Logo (Berkeley) For Windows

You may also be interested in Reading Computer Science Logo Style, Volume 1: Intermediate Programming by Brian Harvey (MIT Press, 1985) for a tutorial on Logo programming with emphasis on symbolic computation.

INTRODUCTION

Why LOGO

This introduction does not do LOGO justice but it's a start. LOGO is a programming language, pure and simple. There are two models that languages come in, compiled and interpreted.

What is a compiled language?

In a compiled language the program is written and fed to a compiler. A compiler reads all your code and converts it to an executable form that your computer understands.

What is an interpreted language?

An interpreted language does not get compiled, instead, as each line is read the interpreter executes it. This is a slow process to execute (on the fly) like this, but has the advantage of not requiring a complete compile for each change. It's ideal in a learning environment.

So have guessed what type of language LOGO is yet?

Right, it's an interpreted language, at least this LOGO is anyway.

LOGO also has another unique feature not offered in many other languages (none that I know of). That is, what's called "Turtle Graphics".

What are turtle graphics?

Turtle graphics is a simple and powerful set of commands to manipulate a turtle.

Why do they call it a turtle?

The first version of LOGO used an electronic robot that resembled a turtle. In the case of a video screen (like this LOGO) it's simply a cursor (or pointer) of where the turtle is.

What does the turtle do?

It draws, lines mostly, on the screen.

The gap that turtle graphics fills is what traditional languages do not. That is, it gives immediate feedback. Immediate feedback makes it fun and easier to learn programming. The purpose of LOGO is to teach young and old how to program. It was modeled after a

very popular and power language called LISP. It is as powerful as any other programming language.

Where to Start

Novices can start in LOGO without having to program at all by just learning how to command the turtle. Learning turtle graphics will teach the user about geometry (and they won't even know it). It's amazing how soon you can introduce the concept of programming once they grasp the turtle concept. Lets look at some simple examples:

Draw a square using the turtle

```
FORWARD 100
RIGHT 90
FORWARD 100
RIGHT 90
FORWARD 100
RIGHT 90
FORWARD 100
RIGHT 90
```

That was easy but too much typing, lets try again.

```
REPEAT 4 [FD 100 RT 90]
```

That's it? Yes, that's the same square. We did two things. We noticed too much redundant code in our first example, so we asked logo to repeat the sequence 4 times. We also used abbreviated forms of the same commands. But we can still do better. A square is a popular item wouldn't you say? Wouldn't it be more useful just to say square when you wanted a square.

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

SQUARE
SQUARE
```

What's the TO and END for? It's to define a procedure (a small program) for the square. The TO can be thought of as to do something, the END terminates the TO. Once square was "defined" we called it twice. That's all you need to get a square now, just type square. There is a problem, however. It only draws squares of 100 by 100. Wouldn't it be better to draw any size square? It sure would and it's easy.

```
EDIT "square
```

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

SQUARE 100
SQUARE 200
```

Note all we did is replace 100 with a variable name called :length. Now when we call square we must specify how big we want. Above we asked logo to draw one square at 100x100 and another at 200x200. Note the ":" in front of the word length tells logo that length is a variable. However, we can still even do better. What's wrong now, you ask. Well wouldn't it be better if we could draw something other than a square like a triangle?

```
TO TRIANGLE :length
REPEAT 3 [FD :length RT 120]
END
TO SQUARE :length
REPEAT 4 [FD :length RT 90]
END
TO HEXAGON :length
REPEAT 5 [FD :length RT 72]
END

TRIANGLE 100
SQUARE 100
HEXAGON 100
```

Lot of typing (programmers hate to type). Why? Because there are more things to break and when a change needs to be made it might have to made in many places. Smaller is not always better but it usually helps. Lets try again.

```
TO POLYGON :length :sides
REPEAT :sides [FD :length RT 360.0/:sides]
END

POLYGON 100 3
POLYGON 100 4
POLYGON 100 5
```

What happened to TRIANGLE, SQUARE and HEXAGON? POLYGON now acts as every equal-sided polygon possible and with only one line of code! We now repeat the sequence based on how many :sides the caller asked for and we turn (RT) the amount of degrees appropriate for that shape. You may not realize it but this is PROGRAMMING.

I could go on for ever, yes for ever, even to the point of writing LOGO within LOGO.

This should get you started. What ever do make sure you have FUN.

EDITOR

When you exit the editor, Logo loads the revised definitions and modifies the workspace accordingly. Multiple Edit sessions are supported. But be careful of having multiple edits going that include the same definition. The last Editor Exited (and saved) is what takes precedence. Also realize if you delete procedures, property lists or names while in the Editor they will be ERASED from the environment at the time of exiting the editor (this is NEW behavior in MswLogo 3.4).

Edit Errors

If an error occurs when Logo "loads" in your edit you will be prompted to reenter the Editor. This situation commonly occurs when a continuation "~" is missing within a list.

Editing with Clipboard

Logo's Editor supports the Clipboard. The Clipboard is where most Windows application store data during cut and paste operations. This means that when you cut text from an application, such as Notepad, it can be pasted into Logo's Editor (the reverse is also true). Even Windows-Help supports the Clipboard. This means you can copy examples in this document directly to the editor (see HELP command).

The Input Box also supports the Clipboard. This means you can test code that you're not sure of or copy code already executed to the Editor. Currently only one line of text is supported between the Input Box and the Clipboard. Note that the Input Box does not have an Edit Menu like the Editor. You must use the "Short-Cut" keys for the desired actions. See the Edit Menu in the Editor for their definitions.

Context Sensitive Help

Logo's Editor also features context sensitive Help. If you select a keyword (such as FORWARD) in the Editor (double-click works best) you can ask the Editor to look up the keyword without going through the Help Menu followed by a Search and so on. You simple ask by clicking the right button on the mouse (abbreviations are also supported).

Testing (executing from Editor)

You can also Test your code by selecting a section of code (with the Mouse) and clicking on Test! in the menu. This will take each line and send it to the commander for execution.

COMMANDER

The commander is where you will spend most of your LOGO session. It is the means by which you instruct LOGO to do what you want. The most important control (box) within the commander is the INPUT BOX. It is located in the bottom left portion of the commander window. For information on the different controls (boxes) see the specific box below.

Input Box

The input box is tied to the Output/Command-Recall List Box and to the Execute Button. The input box can be "filled" with anything from the list box. You can also edit the text in the input box. If what your typing doesn't fit, it will scroll automatically. Once your command is in the box you need to execute it. You can do this by hitting ENTER or clicking on the execute button. Using the Up/Down arrow keys will automatically jump to (set focus to) the output/command-recall list box for the desired selection.

Output/Command-Recall List Box

The output/command-recall list box will record all output including what you type into the Input Box. You can select text by clicking on the desired text, by typing the beginning of the desired string, or by using the arrow keys. If something went out of view, use the scroll bar. Once selected it is automatically copied to the Input Box. A double-click on the mouse will automatically execute what your pointing at. Left/Right arrow keys will automatically jump to (set focus to) the Input Box for editing.

Execute Button

The execute button executes what is in the Input Box and is also "PUSHED" when you hit ENTER key.

Status Button

This button pops up a status window telling you what LOGO is up to. Click it again to close the window. See also the Status and NoStatus commands.

Trace Button

The trace button turns on tracing for debugging your programs. Click again to disable

tracing. You can turn tracing on or off even while Logo it running. Note that the trace button works independent of what you are tracing with the trace command. See also Trace and Untrace commands.

Halt Button

The halt button immediately stops LOGO from processing any further. Logo is now waiting for a new command. See also the Halt command.

Reset Button

The reset button is like a ClearScreen command and resets LOGO.

Yield Button

The yield button asks LOGO to not let other programs use the computer while LOGO is working. The default is Yield. Note that the commander window itself is like another program. That is, if you hit the NoYield button while processing, the commander will lose control (That means Halt Button won't work) until LOGO is idle again. Once Idle You can click the yield button again to enable yielding. The reason the yield button is here is that LOGO runs faster if it can keep the computer all to itself. See also Yield and NoYield commands.

You can achieve the best of both worlds (performance and yielding) by doing (and understanding) the following. Let's say that you have some large multi-nested loop in your code. The most inner loop is where most of the work is done. However, there is no need to YIELD all through the inner loop.

We have 3 cases:

Case 1 (User in control for 10,000 operations, lower performance):

```
yield
repeat 100 ~
  [~
  repeat 100 ~
    [~
    (work to be done)~
    ]~
  ]
```

Case 2 (User out of control for 10,000 operations, good performance):

```
noyield
```

```
repeat 100 ~
  [~
  repeat 100 ~
    [~
    (work to be done)~
    ]~
  ]
```

Case 3 (User out of control for 100 operations, still good performance):

```
repeat 100~
  [~
  noyield~
  repeat 100 ~
    [~
    (work to be done)~
    ]~
  yield~
  ]
```

Pause Button

The pause button stops LOGO so that you can examine variables, make changes or whatever. Once paused the pause button will show what depth you paused to. To continue you must issue a Continue command. You can also issue a Pause command within code to act as a "Break Point". You can think of pause and continue as sort of a Halt-n-Push and Pop-n-Continue of your state respectively.

MENU

The MENU is where you do high level tasks in LOGO, such as loading a LOGO program, loading an BITMAP image, setting up a printer or perhaps even read this help file. See the specific Menu item for more information.

File Menu

The File menu includes commands that enable you to operate on logo procedure files. Note that, as a side effect, any selection in this menu that has a directory in its dialog box will effectively change to that directory as the current working directory. For more information, select the File menu command name.

File New Command

This will clear (DELETE, PURGE, ZAP) all procedures currently loaded in memory. It's like starting with a "New" session.

File Load Command

This allows you to load in procedures from disk into memory so that they can be executed or edited. See also Load command.

File Save Command

This allows you to save everything that is loaded in memory onto the disk. See also Save command.

File Save As Command

This is the same as File Save Command, but prompts your for a new file name.

File Edit Command

This is how you edit procedures that have already been loaded (or developed) in memory. You will be prompted with all existing procedures (currently loaded within

memory) and you can also enter a new one. See also Edit command.

File Erase Command

This is how you erase procedures that have already been loaded (or developed) in memory. You will be prompted with all existing procedures (currently loaded within memory). See also Erase command.

File Exit Command

This is how you exit MswLogo. Also see the Bye command.

Bitmap Menu

The Bitmap menu includes commands that enable you to operate on bitmap files. For more information, see the specific Bitmap menu command name.

Bitmap New Command

This will clear the work done on the screen and create a new environment to save things in.

Bitmap Load Command

This allows you to read in an image you already saved in the past. The format of the file you save things in, is known as, a Microsoft Windows Bitmap (.BMP). You can interchange these files with other applications such as Paint. Note that these files can be BIG and can take a while to read or write.

See also the section on USING COLOR.

Bitmap Save Command

This allows you to save a PICTURE (bitmap image) of your work on the computer's disk so that the computer will not forget it. It also allows you to add more work to an existing piece of work. REMEMBER if your image was generated with a LOGO program you really don't need to save it as an image unless you want to use the image in another application such as Paint or as a Wallpaper.

The format of the file you save things in, is known as, a Microsoft Windows Bitmap (.BMP). You can interchange these files with other applications such as Paint. Note that these files can be BIG and can take a while to read or write.

See also the Bitmap Active Area Command.

Bitmap Save As Command

This is the same as Bitmap Save Command, but prompts your for a new file name.

Bitmap Active Area Command

This allows you to select the work area to be printed or saved. The Active Area dialog box gives you 2 choices. Full image, which is the default, or Custom image. Full image prints or saves the entire bitmap. Custom image initializes the Extents (XLow, XHigh, YLow, YHigh) also to the full image. However, the extents are now enabled and can be adjusted to your needs. The primary purpose of this option is performance and space. You no longer need to wait for the software to build a full image. It takes less time and less memory to print and disk space to save a partial image. As a side effect you can adjust where your image ends up on the page by selecting different extents.

Bitmap Print Setup Command

This allows you to setup your printer before you print.

Bitmap Print Command

This allows you to print your work on the printer.

Set Menu

The Set menu allows you to SET some of the characteristics of how LOGO behaves when drawing.

Set Font Command

This command allows you to select the font in which the command LABEL will draw

with. You can also select the font with the Logo command SETTEXTFONT and obtain it with the Logo command TEXTFONT.

Note: This command is only available to Windows 3.1 systems.

Help Menu

The Help menu allows you to learn more about LOGO. For more information, select the Help menu command name. Also see the Help command

Help Index Command

This command puts you in Microsoft Windows Help for LOGO.

Help MCI Command

This puts you into the MCI help file. It explains the syntax of the arguments to the Mci command.

Help Using Help Command

This will explain how to use Microsoft Windows Help.

Help About Command

This gives some details about the LOGO program like its version.

ENTERING AND LEAVING LOGO

To start MswLogo, just type the command "win logo" to DOS or click the Logo icon if already in Windows. To leave Logo, enter the command Bye or File Exit Command.

Logo allows you to load one or more filenames on the command line when starting Logo. The switch to perform this function is "-l file1 file2 ..." (-l stands for Load). These files will be loaded before the interpreter starts reading commands from the commander. 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 batch scripts from DOS or as new Icons from Windows. Note, the "-l" switch must follow any other switches.

MswLogo also supports switches to initialize the size of the bitmap (graphical workspace) to use. This does NOT select the window size, the Window is a Window INTO the bitmap. The switches are "-h number" (to select height) and "-w number" (to select width). The default is 1000x1000. This feature allows you to trade-off time versus space.

For example, to change to a 500x500 image (using 1/4 the amount of memory) and Autoload myprog.lg then enter:

c:\logo>win logo -h 500 -w 500 -l c:\logo\myprog.lg (Entered at DOS prompt)
or
c:\logo\logo -h 500 -w 500 -l c:\logo\myprog.lg (Entered in property dialogbox for MswLogo Icon)

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.

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., ?3), when runparsed (i.e., where a procedure name is expected), is treated as if it were the sequence

```
( ? 3 )
```

making the number an input to the ? procedure. (See TEMPLATE-BASED ITERATION) 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.

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.

Example:

```
print "abc;comment ~
def
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. If a tilde is typed at the terminal for line continuation, Logo will issue a tilde as a prompt character for the continuation line.

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.

In MswLogo there is no "prompt character" nor is <CR> (carriage return) passed from the input control box of the commander. However, MswLogo has added the "\n" control character which will translate to a <CR>.

Example:

print "Hello\nhow\nare\nyou
Hello
how
are
you

This will work in a procedure or from the input control box.

A character entered with backslash is EQUALP to the same character without the backslash, but can be distinguished by the BACKSLASHEDP predicate. (In Europe, backslashing is effective 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. The same is true of a character typed after a backslash, except that 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

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

Outputs a word formed by concatenating its inputs.

Example:

show word "o "k
ok
show (word "a "o "k "to)
aokto

LIST

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

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

Example:

show (list "This "is "a "List)
[This is a List]
show list [1 2 3] [a b c]
[[1 2 3] [a b c]]

SENTENCE

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.

Example:

show (se "A "Sentence "is "simply "a "list "of "words)
[A Sentence is simply a list of words]

FPUT

FPUT thing list

Outputs a list equal to its second input with one extra member, the first input, at the beginning.

Example:

show fput 1 [2 3 4]
[1 2 3 4]

LPUT

LPUT thing list

Outputs a list equal to its second input with one extra member, the first input, at the end.

Example:

show lput 5 [1 2 3 4]
[1 2 3 4 5]

ARRAY

ARRAY size
(ARRAY size origin)

Outputs an array of "size" elements (must be a positive integer), each of which initially is an empty list. Array elements can be selected with ITEM and changed with SETITEM. The first element of the array is element number 1, unless an "origin" input (must be an integer) is given, in which case the first element 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.

Example:

```
make "myarray (array 3 0)
setitem 2 :myarray 1
setitem 1 :myarray 2
setitem 0 :myarray 3
show :myarray
{3 2 1}
```

MDARRAY

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.

Example:

```
make "myarray (mdarray [2 3] 0)
mdsetitem [0 0] :myarray 1
mdsetitem [0 1] :myarray 2
mdsetitem [0 2] :myarray 3
mdsetitem [1 0] :myarray 4
mdsetitem [1 1] :myarray 5
mdsetitem [1 2] :myarray 6
show :myarray
{{1 2 3} {4 5 6}}
```

LISTTOARRAY

LISTTOARRAY list (library procedure)
(LISTTOARRAY list origin)

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

Example:

```
show listtoarray [1 2 3]
```

{1 2 3}

ARRAYTOLIST

ARRAYTOLIST array (library procedure)

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

Example:

show arraytolist {1 2 3}
[1 2 3]

COMBINE

COMBINE thing1 thing2 (library procedure)

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

Example:

show combine "a "b
ab
show combine "a [b]
[a b]

REVERSE

REVERSE list (library procedure)

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

Example:

show reverse [1 2 3]
[3 2 1]

GENSYM

GENSYM (library procedure)

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

Example:

make gensym 1
show :g1
1
make gensym "Hello
show :g2
Hello

SELECTORS

FIRST

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 element of the array).

Example:

print first [1 2 3]
1
print first "Hello
H

FIRSTS

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 to speed up the iteration tools MAP, MAP.SE, and FOREACH.

Example:

```
show firsts [[1 2 3] [a b c]]
[1 a]
```

Example:

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

LAST

LAST word-or-list

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.

Example:

```
print last [1 2 3]
3
print last "Hello
o
```

BUTFIRST

BUTFIRST word-or-list
BF word-or-list

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.

Example:

```
show butfirst [1 2 3]
```

[2 3]
show butfirst "Hello
ello

BUTFIRSTS

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 to speed up the iteration tools MAP, MAP.SE, and FOREACH.

Example:

show butfirsts [[1 2 3] [a b c]]
[[2 3] [b c]]

BUTLAST

BUTLAST word-or-list
BL word-or-list

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.

Example:

show butlast [1 2 3]
[1 2]
show butlast "Hello
Hell

ITEM

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 element of the array. "Index" starts at 1 for words and lists; the starting index of an array is specified when the array is created.

Example:

show item 2 [a b c]
b
show item 3 "ABC
c

MDITEM

MDITEM indexlist array (library procedure)

Outputs the element of the multidimensional "array" selected by the list of numbers "indexlist".

Example:

show mditem [2 2] {{0 1} {2 3}}
3

PICK

PICK list (library procedure)

Outputs a randomly chosen member of the input list.

Example:

show pick [1 2 3]
2
show pick [1 2 3]
2
show pick [1 2 3]
3

REMOVE

REMOVE thing list (library procedure)

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

Example:

show remove "b [a b c b]
[a c]

REMDUP

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.

Example:

show remdup [a b c b]
[a c b]

QUOTED

QUOTED thing (library procedure)

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

Example:

show "Hello
Hello
show quoted "Hello
"Hello

MUTATORS

SETITEM

SETITEM index array value

Command that replaces the "index"th element 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".

Example:

```
make "myarray (array 3 0)
setitem 2 :myarray 1
setitem 1 :myarray 2
setitem 0 :myarray 3
show :myarray
{3 2 1}
```

MDSETITEM

MDSETITEM indexlist array value (library procedure)

Command that replaces the element of "array" chosen by "indexlist" with the new "value".

Example:

```
make "myarray (mdarray [2 3] 0)
mdsetitem [0 0] :myarray 1
mdsetitem [0 1] :myarray 2
mdsetitem [0 2] :myarray 3
mdsetitem [1 0] :myarray 4
mdsetitem [1 1] :myarray 5
mdsetitem [1 2] :myarray 6
show :myarray
{{1 2 3} {4 5 6}}
```

.SETFIRST

.SETFIRST list value

Command that 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 permanent loss of memory if a circular

structure is released.

Example:

```
make "mylist [1 2 3]
.setfirst :mylist 0
show :mylist
[0 2 3]
```

.SETBF

.SETBF list value

Command that 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 core dumps if the butfirst of a list is not itself a list; and the permanent loss of memory if a circular structure is released.

Example:

```
make "mylist [1 2 3]
.setbf :mylist [a b]
show :mylist
[0 a b]
```

.SETITEM

.SETITEM index array value

Command that changes the "index"th element 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 permanent loss of memory if a circular structure is released.

Example:

```
make "myarray (array 3 0)
.setitem 2 :myarray 1
.setitem 1 :myarray 2
```

.setitem 0 :myarray 3
show :myarray
{3 2 1}

PUSH

PUSH stackname thing (library procedure)

Command that 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. Later, "thing" can be POPed off the "stackname".

Example:

make "mystack []
push "mystack 1
push "mystack 2
show :mystack
[2 1]
show pop "mystack
2
show pop "mystack
1

POP

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.

Example:

make "mystack []
push "mystack 1
push "mystack 2
show :mystack
[2 1]
show pop "mystack
2
show pop "mystack
1

QUEUE

QUEUE queuename thing (library procedure)

Command that 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. Later "thing" can be DEQUEUEed from the "queuename".

Example:

```
make "myqueue []
queue "myqueue 1
queue "myqueue 2
show :myqueue
[1 2]
show dequeue "myqueue
1
show dequeue "myqueue
2
```

DEQUEUE

DEQUEUE queuename (library procedure)

Outputs the least recently (oldest) QUEUEd member of the queue that is the value of the variable whose name is "queuename" and removes that member from the queue.

Example:

```
make "myqueue []
queue "myqueue 1
queue "myqueue 2
show :myqueue
[1 2]
show dequeue "myqueue
1
show dequeue "myqueue
2
```

PREDICATES (Data)

WORDP

WORDP thing

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

Example:

show wordp "Hello
true
show wordp [Hello]
false
show wordp {Hello}
false

LISTP

LISTP thing

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

Example:

show listp "Hello
false
show listp [Hello]
true
show listp {Hello}
false

ARRAYP

ARRAYP thing

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

Example:

show arrayp "Hello

false
show arrayp [Hello]
false
show arrayp {Hello}
true

EMPTYP

EMPTYP thing

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

Example:

show emptyp [1 2 3]
false
show emptyp []
true

EQUALP

EQUALP 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 elements 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.)

Example:

show equalp 1 1
true
show equalp 1 2
false
show equalp [1 2 3] [1 2 3]
true
show equalp [1 2 3] [3 2 1]

false

BEFOREP

BEFOREP 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 before 1.

Example:

show beforep "ABC "abd
true
show beforep "abd "ABC
false

.EQ

.EQ thing1 thing2

Outputs TRUE if its two inputs are the same object, 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.

Example:

make "x 1
make "y 1
show .eq :x :y
false
show .eq :x :x
true

MEMBERP

MEMBERP thing1 thing2

If "thing2" is a list or an array, outputs TRUE if "thing1" is EQUALP to a member or

element of "thing2", FALSE otherwise. If "thing2" is a word, outputs TRUE if "thing1" is EQUALP to a substring of "thing2", FALSE otherwise. Note that this behavior for words is different from other dialects, in which "thing1" must be a single character to make MEMBERP true with "thing2" a word.

Example:

show memberp 1 [1 2 3]
true
show memberp 4 [1 2 3]
false

NUMBERP

NUMBERP thing

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

Example:

show numberp 1
true
show numberp [1]
false

BACKSLASHEDP

BACKSLASHEDP char

Outputs TRUE if the input character was originally entered into Logo with a backslash (\) before it to prevent special syntactic meaning, FALSE otherwise. (In Europe, outputs TRUE only if the character is a backslashed space, tab, newline, or one of ()[]+-*/=<>":;\ ~?)

Example:

show backslashedp "a
false

QUERIES

COUNT

COUNT thing

Outputs the number of characters in the input, if the input is a word; outputs the number of members or elements 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 element, depending on the array's origin.)

Example:

count [1 2 3]
3
count "ab
2

ASCII

ASCII char

Outputs the integer (in the United States, between 0 and 127) that represents the input character in the ASCII code.

Example:

show ascii "a
97
show ascii "A
65
show ascii "b
98

CHAR

CHAR int

Outputs the character represented in the ASCII code by the input, which must be an integer between 0 and 127.

Example:

show ascii 97
a

show ascii 65
A
show ascii 98
b

MEMBER

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.

Example:

show memberp "b [a b c d]
[b c d]
show memberp "c [a b c d]
[c d]

LOWERCASE

LOWERCASE word

Outputs a copy of the input word, but with all uppercase letters changed to the corresponding lowercase letter. (In the United States, letters that were initially read by Logo preceded by a backslash are immune to this conversion.)

Example:

show lowercase "Hello
hello

UPPERCASE

UPPERCASE word

Outputs a copy of the input word, but with all lowercase letters changed to the corresponding uppercase letter. (In the United States, letters that were initially read by Logo preceded by a backslash are immune to this conversion.)

Example:

show uppercase "Hello
HELLO

STANDOUT

STANDOUT thing

(Not supported in MswLogo yet)

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

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.

Example:

show parse "Hello
[Hello]

RUNPARSE

RUNPARSE word-or-list

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.

Example:

show runparse "a<b
[a < b]

TIME

TIME

Outputs the current time on the system as a list.

Example:

show time
[Wed Jul 14 23:34:08 1993]

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 elements or 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 elements will print as [... ...].

If there is a variable named PRINTWIDTHLIMIT with a nonnegative integer value, then only the first so many elements or members of any array or list will be printed. A single ellipsis replaces all missing objects 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 object but to any substructures within it.

PRINT

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

Command that 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.

Example:

print "Hello
Hello
print [Hello how are you]
Hello how are you

TYPE

TYPE thing
(TYPE thing1 thing2 ...)

Command that 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 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 waiting.

Example:

type "Hello
type "How
type "Are
print "You
HelloHowAreYou

SHOW

SHOW thing
(SHOW thing1 thing2 ...)

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

Example:

show [1 2 3]
[1 2 3]
print [1 2 3]
1 2 3

RECEIVERS

READLIST

READLIST

RL

Reads a line from the read stream (initially the terminal) and outputs that line as a list. The line is separated into elements 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 process's 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.

Example:

show readlist
<Enter (Hello how are you <CR>) in dialog box>
[Hello how are you]

READWORD

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 process's 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 has 128 added to its representation. Programs can use BACKSLASHEDP to check for this code. (In Europe, backslashedness is preserved only for certain characters. See BACKSLASHEDP.)

Example:

show readword
<Enter (Hello<CR>) in dialog box>
Hello

READCHAR

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 are invoked or a Logo prompt is printed. Backslash, vertical bar, and tilde characters have no special meaning in this context.

Example:

show readchar
<Enter (H<CR>) in dialog box>
H

READCHARS

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 are invoked or a Logo prompt is printed. Backslash, vertical bar, and tilde characters have no special meaning in this context.

Example:

Currently broken in MswLogo

SHELL

SHELL command
(SHELL command wordflag)

(not implemented in MswLogo yet)

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 was 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 uses "first first" to extract the first word of the first (and only) line of the shell output.

FILE ACCESS

OPENREAD

OPENREAD filename

Command that opens the named file for reading. The read position is initially at the beginning of the file.

Example:

```
openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
print [Good Bye]
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
repeat 2 [show readlist]
[Hello]
[Good Bye]
setread []
close "dummy.fil
```

OPENWRITE

OPENWRITE filename

Command that pens the named file for writing. If the file already existed, the old version is deleted and a new, empty file is created.

Example:

```
openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
print [Good Bye]
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
repeat 2 [show readlist]
[Hello]
[Good Bye]
setread []
close "dummy.fil
```

OPENAPPEND

OPENAPPEND filename

Command that 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.

Example:

```
openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
setwrite []
close "dummy.fil

openappend "dummy.fil
setwrite "dummy.fil
print [Good Bye]
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
repeat 2 [show readlist]
[Hello]
[Good Bye]
```

setread []
close "dummy.fil

OPENUPDATE

OPENUPDATE filename

Command that 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.

Example:

(This is the way it should work but does not seem to?)

openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
print [Good Bye]
setwrite []
close "dummy.fil

openupdate "dummy.fil
setread "dummy.fil
show readlist
[Hello]
setwrite "dummy.fil
setwritepos 6
print [And how are you today]
setwrite []
setread "dummy.fil
setreadpos 0
repeat 3 [show readlist]
[Hello]
[And how are you today]
[Good Bye]
close "dummy.fil

CLOSE

CLOSE filename

Command that closes the named file.

Example:

```
openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
print [Good Bye]
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
repeat 2 [show readlist]
[Hello]
[Good Bye]
setread []
close "dummy.fil
```

ALLOPEN

ALLOPEN

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

Example:

```
openwrite "dummy1.fil
openwrite "dummy2.fil
show allopen
[dummy1.fil dummy2.fil]
```

CLOSEALL

CLOSEALL (library procedure)

Command that closes all open files. Abbreviates FOREACH ALLOPEN [CLOSE ?]

Example:

openwrite "dummy1.fil
openwrite "dummy2.fil
show allopen
[dummy1.fil dummy2.fil]
closeall
show allopen
[]

ERASEFILE

ERASEFILE filename
ERF filename

Command that erases (deletes, removes) the named file, which should not currently be open.

Example:

openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
repeat 2 [show readlist]
[Hello]
setread []
close "dummy.fil

erasefile "dummy.fil
openread "dummy.fil
File system error: I can't open that file

DRIBBLE

DRIBBLE filename

Command that 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.

Example:

```
dribble "dummy.fil
fd 100
rt 90
nodribble
openread "dummy.fil
setread "dummy.fil
repeat 3 [show readlist]
[fd 100]
[rt 90]
[nodribble]
setread []
close "dummy.fil
```

NODRIBBLE

NODRIBBLE

Command that stops copying information into the dribble file, and closes the file.

Example:

```
dribble "dummy.fil
fd 100
rt 90
nodribble
openread "dummy.fil
setread "dummy.fil
repeat 3 [show readlist]
[fd 100]
[rt 90]
[nodribble]
setread []
close "dummy.fil
```

SETREAD

SETREAD filename

Command that 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.

Example:

```
openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
print [Good Bye]
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
repeat 2 [show readlist]
[Hello]
[Good Bye]
setread []
close "dummy.fil
```

SETWRITE

SETWRITE filename

Command that 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.

Example:

```
openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
print [Good Bye]
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
repeat 2 [show readlist]
[Hello]
[Good Bye]
```

setread []
close "dummy.fil

READER

READER

Outputs the name of the current read stream file, or the empty list if the read stream is the terminal.

Example:

openread "dummy.fil
setread "dummy.fil
show reader
dummy.fil

WRITER

WRITER

Outputs the name of the current write stream file, or the empty list if the write stream is the terminal.

Example:

openwrite "dummy.fil
setwrite "dummy.fil
show writer
dummy.fil

SETREADPOS

SETREADPOS charpos

Command that 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.

Example:

```
openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
print [Good Bye]
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
show readlist
[Hello]
setreadpos 0
show readlist
[Hello]
setread []
close "dummy.fil
```

SETWRITEPOS

SETWRITEPOS charpos

Command that 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.

Example:

```
openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
setwritepos 0
type "J
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
show readlist
[Jello]
setread []
close "dummy.fil
```

READPOS

READPOS

Outputs the file position of the current read stream file.

Example:

```
openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
print [Good Bye]
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
repeat 2 [show readpos show readlist]
0
[Hello]
7
[Good Bye]
setread []
close "dummy.fil
```

WRITEPOS

WRITEPOS

Outputs the file position of the current write stream file.

Example:

Note: the output had to be put in a list until the writer is returned to screen ([]).

```
openwrite "dummy.fil
setwrite "dummy.fil
make "history []
make "history lput writepos :history
print "Hello
make "history lput writepos :history
print [Good Bye]
make "history lput writepos :history
setwrite []
```

close "dummy.fil
show :history
[0 7 17]

EOFP

EOFP

Predicate that outputs TRUE if there are no more characters to be read in the read stream file, FALSE otherwise.

Example:

openwrite "dummy.fil
setwrite "dummy.fil
print "Hello
print [Good Bye]
setwrite []
close "dummy.fil

openread "dummy.fil
setread "dummy.fil
repeat 2 [show readlist show eofp]
[Hello]
false
[Good Bye]
true
setread []
close "dummy.fil

Serial and Parallel Port communication

PORTOPEN

PORTOPEN port

This command is used to gain access to the serial and parallel ports of your computer. Once the desired port is open you can read (PORTREADCHAR or PORTREADARRAY) or write (PORTWRITECHAR or PORTWRITEARRAY) to it. You can set the characteristics of the port with PORTMODE. Only one port can be open at any given time. Once finished with the port you should close the port with PORTCLOSE.

port:(WORD) Is the name of the port wish to open (e.g. COM1-COM4 and LPT1-LPT3)

Example:

```
portopen "com1
portclose
```

PORTCLOSE

PORTCLOSE

This command closes a port that was opened by PORTOPEN.

Example:

```
portopen "com1
portclose
```

PORTFLUSH

PORTFLUSH queue

This command is used to flush the ports input or output queue.

queue:(INTEGER) Specifies which queue you want flushed 0 (output) and 1 (input).

Example:

```
portflush 1
```

PORTMODE

PORTMODE mode

This command is used to set the mode (speed, parity, data bits, and stop bits) of the port.

Note that the characteristics (such as speed and flow control) can also be set through the Control Panel PORTS icon. Which is the only way you can specify control flow.

mode:(WORD) Is mode you wish to set to ("COMn:SPEED,PARITY,DATA,STOP). Same format as the DOS MODE command.

Example:

portmode "com1:9600,n,8,1

PORTREADARRAY

PORTREADARRAY count buffer

This command will read the currently open port and write the data into the given buffer array. It will attempt to read "count" many characters from the port if they are available. It will output the actual number of bytes read.

count:(INTEGER) Is the number of characters to read from the port. You can use a larger number than the array size if you just want to fill the array.

buffer:(BUFFER) Is an ARRAY buffer to which input data is written to. It will be filled with byte size integers.

output:(INTEGER) Is the actual number of bytes read off the port.

Example:

portopen "com1
print se [Sending...] portwritearray 3 listtoarray map [ascii ?] arraytolist listtoarray "at
Sending... 2
make "y portwritechar 13
wait 60
make "buff {0 0 0 0 0 0 0 0 0 0}
print se [Receiving...] portreadarray 10 :buff
Receiving... 9
print se [Data Rx...] map [char ?] remove [] arraytolist :buff
Data Rx... at | |
O K |
portclose

PORTREADCHAR

PORTREADCHAR

This command will read one byte from the currently open port and output it as an integer. It will output "-1" if no character was available.

output:(INTEGER) Is the byte data read from the port (-1 if none available or error).

Example:

```
portopen "com1
make "y portwritechar ascii "a
make "y portwritechar ascii "t
make "y portwritechar 13
wait 60
print "Reading...
Reading...
repeat 10 [make "x portreadchar if not :x = -1 [type char :x]]
print ".
at||
OK|
.
portclose
```

PORTWRITEARRAY

PORTWRITEARRAY count buffer

This command will write to the currently open port with the data in the given buffer array. It will attempt to write "count" many characters to the port if possible. It will output the actual number of bytes written.

count:(INTEGER) Is the number of characters to write to the port. You can use a larger number than the array size if you just want to dump the whole array.

buffer:(BUFFER) Is an ARRAY buffer to which output data is read from. It must contain byte size integers.

output:(INTEGER) Is the actual number of bytes written to the port.

Example:

```
portopen "com1
print se [Sending...] portwritearray 3 listtoarray map [ascii ?] arraytolist listtoarray "at
Sending... 2
make "y portwritechar 13
wait 60
make "buff {0 0 0 0 0 0 0 0 0 0}
print se [Receiving...] portreadarray 10 :buff
Receiving... 9
print se [Data Rx...] map [char ?] remove [] arraytolist :buff
```

Data Rx... at | |
O K |
portclose

PORTWRITECHAR

PORTWRITECHAR data

This command will write one data byte to the currently open port and output the number of bytes written (0 or 1).

data:(INTEGER) Is the byte data that is to be written to the port.

output:(INTEGER) Is the number of bytes written (0 or 1).

Example:

portopen "com1
make "y portwritechar ascii "a
make "y portwritechar ascii "t
make "y portwritechar 13
wait 60
print "Reading...
Reading...
repeat 10 [make "x portreadchar if not :x = -1 [type char :x]]
print ".
at||
OK|
.
portclose

TERMINAL and MOUSE ACCESS

KEYP

KEYP

(not supported in MswLogo yet, see KEYBOARDON)

Predicate that 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 any pending characters when it switches modes, so the first KEYP invocation will always output FALSE.

KEYBOARDON

KEYBOARDON keydown
(KEYBOARDON keydown keyup)

This command will enable you to directly trap keyboard events. To obtain what key was involved call KEYBOARDVALUE in your keydown or keyup procedure. Note that the "Screen" window must have focus (NOT the commander) to catch the key events. You can force this by SETFOCUS [MSWLOGO SCREEN] when you issue this command. The second form of this command detects independently the keydown and keyup event. Note that all the "callbacks" for the keyboard are automatically run in a NOYIELD mode.

keydown:(LIST) Is a (short) list of logo commands (or a procedure name) to execute when the key is pushed.

keyup:(LIST) Is a (short) list of logo commands (or a procedure name) to execute when the key is let go.

Example:

```
keyboardon [print char keyboardvalue]
setfocus [MSWLOGO SCREEN]
<a>
a
<b>
b
keyboardoff
```

KEYBOARDOFF

KEYBOARDOFF

This command will disable trapping of keyboard events.

Example:

```
keyboardon [print char keyboardvalue]
```

setfocus [MSWLOGO SCREEN]
<a>
a
<b>
b
keyboardoff

KEYBOARDVALUE

KEYBOARDVALUE

This command will output the value of the last key pushed DOWN or let UP.

output: (INTEGER) Is the ASCII value of the last Keyhit event.

Example:

keyboardon [print char keyboardvalue]
setfocus [MSWLOGO SCREEN]
<a>
a
<b>
b
keyboardoff

Remember "MswLogo Screen" must have focus (be selected) when you hit the keys.

MOUSEON

MOUSEON leftbuttondown leftbuttonup rightbuttondown rightbuttonup move

This command will enable you to directly trap mouse events. To obtain where the mouse was when a button was pushed or the mouse moved call MOUSEPOS in your button or move procedure. Note that the "Screen" window must have focus (NOT the commander) to catch the mouse events. Note that all the "callbacks" for the mouse are automatically run in a NOYIELD mode.

leftbuttondown:(LIST) Is a (short) list of logo commands (or a procedure name) to execute when the Left Button is pushed DOWN.

leftbuttonup:(LIST) Is a (short) list of logo commands (or a procedure name) to execute when the Left Button is let UP.

rightbuttondown:(LIST) Is a (short) list of logo commands (or a procedure name) to execute when the Right Button is pushed DOWN.

rightbuttondown:(LIST) Is a (short) list of logo commands (or a procedure name) to execute when the Right Button is let UP.

move: (LIST) Is a (short) list of logo commands (or a procedure name) to execute when the mouse is moved.

Example:

```
pu
mouseon [setpos mousepos pd] [pu] [] [] [setpos mousepos]
<move mouse around hold button down to draw>
mouseoff
```

MOUSEOFF

MOUSEOFF

This command will disable trapping of mouse events.

Example:

```
pu
mouseon [setpos mousepos pd] [pu] [] [] [setpos mousepos]
<move mouse around hold button down to draw>
mouseoff
```

MOUSEPOS

MOUSEPOS

This command will output the position of the mouse at the last mouse event.

output: (LIST) Is the position ([x y]) of the last mouse event.

Example:

```
pu
mouseon [setpos mousepos pd] [pu] [] [] [setpos mousepos]
<move mouse around hold button down to draw>
mouseoff
```

CLEARTEXT

CLEARTEXT
CT

Command that clears the text screen of the terminal. In MswLogo it clears the output/command recall list box.

Example:

print "Hello
Hello
cleartext

SETCURSOR

(not supported in MswLogo yet)

SETCURSOR vector

Command where 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

CURSOR

(not supported in MswLogo yet)

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

SETMARGINS vector

(not supported in MswLogo yet)

Command where 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

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

Outputs the sum of its inputs.

Example:

show 2 + 3
5

DIFFERENCE

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 non space.

Example:

show 3 - 2
1

MINUS

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 non space. There is a difference in binding strength between the two forms:

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

Example:

show 2 - -3
5

PRODUCT

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

Outputs the product of its inputs.

Example:

show 2 * 3
6

QUOTIENT

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 4, not 4.0 -- it does the right thing.) With a single input, QUOTIENT outputs the reciprocal of the input.

Example:

show 6 / 3
2
show 6 / 2
1.5

REMAINDER

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 num2.

Example:

show 6 / 4
2
show 6 / 2
0

INT

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 requires an integer input will not accept this number.

Example:

show int 8.2
8
show int 8.7
8

ROUND

ROUND num

Outputs the nearest integer to the input.

Example:

show round 8.2
8
show round 8.7
9

SQRT

SQRT num

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

Example:

show sqrt 4
2
show sqrt 9
3

POWER

POWER num1 num2

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

Example:

show power 2 3
8

EXP

EXP num

Outputs e (2.718281828+) to the power of "num".

Example:

show exp 2
7.38905609893065

LOG10

LOG10 num

Outputs the common logarithm of "num". That is, 10 raised to ? = num, where ? will be the Output.

Example:

log10 1
0
log10 10
1
log10 100
2

LN

LN num

Outputs the natural logarithm of the "num". That is, e (2.718281828+) raised to ? = num, where ? will be the Output.

Example:

show ln 1
0
show ln exp 1
1

SIN

SIN degrees

Outputs the sine of "degrees", which is taken in degrees.

Example:

show sin 0
0
show sin 90
1

show sin 180
0

RADSIN

RADSIN radians

Outputs the sine of "radians", which is taken in radians.

Example:

make "pi 4 * radarctan 1
show radsin 0
0
show radsin pi/2
1
show radsin pi
0

COS

COS degrees

Outputs the cosine of "degrees", which is taken in degrees.

Example:

show cos 0
1
show cos 90
0
show cos 180
-1

RADCOS

RADCOS radians

Outputs the cosine of "radians", which is taken in radians.

Example:

make "pi 4 * radarctan 1
show radcos 0
1
show radcos pi/2
0
show radcos pi
-1

ARCTAN

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 non zero, or 90 or -90 depending on the sign of y, if x is zero.

Example:

show arctan 1
45

RADARCTAN

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 non zero, or pi/2 or -pi/2 depending on the sign of y, if x is zero.

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

Example:

show radarctan 1
0.785398163397448

PREDICATES (Arithmetic)

LESSP

LESSP num1 num2

num1 < num2

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

Example:

show 1 < 2
true
show 2 < 1
false

GREATERP

GREATERP num1 num2
num1 > num2

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

Example:

show 2 > 1
true
show 1 > 2
false

RANDOM NUMBERS

RANDOM

RANDOM num

Outputs a random nonnegative integer less than its input, which must be an integer.

Example:

repeat 5 [show random 10]
6
8
3
0
9

RERANDOM

RERANDOM
(RERANDOM seed)

Command that 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.

Example:

```
rerandom 1234
repeat 2 [show random 10]
6
2
rerandom 1234
repeat 2 [show random 10]
6
2
```

PRINT FORMATTING

FORM

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.

Example:

show form 99.99 -1 "|%08X %08X|
0000C28F 000028F5
show form 123.1 10 10
123.1000000000

BITWISE OPERATIONS

BITAND

BITAND num1 num2
(BITAND num1 num2 num3 ...)

Outputs the bitwise AND of its inputs, which must be integers.

Example:

show bitand 5 2
0
show bitand 5 1
1

BITOR

BITOR num1 num2
(BITOR num1 num2 num3 ...)

Outputs the bitwise OR of its inputs, which must be integers.

Example:

show bitor 5 2
7
show bitor 5 1
5

BITXOR

BITXOR num1 num2

(BITXOR num1 num2 num3 ...)

Outputs the bitwise EXCLUSIVE OR of its inputs, which must be integers.

Example:

show bitxor 5 2
7
show bitxor 5 1
4

BITNOT

BITNOT num

Outputs the bitwise NOT of its input, which must be an integer.

Example:

show bitnot 1
-2
show bitnot 5
-6

ASHIFT

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.

Example:

show ashift 5 2
20
show ashift 20 -1
10

LSHIFT

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.

Example:

show lshift 5 2
20
show lshift 20 -1
10

LOGICAL OPERATIONS

AND

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.)

Example:

show and "true "false
false
show and "true "true
true

OR

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.)

Example:

show or "true "false
true
show or "false "false
false

NOT

NOT tf

Outputs TRUE if the input is FALSE, and vice versa.

Example:

not "true
false

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 isosceles triangle; the actual turtle position is at the midpoint of the base (the short side).

MswLogo does take advantage of the hardware and therefore is not completely portable with other ports of ucblogo (Berkeley Logo).

TURTLE MOTION

FORWARD

FORWARD dist
FD dist

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

Example:

repeat 4 [forward 100 rt 90]

BACK

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.)

Example:

repeat 4 [back 100 rt 90]

LEFT

LEFT degrees
LT degrees

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

Example:

repeat 3 [fd 100 left 120]

RIGHT

RIGHT degrees
RT degrees

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

Example:

repeat 3 [fd 100 right 120]

SETPOS

SETPOS pos

Moves the turtle to an absolute screen position. The argument is a list of two numbers, the X and Y coordinates.

Example:

cs
setpos [0 100]
setpos [100 100]

```
setpos [100 0]
setpos [0 0]
```

SETXY

SETXY xcor ycor

Moves the turtle to an absolute screen position. The two arguments are numbers, the X and Y coordinates.

Example:

```
cs
setpos 0 100
setpos 100 100
setpos 100 0
setpos 0 0
```

SETX

SETX xcor

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

Example:

```
cs
setx 100
sety 100
setx 0
sety 0
```

SETY

SETY ycor

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

Example:

```
cs
setx 100
sety 100
setx 0
sety 0
```

HOME

HOME

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

Example:

```
setxy 100 100
home
```

SETHEADING

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.

Example:

```
setheading 90
setheading 180
```

ARC

ARC angle radius

The turtle does not move in this command. It draws an arc (part of a circle) based on the turtle heading, turtle position and the given arguments. The arc starts at the rear of the turtle heading. The size is based on the radius. The current turtle position will be at the center of the arc. Arc will also follow wrap/fence/windows modes. ARC 360 radius will of course draw a circle.

Example:

arc 360 100
arc 90 50

TURTLE MOTION QUERIES

POS

POS

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

Example:

setpos [100 100]
show pos
[100 100]

XCOR

XCOR (library procedure)

Outputs a number, the turtle's X coordinate.

Example:

setx 100
show xcor
100

YCOR

YCOR (library procedure)

Outputs a number, the turtle's Y coordinate.

Example:

sety 100
show ycor
100

HEADING

HEADING

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

Example:

```
setheading 90
show heading
90
```

TOWARDS

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.

Example:

```
cs
show towards [100 100]
45
```

PIXEL

PIXEL

Outputs a list of numbers that represent the Red, Green, and Blue intensity of the pixel currently under the turtle.

Example:

```
cs
show pixel
[255 255 255]
fd 1
show pixel
[255 255 255]
bk 1
show pixel
```

[0 0 0]

SCRUNCH

SCRUNCH

(not supported completely in MswLogo yet)

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

SHOWTURTLE
ST

Makes the turtle visible.

Example:

clearscreen
hideturtle
showturtle

HIDETURTLE

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.

Example:

clearscreen
hideturtle
showturtle

CLEAN

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.

Example:

```
clearscreen
setxy 100 100
clean
```

CLEARSCREEN

CLEARSCREEN
CS

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

Example:

```
clearscreen
setxy 100 100
clearscreen
```

WRAP

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.

Example:

```
wrap
fd 950
```

WINDOW

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 off screen. 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.

Example:

```
window
fd 950
```

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.

Example:

```
fence
fd 950
turtle out of bounds
```

FILL

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.

Examples:

```
cs
repeat 4 [fd 100 rt 90]
rt 45
```

pu
fd 20
fill

LABEL

LABEL text

The input, which may be a word or a list, is printed on the screen. If the object is a list, any sub-lists are delimited by square brackets, but the entire object is not delimited by brackets. You can print any logo object (numbers, lists and strings). Note that the handle of the string (the origin) is the top-left corner of the string. Another thing to be aware of is that the capabilities of the text changes depending on the device (screen or printer), the size, the turtle heading (direction) and the font. It other words sometimes the text can be drawn at the turtle heading and sometimes it cannot. Sometimes what is on the screen will not be exactly what you print.

The color of the text is determined by SETPENCOLOR.
The position of the text is determined by the location of the turtle.
The font of the text is determined by SETTEXTFONT. or Set Font Command.
The angle of the text is determined by the heading (direction) of the turtle.

Example:

label "Hello

SETTEXTFONT

SETTEXTFONT font

The input, is a list structure that completely describes a font. A font determines what your characters look like on the screen when using the command Label. The available fonts depend on your computer. MswLogo has two ways to specify the font you desire. This command can be used for programs. For interactive use you can use Set Font Command (Graphical User Interface) in the Menu.

Font: (LIST) that contains the following information: [[FaceName] Height Weight Italic Underline StrikeOut]. Where:

FaceName: (LIST) Specifies the typeface name of the font.

Height: (INTEGER) Specifies the desired height, in logical units, for the font. If this value is greater than zero, it specifies the cell height of the font. If it is less than zero,

it specifies the character height of the font.

Weight: (INTEGER) Specifies the font weight. This member ranges from 0 to 900 in increments 100. A value of 0 means use default weight.

Italic: (INTEGER) Specifies an italic font if nonzero.

Underline: (INTEGER) Specifies an underlined font if nonzero.

StrikeOut: (INTEGER) Specifies a strikeout font if nonzero.

Note, if you mistype the font name settextfont will list what fonts are available.

Example:

```
settextfont [[Symbol] 40 600 0 0 0]
label "Hello
```

TEXTFONT

TEXTFONT

Outputs a list describing the current font.

Output: (LIST) that contains the following information: [[FaceName] Height Weight Italic Underline StrikeOut].

See also SETTEXTFONT for definitions of of the mebers of this LIST.

Note, if you mistype the font name textfont will list what fonts are available.

Example:

```
settextfont [[Symbol] 40 600 0 0 0]
show textfont
[[Symbol] 40 600 0 0 0]
```

TEXTSCREEN

TEXTSCREEN
TS

(not supported under MswLogo yet)

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

FULLSCREEN
FS

(not supported under MswLogo yet)

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

SPLITSCREEN
SS

(not supported under MswLogo yet)

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

SETSCRUNCH xscale yscale

(not supported under MswLogo yet)

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.

REFRESH

REFRESH

(not supported under MswLogo yet)

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

NOREFRESH

NOREFRESH

(not supported under MswLogo yet)

Tells Logo not to remember the turtle's motions. This will make drawing faster, but prevents recovery if the window is overlaid.

ZOOM

ZOOM scale

This command allows LOGO to control the scale of the "Screen" window. The argument is the amount to zoom (scale) by. A number greater than 1.0 makes things bigger (e.g. 2.0 makes it 2 times bigger), a number smaller than 1.0 makes things smaller (0.5 makes it 1/2 as big). If an existing image is on the screen when you zoom it will be stretched or squeezed (this takes time by the way) according to the zoom (it may look a little jagged). If you "draw" while zoomed things will not be as jagged.

Even though things may appear jagged LOGO remembers everything as if zoom was normal (1.0) and only prints in normal. Once you return to zoom of 1.0 your image will not be stretched or squeezed to fit again. In other words in a zoom of 1.0 lines never get

jagged even if you drew it at zoom 0.5 or 2.0.

NOTE: Zoom works best if you choose a zoom that is a "power" of two. For example 2, 4, 8, 1/2 (0.5), 1/4 (0.25), 1/8 (0.125) etc.

Example:

```
cs
repeat 4 [fd 100 rt 90]
zoom 0.5
zoom 2.0
```

SCROLLX

SCROLLX deltax

This command allows LOGO to control the horizontal scroller of the "Screen" window. The argument is the amount to change the scroller by (delta). A positive number scrolls to the right and negative number scrolls to the left.

Example:

```
repeat 10 [scrollx 10]
```

SCROLLY

SCROLLY deltay

This command allows LOGO to control the vertical scroller of the "Screen" window. The argument is the amount to change the scroller by (delta). A positive number scrolls down and negative number scrolls to the up.

Example:

```
repeat 10 [scrolly 10]
```

SETFOCUS

SETFOCUS caption

This command allows LOGO to control which window is to have focus (is selected). The window desired is specified by its caption (or title). See also SETFOCUS.

caption:(LIST) Is the caption on the Window you wish to set focus to.

Note, the "caption" is not the same as the window's "name" used in the WINDOWS functions.

Example:

setfocus [MswLogo Screen]

GETFOCUS

GETFOCUS

This command will output the caption (title) of the window with the current focus (currently selected). See also SETFOCUS.

output:(WORD) Will be the caption (title) of window with current focus.

Example:

setfocus [MswLogo Screen]
show getfocus
[MswLogo Screen]

ICON

ICON caption

This command allows LOGO to icon a window (if iconable) with the given caption (title). See also UNICON.

caption:(LIST) Is the caption on the Window you wish to ICON.

Note, the "caption" is not the same as the window's "name" used in the WINDOWS functions.

Example:

icon "Commander
unicon "Commander

UNICON

UNICON caption

This command allows LOGO to unicon a window (if iconable) with the given caption (title). See also ICON.

caption:(LIST) Is the caption on the Window you wish to UNICON.

Example:

```
icon "Commander
unicon "Commander
```

USING COLOR

Several commands (SETPENCOLOR, SETFLOODCOLOR, SETSCREENCOLOR) exist in Logo to specify Red Green Blue intensities of color. Each input represents how much Red, Green and Blue you want in the color. Each input has a range of 0-255. By mixing different amounts of colors you can create 16.7 million different colors.

If you're running with a 256 color Windows Driver for your monitor you will get "real" colors. The colors are stored in a table called a palette. The palette has room for 256 colors (chosen from 16.7 possibilities). You can tell if you are running in 256 color mode (have a palette) by popping up the STATUS window and looking at the palette usage. If it shows "N/A" (Not Applicable) you are not running a 256 (or more) Windows driver. See the CLEARPALETTE command for more details about managing the palette.

If you are running with 16 (or less) color Windows Driver for your monitor Windows will simulate all the colors by mixing (dithering) the 16 (or less) colors. If the SETPENSIZE is too narrow (1) Windows cannot mix (dither) colors. Windows does not mix (dither) colors on Fonts either. If your hardware can support a 256 Windows Driver you should look into loading the appropriate Driver. The Documentation with your Graphics card explains the capabilities and how to load new Drivers. MswLogo is fun in any mode but it's even more fun in 256 color mode. You can load 256 color Bitmaps and do some incredible things with them.

Example of setting some common pen colors:

```
setpencolor [000 000 000]       black
setpencolor [255 255 255]       white
setpencolor [128 128 128]       gray
setpencolor [255 000 000]       Red
```

setpencolor [000 255 000] Green
setpencolor [000 000 255] Blue

Note that when running 256 color mode with a palette that several advantages and disadvantages occur:

Printing a 256 color image on a mono printer will be less pleasing than printing in 16 color mode. This is because no "dithering" (mixing dots) is used in 256 color mode.

Running in 256 color mode is slower and takes more memory and may not be possible on some smaller machines or machines with limited video capabilities.

Running in 256 color does give much more pleasant colors and even allows manipulation of 256 color .BMP files.

TURTLE AND WINDOW QUERIES

SHOWNP

SHOWNP

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

Example:

showturtle
show shownp
true
hideturtle
show shownp
false

PEN 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

PENDOWN
PD

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

Example:

repeat 10 [fd 10 pu fd 10 pd]

PENUP

PENUP
PU

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

Example:

repeat 10 [fd 10 pu fd 10 pd]

PENPAINT

PENPAINT
PPT

Sets the pen's position to DOWN and mode to PAINT. PAINT is the normal mode the turtle draws in.

Example:

```
penreverse
fd 100
bk 100
penpaint
fd 100
bk 100
```

PENERASE

PENERASE
PE

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

Example:

fd 100
penerase
bk 100

PENREVERSE

PENREVERSE
PX

Sets the pen's position to DOWN and mode to REVERSE.

Example:

penreverse
fd 100
bk 100
penpaint
fd 100
bk 100

SETPENCOLOR

SETPENCOLOR colorvector

Sets the pen to colorvector. The colorvector is a list of [red green blue] intensities. The pen color effects drawing text (LABEL) and drawing any line with the turtle (such as FORWARD). For an explanation of the arguments see USING COLOR.

Example:

repeat 255 [setpencolor (list repcount 0 0) fd 100 bk 100]

SETFLOODCOLOR

SETFLOODCOLOR colorvector

Sets the flood color to colorvector. The colorvector is a list of [red green blue] intensities. The flood color effects FILL and BITBLOCK commands. For an explanation of the arguments see USING COLOR.

Example:

```
cs
repeat 4 [fd 100 rt 90]
rt 45
pu
fd 20
setfloodcolor [0 255 0]
fill
cs
repeat 4 [fd 100 rt 90]
rt 45
pu
fd 20
setfloodcolor [125 125 125]
fill
```

SETSCREENCOLOR

SETSCREENCOLOR colorvector

Sets the screen color to colorvector. The colorvector is a list of [red green blue] intensities. The screen color immediately sets the screen background color. For an explanation of the arguments see USING COLOR.

Example:

```
setscreencolor [0 0 0]
setpencolor [255 255 255]
repeat 4 [fd 100 rt 90]
```

SETPENSIZE

SETPENSIZE size

Set hardware-dependent pen characteristics. These commands are not guaranteed compatible between implementations on different machines. The "size" is a list of two members, width and height. MswLogo only uses the second one of them. So just set

them both to the same value.

Example:

setpensize [5 5]

SETPENPATTERN

SETPENPATTERN pattern

(not supported under MswLogo yet)

Set hardware-dependent pen characteristics. These commands are not guaranteed compatible between implementations on different machines.

SETPEN

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.

Example:

```
cs repeat 4 [fd 100 rt 90]
make "savepen pen
setpensize [20 20]
cs repeat 4 [fd 100 rt 90]
setpen :savepen
cs repeat 4 [fd 100 rt 90]
```

CLEARPALETTE

CLEARPALETTE

This command clears the color palette. The color palette is filled by using the SETPENCOLOR, SETSCREENCOLOR, and SETFLOODCOLOR commands. The palette is only supported when windows is in 256 color mode (see USING COLOR). Once you run out of colors windows will choose the closest match. For example if the command

repeat 256 [setpencolor (list repcount 0 0)]

is issued it would fill the palette with 256 shades of red. At this point the palette would now be full. If you now wanted 256 shades of green they would NOT be granted (and matched to red). In order for them to be granted you have to let go of the 256 shades of red. This is done by "clearing the palette".

If you want a wide range of colors then select a wide range into the palette. For example the following would give you 216 colors covering a wide range. Once the few colors, left in the palette, are used, windows will have something reasonable to match further requests to (unlike the example above in which only shades of reds could be matched to).

```
repeat 6~
  [~
  make "red repcount*40~
  repeat 6~
    [~
    make "green repcount*40~
    repeat 6~
      [~
      make "blue repcount*40~
      setpencolor (list :red :green :blue)~
      ]~
    ]~
  ]
```

Note also that loading in .BMP (BITLOAD) files use up colors in the palette. Which again can be cleared using clearpalette. Clearing the screen does NOT clear the palette.

Example:

```
repeat 256 [setpencolor (list repcount 0 0)]
status
nostatus
clearpalette
status
nostatus
```

PEN QUERIES

PENDOWNP

PENDOWNP

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

Example:

```
pu
ifelse pendownp [print [Pen is DOWN]] [print [Pen is UP]
Pen is UP
pd
ifelse pendownp [print [Pen is DOWN]] [print [Pen is UP]
Pen is DOWN
```

PENMODE

PENMODE

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

Example:

```
penpaint
show penmode
paint
penerase
show penmode
erase
penreverse
show penmode
reverse
```

PENCOLOR

PENCOLOR

Output pen color information which is a list containing [Red Green Blue] intensities.

Example:

```
setpencolor [100 200 50]
show pencolor
[100 200 50]
setpencolor [0 0 0]
show pencolor
```

[0 0 0]

FLOODCOLOR

FLOODCOLOR

Output flood color information which is a list containing [Red Green Blue] intensities.

Example:

setfloodcolor [100 200 50]
show floodcolor
[100 200 50]
setfloodcolor [0 0 0]
show floodcolor
[0 0 0]

SCREENCOLOR

SCREENCOLOR

Output screen color information which is a list containing [Red Green Blue] intensities.

Example:

setscreencolor [100 200 50]
show screencolor
[100 200 50]
setscreencolor [0 0 0]
show screencolor
[0 0 0]

PENSIZE

PENSIZE

Output pen size information which is a list containing [Width Height]. Width is not used in MswLogo.

Example:

setpensize [10 20]

show pensize
[20 20]
setpensize [1 1]
show pensize
[1 1]

PENPATTERN

PENPATTERN

(not supported under MswLogo yet)

Output hardware-specific pen information.

PEN

PEN (library procedure)

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

Example:

```
cs repeat 4 [fd 100 rt 90]
make "savepen pen
setpensize [20 20]
cs repeat 4 [fd 100 rt 90]
setpen :savepen
cs repeat 4 [fd 100 rt 90]
```

WORKSPACE MANAGEMENT

PROCEDURE DEFINITION

TO

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

Command that 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", "input2", etc. Any number of inputs is 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 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 ">" (pops up a dialog box in MswLogo) and whatever instructions you type become part of the definition until you type a line containing only the word END.

Example:

```
to echo :times :thing
repeat :times [print :thing]
end
echo 2 "Hello
Hello
Hello
echo 3 "Bye
Bye
Bye
Bye
```

END

END (special form)

This is not really a command. It is to let you to depict the END of a procedure. See also TO.

Example:

```
to echo :times :thing
repeat :times [print :thing]
end
echo 2 "Hello
Hello
Hello
echo 3 "Bye
Bye
Bye
Bye
```

DEFINE

DEFINE procname text

Command that 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.

Example:

```
define "abc [[a b] [print :a] [print :b]]
abc "Hello "Bye
Hello
Bye
```

TEXT

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.

Example:

```
define "abc [[a b] [print :a] [print :b]]
abc "Hello "Bye
show text "abc
[[a b] [print :a] [print :b]]
```

FULLTEXT

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 element 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!

Example:

```
define "abc [[a b] [print :a] [print :b]]
abc "Hello "Bye
show fulltext "abc
[[to abc :a :b] [print :a] [print :b] end]
```

COPYDEF

COPYDEF newname oldname

Command that 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."

Example:

```
to welcome
print "Hello
end
welcome
Hello
copydef "sayhello "welcome
sayhello
Hello
```

VARIABLE DEFINITION

MAKE

MAKE varname value

Command that 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.

Example:

make "foo [Hello how are you]
show :foo
[Hello how are you]

NAME

NAME value varname (library procedure)

Command that is the same as MAKE but with the inputs in reverse order.

Example:

name [Hello how are you] "foo
show :foo
[Hello how are you]

LOCAL

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

Command that 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 sub procedure 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.

Example:

to foo
make "bar 1
print :bar
end
foo
1
show :bar

1
to abc
local "xyz
make "xyz 1
print :xyz
end
abc
1
show :xyz
xyz has no value

THING

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.

Example:

make "foo [Hello how are you]
show thing "foo
[Hello how are you]
show :foo
[Hello how are you]

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

PPROP plistname propname value

Command that adds a property to the "plistname" property list with name "propname" and value "value".

Example:

```
pprop "plist1 "p1 1
pprop "plist1 "p2 2
pprop "plist2 "p1 10
pprop "plist2 "p2 20
show gprop "plist1 "p1
1
show gprop "plist1 "p2
2
show gprop "plist2 "p1
10
show gprop "plist2 "p2
20
```

GPROP

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.

Example:

```
pprop "plist1 "p1 1
pprop "plist1 "p2 2
pprop "plist2 "p1 10
pprop "plist2 "p2 20
show gprop "plist1 "p1
1
show gprop "plist1 "p2
2
show gprop "plist2 "p1
10
show gprop "plist2 "p2
20
```

REMPROP

REMPROP plistname propname

Command that removes the property named "propname" from the property list named "plistname".

Example:

```
pprop "plist1 "p1 1
pprop "plist1 "p2 2
show plist "plist1
[p2 2 p1 1]
remprop "plist1 "p1
show plist "plist1
[p2 2]
```

PLIST

PLIST plistname

Outputs a list whose odd-numbered elements are the names, and whose even-numbered elements 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 the list output by PLIST.

Example:

```
pprop "plist1 "p1 1
pprop "plist1 "p2 2
show plist "plist1
[p2 2 p1 1]
```

PREDICATES (Workspace)

PROCEDUREP

PROCEDUREP name

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

Example:

```
make "foo 1
to bar
```

end
show procedurep "foo
false
show procedurep "bar
true

PRIMITIVEP

PRIMITIVEP 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.

Example:

to fwd :arg
forward :arg
end
show primitivep "fwd
false
show primitivep "forward
true

DEFINEDP

DEFINEDP 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.)

Example:

define "abc [[a b] [print :a] [print :b]]
show definedp "forward
false
show definedp "abc
true

NAMEP

NAMEP name

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

Example:

define "abc [[a b] [print :a] [print :b]]
make "xyz 1
show namep "abc
false
show namep "xyz
true

QUERIES

CONTENTS

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.

Example:

to proc1
end
make "name1 1
pprop "prop1 "p1 1
show contents
[[proc1] [name1] [prop1]]

BURIED

BURIED

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

Example:

pots
show buried

[[pots] [] []]

PROCEDURES

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.)

Example:

```
to foo
end
to bar
end
show procedures
```
[bar foo]

NAMES

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.

Example:

```
make "foo 1
make "bar 2
show names
```
[[] [bar foo]]

PLISTS

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.

Example:

pprop "prop1 "p1 1
show plists
[[] [] [prop1]]

NAMELIST

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.

namelist [foo bar]
[[] [foo bar]]

PLLIST

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 CONTENTS above.

pllist [foo bar]
[[] [] [foo bar]]

INSPECTION

PO

PO contentslist

Command that prints to the write stream the definitions of all procedures, variables, and property lists named in the input contents list.

Example:

```
to xxx
print "Hello
end
po contents
to xxx
print "Hello
end
```

POALL

POALL (library procedure)

Command that prints all unburied definitions in the workspace. Abbreviates PO CONTENTS.

Example:

```
to xxx
print "Hello
end
poall
to xxx
print "Hello
end
```

POPS

POPS (library procedure)

Command that prints the definitions of all unburied procedures in the workspace. Abbreviates PO PROCEDURES.

Example:

```
to xxx
print "Hello
end
pops
to xxx
print "Hello
end
```

PONS

PONS (library procedure)

Command that prints the definitions of all unburied variables in the workspace. Abbreviates PO NAMES.

Example:

make "foo 1
pons
Make "foo 1

POPLS

POPLS (library procedure)

Command that prints the contents of all unburied property lists in the workspace. Abbreviates PO PLISTS.

Example:

pprop "plist1 "p1 1
popls
Pprop "plist1 "p1 1

PON

PON varname (library procedure)
PON varnamelist

Command that prints the definitions of the named variable(s). Abbreviates PO NAMELIST varname(list).

Example:

make "foo 1
make "bar 2
pon [foo bar]
Make "foo 1
Make "bar 2

POPL

POPL plname (library procedure)
POPL plnamelist

Command that prints the definitions of the named property list(s). Abbreviates PO PLLIST plname(list).

Example:

pprop "plist1 "p1 1
popl [plist1]
Pprop "plist1 "p1 1

POT

POT contentslist

Command that 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.

Example:

to foo
end
pot procedures
to foo

POTS

POTS (library procedure)

Command that prints the title lines of all unburied procedures in the workspace. Abbreviates POT PROCEDURES.

Example:

to foo
end
pots

to foo

WORKSPACE CONTROL

ERASE

ERASE contentslist
ER contentslist

Command that 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.

Example:

to foo
end
to bar
end
pots
to foo
to bar
erase [foo]
pots
to bar

ERALL

ERALL (library procedure)

Command that erases all unburied procedures, variables, and property lists from the workspace. Abbreviates ERASE CONTENTS.

Example:

to foo
end
to bar
end
pots
to foo
to bar

erall
pots

ERPS

ERPS (library procedure)

Command that erases all unburied procedures from the workspace. Abbreviates ERASE PROCEDURES.

Example:

```
to foo
end
to bar
end
pots
to foo
to bar
erps
pots
```

ERNS

ERNS (library procedure)

Command that erases all unburied variables from the workspace. Abbreviates ERASE NAMES.

Example:

```
make "foo 1
make "bar 2
pons
Make "bar 2
Make "foo 1
erns
pons
```

ERPLS

ERPLS (library procedure)

Command that erases all unburied property lists from the workspace. Abbreviates ERASE PLISTS.

Example:

```
pprop "plist1 "p1 1
popls
Pprop "plist1 "p1 1
erpls
popls
```

ERN

ERN varname (library procedure)
ERN varnamelist

Command that erases from the workspace the variable(s) named in the input. Abbreviates ERASE NAMELIST varname(list).

Example:

```
make "foo 1
make "bar 2
pons
Make "bar 2
Make "foo 1
ern [foo]
pons
Make "bar 2
```

ERPL

ERPL plname (library procedure)
ERPL plnamelist

Command that erases from the workspace the property list(s) named in the input. Abbreviates ERASE PLLIST plname(list).

Example:

```
pprop "plist1 "p1 1
pprop "plist2 "p1 2
```

popls
Pprop "plist1 "p1 1
Pprop "plist2 "p1 2
erpl [plist2]
popls
Pprop "plist1 "p1 1

BURY

BURY contentslist

Command that 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.

Example:

to foo
Print [Here I am]
end
to bar
Print [Here I go]
end
pots
to bar
to foo
bury "foo
pots
to bar
foo
Here I am

BURYALL

BURYALL (library procedure)

Command that abbreviates BURY CONTENTS.

Example:

to foo
Print [Here I am]

end
to bar
Print [Here I go]
end
pots
to bar
to foo
buryall
pots
foo
Here I am
bar
Here I go

BURYNAME

BURYNAME varname (library procedure)
BURYNAME varnamelist

Command that abbreviates BURY NAMELIST varname(list).

Example:

make "foo 1
make "bar 2
pons
Make "bar 2
Make "foo 1
buryname [foo]
pons
Make "bar 2
show :foo
1
show :bar
2

UNBURY

UNBURY contentslist

Command that unburies the procedures, variables, and property lists named in the input. That is, the named items will be returned to view in CONTENTS, etc.

Example:

make "foo 1
make "bar 2
pons
Make "bar 2
Make "foo 1
buryname [foo]
pons
Make "bar 2
unbury [[] [foo] []]
pons
Make "bar 2
Make "foo 1

UNBURYALL

UNBURYALL (library procedure)

Command that abbreviates UNBURY BURIED.

Example:

make "foo 1
make "bar 2
pons
Make "bar 2
Make "foo 1
buryname [foo]
pons
Make "bar 2
unburyall
pons
Make "bar 2
Make "foo 1

UNBURYNAME

UNBURYNAME varname (library procedure)
UNBURYNAME varnamelist

Command that abbreviates UNBURY NAMELIST varname(list).

Example:

make "foo 1
make "bar 2
pons
Make "bar 2
Make "foo 1
buryname [foo]
pons
Make "bar 2
unbury [foo]
pons
Make "bar 2
Make "foo 1

TRACE

TRACE contentslist

Command that 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 STOPs or OUTPUTs. 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.

Example:

to myprog :a
print :a
end
myprog "Hello
Hello
trace "myprog
myprog "Hello
(myprog "Hello)
Hello
untrace "myprog
myprog "Hello
Hello

UNTRACE

UNTRACE contentslist

Command that turns off tracing for the named items.

Example:

```
to myprog :a
print :a
end
myprog "Hello
Hello
trace "myprog
myprog "Hello
( myprog "Hello )
Hello
untrace "myprog
myprog "Hello
Hello
```

STEP

STEP contentslist

Command that 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.

Example:

```
to myprog
fd 10
rt 90
fd 20
lt 90
end
myprog
step "myprog
<Each line of myprog will be displayed and will wait for OK to continue to next line>
myprog
unstep "myprog
myprog
```

UNSTEP

UNSTEP contentslist

Command that turns off stepping for the named items.

Example:

```
to myprog
fd 10
rt 90
fd 20
lt 90
end
myprog
step "myprog
<Each line of myprog will be displayed and will wait for OK to continue to next line>
myprog
unstep "myprog
myprog
```

EDIT

EDIT contentslist
ED contentslist
(EDIT)
(ED)

Command that edits the definitions of the named item(s), using the Logo editor. See also Editor for details of how the editor works.

Example:

```
to "myprog
print "Hrllo
end
myprog
Hrllo
edit "myprog
<change Hrllo to Hello and exit editor>
myprog
Hello
```

EDALL

EDALL (library procedure)

Command that abbreviates EDIT CONTENTS.

Example:

```
to myprog
print :myarg
end
make "myarg "Hrllo
myprog
Hrllo
editall
<change Hrllo to Hello and exit editor>
myprog
Hello
```

EDPS

EDPS (library procedure)

Command that abbreviates EDIT PROCEDURES.

Example:

```
to "myprog1
print "Hrllo
end
to "myprog2
print "Byr
end
myprog1
Hrllo
myprog2
Byr
editall
<change Hrllo to Hello, change Byr to Bye and exit editor>
myprog1
Hello
myprog2
Bye
```

EDNS

EDNS (library procedure)

Command that abbreviates EDIT NAMES.

Example:

```
to myprog
print :myarg
end
make "myarg "Hrllo
myprog
Hrllo
editall
<change Hrllo to Hello and exit editor>
myprog
Hello
```

EDPLS

EDPLS (library procedure)

Command that abbreviates EDIT PLISTS.

Example:

```
pprop "plist1 "p1 1
popls
Pprop "plist1 "p1 1
edpls
<change 1 to 2 and exit editor>
popls
Pprop "plist1 "p1 2
```

EDN

EDN varname (library procedure)
EDN varnamelist

Command that abbreviates EDIT NAMELIST varname(list).

Example:

```
to myprog
print :myarg
end
make "myarg "Hrllo
myprog
Hrllo
edn "myarg
<change Hrllo to Hello and exit editor>
myprog
Hello
```

EDPL

EDPL plname (library procedure)
EDPL plnamelist

Command that abbreviates EDIT PLLIST plname(list).

Example:

```
pprop "plist1 "p1 1
popls
Pprop "plist1 "p1 1
edpl "plist1
<change 1 to 2 and exit editor>
popls
Pprop "plist1 "p1 2
```

SAVE

SAVE filename

Command that 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
```

Example:

```
to myprog1
print "Hello1
end
to myprog2
print "Hello2
end
pots
to myprog1
to myprog2
save "myprogs.lg
erall
pots
load "myprogs.lg
pots
to myprog1
to myprog2
```

SAVEL

SAVEL contentslist filename (library procedure)

Command that saves the definitions of the procedures, variables, and property lists specified by "contentslist" to the file named "filename".

