Appendices

Berkeley Logo Reference Manual

Entering and Leaving Logo

The process to start Logo depends on your operating system:

Unix Type the word `logo` to the shell. (The directory in which you've installed Logo must be in your path.)

DOS Change directories to the one containing Logo (probably `c:\ucblogo`). Then type `ucblogo` for the large memory version, or `bl` for the 640K version.

Mac Double-click on the `logo` icon within the `UCB Logo` folder.

To leave Logo, enter the command `bye`.

Under Unix or DOS, if you include one or more filenames on the command line when starting Logo, those files will be loaded before the interpreter starts reading commands from your terminal. If you load a file that executes some program that includes a `bye` command, Logo will run that program and exit. You can therefore write standalone programs in Logo and run them with shell/batch scripts. To support this technique, Logo does not print its usual welcoming and parting messages if you give file arguments to the logo command.

If you type your interrupt character (see table below) Logo will stop what it's doing and return to toplevel, as if you did `throw "toplevel`. If you type your quit character Logo will pause as if you did `pause`.

	Unix	DOS	Mac
toplevel	usually ctrl-C	ctrl-Q	command-. (period)
pause	usually ctrl-\	ctrl-W	command-, (comma)

If you have an environment variable called `LOGOLIB` whose value is the name of a directory, then Logo will use that directory instead of the default library. If you invoke a procedure that has not been defined, Logo first looks for a file in the current directory named `proc.lg` where `proc` is the procedure name in lower case letters. If such a file exists, Logo loads that file. If the missing procedure is still undefined, or if there is no such file, Logo then looks in the library directory for a file named `proc` (no `.lg`) and, if it exists, loads it. If neither file contains a definition for the procedure, then Logo signals an error. Several procedures that are primitive in most versions of Logo are included in the default library, so if you use a different library you may want to include some or all of the default library in it.

Tokenization

Names of procedures, variables, and property lists are case-insensitive. So are the special words `end`, `true`, and `false`. Case of letters is preserved in everything you type, however.

Within square brackets, words are delimited only by spaces and square brackets. `[2+3]` is a list containing one word. Note, however, that the Logo primitives that interpret such a list as a Logo instruction or expression (`run`, `if`, etc.) reparse the list as if it had not been typed inside brackets.

After a quotation mark outside square brackets, a word is delimited by a space, a square bracket, or a parenthesis.

A word not after a quotation mark or inside square brackets is delimited by a space, a bracket, a parenthesis, or an infix operator `+-*/=<>`. Note that words following colons are in this category. Note that quote and colon are not delimiters.

A word consisting of a question mark followed by a number (e.g., `?37`), when runparsed (i.e., where a procedure name is expected), is treated as if it were the sequence

```
( ? 37 )
```

making the number an input to the `?` procedure. (See the discussion of templates, below.) This special treatment does not apply to words read as data, to words with a non-number following the question mark, or if the question mark is backslashed.

A line (an instruction line or one read by `readlist` or `readword`) can be continued onto the following line if its last character is a tilde (`~`). `Readword` preserves the tilde and the newline; `readlist` does not.

An instruction line or a line read by `readlist` (but not by `readword`) is automatically continued to the next line, as if ended with a tilde, if there are unmatched brackets, parentheses, braces, or vertical bars pending. However, it's an error if the continuation line contains only the word `end`; this is to prevent runaway procedure definitions. Lines eplicitly continued with a tilde avoid this restriction.

If a line being typed interactively on the keyboard is continued, either with a tilde or automatically, Logo will display a tilde as a prompt character for the continuation line.

A semicolon begins a comment in an instruction line. Logo ignores characters from the semicolon to the end of the line. A tilde as the last character still indicates a continuation line, but not a continuation of the comment. For example, typing the instruction

```
print "abc;comment ~
def
```

will print the word `abcdef`. Semicolon has no special meaning in data lines read by `readword` or `readlist`, but such a line can later be reparsed using `runparse` and then comments will be recognized.

To include an otherwise delimiting character (including semicolon or tilde) in a word, precede it with backslash (`\`). If the last character of a line is a backslash, then the newline character following the backslash will be part of the last word on the line, and the line continues onto the following line. To include a backslash in a word, use `\\`. If the combination backslash-newline is entered at the terminal, Logo will issue a backslash as a prompt character for the continuation line. All of this applies to data lines read with `readword` or `readlist` as well as to instruction lines. A character entered with backslash is `equalp` to the same character without the backslash, but can be distinguished by the `backslashedp` predicate. (However, `backslashedp` recgnizes backslashedness only on characters for which it is necessary: whitespace, parentheses, brackets, infix operators, backslash, vertical bar, tilde, quote, question mark, colon, and semicolon.)

An alternative notation to include otherwise delimiting characters in words is to enclose a group of characters in vertical bars. All characters between vertical bars are treated as if they were letters. In data read with `readword` the vertical bars are preserved in the resulting word. In data read with `readlist` (or resulting from a `parse` or `runparse` of a word) the vertical bars do not appear explicitly; all potentially delimiting characters (including spaces, brackets, parentheses, and infix operators) appear as though entered with a backslash. Within vertical bars, backslash may still be used; the only characters that must be backslashed in this context are backslash and vertical bar themselves.

