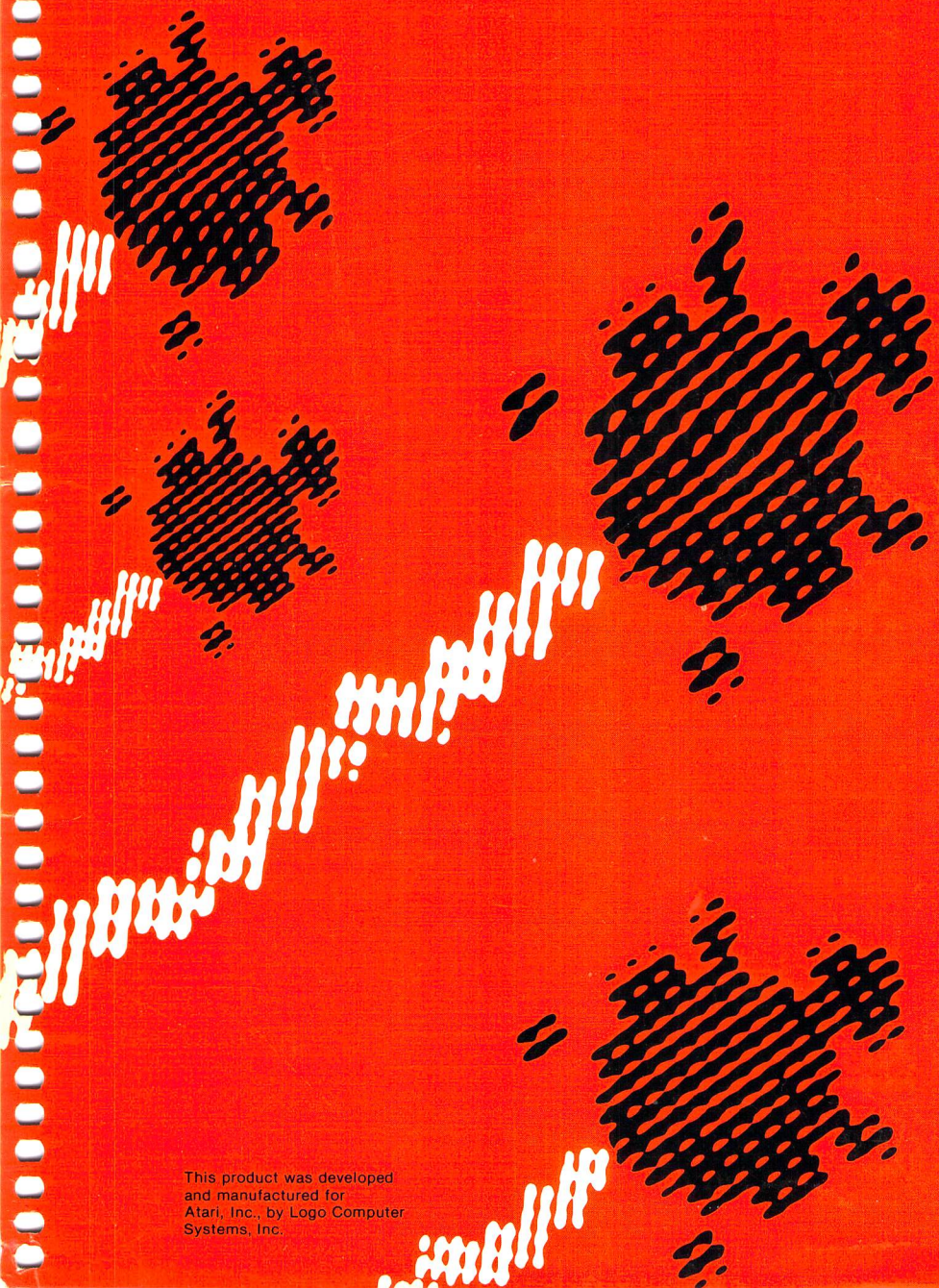
ATARI LOGO

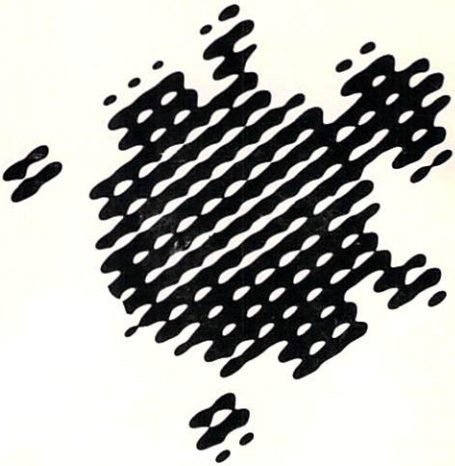
Reference Manual



This product was developed
and manufactured for
Atari, Inc., by Logo Computer
Systems, Inc.

ATARI LOGO

Reference Manual



Disclaimer of all Warranties and Liabilities

Logo Computer Systems, Inc., makes no warranties, expressed or implied, concerning the quality, performance, merchantability or fitness of use for any particular purpose of this manual and the software described in this manual. This manual and the software described in this manual are sold "as is ". The entire risk as to the quality and performance of these goods is with the buyer; if the goods shall prove defective following their purchase, the buyer and not the manufacturer, distributor or retailer assumes the entire cost of all necessary servicing, repair and replacement and any incidental or consequential damages. In no event will Logo Computer Systems, Inc. be responsible for direct, indirect, incidental or consequential damages relating to the purchase or use of these goods, even if Logo Computer Systems, Inc. has been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you.

Notice

Logo Computer Systems, Inc. and ATARI, Inc., reserve the right to make any improvements and changes in the product described in this manual at any time and without notice.

Copyright and Trademark Notices

ATARI is a registered trademark of Atari, Inc. (a Warner Communications Company), used by permission.

Table of Contents

Preface

The *ATARI Logo Reference Manual* should be used for reference rather than as a guide for new users. If you've worked with another version of Logo, this book will help familiarize you with ATARI Logo's special features. First time Logo users should start with the companion manual *Introduction to Programming Through Turtle Graphics*.

The *ATARI Logo Reference Manual* describes Logo primitives. simple commands built into the language, and provides sample programs. Refer to the Table of Contents for organization of the primitives.

There are two useful sections in the front of this manual. The first gives an introduction to Logo grammar. The second explains the conventions used to define the primitives.

There are several ways to use this manual. If you want to know what a *specific* primitive does, look it up in the index. For quick reference, look at Appendix G or H or the *Reference Guide*. If you want to find a primitive to perform a particular task, look at the chapter headings or index.

Appendices include: messages that appear on the screen, handy procedures, technical information, ASCII Codes, and a glossary of ATARI Logo primitives.

Throughout this manual, orange text is used to represent what you type on the computer. Black text is used to represent what the computer displays. Words that are inputs to primitives are in *italics*.

Getting Started

To use the ATARI Logo Cartridge, you need an ATARI Home Computer and a TV set or monitor. If you want to save programs, you need an ATARI Disk Drive or ATARI Program Recorder.

For specific questions about the operation of your ATARI Home Computer, refer to the computer owner's guide. You'll find your computer and ATARI Logo easy to operate. To load ATARI Logo into your computer:

1. With the computer off, turn on your TV set or monitor. If you have one, turn on your ATARI Disk Drive and wait for the busy light to go off. If you are not using a disk drive, skip to step 3.
2. Insert the ATARI Master Diskette in the disk drive and close the disk drive door. You may also use a data diskette if it contains DOS (Disk Operating System) files.
3. Insert the ATARI Logo Cartridge into the console's cartridge slot and turn the computer on.

After a moment you'll see on the screen:

```
(C) 1983 LCSI  ALL RIGHTS RESERVED
WELCOME TO ATARI LOGO.
?■
```

The ? (question mark) is the *prompt* symbol. When ? is on the screen, you can type something. The ■ is the *cursor*. It shows where the next character you type will appear.

The Keyboard

The ATARI Home Computer keyboard is set up like a typewriter.

Character Keys

Character keys — A, B, C, 7, ;, $, etc. They include letters of the alphabet, numbers and punctuation marks.

RETURN

In Logo, the RETURN key serves a programming function. It tells Logo: "Now do what I just typed." Press the RETURN key when you want Logo to obey your instructions.

SPACE BAR

The SPACE BAR prints an invisible but important character called space. Logo uses spaces as word separators. For example, Logo would interpret THISISAWORD as a single word and would interpret THIS IS A WORD as four words.

SHIFT

Holding down SHIFT, while pressing some character keys, changes that particular character key's meaning in Logo. For example, if you hold the SHIFT key down and press], Logo will print] (close bracket) on the screen.

The *bracket*, [ ], symbols are very important in Logo. Do not confuse them with parentheses, ( ), which are SHIFT (and SHIFT).

To make a shift character, always press the SHIFT key first and then *hold* it down *while* typing the other key.

CTRL (CONTROL)

The CTRL key can change character keys into function keys. Press it alone and nothing happens; hold it down and press a certain character key, and something happens. These key combinations do not always print out on the screen, but Logo responds to them.

CTRL Arrow Keys

CTRL ← will move the *cursor* one space to the left and CTRL → will move the *cursor* one space to the right.

The *arrow keys* are useful editing keys. They move the *cursor* in the direction in which they point without affecting the text already there. Note: CTRL ↑ and CTRL ↓ only work in the ATARI Logo Editor. Once the *cursor* is positioned, you can insert or delete characters. To insert text, simply position the *cursor* and begin typing.

DELETE BACK S (DELETE BACK SPACE)

Erases the character to the left of the *cursor*.

BREAK

The BREAK key tells Logo to stop whatever it is doing. It will also get you out of the ATARI Logo Editor without executing the changes. When you press BREAK, Logo types

```
STOPPED!
?
```

then, lets you type the next instruction.

ESC

The ESC (ESCape) key is used to exit the ATARI Logo Editor. This key is discussed, along with other special editing keys, in Chapter 5.

ATARI Key (人) or Reverse Video Key (◩)

If you press the (人) or (◩) key and then type a character key, the character appears in reverse video on the screen (dark character on a light background). You can return to the regular display by pressing the key a second time.

CAPS LOWR (CAPS)

When you first turn on your ATARI Home Computer, anything you type will appear in all uppercase letters. Press the CAPS LOWR key, now only lowercase letters are produced. ATARI Logo primitives must all be typed in uppercase letters. Therefore, if you accidentally press the CAPS LOWR key, Logo will no longer understand your instructions.

SHIFT CAPS LOWR Combination

To lock the keyboard in uppercase, simply hold the SHIFT key and then press the CAPS LOWR key.

SYSTEM RESET (RESET)

Do not use this key once you have booted Logo. You will lose everything in memory.

Logo Grammar

The Logo language is made up of building blocks that can be put together in a number of ways and obey certain rules. These rules are the "grammar" of the language. In order for Logo to understand what you want it to do, you must learn to give proper instructions by following the guidelines that we describe in this section.

Procedures

The building blocks of Logo are procedures and inputs to procedures. Some procedures Logo always knows because they are built into the Logo system. These are called *primitives*. There is a complete list of them in Appendix G.

For example, if you type

```
CT
```

the text is cleared from the screen. You haven't defined CT, but Logo already knows what to do.

There are also procedures, that you define for yourself, using the TO or EDIT commands. There are many examples in both of the manuals.

Here is a procedure definition.

```
TO WELCOME
PRINT "HI
END
```

The first and last lines follow special rules. The first line is called the *title line*. It must always begin with TO followed by the name of a procedure. The last line must contain only the word END.

There is an important difference between "defining" a procedure and asking Logo to "execute" it. When we ask Logo to run a procedure, we say that we have made a *procedure call*.

For example, WELCOME contains a request to run a procedure (which happens to be a primitive) PRINT.

There is another way of asking Logo to make a procedure call. The name is typed in when Logo is at top level (indicated by the question mark prompt at the left of the screen). We have already seen an example with CT. Here is another example.

```
WELCOME
HI
```

If you type in a word and Logo cannot find its definition, you get an error message. Suppose, for example, you haven't defined a procedure called TALK.

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

Within a procedure definition, you can, of course, make a call to a procedure you have previously defined.

```
TO COME.AND.GO
WELCOME
PRINT "BYE
END

COME.AND.GO
HI
BYE
```

We say that WELCOME is a *subprocedure* of COME.AND.GO. COME.AND.GO is a *superprocedure* of WELCOME.

Inputs to Procedures

Some procedures need inputs. For example,

```
PRINT "HI
HI
```

The word "HI is the input to PRINT. The quote mark (") tells Logo that you mean the word HI as itself, not as the name of another procedure. Here is what happens if you don't include the input:

```
PRINT
NOT ENOUGH INPUTS TO PRINT
```

You can use a sentence, instead of a word, for the input to PRINT by putting square brackets around it.

```
PRINT [HAVE A NICE DAY]
HAVE A NICE DAY
```

The procedures that you define can also have inputs. When a procedure you've defined is executed, its inputs are put into *variables*. A variable is like a box which has a name, and which can hold an object (a word or a list, as in the examples of inputs to PRINT before). When you define a procedure with which you want to use inputs, you must provide a variable to "hold" each input. Their names must be written on the title line after the name of the procedure. Each name must have a colon in front. For example,

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

The title line tells Logo that the procedure BIGWELCOME has a single input whose name is NAME. The body of the procedure contains three calls of the procedure PRINT (PR is the short form of PRINT). The second of these uses the input NAME. Here is an example of a request to execute BIGWELCOME at top level.

```
BIGWELCOME "JANE
HI
JANE
HAVE A NICE DAY
```

Here, the input to BIGWELCOME is JANE. Logo makes this the value of NAME when it executes the procedure. Thus, PRINT :NAME does the same thing (in this case) as PRINT "JANE.

Quotes, Colons, and Brackets

When you ask Logo to execute a procedure, you must be very careful about how you write the inputs. A good rule of thumb is that Logo understands every word as a request to run a

procedure unless you specifically indicate that it is not. For example,

```
BIGWELCOME JANE
I DON'T KNOW HOW TO JANE
```

Logo thinks that JANE is a procedure. But since Logo can't find its definition, it doesn't know how to execute it. Here is an example where Logo is able to find the definition.

```
BIGWELCOME SUM 31 28
HI
59
HAVE A NICE DAY
```

SUM is a procedure that adds its inputs. It is a primitive, therefore Logo knows how to do it even though you haven't written a definition for it. We will have more to say about using procedures as inputs in the next section.

In order to tell Logo that an input is *not* a request to run a procedure, you need to use certain characters in a special way.

A word beginning with a quote (for example, "JANE) tells Logo that the input is the word itself and nothing else. We call this a *literal word*. Note that numbers are like literal words but don't need to be quoted.

A word beginning with a colon (for example, :N) tells Logo that the word is the name of a variable and that the input is to be the value of the variable.

A sequence of words surrounded by square brackets (for example, [HAVE A NICE DAY]) indicates that the input is a *list*.

The use of these four special characters is illustrated in the definition of BIGWELCOME.

PRINT "HI tells Logo to display the word HI.

PRINT :N tells Logo to display whatever is the value of N when the procedure is executed.

PRINT [HAVE A NICE DAY] tells Logo to display the list HAVE A NICE DAY. Note that, PRINT leaves out the square brackets in its display. If you want to see the brackets, use SHOW.

The Difference Between Commands and Operations

There are two kinds of procedures in Logo. Those that output a value (like SUM) are called *operations*. Those that do not output a value (like PRINT) are called *commands*. This distinction is so important that we indicate whether each primitive is a command or an operation.

One of the main reasons for this distinction is the fact that an operation can only be written as an input to a procedure. This means that, in every Logo line, the first word must be a *command*.

We have already seen an example with PRINT SUM 31 28. Here are some more examples:

```
PRINT RANDOM 2
1
```

The output of RANDOM 2 is the input to PRINT. The input to RANDOM is 2. When RANDOM 2 is executed, the result is communicated to PRINT.

```
PRINT SUM 3 2
5
```

The result of computing the procedure SUM with inputs 3 and 2 is communicated to PRINT.

```
PRINT SUM 3 PRODUCT 5 2
13
```

The output of PRODUCT is the second input to SUM.

If you try to use a command as an input, this is what happens:

```
PRINT FORWARD 25
FORWARD DIDN'T OUTPUT TO PRINT
```

You get the error message because FORWARD is a command.

Up to now, we have only considered Logo primitives. However, all of the procedures you define yourself are also either commands or operations. For example, the procedure BIGWELCOME (defined previously) is a command. The procedure FLIP is an operation.

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

This outputs the word HEADS if RANDOM 2 outputs 0 or the word TAILS if RANDOM 2 outputs 1. As with primitives, typing only the procedure name alone yields an error message.

```
FLIP
I DON'T KNOW WHAT TO DO WITH HEADS
```

or

```
I DON'T KNOW WHAT TO DO WITH TAILS
```

On the other hand, we have

```
PRINT FLIP
HEADS
```

or

```
TAILS
```

Almost all the procedures in the *Introduction Manual* are commands. On the other hand, procedures involving words, lists, and numbers are frequently operations. To construct your own operations, you will always use the OUTPUT command. For more information, see OUTPUT in Chapter 6.

Variables

The best way to understand variables in Logo is to view them as containers with names on the outside and contents inside. The colon in front of a word tells Logo to make its contents available to the procedure. If you type

```
PRINT :JOHN
```

Logo looks for a container named JOHN. If it finds one, it looks inside the container and makes whatever it finds available to PRINT. PRINT then displays the contents (value) of JOHN on the screen. If it finds nothing, Logo prints the error message:

```
JOHN HAS NO VALUE
```

There are two ways of putting things or placing values inside these containers. The first, which we have already discussed, is by using procedures with inputs. The second is by using the MAKE command.

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

MAKE is a procedure needing two inputs: a word, and a value which can be a word, a list, or a number. Here, it creates a container called JOHN and places 25 inside it. Note that MAKE does not display anything on the screen. It is PRINT that displays this value.

We have here a good illustration of the difference between a quote and a colon. The first input to MAKE is "JOHN because the word JOHN itself is the input, which gives MAKE the *name* of a variable. The input to PRINT is :JOHN because we want to display the *value* of JOHN.

Here is another example:

```
MAKE "X "JOHN
PRINT :X
JOHN
PRINT :JOHN
25
```

In this case, MAKE has two quoted words as inputs. It puts the literal word JOHN inside the container X. The contents of the variable named JOHN from the MAKE of the previous example are left undisturbed.

The Difference Between Global and Local Variables

When Logo is at top level, and you create a variable with MAKE, that variable will remain in your workspace until you erase it. For this reason, it is called a *global variable*. There are also variables that remain in the workspace only as long as a procedure is being executed. These are called *local variables*. Variables that are defined as inputs to procedures are always local variables.

