MiniLogo (based on BYOB)
User Manual

This miniLogo implementation of the Logo programming language has been modelled on Berkeley Logo (UCBlogo, http://www.cs.berkeley.edu/~bh/logo.html, by Brian Harvey). Only parts of UCBlogo have been implemented, namely

- data structure primitives
  - constructors
  - selectors
  - mutators
  - predicates
  - queries
- communication
  - transmitters
  - receivers
  - terminal access
- arithmetic
  - numeric operations
  - predicates
- logical operations
- graphics
  - turtle motion
  - turtle motion queries
  - turtle and window control
  - turtle and window queries
  - pen and background control
  - pen queries
  - mouse queries
- workspace management
  - procedure definition
  - variable definition
  - property lists
  - predicates
  - Control structures
  - template-based iteration
- special variables

Substantial parts of UCBlogo are (some forever) missing, namely

- file access
- terminal access
- print formatting
- bitwise operations
- saving and loading pictures
- queries
- inspection
- workspace control
- macros
- error processing

Still, the miniLogo programming language is widely usable.

What follows is an extract of the UCBlogo manual (http://www.cs.berkeley.edu/~bh/usermanual, thanks Brian!) in which the parts that have been implemented in miniLogo are highlighted in green. Note that shortcuts (given the "block" nature of miniLogo) have not been implemented.

DATA STRUCTURE PRIMITIVES

CONSTRUCTORS

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

outputs a word formed by concatenating its inputs.

LIST thing1 thing2
(LIST thing1 thing2 thing3 ...)

outputs a list whose members are its inputs, which can be any Logo datum (word, list, or array).

SENTENCE thing1 thing2
SE thing1 thing2
(SENTENCE thing1 thing2 thing3 ...)
(SE thing1 thing2 thing3 ...)

outputs a list whose members are its inputs, if those inputs are not lists, or the members of its inputs, if those inputs are lists.

FPUT thing list

outputs a list equal to its second input with one extra member, the first input, at the beginning. If the second input is a word, then the first input must be a one-letter word, and FPUT is equivalent to WORD.

LPUT thing list

outputs a list equal to its second input with one extra member, the first input, at the end. If the second input is a word, then the first input must be a one-letter word, and LPUT is equivalent to WORD with its inputs in the other order.

ARRAY size
(ARRAY size origin)

outputs an array of "size" members (must be a positive integer), each of which initially is an empty list. Array members can be selected with ITEM and changed with SETITEM. The first member of the array is member number 1 unless an "origin" input (must be an integer) is given, in which case the first member of the array has that number as its index. (Typically 0 is used as the origin if anything.) Arrays are printed by PRINT and friends, and can be typed in, inside curly braces; indicate an origin with {a b c}@0.

MDARRAY sizelist (library procedure)
(MDARRAY sizelist origin)

outputs a multi-dimensional array. The first input must be a list of one or more positive integers. The second input, if present, must be a single integer that applies to every dimension of the array. Ex: (MDARRAY [3 5] 0) outputs a two-dimensional array whose members range from [0 0] to [2 4].

LISTTOARRAY list
(LISTTOARRAY list origin)

outputs an array of the same size as the input list, whose members are the members of the input list.

ARRAYTOLIST array

outputs a list whose members are the members of the input array. The first member of the output is the first member of the array, regardless of the array's origin.

COMBINE thing1 thing2 (library procedure)

if thing2 is a word, outputs WORD thing1 thing2. If thing2 is a list, outputs FPUT thing1 thing2.

REVERSE list (library procedure)

outputs a list whose members are the members of the input list, in reverse order.

GENSYM (library procedure)

outputs a unique word each time it's invoked. The words are of the form G1, G2, etc.

SELECTORS

FIRST thing

if the input is a word, outputs the first character of the word. If the input is a list, outputs the first member of the list. If the input is an array, outputs the origin of the array (that is, the INDEX OF the first member of the array).

FIRSTS list

outputs a list containing the FIRST of each member of the input list. It is an error if any member of the input list is empty. (The input itself may be empty, in which case the output is also empty.) This could be written as

```
to firsts :list
output map "first :list
end
```

but is provided as a primitive in order to speed up the iteration tools MAP, MAP.SE, and FOREACH.

```
to transpose :matrix
if emptyp first :matrix [op []]
op fput firsts :matrix transpose bfs :matrix
end
```

LAST wordorlist

if the input is a word, outputs the last character of the word. If the input is a list, outputs the last member of the list.

BUTFIRST wordorlist
BF wordorlist

if the input is a word, outputs a word containing all but the first

character of the input. If the input is a list, outputs a list containing all but the first member of the input.

BUTFIRSTS list
BFS list

outputs a list containing the BUTFIRST of each member of the input list. It is an error if any member of the input list is empty or an array. (The input itself may be empty, in which case the output is also empty.) This could be written as

```
to butfirsts :list
output map "butfirst :list
end
```

but is provided as a primitive in order to speed up the iteration tools MAP, MAP.SE, and FOREACH.

BUTLAST wordorlist
BL wordorlist

if the input is a word, outputs a word containing all but the last character of the input. If the input is a list, outputs a list containing all but the last member of the input.

ITEM index thing

if the "thing" is a word, outputs the "index"th character of the word. If the "thing" is a list, outputs the "index"th member of the list. If the "thing" is an array, outputs the "index"th member of the array. "Index" starts at 1 for words and lists; the starting index of an array is specified when the array is created.

MDITEM indexlist array (library procedure)

outputs the member of the multidimensional "array" selected by the list of numbers "indexlist".

PICK list (library procedure)

outputs a randomly chosen member of the input list.

REMOVE thing list (library procedure)

outputs a copy of "list" with every member equal to "thing" removed.

REMDUP list (library procedure)

outputs a copy of "list" with duplicate members removed. If two or more members of the input are equal, the rightmost of those members is the one that remains in the output.

QUOTED thing (library procedure)

outputs its input, if a list; outputs its input with a quotation mark prepended, if a word.

MUTATORS

SETITEM index array value

command. Replaces the "index"th member of "array" with the new "value". Ensures that the resulting array is not circular, i.e., "value" may not be a list or array that contains "array".

MDSETITEM indexlist array value (library procedure)

command. Replaces the member of "array" chosen by "indexlist" with the new "value".

.SETFIRST list value

command. Changes the first member of "list" to be "value".

WARNING: Primitives whose names start with a period are DANGEROUS. Their use by non-experts is not recommended. The use of .SETFIRST can lead to circular list structures, which will get some Logo primitives into infinite loops, and to unexpected changes to other data structures that share storage with the list being modified.

.SETBF list value

command. Changes the butfirst of "list" to be "value".

WARNING: Primitives whose names start with a period are DANGEROUS. Their use by non-experts is not recommended. The use of .SETBF can lead to circular list structures, which will get some Logo primitives into infinite loops; unexpected changes to other data structures that share storage with the list being modified; or to Logo crashes and coredumps if the butfirst of a list is not itself a list.

.SETITEM index array value

command. Changes the "index"th member of "array" to be "value", like SETITEM, but without checking for circularity.

WARNING: Primitives whose names start with a period are DANGEROUS. Their use by non-experts is not recommended. The use of .SETITEM can lead to circular arrays, which will get some Logo primitives into infinite loops.

PUSH stackname thing (library procedure)

command. Adds the "thing" to the stack that is the value of the variable whose name is "stackname". This variable must have a list as its value; the initial value should be the empty list. New members are added at the front of the list.

POP stackname (library procedure)

outputs the most recently PUSHed member of the stack that is the value of the variable whose name is "stackname" and removes that member from the stack.

QUEUE queuename thing (library procedure)

command. Adds the "thing" to the queue that is the value of the variable whose name is "queuename". This variable must have a list as its value; the initial value should be the empty list. New members are added at the back of the list.