Characters entered between vertical bars are forever special, even if the word or list containing them is later reparsed with `parse` or `runparse`. Characters typed after a backslash are treated somewhat differently: When a quoted word containing a backslashed character is runparsed, the backslashed character loses its special quality and acts thereafter as if typed normally. This distinction is important only if you are building a Logo expression out of parts, to be `run` later, and want to use parentheses. For example,

```
print run (se "\( 2 "+ 3 "\))
```

will print `5`, but

```
run (se "make ""|(| 2)
```

will create a variable whose name is open-parenthesis. (Each example would fail if vertical bars and backslashes were interchanged.)

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.

lput *thing list* outputs a list equal to its second input with one extra member, the first input, at the end.

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* **(**library procedure**)**
(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* **(**library procedure**)** 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 the result of `word` *thing1 thing2*. If *thing2* is a list, outputs the result of `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; unexpected changes to other data structures that share storage with the list being modified; and the loss of memory if a circular structure is released.

.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; Logo crashes and coredumps if the butfirst of a list is not itself a list; and the loss of memory if a circular structure is released.

.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; and the loss of memory if a circular structure is released.

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.)

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.

`backslashedp` *`char`*
`backslashed?` *`char`* outputs `true` if the input character was originally entered into Logo with a backslash (`\`) before it or within vertical bars (`|`) to prevent its usual special syntactic meaning, `false` otherwise. (Outputs `true` only if the character is a backslashed space, tab, newline, or one of `()[]+-*/=<>":;\~?|`.)

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 backslashed punctuation, and returns the character code for the corresponding punctuation character without backslash. (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 terminal does for standout). The word contains terminal-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 terminal will probably not have the desired effect if printed on another type of terminal.

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

Note: If there is a variable named `printdepthlimit` with a nonnegative integer value, then complex list and array structures will be printed only to the allowed depth. That is, members of members of... of members will be allowed only so far. The members omitted because they are just past the depth limit are indicated by an ellipsis for each one, so a too-deep list of two members will print as `[... ...]`.

If there is a variable named `printwidthlimit` with a nonnegative integer value, then only the first so many members of any array or list will be printed. A single ellipsis replaces all missing data within the structure. The width limit also applies to the number of characters printed in a word, except that a `printwidthlimit` between 0 and 9 will be treated as if it were 10 when applied to words. This limit applies not only to the top-level printed datum but to any substructures within it.

print *thing*
pr *thing*
(print *thing1 thing2* **...)**
(pr *thing1 thing2* **...)** command. Prints the input or inputs to the current write stream (initially the terminal). 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 terminal) 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, but the character following the backslash is marked internally; programs can use `backslashedp` to check for this marking. (Backslashedness is preserved only for certain characters. See `backslashedp`.)

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 a terminal, 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.

shell *command*
(shell *command wordflag***)** Under Unix, outputs the result of running *command* as a shell command. (The command is sent to /bin/sh, not csh or other alternatives.) If the command is a literal list in the instruction line, and if you want a backslash character sent to the shell, you must use \\ to get the backslash through Logo's reader intact. The output is a list containing one member for each line generated by the shell command. Ordinarily each such line is represented by a list in the output, as though the line were read using `readlist`. If a second input is given, regardless of the value of the input, each line is represented by a word in the output as though it were read with `readword`. Example:

```
to dayofweek
output first first shell [date]
end
```

This is `first first` to extract the first word of the first (and only) line of the shell output.

Under DOS, `shell` is a command, not an operation; it sends its input to a DOS command processor but does not collect the result of the command.

The Macintosh, of course, is not programmable.

File Access

openread *filename* command. Opens the named file for reading. The read position is initially at the beginning of the file.

openwrite *filename* command. Opens the named file for writing. If the file already existed, the old version is deleted and a new, empty file created.

openappend *filename* command. Opens the named file for writing. If the file already exists, the write position is initially set to the end of the old file, so that newly written data will be appended to it.

openupdate *filename* command. Opens the named file for reading and writing. The read and write position is initially set to the end of the old file, if any. Note: each open file has

only one position, for both reading and writing. If a file opened for update is both `reader` and `writer` at the same time, then `setreadpos` will also affect `writepos` and vice versa. Also, if you alternate reading and writing the same file, you must `setreadpos` between a write and a read, and `setwritepos` between a read and a write.

`close` *`filename`* command. Closes the named file.

`allopen` outputs a list whose members are the names of all files currently open. This list does not include the dribble file, if any.

`closeall` **(**library procedure**)** command. Closes all open files.
Abbreviates `foreach allopen [close ?]`

`erasefile` *`filename`*
`erf` *`filename`* command. Erases (deletes, removes) the named file, which should not currently be open.

`dribble` *`filename`* command. Creates a new file whose name is the input, like `openwrite`, and begins recording in that file everything that is read from the keyboard or written to the terminal. That is, this writing is in addition to the writing to `writer`. The intent is to create a transcript of a Logo session, including things like prompt characters and interactions.

`nodribble` command. Stops copying information into the dribble file, and closes the file.

`setread` *`filename`* command. Makes the named file the read stream, used for `readlist`, etc. The file must already be open with `openread` or `openupdate`. If the input is the empty list, then the read stream becomes the terminal, as usual. Changing the read stream does not close the file that was previously the read stream, so it is possible to alternate between files.