To see the difference, let us modify BIGWELCOME so that it prints the date.

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

Here, DATE could be a global variable which we haven't defined yet. If we try to run BIGWELCOME, we will get the error message

```
DATE HAS NO VALUE IN BIGWELCOME
```

We can use MAKE at top level to give DATE a value.

```
MAKE "DATE [JUNE 23 1983]
BIGWELCOME "BRIAN
JUNE 23 1983
HI
BRIAN
HAVE A NICE DAY
```

On the other hand, NAME is a local variable because it is an input to a procedure. It only contains the word BRIAN while the procedure BIGWELCOME is being executed.

It is easy to forget that you have created a global variable. You can always check which are in your workspace with the command PONS. You can erase them with ERN. Here is an example, that shows that NAME is truly local whereas DATE is truly global.

```
ERN "DATE
MAKE "DATE [JULY 1 1983]
BIGWELCOME "SEYMOUR
JULY 1 1983
HI
SEYMOUR
HAVE A NICE DAY

PONS
MAKE "DATE [JULY 1 1983]
```

There is no value displayed for NAME because NAME has disappeared after BIGWELCOME stops executing.

When you use MAKE inside a procedure definition, the variable can be either local or global. If it is an input to a running procedure then it is local. If it is not an input then it is global.

Note that a procedure does not stop running when a subprocedure is called. Hence a variable that is local to a procedure can be used by its subprocedures.

Understanding a Logo Line

Procedure definitions consist of lines of instructions. We call these Logo lines because they can be much longer than the lines you see on your screen. For example:

```
MAKE "MANYNAMES [BILL MARY JOHN JOE →
FRANK JUDY]
```

The arrow (→) indicates that the next screen line is a continuation of the first Logo line. You get long lines like this by continuing to type without pressing the RETURN key. The right-arrow is automatically displayed and the instruction continues on the next line. The Logo line ends as soon as you press RETURN.

Here are some guidelines or rules-of-thumb to help you interpret a complex Logo line.

1. Whenever you see a procedure name, be sure you know
   (a) how many inputs it has
   (b) whether it is a command or operation.
2. The first word of a Logo line must *always* be a command.
3. An operation is *always* the input to another procedure.
4. Be sure to account for every input to a procedure.
5. When the inputs to a command have been accounted for, the next procedure must be another command.

Here is an example of a complex Logo line. It is part of a procedure COMMENT that illustrates the use of the operation BUTLAST in Chapter 2.

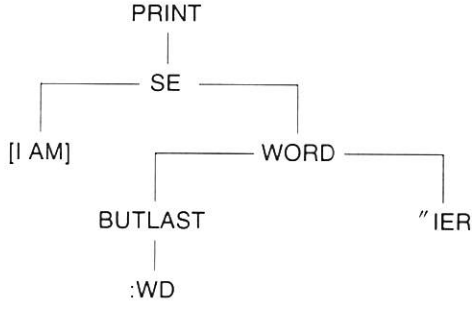
```
PRINT SE [I AM] WORD BUTLAST :WD "IER
```

Let us see how our guidelines help in understanding the line.

PRINT is a command with a single input. This must be the output of SE, which is an operation with two inputs.

The first input to SE is the list [I AM]. The second is the output of the operation WORD. The latter is, once again, an operation with two inputs. The first must be the operation BUTLAST, which has a single input :WD. The second input to WORD must therefore be "IER.

Since there are no more procedure names and every input on the line has been accounted for, we have finished. The following diagram summarizes what we have done.



So, for example, if the value of WD is HAPPY then the line would print I AM HAPPIER.

How Primitives are Described

The rest of this manual consists of a description of each primitive of ATARI Logo.

In bold face at the beginning of each description, you will find the name of the primitive and its short form if one exists. We indicate on the same line whether the primitive is a command, an operation or an infix operation. The difference between a command and an operation is described in Logo Grammar. An infix operation is one that is placed between its inputs. All other primitives are written in front of their inputs.

Below this, we indicate the name of the primitive, followed by the type of each input. All primitives must be entered in uppercase letters. You are to supply all inputs (shown in italics).

This is followed by general information about the primitive and illustrations of how to use the primitive.

How We Describe Formats

If a primitive has more than one format, we write one below the other, with the simplest or most commonly used on the top line. You will see that, with some primitives (such as SUM), an optional format is surrounded by parentheses. This indicates that the primitive will accept as many inputs as you wish. When using more than two inputs with such a primitive, you must always put a left parenthesis before its name and a right parenthesis after the last input.

When we describe the kind of input that a primitive requires, we are *not* speaking about the way the input is written when you define the procedure, the rules for which were described in Logo Grammar. Logo tries to understand a written input by evaluating it and changing it to something else. Table 1 shows what these changes are. For example, if we write

```
MAKE :X 22 + 23
```

and X contains the word JOHN, then the real inputs to MAKE are the word JOHN and the number 45.

Table 1

Written Input	Real Input
Word with quotes in front	Word
Word with colon in front	Contents of word. This can be a word, a list or a number.
Number	Number
List	List
Procedure with inputs	Output of procedure. This can be a word, a list or a number.

In this chapter and throughout the rest of the book, when we describe the kind of input that a primitive requires, we are speaking about the *real* input. With many primitives, an input can be anything you want. In other words, the real input can be a word, a list, or a number. We call this a Logo object. If you look up MAKE, you will see that it must have the following form:

MAKE *name object*

This uses two input words: *name* and *object*. *Name* means that the first input must be a word (we call a word a name if it is to be the name of something like a variable or a procedure) and *object* is an abbreviation for a Logo object. Going back to our example, we see that JOHN is a word and 45 is a Logo object, so we do have the correct inputs.

All of the words that we use in describing the inputs to the Logo primitives are explained on the next few pages.

Input Words

byte	A unit of data used by the computer. An integer from 0 through 255.
character	Letters of the alphabet, numbers, and punctuation marks.
colornumber	An integer from 0 through 127.
condnumber	An integer from 0 through 21. (See COND and WHEN in Chapter 6.)
filename	A file name. (See Chapter 10.)
degrees	Degrees of an angle. A real number between −9999.9999 and 9999.9999. The command REPEAT can be used to exceed this limit.
device	A device name. "C: is Cassette, "D: is Disk, and "P: is Printer. The " (quote mark) and : (colon) are required at all times.
distance	A number from −9999.9999 through 9999.9999. The command REPEAT can be used to exceed this limit.
duration	An integer from 0 through 255.
freq	An integer from 14 through 64,000 in H_Z.
inputs	Words with colons in front. Used in conjunction with TO.
instructionlist	A *list* of procedures that Logo can execute.
joysticknumber	An integer from 0 through 3.
list	Information enclosed in [ ] brackets.
n, a, b, x, y	A number.
name	A word naming a procedure or a variable.
namelist	A *list* of names.

object	A Logo object (a word, a *list* or a number).
paddlenumber	An integer from 0 through 7.
pennumber	An integer from 0 through 2.
position, pos	A *list* of two numbers giving the coordinates of the turtle or the cursor.
pred	A predicate, which is an operation that outputs either the word TRUE or the word FALSE.
shapenumber	An integer from 0 through 15.
shapespec	A *list* of 16 numbers representing the shape grid.
turtlenumber	An integer from 0 through 3.
voice	An integer, either 0 or 1.
volume	An integer from 0 through 15.
word	A sequence of characters (not including a space).

Chapter 1

Turtle Graphics

When you use any primitive or procedure that refers to the turtle, Logo shows you the graphics screen.

We give here a complete list of the commands that change what you see on the graphics screen. Also included are a number of operations that give you information about the turtle's state. Most of them are discussed in the *Introduction to Programming through Turtle Graphics Manual.*

ATARI Logo has four turtles that can perform dynamic actions. They are briefly mentioned in the *Introduction Manual.*

- **Multiple Turtles**
  With ATARI Logo, you can use up to four turtles at once. You can talk to the turtles together or separately. Four primitives allow you to address specific turtles. They are TELL, ASK, EACH and WHO.

- **Dynamic Movement**
  You can set the turtles in motion at the speed you choose by the command SETSP. SPEED tells you the speed of the current turtle(s).

- **Changing the Turtle's Appearance**
  The turtles' shapes and colors can be changed. You can create an unlimited amount of shapes in addition to the predefined turtle shape. The shape editor is used to design any shape you would like to use in place of the original turtle shape. The EDSH command starts the shape editor. SETSH and SETC allow you set the shape and color of the current turtle, whereas SHAPE and COLOR output the appearance of the turtle you are currently talking to.

- **Collision Detection**
  Another feature that affects the extended turtle graphics capabilities is collision detection. The primitives relating to this feature are discussed in Chapter 6 (WHEN, COND, OVER, TOUCHING) and Chapter 9 (POD, PODS).

ASK — command or operation

ASK *turtlenumber instructionlist*
ASK *turtlenumberlist instructionlist*

Asks the turtle(s) specified by *turtlenumber(list)* to run the instructions in the second input. This does not affect the turtle(s) you are currently giving commands to (that is the turtles addressed with TELL). If *instructionlist* is an operation, ASK outputs whatever the operation outputs. *Turtlenumber* is an integer from 0 to 3.

EXAMPLE

These instructions tell turtle 2 to point to the same heading as turtle 1.

```
TELL 1 ST
TELL 2 ST
PR WHO
2
PU SETPOS [-30 0]
SETH 180
SETH ASK 1 [HEADING]
PR WHO
2
```

BACK, BK — command

BACK *distance*

Moves the turtle *distance* steps back. Its heading does not change. Note that BACK 0 (with PENDOWN) displays a single dot at the turtle's current position without moving the turtle. It is an error if *distance* is greater than 9999.9999 or less than −9999.9999.

BG

Stands for BackGround. Outputs a number representing the color of the current background. When Logo starts, BG is light blue (74). See SETBG for setting the background colors. The ATARI computer has 16 colors, each having 8 possible shades, totaling 128 colors to choose from.

0	—	7	gray
8	—	15	light orange (gold)
16	—	23	orange
24	—	31	red-orange
32	—	39	pink
40	—	47	purple
48	—	55	purple-blue
56	—	63	blue
64	—	71	blue
72	—	79	light blue
80	—	87	turquoise
88	—	95	green-blue
96	—	103	green
104	—	111	yellow-green
112	—	119	orange-green
120	—	127	light-orange

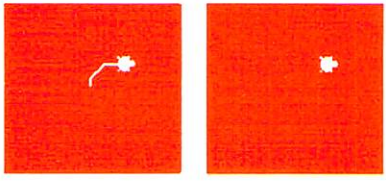
For each color, the lowest number is the darkest shade of that color, and the highest number is the lightest shade of the color. For example, 0 is black and 7 is white.

Note: Colors may vary depending upon the type of TV, monitor, condition, and color adjustments.Colors on PAL systems may be different than the chart above.

CLEAN command

CLEAN

Erases the graphics screen without changing the turtle's state or the text displayed.



CLEAN

COLOR operation

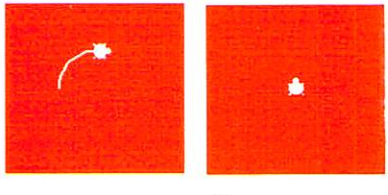
COLOR

Outputs a number representing the turtle's current color. This can be any integer from 0 to 127. When Logo starts, turtle 0 is white (7), turtle 1 is orange (20), turtle 2 is purple (44), and turtle 3 is blue (68). See BG for a chart of colors. See SETC for changing the turtle's color.

CS command

CS

Stands for Clear Screen. Erases the graphics screen, puts the current turtle(s) at position [0 0] (the center of the screen), and sets the turtles' heading to 0 (north). CS also clears any WHEN demons that are in action (see WHEN in Chapter 6). CS does not clear the text (see CT in Chapter 8).



CS

EACH command

EACH *instructionlist*

Makes each turtle, currently in use, separately run the commands in *instructionlist*. If there is more than one active turtle, the first turtle executes all the commands in *instructionlist* before the second turtle does anything. This command is useful when you want each turtle to do slightly different things.

EXAMPLES

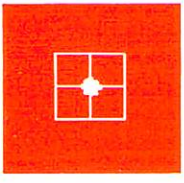
The following instructions make all the turtles line up 20 turtle steps apart and set their colors to the ones corresponding to their numbers.

```
TELL [0 1 2 3]
HOME
EACH [SETX WHO * 20]
EACH [SETC WHO * 8]
```

WHO outputs the identification number corresponding to each turtle. Thus, turtle 0 will do SETX 0 and SETC 0, turtle 1 SETX 20 and SETC 8, and so on.

EACH, like ASK, does not change which turtle(s) you are currently addressing. The difference is that ASK runs each instruction for each turtle at the same time. EACH runs the instructions for one turtle after the other. The following example illustrates this:

```
TO SETUP
CS TELL [0 1 2 3]
EACH [RT 90 * WHO]
END
```



```
SETUP
ASK [0 1 2 3] [REPEAT 4 [FD 50 RT 90→
]]
SETUP
EACH [REPEAT 4 [FD 50 RT 90]]
```

EDSH command

EDSH *shapenumber*

Stands for EDit SHape. Starts up the Logo shape editor which allows you to make up your own shapes. **EDSH** brings the shape corresponding to the *shapenumber* into the editor, *shapenumber* being an integer from 1 to 15. Note that shape number 0 is the normal turtle shape and can't be edited. See the description at the end of this chapter for more information on the Turtle Shape Editor.

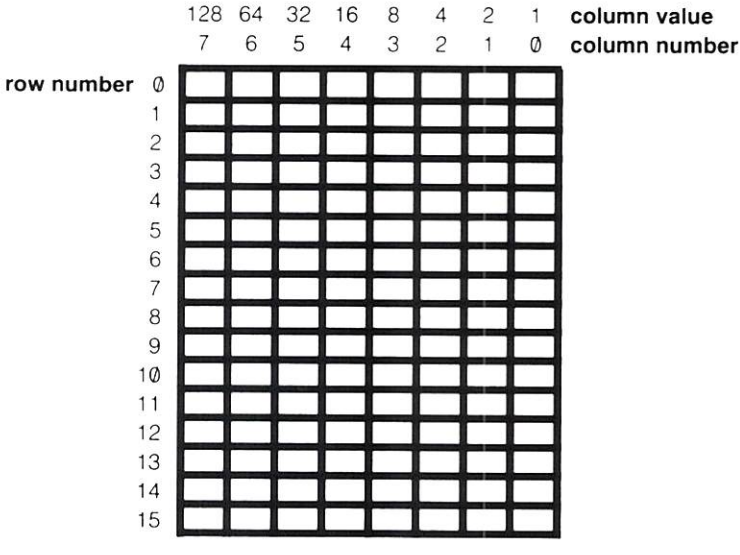
FORWARD, FD command

FORWARD *distance*

Moves the turtle forward *distance* steps in the direction in which it is heading. Note that **FORWARD 0** (with **PENDOWN**) displays a single dot at the turtle's current position without moving the turtle. It is an error if *distance* is greater than 9999.9999 or less than −9999.9999.

GETSH *shapenumber*

Outputs a list of 16 numbers representing the grid of the *shapenumber* (an integer from 1 through 15). Note that *shapenumber* can't be 0. Each shape consists of an 8 column by 16 row grid. Each element in the list is the sum of the bit values for a row of the shape.



Note that each column has a number and a value. The number is the power of 2 which corresponds to the value. For instance, 2 to the power of 2 is 4.

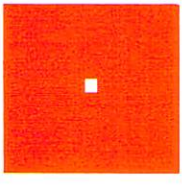
The first element in the list corresponds to the first row of the shape. If the whole row is filled in, this number is 255, the sum of all the column values. Each possible sum is unique. If only the right-most position of this row is filled in, this number is 1. If only the fifth position from the right is filled in, this number is 16.

GETSH is useful for saving shapes on a disk or cassette. You must first store the shapes in variables and then save the workspace. (See Chapter 16 in the *Introduction Manual* for *details.)*

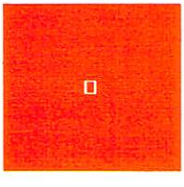
EXAMPLES

Let's suppose that shape number 1 is a filled-in box and shape number 2 is the outline of a box.

```
PR GETSH 1
255 255 255 255 255 255 255 255 255→
255 255 255 255 255 255 255
```



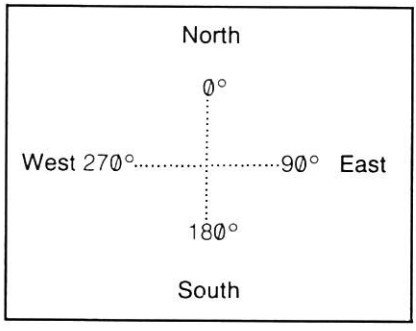
```
PR GETSH 2
255 129 129 129 129 129 129 129 129→
129 129 129 129 129 129 255
```



HEADING operation

HEADING

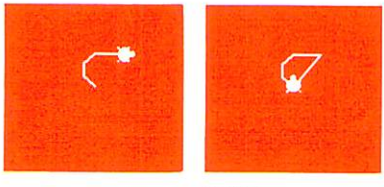
Outputs the turtle's heading, a number greater than or equal to 0 and less than 360. Logo follows the compass system where north is a heading of 0 degrees, east 90 degrees, south 180 degrees, and west 270 degrees. When you start Logo, the turtle has a heading of 0 (straight up).



HOME command

HOME

Moves the turtle to the center of the screen and sets its heading to 0. This command is equivalent to SETPOS [0 0] SETH 0. If the turtle's pen is down, the turtle draws a line from its current position to HOME.



HOME

HT command

HT

Stands for Hide Turtle. Makes the turtle invisible, although it can still draw.

LEFT, LT command

LEFT *degrees*

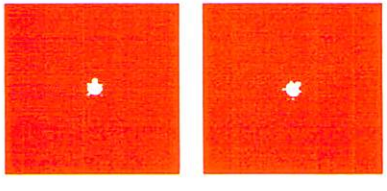
Turns the turtle left (counterclockwise) the specified number of *degrees*. It is an error if degrees is greater than 9999.9999 or less than −9999.9999.

EXAMPLES

LEFT 45 (turns the turtle 45 degrees left)

LEFT 45

LEFT −45 (turns the turtle 45 degrees right)



LEFT −45

PC operation

PC *pennumber*

Stands for Pen Color. Outputs a number representing the color of the current *pennumber* 0, 1, or 2.

When Logo first starts, PC 0 is 15 (gold), PC 1 is 47 (purple), and PC 2 is 121 (orange).