DEQUEUE queuename (library procedure)

outputs the least recently QUEUEd member of the queue that is the value of the variable whose name is "queuename" and removes that member from the queue.

PREDICATES

WORDP thing
WORD? thing

outputs TRUE if the input is a word, FALSE otherwise.

LISTP thing
LIST? thing

outputs TRUE if the input is a list, FALSE otherwise.

ARRAYP thing
ARRAY? thing

outputs TRUE if the input is an array, FALSE otherwise.

EMPTYP thing
EMPTY? thing

outputs TRUE if the input is the empty word or the empty list, FALSE otherwise.

EQUALP thing1 thing2
EQUAL? thing1 thing2
thing1 = thing2

outputs TRUE if the inputs are equal, FALSE otherwise. Two numbers are equal if they have the same numeric value. Two non-numeric words are equal if they contain the same characters in the same order. If there is a variable named CASEIGNOREDP whose value is TRUE, then an upper case letter is considered the same as the corresponding lower case letter. (This is the case by default.) Two lists are equal if their members are equal. An array is only equal to itself; two separately created arrays are never equal even if their members are equal. (It is important to be able to know if two expressions have the same array as their value because arrays are mutable; if, for example, two variables have the same array as their values then performing SETITEM on one of them will also change the other.)

NOTEQUALP thing1 thing2
NOTEQUAL? thing1 thing2
thing1 <> thing2

outputs FALSE if the inputs are equal, TRUE otherwise. See EQUALP for the meaning of equality for different data types.

BEFOREP word1 word2
BEFORE? word1 word2

outputs TRUE if word1 comes before word2 in ASCII collating sequence (for words of letters, in alphabetical order). Case-sensitivity is determined by the value of CASEIGNOREDP. Note that if the inputs are numbers, the result may not be the same as with LESSP; for example, BEFOREP 3 12 is false because 3 collates after 1.

.EQ thing1 thing2

outputs TRUE if its two inputs are the same datum, so that applying a mutator to one will change the other as well. Outputs FALSE otherwise, even if the inputs are equal in value.
WARNING: Primitives whose names start with a period are DANGEROUS. Their use by non-experts is not recommended. The use of mutators can lead to circular data structures, infinite loops, or Logo crashes.

MEMBERP thing1 thing2
MEMBER? thing1 thing2

if "thing2" is a list or an array, outputs TRUE if "thing1" is EQUALP to a member of "thing2", FALSE otherwise. If "thing2" is a word, outputs TRUE if "thing1" is a one-character word EQUALP to a character of "thing2", FALSE otherwise.

SUBSTRINGP thing1 thing2
SUBSTRING? thing1 thing2

if "thing1" or "thing2" is a list or an array, outputs FALSE. If "thing2" is a word, outputs TRUE if "thing1" is EQUALP to a substring of "thing2", FALSE otherwise.

NUMBERP thing
NUMBER? thing

outputs TRUE if the input is a number, FALSE otherwise.

QUERIES

COUNT thing

outputs the number of characters in the input, if the input is a word; outputs the number of members in the input, if it is a list or an array. (For an array, this may or may not be the index of the last member, depending on the array's origin.)

ASCII char

outputs the integer (between 0 and 255) that represents the input character in the ASCII code. Interprets control characters as representing vbarred punctuation, and returns the character code for the corresponding punctuation character without vertical bars. (Compare RAWASCII.)

RAWASCII char

outputs the integer (between 0 and 255) that represents the input character in the ASCII code. Interprets control characters as representing themselves. To find out the ASCII code of an arbitrary keystroke, use RAWASCII RC.

CHAR int

outputs the character represented in the ASCII code by the input, which must be an integer between 0 and 255.

MEMBER thing1 thing2

if "thing2" is a word or list and if MEMBERP with these inputs would output TRUE, outputs the portion of "thing2" from the first instance

of "thing1" to the end. If MEMBERP would output FALSE, outputs the empty word or list according to the type of "thing2". It is an error for "thing2" to be an array.

LOWERCASE word

outputs a copy of the input word, but with all uppercase letters changed to the corresponding lowercase letter.

UPPERCASE word

outputs a copy of the input word, but with all lowercase letters changed to the corresponding uppercase letter.

STANDOUT thing

outputs a word that, when printed, will appear like the input but displayed in standout mode (boldface, reverse video, or whatever your version does for standout). The word contains machine-specific magic characters at the beginning and end; in between is the printed form (as if displayed using TYPE) of the input. The output is always a word, even if the input is of some other type, but it may include spaces and other formatting characters. Note: a word output by STANDOUT while Logo is running on one machine will probably not have the desired effect if printed on another type of machine.

In the Macintosh classic version, the way that standout works is incompatible with the use of characters whose ASCII code is greater than 127. Therefore, you have a choice to make: The instruction

```
CANINVERSE 0
```

disables standout, but enables the display of ASCII codes above 127, and the instruction

```
CANINVERSE 1
```

restores the default situation in which standout is enabled and the extra graphic characters cannot be printed.

PARSE word

outputs the list that would result if the input word were entered in response to a READLIST operation. That is, PARSE READWORD has the same value as READLIST for the same characters read.

RUNPARSE wordorlist

outputs the list that would result if the input word or list were entered as an instruction line; characters such as infix operators and parentheses are separate members of the output. Note that sublists of a runparsed list are not themselves runparsed.

COMMUNICATION

TRANSMITTERS

PRINT thing
PR thing
(PRINT thing1 thing2 ...)
(PR thing1 thing2 ...)

command. Prints the input or inputs to the current write stream (initially the screen). All the inputs are printed on a single

line, separated by spaces, ending with a newline. If an input is a list, square brackets are not printed around it, but brackets are printed around sublists. Braces are always printed around arrays.

TYPE thing
(TYPE thing1 thing2 ...)

command. Prints the input or inputs like PRINT, except that no newline character is printed at the end and multiple inputs are not separated by spaces. Note: printing to the terminal is ordinarily "line buffered"; that is, the characters you print using TYPE will not actually appear on the screen until either a newline character is printed (for example, by PRINT or SHOW) or Logo tries to read from the keyboard (either at the request of your program or after an instruction prompt). This buffering makes the program much faster than it would be if each character appeared immediately, and in most cases the effect is not disconcerting. To accommodate programs that do a lot of positioned text display using TYPE, Logo will force printing whenever SETCURSOR is invoked. This solves most buffering problems. Still, on occasion you may find it necessary to force the buffered characters to be printed explicitly; this can be done using the WAIT command. WAIT 0 will force printing without actually waiting.

SHOW thing
(SHOW thing1 thing2 ...)

command. Prints the input or inputs like PRINT, except that if an input is a list it is printed inside square brackets.

RECEIVERS

READLIST
RL

reads a line from the read stream (initially the keyboard) and outputs that line as a list. The line is separated into members as though it were typed in square brackets in an instruction. If the read stream is a file, and the end of file is reached, READLIST outputs the empty word (not the empty list). READLIST processes backslash, vertical bar, and tilde characters in the read stream; the output list will not contain these characters but they will have had their usual effect. READLIST does not, however, treat semicolon as a comment character.

READWORD
RW

reads a line from the read stream and outputs that line as a word. The output is a single word even if the line contains spaces, brackets, etc. If the read stream is a file, and the end of file is reached, READWORD outputs the empty list (not the empty word). READWORD processes backslash, vertical bar, and tilde characters in the read stream. In the case of a tilde used for line continuation, the output word DOES include the tilde and the newline characters, so that the user program can tell exactly what the user entered. Vertical bars in the line are also preserved in the output. Backslash characters are not preserved in the output.

READRAWLINE

reads a line from the read stream and outputs that line as a word. The output is a single word even if the line contains spaces, brackets, etc. If the read stream is a file, and the end of file is reached, READRAWLINE outputs the empty list (not the empty word). READRAWLINE outputs the exact string of characters as they appear in the line, with no special meaning for backslash, vertical bar, tilde, or any other formatting characters.

