operation)
Outputs TRUE if *word* is the name of a procedure, FALSE otherwise.

GO

GO *word* (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
```

LABEL

LABEL *name* (command)
GO *name* passes control to the instruction following LABEL *name*. See GO.

NODES

NODES (operation)
Outputs the number of free nodes. This gives you an idea of how much space you have in your workspace for procedures, variables, and the running of procedures. NODES is most useful if run immediately after RECYCLE. See Appendix H (Space).

RECYCLE

RECYCLE (command)
Performs a *garbage collection*, freeing up as many nodes as possible. (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.) See NODES. See Appendix H (Space).

REPARSE

REPARSE (command)
When you define or erase a procedure, Logo may have to change the interpretation of many other procedures in your workspace. Like garbage collection (see RECYCLE above), this *reparsing* happens automatically, so you don't have to worry

about it. But, also like garbage collection, the automatic reparsing may slow down your program at an awkward time. The REPARSE command can be used to make all needed reparsing happen right away. After you use it, you are safe from further reparsing at least until the next time you define or erase a procedure.

.BPT

.BPT (command)
Enters the Apple monitor. Logo may be resumed by typing 803G followed by RETURN. The name of this primitive starts with a dot to warn you that it is dangerous. You should save your work before you use it.

.CONTENTS

.CONTENTS (operation)
Outputs a list of all of the objects that Logo "knows about"; this includes your variables and procedures, the Logo primitives, most of the things you've typed in, and some other words. .CONTENTS can use up a lot of node space. The name of this primitive starts with a dot to warn you that it is dangerous. You should save your work before you use it.

.DEPOSIT

.DEPOSIT *n a* (command)
Writes *a* into machine address *n* (decimal). The name of this primitive starts with a dot to warn you that it is dangerous. You should save your work before you use it.

.EXAMINE

.EXAMINE *obj* (operation)
Outputs the contents of machine address *obj* (decimal), if *obj* is a number. Otherwise outputs the address of the first node of *obj*. The name of this

primitive starts with a dot to warn you that it is dangerous. You should save your work before you use it.

.PRINTER

.PRINTER *n* (command)

Tells Logo that any information subsequently displayed on the screen is to be printed on the printer. If *n* is in the range 1 through 7, then *n* is the number of the *slot* which the printer is plugged into. (Warning: if there is no printer plugged into slot *n*, Logo crashes.) If *n* is in the range 9 through 15, then *n-8* is the number of the *slot* which the printer is plugged into, and the text is also copied onto the screen as it is printed. Whatever the value of *n* is, error messages are always copied onto the screen.

The name of this primitive starts with a dot to warn you that it is dangerous. You should save your work before you use it.

If *n* is 0, printing returns to the screen only. If *n* is 6, the Apple boots Logo from the disk.

Example:

The following sequence of instructions prints the definitions of all procedures in the workspace on the printer plugged into slot #7. It displays them on the screen as well. When it is done, printing returns to the screen only.