`setwrite` *`filename`* command. Makes the named file the write stream, used for `print`, etc. The file must already be open with `openwrite`, `openappend`, or `openupdate`. If the input is the empty list, then the write stream becomes the terminal, as usual. Changing the write stream does not close the file that was previously the write stream, so it is possible to alternate between files.

`reader` outputs the name of the current read stream file, or the empty list if the read stream is the terminal.

`writer` outputs the name of the current write stream file, or the empty list if the write stream is the terminal.

`setreadpos` *`charpos`* command. Sets the file pointer of the read stream file so that the next `readlist`, etc., will begin reading at the *`charpos`*th character in the file, counting from 0. (That is, `setreadpos 0` will start reading from the beginning of the file.) Meaningless if the read stream is the terminal.

`setwritepos` *`charpos`* command. Sets the file pointer of the write stream file so that the next `print`, etc., will begin writing at the *`charpos`*th character in the file, counting from 0. (That is, `setwritepos 0` will start writing from the beginning of the file.) Meaningless if the write stream is the terminal.

readpos outputs the file position of the current read stream file.

writepos outputs the file position of the current write stream file.

eofp
eof? predicate, outputs `true` if there are no more characters to be read in the read stream file, `false` otherwise.

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 screen of the terminal.

setcursor *vector* command. The input is a list of two numbers, the x and y coordinates of a screen position (origin in the upper left corner, positive direction is southeast). The screen cursor is moved to the requested position. This command also forces the immediate printing of any buffered characters.

cursor outputs a list containing the current x and y coordinates of the screen cursor. Logo may get confused about the current cursor position if, e.g., you type in a long line that wraps around or your program prints escape codes that affect the terminal strangely.

setmargins *vector* command. The input must be a list of two numbers, as for `setcursor`. The effect is to clear the screen and then arrange for all further printing to be shifted down and to the right according to the indicated margins. Specifically, every time a newline character is printed (explicitly or implicitly) Logo will type `x_margin` spaces, and on every invocation of `setcursor` the margins will be added to the input x and y coordinates. (`Cursor` will report the cursor position relative to the margins, so that this shift will be invisible to Logo programs.) The purpose of this command is to accommodate the display of terminal screens in lecture halls with inadequate TV monitors that miss the top and left edges of the screen.

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.

minus *num*
- *num* outputs the negative of its input. Minus sign means unary minus if it is immediately preceded by something requiring an input, or 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.

Note: Inside the computer numbers are represented in two different forms, one for integers and one for numbers with fractional parts. However, on most computers the largest number that can be represented in integer format is smaller than the largest integer that can be represented (even with exact precision) in floating-point (fraction) format. The `int` operation will always output a number whose value is mathematically an integer, but if its input is very large the output may not be in integer format. In that case, operations like `remainder` that require an integer input will not accept this number.

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 π.

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.

Random Numbers

random *num* outputs a random nonnegative integer less than its input, which must be an integer.

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.

Print Formatting

form *num width precision* outputs a word containing a printable representation of *num*, possibly preceded by spaces (and therefore not a number for purposes of performing

arithmetic operations), with at least *width* characters, including exactly *precision* digits after the decimal point. (If *precision* is 0 then there will be no decimal point in the output.)

As a debugging feature, `(form` *num* `-1` *format*`)` will print the floating point *num* according to the C printf *format*, to allow

```
to hex :num
op form :num -1 "|%08X %08X|
end
```

to allow finding out the exact result of floating point operations. The precise format needed may be machine-dependent.

Bitwise Operations

bitand *num1 num2*
(bitand *num1 num2 num3* **...)** outputs the bitwise and of its inputs, which must be integers.

bitor *num1 num2*
(bitor *num1 num2 num3* **...)** outputs the bitwise or of its inputs, which must be integers.

bitxor *num1 num2*
(bitxor *num1 num2 num3* **...)** outputs the bitwise exclusive-or of its inputs, which must be integers.

bitnot *num* outputs the bitwise not of its input, which must be an integer.

ashift *num1 num2* outputs *num1* arithmetic-shifted to the left by *num2* bits. If num2 is negative, the shift is to the right with sign extension. The inputs must be integers.

lshift *num1 num2* outputs *num1* logical-shifted to the left by *num2* bits. If num2 is negative, the shift is to the right with zero fill. The inputs must be integers.

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.)

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.)

not *tf* outputs `true` if the input is `false`, and vice versa.

Graphics

Berkeley Logo provides traditional Logo turtle graphics with one turtle. Multiple turtles, dynamic turtles, and collision detection are not supported. This is the most hardware-dependent part of Logo; some features may exist on some machines but not others. Nevertheless, the goal has been to make Logo programs as portable as possible, rather than to take fullest advantage of the capabilities of each machine. In particular, Logo attempts to scale the screen so that turtle coordinates `[-100 -100]` and `[100 100]` fit on the graphics window, and so that the aspect ratio is 1:1, although some PC screens have nonstandard aspect ratios.