Example:

```
to myprog1
print "Hello1
end
to myprog2
print "Hello2
end
pots
to myprog1
to myprog2
savel [[myprog1] [] []] "myprogs.lg
erall
pots
load "myprogs.lg
```

pots
to myprog1

LOAD

LOAD filename

Command that 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 instruction list after the file is loaded.

Example:

```
to myprog1
print "Hello1
end
to myprog2
print "Hello2
end
pots
to myprog1
to myprog2
save "myprogs.lg
erall
pots
load "myprogs.lg
pots
to myprog1
to myprog2
```

NOSTATUS

NOSTATUS

This command has the same effect as hitting the NOSTATUS BUTTON. That is, it kills the popup status window. See also STATUS command.

Example:

```
status
nostatus
```

STATUS

STATUS

This command has the same effect as hitting the STATUS BUTTON. That is, it pops up the status window. See also NOSTATUS command.

Example:

status
nostatus

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.

CONTROL COMMANDS

RUN

RUN instructionlist

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

Example:

```
make "thingstodo [print]
make "thingstodo lput "\"Hello :thingstodo
run :thingstodo
Hello
```

RUNRESULT

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
```

Example:

```
make "thingstodo [first [1 2 3]]
make "answer runresult :thingstodo
```

show :answer
[1]

REPEAT

REPEAT num instructionlist

Command that runs the "instructionlist" repeatedly, "num" times.

Example:

repeat 3 [print (list "This "Is "loop repcount)]
This Is loop 1
This Is loop 2
This Is loop 3

REPCOUNT

REPCOUNT

This operation may be used only within the range of a REPEAT command. It outputs the number of repetitions that have been done, including the current one. That is, it outputs 1 the first time through, 2 the second time, and so on.

Example:

repeat 3 [print (list "This "Is "loop repcount)]
This Is loop 1
This Is loop 2
This Is loop 3

IF

IF tf instructionlist
(IF tf instructionlist1 instructionlist2)

Command or operation where 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.

Example:

if 1=1 [print [Yes it is true]]
Yes it is true

IFELSE

IFELSE tf instructionlist1 instructionlist2

Command or operation where 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.

Example:

ifelse 1=1 [print [Yes it is true]] [print [No it is false]]
Yes it is true
ifelse 1=0 [print [Yes it is true]] [print [No it is false]]
No it is false

TEST

TEST tf

Command that 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.

Example:

```
to mytest :arg
test 1=:arg
print [Do this]
print [Do that]
iftrue [print [arg was the number one]]
iffalse [print [arg was NOT the number one]]
```

```
end
mytest 1
Do this
Do that
arg was the number one
```

IFTRUE

IFTRUE instructionlist
IFT instructionlist

Command that 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.

Example:

```
to mytest :arg
test 1=:arg
print [Do this]
print [Do that]
iftrue [print [arg was the number one]]
iffalse [print [arg was NOT the number one]]
end
mytest 1
Do this
Do that
arg was the number one
```

IFFALSE

IFFALSE instructionlist
IFF instructionlist

Command that 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.

Example:

```
to mytest :arg
test 1=:arg
print [Do this]
print [Do that]
iftrue [print [arg was the number one]]
```

```
iffalse [print [arg was NOT the number one]]
end
mytest 0
Do this
Do that
arg was NOT the number one
```

TRUE

TRUE (special form)

This is a special word to indicate a positive condition.

Example:

```
show 1=1
true
if "true [print [True is always true]]
True is always true
```

FALSE

FALSE (special form)

This is a special word to indicate a negative condition.

Example:

```
show 1=0
false
ifelse "false [print [We can not get here]] [print [False is always false]]
False is always false
```

STOP

STOP

Command that 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.

Example:

```
to myprog :arg
print [Before Stop]
if 1=:arg [stop]
print [After Stop]
end

myprog 1
```
Before Stop
```
myprog 2
```
Before Stop
After Stop

OUTPUT

OUTPUT value

Command that 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.

Example:

```
to myprog
output [This is the output]
end
show myprog
```
[This is the output]

CATCH

CATCH tag instructionlist

Command or operation that 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].)

Example:

```
to myprog2
print [Before throw]
throw "tag1
print [We never get here because we THROW back]
end
to myprog1
catch "tag1 [myprog2]
print [I'm back]
end
myprog1
```
Before throw
I'm back

THROW

THROW tag
(THROW tag value)

Command that 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 ^C) has the same effect (or HALT button in MswLogo).

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.

Example (There may be a bug in throw here):

```
to myprog2
print [Before throw]
(throw "tag1 [We need to get back])
print [We never get here because we THROW back]
end
to myprog1
show catch "tag1 [myprog2]
print [I'm back]
end
myprog1
```
Before throw
[We need to get back]
I'm back

ERROR

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. (See also the list of ERROR CODES)

Example:

```
to myprog
fd 1000
end
fence
catch "error [myprog]
show error
```
[3 [turtle out of bounds] myprog [fd 1000]]

PAUSE

PAUSE

Command or operation that 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.

Example:

```
to myprog
repeat 180 [rt 2 if 90=repcount [pause]]
print "Done
end
myprog
Pausing...myprog
<Enter SHOW HEADING in Pause-Box and hit OK>
180
<Enter CO in Pause Box and hit OK>
Done
```

CONTINUE

CONTINUE value
CO value
(CONTINUE)
(CO)

Command that 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.

Example:

```
to myprog
repeat 180 [rt 2 if 90=repcount [pause]]
print "Done
end
myprog
Pausing...myprog
<Enter SHOW HEADING in Pause-Box and hit OK>
```

180
<Enter CONTINUE in Pause Box and hit OK>
Done

YIELD

YIELD

The yield command is like hitting the YIELD BUTTON but is under Logo control.
See also NoYield command.

Example:

See YIELD BUTTON

NOYIELD

NOYIELD

The yield command is like hitting the NOYIELD BUTTON but is under Logo control.
See also Yield command.

Example:

See NOYIELD BUTTON

EVENTCHECK

EVENTCHECK

The EVENTCHECK command is like hitting the YIELD BUTTON and then hitting it again immediately after. It basically checks to see if any events are waiting to be processed. Some commands that issue "callbacks" run in a Non-Yielding state. If the callback code is short and quick you won't see any problems. But if it's long the user will have no control during this period. If you sprinkle a few of these around the user will not loose control for such a "long" period of time.

This command can be used as an alternative to placing YIELD and NOYIELD commands around compute intensive loops. You can encapsulate the whole procedure in a NOYIELD and YIELD pair then you can place EVENTCHECK's appropriately in your loops.

Example:

```
mouseon [repeat 72 [repeat 4 [fd 100 rt 90] rt 5]] [] [] [] []
<now click the mouse in MswLogo Screen>
<note that while it is drawing you cannot HALT it, because mouse callbacks do not yield>
mouseon [repeat 72 [repeat 4 [fd 100 rt 90] rt 5 eventcheck]] [] [] [] []
<now click the mouse in MswLogo Screen>
<note that while it is drawing you CAN now HALT>
```

SETCURSORWAIT

SETCURSORWAIT

This will set the cursor to the familiar hourglass shape. Its purpose is to indicate two things. One is that the operation about to take place will take some time. And second that during that time the user does not have control of Windows (not yielding). This function only works when not yielding. Once yielding the appropriate cursor will be used on the next event involving the cursor (like moving the mouse). This means that you must issue NOYIELD before each SETCURSORWAIT. If you decide to YIELD momentarily during a computation you must SETCURSORWAIT again. In other words if you wish to use SETCURSORWAIT it should always be paired up with a NOYIELD (just before it).

Example:

```
repeat 100~
  [~
  noyield~
  setcursorwait~
  repeat 100 ~
    [~
    (work to be done)~
    ]~
  setcursornowait~
  yield~
  ]
```

HALT

HALT

The halt command is like hitting the HALT BUTTON but is under Logo control.

Example:

repeat 1000 [fd 100 bk 100 rt 1]
<Hit HALT BUTTON while it is drawing>
Stopping...

WAIT

WAIT time

Command that 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 waiting.

Example:

wait 60

SETTIMER

SETTIMER id delay [callback]

This command sets up a timer identified by id (1-31) to call the logo commands in callback in every delay (milliseconds) of time. It will continue to "fire" every delay until you issue a CLEARTIMER id. You can generate as much trouble as you can power with this command.

You can clear the timer at any time. This means that you can clear the timer in the callback (handler) itself (a single fire). You can also issue a timer at any time, including in the callback (handler). Halting will clear all timers. The behavior of timers from 1-16 are slightly different than 17-13. Timers 1-16 will not allow the callback code to Yield. Where as timers 17-31 will Yield.

You must be sure that you are handling (servicing) the timer request faster than they are coming in. If you do not, you will eventually "Stack-Overflow" and MswLogo will shutdown.

You must also be sure that the callback code is error free. If not, you may generate more "OK" boxes then you have ever seen and eventually "Stack OverFlow".

It's a good Idea to save your code frequently to disk when developing code that uses timers. These are not really bugs, your just taping directly into Windows and you have

to be careful.

You can "block" all timers from firing (interrupting) in any code, including the callback (handler), by issuing the NOYIELD command. You can "restore" firing by issuing the YIELD command. Blocking does not loose any "events" (firings), they are queued up. So don't "block" to long.

Each timer can have different callbacks or the same callbacks.

Example:

```
settimer 1 200 [setpencolor (list random 256 random 256 random 256)]
<note MswLogo returns here and the timer is firing (changing the pen randomly)>
repeat 72 [repeat 4 [fd 100 rt 90] rt 5]
cleartimer 1
```

CLEARTIMER

CLEARTIMER id

Command that clears the timer set by SETTIMER and identified by id (1-31).

Example:

```
settimer 1 200 [setpencolor (list random 256 random 256 random 256)]
<note MswLogo returns here and the timer is firing (changing the pen randomly)>
repeat 72 [repeat 4 [fd 100 rt 90] rt 5]
cleartimer 1
```

BYE

BYE

Command that exits from Logo; returns to the operating system.

Example:

```
bye
```

.MAYBEOUTPUT

.MAYBEOUTPUT value (special form)

Works like OUTPUT except that the expression that provides the input value might not 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 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

IGNORE value (library procedure)

Command that does nothing. Used when an expression is evaluated for a side effect and its actual value is unimportant.

Example:

```
to myprog :arg
print :arg
output count :arg
end
myprog "Hello
Hello
I don't know what to do with 5
ignore myprog "Hello
Hello
```

`

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

FOR

FOR forcontrol instructionlist (library procedure)

Command in which 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 element 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 element 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.

Example:

```
for [i 2 7 1.5] [print :i]
2
3.5
5
6.5
```

DO.WHILE

DO.WHILE instructionlist tfexpression (library procedure)

Command that 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.

Example:

```
make "i 0
do.while [make "i :i+1 print :i] [:i<3]
1
2
3
```

WHILE

WHILE tfexpression instructionlist (library procedure)

Command that 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.

Example:

```
make "i 0
while [:i<3] [make "i :i+1 print :i]
1
2
3
```

DO.UNTIL

DO.UNTIL instructionlist tfexpression (library procedure)

Command that 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.

Example:

```
make "i 0
do.until [make "i :i+1 print :i] [:i>3]
1
2
3
4
```

UNTIL

UNTIL tfexpression instructionlist (library procedure)

Command that 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.

Example:

```
make "i 0
until [:i>3] [make "i :i+1 print :i]
1
2
3
4
```

GO

GO label

(NOT IMPLEMENTED YET)

This command can be used only inside a procedure. The input must be a number. The same number must appear at the beginning of some line in the same procedure. (This line number is otherwise ignored.) The next command executed will be the one on the indicated line in the definition. Note: there is always a better way to do it without using go.

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.

Example:

```
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 the available data in order. That is, if data ?1 through ?3 are available, the template "PROC is equivalent to [PROC ?1 ?2 ?3].

Example:

```
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 element, whose first element is itself a list. The first element 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
```

These iteration tools are extended versions of the ones in Appendix B of the book Computer Science Logo Style, Volume 3: Advanced Topics by Brian Harvey [MIT Press, 1987]. The extensions are primarily to allow for variable numbers of inputs.

APPLY

APPLY template inputlist

Command or operation that 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.

Example:

```
show apply "sum [1 2 3]
6
```

INVOKE

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

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

Example:

```
show (invoke "sum 1 2 3)
6
```

FOREACH

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

Command that evaluates the template list repeatedly, once for each element 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.

Example:

foreach [a b c d] [print (se "index # "value ? "rest ?rest)]
index 1 value a rest b c d
index 2 value b rest c d
index 3 value c rest d
index 4 value d rest

MAP

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 element of the output is the result of evaluating the template list, filling the slots with the corresponding element(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.

Example:

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

MAP.SE

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.

Example:

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

FILTER

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.

Example:

```
to vowelp :arg
if :arg="a [output "true]
if :arg="e [output "true]
if :arg="i [output "true]
if :arg="o [output "true]
if :arg="u [output "true]
output "false
end
print filter "vowelp "elephant
eea
```

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

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

FIND

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 constituent in its slot is TRUE. If there is no such constituent, the empty list is output.

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

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

Example:

```
to find1 :arg
```

```
if :arg=1 [output "true]
output "false
end
show (find "find1 [2 4 3 0])
[]
show (find "find1 [2 1 3 0])
1
```

REDUCE

REDUCE template data (library procedure)

Outputs the result of applying the template to accumulate the elements 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 constituent. Otherwise, the template is first applied with ?1 filled with the next-to-last constituent 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.

Example:

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

print reduce "max [2 3 8 7 9 0]
9
```

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
print (max 2 3 8 7 9 0)
9
```

CROSSMAP

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.

Example:

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 CSLS, CROSSMAP templates may use the notation :1 instead of ?1 to indicate slots.

CASCADE

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, see TEMPLATE-BASED ITERATION) 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.

Example:

```
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.

Example:

```
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

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.

Example:

cascade ???

TRANSFER

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 element 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.

Example (for each word in the last input, if that word is already part of the result, forget it; if not, append that word to the result so far. The result is initially empty.):

show transfer [] [ifelse memberp ?in ?out [?out] [lput ?in ?out]] [A B C B D E F B C G]
[A B C D E F G]

MACROS

MACRO COMMANDS

.MACRO

.MACRO procname :input1 :input2 ... (special form)
.DEFMACRO procname text
MACROP name

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 like TO except that the new procedure becomes a macro; .DEFMACRO is like DEFINE with the same exception. MACROP returns TRUE if its input is the name of a macro.

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 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 includes 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 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 slow, since its tree recursive and written in Logo.

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 element 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 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. The first two are so bad that Logo exits immediately.

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

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.

SPECIAL COMMANDS

CASEIGNOREDP

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.

Example:

```
make "caseignoredp "false
show equalp "a "A
false
make "caseignoredp "true
show equalp "a "A
true
```

ERRACT

ERRACT

An instructionlist that will be run if there is an error. Typically has the value [PAUSE] to allow interactive debugging.

Example:

```
fence
fd 1000
Turtle out of bounds
make "erract [print [You really blew it]]
fd 1000
```

You really blew it

PRINTDEPTHLIMIT

PRINTDEPTHLIMIT

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

Example:

print [[1 [2 [3 [4 [5]]]]]]
[1 [2 [3 [4 [5]]]]]
make "printdepthlimit 4
print [[1 [2 [3 [4 [5]]]]]]
[1 [2 [3 [... ...]]]]

PRINTWIDTHLIMIT

PRINTWIDTHLIMIT

If a nonnegative integer, indicates the maximum number of elements in any one list that will be printed by PRINT, etc.

print [1 2 3 4 5 6]
1 2 3 4 5 6
make "printwidthlimit 4
print [1 2 3 4 5 6]
1 2 3 4 ...

REDEFP

REDEFP

If TRUE, allows primitives to be erased (ERASE) or redefined (COPYDEF).

Example:

erase "fd
fd is a primitive
make "redefp "true
erase "fd

fd
I don't know how to fd

STARTUP

STARTUP

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

Example:

to myprog
print "Hello
end
make "startup [myprog]
save "myprog.lg
myprog.lg
erall
load "myprog.lg
Hello

MACHINE

MACHINE

This command outputs the characteristics of the machine in a list. Where the list has the following format [Windows WindowsVersion BitMapWidth BitMapHeight Palette]. Where a 1 is true and 0 is false. This can be used to write code that is compatible between multiple platforms (e.g. Windows 3.0 vs. Windows 3.1).

Example:

show machine
[1 31 1000 1000 0]

GETTING HELP

HELP COMMANDS

HELP

HELP
(HELP keyword)

This command has two forms (with and without a keyword argument). Without a keyword Logo will enter Windows help on LOGO at the top level. With a keyword argument Logo will search the Windows help for the keyword. You must enter the full keyword.

Example:

```
help "introduction      ;(Enter help on introduction)
help "intro             ;(Will fail to find the introduction)
```

For context sensitive help, see the EDIT command.

Note also that Windows Help allows you to Copy text from Help to the Clipboard. Since the Editor within Logo also supports the Clipboard this means that you can copy examples within Help into the Editor, save them, and then execute them. To do this:

Click on EDIT in the Help menu.
Select Copy.
Using the mouse select the desired text (example code).
Click on COPY button (It is now in the Clipboard).
Enter the Logo Editor.
Set the cursor where you want the example inserted.
Click on EDIT in the Edit menu.
Select paste (it's now in the editor).

WINHELP

WINHELP filename
(WINHELP filename keyword)

This command has two forms (with and without a keyword argument). Without a

keyword Logo will enter Windows help on the desired FILENAME at the top level. With a keyword argument Logo will search the Windows help FILENAME for the keyword. You must enter the full keyword.

Example:

```
Winhelp c:\myhelp "introduction       ;(Enter help on introduction)
```

DIRECTORIES

Directories are the same as they are in DOS and Windows. They are added to logo to make things easier.

DIRECTORY COMMANDS

DIR

DIR

This command shows files and directories in the current directory

Example:

dir
File LOGO.EXE
File LOGO.HLP
File MYPROG.LG
Directory .
Directory ..
Directory LOGOLIB
Directory EXAMPLES

CHDIR

CHDIR directory

This command is exactly like the DOS command CHDIR (cd). The parameter must be a word (the name of the directory you want work in). You can use the DIR command to list both procedures and directories. See also POPDIR and MKDIR.