```
?.PRINTER 15
?POPS
...
?.PRINTER 0
```

Appendices

Appendix A Logo Vocabulary

Note: Parentheses around an input indicate that the input is optional. A number sign (#) indicates a procedure which can take any number of inputs; if you give it other than the number indicated, you must enclose the entire expression in parentheses.

Procedures and Their Inputs

Turtle Graphics

BACK, BK *distance*
BACKGROUND, BG
CLEAN
CLEARSCREEN, CS
DOT *position*
FENCE
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*
SCRUNCH
SETBG *colornumber*
SETHEADING, SETH *degrees*
SETPC *colornumber*
SETPEN *pair*
SETPOS *position*
SETSCRUNCH *n*
SETX *x*
SETY *y*
SHOWNP
SHOWTURTLE, ST
TOWARDS *position*
WINDOW
WRAP
XCOR
YCOR

Words and Lists

ASCII *char*
BUTFIRST, BF *object*
BUTLAST, BL *object*
CHAR *n*
COUNT *list*
EMPTYP *object*
EQUALP *object1 object2*
FIRST *object*
FPUT *object list*
ITEM *n object*
LAST *object*
LIST *object1 object2*
LISTP *object*
LPUT *object list*
MEMBERP *object list*
NUMBERP *object*
SENTENCE, SE *object1 object2*
WORD *word1 word2*
WORDP *object*
object1 = *object2*

Variables

LOCAL *name*
MAKE *name object*
NAME *object name*
NAMEP *word*
THING *name*

Arithmetic Operations

ARCTAN *n*
COS *degrees*
INT *n*
PRODUCT *a b*
QUOTIENT *a b*
RANDOM *n*
REMAINDER *a b*
RERANDOM
ROUND *n*
SIN *degrees*
SQRT *a*
SUM *a b*
a + *b*
a − *b*
a * *b*
a / *b*
a < *b*
a = *b*
a > *b*

Defining and Editing

EDIT, ED *(name(s))*
EDNS *(package(list))*
TO *procname (inputs)*

Conditionals and Flow of Control

CO *(object)*
IF *pred list1 (list2)*
IFFALSE, IFF *list*
IFTRUE, IFT *list*
OUTPUT, OP *object*
PAUSE
REPEAT *n list*
RUN *list*
STOP
TEST *pred*

Logical Operations

AND *pred1 pred2*
NOT *pred*
OR *pred1 pred2*

The Outside World

BUTTONP *paddlenumber*
KEYP
PADDLE *paddlenumber*
PRINT, PR *object*
READCHAR, RC
READLIST, RL
SHOW *object*
TYPE *object*
WAIT *n*

Text and Screen Commands

CLEARTEXT
CURSOR
FULLSCREEN
SETCURSOR *position*
SPLITSCREEN
TEXTSCREEN

Workspace Management

BURY *package*
ERALL *(package(list))*
ERASE, ER *name(s)*
ERN *name(s)*
ERNS *(package(list))*
ERPS *(package(list))*
PACKAGE *package name(s)*
PKGALL *package*
PO *name(s)*
POALL *(package(list))*

PONS *(package(list))*
POPS *(package(list))*
POTS *(package(list))*
UNBURY *package*

Files
CATALOG
DISK
ERASEFILE *name*
LOAD *file (package)*
SAVE *file (package(list))*
SETDISK *drive (slot) (vol)*

Property lists
GPROP *name prop*
PLIST *name*
PPROP *name prop object*
PPS *(package(list))*
REMPROP *name prop*

Error handling, CATCH, **and** THROW
CATCH *name list*
ERROR
THROW *name*

Further procedure definition and redefinition
COPYDEF *newname name*
DEFINE *procname list*
PRIMITIVEP *name*
TEXT *procname*

Miscellaneous Procedures
DEFINEDP *word*
GO *word*
LABEL *word*
NODES
RECYCLE
REPARSE
.BPT
.CONTENTS
.DEPOSIT *n a*
.EXAMINE *n*
.PRINTER *slot*

Editing Commands

*← OR DELETE
*←
*CTRL-A
*CTRL-B
CTRL-C
*CTRL-D
*CTRL-E
*CTRL-F
*CTRL-H
*CTRL-K
CTRL-L
*CTRL-M
CTRL-N
CTRL-O
CTRL-P
*CTRL-U
CTRL-V
*CTRL-Y
ESC V
ESC <
ESC >

Magic Words

END
ERRACT
ERROR
FALSE
PROCPKG
REDEFP
STARTUP
TOPLEVEL
TRUE
VALPKG
XYZZY
.SYSTEM

Special Characters

CTRL-G
CTRL-Q
CTRL-S
CTRL-T
CTRL-W
CTRL-Z

An asterisk (*) indicates an editing command which works both inside and outside of the editor.

Appendix B Glossary

Primitives (Prefix Form)

Note: Parentheses around an input indicate that the input is optional. A number sign (#) indicates a procedure which can take any number of inputs; if you give it other than the number indicated, the entire expression must be enclosed in parentheses.

'AND *pred1 pred2*	Outputs TRUE if all its inputs are TRUE.
ARCTAN *n*	Outputs the arctangent of *n*.
ASCII *char*	Outputs ASCII code for *char*.
BACK, BK *n*	Moves turtle *n* steps back.
BACKGROUND, BG	Outputs number representing background color.
BURY *pkg*	Buries all procedures contained in *pkg*.
BUTFIRST, BF *obj*	Outputs all but first element of *obj*.
BUTLAST, BL *obj*	Outputs all but last element of *obj*.
BUTTONP *n*	Outputs TRUE if button on paddle *n* is down.
CATALOG	Displays names of all files on diskette.
CATCH *name list*	Runs *list*; returns when THROW *name* is run.
CHAR *n*	Outputs character whose ASCII code is *n*.
CLEAN	Erases graphics screen without affecting turtle.
CLEARSCREEN, CS	Erases screen, moves turtle to [0 0], sets heading to 0.
CLEARTEXT	Clears text screen.
CO	Resumes a procedure after a pause.
COPYDEF *newname name*	Copies definition of *name* onto *newname*.
COS *n*	Outputs cosine of *n* degrees.

COUNT *list*	Outputs the number of elements in *list.*
CURSOR	Outputs position of cursor.
DEFINE *name list*	Makes *list* the definition of *name.*
DEFINEDP *word*	Outputs TRUE if *word* is the name of a procedure.
DISK	Outputs information about disk and disk drive.
DOT *pos*	Puts a dot at *pos.*
EDIT, ED *(name(s))*	Starts Logo editor (containing named procedure(s)).
EDNS *(package(list))*	Starts Logo editor (containing variables in *package(list)*).
EMPTYP *obj*	Outputs TRUE if *obj* is empty list or empty word.
EQUALP *obj1 obj2*	Outputs TRUE if its inputs are equal.
ERALL *(package(list))*	Erases everything (in *package(list)*).
ERASE *name(s)*	Erases named procedures.
ERASEFILE *name*	Erases file *name*.LOGO from disk.
ERN *name(s)*	Erases all named variables.
ERNS *(package(list))*	Erases variables (in *package(list)*).
ERPS *(package(list))*	Erases procedures (in *package(list)*).
ERROR	Outputs list of information about most recent error.
FENCE	Fences turtle within edges of screen.
FIRST *obj*	Outputs first element of *obj.*
FORWARD, FD *n*	Moves turtle *n* steps forward.
FPUT *obj list*	Outputs list formed by putting *obj* on front of *list.*
FULLSCREEN	Devotes entire screen to graphics.
GO *word*	Transfers control to LABEL *word.*
GPROP *name prop*	Outputs *prop* property of *name.*
HEADING	Outputs turtle's heading.
HIDETURTLE, HT	Makes turtle invisible.

HOME	Moves turtle to [0 0] and sets heading to 0
IF *pred list1 (list2)*	If *pred* is TRUE, runs *list1*, otherwise *list2*.
IFFALSE, IFF *list*	Runs list if most recent TEST was FALSE.
IFTRUE, IFT *list*	Runs list if most recent TEST was TRUE.
INT *n*	Outputs integer portion of *n*.
ITEM *n obj*	Outputs *n*th element of *obj*.
KEYP	Outputs TRUE if a key has been typed but not yet read.
LABEL *word*	Labels line for use by GO.
LAST *obj*	Outputs last element of *obj*.
LEFT, LT *n*	Turns turtle *n* degrees left, i.e. counterclockwise.
'LIST *obj1 obj2*	Outputs list of its inputs.
LISTP *obj*	Outputs TRUE if *obj* is a list.
LOAD *name (pkg)*	Loads file *name*.LOGO into workspace (in package *pkg*).
'LOCAL *name*	Makes *name* local.
LPUT *obj list*	Outputs list formed by putting *obj* on end of *list*.
MAKE *name obj*	Makes *name* refer to *obj*.
MEMBERP *obj list*	Outputs TRUE if *obj* is an element of *list*.
NAME *obj name*	Makes *obj* the value of *name*.
NAMEP *word*	Outputs TRUE if *word* refers to any object.
NODES	Outputs number of free nodes.