READCHAR
RC

reads a single character from the read stream and outputs that character as a word. If the read stream is a file, and the end of file is reached, READCHAR outputs the empty list (not the empty word). If the read stream is the keyboard, echoing is turned off when READCHAR is invoked, and remains off until READLIST or READWORD is invoked or a Logo prompt is printed. Backslash, vertical bar, and tilde characters have no special meaning in this context.

READCHARS num
RCS num

reads "num" characters from the read stream and outputs those characters as a word. If the read stream is a file, and the end of file is reached, READCHARS outputs the empty list (not the empty word). If the read stream is a terminal, echoing is turned off when READCHARS is invoked, and remains off until READLIST or READWORD is invoked or a Logo prompt is printed. Backslash, vertical bar, and tilde characters have no special meaning in this context.

TERMINAL ACCESS

KEYP
KEY?

predicate, outputs TRUE if there are characters waiting to be read from the read stream. If the read stream is a file, this is equivalent to NOT EOFP. If the read stream is the terminal, then echoing is turned off and the terminal is set to CBREAK (character at a time instead of line at a time) mode. It remains in this mode until some line-mode reading is requested (e.g., READLIST). The Unix operating system forgets about any pending characters when it switches modes, so the first KEYP invocation will always output FALSE.

CLEARTEXT
CT

command. Clears the text window.

ARITHMETIC

NUMERIC OPERATIONS

SUM num1 num2
(SUM num1 num2 num3 ...)

num1 + num2

outputs the sum of its inputs.

DIFFERENCE num1 num2
num1 - num2

outputs the difference of its inputs. Minus sign means infix difference in ambiguous contexts (when preceded by a complete expression), unless it is preceded by a space and followed by a nonspace. (See also MINUS.)

MINUS num
- num

outputs the negative of its input. Minus sign means unary minus if the previous token is an infix operator or open parenthesis, or it is preceded by a space and followed by a nonspace. There is a difference in binding strength between the two forms:

```
MINUS 3 + 4     means   -(3+4)
- 3 + 4         means   (-3)+4
```

PRODUCT num1 num2
(PRODUCT num1 num2 num3 ...)
num1 * num2

outputs the product of its inputs.

QUOTIENT num1 num2
(QUOTIENT num)
num1 / num2

outputs the quotient of its inputs. The quotient of two integers is an integer if and only if the dividend is a multiple of the divisor. (In other words, QUOTIENT 5 2 is 2.5, not 2, but QUOTIENT 4 2 is 2, not 2.0 -- it does the right thing.) With a single input, QUOTIENT outputs the reciprocal of the input.

REMAINDER num1 num2

outputs the remainder on dividing "num1" by "num2"; both must be integers and the result is an integer with the same sign as num1.

MODULO num1 num2

outputs the remainder on dividing "num1" by "num2"; both must be integers and the result is an integer with the same sign as num2.

INT num

outputs its input with fractional part removed, i.e., an integer with the same sign as the input, whose absolute value is the largest integer less than or equal to the absolute value of the input.

ROUND num

outputs the nearest integer to the input.

SQRT num

outputs the square root of the input, which must be nonnegative.

POWER num1 num2

outputs "num1" to the "num2" power. If num1 is negative, then num2 must be an integer.

EXP num

outputs e (2.718281828+) to the input power.

LOG10 num

outputs the common logarithm of the input.

LN num

outputs the natural logarithm of the input.

SIN degrees

outputs the sine of its input, which is taken in degrees.

RADSIN radians

outputs the sine of its input, which is taken in radians.

COS degrees

outputs the cosine of its input, which is taken in degrees.

RADCOS radians

outputs the cosine of its input, which is taken in radians.

ARCTAN num
(ARCTAN x y)

outputs the arctangent, in degrees, of its input. With two inputs, outputs the arctangent of y/x, if x is nonzero, or 90 or -90 depending on the sign of y, if x is zero.

RADARCTAN num
(RADARCTAN x y)

outputs the arctangent, in radians, of its input. With two inputs, outputs the arctangent of y/x, if x is nonzero, or pi/2 or -pi/2 depending on the sign of y, if x is zero.

The expression 2*(RADARCTAN 0 1) can be used to get the value of pi.

ISEQ from to (library procedure)

outputs a list of the integers from FROM to TO, inclusive.

```
? show iseq 3 7
[3 4 5 6 7]
? show iseq 7 3
[7 6 5 4 3]
```

RSEQ from to count (library procedure)

outputs a list of COUNT equally spaced rational numbers

between FROM and TO, inclusive.

```
? show rseq 3 5 9
[3 3.25 3.5 3.75 4 4.25 4.5 4.75 5]
? show rseq 3 5 5
[3 3.5 4 4.5 5]
```

PREDICATES

LESSP num1 num2
LESS? num1 num2
num1 < num2

outputs TRUE if its first input is strictly less than its second.

GREATERP num1 num2
GREATER? num1 num2
num1 > num2

outputs TRUE if its first input is strictly greater than its second.

LESSEQUALP num1 num2
LESSEQUAL? num1 num2
num1 <= num2

outputs TRUE if its first input is less than or equal to its second.

GREATEREQUALP num1 num2
GREATEREQUAL? num1 num2
num1 >= num2

outputs TRUE if its first input is greater than or equal to its second.

RANDOM NUMBERS

RANDOM num
(RANDOM start end)

with one input, outputs a random nonnegative integer less than its input, which must be a positive integer.

With two inputs, RANDOM outputs a random integer greater than or equal to the first input, and less than or equal to the second input. Both inputs must be integers, and the first must be less than the second. (RANDOM 0 9) is equivalent to RANDOM 10; (RANDOM 3 8) is equivalent to (RANDOM 6)+3.

RERANDOM
(RERANDOM seed)

command. Makes the results of RANDOM reproducible. Ordinarily the sequence of random numbers is different each time Logo is used. If you need the same sequence of pseudo-random numbers repeatedly, e.g. to debug a program, say RERANDOM before the first invocation of RANDOM. If you need more than one repeatable sequence, you can give RERANDOM an integer input; each possible input selects a unique sequence of numbers.

LOGICAL OPERATIONS

AND tf1 tf2

(AND tf1 tf2 tf3 ...)

outputs TRUE if all inputs are TRUE, otherwise FALSE. All inputs must be TRUE or FALSE. (Comparison is case-insensitive regardless of the value of CASEIGNOREDP. That is, "true" or "True" or "TRUE" are all the same.) An input can be a list, in which case it is taken as an expression to run; that expression must produce a TRUE or FALSE value. List expressions are evaluated from left to right; as soon as a FALSE value is found, the remaining inputs are not examined. Example:

```
MAKE "RESULT AND [NOT (:X = 0)] [(1 / :X) > .5]
```

to avoid the division by zero if the first part is false.

OR tf1 tf2

(OR tf1 tf2 tf3 ...)

outputs TRUE if any input is TRUE, otherwise FALSE. All inputs must be TRUE or FALSE. (Comparison is case-insensitive regardless of the value of CASEIGNOREDP. That is, "true" or "True" or "TRUE" are all the same.) An input can be a list, in which case it is taken as an expression to run; that expression must produce a TRUE or FALSE value. List expressions are evaluated from left to right; as soon as a TRUE value is found, the remaining inputs are not examined. Example:

```
IF OR :X=0 [some.long.computation] [...]
```

to avoid the long computation if the first condition is met.

NOT tf

outputs TRUE if the input is FALSE, and vice versa. The input can be a list, in which case it is taken as an expression to run; that expression must produce a TRUE or FALSE value.

GRAPHICS

Logo begins with a black background and white pen.

TURTLE MOTION

FORWARD dist

FD dist

moves the turtle forward, in the direction that it's facing, by the specified distance (measured in turtle steps).