The center of the graphics window (which may or may not be the entire screen, depending on the machine used) is turtle location `[0 0]`. Positive X is to the right; positive Y is up. Headings (angles) are measured in degrees clockwise from the positive Y axis. (This differs from the common mathematical convention of measuring angles counterclockwise from the positive X axis.) The turtle is represented as an isoceles triangle; the actual turtle position is at the midpoint of the base (the short side).

Colors are, of course, hardware-dependent. However, Logo provides partial hardware independence by interpreting color numbers 0 through 7 uniformly on all computers:

0	black	1	blue	2	green	3	cyan
4	red	5	magenta	6	yellow	7	white

Where possible, Logo provides additional user-settable colors; how many are available depends on the hardware and operating system environment. If at least 16 colors are available, Logo tries to provide uniform initial settings for the colors 8–15:

8	brown	9	tan	10	forest	11	aqua
12	salmon	13	purple	14	orange	15	grey

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 screen position. The argument is a list of two numbers, the X and Y coordinates.

setxy *xcor ycor* moves the turtle to an absolute screen position. The two arguments are numbers, the X and Y coordinates.

setx *xcor* moves the turtle horizontally from its old position to a new absolute horizontal coordinate. The argument is the new X coordinate.

sety *ycor* moves the turtle vertically from its old position to a new absolute vertical coordinate. The argument is the new Y coordinate.

home moves the turtle to the center of the screen. Equivalent to `setpos [0 0]`.

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

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 argument.

scrunch outputs a list containing two numbers, the X and Y scrunch factors, as used by `setscrunch`. (But note that `setscrunch` takes two numbers as inputs, not one list of numbers.)

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`.

fill fills in a region of the graphics window containing the turtle and bounded by lines that have been drawn earlier. This is not portable; it doesn't work for all machines, and may not work exactly the same way on different machines.

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

textscreen
ts rearranges the size and position of windows to maximize the space available in the text window (the window used for interaction with Logo). The details differ among machines. Compare `splitscreen` and `fullscreen`.

fullscreen
fs rearranges the size and position of windows to maximize the space available in the graphics window. The details differ among machines. Compare `splitscreen` and `textscreen`.

In the DOS version, switching from fullscreen to splitscreen loses the part of the picture that's hidden by the text window. Also, since there must be a text window to allow printing (including the printing of the Logo prompt), Logo automatically switches from fullscreen to splitscreen whenever anything is printed. [This design decision follows from the scarcity of memory, so that the extra memory to remember an invisible part of a drawing seems too expensive.]

splitscreen
ss rearranges the size and position of windows to allow some room for text interaction while also keeping most of the graphics window visible. The details differ among machines. Compare `textscreen` and `fullscreen`.

setscrunch *xscale yscale* adjusts the aspect ratio and scaling of the graphics display. After this command is used, all further turtle motion will be adjusted by multiplying the horizontal

and vertical extent of the motion by the two numbers given as inputs. For example, after the instruction `setscrunch 2 1` motion at a heading of 45 degrees will move twice as far horizontally as vertically. If your squares don't come out square, try this. (Alternatively, you can deliberately misadjust the aspect ratio to draw an ellipse.)

For Unix machines and Macintoshes, both scale factors are initially 1. For DOS machines, the scale factors are initially set according to what the hardware claims the aspect ratio is, but the hardware sometimes lies. The values set by `setscrunch` are remembered in a file (called `scrunch.dat`) and are automatically put into effect when a Logo session begins.

`refresh` tells Logo to remember the turtle's motions so that they can be reconstructed in case the graphics window is overlayed. The effectiveness of this command may depend on the machine used.

`norefresh` tells Logo not to remember the turtle's motions. This will make drawing faster, but prevents recovery if the window is overlayed.

Turtle and Window Queries

`shownp`
`shown?` outputs `true` if the turtle is shown (visible), `false` if the turtle is hidden. See `showturtle` and `hideturtle`.

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 hardware-dependent ways with use of color.)

setpencolor *colornumber*
setpc *colornumber* sets the pen color to the given number, which must be a nonnegative integer. Color 0 is always black; color 7 is always white. Other colors may or may not be consistent between machines.

setpalette *colornumber rgblist* sets the actual color corresponding to a given number, if allowed by the hardware and operating system. Colornumber must be an integer greater than or equal to 8. (Logo tries to keep the first 8 colors constant.) The second argument is a list of three nonnegative integers less than 64K (65536) specifying the amount of red, green, and blue in the desired color. The actual color resolution on any screen is probably less than 64K, but Logo scales as needed.

setpensize *size*
setpenpattern *pattern* set hardware-dependent pen characteristics. These commands are not guaranteed compatible between implementations on different machines.

setpen *list* **(**library procedure**)** sets the pen's position, mode, and hardware-dependent characteristics according to the information in the input list, which should be taken from an earlier invocation of `pen`.

setbackground *color*
setbg *color* set the screen background color.

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 by the hardware and operating system.

palette *colornumber* outputs a list of three integers, each in the range 0–65535, representing the amount of red, green, and blue in the color associated with the given number.

pensize
penpattern output hardware-specific pen information.

pen (library procedure**)** outputs a list containing the pen's position, mode, and hardware-specific characteristics, for use by `setpen`.

background
bg outputs the graphics background color.