NOT *pred*	Outputs TRUE if *pred* is FALSE.
NUMBERP *obj*	Outputs TRUE if *obj* is a number.
'OR *pred1 pred2*	Outputs TRUE if any its inputs are TRUE.
OUTPUT, OP *obj*	Returns control to caller, with *obj* as output.
PACKAGE *package name(s)*	Puts named procedures in *package*.
PADDLE *n*	Outputs rotation of dial on paddle *n*.

PAUSE	Makes procedure pause.
PEN	Outputs pen state (list containing type and color).
PENCOLOR, PC	Outputs number representing pen color.
PENDOWN, PD	Puts pen down.
PENERASE, PE	Puts eraser down.
PENREVERSE, PX	Puts reversing pen down.
PENUP, PU	Puts pen up.
PKGALL *package*	Puts in *package* everything that's not packaged.
PLIST *name*	Outputs property list of *name*.
PO *name(s)*	Prints definitions of named procedures.
POALL *(package(list))*	Prints definitions of procedures and names (in *package*(*list*)).
PONS *(package(list))*	Prints names and values of variables (in *package*(*list*)).
POPS *(package(list))*	Prints definitions of procedures (in *package*(*list*)).
POS	Outputs position of turtle.
POTS *(package(list))*	Prints title lines of procedures (in *package*(*list*)).
PPS *(package(list))*	Prints property list(s) of everything (in *package(list)*).
PRIMITIVEP *name*	Outputs TRUE if *name* is a primitive.
PRINT *obj*	Prints *obj* followed by RETURN (no brackets for lists).
PRODUCT *a b*	Outputs product of its inputs.
QUOTIENT *a b*	Outputs integer portion of *a/b*.
RANDOM *n*	Outputs random non-negative integer less than *n*.
READCHAR, RC	Outputs character typed by user (waits if necessary).
READLIST, RL	Outputs line typed by user (waits if necessary).
RECYCLE	Performs a garbage collection.

REMAINDER *a b*	Outputs remainder of *a* divided by *b*.
REMPROP *name prop*	Removes property *prop* from *name*.
REPARSE	Performs a reparsing.
REPEAT *n list*	Runs *list n* times.
RERANDOM	Makes RANDOM behave reproducibly.
RIGHT, RT *n*	Turns turtle *n* degrees right, i.e. clockwise.
ROUND *n*	Outputs *n* rounded off to nearest integer.
RUN *list*	Runs *list*; outputs what *list* outputs.
SAVE *name (package(list))*	Writes whole workspace or *package(list)* onto file *name*.LOGO.
SCRUNCH	Outputs current aspect ratio.
SENTENCE, SE *obj1 obj2*	Outputs list of its inputs.
SETBG *n*	Sets background to color represented by *n*.
SETCURSOR *pos*	Puts cursor at *pos*.
SETDISK *drive (slot) (volume)*	Sets *drive*, *slot*, and *volume* number.
SETHEADING, SETH *n*	Sets turtle's heading to *n* degrees.
SETPC *n*	Sets pen color to *n*.
SETPEN *pair*	Sets pen type and color to elements of *pair*.
SETPOS *pos*	Moves turtle to *pos*.
SETSCRUNCH *n*	Sets aspect ratio to *n*.
SETX *x*	Moves turtle horizontally so that x-coordinate is *x*.
SETY *y*	Moves turtle vertically so that y-coordinate is *y*.
SHOW *obj*	Prints *obj* followed by RETURN (with brackets for lists).
SHOWNP	Outputs TRUE if turtle is shown.
SHOWTURTLE, ST	Makes turtle visible.

SIN *n*	Outputs sine of *n* degrees.
SPLITSCREEN	Splits screen: top for graphics, bottom for text.
SQRT *n*	Outputs square root of *n*.
STOP	Stops procedure and returns control to caller.
'SUM *a b*	Outputs sum of its inputs.
TEST *pred*	Remembers whether *pred* is TRUE or FALSE.
TEXT *name*	Outputs definition of procedure *name* as a list.
TEXTSCREEN	Devotes entire screen to text.
THING *name*	Outputs object referred to by *name*.
THROW *name*	Transfers control to corresponding CATCH.
TO *name (inputs)*	Begins defining procedure *name*.
TOWARDS *pos*	Outputs heading turtle would have if facing *pos*.
'TYPE *obj*	Prints *obj* (no brackets for lists).
UNBURY *(pkg)*	Unburies procedures in *pkg* or workspace.
WAIT *n*	Pauses for *n* 60ths of a second.
WINDOW	Makes turtle field unbounded.
'WORD *word1 word2*	Outputs word made up of its inputs.
WORDP *obj*	Outputs TRUE if *obj* is a word.
WRAP	Makes turtle field wrap around edges of screen.
XCOR	Outputs x-coordinate of turtle.
YCOR	Outputs y-coordinate of turtle.
.BPT	Enters the Apple monitor.
.CONTENTS	Outputs list of names, procedure names, and other words.
.DEPOSIT *n a*	Writes *a* into address *n* (decimal).
.EXAMINE *n*	Outputs contents of address *n* (decimal).
.PRINTER *slot*	Directs printing to printer instead of screen.

Primitives (Infix Form)

a + *b*	Outputs *a* plus *b*.
(a) − *b*	Outputs *a* minus *b*.
a * *b*	Outputs *a* times *b*.
a / *b*	Outputs *a* divided by *b*.
a < *b*	Outputs TRUE if *a* is less than *b*.
obj1 = *obj2*	Outputs TRUE if *obj1* is equal to *obj2*.
a > *b*	Outputs TRUE if *a* is greater than *b*.

Special characters

An asterisk (*) indicates an editing command which works both inside and outside of the editor.

*← or DELETE	Erases character to left of cursor.
*→	Same as CTRL-F.
*CTRL-A	Moves cursor to beginning of current line.
*CTRL-B	Moves cursor one space backward.
CTRL-C	Exits from editor, reading buffer as if typed in.
*CTRL-D	Erases character at cursor position.
*CTRL-E	Moves cursor to end of current line.
*CTRL-F	Moves cursor one space forward.
CTRL-G	Interrupts running procedure, stopping it.
*CTRL-H	Same as ←.
*CTRL-K	Erases everything on current line to right of cursor.
CTRL-L	Scrolls screen to put current line at center (in editor).
CTRL-L	Devotes entire screen to graphics (outside of editor).
CTRL-M	Same as RETURN.
CTRL-N	Moves cursor down to next line in editor.

CTRL-O	Opens new line at position of cursor in editor.
CTRL-P	Moves cursor up to previous line in editor.
*CTRL-Q	Quotes next character you type; prints a backslash (\).
CTRL-S	Splits screen: top for graphics, bottom for text.
CTRL-T	Devotes entire screen to text.
*CTRL-U	Same as CTRL-F.
CTRL-V	Scrolls screen to next page in editor.
CTRL-W	Makes Logo stop until another character is typed.
*CTRL-Y	Inserts the contents of the kill buffer.
CTRL-Z	Interrupts running procedure, making it pause.

Appendix C Magic Words

END	Tells Logo that you are done defining a procedure.
ERRACT	System variable: if TRUE, Logo pauses when error occurs.
ERROR	Tag for THROW when error occurs.
FALSE	Special input for AND, IF, NOT, OR, and TEST.
PROCPKG	Property of procedure name; value is its package.
REDEFP	System variable: if TRUE, primitives can be redefined.
STARTUP	System variable: if a list, Logo runs it after starting up.
TOPLEVEL	Tag for THROW to return control to top-level.
TRUE	Special input for AND, IF, NOT, OR, and TEST.
VALPKG	Property of variable name; value is its package.
SYSTEM	Package containing ERRACT and REDEFP (initially buried).

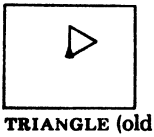
Appendix D An Example of Using the Editor

You have been introduced to defining a procedure using the editor, and to some editing actions (CTRL-A, CTRL-B, CTRL-E, CTRL-F, CTRL-N, CTRL-P, DELETE, CTRL-C) in the *Introduction to Programming through Turtle Graphics*. There are many editing actions, all of which are described in Chapter 6 and are listed in Appendix B.

There are a few more editing actions that are so useful that we will show you in tutorial style how they are used.

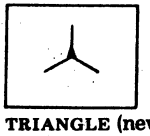
Define the procedure TRIANGLE.