Example:

chdir "examples
Pushed to C:\LOGO\EXAMPLES
popdir
Popped to C:\LOGO

POPDIR

POPDIR

This command pops you up (1 level) out of a directory. It is equivalent to CHDIR ".. command. See also CHDIR and MKDIR

Example:

chdir "examples
Pushed to C:\LOGO\EXAMPLES
popdir
Popped to C:\LOGO

MKDIR

MKDIR directory

This command makes (creates) a directory and then changes (CHDIR) you into it. The parameter must be a word (the name of the directory you want to make). You can use the DIR command to list the files and directories in the current directory. Once done with the directory you can pop (POPDIR) back out and change to another.

Example:

mkdir "junk
Now in newly created directory junk
popdir
Popped to C:\LOGO

RMDIR

RMDIR directory

This command removes (deletes) a directory. The parameter must be a word (a name of an existing directory). You cannot remove a directory if you are (CHDIR) into it. Nor can you remove it, if it contains any files or other directories (DIR must not return anything while changed to the directory you want to remove). So to be sure you can remove it, do a DIR and check if it's there, then change (CHDIR) to it, do DIR again and confirm it's empty, pop (POPDIR) back out and then remove (RMDIR) it. See also MKDIR.

Example:

mkdir "junk
Now in newly created directory junk
popdir
Popped to C:\LOGO
remdir "junk
Logo directory junk removed

WINDOWS FUNCTIONS

This section describes how the LOGO programmer can create powerful Graphical User Interfaces (GUIs). Most any GUI has an Application Programming Interface (API), which is what this section documents for Logo regarding Windows. Both the GUI and the API maintain a parent-child relationship. That is, Windows and controls appear (graphically) in a nested fashion (or owned by one another). The nested appearance is maintained in the code you develop to present it.

For each "somethingCREATE" command a parent must be specified which identifies the relationship. When a parent of any kind is "somethingDELETEd" so will all its "child" Windows and controls. Every Window function (command) specifies a "name". On the "somethingCREATE" commands it is there to label the "something" so that it can later be referenced. On most other commands it is used as a "handle" to identify the Window or control you wish to communicate with. And third it is also used to reference the "parent" of the Window or control that is to "own" this new "something".

Note that the coordinates used in all the Windows functions are in Windows coordinate system (NOT the turtle coordinate system). That is the Y axis is upside down (Positive numbers are down). The origin is NOT the middle of the screen it is the upper left hand corner of the window.

WINDOW COMMANDS

WINDOWCREATE

WINDOWCREATE parent name title xpos ypos width height

This command will create a WINDOW. A WINDOW is used as the frame work to which you add other window objects or controls (sometimes called widget's). You can add things such as buttons, scrollbars, listboxes, etc. to the WINDOW. (see Modal vs. Modeless Windows)

parent: (WORD) Is the name of the WINDOW that is to own this new WINDOW. If this is the first window created use "root as the parent name.

name: (WORD) Is used to identify this WINDOW (perhaps as the parent of another future window or control) and MUST be unique.

title: (LIST) Is used as the title (caption) of the WINDOW.

xpos: (INTEGER) Is the X position you wish to place the upper left corner of the new

WINDOW.

ypos: (INTEGER) Is the Y position you wish to place the upper left corner of the new WINDOW.

width: (INTEGER) Is the width of the new WINDOW.

height: (INTEGER) Is the height of the new WINDOW.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
windowdelete "mywindow
```

WINDOWDELETE

WINDOWDELETE name

This command will delete (close) the WINDOW with the given name. Note all the child windows and controls will be also deleted.

name: (WORD) Is used to identify the WINDOW you want destroyed.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
windowdelete "mywindow
```

DIALOG COMMANDS

DIALOGCREATE

DIALOGCREATE parent name title xpos ypos width height setup

This command will create a DIALOG window. A DIALOG window is used as the frame work to which you add other window objects or controls (sometimes called widget's). You can add things such as buttons, scrollbars, listboxes, etc. to the DIALOG window. This function is similar to WINDOWCREATE except it will not return to the caller until the Window is closed (see Modal vs. Modeless Windows).

parent: (WORD) Is the name of the DIALOG window that is to own this new DIALOG window. If this is the first window created use "root as the parent name.

name: (WORD) Is used to identify this DIALOG window (perhaps as the parent of another future window or control) and MUST be unique.

title: (LIST) Is used as the title (caption) of the DIALOG window.

xpos: (INTEGER) Is the X position you wish to place the upper left corner of the new DIALOG window.

ypos: (INTEGER) Is the Y position you wish to place the upper left corner of the new DIALOG window.

width: (INTEGER) Is the width of the new DIALOG window.

height: (INTEGER) Is the height of the new DIALOG window.

setup: (LIST) Is a (short) list of logo commands (or a procedure name) to execute when the DIALOG window is created. The commands to be executed should most likely be other somethingCREATE functions to add controls to the window. The reason is, is that DIALOGCREATE will not return until the window is closed (the caller looses control and cannot add controls). So be sure to add an (OK, END, CLOSE, CANCEL or whatever) button that will call DIALOGDELETE on this window.

Example:

```
to mysetup
buttoncreate "mydialog "myok "OK 25 25 50 50 [dialogdelete "mydialog]
end
dialogcreate "main "mydialog "mytitle 0 0 100 100 [mysetup]
```

DIALOGDELETE

DIALOGDELETE name

This command will delete (close) the DIALOG window with the given name. Note all the child windows and controls will be also deleted.

name: (WORD) Is used to identify the DIALOG window you want destroyed.

Example:

```
to mysetup
buttoncreate "mydialog "myok "OK 25 25 50 50 [dialogdelete "mydialog]
end
dialogcreate "main "mydialog "mytitle 0 0 100 100 [mysetup]
```

LISTBOX COMMANDS

LISTBOXCREATE

LISTBOXCREATE parent name xpos ypos width height

This command will create a LISTBOX control. A LISTBOX control is used to give the user a selection of items.

parent: (WORD) Is the name of the DIALOG window that is to own this new LISTBOX control.

name: (WORD) Is used to identify this LISTBOX control.

xpos: (INTEGER) Is the X position you wish to place the upper left corner of the new LISTBOX control.

ypos: (INTEGER) Is the Y position you wish to place the upper left corner of the new LISTBOX control.

width: (INTEGER) Is the width of the new LISTBOX control.

height: (INTEGER) Is the height of the new LISTBOX control.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
listboxcreate "mywindow "mylist 25 25 50 50
windowdelete "mywindow
```

LISTBOXDELETE

LISTBOXDELETE name

This command will delete (close) the LISTBOX control with the given name.

name: (WORD) Is used to identify the LISTBOX control you want destroyed.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
```

```
listboxcreate "mywindow "mylist 25 25 50 50
listboxdelete "mylist
windowdelete "mywindow
```

LISTBOXGETSELECT

LISTBOXGETSELECT name

This command will solicit (ask) the LISTBOX control for the selected item and output a copy of it.

output: (LIST) Represents the selected item of the LISTBOX control.

name: (WORD) Is used to identify the LISTBOX you wish to solicit.

Example:

```
to dodraw
cs
if equalp "TRIANGLE listboxgetselect "mylist [repeat 3 [fd 100 rt 120]]
if equalp "SQUARE listboxgetselect "mylist [repeat 4 [fd 100 rt 90]]
end
windowcreate "main "mywindow "mytitle 0 0 100 100
listboxcreate "mywindow "mylist 25 0 50 50
listboxaddstring "mylist [TRIANGLE]
listboxaddstring "mylist [SQUARE]
buttoncreate "mywindow "mydraw "Draw 25 50 50 25 [dodraw]
<select figure from listbox and then click on Draw button>
windowdelete "mywindow
```

LISTBOXADDSTRING

LISTBOXADDSTRING name item

This command will add the "item" to the LISTBOX control with the given "name".

name: (WORD) Is used to identify the LISTBOX control you wish to add a string to.

item: (WORD) Is the item you wish to insert into the LISTBOX control.

Example:

```
to dodraw
```

```
cs
if equalp "TRIANGLE listboxgetselect "mylist [repeat 3 [fd 100 rt 120]]
if equalp "SQUARE listboxgetselect "mylist [repeat 4 [fd 100 rt 90]]
end
windowcreate "main "mywindow "mytitle 0 0 100 100
listboxcreate "mywindow "mylist 25 0 50 50
listboxaddstring "mylist [TRIANGLE]
listboxaddstring "mylist [SQUARE]
buttoncreate "mywindow "mydraw "Draw 25 50 50 25 [dodraw]
<select figure from listbox and then click on Draw button>
windowdelete "mywindow
```

LISTBOXDELETESTRING

LISTBOXDELETESTRING name index

This command will delete an item at "index" of the LISTBOX control.

name: (WORD) Is used to identify the LISTBOX control you want to delete a string from.

index: (INTEGER) Index of item you wish deleted (starting at 0).

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
listboxcreate "mywindow "mylist 25 0 50 50
listboxaddstring "mylist [TRIANGLE]
listboxaddstring "mylist [SQUARE]
listboxaddstring "mylist [HEXAGON]
listboxdeletestring "mylist 1
windowdelete "mywindow
```

COMBOBOX COMMANDS

COMBOBOXCREATE

COMBOBOXCREATE parent name xpos ypos width height

This command will create a COMBOBOX control. A COMBOBOX control is used to give the user a selection of items and allow the user to enter a selection not listed. A COMBOBOX is two controls in one (A LISTBOX control and an EDIT control). If you

wish to create an EDIT control (a COMBOBOX without a LISTBOX) just set the height to a size in which the LISTBOX doesn't fit.

parent: (WORD) Is the name of the DIALOG window that is to own this new COMBOBOX control.

name: (WORD) Is used to identify this COMBOBOX control and MUST be unique.

xpos: (INTEGER) Is the X position you wish to place the upper left corner of the new COMBOBOX control.

ypos: (INTEGER) Is the Y position you wish to place the upper left corner of the new COMBOBOX control.

width: (INTEGER) Is the width of the new COMBOBOX control.

height: (INTEGER) Is the height of the new COMBOBOX control.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
comboboxcreate "mywindow "mycombo 25 25 50 50
windowdelete "mywindow
```

COMBOBOXDELETE

COMBOBOXDELETE name

This command will delete (close) the COMBOBOX control with the given name.

name: (WORD) Is used to identify the COMBOBOX you want destroyed.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
comboboxcreate "mywindow "mycombo 25 25 50 50
comboboxdelete "mycombo
windowdelete "mywindow
```

COMBOBOXGETTEXT

COMBOBOXGETTEXT name

This command will solicit (ask) the COMBOBOX, for the contents of the EDIT control portion of the COMBOBOX (which may or may not be a selected item).

output: (LIST) Represents the EDIT control contents in the COMBOBOX control.

name: (WORD) Is used to identify the COMBOBOX control you wish to solicit.

Example:

```
to dodraw
cs
make "sides first comboboxgettext "mycombo
repeat :sides [fd 50 rt 360.0/:sides]
end
windowcreate "main "mywindow "mytitle 0 0 100 100
comboboxcreate "mywindow "mycombo 25 0 50 50
comboboxaddstring "mycombo [3]
comboboxaddstring "mycombo [4]
comboboxaddstring "mycombo [5]
comboboxaddstring "mycombo [6]
buttoncreate "mywindow "mydraw "Draw 25 50 50 25 [dodraw]
<select or enter number of sides from combobox and then click on Draw button>
windowdelete "mywindow
```

COMBOBOXSETTEXT

COMBOBOXSETTEXT name text

This command will set the contents of the (EDIT control component of) COMBOBOX with "text".

name: (WORD) Is used to identify the COMBOBOX control you wish to SETTEXT to.

text: (WORD) Is the item you wish to insert into the (EDIT control component of) COMBOBOX control.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
comboboxcreate "mywindow "mycombo 25 0 50 50
comboboxaddstring "mycombo [3]
comboboxaddstring "mycombo [4]
comboboxaddstring "mycombo [5]
comboboxaddstring "mycombo [6]
comboboxsettext "mycombo [3]
```

windowdelete "mywindow

COMBOBOXADDSTRING

COMBOBOXADDSTRING name item

This command will add the "item" to the COMBOBOX with the given "name". Note that items in the LISTBOX are automatically sorted as they are inserted.

name: (WORD) Is used to identify the COMBOBOX you wish to add to.

item: (LIST) Is the item you wish to insert into the (LISTBOX component of) COMBOBOX.

Example:

```
to dodraw
cs
make "sides first comboboxgettext "mycombo
repeat :sides [fd 50 rt 360.0/:sides]
end
windowcreate "main "mywindow "mytitle 0 0 100 100
comboboxcreate "mywindow "mycombo 25 0 50 50
comboboxaddstring "mycombo [3]
comboboxaddstring "mycombo [4]
comboboxaddstring "mycombo [5]
comboboxaddstring "mycombo [6]
buttoncreate "mywindow "mydraw "Draw 25 50 50 25 [dodraw]
<select or enter number of sides from combobox and then click on Draw button>
windowdelete "mywindow
```

COMBOBOXDELETESTRING

COMBOBOXDELETESTRING name index

This command will delete an item at "index" of the COMBOBOX with the given "name".

name: (WORD) Is used to identify the COMBOBOX you want to delete string from.

index: (INTEGER) Index of item you wish deleted (starting at 0).

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
comboboxcreate "mywindow "mycombo 25 0 50 50
comboboxaddstring "mycombo [3]
comboboxaddstring "mycombo [4]
comboboxaddstring "mycombo [5]
comboboxaddstring "mycombo [6]
comboboxdeletestring "mycombo 1
windowdelete "mywindow
```

SCROLLBAR COMMANDS

SCROLLBARCREATE

SCROLLBARCREATE parent name xpos ypos width height callback

This command will create a SCROLLBAR control. A SCROLLBAR control is used to solicit, from the user, a variable value (although you can map its function to anything you desire). It is also common to link it to a STATIC control to inform the user of its setting (but this is not required). You must also not forget to set the SCROLLBAR range and initial value using SCROLLBARSET.

The orientation (vertical or horizontal) of the SCROLLBAR is determined by the longest dimension. That is, if X > Y then horizontal is chosen otherwise vertical is chosen.

parent: (WORD) Is the name of the window that is to own this new SCROLLBAR control.

name: (WORD) Is used to identify this SCROLLBAR control and MUST be unique.

xpos: (INTEGER) Is the X position you wish to place the upper left corner of the new SCROLLBAR control.

ypos: (INTEGER) Is the Y position you wish to place the upper left corner of the new SCROLLBAR control.

width: (INTEGER) Is the width of the new SCROLLBAR control.

height: (INTEGER) Is the height of the new SCROLLBAR control.

callback:(LIST) Is a (short) list of logo commands (or a procedure name) to execute when the user adjusts the SCROLLBAR. It is common to call a procedure that informs the user of what the SCROLLBAR state is.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
scrollbarcreate "mywindow "myscroll 25 25 50 25 [setheading scrollbarget "myscroll]
scrollbarset "myscroll 0 360 0
<try moving scrollbar and notice what happens to the turtle>
windowdelete "mywindow
```

SCROLLBARDELETE

SCROLLBARDELETE name

This command will delete (close) the SCROLLBAR control with the given name.

name: (WORD) Is used to identify the SCROLLBAR control you want destroyed.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
scrollbarcreate "mywindow "myscroll 25 25 50 25 []
scrollbardelete "myscroll
windowdelete "mywindow
```

SCROLLBARSET

SCROLLBARSET name lorange hirange position

This command will set the output range and current position of the SCROLLBAR control. You can issue a SCROLLBARSET as many times as you want.

name: (WORD) Is used to identify the SCROLLBAR control you want to set.

lorange:(INTEGER) Is used as the low range of the output values of the SCROLLBAR control.

hirange:(INTEGER) Is used as the high range of the output values of the SCROLLBAR control.

position:(INTEGER) Is used to set the current position (and output value) of the SCROLLBAR control.

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100
scrollbarcreate "mywindow "myscroll 25 25 50 25 [setheading scrollbarget "myscroll]
scrollbarset "myscroll 0 360 0
<try moving scrollbar and notice what happens to the turtle>
windowdelete "mywindow

SCROLLBARGET

SCROLLBARGET name

This command will output the position of the SCROLLBAR control of the given name.

output: (INTEGER) Is the position of the scrollbar (always within the SET range).

name: (WORD) Is used to identify the SCROLLBAR control you wish to solicit.

Example:

windowcreate "main "mywindow "mytitle 0 0 100 100
scrollbarcreate "mywindow "myscroll 25 25 50 25 [setheading scrollbarget "myscroll]
scrollbarset "myscroll 0 360 0
<try moving scrollbar and notice what happens to the turtle>
windowdelete "mywindow

BUTTON COMMANDS

BUTTONCREATE

BUTTONCREATE parent name label xpos ypos width height callback

This command will create a BUTTON control. A BUTTON control is used to allow the user to trigger events. The only function of the BUTTON control is to execute the "callback" list.

parent: (WORD) Is the name of the window that is to own this new BUTTON control.

name: (WORD) Is used to identify this BUTTON control.

label: (LIST) Is used as the label of the BUTTON control.

xpos: (INTEGER) Is the X position you wish to place the upper left corner of the new BUTTON control.

ypos: (INTEGER) Is the Y position you wish to place the upper left corner of the new BUTTON control.

width: (INTEGER) Is the width of the new BUTTON control.

height: (INTEGER) Is the height of the new BUTTON control.

callback:(LIST) Is a (short) list of logo commands (or a procedure name) to execute when the user clicks on the BUTTON.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
buttoncreate "mywindow "myleft "Left 25 25 25 25 [fd 2 lt 1]
buttoncreate "mywindow "myright "Right 50 25 25 25 [fd 2 rt 1]
<click left or right repeatedly and watch the turtle>
windowdelete "mywindow
```

BUTTONDELETE

BUTTONDELETE name

This command will delete (close) the BUTTON control with the given name.

name: (WORD) Is used to identify the BUTTON you want destroyed.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
buttoncreate "mywindow "mybutton "PUSH 25 25 25 25 [print "pushed]
buttoncreate "mybutton
windowdelete "mywindow
```

STATIC COMMANDS

STATICCREATE

STATICCREATE parent name text xpos ypos width height

This command will create a STATIC control. A STATIC control is used to simply display text. The name can be a bit misleading. In that a STATIC control can be very

dynamic by using the STATICUPDATE command.

parent: (WORD) Is the name of the DIALOG window that is to own this new STATIC control.

name: (WORD) Is used to identify this STATIC control and MUST be unique.

text: (LIST) Is used as the (perhaps initial) contents of the STATIC control.

xpos: (INTEGER) Is the X position you wish to place the upper left corner of the new STATIC control.

ypos: (INTEGER) Is the Y position you wish to place the upper left corner of the new STATIC control.

width: (INTEGER) Is the width of the new STATIC control.

height: (INTEGER) Is the height of the new STATIC control.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
staticcreate "mywindow "mystatic [Heading=0] 25 25 50 25
repeat 360 [rt 1 staticupdate "mystatic se [Heading=] heading]
windowdelete "mywindow
```

STATICDELETE

STATICDELETE name

This command will delete (close) the STATIC control with the given name.

name: (WORD) Is used to identify the STATIC control you want destroyed.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
staticcreate "mywindow "mystatic [This is Static] 25 25 50 25
staticcreate "mystatic
windowdelete "mywindow
```

STATICUPDATE

STATICUPDATE name text

This command will replace the contents of the STATIC control with "text".

name: (WORD) Is used to identify the STATIC control you want to update.

text: (LIST) Is used as the new contents of the STATIC control.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
staticcreate "mywindow "mystatic [Heading=0] 25 25 50 25
repeat 360 [rt 1 staticupdate "mystatic se [Heading=] heading]
windowdelete "mywindow
```

GROUPBOX COMMANDS

GROUPBOXCREATE

GROUPBOXCREATE parent name xpos ypos width height

This command will create a GROUPBOX control. A GROUPBOX control is a unique control compared with most other Windows functions. It is unique because all it does is group RadioButtons (RADIOBUTTONCREATE) and CheckBoxs (CHECKBOXCREATE) both graphically and logically. RadioButtons and CheckBoxs must belong to a GROUPBOX. Also realize that RadioButtons and CheckBoxs placed in the GROUPBOX still use the parents origin NOT the GROUPBOX origin.

parent: (WORD) Is the name of the DIALOG window that is to own this new GROUPBOX control.

name: (WORD) Is used to identify this GROUPBOX control and MUST be unique.

xpos: (INTEGER) Is the X position you wish to place the upper left corner of the new GROUPBOX control.

ypos: (INTEGER) Is the Y position you wish to place the upper left corner of the new GROUPBOX control.

width: (INTEGER) Is the width of the new GROUPBOX control.

height: (INTEGER) Is the height of the new GROUPBOX control.