PE command

PE

Stands for Pen Erase. Puts the turtle's eraser down. When the turtle moves, it will erase any previously drawn lines it passes over. To lift the eraser, use either PD, PU, or PX.

PEN operation

PEN

Outputs a word describing the current state of the turtle's pen: PD, PU, PE, or PX. (See individual entries for further information.) When Logo first starts up, PEN outputs PD.

PENDOWN, PD command

PENDOWN

Puts the turtle's pen down: when the turtle moves, it draws a line in the current pen color. The turtle begins with its pen down.

PENUP, PU command

PENUP

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

PN — operation

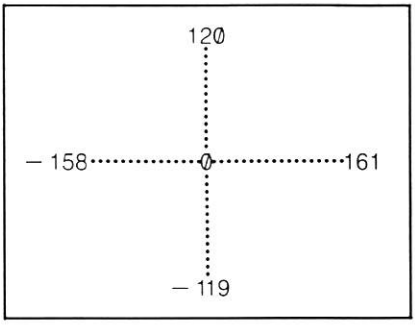
PN

Stands for Pen Number. Outputs an integer (0, 1 or 2) representing the current pen number in use. ATARI turtles can use one of the three pens to draw. When Logo starts, PN is 0. SETPN is the command to tell the turtle which pen it should use. The color of each pen can be changed by the SETPC command.

POS — operation

POS

Stands for POSition. Outputs the coordinates of the current position of the turtle in the form of a list [x y]. When you start Logo, the turtle is at [0 0], the center of the turtle field. See SETPOS for setting the turtle's position.



Due to aspect ratio this graph may not be the same. The correct aspect ratio for this graph is .SETSCR.8

PUTSH — command

PUTSH *shapenumber shapespec*

Gives *shapenumber* the specified *shapespec* as its shape. The output of GETSH can be the input *shapespec*, in PUTSH. PUTSH allows you to define shapes under program control, as an alternative to using the shape editor.

EXAMPLES

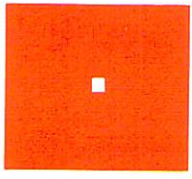
Using the **REPLACE** procedure, you can change a row in an already-defined shape.

```
TO REPLACE :POS :NEWROW :SH
IF :POS = 1 [OP SE :NEWROW BF :SH]
OP SE FIRST :SH REPLACE :POS - 1 :NEW→
ROW BF :SH
END

PR GETSH 1
255 255 255 255 255 255 255 255 255→
255 255 255 255 255 255 255
```

Shapenumber 1 is a filled-in box.

```
PUTSH 1 REPLACE 8 0 GETSH 1
```

will put an empty line in the middle.



Filled-in box

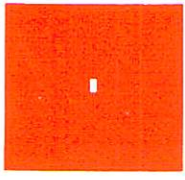
New Shape

```
TO CHANGESH :SH :POS :N
IF :POS > 16 [OP :SH]
OP SE (FIRST :SH) - :N CHANGESH BF :S→
H :POS + 1 :N
END
```

If shapenumber 1 is still a filled-in box,

```
PUTSH 1 CHANGESH GETSH 1 1 15
```

will halve the size of each row.



PX command

PX

Puts the "reversing pen" down: when the turtle moves, it draws where there aren't lines and erases where there are.

The exact effect of this reversal can be complex; what it looks like on the screen depends on the pen color, background color, and whether lines are horizontal or vertical. The best results are on a black background. To pick up the reversing pen, use PD, PU, or PE.

PX will work with SETSP but the results are very inconsistent. Using these two primitives together is not recommended.

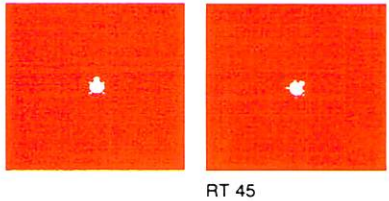
RIGHT, RT command

RIGHT *degrees*

Turns the turtle right (clockwise) the specified number of *degrees*. It is an error if *degrees* is greater than 9999.9999 or less than −9999.9999.

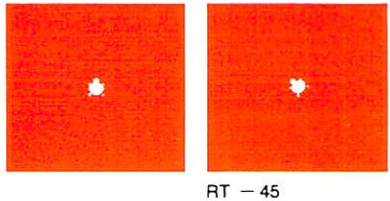
EXAMPLES

RIGHT 45 (turns the turtle 45 degrees right)



RT 45

RIGHT −45 (turns the turtle 45 degrees left)



RT −45

SETBG command

SETBG *colornumber*

Stands for SET BackGround. Sets the background color to the color represented by *colornumber*. There are 128 background colors to choose from (0 through 127).

EXAMPLE

The following procedure cycles through all the possible background colors.

```
TO CHANGEBG
IF BG = 127 [SETBG 0 WAIT 30]
SETBG 1 + BG
PR BG WAIT 30
CHANGEBG
END

CHANGEBG
```

To stop this procedure, press the BREAK key.

SETC command

SETC *colornumber*

Stands for SET turtle's Color. Sets the color of the current turtle to *colornumber* (an integer from 0 through 127).

SETH command

SETH *degrees*

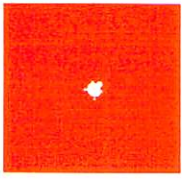
Stands for SETHeading. Turns the turtle at its position so that it is heading in the direction *degrees*. Positive numbers are clockwise from north. Note that RIGHT and LEFT produce turns relative to the turtle's heading, but SETH sets an absolute heading without reference to its prior heading. It is an error if *degrees* is greater than 9999.9999 or less than −9999.9999.

See HEADING.

EXAMPLES

```
SETH 45
SETH -45
PRINT HEADING
315
```

Heads the turtle northeast.
Heads the turtle northwest.



SETH 45

SETH −45

SETPC command

SETPC *pennumber colornumber*

Stands for SET Pen Color. Sets the color of the *pennumber* (0, 1, 2) to *colornumber* (0 through 127). You can change the color of an already-drawn shape by changing its pen number or by assigning a new *colornumber* to that particular *pennumber*.

You must assign a *pennumber* with SETPN prior to using SETPC unless you are changing the current pen number.

EXAMPLE

```
REPEAT 4 [FD 20 RT 90]
SETPC 0 120
```

If the above is done on starting Logo, the square will change color from gold to orange.

SETPN command

SETPN *pennumber*

Stands for SET Pen Number. Sets the pen, that the current turtle(s) are using, to *pennumber*. There are three pens to choose from (0, 1, 2). This determines which pen the turtle uses to draw. Use SETPC to set the pen's color. When Logo starts, the turtle(s) use(s) pen number 0.

SETPOS command

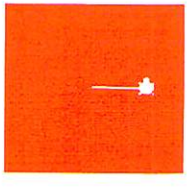
SETPOS *position*

Stands for SET POSition. Moves the turtle to the *position* indicated by a list of two numbers, [x and y coordinates]. (See POS). Both x and y take a maximum input of 9999.9999 whether in WINDOW or WRAP. If the turtle's pen is down, the turtle leaves a trace between its original and new positions.

EXAMPLE

```
SETPOS [80 0]
```

moves the turtle to a point half way down the right edge of the screen.



SETPOS [80 0]

SETSH command

SETSH *shapenumber*

Stands for SET SHape. Sets the shape of the current turtle to the shape specified by *shapenumber*, which must be an integer in the range of 0 through 15. You create your own shapes using EDSH or PUTSH. Shape 0, the turtle shape, cannot be changed. Shape numbers 1 through 15 start out blank every time Logo is booted. For more information, see the Turtle Shape Editor at the end of this chapter.

EXAMPLE

If you've changed the turtles to another shape this command changes every turtle to its normal shape:

```
TELL [0 1 2 3] SETSH 0
```

SETSP command

SETSP *speed*

Stands for SET SPeed. Sets the current turtle's *speed* (without altering its heading). If *speed* is greater than 0, the turtle will move forward. If *speed* is less than 0, the turtle will move backwards. If *speed* is equal to 0, the turtle stops moving. It is an error if *speed* is greater than 200, or less than −200. Note that SETSP's input does not need to be an integer.

EXAMPLE

This procedure makes each turtle move eastward at a random speed from 1 to 30:

```
TO EASTWARD
TELL [0 1 2 3] ST
SETH 90
EACH [SETSP 1 + RANDOM 30]
END
```

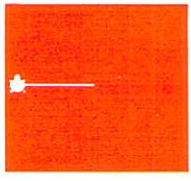
SETX command

SETX *x*

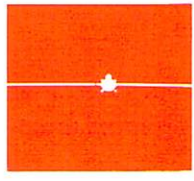
Puts the turtle at a point with x-coordinate *x* (y-coordinate is unchanged). If the turtle's pen is down, it will leave a horizontal trace.

```
SETX -158
```

moves the turtle horizontally to the left edge of the screen.



SETX – 158



SETX 2 * XCOR

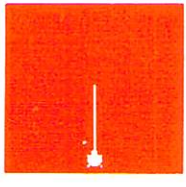
SETY command

SETY *y*

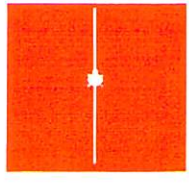
Puts the turtle at a point with y-coordinate *y* (x-coordinate is unchanged).

```
SETY -119
```

moves the turtle vertically down to the lower edge of the screen.



SETY – 119



SETY 2 * YCOR

SHAPE operation

SHAPE

Outputs the number representing the shape of the current turtle. The normal turtle shape is 0. Note that the shape numbers are not the same as the turtle numbers (see example).

EXAMPLE

```
PUTSH 12 [255 255 255 255 255 255 25→
5 255 255 255 255 255 255 255 255 25→
5]
TELL 3 SETSH 12
PRINT SHAPE
12
PRINT WHO
3
```

SHOWNP operation

SHOWNP

Outputs TRUE if the turtle is visible, FALSE otherwise. Logo thinks the turtle is visible (even if you can't see it) as long as it is within its boundaries. If the boundaries are set by WINDOW and you can't see the turtle, SHOWNP will still output TRUE.

SPEED operation

SPEED

Outputs the current turtle's speed. Note that speed is defined as turtle steps per 16/60th's of a second.

SPEED may *not* output the exact speed that you originally gave as input to SETSP. This has to do with the way Logo handles its arithmetic.

EXAMPLE

This procedure halts any turtle that is exceeding the "speed limit":

```
TO HALT.AT :SPEED.LIMIT
EACH [IF SPEED > :SPEED.LIMIT [SETSP →
0]]
END

TELL [0 1 2 3]
EACH [PU SETSP 10 + 20 * WHO]
HALT.AT 40
```

EACH is used because we want Logo to check each turtle's speed.

ST command

ST

Stands for Show Turtle. Makes the turtle visible. See also HT.

Note that if you have set the turtle's shape to an undefined shape, (GETSH outputs a list of zeros), ST will not make the turtle visible.

TELL command

TELL *turtlenumber*
TELL *turtlenumberlist*

Announces to Logo which turtle(s) you want to use. Unless you use TELL to specify otherwise, the turtle commands you give will be addressed to turtle 0.

The first time you address the other turtles with TELL after Logo starts, they appear on the screen without having used the command ST. This is the only time that TELL has this effect.

EXAMPLES

The following instructions make turtle 3 red (40) and turtles 1, 2, and 0 blue (70):

```
TELL 3
SETC 40
TELL [1 2 0]
SETC 70
```

TELL can take a list of the same turtle numbers as its input.

```
TELL [0 0 0 0]
FD 10
```

In this case FD 10 is repeated four times.

WHO operation

WHO

Outputs the turtle number(s) you are currently talking to. The output is an integer or a list of integers from 0 through 3 representing the four turtles available in ATARI Logo.

This operation is useful with the EACH command when you want each turtle to carry out different instructions at the same time.

EXAMPLES

```
TELL [1 2]
PR WHO
1 2
```

The following instructions make the four turtles go forward in four directions at once.

```
TELL [0 1 2 3]
HOME
EACH [SETH 90 * WHO]
FD 30
```

WINDOW command

WINDOW

Makes the turtle field unbounded; what you see is a portion of the turtle field as if you were 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 entire turtle field is 25119 steps high and 19841 steps wide. To hit the boundaries of **WINDOW**, you must repeat **FD** (or **BK**) a number of times with its highest input (9999.9999). When you give the **WINDOW** command, the screen is cleared. See also **WRAP**.

To create a smaller turtle field, use collision detection.

EXAMPLE

```
WINDOW
CS RT 5
FD 500
PRINT POS
43.57787 498.09735
```

WRAP command

WRAP

Makes the turtle field wrap around the edges of the screen: if the turtle moves beyond one edge of the screen it appears and continues from the opposite edge. The turtle never leaves the visible bounds of the screen; when it tries to, it "wraps around". Thus, the turtle can move FORWARD (or BACK) an infinite amount of times without hitting the limits of the turtle field.

When you give the WRAP command, the screen is cleared. See also WINDOW.

EXAMPLE

```
WRAP
CS RT 5
FD 500
PRINT POS
43.57787 18.09735
```

XCOR operation

XCOR

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

YCOR operation

YCOR

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

EXAMPLES

```
CS PRINT YCOR
0
FD 100
PRINT YCOR
100
```

The following procedure outputs the sine of an angle. The result is equivalent to the primitive SIN.

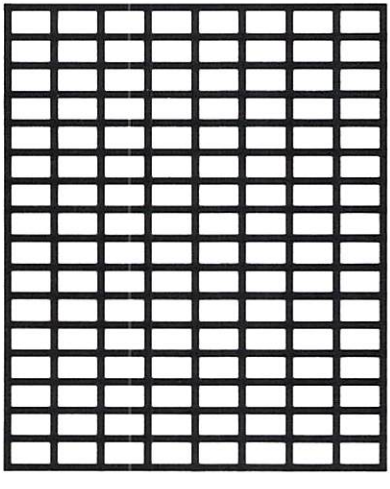
```
TO SINE :ANGLE
HOME
SETH 90
LEFT :ANGLE
FORWARD 100
OUTPUT YCOR / 100
END

PRINT SINE 30
0.500002136 2
```

Turtle Shape Editor

You can create as many shapes as you want using the shape editor. But there are sixteen possible turtle shapes available at one time. Shape 0, the turtle shape cannot be changed.

EDSH is the command to start the Logo shape editor. Its input is the *shapenumber* (1 through 15). These shapes start out blank every time Logo starts up. EDSH brings that *shapenumber* into the editor. Note that if you are defining *shapenumber* for the first time, the shape will be a large rectangular grid made out of 8 by 16 small empty boxes. For example:

```
EDSH 1
```



There is no prompt character, but the *cursor* shows you where you are working. When you enter the shape editor, the *cursor* is in the top left box.

Once you start the shape editor, you can move the *cursor* anywhere in the shape. You are able to pass over the boxes, and can create a shape by filling in the boxes or erasing them again using the SPACE BAR.

Moving the Cursor and Changing the Shape

Use the CTRL arrow keys to move the *cursor* around without changing the shape. To change what is under the *cursor*, press the SPACE BAR: a blank spot will become filled and a filled spot will become blank. This is how you define your shape. Remember to position the *cursor* before pressing the SPACE BAR.

CTRL →	Moves the *cursor* right one space.
CTRL ←	Moves the *cursor* left one space.
CTRL ↑	Moves the *cursor* up one line.
CTRL ↓	Moves the *cursor* down one line.

On those ATARI Home Computers that are equipped with them, the function keys F1, F2, F3 and F4 can be used to control the *cursor*.

Do not press the ATARI key(🜲)or reverse video key(◪) while in the shape editor. It disables the SPACE BAR function.

Leaving the Shape Editor

To leave the shape editor, press either ESC or BREAK.

ESC	Exits the shape editor saving the changes you have made.
BREAK	Aborts the shape editor without saving any changes.

See Chapter 16 of the *Introduction Manual* for an example of using the Shape Editor.

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.

EXAMPLES

HELLO
X
314
3.14
R2D2
PIGLATIN
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 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 "\" before it.

EXAMPLE

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

Note that quotation marks (") and colon (:) are not word delimiters.

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 a quotation mark, ", followed by a space. See entry for **EMPTYP** for examples of both the empty *list* and the empty word.

The operations FIRST, BUTFIRST (BF), LAST and BUTLAST (BL), are used to break words and lists into pieces. The following chart shows how they work. If you want to try out the operations in the table below use the command SHOW.

Operation	Input	Output
FIRST	"JOHN	J
BF	"JOHN	OHN
FIRST	[MARY JOHN BILL]	MARY
BF	[MARY JOHN BILL]	[JOHN BILL]
FIRST	[[MARY JOHN] BILL]	[MARY JOHN]
BF	[[MARY JOHN] BILL]	[BILL]
FIRST	[ ] or "	error
BF	[ ] or "	error

LAST and BUTLAST (BL) work in the same way separating the last element.

Logo uses five operations to put words and lists together. These are FPUT, LPUT, LIST, SE, and WORD. The following chart compares these five primitives:

Operation	input 1	input 2	output
FPUT	"LOGO	"TIME	error
LIST	"LOGO	"TIME	[LOGO TIME]
LPUT	"LOGO	"TIME	error
SE	"LOGO	"TIME	[LOGO TIME]
WORD	"LOGO	"TIME	LOGOTIME
FPUT	[AND MORE]	[TO COME]	[[AND MORE] TO COME]
LIST	[AND MORE]	[TO COME]	[[AND MORE] [TO COME]]
LPUT	[AND MORE]	[TO COME]	[TO COME [AND MORE]]
SE	[AND MORE]	[TO COME]	[AND MORE TO COME]
WORD	[AND MORE]	[TO COME]	error

ASCII operation

ASCII *character*

Outputs the ASCII code for *character*. Appendix F contains a chart of all ASCII codes. If the input word contains more than one character, ASCII uses only its first character. See also CHAR.

EXAMPLE

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 CODE FIRST :WD SECRETCODE→
BF :WD
END

TO CODE :LET
MAKE "NUM (ASCII :LET) + 3
IF :NUM > ASCII "Z [MAKE "NUM :NUM - →
26]
OUTPUT CHAR :NUM
END

PRINT SECRETCODE "CAT
FDW

PRINT SECRETCODE "CRAYON
FUDBRQ
```

BUTFIRST *object*

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

EXAMPLES

```
SHOW BF [BRIAN J. SMITH]
[J. SMITH]

SHOW BF "DOGS
OGS

SHOW BF [DOGS]
[ ]                                The empty list

SHOW BF [[THE A AN] [DOG CAT MOUSE] →
[BARKS MEOWS]]
[[DOG CAT MOUSE] [BARKS MEOWS]]

PRINT BF "
BF DOESN'T LIKE   AS INPUT

PRINT BF [ ]
BF DOESN'T LIKE [ ] AS INPUT
```

The following procedure removes one element at a time from a word or a list.