```
TO TRIANGLE :SIDE
FD :SIDE
RT 120
FD :SIDE
RT 120
FD :SIDE
RT 120
END
```



TRIANGLE (old version)

Let's imagine we want to change it so that it draws this:



TRIANGLE (new version)

Here is what the new procedure will look like:

```
TO TRIANGLE :SIDE
FD :SIDE
BK :SIDE
RT 120
FD :SIDE
BK :SIDE
RT 120
FD :SIDE
BK :SIDE
FT 120
END
```

Let's EDIT our old procedure and transform it into this new one. Type

```
EDIT "TRIANGLE
```

Now, use CTRL-N to move the cursor to the beginning of the third line

```
TO TRIANGLE :SIDE
FD :SIDE
RT 120
FD :SIDE
RT 120
FD :SIDE
RT 120
END
```

We want to add our new line where the cursor is now. You could just type the new instruction and then RETURN. That would work fine. Try typing BK; notice that the text on the line moves out of the way.

```
TO TRIANGLE :SIDE
FD :SIDE
BKRT 120
FD :SIDE
RT 120
FD :SIDE
RT 120
END
```

Now DELETE the characters you just typed (so that the cursor is at the beginning of the line again) and type CTRL-O. This editor action means Open new line. The screen looks like

```
TO TRIANGLE :SIDE
FD :SIDE

RT 120
FD :SIDE
RT 120
FD :SIDE
RT 120
END
```

You have a fresh open line right in front of the cursor on which to type your new instructions. Type BK :SIDE. Put in the other two occurrences of this new instruction in the same way:

```
TO TRIANGLE :SIDE
FD :SIDE
BK :SIDE
RT 120
FD :SIDE
BK :SIDE
RT 120
FD :SIDE
BK :SIDE
RT 120
END
```

Now type CTRL-C to exit from the editor. Try your new TRIANGLE procedure.

Get back into the editor to see a few other tricks. Type EDIT "TRIANGLE.

Now move the cursor to the beginning of the third line again by typing CTRL-N. Type CTRL-K. This means Kill the rest of the current line. You will see that line disappear. You can bring the line back by typing CTRL-Y. This means Yank the kill buffer. Try it.

When you type CTRL-K, the editor puts the text on the rest of the current line into a special place called the *kill buffer,* and then erases that text from the screen. The text stays in the kill buffer. You can then insert a copy of that text at the position of the cursor (as if you typed it again) by typing CTRL-Y. This can be used for moving lines around in a procedure, or for putting in multiple copies of a line.

Try this: move the cursor to the beginning of the last line (get it there however you want; CTRL-N is recommended). Type CTRL-O. Now type CTRL-Y. This yanks the text from the kill buffer again. Type RETURN. Type CTRL-Y again. This puts another copy of that text in at the position of the cursor.

