PC Logo

PC Logo for Windows On-Line Help
©1994 Harvard Associates, Inc.
10 Holworthy Street
Cambridge, MA 02138
U.S.A.

voice: (617) 492-0660
fax: (617) 492-4610
CompuServe 70312,243
Internet 70312.243@compuserve.com
toll-free fax within USA: (800) 776-4610

Logo Environment

Logo Fundamentals

Logo Programming

List of commands

Error messages

The Logo Environment

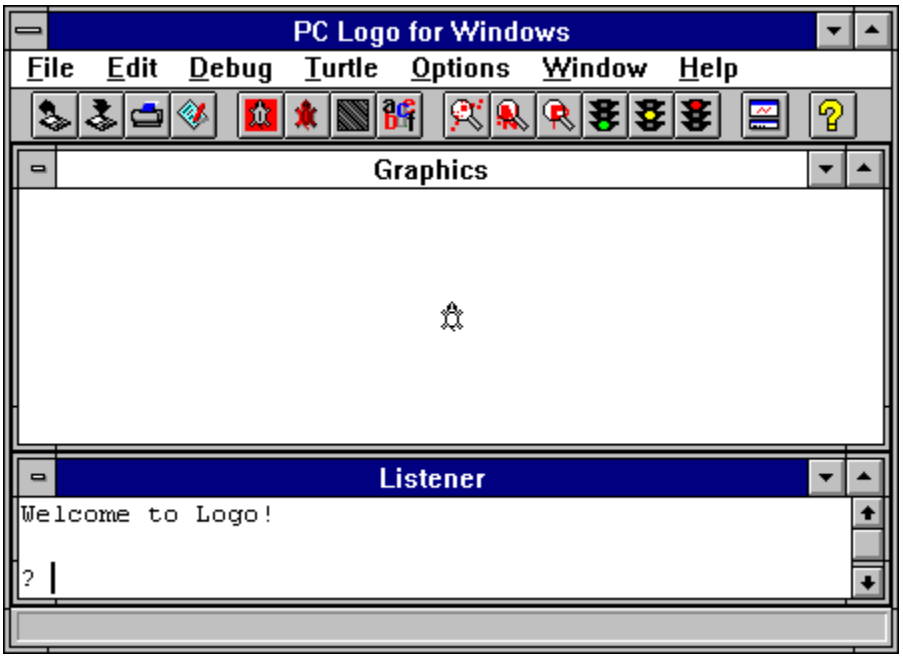
When you start PC Logo for Windows by running the program from the Windows Program Manager, you enter the Logo environment. The Logo environment contains common Windows elements, including windows, pull-down menus, and buttons. Logo also provides access to all the parts of your computer, including not only memory and the monitor, but the mouse, disk drive(s) and printer as well. This chapter describes the various elements of the Logo environment that you see when you start Logo.

 The Button bar

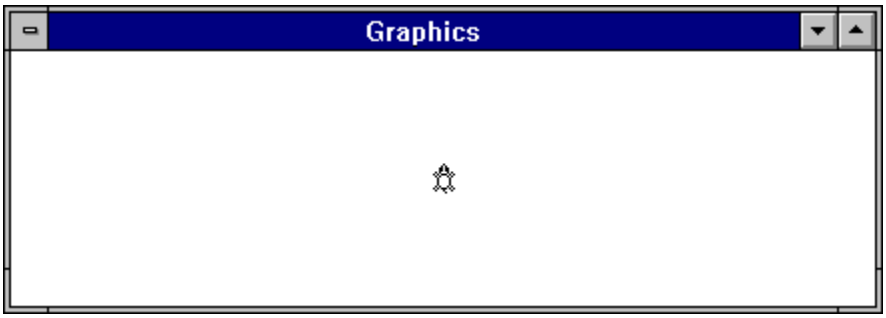
 The file LOGO.INI

Standard Layout

When you first begin Logo, the windows are arranged in "standard layout" or the familiar Logo appearance of a large section of the screen devoted to graphics and a smaller section of several below the graphics devoted to text commands that you type. In PC Logo for Windows, these two elements are separate windows, each with a different function.



The Graphics Window



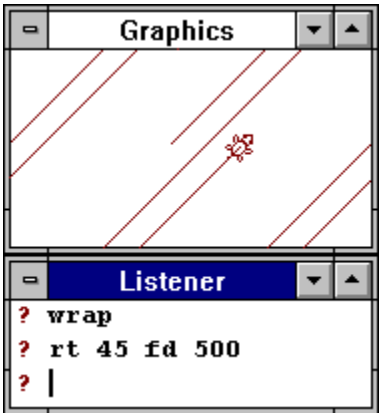
The Graphics window is where the turtle moves around and draws. All commands that affect the turtle cause the turtle to move within the graphics window.

Graphics Modes

The Logo turtles operate on a conceptual plane that is larger than the Graphics Window. It is possible to enter a turtle command that would cause the turtle to move beyond the edge of the Graphics Window. Logo has three graphics modes that allow you to determine what you want to occur visually when that happens.

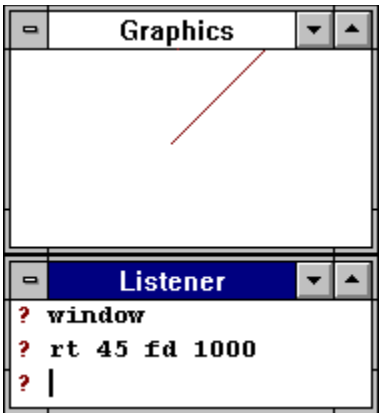
WRAP

The default graphics mode in Logo is `WRAP` mode, meaning that when the turtle moves off one edge of the Graphics Window it reappears at the opposite edOge. In `WRAP` mode, the command `FORWARD 1000` causes the turtle to draw a straight line in the middle of the screen and travel across it several times.