```
TO TRIANGLE :MESSAGE
IF EMPTYP :MESSAGE [STOP]
PRINT :MESSAGE
TRIANGLE BF :MESSAGE
END

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

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

BUTLAST, BL — operation

BUTLAST *object*

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

EXAMPLES

```
SHOW BL [I YOU HE SHE IT]
[I YOU HE SHE]

SHOW BL "FLOWER
FLOWE

SHOW BL [FLOWER]
[ ]
```

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

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

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

CHAR *n*

Outputs the character whose ASCII code is *n*, an integer from 0 through 255. Appendix F contains a chart of all ASCII codes.

The ASCII codes are organized as follows:

0	—	31	graphic characters
32	—	47	punctuation
48	—	57	digits
58	—	64	punctuation
65	—	90	upper-case alphabet
91	—	96	punctuation
97	—	122	lower-case alphabet
123	—	127	graphic characters
128	—	255	reverse video of characters 0 to 127

EXAMPLE

```
TO LOWERCASE :LETTER
MAKE "LC 32 + ASCII :LETTER
IF AND :LC > 96 :LC < 123 [OP CHAR :L→
C] [OP :LETTER]
END
```

This procedure outputs the lowercase of an alphabet character. If you give it a character other than a letter of the alphabet, it outputs the same character.

```
PRINT LOWERCASE "A
a

PRINT LOWERCASE "R
r
```

COUNT operation

COUNT *object*

Outputs the number of elements in a word or a list.

EXAMPLES

```
PRINT COUNT [A QUICK BROWN FOX]
4

PRINT COUNT [A [QUICK BROWN] FOX]
3

PRINT COUNT "COMPUTER
8

MAKE "CLASS [PAT JENNY CHRIS SCOT TO→
M MARY JUDY]
PRINT COUNT :CLASS
7
```

The following procedure prints a random element of its input:

```
TO RANPICK :DATA
PR ITEM (1 + RANDOM COUNT :DATA) :DAT→
A
END

TO ITEM :N :OBJECT
IF :N = 1 [OUTPUT FIRST :OBJECT]
OUTPUT ITEM :N - 1 BF :OBJECT
END

RANPICK :CLASS                    see list CLASS above
SCOT

RANPICK "COMPUTER
M
```

EMPTYP *object*

Outputs TRUE if *object* is the empty word or the empty list; otherwise FALSE.

EXAMPLES

```
PR EMPTYP "
TRUE

PR EMPTYP 0
FALSE

PR EMPTYP BF "UNICORN
FALSE

PR EMPTYP BL "U
TRUE

PR EMPTYP BF [UNICORN]
TRUE
```

The procedure, TALK, matches animal sounds to animals:

```
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!
```

EQUALP *operation*

EQUALP *object1 object2*

Outputs TRUE if *object1* and *object2* are equal numbers, identical words, or identical lists; otherwise outputs FALSE. Equivalent to =, an infix operation.

EXAMPLES

```
PR EQUALP "RED FIRST [RED YELLOW]
TRUE

PR EQUALP 100 50 * 2
TRUE

PR EQUALP [THE A AN] [THE A]
FALSE

PR EQUALP " [ ]
FALSE
```

(The empty word and the empty list are not identical.)

The following operation outputs the position that the first input has in the second input and outputs 0 if it is not an element of the second.

```
TO RANK :ONE :ALL
IF EMPTYP :ALL [OUTPUT 0]
IF EQUALP :ONE LAST :ALL [OUTPUT COUN→
T :ALL]
OUTPUT RANK :ONE BL :ALL
END

PRINT RANK "TWO [ONE TWO THREE]
2

PRINT RANK "S "PERSONAL
4
```

FIRST operation

FIRST *object*

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

EXAMPLES

```
SHOW FIRST "HOUSE
H

SHOW FIRST [HOUSE]
HOUSE

SHOW FIRST [[THE A AN] [UNICORN RHIN→
O] [SWIMS FLIES GROWLS RUNS]]
[THE A AN]
```

The procedure ITEM outputs the :Nth element of its second input.

```
TO ITEM :N :OBJECT
IF :N = 1 [OUTPUT FIRST :OBJECT]
OUTPUT ITEM :N - 1 BF :OBJECT
END

PR ITEM 3 [CUP PUT TUB BUD]
TUB

PR ITEM 4 "STRAWBERRY
A
```

FPUT — operation

FPUT *object list*

Stands for First PUT. Outputs a new list formed by putting *object* at the beginning of *list*. See the chart at the beginning of this chapter comparing FPUT with other operations that combine words and lists.

EXAMPLE

The procedure REV puts the elements of the input list in reverse order.

```
TO REV :LIST
IF EMPTYP :LIST [OUTPUT [ ]]
OUTPUT FPUT LAST :LIST REV BL :LIST
END

SHOW REV [[FD 20] PU [RT 90] [FD 20]→
 PD [BK 20]]
[[BK 20] PD [FD 20] [RT 90] PU [FD 2→
0]]
```

LAST — operation

LAST *object*

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

EXAMPLES

```
SHOW LAST [JUDY SUSAN BRIAN]
BRIAN

SHOW LAST "VANILLA
A

SHOW LAST [[THE A] FLAVOR IS [VANILL→
A CHOCOLATE STRAWBERRY]]
[VANILLA CHOCOLATE STRAWBERRY]
```

The following procedure prints a word in reverse order.

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

PRINTBACK "REVERSE
ESREVER
```

LIST operation

LIST *object1 object2*

Outputs a list whose elements are *object1*, *object2*. Each input of LIST can be a word or a list.

EXAMPLES

```
SHOW LIST "ROSE [TULIP IRIS]
[ROSE [TULIP IRIS]]

SHOW LIST [ROSE] [TULIP IRIS]
[[ROSE] [TULIP IRIS]]
```

LISTP — operation

LISTP *object*

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

EXAMPLES

```
PRINT LISTP 3
FALSE

PRINT LISTP [3]
TRUE

PRINT LISTP [ ]
TRUE

PRINT LISTP "
FALSE

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

PRINT LISTP BF "CHOCOLATE
FALSE

PRINT LISTP BF [CHOCOLATE]
TRUE
```

LPUT *object list*

Stands for Last PUT. Outputs a new list formed by putting *object* at the end of *list*. See chart at the beginning of the chapter comparing LPUT with other primitives that combine words and lists.

EXAMPLE

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 operation

MEMBERP *object list*

Outputs TRUE if *object* is an element of *list*; otherwise outputs FALSE.

EXAMPLES

```
PRINT MEMBERP 3 [2 5 3 6]
TRUE

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

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

PRINT MEMBERP BF "FOG [OE FO OG OH]
TRUE
```

The following procedure determines whether its input is a vowel:

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

PRINT VOWELP "F
FALSE

PRINT VOWELP "A
TRUE
```

NUMBERP — operation

NUMBERP *object*

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

EXAMPLES

```
PRINT NUMBERP 3
TRUE

PRINT NUMBERP [3]
FALSE

PRINT NUMBERP "7PM
FALSE

PRINT NUMBERP ""
FALSE

PRINT NUMBERP BF 3165.2
TRUE
```

SE — operation

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

Stands for SEntence. Outputs a list made up of the elements included in its inputs. See the chart at the beginning of this chapter comparing SE with other operations that combine words and lists.

EXAMPLES

```
SHOW SE "PAPER "BOOKS
[PAPER BOOKS]

SHOW SE "APPLE [PEAR PLUM BANANA]
[APPLE PEAR PLUM BANANA]

SHOW SE [TIME AND TIDE] [WAIT FOR NO→
 PERSON]
[TIME AND TIDE WAIT FOR NO PERSON]
```

If SE has more than two inputs, you must enclose SE and its inputs in parentheses.

```
SHOW (SE "HOP "STEP "JUMP)
[HOP STEP JUMP]

SHOW SE "BONNIE [ ]
[BONNIE]
```

The following procedure prints a birth anouncement:

```
TO ANNOUNCE :FIRSTNAME :LASTNAME
PRINT [WE'RE HAPPY TO ANNOUNCE THE BI→
RTH OF]
PRINT (SE :FIRSTNAME "Q. :LASTNAME)
PRINT [5 POUNDS 14 OZ]
END

ANNOUNCE "RALPH "DOE
WE'RE HAPPY TO ANNOUNCE THE BIRTH OF
RALPH Q. DOE
5 POUNDS 14 OZ
```

WORD — operation

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

Outputs a word made up of its inputs. If WORD has more than two inputs, you must enclose WORD and its inputs in parentheses. WORD does not work with a list as its input.

EXAMPLES

```
PRINT WORD "SUN "SHINE
SUNSHINE

PRINT (WORD "CHEESE "BURG "ER)
CHEESEBURGER

PRINT WORD "BURG [ER]
WORD DOESN'T LIKE [ER] AS INPUT
```

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

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

PRINT 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] [O→
P WORD :WORD "AY]
OP PIG WORD BF :WORD FIRST :WORD
END

PRINT LATIN [NO PIGS HAVE EVER SPOKE→
N PIG LATIN AMONG HUMANS]
ONAY IGSPAY AVEHAY EVERAY OKENSPAY I→
GPAY ATINLAY AMONGAY UMANSHAY
```

WORDP operation

WORDP *object*

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

EXAMPLES

```
PRINT WORDP "ZAM
TRUE

PRINT WORDP 3
TRUE

PRINT WORDP [3]
FALSE

PRINT WORDP [E GRESS]
FALSE
```

= (Equal Sign) — infix operation

object1 = *object2*

Outputs TRUE if *object1* and *object2* are equal numbers, identical words, or identical lists; otherwise outputs FALSE. Equivalent to EQUALP, a prefix operation.

```
PRINT 3 = FIRST "3.1416
TRUE

PRINT [THE A AN] = [THE A]
FALSE

PRINT 7. = 7
TRUE
```

A decimal number is equivalent to the corresponding integer.

```
PRINT '' = [ ]
FALSE
```

The empty word and the empty list are not identical.

Chapter 3

Variables

A Logo word can be used as a *variable*; a variable is a "container" that holds a Logo object. This object is called the word's *value*. A variable can be assigned a value either by using MAKE or by using procedure *inputs*.

MAKE command

MAKE *name object*

Creates the variable *name* and gives it the value *object*. Once the variable is created, you can have access to its value by THING *name*. The abbreviation *:name* means THING *"name*. The : (colon) means "the thing that is called".

EXAMPLES

```
MAKE "NATIONS [CANADA USA FRANCE GER→
MANY ITALY]
PRINT :NATIONS
CANADA USA FRANCE GERMANY ITALY

PRINT "NATIONS
NATIONS

PRINT THING "NATIONS
CANADA USA FRANCE GERMANY ITALY

MAKE "USA [WASHINGTON]
PRINT :USA
WASHINGTON

PRINT THING FIRST BUTFIRST :NATIONS
WASHINGTON
```

FIRST BUTFIRST :NATIONS is the second word in the nation list, which is USA, and the value of THING *"*USA is WASHINGTON.

```
MAKE "CANADA [OTTAWA]
PRINT FIRST :NATIONS
CANADA

PRINT THING FIRST :NATIONS
OTTAWA
```

The following procedure CAPITAL asks for the capital cities of given countries.

```
TO CAPITAL :NATIONS
IF EMPTYP :NATIONS [STOP]
MAKE "COUNTRY FIRST :NATIONS
PR SE [THE CAPITAL OF] :COUNTRY
MAKE "ANSWER RL
IF :ANSWER = THING :COUNTRY [PR [CORR→
ECT!]] [PR [OH! SINCE WHEN?]]
CAPITAL BF :NATIONS
END

CAPITAL :NATIONS
THE CAPITAL OF CANADA
OTTAWA
CORRECT!
THE CAPITAL OF USA
NEW YORK
OH! SINCE WHEN?
```

NAMEP — operation

NAMEP *name*

Outputs TRUE if *name* has a value, that is, if *:name* exists, FALSE otherwise.

EXAMPLES

```
PRINT :ANIMAL
ANIMAL HAS NO VALUE

PRINT NAMEP "ANIMAL
FALSE

MAKE "ANIMAL "AARDVARK
PRINT NAMEP "ANIMAL
TRUE

PRINT :ANIMAL
AARDVARK
```

The procedure INC listed under THING below shows a use of NAMEP.

THING operation

THING *name*

Outputs the thing (or value) associated with the variable *name*. THING "ANY is equivalent to :ANY. The variable can be created by the command MAKE or by defining a procedure with inputs.

EXAMPLES

```
MAKE "WINNER "COMPUTER
MAKE "COMPUTER [100 POINTS]
PRINT THING "WINNER
COMPUTER

PRINT :WINNER
COMPUTER

PRINT THING :WINNER
100 POINTS
```

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 + THIN→
G :X]
END
```

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

```
MAKE "TOTAL 7
PRINT :TOTAL
7

INC "TOTAL
PRINT :TOTAL
8

INC "TOTAL
PRINT :TOTAL
9
```

For other examples, see entry for MAKE.

Chapter 4

Arithmetic Operations

Logo has *integer* and *decimal* numbers:

3 is an integer.
3.14 is a decimal number.

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

Some arithmetic operations (INT, RANDOM, REMAINDER, ROUND) always output integers while others vary by the result of the operation.

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;
2.59E-2 means 2.59 times 10^{-2}, or 0.0259

Exponents range from −99 to 97.

Logo truncates a decimal number if it contains more than nine digits. For example, the number 2718281828459.045 is converted to 2.71828182E+12.

Addition, subtraction, multiplication, and division are available in *infix notation*; that is, the operation 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 = (Equal Sign) is equivalent to EQUALP.

COS operation

COS *n*

Outputs the cosine of *n* degrees. It is an error if *n* is greater than 9999.9999 or less than −9999.9999

EXAMPLES

```
PRINT COS 45
0.70714

PRINT COS 30
0.86605
```

Here is a definition of the tangent function:

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

PRINT TAN 45
1
```

INT operation

INT *n*

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

EXAMPLES

```
PRINT INT 5.2129
5

PRINT INT 5.5129
5

PRINT INT 5
5

PRINT INT -5.8
-5

PRINT INT -12.3
-12
```

The procedure INTP tells whether its input is an integer:

```
TO INTP :N
IF NOT NUMBERP :N [OUTPUT [NOT A NUMB→
ER]]
OUTPUT :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 operation

PRODUCT *a b*
(PRODUCT *a b c . . .*)

Outputs the product of its inputs. Equivalent to *, an infix operation. If PRODUCT has more than two inputs, you must put parentheses around PRODUCT and its inputs.

EXAMPLES

```
PRINT PRODUCT 6 2
12

PRINT (PRODUCT 2 3 4)
24

PRINT PRODUCT 2.5 4
10

PRINT PRODUCT 2.5 2.5
6.25
```

RANDOM operation

RANDOM *n*

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

PRINT D6
3

PRINT D6
5

PRINT D6
6
```

Note: The outputs of D6 printed here are just possible numbers and will change because of RANDOM.

REMAINDER operation

REMAINDER *a b*

Divides *a* by *b* and outputs the remainder obtained. It is an error if *b* is 0.

EXAMPLES

```
PRINT REMAINDER 13 5
3
```

13 divided by 5 is 2 and the remainder is 3.

```
PRINT REMAINDER 13 15
13

PRINT REMAINDER -13 5
-3
```

The following procedure tells you whether its input is even:

```
TO EVENP :NUMBER
OUTPUT Ø = REMAINDER :NUMBER 2
END

PRINT EVENP 5
FALSE

PRINT EVENP 12462
TRUE
```

RERANDOM command

RERANDOM

Makes RANDOM behave reproducibly. Once you run RERANDOM, Logo will remember the sequence of numbers obtained by the next RANDOM calls. After that, each time you run RERANDOM, RANDOM restarts the same sequence of random numbers from the beginning. (The input to RANDOM must be the same as the first time RERANDOM was run.)

EXAMPLES

```
REPEAT 4 [PR RANDOM 1Ø]
5
2
8
4

RERANDOM REPEAT 4 [PR RANDOM 1Ø]
8
2
3
2

RERANDOM REPEAT 4 [PR RANDOM 1Ø]
8
2
3
2
```

ROUND operation

ROUND *n*

Outputs *n* rounded off to the nearest integer. Compare with examples under INT.

EXAMPLES

```
PRINT ROUND 5.2129
5

PRINT ROUND 5.5129
6
```

INT works differently.

```
PRINT INT 5.5129
5

PRINT ROUND .5
1

PRINT ROUND -5.8
-6

PRINT ROUND -12.3
-12
```

SIN operation

SIN *n*

Outputs the sine value of *n* degrees. See also COS.

EXAMPLE

```
PRINT SIN 45
0.70714
```

SQRT — operation

SQRT *n*

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

EXAMPLES

```
PRINT SQRT 25
5

PRINT SQRT 259
16.093477
```

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

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

FD 50
PR FROM.HOME
50
```

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

```
TO DISTANCE :POS1 :POS2
MAKE "X (FIRST :POS1)-FIRST :POS2
MAKE "Y (LAST :POS1)-LAST :POS2
OUTPUT SQRT :X * :X + :Y * :Y
END

PRINT DISTANCE [-70 10] [50 60]
129.9999
```

SUM — operation

SUM *a b*
(SUM *a b c* ...)

Outputs the sum of its inputs. Equivalent to +, an infix operation. If SUM has more than two inputs, SUM and its inputs must be enclosed in parentheses.

EXAMPLES

```
PRINT SUM 5 2
7

PRINT (SUM 1 3 2 -1)
5

PRINT SUM 2.3 2.561
4.861
```

+ (Plus Sign) — infix operation

a + *b*

Outputs the sum of its inputs, *a* and *b*. This is equivalent to SUM, a prefix operation.

EXAMPLES

```
PRINT 5 + 2
7

PRINT 1 + 3 + 2 + 1
7

PRINT 2.54 + 12.3
14.84
```

– (Minus Sign) — infix operation

a – *b*

Outputs the result of subtracting *b* from *a*. It may be used as the sign for a negative number.