```
TO TRIANGLE :SIDE
FD :SIDE
BK :SIDE
RT 120
FD :SIDE
BK :SIDE
RT 120
FD :SIDE
BK :SIDE
RT 120
BK :SIDE
BK :SIDE
END
```

Experiment with moving lines around using CTRL-O, CTRL-K, and CTRL-Y. Don't worry about messing up the TRIANGLE procedure.

Another important editing action happens when you type CTRL-D (which stands for Delete character). This deletes the character *at the current cursor position.* The DELETE (←) key, in contrast,

deletes the character *to the left of the current cursor position.* Both of these actions are useful when you are typing.

Move the cursor to various places on the screen and type CTRL-D. See how the character that the cursor is on is deleted. This is a useful action to use when you discover a typing mistake you have made earlier. Just move the cursor onto the wrong characters, and delete them.

What happens if you move the cursor to the end of a line (use CTRL-E) and type CTRL-D? Try it. The next line "moves up" and gets merged with the current line. What really happened is that you Deleted the RETURN at the end of the line you were on. This emphasizes that there is a RETURN at the end of each line, and even though it is invisible it is a real character and can be deleted. Put it back in by pressing the RETURN key.

Now you know the fundamentals of using the Logo editor. By this time, you should have really messed up the appearance of the TRIANGLE definition on the screen. If things are damaged like this, and you want to get back to Logo *without* giving Logo the new (messed up) definition, exit from the editor by typing CTRL-G. This gets you out of the editor but does not give any new information to Logo. TRIANGLE does not get redefined. Try it. You should get back to Logo:

```
?PO "TRIANGLE
```

Logo should print the last definition of TRIANGLE that we gave it:

```
TO TRIANGLE :SIDE
FD :SIDE
BK :SIDE
RT 120
FD :SIDE
BK :SIDE
RT 120
FD :SIDE
BK :SIDE
RT 120
END
```

Appendix E Error Messages

Number	Message
1	(procedure) IS ALREADY DEFINED
2	NUMBER TOO BIG
3	(symbol) ISN'T A PROCEDURE
4	(symbol) ISN'T A WORD
5	(procedure) CAN'T BE USED IN A PROCEDURE
6	(symbol) IS A PRIMITIVE
7	CAN'T FIND LABEL (symbol)
8	CAN'T (symbol) FROM THE EDITOR
9	(symbol) IS UNDEFINED
10	(procedure) DIDN'T OUTPUT TO (symbol)
11	I'M HAVING TROUBLE WITH THE DISK
12	DISK FULL
13	CAN'T DIVIDE BY ZERO
14	END OF DATA
15	FILE ALREADY EXISTS
16	FILE LOCKED
17	FILE NOT FOUND
18	FILE IS WRONG TYPE
19	TOO FEW ITEMS IN (list)
20	NO MORE FILE BUFFERS
21	CAN'T FIND CATCH FOR (symbol)
22	(symbol) NOT FOUND
23	OUT OF SPACE
24	(procedure) CAN'T BE USED IN A PROCEDURE
25	(symbol) IS NOT TRUE OR FALSE
26	PAUSING . . .
27	YOU'RE AT TOPLEVEL
28	STOPPED!
29	NOT ENOUGH INPUTS TO (procedure)
30	TOO MANY INPUTS TO (procedure)
31	TOO MUCH INSIDE PARENTHESES
32	TOO FEW ITEMS IN (list)
33	CAN ONLY DO THAT IN A PROCEDURE
34	TURTLE OUT OF BOUNDS
35	I DON'T KNOW HOW TO (symbol)

36	(symbol) HAS NO VALUE
37	) WITHOUT (
38	I DON'T KNOW WHAT TO DO WITH (symbol)
39	DISK VOLUME MISMATCH
40	DISK IS WRITE PROTECTED
41	(procedure) DOESN'T LIKE (symbol) AS INPUT
42	(procedure) DIDN'T OUTPUT

!!! LOGO SYSTEM BUG !!!
Should not occur. Please write to LCSI if it does.

Appendix F Programs Provided in Startup File

On the Logo file diskette there is a file named STARTUP. It is loaded into your workspace when Logo starts up. We have put the following procedures in that file:

ARCLEFT ARCL
ARCRIGHT ARCR
CIRCLEL
CIRCLER
READWORD RW

If you have rewritten your STARTUP file, you can type in the definitions below and put them back in your STARTUP file.

These procedures, as well as ARCR1 and ARCL1, are organized into a package named AIDS. The package is *buried.*

Here is a listing of these procedures.

ARCS AND CIRCLES

```
TO ARCRIGHT :RADIUS :DEGREES
ARCR1 .174532 * :RADIUS :DEGREES/10
IF 0 = REMAINDER :DEGREES 10 [STOP]
FD .174532 * :RADIUS/20/REMAINDER :D!
EGREES 10
RT REMAINDER :DEGREES 10
END

TO ARCR :RADIUS :DEGREES
ARCRIGHT :RADIUS :DEGREES
END

TO ARCLEFT :RADIUS :DEGREES
ARCL1 .174532 * :RADIUS :DEGREES/10
IF 0 = REMAINDER :DEGREES 10 [STOP]
FD .174532 * :RADIUS/20/REMAINDER :D!
EGREES 10
LT REMAINDER :DEGREES 10
END

TO ARCL :RADIUS :DEGREES
ARCLEFT :RADIUS :DEGREES
END
```

```
TO ARCR1 :STEP :TIMES
REPEAT :TIMES [RT 5 FD :STEP RT 5]
END

TO ARCL1 :STEP :TIMES
REPEAT :TIMES [LT 5 FD :STEP LT 5]
END

TO CIRCLEL :RADIUS
ARCL1 .174532 * :RADIUS 36
END

TO CIRCLER :RADIUS
ARCR1 .174532 * :RADIUS 36
END
```

Comments: These arc and circle procedures actually draw a 36-sided polygon. (The number .174532 is the result of computing 2* PI / 36; PI is rounded to 3.1416; and 36 is the number of sides of the polygon.)

READWORD AND RW