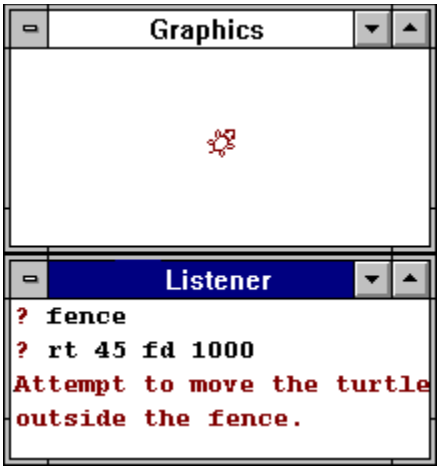
WINDOW

In `WINDOW` mode, the Graphics Window becomes a window on the larger plane of the turtle's domain. When the turtle moves beyond the edge of the Graphics Window, it does not reappear until another command brings it back to the region (centered around the [0 0] turtle coordinate) visible on in the Graphics Window. In `WINDOW` mode, the command `FORWARD 1000` causes the turtle to draw a line to the top of the Graphics Window and then disappear. The `HOME` command brings the turtle back to the center of the screen.



FENCE

In `FENCE` mode, Logo does not accept commands that cause the turtle to move beyond the edge of the edge of the Graphics Window or **fence**. In `FENCE` mode, for example, the command `FORWARD 1000` results in the message **Attempt to move turtle 0 outside the fence**. The turtle would not move.



The Coordinate System

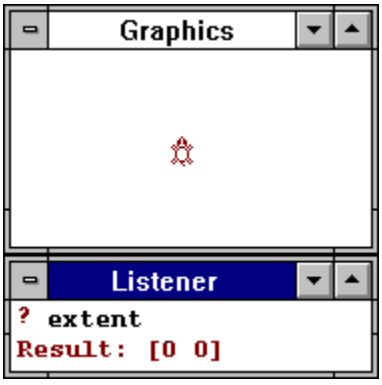
Logo turtle movements are measured in turtle steps while computer monitors display graphics in pixels. Logo provides several ways to relate turtle steps to graphic pixels. When Logo starts, each turtle step is equal to one screen pixel in any direction. The turtle is in WRAP mode, meaning it appears at the opposite border when it crosses one of the boundaries of the Graphics Window. If the window is enlarged with the mouse or the SETWINSIZE command, any wrapped lines reflect the previous border of the window but wrapping now occurs at the new window border. The HOME position is always in the middle of the Graphics Window regardless of its shape.

The SETWINSIZE command takes two inputs, the X and Y size of the drawing area as pixels. Since by default a turtle step is equal to a pixel, SETWINSIZE also determines the size of the Graphics Window in terms of turtle steps.

Sometimes it is useful to define the size of the Graphics Window in turtle steps regardless of its physical appearance in terms of pixels. Logo provides the SETEXTENT command to define the maximum distance from the HOME position of the turtle to the edge of the Graphics Window in terms of turtle steps (the *extent* of the Graphics Window). The SETEXTENT command takes one argument, the extent. You can, for example, establish that the border of the Graphics Window is 100 turtle steps from the HOME position no matter what the physical size or shape of the Graphics Window with the SETEXTENT 100 command. SETEXTENT can optionally take a second argument if you want to define different extents for the X and Y directions. For example, (SETEXTENT 100 200) defines a window that is 200 steps wide and 400 steps high.

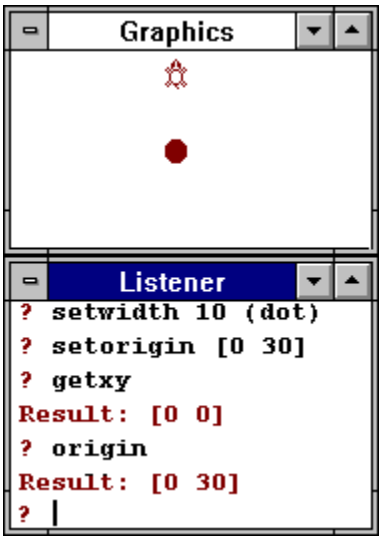
By using the SETEXTENT command and scaling the Graphics Windows to turtle steps, you can be assured that your graphics fit within the Graphics Window no matter how it is resized. This creates predictable relationships between your graphics and the Graphics Window but may result in your graphics being compressed or stretched depending on the ratio between the window size in pixels and turtle steps.

```
TO HEXDESIGN
   REPEAT 12[REPEAT 6 [FD 40 LT 60] RT 30]
END
```



The (SETWINSIZE) command used in parentheses without inputs restores the default extent of the Graphics Window to one turtle step per pixel.

You can also move the HOME coordinates of one or more turtles with the SETORIGIN command. SETORIGIN takes two inputs, X and Y, and defines the new HOME position for all active turtles. The coordinates are relative to the original HOME position, which is in the center of the Graphics Window. Use the command (SETORIGIN) to revert your turtles to the original HOME position.


Graphics
Listener
? setwidth 10 (dot)
? setorigin [0 30]
? getxy
Result: [0 0]
? origin
Result: [0 30]
?

Printing Graphics

You can print graphics you have created and displayed in the Graphics Window by choosing **Print** from the **File** menu when the Graphics Window is selected or by issuing the PRINTSCREEN command. Logo prints your graphics on the currently defined Windows printer.

With the command SETEXTENT "PRINTER you can automatically scale your graphics to the size of your printer's paper. When you issue this command the window extents are set to the size of the currently defined printer paper in printer pixels and the Graphics Window is resized automatically to reflect the paper format. You can then draw in the Graphics Window as if it were a sheet of paper. When you have finished drawing, you issue the PRINTSCREEN command or select **File/Print** and your graphics are printed as they appear in the Graphics Window.

The Listener

The Listener window "listens" to your computer keyboard for Logo commands. As you type, what you are typing appears in the Listener window. When you press the `ENTER` key, Logo carries out your commands. Depending on what commands you have given, the turtle may move in the Graphics window, additional text may appear in the Listener window, or something else may happen.

Adjusting Standard Layout

Standard layout, with the Graphics window occupying the top two thirds of the computer screen and the Listener window occupying the bottom third of the screen is the traditional Logo organization. You can adjust this arrangement as you like by moving and/or resizing the Graphics and Listener windows. You can return to standard layout by using the `SPLITSCREEN` (`SS`) command, choosing Standard Layout from the Window Menu, or with the `Contol-L` key combination.