BACK dist

BK dist

moves the turtle backward, i.e., exactly opposite to the direction that it's facing, by the specified distance. (The heading of the turtle does not change.)

LEFT degrees

LT degrees

turns the turtle counterclockwise by the specified angle, measured in degrees (1/360 of a circle).

RIGHT degrees
RT degrees

turns the turtle clockwise by the specified angle, measured in degrees (1/360 of a circle).

SETPOS pos

moves the turtle to an absolute position in the graphics window. The input is a list of two numbers, the X and Y coordinates.

SETXY xcor ycor

moves the turtle to an absolute position in the graphics window. The two inputs are numbers, the X and Y coordinates.

SETX xcor

moves the turtle horizontally from its old position to a new absolute horizontal coordinate. The input is the new X coordinate.

SETY ycor

moves the turtle vertically from its old position to a new absolute vertical coordinate. The input is the new Y coordinate.

SETHEADING degrees
SETH degrees

turns the turtle to a new absolute heading. The input is a number, the heading in degrees clockwise from the positive Y axis.

HOME

moves the turtle to the center of the screen. Equivalent to SETPOS [0 0] SETHEADING 0.

ARC angle radius

draws an arc of a circle, with the turtle at the center, with the specified radius, starting at the turtle's heading and extending clockwise through the specified angle. The turtle does not move.

TURTLE MOTION QUERIES

POS

outputs the turtle's current position, as a list of two numbers, the X and Y coordinates.

XCOR (library procedure)

outputs a number, the turtle's X coordinate.

YCOR (library procedure)

outputs a number, the turtle's Y coordinate.

HEADING

outputs a number, the turtle's heading in degrees.

TOWARDS pos

outputs a number, the heading at which the turtle should be facing so that it would point from its current position to the position given as the input.

TURTLE AND WINDOW CONTROL

SHOWTURTLE
ST

makes the turtle visible.

HIDETURTLE
HT

makes the turtle invisible. It's a good idea to do this while you're in the middle of a complicated drawing, because hiding the turtle speeds up the drawing substantially.

CLEAN

erases all lines that the turtle has drawn on the graphics window. The turtle's state (position, heading, pen mode, etc.) is not changed.

CLEARSCREEN
CS

erases the graphics window and sends the turtle to its initial position and heading. Like HOME and CLEAN together.

WRAP

tells the turtle to enter wrap mode: From now on, if the turtle is asked to move past the boundary of the graphics window, it will "wrap around" and reappear at the opposite edge of the window. The top edge wraps to the bottom edge, while the left edge wraps to the right edge. (So the window is topologically equivalent to a torus.) This is the turtle's initial mode. Compare WINDOW and FENCE.

WINDOW

tells the turtle to enter window mode: From now on, if the turtle is asked to move past the boundary of the graphics window, it will move offscreen. The visible graphics window is considered as just part of an infinite graphics plane; the turtle can be anywhere on the plane. (If you lose the turtle, HOME will bring it back to the center of the window.) Compare WRAP and FENCE.

FENCE

tells the turtle to enter fence mode: From now on, if the turtle

is asked to move past the boundary of the graphics window, it will move as far as it can and then stop at the edge with an "out of bounds" error message. Compare WRAP and WINDOW.

LABEL text

takes a word or list as input, and prints the input on the graphics window, starting at the turtle's position.

SETLABELHEIGHT height

command (wxWidgets only). Takes a positive integer argument and tries to set the font size so that the character height (including descenders) is that many turtle steps. This will be different from the number of screen pixels if SETSCRUNCH has been used. Also, note that SETSCRUNCH changes the font size to try to preserve this height in turtle steps. Note that the query operation corresponding to this command is LABELSIZE, not LABELHEIGHT, because it tells you the width as well as the height of characters in the current font.

TURTLE AND WINDOW QUERIES

SHOWNP
SHOWN?

outputs TRUE if the turtle is shown (visible), FALSE if the turtle is hidden. See SHOWTURTLE and HIDETURTLE.

TURTLEMODE

outputs the word WRAP, FENCE, or WINDOW depending on the current turtle mode.

LABELSIZE

(wxWidgets only) outputs a list of two positive integers, the width and height of characters displayed by LABEL measured in turtle steps (which will be different from screen pixels if SETSCRUNCH has been used). There is no SETLABELSIZE because the width and height of a font are not separately controllable, so the inverse of this operation is SETLABELHEIGHT, which takes just one number for the desired height.

PEN AND BACKGROUND CONTROL

The turtle carries a pen that can draw pictures. At any time the pen can be UP (in which case moving the turtle does not change what's on the graphics screen) or DOWN (in which case the turtle leaves a trace). If the pen is down, it can operate in one of three modes: PAINT (so that it draws lines when the turtle moves), ERASE (so that it erases any lines that might have been drawn on or through that path earlier), or REVERSE (so that it inverts the status of each point along the turtle's path).

PENDOWN
PD

sets the pen's position to DOWN, without changing its mode.

PENUP

PU

sets the pen's position to UP, without changing its mode.

PENPAINT
PPT

sets the pen's position to DOWN and mode to PAINT.

PENERASE
PE

sets the pen's position to DOWN and mode to ERASE.

PENREVERSE
PX

sets the pen's position to DOWN and mode to REVERSE.
(This may interact in system-dependent ways with use of color.)

SETPENCOLOR colornumber.or.rgblist
SETPC colornumber.or.rgblist

sets the pen color to the given number, which must be a nonnegative integer. There are initial assignments for the first 16 colors:

0	black	1	blue	2	green	3	cyan
4	red	5	magenta	6	yellow	7	white
8	brown	9	tan	10	forest	11	aqua
12	salmon	13	purple	14	orange	15	grey

but other colors can be assigned to numbers by the PALETTE command. Alternatively, sets the pen color to the given RGB values (a list of three nonnegative numbers less than 100 specifying the percent saturation of red, green, and blue in the desired color).

SETPENSIZE size

sets the thickness of the pen. The input is either a single positive integer or a list of two positive integers (for horizontal and vertical thickness). Some versions pay no attention to the second number, but always have a square pen.

SETPEN list (library procedure)

sets the pen's position, mode, thickness, and hardware-dependent characteristics according to the information in the input list, which should be taken from an earlier invocation of PEN.

SETBACKGROUND colornumber.or.rgblist
SETBG colornumber.or.rgblist

set the screen background color by slot number or RGB values.
See SETPENCOLOR for details.

PEN QUERIES

PENDOWNP
PENDOWN?

outputs TRUE if the pen is down, FALSE if it's up.

PENMODE

outputs one of the words PAINT, ERASE, or REVERSE according to the current pen mode.

PENCOLOR
PC

outputs a color number, a nonnegative integer that is associated with a particular color, or a list of RGB values if such a list was used as the most recent input to SETPENCOLOR. There are initial assignments for the first 16 colors:

0	black	1	blue	2	green	3	cyan
4	red	5	magenta	6	yellow	7	white
8	brown	9	tan	10	forest	11	aqua
12	salmon	13	purple	14	orange	15	grey

but other colors can be assigned to numbers by the PALETTE command.

PENSIZE

outputs a list of two positive integers, specifying the horizontal and vertical thickness of the turtle pen. (In some implementations, including wxWidgets, the two numbers are always equal.)

PEN (library procedure)

outputs a list containing the pen's position, mode, thickness, and hardware-specific characteristics, for use by SETPEN.

BACKGROUND
BG

outputs the graphics background color, either as a slot number or as an RGB list, whichever way it was set. (See PENCOLOR.)

MOUSE QUERIES

MOUSEPOS

outputs the coordinates of the mouse, provided that it's within the graphics window, in turtle coordinates. If the mouse is outside the graphics window, then the last position within the window is returned. Exception: If a mouse button is pressed within the graphics window and held while the mouse is dragged outside the window, the mouse's position is returned as if the window were big enough to include it.