```
TO READWORD
OP FIRST READLIST
END

TO RW
OP READWORD
END
```

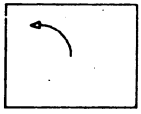
ARCLEFT
ARCL

ARCLEFT *radius degrees* (command) short form: ARCL

Draws an arc of the specified number of *degrees* curving toward the left, taken from a circle with the specified *radius.*

Example:

ARCLEFT 30 90 draws a quarter of a circle with a radius of 30 turtle steps.



ARCLEFT 30 90

ARCRIGHT
ARCR

ARCRIGHT *radius degrees* (command) short form: ARCR

Draws an arc of the specified number of *degrees* curving toward the right, taken from a circle with the specified *radius.*

Example:

ARCRIGHT 30 90 draws a quarter of a circle with a radius of 30 turtle steps.

ARCRIGHT 30 90

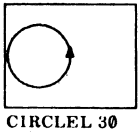
CIRCLEL

CIRCLEL *radius* (command)

Draws a circle of the specified radius curving toward the left.

Example:

CIRCLEL 30 draws a circle with a radius of 30 turtle steps.



CIRCLEL 30

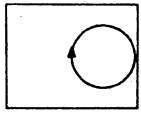
CIRCLER

CIRCLER *radius* (command)

Draws a circle of the specified radius curving toward the right.

Example:

CIRCLER 30 draws a circle with a radius of 30 turtle steps.



CIRCLER 30

READWORD
RW

READWORD (operation) short form: RW

Outputs the first word typed. If the input buffer is empty, READWORD waits until the user types something. If a line containing more than one word is typed, READWORD picks up only the first word.

Appendix G Useful Tools

Some of the procedures defined earlier throughout this manual are likely to be useful in constructing your own procedures. They are collected here in alphabetical order for your convenience. Refer to the index to find out where examples of their use appear.

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

TO DIVISORP :A :B
OP Ø = REMAINDER :B :A
END

TO FOREVER :INSTRUCTIONLIST
RUN :INSTRUCTIONLIST
FOREVER :INSTRUCTIONLIST
END

TO INV.VIDEO :WORD
SPECIAL.TYPE :WORD 128
END

TO SPECIAL.TYPE :WORD :ADDEND
IF EMPTYP :WORD [STOP]
TYPE CHAR :ADDEND + REMAINDER ASCII!
FIRST :WORD 64
SPECIAL.TYPE BF :WORD :ADDEND
END

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

TO POLY :SIDE :ANGLE
FD :SIDE
RT :ANGLE
POLY :SIDE :ANGLE
END
```

```
TO PROMPT :MESSAGE
TYPE :MESSAGE
TYPE " \                    (CTRL-Q followed by a space)
END

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

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

Appendix H Space

Logo procedures and variables take up space; more space is used when the procedures are run.

Some Logo users may wish to know how space is used in Logo and how to conserve it. In general, saving space is not something you should worry about. Instead you should try to write procedures as clearly and elegantly as possible. However, we recognize that the Apple II has only a finite memory. This appendix discusses how space is allocated in Logo and how you can use less of it.

How It Works

Space in Logo is allocated in *nodes,* each of which is five bytes long. All Logo objects and procedures are built out of nodes. 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.

For example, during execution of the following procedure

```
TO PRINT.SUM :N :M
MAKE "N :N + :M
PRINT :N
END
```

after you say PRINT.SUM 3 4, N is assigned to a new node that holds the value 7. After PRINT.SUM stops, the old value of N (if any) is restored and the 7 will be lost forever. The node containing the 7 can be reused, and it will be reclaimed as a free node 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.

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
259
```

How Space Is Used

Every Logo word used is stored only once: all occurrences of that word are actually *pointers* to the word. A word takes up two nodes, plus one node for every two letters in its name and additional node if this word has a property list. Some words can share the ends of their names with other words; for instance, if you say WORD "SAN "FRANCISCO the resulting word SANFRANCISCO takes only two nodes for its name, since the FRANCISCO part is shared with the word FRANCISCO.

Not all words that end in the same letters will be shared—mostly only those constructed with WORD.

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). You can get a rough estimate of the size of a procedure by taking the size of the list that would be output by TEXT.

Space Saving Hints

1. It is important to remember that it is bad form to save space by writing procedures that are less readable because of the use of short or obscure words.

2. Rewrite the program. Use procedures to replace repetitive sections of the program.

3. Space can be saved in Logo by not creating new words. The names of local variables of procedures can be the same as names of local variables of other procedures. The names of procedures and primitives can also be used as variable names. Lastly and most extreme, you can reuse or erase primitive names; doing so does not destroy the word but does reclaim two nodes that were used to store internal information about the primitive.

4. It should be noted that misspellings, typing errors, and words that are no longer being used are not destroyed. All currently existing words can be seen on the list output by CONTENTS. For instance if you type:

```
?PRIMT "FOO
I DON'T KNOW HOW TO PRIMT
?ALSJKDFLKDFJKLFJ
```

the words PRIMT, FOO, and ALSJKDFLKDFJKLFJ will be created and will not go away. However, if a word has no value, property list, or procedure definition, it will not be written out to a file. So if you are running out of space and have a lot of these words (sometimes known as *truly worthless atoms*) you can write out your workspace and then read it into a freshly started Logo.

Appendix I Parsing

When you type a line at Logo, it recognizes the characters as words and lists, and builds a list which is Logo's internal representation of the line. This process is called *parsing*. The list is similar to the list that would be output by READLIST. This appendix will help you understand how lines are parsed.

Delimiters

A word is usually *delimited* by spaces. That is, there is a space before the word and a space after the word; they set the word off from the rest of line. There are a few other delimiting characters:

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

There is no need to type a space between a word and any of these characters. For example, the line

```
IF 1<2[PRINT(3+4)/5][PRINT :X+6]
```

is parsed exactly like

```
IF 1 < 2 [PRINT ( 3 + 4 ) / 5] [PRI
NT :X + 6]
```

And if you define a procedure to contain a line in the first form, you will see that Logo has converted it into the second.