Other window arrangements for Logo include the Windows standard Tile and Cascade arrrangements. These can be selected by choosing the appropriate command from the Window Menu. The `TEXTSCREEN` (`TS`) command maximized the Listener window providing the traditional Logo text screen. Similarly the `FULLSCREEN` (`FS`) command maximizes the Graphics window, providing the traditional Logo full graphics screen.

Entering Commands

As you type, your commands are written in the Listener Window, no matter what its size. The Listener Window wraps the words as you type when they reach the edge of the window, but to Logo a command line is not complete until you press the ENTER key. When you press the ENTER key, Logo evaluates what you have typed and tries to carry out your commands. If your commands cause any text to appear, it appears in the Listener Window. Any Logo messages are displayed in the Listener Window.

The Listener Window has been designed to be intuitive to use--it "listens" for your commands and executes them as soon as you press ENTER. The Listener Window also includes several features, such as editing and recalling that make it a powerful as well as easy-to-use Logo element.

Editing in the Listener Window

If you want to change commands as you type them in the Listener Window, you can do so by utilizing special keys that allow you to change and adjust your commands prior to pressing ENTER.

The following are the line editor keys:


deletes character to the left of the cursor

deletes character at cursor location



cursor moves one character in the direction of the arrow

cursor moves to the beginning of the line

cursor moves to the end of the line

toggles between insert mode (cursor is small blinking underscore) and overwrite mode (cursor is large blinking rectangle)

Recalling Previous Commands

Rather than retyping, you can recall previous command lines that appear in the Listener Window even after you press ENTER. To do so, use the mouse or one of the keys described below to position the cursor somewhere in the line you want to recall.



Moves the cursor up or down one line.

Moves the cursor up or down one page.

+


Scrolls one page up.

Scrolls one page down.

Moves the cursor to the first line of the window.

Moves the cursor to the last line of the window.

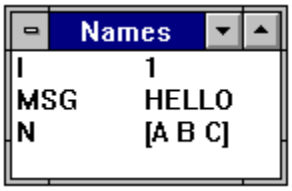
When you place the cursor anywhere in a line and press ENTER the line is copied to end of the text in the Listener Window and then executed. If you prefer you can have all the lines after the line you edit erased when you press ENTER by disabling the option **Copy input lines to end of text** in the dialog which appears when you choose the menu item **Options/Environment...**.

Combining the recall capability with the line editing capability makes it simple to correct a mistake in Logo and issue the command again without retyping it. When you type a line of Logo commands and see an aspect that you want to change, move the cursor within the line of commands again. Use the line editor keys to make the change and then press ENTER to execute the commands as you have amended them.

The Listener Window retains approximately 100 lines of Logo commands giving you a long history of your Logo commands that you can edit. In addition to editing commands and executing them again, you can copy and paste them into an edit window to turn them into a procedure or save them as a file.

The Names Window

To monitor the value of one or more Logo names, click  the Watch Names button, or choose **Names...** from the **Debug** Menu. This opens a dialog box where you can specify the names you want to watch. The values of these names are displayed in the Names window which opens automatically. Within the Names window you can double-click one of the displayed names and directly alter its value by typing a new one while your procedure is running.

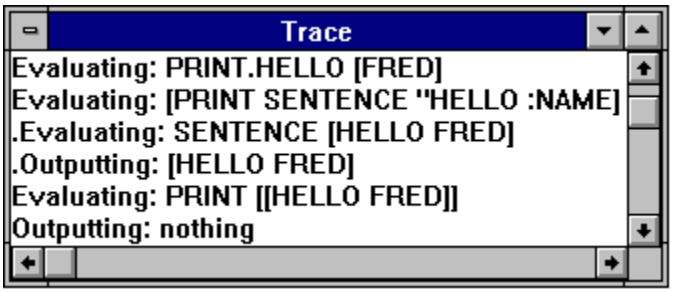


The Properties Window

To monitor the value of one or more Logo property lists, click  the Watch Properties button, or choose **Properties...** from the **Debug** Menu. This opens a dialog box where you can specify the property lists you want to watch. The values of these property lists are displayed in the Properties window which opens automatically. Within the Properties window you can double-click one of the displayed names and directly alter its value by typing a new one while your procedure is running.

The Trace Window

To watch a step-by-step execution of one or more Logo procedures, click  the Trace Procedures button, or choose **Procedures...** from the **Debug** Menu. This opens a dialog box where you can specify the procedures lists you want to watch. Each step Logo takes to run the specified procedure(s) is displayed in the Trace window which opens automatically. A maximum of 100 lines of output is displayed. Use the mouse to resize the window to see more output or the scroll bars to scroll through it.



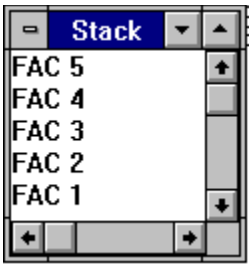
The window above displays the trace output for the following procedure:

```
? TO PRINT.HELLO :NAME
> PRINT SENTENCE "HELLO :NAME
> END
PRINT.HELLO defined.
? PRINT.HELLO "FRED
HELLO FRED
? _
```

Additional Logo primitives and system names that allow you to monitor procedure operation in the Trace window include TRON, TROFF, TRACE, and TRACELEVEL.

The Stack Trace Window

Logo allows you to monitor its stack, or temporary memory storage area for processing running procedures. The **Stack Trace** command in the **Options** Menu is a toggle which automatically turns the display of the Stack window on or off. When the Stack window is open, it displays the values stored on the Logo stack.