CLICKPOS

outputs the coordinates that the mouse was at when a mouse button was most recently pushed, provided that that position was within the graphics window, in turtle coordinates. (wxWidgets only)

BUTTONP
BUTTON?

outputs TRUE if a mouse button is down and the mouse is over the

graphics window. Once the button is down, BUTTONP remains true until the button is released, even if the mouse is dragged out of the graphics window.

BUTTON

outputs 0 if no mouse button has been pushed inside the Logo window since the last call to BUTTON. Otherwise, it outputs an integer between 1 and 3 indicating which button was most recently pressed. Ordinarily 1 means left, 2 means right, and 3 means center, but operating systems may reconfigure these.

WORKSPACE MANAGEMENT

PROCEDURE DEFINITION

TO procname :input1 :input2 ... (special form)

command. Prepares Logo to accept a procedure definition. The procedure will be named "procname" and there must not already be a procedure by that name. The inputs will be called "input1" etc. Any number of inputs are allowed, including none. Names of procedures and inputs are case-insensitive.

Unlike every other Logo procedure, TO takes as its inputs the actual words typed in the instruction line, as if they were all quoted, rather than the results of evaluating expressions to provide the inputs. (That's what "special form" means.)

This version of Logo allows variable numbers of inputs to a procedure. After the procedure name come four kinds of things, *in this order*:

1. 0 or more REQUIRED inputs :FOO :FROBOZZ
2. 0 or more OPTIONAL inputs [:BAZ 87] [:THINGO 5+9]
3. 0 or 1 REST input [:GARPLY]
4. 0 or 1 DEFAULT number 5

Every procedure has a MINIMUM, DEFAULT, and MAXIMUM number of inputs. (The latter can be infinite.)

The MINIMUM number of inputs is the number of required inputs, which must come first. A required input is indicated by the

```
:inputname
```

notation.

After all the required inputs can be zero or more optional inputs, each of which is represented by the following notation:

```
[:inputname default.value.expression]
```

When the procedure is invoked, if actual inputs are not supplied for these optional inputs, the default value expressions are evaluated to set values for the corresponding input names. The inputs are processed from left to right, so a default value expression can be based on earlier inputs. Example:

```
to proc :inlist [:startvalue first :inlist]
```

If the procedure is invoked by saying

```
proc [a b c]
```

then the variable INLIST will have the value [A B C] and the variable STARTVALUE will have the value A. If the procedure is invoked by saying

```
(proc [a b c] "x)
```

then INLIST will have the value [A B C] and STARTVALUE will have the value X.

After all the required and optional input can come a single "rest" input, represented by the following notation:

```
[:inputname]
```

This is a rest input rather than an optional input because there is no default value expression. There can be at most one rest input. When the procedure is invoked, the value of this inputname will be a list containing all of the actual inputs provided that were not used for required or optional inputs. Example:

```
to proc :in1 [:in2 "foo] [:in3 "baz] [:in4]
```

If this procedure is invoked by saying

```
proc "x
```

then IN1 has the value X, IN2 has the value FOO, IN3 has the value BAZ, and IN4 has the value [] (the empty list). If it's invoked by saying

```
(proc "a "b "c "d "e)
```

then IN1 has the value A, IN2 has the value B, IN3 has the value C, and IN4 has the value [D E].

The MAXIMUM number of inputs for a procedure is infinite if a rest input is given; otherwise, it is the number of required inputs plus the number of optional inputs.

The DEFAULT number of inputs for a procedure, which is the number of inputs that it will accept if its invocation is not enclosed in parentheses, is ordinarily equal to the minimum number. If you want a different default number you can indicate that by putting the desired default number as the last thing on the TO line. example:

```
to proc :in1 [:in2 "foo] [:in3] 3
```

This procedure has a minimum of one input, a default of three inputs, and an infinite maximum.

Logo responds to the TO command by entering procedure definition mode. The prompt character changes from "?" to ">" and whatever instructions you type become part of the definition until you type a line containing only the word END.

DEFINE procname text

command. Defines a procedure with name "procname" and text "text".
If there is already a procedure with the same name, the new
definition replaces the old one. The text input must be a list
whose members are lists. The first member is a list of inputs;
it looks like a TO line but without the word TO, without the
procedure name, and without the colons before input names. In
other words, the members of this first sublist are words for
the names of required inputs and lists for the names of optional
or rest inputs. The remaining sublists of the text input make
up the body of the procedure, with one sublist for each instruction
line of the body. (There is no END line in the text input.)
It is an error to redefine a primitive procedure unless the variable
REDEFP has the value TRUE.

TEXT procname

outputs the text of the procedure named "procname" in the form
expected by DEFINE: a list of lists, the first of which describes
the inputs to the procedure and the rest of which are the lines of
its body. The text does not reflect formatting information used
when the procedure was defined, such as continuation lines and
extra spaces.

FULLTEXT procname

outputs a representation of the procedure "procname" in which
formatting information is preserved. If the procedure was defined
with TO, EDIT, or LOAD, then the output is a list of words. Each
word represents one entire line of the definition in the form
output by READWORD, including extra spaces and continuation lines.
The last member of the output represents the END line. If the
procedure was defined with DEFINE, then the output is a list of
lists. If these lists are printed, one per line, the result will
look like a definition using TO. Note: the output from FULLTEXT
is not suitable for use as input to DEFINE!

COPYDEF newname oldname

command. Makes "newname" a procedure identical to "oldname".
The latter may be a primitive. If "newname" was already defined,
its previous definition is lost. If "newname" was already a
primitive, the redefinition is not permitted unless the variable
REDEFP has the value TRUE.

Note: dialects of Logo differ as to the order of inputs to COPYDEF.
This dialect uses "MAKE order," not "NAME order."

VARIABLE DEFINITION

MAKE varname value

command. Assigns the value "value" to the variable named "varname",
which must be a word. Variable names are case-insensitive. If a
variable with the same name already exists, the value of that
variable is changed. If not, a new global variable is created.

LOCAL varname
LOCAL varnamelist
(LOCAL varname1 varname2 ...)

command. Accepts as inputs one or more words, or a list of words. A variable is created for each of these words, with that word as its name. The variables are local to the currently running procedure. Logo variables follow dynamic scope rules; a variable that is local to a procedure is available to any subprocedure invoked by that procedure. The variables created by LOCAL have no initial value; they must be assigned a value (e.g., with MAKE) before the procedure attempts to read their value.

LOCALMAKE varname value (library procedure)

command. Makes the named variable local, like LOCAL, and assigns it the given value, like MAKE.

THING varname
:quoted.varname

outputs the value of the variable whose name is the input. If there is more than one such variable, the innermost local variable of that name is chosen. The colon notation is an abbreviation not for THING but for the combination

thing "

so that :FOO means THING "FOO.

GLOBAL varname
GLOBAL varnamelist
(GLOBAL varname1 varname2 ...)

command. Accepts as inputs one or more words, or a list of words. A global variable is created for each of these words, with that word as its name. The only reason this is necessary is that you might want to use the "setter" notation SETXYZ for a variable XYZ that does not already have a value; GLOBAL "XYZ makes that legal. Note: If there is currently a local variable of the same name, this command does *not* make Logo use the global value instead of the local one.

PROPERTY LISTS

Note: Names of property lists are always case-insensitive. Names of individual properties are case-sensitive or case-insensitive depending on the value of CASEIGNOREDP, which is TRUE by default.

In principle, every possible name is the name of a property list, which is initially empty. So Logo never gives a "no such property list" error, as it would for undefined procedure or variable names. But the primitive procedures that deal with "all" property lists (CONTENTS, PLISTS, etc.) list only nonempty ones. To "erase" a property list (see ERASE below) means to make it empty, removing all properties from it.

PPROP plistname propname value

command. Adds a property to the "plistname" property list with name "propname" and value "value".

GPROP plistname propname

outputs the value of the "propname" property in the "plistname"

property list, or the empty list if there is no such property.