To treat any of the characters mentioned above as a normal alphabetic character, put a backslash "\" (typed with CTRL-Q) before it. For example:

```
?PRINT "SAN\ FRANCISCO
SAN FRANCISCO          (a single word, containing a space)
```

Infix Procedures

The characters =, <, >, +, −, *, / are the names of *infix procedures*. They are treated as procedures of two inputs, but the name is written between

the two inputs. This is a property of the *procedure,* not the name, as the following example shows (REDEFP must be TRUE).

```
?COPYDEF "PLUS "+
?PRINT 3 PLUS 4
7
?
```

Note: the fact that a particular symbol is the name of an infix procedure is independent from the fact that it is a delimiter: e.g., if you redefined +, it would no longer be an infix procedure but would still be a self-delimiting character. (These sorts of redefinitions are not recommended.)

In many cases (those for which the behavior of the procedure would not be ambiguous), the infix procedures may be used in prefix form like normal Logo procedures:

```
?PRINT * 4 5
20
?PRINT (+ (* 2 3) (* 4 5))
26
?PRINT + * 2 3 * 4 5
29                          (this parsed as (+ (* 2 (3*4)) 5)
```

Brackets

Left bracket "[" and right bracket "]" indicate the start and end of a list or sublist. If the end of a Logo line is reached (that is, the RETURN key is pressed) and brackets are still open, all sublists are closed. 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 a right bracket is found for which there was no corresponding left bracket, it is ignored. Brackets (along with space) are somewhat different from the other delimiting characters in that they are not Logo words.

Quotes and Dots

There is an exception to the rule that the delimiting characters delimit words. If a delimiting word is preceded by quotes or dots, the delimiting character is parsed as a quoted or dotted one-letter word (where normally you would expect the delimiter to delimit the quoted or dotted zero-character word). For example:

```
?PRINT "+
+
?PRINT "+PRINT "(
+
(
?PRINT "\+PRINT          (the backslash is typed by CTRL-Q)
+PRINT
```

The Minus Sign

The way in which the minus sign "−" is parsed is also a little strange. The problem here is that one character is used to represent three different things:

1. as part of a number to indicate that it is negative, as in −3
2. as a procedure of one input, called *unary minus*, which outputs the additive inverse of its input, as in −XCOR or −:DISTANCE
3. as 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 figure out which one was meant by the following rules:

1. If the "−" immediately precedes a number, and follows any delimiter except right parenthesis ")", the number is parsed as a negative number. This allows the following behavior:
   PRINT SUM 20−20 (parses as 20 minus 20)
   PRINT 3*−4 (parses as 3 times negative 4)
   PRINT (3+4)−5 (parses as 3 plus 4 minus 5)
   FIRST [−3 4] (outputs −3)

2. If the "−" immediately precedes a word or left parenthesis "(", and follows any delimiter except right parenthesis, it is parsed as the unary minus procedure:
   SETPOS LIST :X −:Y
   SETPOS LIST YCOR −XCOR
   FIRST [−XCOR] (outputs −)

3. In all other cases, "−" is parsed like the other infix characters—as a procedure with two inputs:
   PRINT 3−4 (parses as 3 minus 4)
   PRINT 3 − 4 (parses exactly like the previous example)
   PRINT − 3 4 (procedurally the same as the previous example)

Appendix J Redefining Primitives

You can change the definition of a Logo primitive using DEFINE. DEFINE won't work for primitives unless :REDEFP is TRUE. (It is initially FALSE, to prevent you from redefining primitives accidentally.)

The following example of redefining FD is a trick you could play on an unsuspecting friend.

```
MAKE "REDEFP "TRUE
DEFINE "FD [[N] [BK :N]]
```

Try FD 100

Later you could restore FD's definition, by typing

```
COPYDEF "FD "FORWARD
```

There are less frivolous reasons for redefining primitives. One possible reason is to get information about the running of your program to help in debugging. For example, if you find that your variables have the wrong values, you might find it helpful to have a version of MAKE that records what it's doing. Here's how:

```
?COPYDEF "TRUE.MAKE "MAKE
?DEFINE "MAKE [[NAME THING] [PR (SE !
"MAKING :NAME "= :THING) TRUE.MAKE :!
NAME :THING]]
```

After you have redefined primitives you may want to make :REDEFP FALSE to protect primitives from accidental changes.

Appendix K The ASCII Code

This appendix contains a chart of ASCII code values (in decimal) for all characters in LCSI Apple Logo. Note that characters can be

- *normal* (white characters on black background)
- *inverse video* (black characters on white background)
- *flashing* (rapidly alternating between normal and inverse)

Apple Logo cannot print the full ASCII character set. Lower-case and CTRL characters print as their upper-case equivalents; characters 9b, 123, 124, 125, and 126 print as other punctuation marks.

Inverse and flashing are extensions to ASCII. They may not work in other implementations of Logo.

Notes

To change a normal character to inverse, use the expression

```
CHAR 128 + REMAINDER ASCII :CHARACTER 64
```

To change a normal charcter to flashing, use the expression

```
CHAR 192 + REMAINDER ASCII :CHARACTER 64
```

Lower-case letters print as upper-case on the screen when using the Apple.

(NOTE: The prefix ^ indicates CTRL.)