EXAMPLES

```
PRINT 7 - 1
6

PRINT 7-1
6

PRINT PRODUCT 7 -1
-7

PRINT -XCOR
-50                    This number varies according to the
                       turtle's position.
PRINT - 3
-3

PRINT -3 - -2
-1
```

Note that there could be a confusion between the negative sign with one input and the minus sign with two inputs. Logo resolves this as follows:

```
PRINT 3 * -4           3 times negative 4
-12

PRINT 3 + 4 - 5        3 plus 4 minus 5
2
```

If there is a space before the " – " and a number immediately after it, Logo reads that as a negative number. So 7 – 1 is 6 but 7 –1 is the pair of numbers 7 and –1.

The procedure ABS outputs the absolute value of its input:

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

PRINT ABS -35
35

PRINT ABS 35
35
```

* (Multiplication Sign) — infix operation

a * *b*

Outputs the product of its inputs *a* and *b*. This is equivalent to PRODUCT, a prefix operation.

EXAMPLES

```
PRINT 6 * 2
12

PRINT 2 + 3 * 4
14

PRINT 1.3 * -1.3
-1.69

PRINT 48 * (.3 + .2)
24
```

The procedure FACTORIAL outputs the factorial of its input. For example, FACTORIAL 5 outputs the result of 5 * 4 * 3 * 2 * 1 (120).

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

PRINT FACTORIAL 4
24

PRINT FACTORIAL 1
1
```

/ (Division Sign) — infix operation

a / *b*

Outputs the result of *a* divided by *b*.

EXAMPLES

```
PRINT 6 / 3
2

PRINT 8 / 3
2.66666666

PRINT 2.5 / -3.8
-0.6578947368

PRINT 0 / 7
0
```

It gives an error if *b* is 0.

```
PRINT 7 / 0
/ DOESN'T LIKE 0 AS INPUT
```

< (Less Than Sign) — infix operation

a < *b*

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

EXAMPLES

```
PRINT 2 < 3
TRUE

PRINT -7 < -10
FALSE
```

= (Equal Sign) infix operation

a = *b*

Outputs TRUE if *a* and *b* are equal numbers, identical words, or identical lists; otherwise outputs FALSE. Equivalent to EQUALP, a prefix operation.

EXAMPLES

```
PRINT 100 = 50 * 2
TRUE

PRINT 3 = FIRST "3.1416
TRUE

PRINT 7. = 7
TRUE
```

A decimal number is equivalent to the corresponding integer.

```
PRINT "" = []
FALSE
```

The empty word and the empty list are not identical.

> (Greater Than Sign) infix operation

a > *b*

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

EXAMPLES

```
PRINT 4 > 3
TRUE
```

The procedure BETWEEN outputs TRUE if the number given as the first input is greater than the second input and less than the third.

```
TO BETWEEN :N :LOW :HI
OP AND :N > :LOW :HI > :N
END

PRINT BETWEEN 15 0 16
TRUE

PRINT BETWEEN -5 -2 5
FALSE
```

Chapter 5

Defining and Editing Procedures

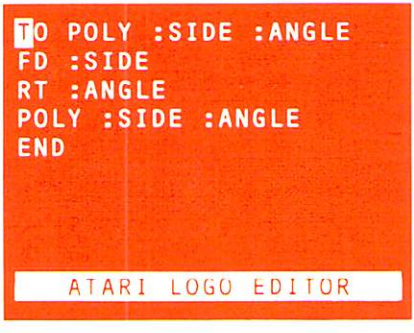
There are two ways to define procedures. One way is with TO and the other is with EDIT. TO allows you to define a new procedure at top level without disrupting the graphics screen. EDIT allows you to use an interactive screen-oriented text editor, but at the same time you lose your graphics. You may define more than one procedure at a time in the editor. This is more flexible and convenient when you need to make some modifications. Although the editor is more extensively used, each method has its advantages and it is up to you to decide which one to use.

ATARI Logo Editor

How the Editor Works

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.

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

You can have more characters on a line of text than fit across the screen. When you get to the end of the line on the screen, simply continue typing without pressing the RETURN key. An arrow (→) will appear at the end of the line (column 37) and the *cursor* will move to the next line. Logo does the same thing outside of the editor. While in the editor you can type lines of any length. Outside the editor, a Logo line has a maximum length of 110 characters.

This is how a long line would appear on the screen:

```
TO PRINTMESSAGE :PERSON
PRINT SE :PERSON [, I AM GOING TO TYP→
E A VERY LONG MESSAGE FOR YOU]
END
```

The editor has a line buffer called the *delete buffer.* SHIFT DELETE BACK S deletes a line of text and puts it in this buffer. CTRL Y reinserts this line of text later at the place marked by the *cursor.* The delete buffer can hold a maximum of 110 characters.

The text that you edit is in an *edit buffer.* The buffer has a capacity of 3840 characters.

The arrow keys, CTRL or SHIFT function keys and some CTRL character key combinations have special meanings to help you edit.

Editing Actions

When you are in the editor, you can use the following editing keys:

*The star represents editing keys that work both inside and outside the ATARI Logo Editor.

Cursor Motion

*CTRL →	Moves the *cursor* right one space.
*CTRL ←	Moves the *cursor* left one space.
CTRL ↓	Moves the *cursor* down to the next line.
CTRL ↑	Moves the *cursor* up to the previous line.
*CTRL A	Moves the *cursor* to the beginning of the current line.
*CTRL E	Moves the *cursor* to the end of the current line.
CTRL X	Moves the *cursor* to the beginning of the editor.
CTRL Z	Moves the *cursor* to the end of the editor.

Note: Any time you try to make the *cursor* go where there is no text, Logo will beep.

Inserting and Deleting

*RETURN	RETURN creates a new line at the current *cursor* position and moves the *cursor* to the beginning of the new line.
*CTRL INSERT	Opens a new line at the position of the *cursor* but does not move the *cursor*.
*DELETE BACK S	Erases the character to the left of the *cursor*.
*CTRL DELETE BACK S	Erases the character *at* the *cursor* position. Compare with DELETE BACK S.
*CTRL CLEAR	Deletes text from the *cursor* position to the end of the current line. This text is placed in the delete buffer, which can hold up to 110 characters.
*CTRL Y	Inserts the text that is currently in the delete buffer.
*SHIFT DELETE BACK S	Same as CTRL CLEAR.
*SHIFT INSERT	Same as CTRL INSERT.

Scrolling the Screen

*CTRL 1	Makes Logo stop scrolling until CTRL 1 is pressed again.
CTRL V	Scrolls the screen to the next page in the editor.
CTRL W	Scrolls the screen back to the previous page in the editor.

Exiting From the Editor

ESC

ESC is the standard way to exit from the editor.

When you exit from the editor by pressing ESC, Logo reads each line in the edit buffer as though you had typed it outside the editor.

If you forgot to type the END at the end of the definition, Logo inserts END for you.

You can define more than one procedure while in the editor, as long as each procedure is terminated by END.

If there are Logo instructions in the edit buffer that are not contained in the procedure definition (within TO . . . END), Logo carries them out as you exit from the editor just as if you had typed them in at top level (outside the editor). Logo will not carry out any graphics commands or editing commands.

BREAK

Aborts editing. Use it if you don't like the changes you are making, or if you decide not to make changes. If you were defining a procedure, the definition will be the same as before you started editing.

EDIT, ED command

EDIT *name*
EDIT *namelist*

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

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

Press the ESC key to complete the definition and exit the editor. Logo reads every line from the edit buffer as though you had typed it outside the editor. If the end of the buffer is reached while there is a procedure definition in the editor, Logo completes the procedure definition and inserts END.

EDNS — command

EDNS

Stands for EDit NameS. Starts up the Logo Editor with all names and their values in it. These variables' names and values can then be edited. When you exit the editor the MAKE commands are run, so whatever variables and values have been changed in the editor are changed in Logo.

EXAMPLE

Type

```
EDNS
```

The screen now looks like:

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

Edit the names so they will look like the list below. Then press ESC to exit the editor.

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

Then

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

END — special word

END

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. END also must be used to separate procedures when defining multiple procedures in the Logo Editor.

TO command

TO *name input1 input2...*

```
?TO GREET
>PRINT [HI THERE]
>END
GREET DEFINED
?
```

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

TO tells Logo that you are defining a procedure called *name*, with inputs (if any) as indicated. (It is not necessary to quote *name*, since TO quotes it automatically.) The prompt changes from "?" to ">" to remind you that you are defining a procedure. While defining a procedure Logo does not carry out the instructions that you type; it makes them part of the procedure definition.

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

If you change your mind while defining a procedure with TO, press the BREAK key to abort the definition. If a procedure is already defined, you can't change the definition with TO. You must use EDIT or erase the old definition first with ER.

Chapter 6

Flow of Control and Conditionals

Logo reads procedure definitions line by line, following the instructions. If a procedure contains a subprocedure, Logo reads the lines of the subprocedure before continuing in the superprocedure. Flow of control refers to the order in which Logo follows instructions. There are times you want to alter Logo's normal flow of control. There are several ways to do it.

conditionals	"if such-and-such is true, do one thing; otherwise, do something else."
repetition	"run a list of instructions one or more times."
halting	"stop this procedure before it reaches the **END**."

Conditionals enable Logo to carry out different instructions, depending on whether a condition is met. Logo predicates, operations that output **TRUE** or **FALSE**, create this condition, which is the first input to **IF**.

Repetition can be done by using **REPEAT** or a recursive procedure. There are many examples of such procedures throughout this manual. (See **RUN** for examples of some complex repetitive procedures.)

You can *halt* a procedure before it reaches an **END** statement. The commands **STOP** and **OUTPUT** are used for this. Control is then transferred back to its calling procedure (the procedure using it) or to top level. As described in *Logo Grammar*, **OUTPUT** can communicate information to the calling procedure. Note that these commands (**STOP** and **OUTPUT**) only halt the procedure they appear in.

The **WHEN** demon is a completely different way to alter the flow of control. It is a global condition that needs to be set up only once. Whenever that condition is met within any procedure, or at top level, a set of instructions is run.

You can think of the **WHEN** demon as sitting inside the Logo world, spending all its time watching for a certain event. Whenever this event occurs, it jumps up and tells Logo to run a list of instructions. Then the **WHEN** demon resumes its watch.

The primitive that sets up the demon is called WHEN. The events that WHEN can check are listed in the following table. If you forget the number corresponding to an event, there are two primitives, OVER and TOUCHING, that can help you (see Appendix B in the *Introduction Manual* for examples).

COND is another primitive which can check these events. Unlike WHEN, COND can only check an event at the moment Logo reads the line containing it.

Table of Collisions and Events

Code Number		Inputs		
Collision	Special event	Turtle number	Pen number	Description of event
0		0	0	
1		0	1	
2		0	2	
	3			Button on Joystick is pressed
4		1	0	
5		1	1	
6		1	2	
	7			Once per second
8		2	0	
9		2	1	
10		2	2	
11				Not used
12		3	0	
13		3	1	
14		3	2	
	15			Joystick position is changed

Collision number	Turtle number	Turtle number
16	3	0
17	3	1
18	3	2
19	0	1
20	0	2
21	1	2

Each number in the table is a symbol for a collision or event. For example, collision number 0 stands for a collision between turtle 0 and a line drawn by pen number 0. Event number 3 stands for whether a button on a joystick is pressed.

Note: It is best to work with a full screen of graphics (FS) when using WHEN demons 2, 6, 10 and 14. In SS (splitscreen), the turtle(s) may collide with text.

COND operation

COND *condnumber*

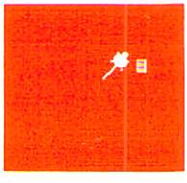
Outputs TRUE if the collision or event specified by *condnumber* is happening at the exact time COND is run, otherwise FALSE. The input is an integer between 0 and 21 indicating which collision or event you want to check (see the table page 104). COND is most useful when you want to check for an event only once. Compare with WHEN.

EXAMPLE

In the following example, shape number 1 is a filled-in box that acts as a target. COND checks whether you have hit the target by looking for a collision between turtle 0 and turtle 1.

```
TO SHOOT
SETUP
PR [HOW MUCH DO YOU WANT TO TURN RIGH→
T?]
RT FIRST RL
PR [HOW MUCH DO YOU WANT TO MOVE?]
FD FIRST RL
IF COND 19 [PR [YOU GOT IT!]][PR [BAD→
 LUCK]]
END
```

```
TO SETUP
TELL [0 1] CS ST
TELL 0 PU
RT RANDOM 360
FD RANDOM 80
SETH 0 PD
SETSH 1
TELL 1
END

SHOOT
HOW MUCH DO YOU WANT TO TURN RIGHT?
45
HOW MUCH DO YOU WANT TO MOVE?
50
BAD LUCK
```



IF — command or operation

IF *pred instructionlist*
IF *pred instructionlist1 instructionlist2*

The first input, *pred*, is a predicate or condition that IF tests to be TRUE or FALSE. If *pred* is TRUE, *instructionlist1* is run. If *pred* is FALSE, *instructionlist2* is run. (Nothing is done if there is no *instructionlist2*.)

In either case, if the selected instructionlist outputs, then IF outputs the same thing. If the list does not output, neither does IF. Note that if you use IF with just one instructionlist, and follow it on the same line with another command, Logo will print an error message.

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
```

You will get the answer YES or NO with any definition.

```
PRINT DECIDE
YES
```

IF can be used inside of another IF clause. For example,

```
TO POSITIVE :NUM
IF NUMBERP :NUM [IF :NUM > 0 [PR [POS→
ITIVE NUMBER]] [PR [NEGATIVE NUMBER]]→
][PR [NOT A NUMBER]]
END
```

OUTPUT, OP command

OUTPUT *object*

This command can be used only within a procedure, not at top level. It makes *object* the output of this procedure and returns control to the caller. Note that OUTPUT is itself a command, but the procedure containing it is an operation because the procedure is made to output (compare with STOP).

EXAMPLES

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

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

ITEM outputs the *n*th element in the list:

```
TO ITEM :N :OBJ
IF EMPTYP :OBJ [OUTPUT "]
IF :N = 1 [OP FIRST :OBJ]
OP ITEM :N-1 BF :OBJ
END

MAKE "VOWELS [A E I O U]
PR ITEM 2 :VOWELS
E

PR ITEM 5 :VOWELS
U

PR ITEM 6 :VOWELS
```

The following procedure tells whether its first input is a subset of its second input. It outputs TRUE or FALSE. This is how you make your own *predicate*.

```
TO SUBSET :SUB :ALL
IF EMPTYP :SUB [OUTPUT "TRUE]
IF MEMBERP FIRST :SUB :ALL [OP SUBSET→
 BF :SUB :ALL] [OP "FALSE]
END

PRINT SUBSET [W E] [A E I O U]
FALSE

IF SUBSET [I E] [A E I O U] [PR "VOWEL→
S]
VOWELS
```

OVER — operation

OVER *turtlenumber pennumber*

Outputs number symbolizing a collision between *turtlenumber* and *pennumber*. (See table of collisions and events at the beginning of this chapter.) OVER can be used as an input to WHEN or COND.

EXAMPLE

```
PR OVER 1 0
4
```

REPEAT — command

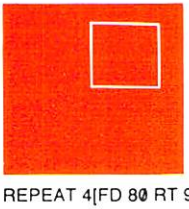
REPEAT *n instructionlist*

Runs a list of instructions the specified number of 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 80 RT 90]
```

draws a square 80 turtle steps on a side.



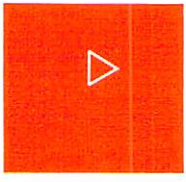
REPEAT 4[FD 80 RT 90]

```
REPEAT 5 [PRINT RANDOM 20]
```

Prints 5 random numbers from 0 to 19.

The following procedure draws polygons:

```
TO POLY :SIDE :ANGLE
REPEAT 360 / :ANGLE [FD :SIDE RT :ANG→
LE]
END

POLY 50 120
```



POLY 50 120

RUN — command or operation

RUN *instructionlist*

Runs the specified list of instructions as if it were typed in directly. If *instructionlist* is an operation, then RUN outputs whatever *instructionlist* outputs.

EXAMPLES

The following procedure simulates a calculator:

```
TO CALCULATOR
PRINT RUN RL
PRINT []
CALCULATOR
END

CALCULATOR
2 + 3
5

17.5 * 3
52.5

42 = 8 * 7
FALSE

REMAINDER 12 5
2
```

Press the **BREAK** key to stop.

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

```
TO WHILE :CONDITION :INSTRUCTIONLIST
IF NOT RUN :CONDITION [STOP]
RUN :INSTRUCTIONLIST
WHILE :CONDITION :INSTRUCTIONLIST
END

RT 90
WHILE [XCOR < 100] [FD 25 PR POS]
25 0
50 0
75 0
100 0
```

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

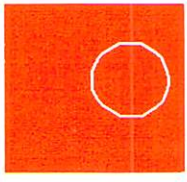
```
TO MAP :CMD :LIST
IF EMPTYP :LIST [STOP]
RUN LIST :CMD WORD "" FIRST :LIST
MAP :CMD BF :LIST
END

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

MAP "SQUARE [10 20 40 80]
```

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

The following procedure, FOREVER, repeats its input forever (unless it encounters an error or is stopped with the BREAK key):

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

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



FOREVER [FD1 RT1]

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

RUN RL runs any commands or operations typed in by the user.

PR RUN RL prints the output from any expression typed in by the user.

STOP command

STOP

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).

EXAMPLE

```
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!
```

TOUCHING — operation

TOUCHING *turtlenumber1 turtlenumber2*

Outputs the number symbolizing a collision between *turtlenumber1* and *turtlenumber2*. (See table at the beginning of the chapter.) TOUCHING can be used as an input to WHEN or COND.

EXAMPLE

```
PR TOUCHING 2 3
18
```

WAIT — command

WAIT *n*

Tells Logo to wait for *n* 60ths of a second.

EXAMPLE

The procedure SLOWFD makes the turtle go forward very slowly.

```
TO SLOWFD :DIST
REPEAT :DIST [FD 1 WAIT 1]
END

SLOWFD 80
CS
REPEAT 4 [SLOWFD 80 RT 90]
```

WHEN *condnumber instructionlist*
WHEN *condnumber* [ ]

Sets up a WHEN demon for detecting a collision or event *condnumber*. (See table at the beginning of the chapter.) *Condnumber* is an integer from 0 to 21 symbolizing an event. When this event occurs, *instructionlist* is run. If the *instructionlist* includes turtle commands, the current turtle(s) carries them out.

The WHEN command must be given while in splitscreen or fullscreen.

Note that WHEN's effect is global: this command needs to be given only once. See POD and PODS in Chapter 9 for checking which demons are in action.