Example:

```
to checkonthings
ifelse checkboxget "myhideturtle [ht] [st]
end
windowcreate "main "mywindow "mytitle 0 0 100 100
groupboxcreate "mywindow "mygroupbox 10 10 80 40
checkboxcreate "mywindow "mygroupbox "myhideturtle [Hide Turtle] 20 20 60 20
buttoncreate "mywindow "mybutton "GO 40 50 25 25 [checkonthings]
windowdelete "mywindow
```

GROUPBOXDELETE

GROUPBOXDELETE name

This command will delete (close) the GROUPBOX control with the given name.

name: (WORD) Is used to identify the GROUPBOX you want destroyed.

```
windowcreate "main "mywindow "mytitle 0 0 100 100
groupboxcreate "mywindow "mygroupbox 10 10 80 40
groupboxdelete "mygroupbox
windowdelete "mywindow
```

CHECKBOX COMMANDS

CHECKBOXCREATE

CHECKBOXCREATE parent group name label xpos ypos width height

This command will create a CHECKBOX control. A CHECKBOX control is used to give the user a selection of a two state (True or False) item. A CHECKBOX must also be associated with a GROUPBOX (GROUPBOXCREATE).

parent: (WORD) Is the name of the DIALOG window that is to own this new CHECKBOX control.

group: (WORD) Is the name of the GROUPBOX control that is to be associated with this new CHECKBOX control.

name: (WORD) Is used to identify this COMBOBOX control and MUST be unique.

label: (LIST) Is used as the label of this new CHECKBOX control.

xpos: (INTEGER) Is the X position you wish to place the upper left corner of the new CHECKBOX control.

ypos: (INTEGER) Is the Y position you wish to place the upper left corner of the new CHECKBOX control.

width: (INTEGER) Is the width of the new CHECKBOX control.

height: (INTEGER) Is the height of the new CHECKBOX control.

Example:

```
to checkonthings
ifelse checkboxget "myhideturtle [ht] [st]
end
windowcreate "main "mywindow "mytitle 0 0 100 100
groupboxcreate "mywindow "mygroupbox 10 10 80 40
checkboxcreate "mywindow "mygroupbox "myhideturtle [Hide Turtle] 20 20 60 20
buttoncreate "mywindow "mybutton "GO 40 50 25 25 [checkonthings]
windowdelete "mywindow
```

CHECKBOXDELETE

CHECKBOXDELETE name

This command will delete (close) the CHECKBOX control with the given name.

name: (WORD) Is used to identify the CHECKBOX you want destroyed.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
groupboxcreate "mywindow "mygroupbox 10 10 80 40
checkboxcreate "mywindow "mygroupbox "mycheckbox [Check Me] 20 20 60 20
checkboxdelete "mycheckbox
windowdelete "mywindow
```

CHECKBOXGET

CHECKBOXGET name

This command will solicit (ask) the CHECKBOX, for its state (True or False).

output: (SPECIAL) Represents the state (True or False) of the CHECKBOX control.

name: (WORD) Is used to identify the COMBOBOX control you wish to solicit.

Example:

```
to checkonthings
ifelse checkboxget "myhideturtle [ht] [st]
end
windowcreate "main "mywindow "mytitle 0 0 100 100
groupboxcreate "mywindow "mygroupbox 10 10 80 40
checkboxcreate "mywindow "mygroupbox "myhideturtle [Hide Turtle] 20 20 60 20
buttoncreate "mywindow "mybutton "GO 40 50 25 25 [checkonthings]
windowdelete "mywindow
```

CHECKBOXSET

CHECKBOXSET name state

This command will set the state of the CHECKBOX with state (True or False).

name: (WORD) Is used to identify the CHECKBOX control you wish to SET to.

state: (WORD) Is the state you wish to set the CHECKBOX control to.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
groupboxcreate "mywindow "mygroupbox 10 10 80 40
checkboxcreate "mywindow "mygroupbox "mycheckbox [Check Me] 20 20 60 20
checkboxset "mycheckbox "true
checkboxset "mycheckbox "false
windowdelete "mywindow
```

RADIOBUTTON COMMANDS

RADIOBUTTONCREATE

RADIOBUTTONCREATE parent group name label xpos ypos width height

This command will create a RADIOBUTTON control. A RADIOBUTTON control is used to give the user a selection of a two state (True or False) item. But along with this the user will be restricted to only have one RADIOBUTTON set True within a GROUPBOX at any given time. A RADIOBUTTON must also be associated with a GROUPBOX (GROUPBOXCREATE).

parent: (WORD) Is the name of the DIALOG window that is to own this new RADIOBUTTON control.

group: (WORD) Is the name of the GROUPBOX control that is to be associated with this new RADIOBUTTON control.

name: (WORD) Is used to identify this RADIOBUTTON control and MUST be unique.

label: (LIST) Is used as the label of this new RADIOBUTTON control.

xpos: (INTEGER) Is the X position you wish to place the upper left corner of the new RADIOBUTTON control.

ypos: (INTEGER) Is the Y position you wish to place the upper left corner of the new RADIOBUTTON control.

width: (INTEGER) Is the width of the new RADIOBUTTON control.

height: (INTEGER) Is the height of the new RADIOBUTTON control.

Example:

```
to checkonthings
ifelse radiobuttonget "myhideturtle [ht] [st]
end
windowcreate "main "mywindow "mytitle 0 0 100 100
groupboxcreate "mywindow "mygroupbox 10 10 80 60
radiobuttoncreate "mywindow "mygroupbox "myhideturtle [Hide Turtle] 20 20 60 20
radiobuttoncreate "mywindow "mygroupbox "myshowturtle [Show Turtle] 20 40 60 20
radiobuttonset "myhideturtle "true
radiobuttonset "myshowturtle "false
buttoncreate "mywindow "mybutton "GO 40 70 25 20 [checkonthings]
windowdelete "mywindow
```

RADIOBUTTONDELETE

RADIOBUTTONDELETE name

This command will delete (close) the RADIOBUTTON control with the given name.

name: (WORD) Is used to identify the RADIOBUTTON you want destroyed.

Example:

```
windowcreate "main "mywindow "mytitle 0 0 100 100
groupboxcreate "mywindow "mygroupbox 10 10 80 60
radiobuttoncreate "mywindow "mygroupbox "myradiobutton [Switch Me] 20 20 60 20
radiobuttondelete "myradiobutton
windowdelete "mywindow
```

RADIOBUTTONGET

RADIOBUTTONGET name

This command will solicit (ask) the RADIOBUTTON, for its state (True or False).

output: (SPECIAL) Represents the state (True or False) of the RADIOBUTTON control.

name: (WORD) Is used to identify the RADIOBUTTON control you wish to solicit.

Example:

```
to checkonthings
ifelse radiobuttonget "myhideturtle [ht] [st]
end
windowcreate "main "mywindow "mytitle 0 0 100 100
groupboxcreate "mywindow "mygroupbox 10 10 80 60
radiobuttoncreate "mywindow "mygroupbox "myhideturtle [Hide Turtle] 20 20 60 20
radiobuttoncreate "mywindow "mygroupbox "myshowturtle [Show Turtle] 20 40 60 20
radiobuttonset "myhideturtle "true
radiobuttonset "myshowturtle "false
buttoncreate "mywindow "mybutton "GO 40 70 25 20 [checkonthings]
windowdelete "mywindow
```

RADIOBUTTONSET

RADIOBUTTONSET name state

This command will set the state of the RADIOBUTTON with state (True or False). Note that even though the user can only have one RADIOBUTTON set True at any given time this can be violated through this command. If you choose to use this command you must maintain a correct state. That is, if you set a RADIOBUTTON True make sure you set

all the other RADIOBUTTONs within the GROUPBOX to False.

name: (WORD) Is used to identify the RADIOBUTTON control you wish to SET to.

state: (SPECIAL) Is the state (True or False) you wish to set the RADIOBUTTON control to.

Example:

```
to checkonthings
ifelse radiobuttonget "myhideturtle [ht] [st]
end
windowcreate "main "mywindow "mytitle 0 0 100 100
groupboxcreate "mywindow "mygroupbox 10 10 80 60
radiobuttoncreate "mywindow "mygroupbox "myhideturtle [Hide Turtle] 20 20 60 20
radiobuttoncreate "mywindow "mygroupbox "myshowturtle [Show Turtle] 20 40 60 20
radiobuttonset "myhideturtle "true
radiobuttonset "myshowturtle "false
buttoncreate "mywindow "mybutton "GO 40 70 25 20 [checkonthings]
windowdelete "mywindow
```

DEBUG COMMANDS

DEBUGWINDOWS

DEBUGWINDOWS name

This command will print the tree (window hierarchy) starting at the window called "name".

name: (WORD) Is used to identify the root Window you wish print.

```
windowcreate "main "mywindow "mytitle 0 0 100 100
listboxcreate "mywindow "mylist 25 0 50 50
buttoncreate "mywindow "mydraw "Draw 25 50 50 25 [Print "Click]
debugwindows "mywindow
Window mywindow
  Button mydraw
  ListBox mylist
windowdelete "mywindow
```

Modal vs. Modeless Windows

Windows programming supports two modes, Modal and Modeless. The Modal mode (DIALOGCREATE) is similar to a non-windows programming model (the application is in control). Similar, in that, in midstream of processing you, as the programmer, decide to prompt the user for information (e.g., readlist). That is, processing is halted until the information is acquired and other components of the application are inaccessible to the user. For example prompting the user for a file name to open a document is Modal.

In the Modeless mode (WINDOWCREATE) the tables are turned, the Window (user) is in control. For example the commander in LOGO is Modeless. This takes some getting used to but is a very important idea. The program is now idle while. The application executes when the user triggers and event (such as pushing a button).

Full custom Windows are available in Modeless and Modal mode. Besides READLIST and READCHAR the following Built-in Modal windows are available.

Predfined Windows

Predefined windows allows you access to several very common dialogs that are available under Windows.

MESSAGEBOX

MESSAGEBOX banner body

This command will stop processing and popup a message window using banner and body. Processing will not continue until the user clicks on the OK button. Also note the LOGO commander is also disabled until OK is clicked.

banner:(LIST) Is used label the banner of the window.

body: (LIST) Is used to fill the message box with the given text. The box will automatically be sized.

Example:

messagebox [This is the banner] [This is the body]

DIALOGFILEOPEN

DIALOGFILEOPEN filename

This command will pop up a standard "FILE OPEN" windows dialog box. It does NOT open a file it just allows the user to select the desired file using a Graphical User Interface. The filename argument is used to set the defaults of name, directory and/or extension. The output is the fully specified file name that the user selected. In the case that the user selects
cancel the output will be an empty list ([]).

filename: (WORD) Specified the default directory, filename and/or extension of the filename. ("*" are permitted as the filename or extension).

Output: (LIST) The fully specified filename (includes drive and directory).

Example:

show dialogfileopen "c:\\logo\\examples*.me
<This will list all files ending in ".me" in the specified directory>
[c:\logo\examples\read.me]

DIALOGFILESAVE

DIALOGFILESAVE filename

This command will pop up a standard "FILE SAVE" windows dialog box. It does NOT save a file it just allows the user to select the desired file using a Graphical User Interface. The filename argument is used to set the defaults of name, directory and/or extension. The output is the fully specified file name the user selected. In the case that the user selects
cancel the output will be an empty list ([]).

filename: (WORD) Specified the default directory, filename and/or extension of the filename. ("*" are permitted as the filename or extension).

Output: (LIST) The fully specified filename (includes drive and directory).

Example:

show dialogfilesave "c:\\logo\\examples*.me
<This will list all files ending in ".me" in the specified directory>
[c:\logo\examples\read.me]

WINDOWFILEEDIT

WINDOWFILEEDIT filename callback

This command will pop up a standard editor on the given filename. When the user exits the editor the contents of callback will be executed.

filename: (WORD) Specifies the file to be edited.

callback:(LIST) Is a (short) list of logo commands (or a procedure name) to execute when the user exits the editor.

Example:

windowfileedit dialogfileopen "c:\\logo\\examples*.me [Print "Done]

WINDOWS Example

Note: That for the modeless case "setup" is called AFTER the WINDOWCREATE returns. And that for the modal case "setup" is called DURING the DIALOGCREATE and DIALOGCREATE does not return until the window is closed.

```
to win
; For modeless example use this line
windowcreate "root "d1 [Draw Pictures] 0 0 150 110 setup  ;Create main window
; For modal example use this line
; dialogcreate "root "d1 [Draw Pictures] 0 0 150 110 [setup] ;Create main window
end

to setup
staticcreate "d1 "st4 [Select Shape] 5 10 50 10 ; Label the List box
listboxcreate "d1 "l1 5 25 80 40 ;Create List box with 3 Items owned by d1
listboxaddstring "l1 "SQUARE
listboxaddstring "l1 "TRIANGLE
listboxaddstring "l1 "HEXAGON

staticcreate "d1 "st11 [Red] 100 10 40 10 ;Label the scrollbar

scrollbarcreate "d1 "s1 100 25 10 50 [myred] ;Create scroll bar, call myred when clicked
scrollbarset "s1 1 255 125 myred ;Init

buttoncreate "d1 "b1 "END 5 80 40 10 [myend] ;Create button to call myend
buttoncreate "d1 "b3 "CLEAR 55 80 35 10 [cs] ;Create button to clear
buttoncreate "d1 "b2 "DRAW 100 80 35 10 [drawthing] ;Create button to call drawthing
```

```
end

; execute this routine when DRAW button pushed

to drawthing
setpencolor (list scrollbarget "s1 0 0)  ;Ask scrollbar what to setpencolor to

; Draw appropriate shape according to the listbox

if equalp "HEXAGON listboxgetselect "l1 [repeat 6 [fd 100 rt 60]]
if equalp "SQUARE listboxgetselect "l1 [repeat 4 [fd 100 rt 90]]
if equalp "TRIANGLE listboxgetselect "l1 [repeat 3 [fd 100 rt 120]]
end

; execute this routine when END button is pushed

to myend
; For modeless example use this
windowdelete "d1
; For modal example use this
; dialogdelete "d1
end

; execute this routine when RED scroll bar is adjusted

to myred
staticupdate "st11 sentence [Red] scrollbarget "s1 ;Update static label of position
end
```

BITMAP FUNCTIONS

Bitmap functions allow you to manipulate sub-images within the primary image.

BITMAP COMMANDS

BITCUT

BITCUT width height

This command will "cut" out part of the image and put it into Logo's memory (Clipboard if the "index" is 0). Later at anytime you can "paste" (BITPASTE) it back into the image. LOGO will cut starting at the turtles' position with a width of the first argument and a height of the second argument. See also SETBITINDEX.

Example:

```
cs
setpensize [2 2]
repeat 72 [repeat 4 [fd 100 rt 90] setpencolor (list repcount*3 0 0) rt 5]
pu
setxy -50 -50
bitcut 100 100
cs
pu
repeat 36 [fd 150 bitpaste bk 150 rt 10]
```

BITCOPY

BITCOPY width height

This command will "copy" part of the image and put it into Logo's memory (Clipboard if the "index" is 0). Later at anytime you can "paste" (BITPASTE) it back into the image. LOGO will copy starting at the turtles' position with a width of the first argument and a height of the second argument.
See also SETBITINDEX.

Example:

```
cs
setpensize [2 2]
repeat 72 [repeat 4 [fd 100 rt 90] setpencolor (list repcount*3 0 0) rt 5]
pu
setxy -50 -50
bitcopy 100 100
cs
pu
repeat 36 [fd 150 bitpaste bk 150 rt 10]
```

BITPASTE

BITPASTE

This command will "paste" back into the image what was "cut" (BITCUT) (or in the Clipboard if index is 0). LOGO will always "paste" at the location of the turtle with the turtle being the lower left corner. See also SETBITINDEX and SETBITMODE.

Example:

```
cs
setpensize [2 2]
repeat 72 [repeat 4 [fd 100 rt 90] setpencolor (list repcount*3 0 0) rt 5]
pu
setxy -50 -50
bitcut 100 100
cs
pu
repeat 36 [fd 150 bitpaste bk 150 rt 10]
```

BITFIT

BITFIT width height

This command will "fit" the currently "cut" (BITCUT or BITCOPY) image into the specified dimensions. Later at anytime you can "paste" (BITPASTE) it back into the image. LOGO will fit the "cut" image to a width of the first argument and a height of the second argument. The original "cut" image is replaced by this newly "fit" image. You can permanently "scale" your image with bitfit. Whereas zoom only views it temporarily at a different scale.