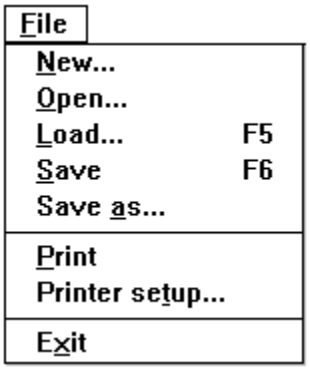
The following example generate a display like that in the window above. Note the PAUSE command to halt the procedure before it terminates.

```
? TO FAC :N
> IF :N = 1 THEN \
>   PAUSE \
>   OUTPUT 1 \
>  ELSE \
>   OUTPUT :N * FAC :N - 1
> END
FAC defined.
? FAC 5
PAUSE> _
```

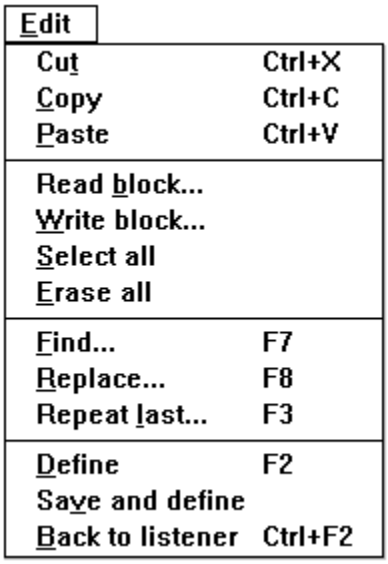
Menus

The titles of available pull-down menus appear across the top of your Logo screen. These are: File, Edit, Debug, Turtle, Options, Window and Help. By locating the mouse cursor on the title of a menu, and pressing the left mouse button, you cause the menu to extend downward offering a variety of Logo commands. Any of these commands can be selected by continuing to hold the left mouse button down and positioning the mouse cursor on the name of the command. These menu commands provide convenient shortcuts to access a wide variety of Logo features and execute many different Logo commands.

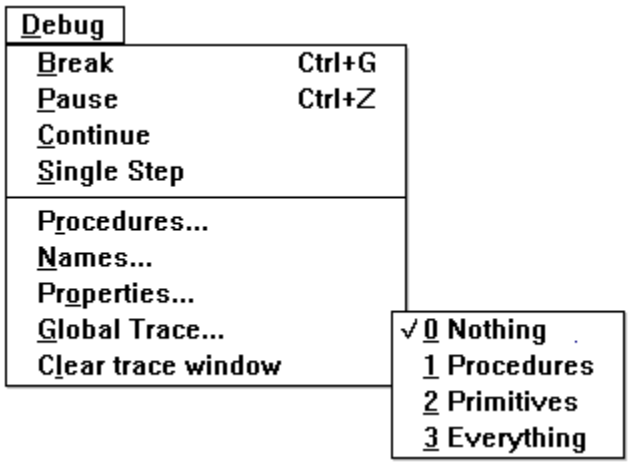
The **File** menu contains commands related to files and printing.



The **Edit** menu is a collection of commands which help you manage your Logo programs as well as the clipboard commands.

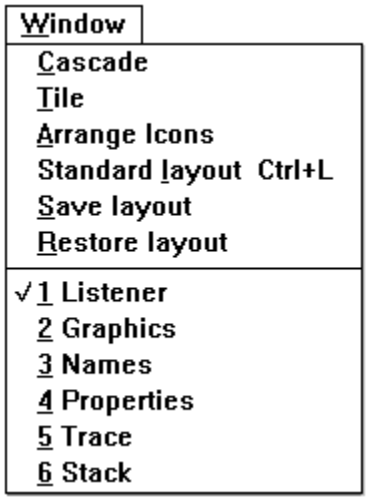


The **Debug** menu aids you in developing your programs.

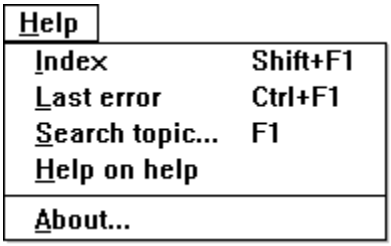


The **Turtle** menu gives you access to frequently used turtle commands.

The **Windows** menu contains commands which lets you arrange and select specific windows.



The **Help** menu gives you access to the Help system.



Button Bar

The button bar appears below the menu titles. The buttons provide additional shortcuts for Logo commands. To execute a button command, position the mouse cursor on the button and press the left mouse button quickly.

Keyboard

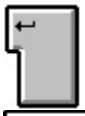
The main way to enter commands and text information in Logo is via the computer keyboard. In many respects the keyboard in Logo operates like a typewriter: letters and numbers that are typed appear in the Listener window and are read into Logo workspace when you press ENTER. In this method you can type Logo commands and watch Logo perform the action you specify when you tell it to read and act on the commands by pressing ENTER.

Function Keys

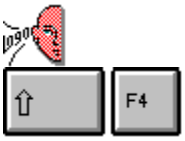
Most computer keyboards contain special keys called function keys that begin with the letter F and are numbered from 1 to 10 or 1 to 12. The Windows operating environment utilizes these function keys to execute certain common commands. These commands work while you are using Logo as well. Consult your documentation for Windows for a listing of the function key commands.

Logo also has built-in function key commands that operate when the function key is pressed in conjunction with the SHIFT key. These commands are:

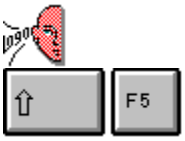
SS



TS



FS



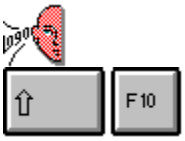
LOAD "

SAVE "

LOADPIC "

SAVEPIC "

CT



EDIT

Logo allows you to change commands that will be executed when a function key is pressed in conjunction with the SHIFT key. Each SHIFT-function key combination is a system variable which can be defined to perform the Logo command you prefer. The system name FKEY.n is assigned to each function key where n represents the number of the function key. By changing the definition of FKEY.n you change the action performed when you press SHIFT and the specified function key.

The SHIFT+F2 key combination issues the SPLITSCREEN command. You can change the effect of this key combination by redefining the variable FKEY.2 using the Logo command MAKE.

```
MAKE "FKEY.2 "HELLO
```

causes the text HELLO to be printed when you press SHIFT+F2 and ENTER. For a command stored in a function key variable to be executed without the ENTER key being pressed, the ENTER character can be appended to the end of the command you assign. This can easily be done with the CHAR command.

```
MAKE "FKEY.2 WORD "DRAW CHAR 13
```

This causes the Logo command DRAW to be executed automatically when you press SHIFT+F2.