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	BELL	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	ESC	59	;	91	[	123	{
28	^\	60	<	92	\	124	|
29	^]	61	=	93	]	125	}
30	^↑	62	>	94	^	126	~
31	^_	63	?	95	_	127	DEL

ASCII code	(inverse) char	ASCII code	(inverse) char	ASCII code	(flashing) char	ASCII code	(flashing) char
128	@	160	SPACE	192	@	224	SPACE
129	A	161	!	193	A	225	!
130	B	162	"	194	B	226	"
131	C	163	#	195	C	227	#
132	D	164	$	196	D	228	$
133	E	165	%	197	E	229	%
134	F	166	&	198	F	230	&
135	G	167	'	299	G	231	'
136	H	168	(	200	H	232	(
137	I	169	)	201	I	233	)
138	J	170	*	202	J	234	*
139	K	171	+	203	K	235	+
140	L	172	,	204	L	236	,
141	M	173	-	205	M	237	-
142	N	174	.	206	N	238	.
143	O	175	/	207	O	239	/
144	P	176	0	208	P	240	0
145	Q	177	1	209	Q	241	1
146	R	178	2	210	R	242	2
147	S	179	3	211	S	243	3
148	T	180	4	212	T	244	4
149	U	181	5	213	U	245	5
150	V	182	6	214	V	246	6
151	W	183	7	215	W	247	7
152	X	184	8	216	X	248	8
153	Y	185	9	217	Y	249	9
154	Z	186	:	218	Z	250	:
155	[	187	;	219	[	251	;
156	\	188	<	220	\	252	<
157	]	189	=	221	]	253	=
158	^	190	>	222	^	254	>
159	_	191	?	223	_	255	?

Logo Vocabulary

Note: Parentheses around an input indicate that the input is optional. A number sign (#) indicates a procedure which can take any number of inputs; if you give it other than the number indicated, you must enclose the entire expression in parentheses.

Procedures and Their Inputs

Turtle Graphics

BACK, BK *distance*
BACKGROUND, BG
CLEAN
CLEARSCREEN, CS
DOT *position*
FENCE
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*
SCRUNCH
SETBG *colornumber*
SETHEADING, SETH *degrees*
SETPC *colornumber*
SETPEN *pair*
SETPOS *position*
SETSCRUNCH *n*
SETX *x*
SETY *y*
SHOWNP
SHOWTURTLE, ST
TOWARDS *position*
WINDOW
WRAP
XCOR
YCOR

Words and Lists

ASCII *char*
BUTFIRST, BF *object*
BUTLAST, BL *object*
CHAR *n*
COUNT *list*
EMPTYP *object*
EQUALP *object1 object2*
FIRST *object*
FPUT *object list*
ITEM *n object*
LAST *object*
LIST *object1 object2*
LISTP *object*
LPUT *object list*
MEMBERP *object list*
NUMBERP *object*
SENTENCE, SE *object1 object2*
WORD *word1 word2*
WORDP *object*
object1 = *object2*

Variables

LOCAL *name*
MAKE *name object*
NAME *object name*
NAMEP *word*
THING *name*

Arithmetic Operations

ARCTAN *n*
COS *degrees*
INT *n*
PRODUCT *a b*
QUOTIENT *a b*
RANDOM *n*
REMAINDER *a b*
RERANDOM
ROUND *n*
SIN *degrees*
SQRT *a*
SUM *a b*
a + *b*
a − *b*
a * *b*
a / *b*
a < *b*
a = *b*
a > *b*

Defining and Editing

EDIT, ED *(name(s))*
EDNS *(package(list))*
TO *procname (inputs)*

Conditionals and Flow of Control

CO *(object)*
IF *pred list1 (list2)*
IFFALSE, IFF *list*
IFTRUE, IFT *list*
OUTPUT, OP *object*
PAUSE
REPEAT *n list*
RUN *list*
STOP
TEST *pred*

Logical Operations

AND *pred1 pred2*
NOT *pred*
OR *pred1 pred2*

The Outside World

BUTTONP *paddlenumber*
KEYP
PADDLE *paddlenumber*
PRINT, PR *object*
READCHAR, RC
READLIST, RL
SHOW *object*
TYPE *object*
WAIT *n*

Text and Screen Commands

CLEARTEXT
CURSOR
FULLSCREEN
SETCURSOR *position*
SPLITSCREEN
TEXTSCREEN

Workspace Management

BURY *package*
ERALL *(package(list))*
ERASE, ER *name(s)*
ERN *name(s)*
ERNS *(package(list))*
ERPS *(package(list))*
PACKAGE *package name(s)*
PKGALL *package*
PO *name(s)*
POALL *(package(list))*

PONS *(package(list))*
POPS *(package(list))*
POTS *(package(list))*
UNBURY *package*

Files
CATALOG
DISK
ERASEFILE *name*
LOAD *file (package)*
SAVE *file (package(list))*
SETDISK *drive (slot) (vol)*

Property lists
GPROP *name prop*
PLIST *name*
PPROP *name prop object*
PPS *(package(list))*
REMPROP *name prop*

Error handling, CATCH, **and** THROW
CATCH *name list*
ERROR
THROW *name*

Further procedure definition and redefinition
COPYDEF *newname name*
DEFINE *procname list*
PRIMITIVEP *name*
TEXT *procname*

Miscellaneous Procedures
DEFINEDP *word*
GO *word*
LABEL *word*
NODES
RECYCLE
REPARSE
.BPT
.CONTENTS
.DEPOSIT *n a*
.EXAMINE *n*
.PRINTER *slot*

Editing Commands

•← OR DELETE
•←
•CTRL-A
•CTRL-B
CTRL-C
•CTRL-D
•CTRL-E
•CTRL-F
•CTRL-H
•CTRL-K
CTRL-L
•CTRL-M
CTRL-N
CTRL-O
CTRL-P
•CTRL-U
CTRL-V
•CTRL-Y
ESC V
ESC <
ESC >

Magic Words

END
ERRACT
ERROR
FALSE
PROCPKG
REDEFP
STARTUP
TOPLEVEL
TRUE
VALPKG
XYZZY
.SYSTEM

Special Characters

CTRL-G
CTRL-Q
CTRL-S
CTRL-T
CTRL-W
CTRL-Z

An asterisk (*) indicates an editing command which works both inside and outside of the editor.