Example:

```
cs
setpensize [2 2]
repeat 72 [repeat 4 [fd 100 rt 90] setpencolor (list repcount*3 0 0) rt 5]
pu
setxy -50 -50
bitcut 100 100
cs
bitpaste
cs
bitfit 200 100
bitpaste
```

SETBITINDEX

SETBITINDEX index

This command sets the current bitmap cut buffer according to index. The index can range from 0 to 1023. Its purpose was to allow multiple images to be stored in memory ready for quick pasting in animation. The index of 0 is the default and also has the behavior of using the CLIPBOARD as the cut buffer. That is if you "cut" and image in PAINT you can paste it directly in MswLogo. The reverse is also true, if "cut" an image in MswLogo it is available to PAINT.

Example:

```
cs
setbitindex 0
repeat 3 [fd 50 rt 120]
bitcut 100 100
cs
setbitindex 1
repeat 4 [fd 50 rt 90]
bitcut 100 100
cs
setbitmode 3
pu
ht
repeat 72 [fd 50 bitpaste bk 50 rt 5]
setbitindex 0
repeat 72 [fd 100 bitpaste bk 100 rt 5]
pd
```

BITINDEX

BITINDEX

This command will output the current bitmap index set by SETBITINDEX.

Example:

setbitindex 99
show bitindex
99

SETBITMODE

SETBITMODE mode

This command sets the current bitmap mode according to mode. The mode can range from 1 to 9. Its purpose is to allow images to be pasted using different methods. Sometimes you want the background erased and sometimes not. Sometimes you wish to invert the image before pasting it and sometimes not. These are the 9 methods:

mode 1: Take COPY of Memory then COPY to Screen.
mode 2: Take COPY of Memory OR with COPY of Screen then COPY to Screen.
mode 3: Take COPY of Memory AND with COPY of Screen then COPY to Screen
mode 4: Take COPY of Memory XOR with COPY of Screen then COPY to Screen
mode 5: Take COPY of Memory AND with INVERT of Screen then COPY to Screen
mode 6: Take INVERT of Memory then COPY to Screen
mode 7: Take COPY of Memory OR with COPY of Screen then INVERT to Screen
mode 8: Take INVERT of Memory OR with COPY of Screen then COPY to Screen
mode 9: Take INVERT of Screen then COPY to Screen

Example:

```
cs
pu
repeat 9 [setfloodcolor (list repcount*25 repcount*25 repcount*25) bitblock 50 5 fd 5]
setxy 0 0
bitcut 50 50
rt 90
setfloodcolor [125 125 125]
setxy -250 -50
bitblock 550 150
setxy -200 0
repeat 9 [setbitmode repcount bitpaste fd 50]
```

BITMODE

BITMODE

This command will output the current bitmap mode set by SETBITMODE.

Example:

setbitmode 8
show bitmode
8

BITBLOCK

BITBLOCK width height

This command will draw an opaque rectangle of the given dimensions. The color will be the color of SETFLOODCOLOR.

Example:

bitblock 200 100
bitblock 100 200

BITLOAD

BITLOAD bitmapname

This command is the same as the Bitmap Load Command from the menu. Its one input must be a word that describes the bitmap file to load. See also BITSAVE command.

Example:

bitload "c:\\windows\\leaves.bmp

BITSAVE

BITSAVE bitmapname

This command is the same as the Bitmap Save Command from the menu. Its one input must be a word that describes the bitmap file to save. See also BITLOAD command.

Example:

```
repeat 72 [repeat 4 [fd 100 rt 90] rt 5]
bitsave "myfile.bmp
cs
bitload "myfile.bmp
```

MULTI MEDIA/MIDI/SOUND

Multi Media in Logo means that you, as a logo programmer, can manipulate Multi Media devices such as cdplayers, sound boards, and more.

MIDI COMMANDS

MIDIOPEN

MIDIOPEN
(MIDIOPEN id)

This command opens the MIDI device and accesses it through a MIDI device driver. The device driver chosen depends on several things. In form 1 (no arguments) the MIDI-MAPPER is attempted to be used. In form 2 (id argument specified) lets you choose any available MIDI driver available on your system. The "id" starts at 0 up to 1 less than number of MIDI drivers available. To determine which "id" maps to which driver try several MIDIOPEN commands with increasing "id". MIDIOPEN will output the name of the driver being used.

Basically MIDI allows you to generate sound on your sound card. You will need to install the appropriate drivers under Windows for your sound card to work. Most sound cards come with a MIDI player which basically reads MIDI messages from the file and passes them to the MIDI driver. If you have your MIDI player working under Windows then MswLogo should work too.

MIDI commands in MswLogo does not use MIDI files (.MID or .MDI). Instead you directly build up sequences of MIDI messages and send them directly to the MIDI device. Note, you can manipulate (play) MIDI files using the MCI command. MswLogo is not a MIDI sequencer. You can think of MIDI commands in MswLogo as a programmable keyboard. It just so happens that the link between your programmable keyboard (MIDI commands) and your Speaker is MIDI.

This is only available to Windows 3.1 or compatible systems. See also the MIDIMESSAGE and MIDICLOSE commands.

id: (INTEGER) Is an index that specifies which MIDI device driver you wish to open. When no id given it selects the MIDI MAPPER device driver.

Output: (LIST) When successfully opened outputs the name of the driver being used.

Example:

PRINT MIDIOPEN
[MIDI Mapper]
MIDICLOSE

MIDICLOSE

MIDICLOSE

This command closes the MIDI device. There are no inputs or outputs. See also the MIDIOPEN command.

Example:

MIDICLOSE

MIDIMESSAGE

MIDIMESSAGE [status data1 data2]
MIDIMESSAGE [status data1 data2 status data1 data2 ...]
MIDIMESSAGE [240 data1 data2 data3 data4 ...]

The input must be a list. You must already of issued a MIDIOPEN command to use this command. There are 3 forms of this command, the SHORT form, LONG form and the SYSTEM EXCLUSIVE form.

The SHORT form is the common form and always has a 3 integer list. The first integer is known as the STATUS BYTE (it can also be thought of as a COMMAND BYTE). It must be followed by 2 data bytes even if the message requires only 1 (just use 0).

The LONG form is similar to the SHORT form but integer list contains many SHORT messages (triples).

The SYSTEM EXCLUSIVE form must be led by the system exclusive status byte 240 (F0 hex). It can then be followed by any amount of data bytes.

See the MIDI TABLE which basically is a Specification of the MIDI Message.

This documentation is not attempt to teach you MIDI. But there is enough information here to hopefully get you started. For more information you may be interested in puchasing a book on MIDI such as:

MIDI BASICS by Akira Otsuka and Akihiko Nakajima.

Example:

```
PRINT MIDIOPEN
[MIDI Mapper]
MIDIMESSAGE [192+13 56 0 192+13 56 0]
MIDIMESSAGE [144+13 100 100]
MIDICLOSE
```

MIDI TABLE

COMMAND NAME	COMMAND CODE	DATA BYTE 1	DATA BYTE 2
Note Off	128 + **Channel**	0-127 **Pitch**	0-127 **Velocity**
Note On	144 + **Channel**	0-127 **Pitch**	0-127 **Velocity**
Poly Pressure	160 + **Channel**	0-127 **Pitch**	0-127 **Pressure**
Control Change	176 + **Channel**	0-127 MIDI Control	0-127 **MSB**
Program Change	192 + **Channel**	0-127 **Program**	Not used
Channel Pressure	208 + **Channel**	0-127 **Pressure**	Not used
Pitch Wheel	224 + **Channel**	0-127 **LSB**	0-127 **MSB**
System Exclusive	240	0-127 **Id Code**	Any number of bytes
Undefined	241	Not used	Not used
Song Posittion	242	0-127 **LSB**	0-127 **MSB**
Song Select	243	0-127 **Song**	Not used
Undefined	244	Not used	Not used
Undefined	245	Not used	Not used
Tune Request	246	Not used	Not used
End of Exclusive	247	Not used	Not used
Timing Clock	248	Not used	Not used
Undefined	249	Not used	Not used
Start	250	Not used	Not used
Continue	251	Not used	Not used
Stop	252	Not used	Not used
Undefined	253	Not used	Not used
Active Sensing	254	Not used	Not used
System Reset	255	Not used	Not used

See also the MIDI GLOSSARY

MIDI CONTROL

COMMAND NAME	COMMAND CODE	DATA BYTE 1	DATA BYTE 2

Control Change	176 + **Channel**	0 Undefined	0-127 **MSB**
Control Change	176 + **Channel**	1 Modulation Wheel	0-127 **MSB**
Control Change	176 + **Channel**	2 Breath Controller	0-127 **MSB**
Control Change	176 + **Channel**	3 After Touch	0-127 **MSB**
Control Change	176 + **Channel**	4 Foot Controller	0-127 **MSB**
Control Change	176 + **Channel**	5 Portamento Time	0-127 **MSB**
Control Change	176 + **Channel**	6 Data Entry	0-127 **MSB**
Control Change	176 + **Channel**	7 Main Volume	0-127 **MSB**
Control Change	176 + **Channel**	8-31 Undefined	0-127 **MSB**
Control Change	176 + **Channel**	32-63 **LSB** of 0-31	0-127 **LSB**
Control Change	176 + **Channel**	64 Damper Pedal	0:Off 127:On
Control Change	176 + **Channel**	65 Portamento	0:Off 127:On
Control Change	176 + **Channel**	66 Sostenuto	0:Off 127:On
Control Change	176 + **Channel**	67 Soft Pedal	0:Off 127:On
Control Change	176 + **Channel**	68-92 Undefined	0:Off 127:On
Control Change	176 + **Channel**	93 Chorus	0:Off 127:On
Control Change	176 + **Channel**	94 Celeste	0:Off 127:On
Control Change	176 + **Channel**	95 Phaser	0:Off 127:On
Control Change	176 + **Channel**	96 Data Entry + 1	0:Off 127:On
Control Change	176 + **Channel**	97 Data Entry - 1	0:Off 127:On
Control Change	176 + **Channel**	98-121 Undefined	0:Off 127:On
Control Change	176 + **Channel**	122 **Local Control**	0-127
Control Change	176 + **Channel**	123 All Notes Off	0
Control Change	176 + **Channel**	124 Omni Mode off	0-15
Control Change	176 + **Channel**	125 Omni Mode on	0
Control Change	176 + **Channel**	126 Mono on/Poly off	0
Control Change	176 + **Channel**	127 Poly on/Mono off	0

See also the MIDI GLOSSARY

MIDI GLOSSARY

Channel: A channel is a number from 0-15 which corresponds to channels 1-16.

Pitch: A pitch is a number from 0-127 and corresponds to a note on the instrument.

Velocity: A velocity is a number from 0-127 and corresponds to how fast the key (or string) is pressed or released (most terminoligy is in reference to keyboards). 0 means is released.

Pressure: A pressure is a number from 0-127 and corresponds to the characteristics of how the key is hit.

Program: A program is a number from 0-127 and corresponds to the instrument to use.

See the MIDI INSTRUMENT table.

MSB: Most Significant Bits.

LSB: Least Significant Bits.

Id Code: Manufactures Id Code. Used to enter System Exclusive Mode which is specific to the Manufacturer of the device.

Song: A song is a rhythm machine.

MIDI INSTRUMENT

Piano

0 - Acoustic Grand Piano
1 - Bright Acoustic Piano
2 - Electric Grand Piano
3 - Honky-tonk Piano
4 - Rhodes Piano
5 - Chorused Piano
6 - Harpsichord
7 - Clavinet

Chromatic Percussion

8 - Celesta
9 - Glockenspiel
10 - Music box
11 - Vibraphone
12 - Marimba
13 - Xylophone
14 - Tubular Bells
15 - Dulcimer

Organ

16 - Hammond Organ
17 - Percussive Organ
18 - Rock Organ
19 - Church Organ
20 - Reed Organ
21 - Accordian
22 - Harmonica
23 - Tango Accordian

Guitar

24 - Acoustic Guitar (nylon)
25 - Acoustic Guitar (steel)
26 - Electric Guitar (jazz)
27 - Electric Guitar (clean)
28 - Electric Guitar (muted)
29 - Overdriven Guitar
30 - Distortion Guitar
31 - Guitar Harmonics

Bass

32 - Acoustic Bass
33 - Electric Bass (finger)
34 - Electric Bass (pick)
35 - Fretless Bass
36 - Slap Bass 1
37 - Slap Bass 2
38 - Synth Bass 1
39 - Synth Bass 2

Strings

40 - Violin
41 - Viola
42 - Cello
43 - Contrabass
44 - Tremolo Strings
45 - Pizzicato Strings
46 - Orchestral Harp
47 - Timpani

Ensemble

48 - String Ensemble 1
49 - String Ensemble 2
50 - Synth Strings 1
51 - Synth Strings 2
52 - Choir Aahs
53 - Voice Oohs
54 - Synth Voice
55 - Orchestra Hit

Brass

56 - Trumpet
57 - Trombone
58 - Tuba
59 - Muted Trumpet
60 - French Horn
61 - Brass Section
62 - Synth Brass 1
63 - Synth Brass 2

Reed

64 - Soprano Sax
65 - Alto Sax
66 - Tenor Sax
67 - Baritone Sax
68 - Oboe
69 - English Horn
70 - Bassoon
71 - Clarinet

Pipe

72 - Piccolo
73 - Flute
74 - Recorder
75 - Pan Flute
76 - Bottle Blow
77 - Shakuhachi
78 - Whistle
79 - Ocarina

Synth Lead

80 - Lead 1 (square)
81 - Lead 2 (sawtooth)
82 - Lead 3 (caliope lead)
83 - Lead 4 (chiff lead)
84 - Lead 5 (charang)
85 - Lead 6 (voice)
86 - Lead 7 (fifths)
87 - Lead 8 (brass + lead)

Synth Pad

88 - Pad 1 (new age)

89 - Pad 2 (warm)
90 - Pad 3 (polysynth)
91 - Pad 4 (choir)
92 - Pad 5 (bowed)
93 - Pad 6 (metallic)
94 - Pad 7 (halo)
95 - Pad 8 (sweep)

Synth Effects

96 - FX 1 (rain)
97 - FX 2 (soundtrack)
98 - FX 3 (crystal)
99 - FX 4 (atmosphere)
100 - FX 5 (brightness)
101 - FX 6 (goblins)
102 - FX 7 (echoes)
103 - FX 8 (sci-fi)

Ethnic

104 - Sitar
105 - Banjo
106 - Shamisen
107 - Koto
108 - Kalimba
109 - Bagpipe
110 - Fiddle
111 - Shanai

Percussive

112 - Tinkle Bell
113 - Agogo
114 - Steel Drums
115 - Woodblock
116 - Taiko Drum
117 - Melodic Tom
118 - Synth Drum
119 - Reverse Cymbal

Sound Effects

120 - Guitar Fret Noise
121 - Breath Noise
122 - Seashore

123 - Bird Tweet
124 - Telephone Ring
125 - Helicopter
126 - Applause
127 - Gunshot

SOUND COMMANDS

SOUND

SOUND [frequency duration]
SOUND [frequency duration frequency duration ...]

The input must be a list of pairs. Each pair specifies a frequency (in hertz) and duration (clock ticks). MswLogo cannot yield to other applications while a sound vector is being played. The larger the frequency the high the pitch. The large the duration the longer the sound. The sound will only come out the PC speaker and will work on all Windows systems.

Example:

sound [100 200]

Example:

make "zing []
repeat 50 [make "zing lput repcount :zing make "zing lput 100 :zing]
sound :zing

MULTI MEDIA COMMANDS

MCI

MCI [mci-command-list]
(MCI [mci-command-list **notify**] callback)

The input must be a list. It may or may not output a list depending on the context. The MCI interface is very powerful. It opens the door to letting Logo control any Windows Multi Media device. These include Sound cards (with or without MIDI interfaces), CD-ROM players and more.

The MCI command is designed to let YOU (the programmer) write procedures to manipulate Multi Media devices. You can now link sounds to the steps of drawing a picture. You can narrate your own slide show. You can even ask your user questions in your own voice.

The mci-command-list is described in a separate help file. See Help MCI Command.

callback:(LIST) Is a (short) list of logo commands (or a procedure name) to execute when the MCI command completes.

Output: (LIST) sometimes an output from MCI and sometimes nothing depending on the command.

Current limitations:

The MCI interface allows you to start a device and optionally "wait" for it to finish or "notify" you when it has finished the request.

Example:

```
to soundit
print mci [open c:\windows\tada.wav type waveaudio alias wa1]
print mci [open c:\windows\ding.wav type waveaudio alias wa2]
mci [seek wa1 to start]
mci [play wa1 wait]
repeat 2~
  [~
  mci [seek wa2 to start]~
  mci [play wa2 wait]~
  ]
mci [close wa1]
mci [close wa2]
end
```

Note: That the **Microsoft SPEAKER** sound card emulator does NOT work with MCI.
Note: MCI is only available to **Microsoft Windows 3.1** systems or **Microsoft Windows 3.0** with Multi Media extensions.

ABBREVIATIONS

This section is primarily here for On-line HELP.

ABBREVIATION LIST

+

This is an abbreviation for sum.

bk

This is an abbreviation for back.

bfs

This is an abbreviation for butfirsts.

bf

This is an abbreviation for butfirst.

bl

This is an abbreviation for butlast.

cs

This is an abbreviation for clearscreen.

ct

This is an abbreviation for cleartext.

co

This is an abbreviation for continue.

/

This is an abbreviation for quotient.

ed

This is an abbreviation for edit.

=

This is an abbreviation for equalp.

er

This is an abbreviation for erase.

erf

This is an abbreviation for erasefile.

fd

This is an abbreviation for forward.

fs

This is an abbreviation for fullscreen.

>

This is an abbreviation for greaterp.

ht

This is an abbreviation for hideturtle.

iff

This is an abbreviation for iffalse.

ift

This is an abbreviation for iftrue.

lt

This is an abbreviation for left.

<

This is an abbreviation for lessp.

*

This is an abbreviation for product.

op

This is an abbreviation for output.

pd

This is an abbreviation for pendown.

pe

This is an abbreviation for penerase.

ppt

This is an abbreviation for penpaint.

px

This is an abbreviation for penreverse.

pu

This is an abbreviation for penup.

pr

This is an abbreviation for print.

rc

This is an abbreviation for readchar.

rcs

This is an abbreviation for readchars.

rl

This is an abbreviation for readlist.

rw

This is an abbreviation for readword.

rt

This is an abbreviation for right.

se

This is an abbreviation for sentence.

setfc

This is an abbreviation for setfloodcolor.

seth

This is an abbreviation for setheading.

setpc

This is an abbreviation for setpencolor.

setsc

This is an abbreviation for setscreencolor.

st

This is an abbreviation for showturtle.

ss

This is an abbreviation for splitscreen.

-

This is an abbreviation for difference.

ts

This is an abbreviation for textscreen.

END OF DOCUMENT