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. 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, 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 input 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]
```

If this procedure is invoked by saying

```
proc "x
```

then `in1` has the value `x`, `in2` has the value `foo`, and `in3` has the value `[]` (the empty list). If it's invoked by saying

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

then `in1` has the value `a`, `in2` has the value `b`, and `in3` has the value `[c d]`.

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`. Definitions created by `copydef` are not saved by `save`; primitives are never saved, and user-defined procedures created by `copydef` are buried. (You are likely to be confused if you `po` or `pot` a procedure defined with `copydef` because its title line will contain the old name. This is why it's buried.)

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.

name *value varname* **(**library procedure**)** command. Same as `make` but with the inputs in reverse order.

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`.

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.

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. (However, Logo does not know about a library procedure until that procedure has been invoked.)

namep *name*
name? *name* outputs `true` if the input is the name of a variable.

Queries

contents outputs a *contents list,* i.e., a list of three lists containing names of defined procedures, variables, and property lists respectively. This list includes all unburied named items in the workspace.

buried outputs a contents list including all buried named items in the workspace.

procedures outputs a list of the names of all unburied user-defined procedures in the workspace. Note that this is a list of names, not a contents list. (However, procedures that require a contents list as input will accept this list.)

names outputs a contents list consisting of an empty list (indicating no procedure names) followed by a list of all unburied variable names in the workspace.

plists outputs a contents list consisting of two empty lists (indicating no procedures or variables) followed by a list of all unburied property lists in the workspace.

namelist *varname* **(**library procedure**)**
namelist *varnamelist* outputs a contents list consisting of an empty list followed by a list of the name or names given as input. This is useful in conjunction with workspace control procedures that require a contents list as input.

pllist *plname* **(**library procedure**)**
pllist *plnamelist* outputs a contents list consisting of two empty lists followed by a list of the name or names given as input. This is useful in conjunction with workspace control procedures that require a contents list as input.

Note: All procedures whose input is indicated as *contentslist* will accept a single word (taken as a procedure name), a list of words (taken as names of procedures), or a list of three lists as described under the `contents` command above.

Inspection

po *contentslist* command. Prints to the write stream the definitions of all procedures, variables, and property lists named in the input contents list.

poall (library procedure**)** command. Prints all unburied definitions in the workspace. Abbreviates `po contents`.

pops (library procedure**)** command. Prints the definitions of all unburied procedures in the workspace. Abbreviates `po procedures`.

pons (library procedure**)** command. Prints the definitions of all unburied variables in the workspace. Abbreviates `po names`.

popls (library procedure**)** command. Prints the contents of all unburied property lists in the workspace. Abbreviates `po plists`.

pon *varname* **(**library procedure**)**
pon *varnamelist* command. Prints the definitions of the named variable(s). Abbreviates the instruction `po namelist` *varname(list)*.

popl *plname* **(**library procedure**)**
popl *plnamelist* command. Prints the definitions of the named property list(s). Abbreviates the instruction `po pllist` *plname(list)*.

pot *contentslist* command. Prints the title lines of the named procedures and the definitions of the named variables and property lists. For property lists, the entire list is shown on one line instead of as a series of `pprop` instructions as in `po`.

pots (library procedure**)** command. Prints the title lines of all unburied procedures in the workspace. Abbreviates `pot procedures`.

erase *contentslist*
er *contentslist* command. Erases from the workspace the procedures, variables, and property lists named in the input. Primitive procedures may not be erased unless the variable `redefp` has the value `true`.

erall (library procedure**)** command. Erases all unburied procedures, variables, and property lists from the workspace. Abbreviates `erase contents`.

erps (library procedure**)** command. Erases all unburied procedures from the workspace. Abbreviates the instruction `erase procedures`.

erns (library procedure**)** command. Erases all unburied variables from the workspace. Abbreviates `erase names`.

erpls (library procedure**)** command. Erases all unburied property lists from the workspace. Abbreviates `erase plists`.

ern *varname* **(**library procedure**)**
ern *varnamelist* command. Erases from the workspace the variable(s) named in the input. Abbreviates `erase namelist` *varname(list)*.

erpl *plname* **(**library procedure**)**
erpl *plnamelist* command. Erases from the workspace the property list(s) named in the input. Abbreviates `erase pllist` *plname(list)*.

bury *contentslist* command. Buries the procedures, variables, and property lists named in the input. A buried item is not included in the lists output by `contents`, `procedures`, `variables`, and `plists`, but is included in the list output by `buried`. By implication, buried things are not printed by `poall` or saved by `save`.

buryall (library procedure**)** command. Abbreviates `bury contents`.

buryname *varname* **(**library procedure**)**
buryname *varnamelist* command. Abbreviates the instruction `bury namelist` *varname(list)*.

unbury *contentslist* command. Unburies the procedures, variables, and property lists named in the input. That is, the named items will be returned to view in `contents`, etc.

unburyall (library procedure**)** command. Abbreviates `unbury buried`.

unburyname *varname* **(**library procedure**)**
unburyname *varnamelist* command.
Abbreviates `unbury namelist` *varname(list)*.

trace *contentslist* command. Marks the named items for tracing. A message is printed whenever a traced procedure is invoked, giving the actual input values, and whenever a traced procedure `stop`s or `output`s. A message is printed whenever a new value is assigned to a traced

variable using `make`. A message is printed whenever a new property is given to a traced property list using `pprop`.

untrace *contentslist* command. Turns off tracing for the named items.

step *contentslist* command. Marks the named items for stepping. Whenever a stepped procedure is invoked, each instruction line in the procedure body is printed before being executed, and Logo waits for the user to type a newline at the terminal. A message is printed whenever a stepped variable name is *shadowed* because a local variable of the same name is created either as a procedure input or by the `local` command.

unstep *contentslist* command. Turns off stepping for the named items.

edit *contentslist*
ed *contentslist*
(edit)
(ed) command. Edits the definitions of the named items, using your favorite editor as determined by the `EDITOR` environment variable. If you don't have an `EDITOR` variable, edits the definitions using jove. If invoked without an argument, `edit` edits the same temporary file left over from a previous `edit` instruction. When you leave the editor, Logo reads the revised definitions and modifies the workspace accordingly.

Exceptionally, the `edit` command can be used without its default input and without parentheses provided that nothing follows it on the instruction line.

edall **(**library procedure**)** command. Abbreviates `edit contents`.

edps **(**library procedure**)** command. Abbreviates `edit procedures`.

edns **(**library procedure**)** command. Abbreviates `edit names`.

edpls **(**library procedure**)** command. Abbreviates `edit plists`.

edn *varname* **(**library procedure**)**
edn *varnamelist* command. Abbreviates `edit namelist` *varname(list)*.

edpl *plname* **(**library procedure**)**
edpl *plnamelist* command. Abbreviates `edit pllist` *plname(list)*.

save *filename* command. Saves the definitions of all unburied procedures, variables, and property lists in the named file. Equivalent to