WHEN *condnumber* [ ]

Since most *condnumbers* refer to a graphics command, it may be impossible to work in the textscreen mode while a WHEN demon is still alive. There are two ways you can clear a WHEN demon so that it no longer watches for an event or collision. The simplest way is to give the CS command. A side-effect is your design on the graphics screen is also cleared. The best method is to give the command WHEN *condnumber* [ ], since a demon is inactive if it has no task to perform. For example, if you want to clear WHEN demons 0 and 4, type

```
WHEN 0 [ ]
WHEN 4 [ ]
```

PODS allows you to check if these demons are still active,

```
PODS
```

There are no active demons.

Note: an error message or the EDIT command will automatically clear the active demons.

It is possible to give more than one WHEN command at one time, but the demons will not be active simultaneously. Their speed in detecting a collision or event depends on their strength. WHEN demon 0 is the strongest and therefore the fastest demon; WHEN demon 21 is the weakest and slowest. When one demon is busy (its event is occurring), the other demons go to sleep and don't wake up until the first demon has completed its task.

When setting up a game or project using demons, it is helpful to follow these guidelines:

1. Try to give a WHEN demon a task (*instructionlist*) that can be executed as fast as possible.
2. The best way to use WHEN demons is to give the instruction SETSP 0 and then use a helping procedure to examine each turtle's state.
3. The helping procedure shouldn't do anything except watch for a condition, and call a collision processing procedure if the collision occurred.
4. The collision processing procedure must always leave the turtle affected by a WHEN demon out of a collision situation. If not, the turtle could be caught on a line and eventually escape its bounds.

EXAMPLES

Whenever the joystick changes position (event number 15), JOYH is executed, allowing you to draw with the joystick.

```
TO JOYH
IF (JOY 0) < 0 [STOP]
SETH 45 * JOY 0
FD 5
JOYH
END

WHEN 15 [JOYH]
```

The following program sets the turtle in motion. When you press the button on your joystick (event number 3), the turtle acts like a spring.

```
TO PLAY
CS ST PD
REPEAT 4 [FD 100 RT 90]
PU SETPOS [50 50]
WHEN 3 [SPRING 100]
SETSP 10
END

TO SPRING :SPEED
IF :SPEED < 1 [STOP]
FD :SPEED WAIT 50 BACK :SPEED
SPRING :SPEED/2
END

PLAY
```

Here is a set of procedures that makes a square and keeps four turtles inside it.

```
TO SETUP
TELL [0 1 2 3] CS ST PU
SETPN 0 SETPC 0 120
ASK 0 [SETPOS [-50 -50] PD REPEAT 4 [→
FD 100 RT 90] PU]
ASK 0 [SETPOS [-20 -20]]
ASK 1 [SETPOS [-20 20]]
ASK 2 [SETPOS [20 -20]]
ASK 3 [SETPOS [20 20]]
EACH [RT 90 * WHO]
END

TO DEMONS.TASK
WHEN 0 [SETSP 0]
WHEN 4 [SETSP 0]
WHEN 8 [SETSP 0]
WHEN 12 [SETSP 0]
END
```

```
TO WATCH
IF SPEED = 0 [FIND.THEM]
WATCH
END

TO FIND.THEM
IF COND 0 [ASK 0 [BK 10 RT 180]]
IF COND 4 [ASK 1 [BK 10 RT 180]]
IF COND 8 [ASK 2 [BK 10 RT 180]]
IF COND 12 [ASK 3 [BK 10 RT 180]]
SETSP 30
END

SETUP
DEMONS.TASK
WATCH
```

SETUP sets up the square and the four turtles in it. DEMONS.TASK sets up the WHEN demons. WATCH is a helping procedure that calls the collision processing procedure, FIND.THEM.

Chapter 7

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 ATARI Logo have chosen to retain the traditional names AND, OR, and NOT for these logical operations. They are used to combine predicates into logical expressions. This is similar to the way in which arithmetic operations form arithmetic expressions. Just as arithmetic operations receive and output only numbers, so logical operations receive and output only TRUE or FALSE.

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

Predicate	Chapter
COND	6
EMPTYP	2
EQUALP	2
JOYB	8
KEYP	8
LISTP	2
MEMBERP	2
NAMEP	3
NUMBERP	2
PADDLEB	8
SHOWNP	1
WORDP	2
<	4
=	4
>	4

AND operation

AND *pred1 pred2*
(AND *pred1 pred2 pred3 . . .*)

Receives two or more inputs. AND outputs TRUE if all its inputs are true, FALSE otherwise.

EXAMPLES

```
PRINT AND "TRUE "TRUE
TRUE

PRINT AND "TRUE "FALSE
FALSE

PRINT AND "FALSE "FALSE
FALSE

PRINT (AND "TRUE "TRUE "FALSE "TRUE)
FALSE

PRINT AND 5 7
7 IS NOT TRUE OR FALSE

PRINT AND (PC 1) = 0 BG = 0
FALSE
```

(The infix operation = returns TRUE or FALSE to AND.)

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

```
TO DECIMALP :OBJ
OP AND NUMBERP :OBJ CHECK :OBJ
END

TO CHECK :OBJ
IF EMPTYP :OBJ [OP "FALSE]
IF EQUALP FIRST :OBJ ". [OP "TRUE]
OP CHECK BF :OBJ
END

PRINT DECIMALP 17
FALSE

PRINT DECIMALP 17.0
FALSE
```

Note that Logo interprets a number as an integer if it ends with a decimal point and a zero or just a decimal point.

```
PRINT DECIMALP 48.098
TRUE

PRINT DECIMALP "STOP.
FALSE
```

FALSE — special word

FALSE

FALSE is a special input for AND, IF, NOT and OR.

NOT — operation

NOT *pred*

Outputs TRUE if *pred* is FALSE; outputs FALSE if *pred* is TRUE.

EXAMPLES

```
PRINT NOT EQUALP "A "B
TRUE

PRINT NOT EQUALP "A "A
FALSE

PRINT NOT "A = FIRST "DOG
TRUE

PRINT NOT "A
A IS NOT TRUE OR FALSE
```

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

```
TO WORD? :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 :OB→
J
END

PRINT REALWORDP HEADING
FALSE

PRINT REALWORDP "KANGAROO
TRUE

PRINT REALWORDP PEN
TRUE
```

OR operation

OR *pred1 pred2*
(OR *pred1 pred2 pred3* ...)

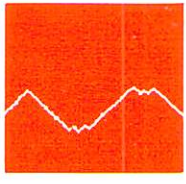
Outputs TRUE if any of its inputs are true, FALSE otherwise.

EXAMPLES

```
PRINT OR "TRUE "TRUE
TRUE

PRINT OR "TRUE "FALSE
TRUE

PRINT OR "FALSE "FALSE
FALSE

PRINT OR 5 7
7 IS NOT TRUE OR FALSE
```

The procedure **MOUNTAINS** draws "mountains":

```
TO MOUNTAINS
CS
RT 45
SUBMOUNTAIN
END

TO SUBMOUNTAIN
FD 5 + RANDOM 10
IF OR YCOR > 50 YCOR < 0 [SETH 180 - →
HEADING]
SUBMOUNTAIN
END
```



MOUNTAINS

TRUE special word

TRUE

TRUE is a special input for **AND**, **IF**, **NOT**, and **OR**.

Chapter 8

The Outside World

This chapter describes primitives for communicating with various devices through the computer. The devices include the keyboard, the TV screen and special purpose devices such as joysticks. If you are using a television or a monitor with volume control, you can also take advantage of the ATARI Logo music primitives, TOOT and SETENV.

The ATARI Computer has 24 lines of text on the screen, with 38 characters on each line. The screen can be used entirely for text or entirely for graphics. You can also split the screen using the top nineteen lines for graphics and the bottom five lines for text. When you start up Logo, the entire screen is available for text. The cursor on the text screen is similar to the turtle on the graphics screen. You can put characters anywhere on the text screen by setting the cursor at the desired place.

In addition to those primitives described in this section, the commands SAVE, LOAD, SETREAD, and SETWRITE are related to communication with the outside world. They are described in Chapter 10.

CT — command

CT

Stands for Clear Text. Clears the text from the screen and puts the cursor at the upper left corner of the text part of the screen.

FS — command

FS

Stands for Full Screen. Devotes the entire screen to graphics. Only the turtle graphics show; any text you type will be invisible to you, although Logo will still carry out your instructions. The text will reappear when you switch back to SS or TS mode.

If Logo needs to type an error message while you are in FS, it automatically goes back to SS.

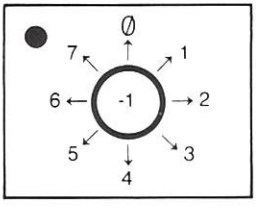
The CTRL F key combination has the same effect as FS. In addition, CTRL F can be pressed while a procedure is running, whereas you must wait to get the ? (prompt) in order to type FS.

Note that if you give the CT command when the screen has been switched to FS, the cursor goes to the top of the textscreen. When you switch the screen back to SS (splitscreen), the cursor will be hidden by the graphics portion of the screen.

JOY — operation

JOY *joysticknumber*

Outputs a number between −1 and 7 representing the position of the joystick.



The input must be Ø, 1, 2, or 3, that is, the number of the joystick being used. It is an error if you give any other input. If the joystick is in its initial position, (you have not moved it), JOY outputs −1. See Appendix C in the *Introduction Manual* for examples.

JOYB — operation

JOYB *joysticknumber*

Outputs TRUE if the button on the specified joystick is down, FALSE otherwise. The input must be Ø, 1, 2, or 3, since there are 4 joysticks. It is an error if you give any other input. If there are no joysticks connected, JOYB outputs FALSE. See Appendix C in the *Introduction Manual* for examples.

KEYP — operation

KEYP

Outputs TRUE if there is at least one character waiting to be read on the keyboard or any other device set by SETREAD, FALSE if there isn't any.

EXAMPLE

The following procedures keep the turtle going forward by small steps. Whenever you press R the turtle turns RT 10; when you press L the turtle turns LT 10.

```
TO STEER
FD 2
IF KEYP [TURN RC]
STEER
END

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

PADDLE — operation

PADDLE *paddlenumber*

Outputs a number between 0 and 247, representing the rotation of the dial on the specified paddle. *Paddlenumber* is an integer from 0 through 7. It is an error if you give any other input. If there is no paddle connected, PADDLE outputs -1.

EXAMPLE

The following procedure allows you to draw on the screen by rotating paddle 0 to change the turtle's heading, and paddle 1 to move the turtle forward.

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

PADDLEB operation

PADDLEB *paddlenumber*

Outputs TRUE if the button on the specified paddle is down, FALSE otherwise. *Paddlenumber* must be an integer from 0 through 7 since there are a maximum of eight paddles. It is an error if any other input is given. If there are no paddles connected, PADDLEB outputs FALSE.

EXAMPLE

The procedure DRIVE allows you to control the turtle's movement by the button. It will turn around in a circle while you hold down the button of paddle number 0, and will go in a straight line when you release it.

```
TO DRIVE
IF PADDLEB 0 [RIGHT 5]
FORWARD 2
DRIVE
END
```

PRINT, PR command

PRINT *object*
(PRINT *object1 object2* . . .)

Prints its input(s) on the screen, followed by 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
REPRINT :MESSAGE :HOWMANY - 1
END

REPRINT [TODAY IS FRIDAY!] 4
TODAY IS FRIDAY!
TODAY IS FRIDAY!
TODAY IS FRIDAY!
TODAY IS FRIDAY!
```

RC — operation

RC

Stands for Read Character. Outputs the first character read from a device or the keyboard. This character can even be a CTRL character. If no character is waiting to be read, RC waits until the user types something. This character is not echoed on the screen. If the end of file position has been reached in a file being read, RC outputs an empty word. See also KEYP.

EXAMPLE

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

```
TO DRIVE
MAKE "CHAR RC
IF :CHAR = "F [FD 5]
IF :CHAR = "R [RT 10]
IF :CHAR = "L [LT 10]
DRIVE
END
```

RL — operation

RL

Stands for Read List. Outputs as a list the first line of words read from the keyboard or a device. If no list is waiting to be read, RL waits for the user to type something. If lists have already been typed, it outputs the first line that has been typed but not read. Whatever you type will be echoed on the screen. If the end-of-file position has been reached in a file being read, RL outputs an empty list.

EXAMPLES

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

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

SETCURSOR — command

SETCURSOR *position*

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 0 to 23 character positions, columns from 0 to 37.

It is an error if the line number is not between 0 and 23, or if the column number is not between 0 and 37, or if an element of *position* is not an integer. Note that column 37 is reserved for the → (line continuation arrow).

EXAMPLE

```
SETCURSOR [35 12]
```

The cursor moves half-way down the right edge of the screen.

SETENV command

SETENV *voice duration*

SETENV is an envelope shaper which reduces the volume of the given *voice* (0 or 1) by 1 unit every *duration* (units of 1/60 second). The default duration is zero, which bypasses this modification, and consequently "sounds just like a computer".

EXAMPLE

```
TO TONE0 :DUR
TOOT 0 440 15 :DUR
END

TO TONE1 :FR :DUR
TOOT 1 :FR 15 :DUR
END

TO TIMEOUT
TONE0 120 TONE1 110 30 TONE1 220 30
TONE0 60 TONE1 330 30 TONE1 448 30
END

SETENV 0 6
SETENV 1 2
REPEAT 6 [TIMEOUT]
```

SHOW command

SHOW *object*

Prints *object* on the screen, followed by a carriage 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 "A TYPE [A B C]
AA B C

PRINT "A PRINT [A B C]
A
A B C
```

SS command

SS

Stands for Split Screen. Splits the screen into the turtle field and the text field. The first graphics command given after you start up Logo will automatically switch to SS: the top nineteen lines of the screen are available for graphics, and the bottom five lines are reserved for text.

The CTRL S key combination gives the same affect as the SS command. See also FS and TS.

Note: that if you give the CT command while the screen is in SS, the bottom five lines are cleared of text, but the top nineteen lines on the text screen remain unchanged.

TOOT command

TOOT *voice frequency volume duration*

Generates a tone via audio output specified by voice (Ø OR 1). *Frequency* is specified in Hertz (cycles per second) and can go from 14 to above audibility. (44Ø is the tuning note A.) *Volume* may range from Ø to 15. *Duration* may range from Ø to 255; it is measured in units of 1/6Ø second.

If a second TOOT to the same voice is attempted, Logo will wait until the first TOOT is finished.

EXAMPLE

```
TO SOUND.RANGE :FREQ
TOOT Ø :FREQ 15 15
PR :FREQ
SOUND.RANGE :FREQ + 5Ø
END

SOUND.RANGE 14
```

TS command

TS

Stands for Text Screen. Devotes the entire screen to text; the turtle field will be temporarily invisible to you until a graphics procedure is run. The CTRL T key combination is equivalent to TS. In addition, CTRL T can be used while a procedure is still running, whereas to type TS, you have to wait until you get the prompt. See also SS and FS.

TYPE command

TYPE *object*
(TYPE *object1 object2* . . .)

Prints its input(s) on the screen, not followed by a carriage return. The outermost brackets of list 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 "\                          Backslash followed by a space.
END

TO MOVE
PROMPT [HOW MANY STEPS SHOULD I TAKE?→
]
FD FIRST RL
MOVE
END

MOVE
HOW MANY STEPS SHOULD I TAKE? 50

HOW MANY STEPS SHOULD I TAKE? 37

HOW MANY STEPS SHOULD I TAKE? 2

HOW MANY STEPS SHOULD I TAKE? 108
```

Chapter 9

Workspace Management

Your *workspace* comprises the variables and procedures 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 procedures from your workspace.

The workspace is a temporary space. Your procedures and variables will be erased when you turn off the power of the computer. If you want to keep them for future use, you must store them on a diskette or cassette in the form of files. See Chapter 10 for information on files.

Note that any command starting with ER clears the edit buffer. If after giving such a command you give the EDIT command with no input, the editor will not contain any procedures.

ERALL command

ERALL

Stands for ERase ALL. Erases all procedures and variables from the workspace. This command also frees up all nodes of the system. Make sure that all the procedures you want to keep are saved in a file before you use this command.

ERASE command

ERASE *name*
ERASE *namelist*

Erases the named procedure(s) from the workspace. This command does not affect the procedure(s) saved in a file.

EXAMPLES

ERASE "TRIANGLE
erases the TRIANGLE procedure.

ERASE [TRIANGLE SQUARE]
erases the TRIANGLE and SQUARE procedures.

ERN — command

ERN *name*
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 — command

ERNS

Stands for ERase NameS. Erases all variables from the workspace.

ERPS — command

ERPS

Stands for ERase ProcedureS. Erases all procedures from the workspace.

NODES — operation

NODES

Outputs the number of free nodes. This gives you an idea of how much space you have in your workspace for procedures, variables, and running procedures. If you want to find out exactly how many nodes you have left, run NODES immediately after RECYCLE.

PO — command

PO *name*
PO *namelist*

Stands for Print Out. Prints the definitions of the named procedure(s). You cannot print out any Logo primitives.

EXAMPLES

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

PO [POLY GREET]
TO POLY :SIDE :ANGLE
FD :SIDE
RT :ANGLE
POLY :SIDE :ANGLE
END

TO GREET
PRINT [HI THERE]
END
```

POALL — command

POALL

Stands for Print Out ALL. Prints the definition of every procedure and the value of every variable in the workspace.

EXAMPLES

```
POALL
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

MAKE "ANIMAL "AARDVARK
MAKE "LENGTH 3.98
MAKE "MYNAME "PAT
```

POD command

POD *condnumber*

Stands for Print Out Demon. Prints out the condition and action set up for WHEN demon *condnumber*. *Condnumber* stands for collision number or event number (see table at the beginning of Chapter 6). See WHEN for setting up a WHEN demon.

EXAMPLES

```
POD 0
```

There is no WHEN demon 0 set up.