REMPROP plistname propname

command. Removes the property named "propname" from the property list named "plistname".

PLIST plistname

outputs a list whose odd-numbered members are the names, and whose even-numbered members are the values, of the properties in the property list named "plistname". The output is a copy of the actual property list; changing properties later will not magically change a list output earlier by PLIST.

PREDICATES

PROCEDUREP name
PROCEDURE? name

outputs TRUE if the input is the name of a procedure.

PRIMITIVEP name
PRIMITIVE? name

outputs TRUE if the input is the name of a primitive procedure (one built into Logo). Note that some of the procedures described in this document are library procedures, not primitives.

DEFINEDP name
DEFINED? name

outputs TRUE if the input is the name of a user-defined procedure, including a library procedure.

NAMEP name
NAME? name

outputs TRUE if the input is the name of a variable.

PLISTP name
PLIST? name

outputs TRUE if the input is the name of a *nonempty* property list. (In principle every word is the name of a property list; if you haven't put any properties in it, PLIST of that name outputs an empty list, rather than giving an error message.)

CONTROL STRUCTURES

RUN instructionlist

command or operation. Runs the Logo instructions in the input list; outputs if the list contains an expression that outputs.

RUNRESULT instructionlist

runs the instructions in the input; outputs an empty list if those instructions produce no output, or a list whose only

member is the output from running the input instructionlist.
Useful for inventing command-or-operation control structures:

```
local "result
make "result runresult [something]
if emptyp :result [stop]
output first :result
```

REPEAT num instructionlist

command. Runs the "instructionlist" repeatedly, "num" times.

FOREVER instructionlist

command. Runs the "instructionlist" repeatedly, until something inside the instructionlist (such as STOP or THROW) makes it stop.

REPCOUNT

outputs the repetition count of the innermost current REPEAT or FOREVER, starting from 1. If no REPEAT or FOREVER is active, outputs -1.

The abbreviation # can be used for REPCOUNT unless the REPEAT is inside the template input to a higher order procedure such as FOREACH, in which case # has a different meaning.

IF tf instructionlist

(IF tf instructionlist1 instructionlist2)

command. If the first input has the value TRUE, then IF runs the second input. If the first input has the value FALSE, then IF does nothing. (If given a third input, IF acts like IFELSE, as described below.) It is an error if the first input is not either TRUE or FALSE.

For compatibility with earlier versions of Logo, if an IF instruction is not enclosed in parentheses, but the first thing on the instruction line after the second input expression is a literal list (i.e., a list in square brackets), the IF is treated as if it were IFELSE, but a warning message is given. If this aberrant IF appears in a procedure body, the warning is given only the first time the procedure is invoked in each Logo session.

IFELSE tf instructionlist1 instructionlist2

command or operation. If the first input has the value TRUE, then IFELSE runs the second input. If the first input has the value FALSE, then IFELSE runs the third input. IFELSE outputs a value if the instructionlist contains an expression that outputs a value.

TEST tf

command. Remembers its input, which must be TRUE or FALSE, for use by later IFTRUE or IFFALSE instructions. The effect of TEST is local to the procedure in which it is used; any corresponding IFTRUE or IFFALSE must be in the same procedure or a subprocedure.

IFTRUE instructionlist

IFT instructionlist

command. Runs its input if the most recent TEST instruction had

a TRUE input. The TEST must have been in the same procedure or a superprocedure.

IFFALSE instructionlist
IFF instructionlist

command. Runs its input if the most recent TEST instruction had a FALSE input. The TEST must have been in the same procedure or a superprocedure.

STOP

command. Ends the running of the procedure in which it appears. Control is returned to the context in which that procedure was invoked. The stopped procedure does not output a value.

OUTPUT value
OP value

command. Ends the running of the procedure in which it appears. That procedure outputs the value "value" to the context in which it was invoked. Don't be confused: OUTPUT itself is a command, but the procedure that invokes OUTPUT is an operation.

WAIT time

command. Delays further execution for "time" 60ths of a second. Also causes any buffered characters destined for the terminal to be printed immediately. WAIT 0 can be used to achieve this buffer flushing without actually waiting.

BYE

command. Exits from Logo; returns to the operating system.

.MAYBEOUTPUT value (special form)

works like OUTPUT except that the expression that provides the input value might not, in fact, output a value, in which case the effect is like STOP. This is intended for use in control structure definitions, for cases in which you don't know whether or not some expression produces a value. Example:

```
to invoke :function [:inputs] 2
.maybeoutput apply :function :inputs
end

? (invoke "print "a "b "c)
a b c
? print (invoke "word "a "b "c)
abc
```

This is an alternative to RUNRESULT. It's fast and easy to use, at the cost of being an exception to Logo's evaluation rules. (Ordinarily, it should be an error if the expression that's supposed to provide an input to something doesn't have a value.)

IGNORE value (library procedure)

command. Does nothing. Used when an expression is evaluated for a side effect and its actual value is unimportant.

` list (library procedure)

outputs a list equal to its input but with certain substitutions. If a member of the input list is the word "," (comma) then the following member should be an instructionlist that produces an output when run. That output value replaces the comma and the instructionlist. If a member of the input list is the word ",@" (comma atsign) then the following member should be an instructionlist that outputs a list when run. The members of that list replace the ,@ and the instructionlist. Example:

```
show `[foo baz ,[bf [a b c]] garply ,@[bf [a b c]]]
```

will print

```
[foo baz [b c] garply b c]
```

A word starting with , or ,@ is treated as if the rest of the word were a one-word list, e.g., ,:FOO is equivalent to ,[:FOO].

A word starting with ", (quote comma) or :, (colon comma) becomes a word starting with " or : but with the result of running the substitution (or its first word, if the result is a list) replacing what comes after the comma.

Backquotes can be nested. Substitution is done only for commas at the same depth as the backquote in which they are found:

```
? show `[a `[b ,[1+2] ,[foo ,[1+3] d] e] f]
[a ` [b , [1+2] , [foo 4 d] e] f]

?make "name1 "x
?make "name2 "y
? show `[a `[b ,:,:name1 ,",:name2 d] e]
[a ` [b , [:x] , ["y] d] e]
```

FOR forcontrol instructionlist (library procedure)

command. The first input must be a list containing three or four members: (1) a word, which will be used as the name of a local variable; (2) a word or list that will be evaluated as by RUN to determine a number, the starting value of the variable; (3) a word or list that will be evaluated to determine a number, the limit value of the variable; (4) an optional word or list that will be evaluated to determine the step size. If the fourth member is missing, the step size will be 1 or -1 depending on whether the limit value is greater than or less than the starting value, respectively.

The second input is an instructionlist. The effect of FOR is to run that instructionlist repeatedly, assigning a new value to the control variable (the one named by the first member of the forcontrol list) each time. First the starting value is assigned to the control variable. Then the value is compared to the limit value. FOR is complete when the sign of (current - limit) is the same as the sign of the step size. (If no explicit step size is provided, the instructionlist is always run at least once. An explicit step size can lead to a zero-trip FOR, e.g., FOR [I 1 0 1] ...) Otherwise, the instructionlist is run, then the step is added to the current value of the control variable and FOR returns to the comparison step.

```
? for [i 2 7 1.5] [print :i]
2
3.5
5
```

DO.WHILE instructionlist tfexpression (library procedure)

command. Repeatedly evaluates the "instructionlist" as long as the evaluated "tfexpression" remains TRUE. Evaluates the first input first, so the "instructionlist" is always run at least once. The "tfexpression" must be an expressionlist whose value when evaluated is TRUE or FALSE.

WHILE tfexpression instructionlist (library procedure)

command. Repeatedly evaluates the "instructionlist" as long as the evaluated "tfexpression" remains TRUE. Evaluates the first input first, so the "instructionlist" may never be run at all. The "tfexpression" must be an expressionlist whose value when evaluated is TRUE or FALSE.