```
to save :filename
local "oldwriter
make "oldwriter writer
openwrite :filename
setwrite :filename
poall
setwrite :oldwriter
close :filename
end
```

savel *contentslist filename* **(**library procedure**)** command. Saves the definitions of the procedures, variables, and property lists specified by *contentslist* to the file named *filename*.

load *filename* command. Reads instructions from the named file and executes them. The file can include procedure definitions with `to`, and these are accepted even if a procedure by the same name already exists. If the file assigns a list value to a variable named `startup`, then that list is run as an instructionlist after the file is loaded.

help *name*
(help) command. Prints information from the reference manual about the primitive procedure named by the input. With no input, lists all the primitives about which help is available. If there is an environment variable `LOGOHELP`, then its value is taken as the directory in which to look for help files, instead of the default help directory.

Exceptionally, the `help` command can be used without its default input and without parentheses provided that nothing follows it on the instruction line.

Control Structures

Note: in the following descriptions, an *instructionlist* can be a list or a word. In the latter case, the word is parsed into list form before it is run. Thus, `run readword` or `run readlist` will work. The former is slightly preferable because it allows for a continued line (with `~`) that includes a comment (with `;`) on the first line.

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.

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`* 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.

`catch` *`tag instructionlist`* command or operation. Runs its second input. Outputs if that instructionlist outputs. If, while running the instructionlist, a `throw` instruction is executed with a tag equal to the first input (case-insensitive comparison), then the running of the instructionlist is terminated immediately. In this case the `catch` outputs if a value input is given to `throw`. The tag must be a word.

If the tag is the word `error`, then any error condition that arises during the running of the instructionlist has the effect of `throw "error` instead of printing an error message and returning to toplevel. The `catch` does not output if an error is caught. Also, during the running of the instructionlist, the variable `erract` is temporarily unbound. (If there is an error while `erract` has a value, that value is taken as an instructionlist to be run after printing the error message. Typically the value of `erract`, if any, is the list `[pause]`.)

`throw` *`tag`*
`(throw` *`tag value`***`)`** command. Must be used within the scope of a `catch` with an equal tag. Ends the running of the instructionlist of the `catch`. If `throw` is used with only one input, the corresponding `catch` does not output a value. If `throw` is used with two inputs, the second provides an output for the `catch`.

`Throw "toplevel` can be used to terminate all running procedures and interactive pauses, and return to the toplevel instruction prompt. Typing the system interrupt character (normally control-C for Unix, control-Q for DOS, or command-period for Mac) has the same effect.

`Throw "error` can be used to generate an error condition. If the error is not caught, it prints a message (`throw "error`) with the usual indication of where the error (in this case the `throw`) occurred. If a second input is used along with a tag of `error`, that second input is used as the text of the error message instead of the standard message. Also, in this case, the location indicated for the error will be, not the location of the `throw`, but the location where the procedure containing the `throw` was invoked. This allows user-defined procedures to generate error messages as if they were primitives. Note: in this case the corresponding `catch "error`, if any, does not output, since the second input to `throw` is not considered a return value.

`Throw "system` immediately leaves Logo, returning to the operating system, without printing the usual parting message and without deleting any editor temporary file written by `edit`.

`error` outputs a list describing the error just caught, if any. If there was not an error caught since the last use of `error`, the empty list will be output. The error list contains four members: an integer code corresponding to the type of error, the text of the error message, the name of the procedure in which the error occurred, and the instruction line on which the error occurred.

`pause` command or operation. Enters an interactive pause. The user is prompted for instructions, as at toplevel, but with a prompt that includes the name of the procedure in which `pause` was invoked. Local variables of that procedure are available during the pause. `Pause` outputs if the pause is ended by a `continue` with an input.

If the variable `erract` exists, and an error condition occurs, the contents of that variable are run as an instructionlist. Typically `erract` is given the value `[pause]` so that an interactive pause will be entered on the event of an error. This allows the user to check values of local variables at the time of the error.

Typing the system quit character (normally control-\ for Unix, control-W for DOS, or command-comma for Mac) will also enter a pause.

`continue` *`value`*
`co` *`value`*
`(continue)`
`(co)` command. Ends the current interactive pause, returning to the context of the `pause` invocation that began it. If `continue` is given an input, that value is used as the output from the `pause`. If not, the `pause` does not output.

Exceptionally, the `continue` command can be used without its default input and without parentheses provided that nothing follows it on the instruction line.

`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]
```