The Mouse

Logo utilizes the mouse in the standard ways employed with most programs in the Windows operating environment. You can use the mouse to size and shape windows, select the active window, diplay a pull-down menu and select an item from it, click a button, and operate a dialog box.

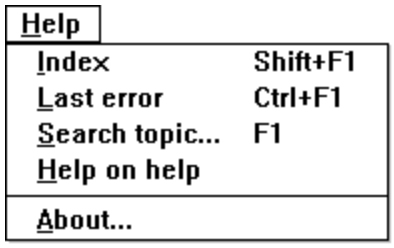
In addition, Logo allows you to monitor the position of the mouse and the state of the mouse buttons and respond to this information in your Logo programs. The `MOUSE` primitive outputs the current turtle coordinates of the mouse cursor and the `BUTTON?` primitive tells whether one of the buttons is pressed allowing you to incorporate information from the mouse into your Logo procedures.

Help

Logo provides complete on-line Help available from the Help menu or the Help button. Logo Help includes definitions, examples, and illustrations of all Logo primitives and system names. In addition, it contains discussions of a variety of topics related to Logo programming and the Logo environment including the full text of this Reference Manual. Logo Help is a separate program from Logo and both can be run in conjuntion under the Windows operating system.

You can access Logo Help by clicking the Help button. When you press the Help button, the Help title screen appears. From the title screen you can use the mouse to select the subject area you want help with, search for a particular topic, or perform another Help function.

Alternatively, you can access Help through the Help Menu. By selecting a particular item on the Help Menu, you can go directly to the part of Logo Help that you want.



Memory

As a Windows program Logo uses the memory Windows provides. In enhanced mode, available memory is only limited by the available disk space.

Memory Organization

When Logo is loaded, it divides the computer's memory into various segments for its own use. These include space for symbols and numbers, and space for lists. The available memory, or workspace, determines the number of procedures and commands you can issue in Logo or the size of files you can load.

Workspace

Logo workspace is the area of the computer's memory allocated by Logo for the variables, procedures, property lists, and arrays that you create. Access to Logo workspace is attained at toplevel, the command level of Logo where all commands are executed when ENTER is pressed. The question mark (?) Logo prompt displays at toplevel. Commands you type, names you create, and procedures you define are added to workspace as you use Logo.

Names, procedures, property lists, and arrays can also be created in a Logo editor. While in a Logo editor, this information is contained in the edit buffer section of memory. When you exit the editor and define the Logo objects you have created, they are copied from the edit buffer to toplevel as if you had typed them in at the Logo command level. They are then installed in the Logo workspace and are available for your use. (Note that a copy remains in the edit buffer.)

Garbage Collection

Logo adds the information you enter to its workspace until there is no more room. Then, Logo reviews all the information in workspace and discards objects in memory that are no longer being used. This space is reallocated for new storage. This process is known as recycling or in Logo as garbage collection.

Logo normally performs a garbage collection when the workspace is full. A garbage collection causes a pause in Logo of as much as one second depending on the speed of the computer. As the workspace begins to fill up, Logo performs more frequent garbage collections which may cause pauses at inconvenient times in a procedure. In this case, the Logo command RECYCLE, which causes Logo to perform a garbage collection, can be used at a point where the pause would not be noticeable.

Workspace Contents

Logo provides a number of commands that allow you to view what is in your Logo workspace. The PRINTOUT command will display any of the types of Logo objects in workspace or ALL of workspace. You can ERASE any object in workspace if you no longer need it.

The contents of the Logo workspace exist in the computer's memory and are available only as long as Logo is loaded and the computer is turned on. The SAVE command will preserve workspace contents in a file on a computer disk. By preserving your workspace on disk, you can exit from Logo and use the LOAD command to restore your Logo workspace to continue your Logo programming at a later time. See the section in this chapter on Files for more information.

Workspace Management

Logo workspace contains the names, procedures, property lists, and arrays that you create once Logo is loaded. Logo primitives and system names are part of Logo. Although they are available for use in workspace, they are not part of workspace in that they are not saved in workspace files and cannot be erased.

Sometimes it can be convenient to have the Logo objects you create act as primitives or system names in the sense that they are available for use, but are not saved in workspace files and cannot be erased. The BURY primitives provide this capability. These commands include BURY, BURYALL, BURYNAME, BURYPROC, and BURYPROP. When a Logo object, such as a procedure, is **buried** in Logo workspace it remains available as a command, but does not respond to PRINTOUT commands, cannot be erased, and is not saved in disk files when the SAVE command is used. Buried Logo objects can also be **unburied**, using commands including UNBURY, UNBURYALL, UNBURYNAME, UNBURYPROC, and UNBURYPROP.

Conserving Workspace Memory

Like any finite resource, the memory available for Logo workspace is more productive if used efficiently. The following suggestions provide ways for you to achieve more Logo programming power by conserving workspace memory.

1) PRINTOUT workspace contents and ERASE elements you no longer need.
2) Create subprocedures to replace repetitive sections of procedures to reduce text, since workspace must store the complete text of all defined procedures.
3) Use local rather than global variables since global variables always exist in memory and local variable occupy memory space only when their procedure is running.
4) Use tail recursion by putting the recursive procedure call in the last line of the procedure. Recursion requires a copy of the procedure in workspace for each level of recursion except for tail recursion, which only requires one copy, and so uses much less memory. If tail recursion is not possible, sometimes REPEAT, FOR, WHILE or GOTO..LABEL constructions can be substituted which require less workspace memory.