DO.UNTIL instructionlist tfexpression (library procedure)

command. Repeatedly evaluates the "instructionlist" as long as the evaluated "tfexpression" remains FALSE. Evaluates the first input first, so the "instructionlist" is always run at least once. The "tfexpression" must be an expressionlist whose value when evaluated is TRUE or FALSE.

UNTIL tfexpression instructionlist (library procedure)

command. Repeatedly evaluates the "instructionlist" as long as the evaluated "tfexpression" remains FALSE. Evaluates the first input first, so the "instructionlist" may never be run at all. The "tfexpression" must be an expressionlist whose value when evaluated is TRUE or FALSE.

CASE value clauses (library procedure)

command or operation. The second input is a list of lists (clauses); each clause is a list whose first element is either a list of values or the word ELSE and whose butfirst is a Logo expression or instruction. CASE examines the clauses in order. If a clause begins with the word ELSE (upper or lower case), then the butfirst of that clause is evaluated and CASE outputs its value, if any. If the first input to CASE is a member of the first element of a clause, then the butfirst of that clause is evaluated and CASE outputs its value, if any. If neither of these conditions is met, then CASE goes on to the next clause. If no clause is satisfied, CASE does nothing. Example:

```
to vowelp :letter
output case :letter [ [[a e i o u] "true] [else "false] ]
end
```

COND clauses (library procedure)

command or operation. The input is a list of lists (clauses); each clause is a list whose first element is either an expression whose value is TRUE or FALSE, or the word ELSE, and whose butfirst is a Logo expression or instruction. COND examines the clauses in order. If a clause begins with the word ELSE (upper or lower case), then the butfirst of that clause is evaluated and CASE outputs its value, if any. Otherwise, the first element of the clause is evaluated; the resulting value must be TRUE or FALSE. If it's TRUE, then the butfirst of that clause is evaluated and COND outputs its value, if

any. If the value is FALSE, then COND goes on to the next clause. If no clause is satisfied, COND does nothing. Example:

```
to evens :numbers      ; select even numbers from a list
op cond [ [[emptyp :numbers] []]
          [[evenp first :numbers]  ; assuming EVENP is defined
           fput first :numbers evens butfirst :numbers]
          [else evens butfirst :numbers] ]
end
```

TEMPLATE-BASED ITERATION

The procedures in this section are iteration tools based on the idea of a "template." This is a generalization of an instruction list or an expression list in which "slots" are provided for the tool to insert varying data. Four different forms of template can be used.

The most commonly used form for a template is "explicit-slot" form, or "question mark" form. Example:

```
? show map [? * ?] [2 3 4 5]
[4 9 16 25]
?
```

In this example, the MAP tool evaluated the template [? * ?] repeatedly, with each of the members of the data list [2 3 4 5] substituted in turn for the question marks. The same value was used for every question mark in a given evaluation. Some tools allow for more than one datum to be substituted in parallel; in these cases the slots are indicated by ?1 for the first datum, ?2 for the second, and so on:

```
? show (map [(word ?1 ?2 ?1)] [a b c] [d e f])
[ada beb cfc]
?
```

If the template wishes to compute the datum number, the form (? 1) is equivalent to ?1, so (? ?1) means the datum whose number is given in datum number 1. Some tools allow additional slot designations, as shown in the individual descriptions.

The second form of template is the "named-procedure" form. If the template is a word rather than a list, it is taken as the name of a procedure. That procedure must accept a number of inputs equal to the number of parallel data slots provided by the tool; the procedure is applied to all of the available data in order. That is, if data ?1 through ?3 are available, the template "PROC is equivalent to [PROC ?1 ?2 ?3].

```
? show (map "word [a b c] [d e f])
[ad be cf]
?

to dotprod :a :b       ; vector dot product
op apply "sum (map "product :a :b)
end
```

The third form of template is "named-slot" or "lambda" form. This form is indicated by a template list containing more than one member, whose first member is itself a list. The first member is taken as a list of names; local variables are created with those names and given the available data in order as their values. The number of names must equal the number of available data. This form is needed primarily when one iteration tool must

be used within the template list of another, and the ? notation would be ambiguous in the inner template. Example:

```
to matmul :m1 :m2 [:tm2 transpose :m2]; multiply two matrices
output map [[row] map [[col] dotprod :row :col] :tm2] :m1
end
```

The fourth form is "procedure text" form, a variant of lambda form. In this form, the template list contains at least two members, all of which are lists. This is the form used by the DEFINE and TEXT primitives, and APPLY accepts it so that the text of a defined procedure can be used as a template.

Note: The fourth form of template is interpreted differently from the others, in that Logo considers it to be an independent defined procedure for the purposes of OUTPUT and STOP. For example, the following two instructions are identical:

```
? print apply [[x] :x+3] [5]
8
? print apply [[x] [output :x+3]] [5]
8
```

although the first instruction is in named-slot form and the second is in procedure-text form. The named-slot form can be understood as telling Logo to evaluate the expression :x+3 in place of the entire invocation of apply, with the variable x temporarily given the value 5. The procedure-text form can be understood as invoking the procedure

```
to foo :x
output :x+3
end
```

with input 5, but without actually giving the procedure a name. If the use of OUTPUT were interchanged in these two examples, we'd get errors:

```
? print apply [[x] output :x+3] [5]
Can only use output inside a procedure
? print apply [[x] [:x+3]] [5]
You don't say what to do with 8
```

The named-slot form can be used with STOP or OUTPUT inside a procedure, to stop the enclosing procedure.

The following iteration tools are extended versions of the ones in Appendix B of the book _Computer_Science_Logo_Style,_Volume_3:_Advanced_Topics_ by Brian Harvey [MIT Press, 1987]. The extensions are primarily to allow for variable numbers of inputs.

APPLY template inputlist

command or operation. Runs the "template," filling its slots with the members of "inputlist." The number of members in "inputlist" must be an acceptable number of slots for "template." It is illegal to apply the primitive TO as a template, but anything else is okay. APPLY outputs what "template" outputs, if anything.

INVOKE template input (library procedure)
(INVOKE template input1 input2 ...)

command or operation. Exactly like APPLY except that the inputs are provided as separate expressions rather than in a list.

FOREACH data template (library procedure)
(FOREACH data1 data2 ... template)

command. Evaluates the template list repeatedly, once for each member of the data list. If more than one data list are given, each of them must be the same length. (The data inputs can be words, in which case the template is evaluated once for each character.)

In a template, the symbol ?REST represents the portion of the data input to the right of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then ?REST would be replaced by [C D E]. If multiple parallel slots are used, then (?REST 1) goes with ?1, etc.

In a template, the symbol # represents the position in the data input of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then # would be replaced by 2.

MAP template data (library procedure)
(MAP template data1 data2 ...)

outputs a word or list, depending on the type of the data input, of the same length as that data input. (If more than one data input are given, the output is of the same type as data1.) Each member of the output is the result of evaluating the template list, filling the slots with the corresponding member(s) of the data input(s). (All data inputs must be the same length.) In the case of a word output, the results of the template evaluation must be words, and they are concatenated with WORD.

In a template, the symbol ?REST represents the portion of the data input to the right of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then ?REST would be replaced by [C D E]. If multiple parallel slots are used, then (?REST 1) goes with ?1, etc.

In a template, the symbol # represents the position in the data input of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then # would be replaced by 2.

MAP.SE template data (library procedure)
(MAP.SE template data1 data2 ...)

outputs a list formed by evaluating the template list repeatedly and concatenating the results using SENTENCE. That is, the members of the output are the members of the results of the evaluations. The output list might, therefore, be of a different length from that of the data input(s). (If the result of an evaluation is the empty list, it contributes nothing to the final output.) The data inputs may be words or lists.

In a template, the symbol ?REST represents the portion of the data input to the right of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then

?REST would be replaced by [C D E]. If multiple parallel slots are used, then (?REST 1) goes with ?1, etc.