```
WHEN 0 [BK 10]
POD 0
WHEN 0 [BK 10]
```

PODS command

PODS

Stands for Print Out DemonS. Prints out the conditions and actions set up for all the WHEN demons.

EXAMPLES

```
PODS
WHEN 0 [BK 10]
WHEN 3 [SETSP 0]
```

PONS command

PONS

Stands for Print Out NameS. Prints the name and value of every variable in the workspace.

EXAMPLE

```
PONS
MAKE "ANIMAL "AARDVARK
MAKE "LENGTH 3.98
MAKE "NAMES [LINDA MIKE]
```

POPS command

POPS

Stands for Print Out ProcedureS. Prints the definition of every procedure in the workspace.

EXAMPLE

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

TO GREET
PRINT [HI THERE]
END

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

POTS command

POTS

Stands for Print Out TitleS. Prints the title line of every procedure in the workspace.

EXAMPLE

```
POTS
TO POLY :SIDE :ANGLE
TO GREET
TO SPI :SIDE :ANGLE :INC
```

RECYCLE command

RECYCLE

Performs a garbage collection, freeing 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.

Chapter 10

Files

The procedures and variables you created in the workspace will be erased when you turn off the power of the computer. If you want to keep them for future use, you can store them on a diskette or cassette. The information is organized in *files*. You decide what should go into each file.

You can create a file containing a copy of all characters displayed on the textscreen. This "dribble" file created by the command SETWRITE gives a record of the interactions between the person at the keyboard and the computer. You can read any file line by line with the command SETREAD.

A printer is considered as a special kind of a file. For example, you can list the contents of the procedures and names in your workspace by saving them on a printer.

The input for a file command always specifies the device being used:

C: stands for cassette

D: stands for disk drive 1

D*n*: stands for disk drive *n* (*n* is a disk drive number from 1 through 4)

P: stands for printer

When D:, D1: or D*n*: (disk drive) is the input, a file name must also be specified. If you use a file name with any other device, this input will be ignored. The only exception is CATALOG where D:, D1:, or D*n*: is used alone.

A filename can be 1 to 8 characters long with an optional 3 character extension. The first character of the filename must be a letter. All letters in the filename and extension must be uppercase. If an extension is used, a period must be used to separate the filename from the extension.

CATALOG command

CATALOG *device:*

Prints on the screen the names of all the files on the disk, if *device:* is a disk drive. If *device:* is a cassette ("C:), all the procedure definitions and names will be displayed.

EXAMPLES

CATALOG "D:
lists the files on disk in the current drive.

CATALOG "D2:
lists the files on disk in drive 2.

CATALOG "C:
lists all the procedure definitions and names in the cassette file.

ERF command

ERF *device:filename*

Erases the file named *filename* from the diskette. It is an error if there is no file by the name you have specified.

The only *device* that can be used with ERF is a disk drive. If you have more than one drive, the drive number must be specified.

EXAMPLE

ERF "D:BEAR
erases the file called BEAR from your disk.

LOAD command

LOAD *device:filename*

Loads the contents of *filename* into the workspace, as if typed in directly. It is an error if *filename* doesn't exist or if you try to LOAD from the printer. The BREAK key interrupts LOAD.

After the file is loaded, you can verify the content using various print out commands. (See Chapter 9 — Workspace Management.) For specific information on using a cassette to LOAD or SAVE, see Chapter 5 in the *Introduction Manual*.

EXAMPLES

```
ERALL
```

Your workspace is now empty.

```
LOAD "D1:BEAR
EYES DEFINED
PLAY DEFINED
JOYH DEFINED

LOAD "C:
EYES DEFINED
PLAY DEFINED
JOYH DEFINED
```

SAVE command

SAVE *device:filename*

Creates a file named *filename* ar saves in it all procedures and variables in the workspace.

Never use the BREAK key when a file is being saved: you will lose your workspace.

It is good practice to check before SAVE what you are saving and erase names of the procedures you don't need. See POTS, PO, POALL and ERASE in Chapter 9.

EXAMPLES

SAVE "D:MARIO.001
saves the contents of the workspace into the file called MARIO.001 on a disk.

SAVE "C:
saves the contents of the workspace onto cassette. (See Chapter 5 in the *Introduction Manual* for details on saving files on a cassette.)

SAVE "P:
prints the contents of the workspace on a printer.

SETREAD command

SETREAD *device:filename*
SETREAD [ ]

Sets the *device:* from which to receive input. *Filename* can be a program file or a file created by SETWRITE. After the command SETREAD is given, RC and RL read information from this *device:filename*.

SETREAD [ ]
SETREAD [ ] closes the file being read.

You can only SETREAD to one file at a time but you can open a file for reading (SETREAD) and writing (SETWRITE) at the same time.

EXAMPLES

```
SETREAD "D:BEAR
REPEAT 4 [PR RL]
TO EYES
REYE
LEYE
END
```

The first lines of the BEAR file are printed.

```
SETREAD [ ]
```

The file is closed. Now RL and RC will be read from the keyboard.

See SETWRITE for more examples.

SETWRITE — command

SETWRITE *device:filename*
SETWRITE [ ]

Opens file named *device:filename* and starts the process of sending a copy of all the characters displayed on the textscreen to *device:filename*.

SETWRITE [ ]
SETWRITE [ ] closes the file.

You can only SETWRITE to one file at a time but you can SETREAD and SETWRITE at the same time. It is an error if you SETWRITE to a device that is not connected or turned on after booting Logo. To read a file created with SETWRITE, use the command SETREAD.

EXAMPLES

```
SETWRITE "D:SEPT1
```

opens a file called SEPT1 on diskette. Now everything appearing on the textscreen will be sent to the SEPT1 file.

```
FD 40
RT 90
SETWRITE "D:SEPT2
```

The SEPT1 file is automatically closed and the SEPT2 file is opened.

```
FD 30
RT 45
SETWRITE [ ]
```

The SEPT2 file is closed.

```
SETREAD "D:SEPT1
```

opens the SEPT1 file for reading.

```
REPEAT 4 [PR RL]
FD 40
RT 90
SETWRITE "D:SEPT2
```

Everything in the SEPT1 file is printed on the screen.

```
SETREAD "D:SEPT2
REPEAT 4 [PR RL]
FD 30
RT 45
SETWRITE [ ]

SETREAD [ ]
```

Everything in the SEPT2 file is printed on the screen.

Chapter 11

Special Primitives

There are some special primitives that may affect the Logo system itself. They give you the power of directly accessing the computer memory or modifying what's in it. At the same time they are dangerous primitives because you can destroy the contents of your workspace in Logo by using them carelessly. If that happens, you will need to restart Logo. The names of these primitives start with a dot to warn you that they are dangerous. You should save your work before experimenting with them. For further information see *ATARI's Technical Reference Notes*.

.CALL command

.CALL *n*

Tranfers control to the indicated machine language subroutine starting at address *n* (decimal).

.DEPOSIT command

.DEPOSIT *n byte*

Writes byte into machine address *n* (decimal).

EXAMPLES

The following procedures change the size of the turtle.

```
TO BIG
.DEPOSIT 53256 1
END
TO SMALL
.DEPOSIT 53256 0
END

TO BIGGER
.DEPOSIT 53256 3
END
```

.EXAMINE — operation

.EXAMINE *n*

Outputs the contents of machine address *n* (decimal).

.PRIMITIVES — command

.PRIMITIVES

Prints a list of all the Logo primitives.

.SETSCR — command

.SETSCR *n*

Sets the aspect ratio (the ratio of the size of a vertical turtle step to the size of a horizontal one) to *n* (– 2 through 2). The screen is cleared.

.SETSCR .5 makes each vertical turtle step half the length of a horizontal one.

.SETSCR is intended to be used when "squares" turn out looking like rectangles on some particular screens. (An aspect ratio of .8 is correct for most screens.)

PAL systems will be set for .SETSCR 1.

Appendix A

Error Messages

OUT OF SPACE

Your workspace is almost completely filled. It's best to erase some procedures and names from your workspace.

YOU DON'T SAY WHAT TO DO WITH *OBJECT*

A Logo object was given without preceding it by a command.

TOO MUCH INSIDE ( )'S

Parentheses were incorrectly placed in a Logo instruction. For example, parentheses surround more than one Logo expression.

NOT ENOUGH INPUTS TO *PROCEDURE*

A procedure or primitive is being run that requires more inputs.

UNEXPECTED ')'

A closing parenthesis has no corresponding opening parenthesis. A closing parenthesis was found when an input was expected.

I DON'T KNOW HOW TO *PROCEDURE*

Logo has tried to execute *PROCEDURE* but can't find its definition.

PROCEDURE **DIDN'T OUTPUT TO** *PROCEDURE*

A procedure or primitive that requires an input was not given one and was followed on the same line by another procedure or primitive.

NUMBER TOO BIG

The result of an arithmetic operation is more than 1E98 (10^{98}) or less than 1E−98 (10^{-98}).

PRIMITIVE **DOESN'T LIKE** *OBJECT* **AS INPUT**

An incorrect input was given to a primitive.

WORD **HAS NO VALUE**

A variable was used that was not given a value.

PRIMITIVE **IS A PRIMITIVE**

A primitive name was given as an input to **TO** or **EDIT**.

PROCEDURE **IS ALREADY DEFINED**

The name given as an input to **TO** or **EDIT** has already been used as a procedure name.

OBJECT **IS NOT TRUE OR FALSE**

An input was given to **IF**, **AND**, **OR**, or **NOT** that was not a predicate (didn't output **TRUE** or **FALSE**).

FILE *NAME* **NOT FOUND**

The file name given as input to **LOAD** or **SETREAD** is nonexistent.

I CAN'T OPEN *DEVICE:FILENAME*

The input to **SAVE**, **LOAD**, **SETREAD** or **SETWRITE** is incorrect. For example, the device was not specified.

YOU'RE AT TOPLEVEL

The command **STOP** or **OUTPUT** was used outside of a procedure.

STOPPED!

The **BREAK** key was pressed, interrupting whatever was running.

Appendix B

Special Keys

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

***BREAK**	Aborts whatever Logo is doing. If editing, changes made in the edit buffer will be ignored.
***CTRL →**	Moves the *cursor* one position to the right.
***CTRL ←**	Moves the *cursor* one position to the left.
CTRL ↑	Moves the *cursor* up to the previous line.
CTRL ↓	Moves the *cursor* down to the next line.
***CTRL 1**	Makes Logo stop scrolling until CTRL 1 is typed again.
***CTRL A**	Moves the *cursor* to the beginning of the current line.
***CTRL CLEAR**	Deletes text from the *cursor* position to the end of the current line.
***CTRL DELETE BACK S**	Erases the character at the *cursor* position.
***CTRL E**	Moves the *cursor* to the end of the current line.
CTRL F	Devotes full screen to graphics.
CTRL INSERT	Opens a new line at the position of the *cursor*.
CTRL S	Split screen: top for graphics, bottom for text.
CTRL T	Devotes entire screen to text.
CTRL V	Scrolls screen to next page in editor.
CTRL W	Scrolls screen back to previous page in editor.

CTRL X	Moves the *cursor* to beginning of editor.
***CTRL Y**	Inserts the contents of the delete buffer.
CTRL Z	Moves the *cursor* to end of editor.
***DELETE BACK S**	Erases the character to the left of the *cursor*.
ESC	Completes editing and exits to top level.
***RETURN**	Completes the line and puts the *cursor* to the beginning of the next line.
***SHIFT DELETE BACK S**	Deletes text from the *cursor* position to the end of the current line.
SHIFT INSERT	Opens a new line at the position of the *cursor*.
\ (Backslash)	Tells Logo to interpret the character that follows it literally *as a character*, rather than keeping some special meaning it might have. You have to backslash [,], (,), +, −, *, /, =, <, >, and itself.

Other special keys are listed in Getting Started.

Appendix C

Useful Tools

The procedures collected here in alphabetical order are likely to be useful in constructing your own procedures. Examples of the use of some of these procedures appear in this manual (refer to the Index). The other procedures appear here for the first time.

ABS outputs the absolute value of its input.

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

CLEAR.DEMONS clears all the WHEN demons if given the input of 21. (The Collision Detection Chart can be found on pg. 104.)

```
TO CLEAR.DEMONS :DEMON
IF :DEMON < 0 [STOP]
WHEN :DEMON [ ]
CLEAR.DEMONS :DEMON - 1
END
```

COPYDEF copies the definition of an "old" procedure name onto a "new" procedure name. COPYDEF "SQ "SQUARE would copy the definition of SQUARE onto the name SQ. Note that COPYDEF uses DEFINE and TEXT (page 169).

```
TO COPYDEF :NEW :OLD
MAKE "OLD TEXT :OLD
DEFINE :NEW BF BF FIRST :OLD BF :OLD
END
```

DEFINE makes a list the definition of the name you give as input.

```
TO DEFINE :NAME :INPUT :LIST
SETWRITE "D:PROG
PR ( SE "TO :NAME :INPUT)
PR.OUT :LIST
PR "END
SETWRITE [ ]
LOAD "D:PROG
ERF "D:PROG
END
```

```
TO PR.OUT :LIST
IF EMPTYP :LIST [STOP]
PR FIRST :LIST
PR.OUT BF :LIST
END
DEFINE "SQUARE ":SIZE [[REPEAT 4 [FD→
 :SIZE RT 90]]]
```

gives SQUARE this definition:

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

DIVISORP indicates whether its first input divides evenly into its second.

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

DOT places a dot on the screen at the position given as input. Note that the turtle is left in the same state as before DOT is run.

```
TO DOT :POS
ASK FIRST WHO [DOT1 POS :POS PEN SHOW→
NP]
END

TO DOT1 :OLDPOS :POS :PEN :SHOWNP
HT PU
SETPOS :POS
PD FD 0 PU
SETPOS :OLDPOS
RUN FPUT :PEN [ ]
IF :SHOWNP [ST]
END
```

FOREVER runs a list of instructions until the BREAK key is pressed or the power is turned off.

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

INIT.TURTLE clears the screen of all the turtles, just leaving turtle 0 in the regular turtle shape.

```
TO INIT.TURTLE
TELL [0 1 2 3] CS
SETSH 0 HT
TELL 0 ST
END
```

ITEM outputs the :Nth element of a word or a list.

```
TO ITEM :N :OBJECT
IF EMPTYP :OBJECT [OP "]
IF :N = 1 [OP FIRST :OBJECT]
OP ITEM :N-1 BF :OBJECT
END
```

SORT takes a list of words and outputs them alphabetically. **SUPERSORT** arranges them in a flat list.

```
TO SORT :ARG :LIST
IF EMPTYP :ARG [OP :LIST]
MAKE "LIST INSERT FIRST :ARG :LIST
OP SORT BF :ARG :LIST
END

TO INSERT :A :L
IF EMPTYP :L [OP FPUT [] LIST :A []]
IF BEFORE :A FIRST BF :L [OP FPUT INS→
ERT :A FIRST :L BF :L]
OP LPUT INSERT :A LAST :L BL :L
END

TO BEFORE :A :B
IF OR EMPTYP :A EMPTYP :B [OP EMPTYP →
:A]
IF NOT EQUALP FIRST :A FIRST :B [OP (→
 ASCII :A ) < ( ASCII :B )]
OP BEFORE BF :A BF :B
END

TO SUPERSORT :L
IF EMPTYP :L [OP []]
OP ( SE SUPERSORT FIRST :L FIRST BF :→
L SUPERSORT LAST :L )
END
```

Try this:

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

PR SUPERSORT :SORTLIST
A C D E F T Z
```

Then type

```
MAKE "SORTLIST SORT [FOO BAR BAZ] :S→
ORTLIST
PR SUPERSORT :SORTLIST
A BAR BAZ C D E F FOO T Z
```

TEXT outputs the definition of a procedure name. TEXT "SQUARE could output [[TO SQUARE :SIZE] [REPEAT 4 [FD :SIZE RT 90]]]

```
TO TEXT :NAME
SETWRITE "D:PROG
PO :NAME
SETWRITE [ ]
SETREAD "D:PROG
OP READLINE LIST RL ''
END

TO READLINE :TX
MAKE "LINE RL
IF [END] = :LINE [ERF "D:PROG OP :TX]
OP READLINE LPUT :LINE :TX
END
```

WHICH outputs which position an element has in its list. WHICH "C [A B C] outputs 3. Complement to the procedure ITEM.

```
TO WHICH :MEMBER :LIST
IF EMPTYP :LIST [OP 0]
IF :MEMBER = FIRST :LIST [OUTPUT 1]
OUTPUT 1 + WHICH :MEMBER BF :LIST
END
```

WHILE repeats a group of instructions until :CONDITION becomes FALSE.

```
TO WHILE :CONDITION :INSTRUCTIONLIST
IF NOT RUN :CONDITION [STOP]
RUN :INSTRUCTIONLIST
WHILE :CONDITION :INSTRUCTIONLIST
END
```

Appendix D

Memory 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 ATARI Logo 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 statements

```
MAKE "NUMBER 7
MAKE "NUMBER 90
```

after you say MAKE "NUMBER 7, NUMBER is assigned to two nodes that hold the value 7. After executing MAKE "NUMBER 90, the nodes containing the 7 can be reused, and they will be reclaimed as free nodes the next time the garbage collector runs. The garbage collector runs automatically when necessary, but you can make it run with the Logo command RECYCLE.

The operation NODES outputs the number of free nodes; however, if you really want to find out how much space you have, you should do something like the following:

```
RECYCLE PRINT NODES
1259
```

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.

A number, whether integer or decimal, takes up two nodes (exponent and mantissa). A list takes up one node for each element (plus the size of the element itself).

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 inputs of procedures can be the same as names of inputs of other procedures. The names of procedures and primitives can also be used as variable names.

4. It should be noted that misspellings, typing errors, and words that are no longer being used are not destroyed.

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

KISS
I DON'T KNOW HOW TO KISS
```

The words PRIMT, FOO, and KISS will be created and will not go away. However, if a word has no value 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 to a file and then read it into a freshly started Logo.

Appendix E

Parsing

When you type a line in 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. This appendix will help you understand how lines are parsed. To see the parsing effect, type the line in a procedure definition with the command TO and use the Logo editor to see the result.

Delimiters

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, to find out how this line is parsed:

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

Type

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

ED "TEST
```

The screen will look like this:

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

To treat any of the characters mentioned above as a normal alphabetic character, put a backslash "\" before it. For example:

```
PRINT "SAN\ FRANCISCO
SAN FRANCISCO
```

Infix Procedures

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

Brackets and Parentheses

Left bracket "[" and *right bracket* "]" indicate the start and end of a list or sublist.

Parentheses ( ) group things in ways Logo ordinarily would not, and vary the number of inputs for certain primitives.

If the end of a Logo line is reached (that is, the RETURN key is pressed) and brackets or parentheses are still open, all sublists or expressions 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, Logo stops execution of the rest of the line or procedure. For example:

```
]PRINT "ABC
```

Logo prints an empty line.