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
6.5
```

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`.

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. Three 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
```

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]`. 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`.

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]`. 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`.

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 *Computer Science Logo Style,* `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` 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.

Macros

.macro *procname :input1 :input2* **...** **(**special form**)**
.defmacro *procname text* command. A macro is a special kind of procedure whose output is evaluated as Logo instructions in the context of the macro's caller. `.Macro` is exactly like `to` except that the new procedure becomes a macro; `.defmacro` is exactly like `define` with the same exception.

Macros are useful for inventing new control structures comparable to `repeat`, `if`, and so on. Such control structures can almost, but not quite, be duplicated by ordinary Logo procedures. For example, here is an ordinary procedure version of `repeat`:

```
to my.repeat :num :instructions
if :num=0 [stop]
run :instructions
my.repeat :num-1 :instructions
end
```

This version works fine for most purposes, e.g.,

```
my.repeat 5 [print "hello]
```

But it doesn't work if the instructions to be carried out include `output`, `stop`, or `local`. For example, consider this procedure:

```
to example
print [Guess my secret word.  You get three guesses.]
```

```
repeat 3 [type "|?? |
          if readword = "secret [pr "Right! stop]]
print [Sorry, the word was "secret"!]
end
```

This procedure works as written, but if `my.repeat` is used instead of `repeat`, it won't work because the `stop` will stop `my.repeat` instead of stopping `example` as desired.

The solution is to make `my.repeat` a macro. Instead of actually carrying out the computation, a macro must return a list containing Logo instructions. The contents of that list are evaluated as if they appeared in place of the call to the macro. Here's a macro version of `repeat`:

```
.macro my.repeat :num :instructions
if :num=0 [output []]
output sentence :instructions ~
                (list "my.repeat :num-1 :instructions)
end
```

Every macro is an operation—it must always output something. Even in the base case, `my.repeat` outputs an empty instruction list. To show how `my.repeat` works, let's take the example

```
my.repeat 5 [print "hello]
```

For this example, `my.repeat` will output the instruction list

```
[print "hello my.repeat 4 [print "hello]]
```

Logo then executes these instructions in place of the original invocation of `my.repeat`; this prints `hello` once and invokes another repetition.

The technique just shown, although fairly easy to understand, has the defect of slowness because each repetition has to construct an instruction list for evaluation. Another approach is to make `my.repeat` a macro that works just like the non-macro version unless the instructions to be repeated include `output` or `stop`:

```
.macro my.repeat :num :instructions
catch "repeat.catchtag ~
      [op repeat.done runresult [repeat1 :num :instructions]]
op []
end

to repeat1 :num :instructions
if :num=0 [throw "repeat.catchtag]
run :instructions
.maybeoutput repeat1 :num-1 :instructions
end

to repeat.done :repeat.result
if emptyp :repeat.result [op [stop]]
op list "output quoted first :repeat.result
end
```

If the instructions do not include `stop` or `output`, then `repeat1` will reach its base case and invoke `throw`. As a result, my.repeat's last instruction line will output an empty list, so the second evaluation of the macro result will do nothing. But if a `stop` or `output` happens, then `repeat.done` will output a `stop` or `output` instruction that will be re-executed in the caller's context.

The macro-defining commands have names starting with a dot because macros are an advanced feature of Logo; it's easy to get in trouble by defining a macro that doesn't terminate, or by failing to construct the instruction list properly.

Lisp users should note that Logo macros are *not* special forms. That is, the inputs to the macro are evaluated normally, as they would be for any other Logo procedure. It's only the output from the macro that's handled unusually.

Here's another example:

```
.macro localmake :name :value
output (list "local
             word "" :name
             "apply
             ""make
             (list :name :value))
end
```

It's used this way:

```
to try
localmake "garply "hello
print :garply
end
```

`Localmake` outputs the list

```
[local "garply apply "make [garply hello]]
```

The reason for the use of `apply` is to avoid having to decide whether or not the second input to `make` requires a quotation mark before it. (In this case it would—`make "garply "hello`—but the quotation mark would be wrong if the value were a list.)