In a template, the symbol # represents the position in the data input of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then # would be replaced by 2.

FILTER tftemplate data (library procedure)

outputs a word or list, depending on the type of the data input, containing a subset of the members (for a list) or characters (for a word) of the input. The template is evaluated once for each member or character of the data, and it must produce a TRUE or FALSE value. If the value is TRUE, then the corresponding input constituent is included in the output.

```
? print filter "vowelp "elephant
eea
?
```

In a template, the symbol ?REST represents the portion of the data input to the right of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then ?REST would be replaced by [C D E].

In a template, the symbol # represents the position in the data input of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then # would be replaced by 2.

FIND tftemplate data (library procedure)

outputs the first constituent of the data input (the first member of a list, or the first character of a word) for which the value produced by evaluating the template with that consituent in its slot is TRUE. If there is no such constituent, the empty list is output.

In a template, the symbol ?REST represents the portion of the data input to the right of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then ?REST would be replaced by [C D E].

In a template, the symbol # represents the position in the data input of the member currently being used as the ? slot-filler. That is, if the data input is [A B C D E] and the template is being evaluated with ? replaced by B, then # would be replaced by 2.

REDUCE template data (library procedure)

outputs the result of applying the template to accumulate the members of the data input. The template must be a two-slot function. Typically it is an associative function name like SUM. If the data input has only one constituent (member in a list or character in a word), the output is that consituent. Otherwise, the template is first applied with ?1 filled with the next-to-last consitient and ?2 with the last constituent. Then, if there are

more constituents, the template is applied with ?1 filled with the next constituent to the left and ?2 with the result from the previous evaluation. This process continues until all constituents have been used. The data input may not be empty.

Note: If the template is, like SUM, the name of a procedure that is capable of accepting arbitrarily many inputs, it is more efficient to use APPLY instead of REDUCE. The latter is good for associative procedures that have been written to accept exactly two inputs:

```
to max :a :b
output ifelse :a > :b [:a] [:b]
end

print reduce "max [...]
```

Alternatively, REDUCE can be used to write MAX as a procedure that accepts any number of inputs, as SUM does:

```
to max [:inputs] 2
if emptyp :inputs ~
   [(throw "error [not enough inputs to max])]
output reduce [ifelse ?1 > ?2 [?1] [?2]] :inputs
end
```

CROSSMAP template listlist (library procedure)

(CROSSMAP template data1 data2 ...)

outputs a list containing the results of template evaluations. Each data list contributes to a slot in the template; the number of slots is equal to the number of data list inputs. As a special case, if only one data list input is given, that list is taken as a list of data lists, and each of its members contributes values to a slot. CROSSMAP differs from MAP in that instead of taking members from the data inputs in parallel, it takes all possible combinations of members of data inputs, which need not be the same length.

```
? show (crossmap [word ?1 ?2] [a b c] [1 2 3 4])
[a1 a2 a3 a4 b1 b2 b3 b4 c1 c2 c3 c4]
?
```

For compatibility with the version in the first edition of CSLS, CROSSMAP templates may use the notation :1 instead of ?1 to indicate slots.

CASCADE endtest template startvalue (library procedure)

(CASCADE endtest tmp1 sv1 tmp2 sv2 ...)
(CASCADE endtest tmp1 sv1 tmp2 sv2 ... finaltemplate)

outputs the result of applying a template (or several templates, as explained below) repeatedly, with a given value filling the slot the first time, and the result of each application filling the slot for the following application.

In the simplest case, CASCADE has three inputs. The second input is a one-slot expression template. That template is evaluated some number of times (perhaps zero). On the first evaluation, the slot is filled with the third input; on subsequent evaluations, the slot is filled with the result of the previous evaluation. The number of evaluations is determined by the first input. This can be either a nonnegative integer, in which case the template is evaluated that many times, or a predicate expression template, in

which case it is evaluated (with the same slot filler that will be used for the evaluation of the second input) repeatedly, and the CASCADE evaluation continues as long as the predicate value is FALSE. (In other words, the predicate template indicates the condition for stopping.)

If the template is evaluated zero times, the output from CASCADE is the third (startvalue) input. Otherwise, the output is the value produced by the last template evaluation.

CASCADE templates may include the symbol # to represent the number of times the template has been evaluated. This slot is filled with 1 for the first evaluation, 2 for the second, and so on.

```
? show cascade 5 [lput # ?] []
[1 2 3 4 5]
? show cascade [vowelp first ?] [bf ?] "spring
ing
? show cascade 5 [# * ?] 1
120
?
```

Several cascaded results can be computed in parallel by providing additional template-startvalue pairs as inputs to CASCADE. In this case, all templates (including the endtest template, if used) are multi-slot, with the number of slots equal to the number of pairs of inputs. In each round of evaluations, ?2, for example, represents the result of evaluating the second template in the previous round. If the total number of inputs (including the first endtest input) is odd, then the output from CASCADE is the final value of the first template. If the total number of inputs is even, then the last input is a template that is evaluated once, after the end test is satisfied, to determine the output from CASCADE.

```
to fibonacci :n
output (cascade :n [?1 + ?2] 1 [?1] 0)
end

to piglatin :word
output (cascade [vowelp first ?] ~
                [word bf ? first ?] ~
                :word ~
                [word ? "ay])
end
```

CASCADE.2 endtest temp1 startval1 temp2 startval2 (library procedure)

outputs the result of invoking CASCADE with the same inputs. The only difference is that the default number of inputs is five instead of three.

TRANSFER endtest template inbasket (library procedure)

outputs the result of repeated evaluation of the template. The template is evaluated once for each member of the list "inbasket." TRANSFER maintains an "outbasket" that is initially the empty list. After each evaluation of the template, the resulting value becomes the new outbasket.

In the template, the symbol ?IN represents the current member from the inbasket; the symbol ?OUT represents the entire current outbasket. Other slot symbols should not be used.

If the first (endtest) input is an empty list, evaluation continues until all inbasket members have been used. If not, the first input must be a predicate expression template, and evaluation continues until either that template's value is TRUE or the inbasket is used up.

SPECIAL VARIABLES

Logo takes special action if any of the following variable names exists. They follow the normal scoping rules, so a procedure can locally set one of them to limit the scope of its effect. Initially, no variables exist except for ALLOWGETSET, CASEIGNOREDP, and UNBURYONEDIT, which are TRUE and buried.

ALLOWGETSET (variable)

if TRUE, indicates that an attempt to use a procedure that doesn't exist should be taken as an implicit getter or setter procedure (setter if the first three letters of the name are SET) for a variable of the same name (without the SET if appropriate).

BUTTONACT (variable)

if nonempty, should be an instruction list that will be evaluated whenever a mouse button is pressed. Note that the user may have released the button before the instructions are evaluated. BUTTON will still output which button was most recently pressed. CLICKPOS will output the position of the mouse cursor at the moment the button was pressed; this may be different from MOUSEPOS if the user moves the mouse after clicking.

Note that it's possible for the user to press a button during the evaluation of the instruction list. If this would confuse your program, prevent it by temporarily setting BUTTONACT to the empty list. One easy way to do that is the following:

```
make "buttonact [button.action]

to button.action [:buttonact []]
... ; whatever you want the button to do
end
```

KEYACT (variable)

if nonempty, should be an instruction list that will be evaluated whenever a key is pressed on the keyboard. The instruction list can use READCHAR to find out what key was pressed. Note that only keys that produce characters qualify; pressing SHIFT or CONTROL alone will not cause KEYACT to be evaluated.

Note that it's possible for the user to press a key during the evaluation of the instruction list. If this would confuse your program, prevent it by temporarily setting KEYACT to the empty list. One easy way to do that is the following:

```
make "keyact [key.action]

to key.action [:keyact []]
... ; whatever you want the key to do
end
```