Additional Workspace Commands

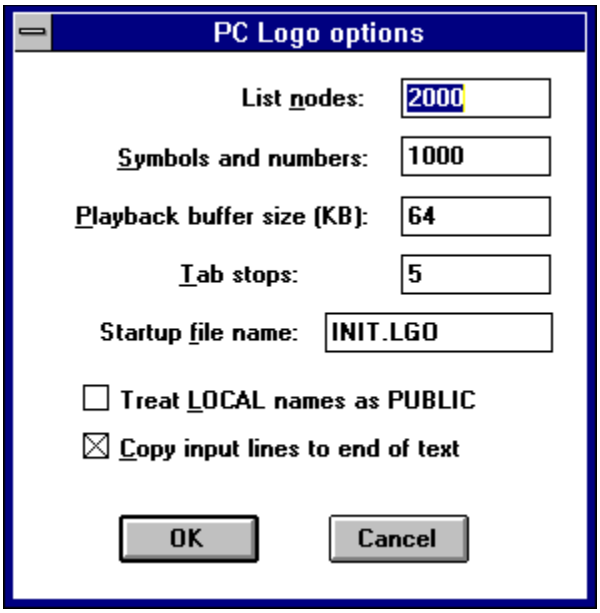
Additional Logo primitives and system names that help you manage your Logo workspace include:

CONTENTS	PONS	ERN
POPLS	PROCLIST	NODES
POPS	POTS	

Changing Memory Organization

When Logo loads, it allocates the computer's available memory in a pattern that is convenient for the majority of Logo users. The Logo command (RECYCLE "TRUE) lists how memory has been divided.

For some purposes, it may be useful to allocate memory differently than Logo does automatically. This can be conveniently done via the Environment item on the Options Menu. The dialog box that appears when you choose Options/Environment allows you to change the allocation of memory between nodes, atoms, and playback buffer.



The number of **nodes** determines list space, or general memory storage space, Logo has available. The larger this number, the more variables and procedures that can be kept in Logo workspace. The number of **atoms** is the total amount of space for numbers and symbols. Each number or symbol is considered an atom. The picture playback buffer determines how many graphics elements can be recalled when the Graphics window is redisplayed.

Editing

Logo provides full editing capability for your files and procedures. In Logo, your commands and procedures are executed as soon as you type them and press the ENTER key. In an edit window, the text of commands and procedures is diplayed and can be changed. You can execute these commands by defining them to make them part of Logo. You can then return to the edit window to make any changes you would like. You can also store commands and procedures permanently from an edit window by saving them in a disk file.

The Edit Command

The EDIT command in Logo automatically opens an edit window. When you type EDIT and press ENTER, an empty edit window is opened. You can type the definitions of procedures in the edit window using the mouse and cursor keys to position the cursor until you have written one or more procedures.

If you type EDIT followed by one or more procedure names, an edit window opens containing the text of the specified procedure(s). You can make any changes you like to the definition of the procedure(s) in the edit window.

The **Edit** Menu offers additional commands that make it convenient to perform editing functions in an edit window.

Defining Edit Window Contents

When you have created or changed procedures in an edit window, you can load them back into Logo to try them out. To do so, choose the **Define** command from the **Edit** Menu (or press F2). This command causes the edit window to minimize and loads the contents of the edit window into Logo workspace. You can now try your procedure(s) by typing its name in the Listener and pressing ENTER.

You can also define only some of the text in an Edit Window. For example, to define one procedure of several in an Edit Window highlight the text of the procedure and select Define from the Edit Menu or press F2.

Edit Windows

Logo provides an unlimited number of edit windows for you to use to write, store, and save your procedures. An edit window is similar to the Listener Window in that what you type appears as you type it when the edit window is selected. All of the editing commands that work in the Listener Window also work in edit windows. The difference is that commands you type are not executed when you press ENTER. The information you enter into an edit window, such as procedure definitions, is only entered into the Logo workspace when you choose to define the edit window's contents.

Opening Edit Windows

There are several ways to open an edit buffer in Logo. The `EDIT` command typed at toplevel causes an edit window to open. The `EDIT` command by itself opens an empty edit window. If the `EDIT` command is followed by one or more procedure names, the window displays the definitions of the procedure(s) when it opens. You can open a minimized edit window by double-clicking its icon on the screen. The menu command File/Load automatically opens an edit window and places the contents of the file you choose from the dialog box in it. Once an edit window is open it can be moved, sized, and positioned on the screen with the same commands as any other window.

Edit Window Name Conventions

Logo automatically assigns a name to any edit windows you open. This name is used by Logo until you change it or save the contents of the edit window. If you open an edit window with the command `EDIT` , the edit window takes the name `Editor`. If you open an edit window with the `EDIT` command followed by one or more procedure names, the edit window takes the name of the first procedure in the list. If you open an edit window with the File/Load command, the edit window takes the name of the file you select to load into it.

Files and Edit Windows

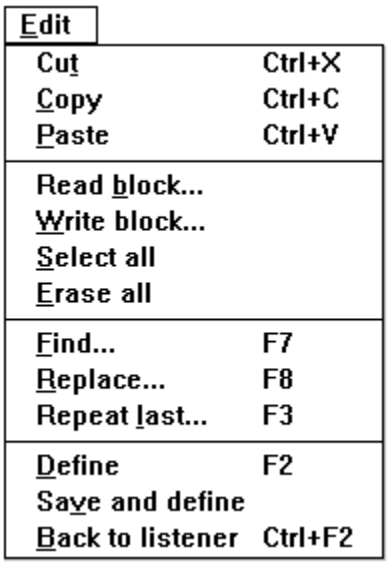
Disk files can be loaded directly into and saved directly from edit windows. To load a file from a disk into an edit window, choose **Open** from the **File** Menu. This automatically opens an empty edit window and loads the contents of the file you specify in the subsequent dialog box into it. The edit window takes the name of the file as its name. The contents of the file in the edit window can be edited and/or loaded into Logo workspace from the edit window.

The contents of an edit window can be saved directly to disk from the edit window. To save the contents of an edit window and then load the contents into Logo workspace, choose **Save and Define** from the **Edit** Menu. A dialog box appears in which you can specify the name of the file and where it is to be saved. After the contents of the edit window are saved in the disk file, the contents are loaded into Logo and the edit window is minimized. The name of the edit window is the same as the file name you assign it.

To save the contents of an edit window in a disk file without loading the contents into Logo and minimizing the edit window, choose **Save** or **Save as...** from the **File** Menu when the edit window is selected. A dialog box appears in which you can assign the file a name. After the file is saved you can continue to work in the edit window.