Quotes and Delimiters

Normally, you have to put a backslash before the characters [,], (,), +, −, *, /, =, <, >, and \ itself. But the first character after a quote (") does not need to have a backslash preceding it. For example:

```
PRINT "*
*
```

If a delimiter is occupying any position but the first after the quote, it must have a backslash preceding it. For example:

```
PRINT "****
NOT ENOUGH INPUTS TO *
```

The only exception to the above general rule is [ ] (brackets). You must always precede a bracket that is being quoted by the backslash.

```
PRINT "[
YOU DON'T SAY WHAT TO DO WITH [ ]

PRINT "\[
[
```

The Minus Sign

The way in which the minus sign " – " is parsed is an unusual case. The problem here is that one character is used to represent three different things:

1. Part of a number to indicate that it is negative, as in –3.
2. A procedure of one input, called unary minus, which outputs the additive inverse of its input, as in –XCOR or –:DISTANCE.
3. 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 (including a space) except right parenthesis ")", the number is parsed as a negative number. This allows the following behavior:

```
PRINT 3 * -1      (parses as 3 times negative 1)
PRINT 3*-4        (parses as 3 times negative 4)
FIRST [- 3 4]     (outputs -1)
FIRST [-3 4]      (outputs -3)
```

2. If " – " is preceded by a numeric expression, it works like an infix " – ".

```
PR 3-4 IS -1
PR XCOR - YCOR
```

3. If " – " is not preceded by a numeric expression, it works like a unary minus.

```
PR -XCOR
PR -(3+4)
```

Appendix F

ASCII Code

DECIMAL CODE	CODE CHARACTER	DECIMAL CODE	CODE CHARACTER
0		15	
1		16	
2		17	
3		18	
4		19	
5		20	
6		21	
7		22	
8		23	
9		24	
10		25	
11		26	
12		27	
13		28	
14		29	

DECIMAL CODE	CODE CHARACTER	DECIMAL CODE	CODE CHARACTER
30	←	45	-
31	→	46	.
32	Space	47	/
33	!	48	0
34	"	49	1
35	#	50	2
36	$	51	3
37	%	52	4
38	&	53	5
39	'	54	6
40	(	55	7
41	)	56	8
42	*	57	9
43	+	58	:
44	,	59	;

DECIMAL CODE	CODE CHARACTER	DECIMAL CODE	CODE CHARACTER
60	<	75	K
61	=	76	L
62	>	77	M
63	?	78	N
64	@	79	O
65	A	80	P
66	B	81	Q
67	C	82	R
68	D	83	S
69	E	84	T
70	F	85	U
71	G	86	V
72	H	87	W
73	I	88	X
74	J	89	Y

DECIMAL CODE	CODE CHARACTER	DECIMAL CODE	CODE CHARACTER
90	Z	105	i
91	[	106	j
92	\	107	k
93	]	108	l
94	^	109	m
95	—	110	n
96	✤	111	o
97	a	112	p
98	b	113	q
99	c	114	r
100	d	115	s
101	e	116	t
102	f	117	u
103	g	118	v
104	h	119	w

DECIMAL CODE	CODE CHARACTER	DECIMAL CODE	CODE CHARACTER
120		135	
121		136	
122		137	
123		138	
124		139	
125		140	
126		141	
127		142	
128		143	
129		144	
130		145	
131		146	
132		147	
133		148	
134		149	

DECIMAL CODE	CODE CHARACTER	DECIMAL CODE	CODE CHARACTER
150		165	%
151		166	&
152		167	'
153		168	(
154		169	)
155	(EOL) RETURN	170	*
156		171	+
157		172	,
158		173	—
159		174	.
160	Space	175	/
161	!	176	0
162	"	177	1
163	#	178	2
164	$	179	3

DECIMAL CODE	CODE CHARACTER	DECIMAL CODE	CODE CHARACTER
180	4	195	C
181	5	196	D
182	6	197	E
183	7	198	F
184	8	199	G
185	9	200	H
186	:	201	I
187	;	202	J
188	<	203	K
189	=	204	L
190	>	205	M
191	?	206	N
192	@	207	O
193	A	208	P
194	B	209	Q

DECIMAL CODE	CODE CHARACTER	DECIMAL CODE	CODE CHARACTER
210	R	225	a
211	S	226	b
212	T	227	c
213	U	228	d
214	V	229	e
215	W	230	f
216	X	231	g
217	Y	232	h
218	Z	233	i
219	[	234	j
220	\	235	k
221	]	236	l
222	^	237	m
223	—	238	n
224		239	o

DECIMAL CODE	CODE CHARACTER	DECIMAL CODE	CODE CHARACTER
240	p	248	x
241	q	249	y
242	r	250	z
243	s	251	♠
244	t	252	|
245	u	253	↖
246	v	254	◀
247	w	255	▶

ASCII* CODE

The white characters in black squares represent normal video characters. Black characters in white squares represent reverse video characters.

*For the special characters found on the ATARI Computer, an extended version of ASCII code is used.

Appendix G

Logo Vocabulary

Turtle Graphics

ASK
BACK, BK
BG
CLEAN
COLOR
CS
EACH
EDSH
FORWARD, FD
GETSH
HEADING
HOME
HT
LEFT, LT
PC
PE
PEN
PENDOWN, PD
PENUP, PU
PN
POS
PUTSH
PX
RIGHT, RT
SETBG
SETC
SETH
SETPC
SETPN
SETPOS
SETSH
SETSP
SETX
SETY
SHAPE
SHOWNP
SPEED
ST
TELL
WHO
WINDOW
WRAP
XCOR
YCOR

Words and Lists

ASCII
BUTFIRST, BF
BUTLAST, BL
CHAR
COUNT
EMPTYP
EQUALP
FIRST
FPUT
LAST
LIST
LISTP
LPUT
MEMBERP
NUMBERP
SE
WORD
WORDP
obj1 = obj2

Variables

MAKE
NAMEP
THING

Arithmetic Operations

COS
INT
PRODUCT
RANDOM
REMAINDER
RERANDOM
ROUND
SIN
SQRT
SUM
a + *b*
a - *b*
a * *b*
a / *b*
a < *b*
a = *b*
a > *b*

Defining and Editing Procedures

EDIT, ED
EDNS
END
TO

Flow of Control and Conditionals

COND
IF
OUTPUT, OP
OVER
REPEAT
RUN
STOP

TOUCHING
WAIT
WHEN
WHEN [ ]

Logical Operations

AND
FALSE
NOT
OR
TRUE

The Outside World

CT
FS
JOY
JOYB
KEYP
PADDLE
PADDLEB
PRINT, PR
RC
RL
SETCURSOR
SETENV
SHOW
SS
TOOT
TS
TYPE

Workspace Management

ERALL
ERASE, ER
ERN
ERNS
ERPS
NODES
PO
POALL
POD
PODS
PONS
POPS
POTS
RECYCLE

Files

CATALOG
ERF
LOAD
SAVE
SETREAD
SETREAD [ ]
SETWRITE
SETWRITE [ ]

Special Primitives

.CALL
.DEPOSIT
.EXAMINE
.PRIMITIVES
.SETSCR

Special Keys

BREAK
CTRL →
CTRL ←
CTRL ↑
CTRL ↓
CTRL 1
CTRL A
CTRL CLEAR
CTRL DELETE BACK S
CTRL E
CTRL F
CTRL INSERT
CTRL S
CTRL T
CTRL V
CTRL W
CTRL X
CTRL Y
CTRL Z
DELETE BACK S
ESC
RETURN
SHIFT DELETE BACK S
SHIFT INSERT
\ (Backslash)

Note: See glossary for definitions and required inputs.

Appendix H

Glossary

Note: 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. An asterisk (*) indicates an editing command which works inside and outside of the editor. For definitions of Input Words see page 21.

#**AND** *pred1 pred2*	Outputs TRUE if all its inputs are TRUE.
ASCII *char*	Outputs ASCII code for *char*.
ASK *turtlenumber list*	Asks the *turtlenumber(s)* to run the instructions in *list*.
BACK, BK *distance*	Moves turtle *distance* steps back.
BG	Outputs number representing background color.
BUTFIRST, BF *obj*	Outputs all but first element of *obj*.
BUTLAST, BL *obj*	Outputs all but last element of *obj*.
.CALL *n*	Transfers control to a machine language subroutine starting at address *n* (decimal).
CATALOG *device:*	Displays names of all files on diskette. On cassette, prints definitions of procedures and names in the file.
CHAR *n*	Outputs character whose ASCII code is *n*.
CLEAN	Erases graphics screen without affecting turtle's state.

COLOR	Outputs number representing the turtle color.
COND *condnumber*	Outputs TRUE if condition *condnumber* is occurring.
COS *n*	Outputs cosine of *n* degrees.
COUNT *obj*	Outputs the number of elements in *obj*.
CS	Erases screen, moves turtle to the position [0 0]. Sets heading to 0.
CT	Clears text screen.
.DEPOSIT *n byte*	Writes *byte* into address *n* (decimal).
EACH *list*	Makes each turtle separately run the commands in *list*.
EDIT, ED *name(s)*	Starts Logo editor with named procedure(s).
EDNS	Starts Logo editor with all variables in the workspace.
EDSH *shapenumber*	Starts the Logo shape editor, displaying the shape *shapenumber*.
EMPTYP *obj*	Outputs TRUE if *obj* is empty.
END	Ends the procedure definition started out by TO.
EQUALP *obj1 obj2*	Outputs TRUE if its inputs are equal.
ERALL	Erases everything from the workspace.
ERASE, ER *name(s)*	Erases all named procedure(s).

ERF *device:filename*	Erases *filename* from *device*.
ERN *name(s)*	Erases all named variables.
ERNS	Erases variables from the workspace.
ERPS	Erases all procedures from the workspace.
.EXAMINE *n*	Outputs contents of address *n* (decimal).
FALSE	Special input for AND, IF, NOT and OR.
FIRST *obj*	Outputs first element of *obj*.
FORWARD, FD *distance*	Moves turtle *distance* steps forward.
FPUT *obj list*	Outputs list formed by putting *obj* on front of *list*.
FS (CTRL F)	Devotes entire screen to graphics.
GETSH *shapenumber*	Returns a list of 16 numbers; these numbers correspond to bits in the shape.
HEADING	Outputs turtle's heading.
HOME	Moves turtle to [0 0] and sets heading to 0.
HT	Makes turtle invisible.
IF *pred list1 (list2)*	If *pred* is TRUE, runs *list1*, otherwise *list2*.
INT *n*	Outputs the integer portion of *n*.

JOY *joysticknumber*	Outputs current position of *joysticknumber*.
JOYB *joysticknumber*	Outputs TRUE if the button on *joysticknumber* is pressed.
KEYP	Outputs TRUE if a key has been typed but not yet read.
LAST *obj*	Outputs last element of *obj*.
LEFT, LT *degrees*	Turns turtle *degrees* left (counter-clockwise).
LIST *obj1 obj2*	Outputs list of its inputs.
LISTP *obj*	Outputs TRUE if *obj* is a list.
LOAD *device:filename*	Loads file called *filename* from *device* into the computer.
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 included in *list*.
NAMEP *name*	Outputs TRUE if *name* has a value.
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 of its inputs are TRUE.
OUTPUT, OP *obj*	Returns control to caller, with *obj* as output.

OVER *turtlenumber pennumber*	Outputs number symbolizing collision between *turtlenumber* and *pennumber*.
PADDLE *paddlenumber*	Outputs rotation on dial of *paddlenumber*.
PADDLEB *paddlenumber*	Outputs TRUE if the button is pressed on *paddlenumber*.
PC *pennumber*	Outputs number representing pen color of *pennumber*.
PE	Puts pen eraser down.
PEN	Outputs pen state (PD, PU, PE or PX).
PENDOWN, PD	Puts turtle's pen down.
PENUP, PU	Raises turtle's pen.
PN	Outputs the pen number (0, 1 or 2) being used.
PO *name(s)*	Prints definitions of named procedures.
POALL	Prints definitions of procedures and names (variables).
POD *condnumber*	Prints WHEN demon *condnumber* currently in action.
PODS	Print out all active WHEN demons.
PONS	Prints names and values of all variables.
POPS	Prints definitions of all procedures.

POS	Outputs coordinates of turtle's position.
POTS	Prints title lines of procedures.
.PRIMITIVES	Prints the list of Logo primitives.
# **PRINT, PR** *obj*	Prints *obj* followed by carriage return (strips off outer brackets of lists).
# **PRODUCT** *a b*	Outputs product of its inputs.
PUTSH *shapenumber shapespec*	Gives *shapenumber* the form of *shapespec*, the grid of bits.
PX	Puts reversing pen down.
RANDOM *n*	Outputs random integer between 0 and $n - 1$.
RC	Outputs character read by the current device (default is keyboard). Waits if necessary.
RECYCLE	Performs a garbage collection.
REMAINDER *a b*	Outputs remainder of *a* divided by *b*.
REPEAT *n list*	Runs *list n* times.
RERANDOM	Makes RANDOM behave reproducibly.
RIGHT, RT *degrees*	Turns turtle *degrees* right (clockwise).
RL	Outputs line read by current device (default is keyboard). Waits if necessary.

ROUND *n*	Outputs *n* rounded off to nearest integer.
RUN *list*	Runs *list*; outputs what *list* outputs.
SAVE *device:filename*	Saves workspace onto the *device*.
#**SE** *obj1 obj2*	Outputs list of its inputs.
SETBG *colornumber*	Sets background to *colornumber*.
SETC *colornumber*	Sets the turtle's *colornumber*.
SETCURSOR *pos*	Puts cursor at *pos*.
SETENV *voice duration*	Sets envelope of *voice* for TOOT so volume reduces by one unit every *duration*.
SETH *degrees*	Sets turtle's heading to *degrees*.
SETPC *pennumber colornumber*	Sets *pennumber* (0, 1 or 2) to *colornumber*.
SETPN *pennumber*	Sets the pen to *pennumber* (0, 1 or 2).
SETPOS *position*	Moves turtle to *position*.
SETREAD *device:filename*	Sets the *device:filename* from which the output of RC and RL will be read.
SETREAD [ ]	Closes the file that was opened with SETREAD.
.SETSCR *n*	Sets aspect ratio to *n*.
SETSH *shapenumber*	Sets shape of turtle to *shapenumber*.

SETSP *speed*	Sets the turtle's *speed*.
SETWRITE *device:filename*	Starts the process of sending a copy of all the characters displayed on the screen to *device:filename*.
SETWRITE [ ]	Closes the file that was opened with SETWRITE.
SETX *x*	Moves turtle horizontally to x-coordinate at *x*.
SETY *y*	Moves turtle vertically to y-coordinate at *y*.
SHAPE	Outputs number representing shape of the current turtle.
SHOW *obj*	Prints *obj* followed by RETURN with brackets for list.
SHOWNP	Outputs TRUE if turtle is shown.
SIN *n*	Outputs sine of *n* degrees.
SPEED	Outputs current turtle's speed.
SQRT *n*	Outputs square root of *n*.
SS (CTRL S)	Splits screen: top for graphics, bottom for text.
ST	Makes the turtle(s) visible.
STOP	Stops procedure and returns control to caller.
SUM *a b*	Outputs sum of its inputs.
TELL *turtlenumber(s)*	Addresses all following commands to *turtlenumber(s)*.
THING *name*	Outputs object referred to by *name*.

TO *name (inputs)*	Begins defining procedure *name.*
TOOT *voice freq volume duration*	Produces sound on *voice* of frequency *freq* and *volume* for *duration.*
TOUCHING *turtlenumber1 turtlenumber2*	Outputs number symbolizing collision between *turtlenumber1* and *turtlenumber2.*
TRUE	Special input for AND, IF, NOT and OR.
TS (CTRL T)	Devotes entire screen to text.
# **TYPE** *obj*	Prints *obj* leaving the *cursor* at the end of the printed line.
WAIT *n*	Pauses for *n* 60ths of a second.
WHEN *condnumber list*	Sets up WHEN demon so whenever condition *condnumber* occurs, *list* is run.
WHEN *condnumber* [ ]	Clears (stops) WHEN demon for *condnumber.*
WHO	Outputs number of current turtle.
WINDOW	Makes graphics screen a window of an expanded turtle field. Clears screen.
# **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. Clears screen.
XCOR	Outputs x-coordinate of turtle's position.
YCOR	Outputs y-coordinate of turtle's position.
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*.
a > *b*	Outputs TRUE if *a* is greater than *b*.
obj1 = *obj2*	Outputs TRUE if *obj1* is equal to *obj2*.

Special Keys

ATARI Key (人) **REVERSE VIDEO KEY (◪)**	After this key is pressed, all characters typed appear in reverse video on the screen.
***BREAK**	Aborts whatever Logo is doing. If editing, changes made in the edit buffer will be ignored.
***CTRL →**	Moves the *cursor* one position to the right.
***CTRL ←**	Moves the *cursor* one position to the left.

CTRL ↑	Moves the *cursor* up to the previous line.
CTRL ↓	Moves the *cursor* down to the next line.
***CTRL 1**	Makes Logo stop scrolling until CTRL 1 is typed again.
***CTRL A**	Moves the *cursor* to the beginning of the current line.
***CTRL CLEAR**	Deletes text from the *cursor* position to the end of the current line.
***CTRL DELETE BACK S**	Erases the character at the *cursor* position.
***CTRL E**	Moves the *cursor* to the end of the current line.
CTRL F	Devotes full screen to graphics.
CTRL INSERT	Opens a new line at the position of the *cursor*.
CTRL S	Split screen: top for graphics, bottom for text.
CTRL T	Devotes entire screen to text.
CTRL V	Scrolls screen to next page in editor.
CTRL W	Scrolls screen back to previous page in editor.
CTRL X	Moves the *cursor* to beginning of editor.
***CTRL Y**	In the editor, CTRL Y inserts the contents of the delete buffer. Outside the editor, inserts the last command line typed.

CTRL Z	Moves the *cursor* to end of editor.
***DELETE BACK S**	Erases the character to the left of the *cursor*.
ESC	Completes editing and exits to top level.
F1, F2, F3, F4	*Cursor* control keys that can be programmed.
***RETURN**	Completes the line and puts the *cursor* to the beginning of the next line.
***SHIFT DELETE BACK S**	Deletes text from the *cursor* position to the end of the current line.
SHIFT INSERT	Opens a new line at the position of the *cursor*.
SYSTEM RESET	Reboots Logo, erasing the memory space.
\ (Backslash)	Tells Logo to interpret the character that follows it literally *as a character*, rather than keeping some special meaning it might have. You have to backslash [,], (,), +, −, *, /, =, <, >, and itself.

Index

Index

Notes

Notes

Notes

Notes

Every effort has been made to ensure the accuracy of the product documentation in this manual. However, because we are constantly improving and updating our computer software and documentation, Logo Computer Systems, Inc. is unable to guarantee the accuracy of printed material after the date of publication and disclaims liability for changes, errors or omissions.

Logo Computer Systems, Inc.
9960 Cote de Liesse
Lachine, Quebec
Canada H8T 1A1

Printed in Canada
C061589