It's often convenient to use the ` function to construct the instruction list:

```
.macro localmake :name :value
op `[local ,[word "" :name] apply "make [,[:name] ,[:value]]]
end
```

On the other hand, ` is pretty slow, since it's tree recursive and written in Logo.

macrop *name*
macro? *name* outputs `true` if its input is the name of a macro.

macroexpand *expr* **(**library procedure**)** takes as its input a Logo expression that invokes a macro (that is, one that begins with the name of a macro) and outputs the the Logo expression into which the macro would translate the input expression.

```
.macro localmake :name :value
op `[local ,[word "" :name] apply "make [,[:name] ,[:value]]]
end

? show macroexpand [localmake "pi 3.14159]
[local "pi apply "make [pi 3.14159]]
```

Error Processing

If an error occurs, Logo takes the following steps. First, if there is an available variable named `erract`, Logo takes its value as an instructionlist and runs the instructions. The operation `error` may be used within the instructions (once) to examine the error condition. If the instructionlist invokes `pause`, the error message is printed before the pause happens. Certain errors are *recoverable;* for one of those errors, if the instructionlist outputs a value, that value is used in place of the expression that caused the error. (If `erract` invokes `pause` and the user then invokes `continue` with an input, that input becomes the output from `pause` and therefore the output from the `erract` instructionlist.)

It is possible for an `erract` instructionlist to produce an inappropriate value or no value where one is needed. As a result, the same error condition could recur forever because of this mechanism. To avoid that danger, if the same error condition occurs twice in a row from an `erract` instructionlist without user interaction, the message "Erract loop" is printed and control returns to toplevel. "Without user interaction" means that if `erract` invokes `pause` and the user provides an incorrect value, this loop prevention mechanism does not take effect and the user gets to try again.

During the running of the `erract` instructionlist, `erract` is locally unbound, so an error in the `erract` instructions themselves will not cause a loop. In particular, an error during a pause will not cause a pause-within-a-pause unless the user reassigns the value `[pause]` to `erract` during the pause. But such an error will not return to toplevel; it will remain within the original pause loop.

If there is no available `erract` value, Logo handles the error by generating an internal `throw "error`. (A user program can also generate an error condition deliberately by invoking `throw`.) If this throw is not caught by a `catch "error` in the user program, it is eventually caught either by the toplevel instruction loop or by a pause loop, which prints the error message. An invocation of `catch "error` in a user program locally unbinds `erract`, so the effect is that whichever of `erract` and `catch "error` is more local will take precedence.

If a floating point overflow occurs during an arithmetic operation, or a two-input mathematical function (like `power`) is invoked with an illegal combination of inputs, the "doesn't like" message refers to the second operand, but should be taken as meaning the combination.

Error Codes

Here are the numeric codes that appear as the first member of the list output by `error` when an error is caught, with the corresponding messages. Some messages may have two different codes

depending on whether or not the error is recoverable (that is, a substitute value can be provided through the `erract` mechanism) in the specific context. Some messages are warnings rather than errors; these will not be caught. Errors 0 and 32 are so bad that Logo exits immediately.

```
 0   Fatal internal error (can't be caught)
 1   Out of memory
 2   Stack overflow
 3   Turtle out of bounds
 4   proc doesn't like datum as input (not recoverable)
 5   proc didn't output to proc
 6   Not enough inputs to proc
 7   proc doesn't like datum as input (recoverable)
 8   Too much inside ()'s
 9   You don't say what to do with datum
10   ')' not found
11   var has no value
12   Unexpected ')'
13   I don't know how to proc (recoverable)
14   Can't find catch tag for throwtag
15   proc is already defined
16   Stopped
17   Already dribbling
18   File system error
19   Assuming you mean IFELSE, not IF (warning only)
20   var shadowed by local in procedure call (warning only)
21   Throw "Error
22   proc is a primitive
23   Can't use TO inside a procedure
24   I don't know how to proc (not recoverable)
25   IFTRUE/IFFALSE without TEST
26   Unexpected ']'
27   Unexpected '}'
28   Couldn't initialize graphics
29   Macro returned value instead of a list
30   You don't say what to do with value
31   Can only use STOP or OUTPUT inside a procedure
32   APPLY doesn't like badthing as input
33   END inside multi-line instruction
34   Really out of memory (can't be caught)
```

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 `caseignoredp`, which is `true` and buried.

caseignoredp If `true`, indicates that lower case and upper case letters should be considered equal by `equalp`, `beforep`, `memberp`, etc. Logo initially makes this variable `true`, and buries it.

erract An instructionlist that will be run in the event of an error. Typically has the value `[pause]` to allow interactive debugging.

loadnoisily If `true`, prints the names of procedures defined when loading from a file (including the temporary file made by `edit`).

printdepthlimit If a nonnegative integer, indicates the maximum depth of sublist structure that will be printed by `print`, etc.

printwidthlimit If a nonnegative integer, indicates the maximum number of members in any one list that will be printed by `print`, etc.

redefp If `true`, allows primitives to be erased (`erase`) or redefined (`copydef`).

startup If assigned a list value in a file loaded by `load`, that value is run as an instructionlist after the loading.

Index of Defined Procedures

This index lists example procedures whose definitions are in the text. The general index lists technical terms and primitive procedures.

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

W

X

Y

Z

General Index

This index lists technical terms and primitive procedures. There is also an index of defined procedures, which lists procedures whose definitions are in the text.

C

D

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y