The Edit Menu

The Logo edit windows are designed to provide you with a convenient way of developing and working with your Logo programs and procedures. They provide basic text editing functions via the same key combinations as the Listener Window. In addition, the Edit Menu provides additional functions, including the ability to search for and search and replace text strings, append file contents, write partial edit contents to a file, and globally select and/or erase the text in the edit window.



You can also define only some of the text in the Edit Window. For example, to define one procedure of several in the Edit Window mark the text of the procedure and select **Edit/Define** from the Edit Menu or press `F2`.

Managing Edit Windows

Multiple edit windows are a powerful feature of Logo but require you to think about the best way to use them for your Logo programs and procedures. When you open an edit window and choose to define its contents with `F2` or **Edit/Define**, Logo places the contents of the edit window in workspace and minimizes the edit window. In some cases, it may be useful to keep each procedure in workspace in its own edit window. The minimized edit window icons provide a visual representation of the contents of Logo workspace. To save all of them in a file however, you would need to copy and paste them all together in one edit window and then save its contents with **File/Save** or **Edit/Save and Define**. Alternatively, the Logo command `SAVE` *`filename`* saves the entire workspace contents in the specified file.

In other situations, you may want to work with many procedure definitions in an edit window. This makes it convenient to save only some of the procedures in workspace or group procedures according to your preference. The edit windows are designed to provide maximum flexibility for you to adapt your own style of Logo programming.

As a convenience, each time you exit from Logo, you are asked if you want to save the contents of any edit windows that have not already been closed. This is designed to help you avoid losing any of your procedures or programs.

Printing

Logo can print text or graphics displayed on the screen or stored in a file on your computer's printer. Logo supports the printer currently installed in Windows. Your Windows documentation contains information on configuring the Windows operating system to work with a wide variety of printers.

Within Logo,  the Print button or the **Print** command on the **File** Menu provides a convenient way to print the contents of the active window. If the active window is the Listener or an edit window, **File/Print** automatically prints text on your printer. If the active window is the Graphics window, **File/Print** automatically prints graphics on your printer. In addition, Logo provides the printer primitives `PRINTER`, `SETPRINTER`, and `PRINTSCREEN`.

Files

Since Logo can save and load information from computer disk files, the work you do with Logo can be saved on disk for resumption at a later time or display at another place. Logo files can be saved on your computer's hard disk or on floppy disks. Information stored in disk files is permanently recorded. Logo workspace exists only in the computer's memory and is erased when you exit Logo or turn the computer off. Information in disk files is not affected by what program is run or by turning off the computer's power.

Loading and Saving Files

The contents of the active Logo window can be saved by clicking the  Save button or by choosing the **Save** command from the **File** Menu. Each of these methods provides a dialog box through which you can indicate the name you want for the file and where it is to be saved.

Similarly, disk files can be loaded into Logo by clicking the  Load button or by choosing the **Load** command from the **File** Menu. Each of these methods provides a dialog box through which you can indicate which file you want to load. If the Listener or an editor window is active the dialog box offers text files to be loaded. If the Graphics window is active, the dialog box offers graphics files to be loaded.

In addition, the **File/Open** command allows you to open a file to be viewed in a Logo editor without loading it into Logo workspace. When you choose **File/Open**, Logo creates an editor and then inserts the contents of the specified file in it. You can view the contents of the file, change it and/or load it into Logo as you wish.

In addition, Logo objects and graphics can be saved directly from workspace with the `SAVE` and `SAVEPIC` commands or loaded with the `LOAD` and `LOADPIC` commands.

File Names

You can choose the names you want for your Logo text and graphics files within the constraints of the Windows file naming system. This means they can have a maximum of eight alphanumeric characters, followed by a period, and three more alphanumeric characters.

File Extensions

The three characters following the period in a file name are know as the extension. If you do not specify an extension, Logo adds file extensions for your Logo files to help organize them by type. The `.LGO` extension is added to text files which are those saved from an editor or from the Listener. The `.PCX` extension is added to graphics files. This makes it easy to identify what type of file it is and that it was created by Logo.

File Formats

Computer disks files are organized in a specified way. This method of organization is called the format of the file. Different computer programs require that the files they save and load be formatted or organized in a certain way.

Text Files

Logo saves text information in a standard format called ASCII (American Standard Code for Information Interchange). When you use `SAVE` to save the contents of Logo workspace, save the contents of the edit

buffer, or save information from Logo Help, the data is stored in an ASCII format file. Many programs, including almost all word processors, can read ASCII files meaning you can load your Logo information into other programs if you want. Conversely, information created in other programs and stored as ASCII files can be loaded into Logo.

Graphics Files

Logo saves graphics information in PCX, BMP (Windows bitmaps), or WMF (Windows Placeable Metafiles) file formats. When you use SAVEPIC to save your Logo graphics, the picture is stored in PCX format. PCX is the default format, but one of the others can be selected via the save file dialog box. Logo graphics files conform to the format of the standard selected, so can be loaded into other programs that work with the selected format. Logo can also load files in these formats that were created with another program.

Default Drive

When Logo loads, the disk and directory from which it loads becomes the default disk. This is the location where Logo files are saved unless you specify a different location in the file name or via the dialog box. The Logo primitive DISK outputs the current default drive and SETDISK changes it.

File Locations

A file location can be given as part of the file name when using LOAD or SAVE in Logo. The Logo conventions for specifying file location are the same as Windows. Certain characters have special meaning in Logo and must be preceded with the backslash character to operate properly.

Command	File	Location
SAVE "MYFILE	MYFILE.LGO	default drive
SAVEPIC "MYPIC	MYPIC.PCX	default drive
SAVE "TEXTFILE.TXT	TEXTFILE.TXT	default drive
SAVEPIC "|B\:MYPIC|	MYPIC.PCX	B:
SAVE "|C:\DATA\LOGODATA.|	LOGODATA	DATA directory on drive C:

Additional File Commands

Additional Logo commands that help you manage disks and files include:

CURIR	DELETE	DISK	FILE.INFO
DIRECTORY	FILE?	RENAME	SETCURDIR
SUBDIR			

Logo Fundamentals

Basic building blocks of Logo are Logo procedures. A Logo command or series of commands consists of one or more procedures which instruct Logo to carry out certain actions.

 Primitives

 Names

 Procedures

 Inputs and Outputs

 Objects

 Language syntax

 Sounds and Multimedia

Primitives

Primitive procedures, or primitives, are built into Logo and are always present in the language. These are the basic Logo commands that are available when Logo is loaded. There are approximately 300 primitives in PC Logo for Windows. Use Logo Help to obtain a definition and examples of any Logo primitives once you have loaded Logo. To do so, type the word `HELP` (itself a primitive) and the name of a primitive you want to learn about. The Logo Help explanation for that primitive will appear. You can also access Logo Help via the Help menu or the Help button. Explanations and examples of all the primitives are also included in the Glossary.

Procedures

Procedures are the new Logo commands you create which instruct Logo to carry out the actions that you request. Procedures can consist of both primitives and other procedures. Once you have written a procedure, it functions in the same manner as a primitive procedure. The section on Logo Programming in the Reference provides information on how to write procedures.

Objects

Logo primitives and procedures are the basic commands in Logo that initiate the actions you want. Logo objects are the subjects of the commands or the data on which the commands operate. In Logo there are six types of objects: turtles, numbers, words, lists, property lists, and arrays.

 Turtles

 Lists

 Numbers

 Property lists

 Logo data

 Arrays

 Words

Turtles

The most popular and widely recognized Logo object is the turtle. A Logo turtle is a special type of graphics cursor that makes it easy and fun to make colorful computer graphics in Logo.

Think of the Logo turtle as your actor or agent in carrying out the commands you give it to create graphic images on your computer screen. Logo provides a wide range of intuitive commands to control the turtle.

Turtle commands to which the turtle responds directly include:

BACK	LEFT	SNAP
DOT	PENDOWN	STAMP
DOTCOLOR	PENUP	STAMPOVAL
DRAW	RIGHT	STAMPRECT
FILL	SETHEADING	TOWARDS
FORWARD	SETX	TOWARDS
GETXY	SETXY	TURTLETEXT
HEADING	SETY	XCOR
HIDETURTLE	SHOWN?	YCOR
HOME	SHOWTURTLE	

Turtle Tools

The Logo turtle has many tools it can use to draw in various ways as it moves about the computer screen. The turtle carries a pen which can be set to draw in a variety of colors and widths. By combining Logo commands to define the type of pen the turtle uses with commands to move the turtle around the screen, you can use the turtle to create colorful and intricate drawings. Experiment until you are satisfied with the graphic you have created and then define your commands as procedures. Then with just a few keystrokes you can instruct the turtle to fill the computer screen with colors, patterns, and pictures that you have drawn. Commands which control how the turtle appears and how it draws include:

COLOR	PENREVERSE	SETFONT
GETPALLET	SETCOLOR	SETTURTLEFACTS
PATTERN	SETPALLET	SETWIDTH
PEN	SETPATTERN	SHAPE
PENCOLOR	SETPC	FONT
PENERASE	SETPEN	TURTLEFACTS
SETSHAPE	WIDTH	

Multiple Turtles

The number of Logo turtles available is limited only by your computer's memory. When Logo starts, there are 16 turtles automatically available, each of which is identified by a number from 0 to 15. When Logo starts or you issue the DRAW command, turtle 0 becomes the active turtle. Use the multiple turtle commands to control any or all of the different turtles. Logo turtles can all carry out the same commands, each carry out different commands, or any combination providing an infinite number of programming and drawing possibilities. All currently active turtles follow the standard turtle commands. The following are commands specifically designed to work with multiple turtles:

ASK	TELL	TURTLES
EACH	TELLALL	WHO
SETTURTLES		

Numbers

Numbers are another kind of object or data on which Logo commands operate. A number can be an integer, such as 3 or 26; a decimal number, such as 2.14 or 57.86, or a number in exponential notation, such as 3.62E3 or 1.0E-2. Valid numbers in Logo can consist of any combination of the digits 0 through 9, a decimal point, the characters + or -, and the letter E when indicating exponential notation.

Logo reads a minus sign preceding a number as indicating a negative number. However, if there is a space between the minus sign and the number, Logo reads it as the subtraction operator.

Valid Numbers

Logo accurately performs calculations and mathematical operations on numbers in the range of 1E-38 and 1E38. All calculations in Logo have a precision of six digits.

Numeric Display

Logo floating point calculations are accurate to six significant digits, but when Logo first loads, decimal numbers are rounded to display only two decimal places. You can display more or fewer digits by changing the value of the system name PRECISION.

```
235 * 2.543
Result: 8.23
? MAKE "PRECISION 4
? 3.235 * 2.543
Result: 8.2266
? MAKE "PRECISION 8
The procedure PRECISION \
needs a number between 1 and 7 as its first input.
? _
```

Once you have changed the value of PRECISION, it maintains the new value until you change it again or exit Logo.

Logo expresses numbers in the range between 1E-6 and 1E6 in decimal notation. Numbers outside that range are converted to scientific notation.

Mathematic Operations

Logo has a wide variety of commands or primitives that deal with numbers, including traditional arithmetic operators as well as trigonometric, logarithmic, and other mathematical functions. All Logo numeric commands can be expressed with the primitive name followed by the number(s) on which it operates. For example:

```
2 2
Result: 4
? COS 90
Result: 0
? _
```

In addition, the arithmetic operators can be used as infix operators as they are traditionally, coming between the numbers on which they operate. For example:

```
+ 2
Result: 4
? 10.5 * 11.33
Result: 118.96
? _
```

When there is more than one arithmetic operation in a Logo line they are evaluated in the standard order of mathematical hierarchy, as follows:

1	-	(unary minus indicating a negative number)
2	*, /	(multiply and divide)
3	+, -	(add and subtract)
4	other math operations	
5	<, >, <=, >=	(comparative functions)
6	=	(equal)

Parentheses or other delimiters can be used to change the order of operation. See the section on Syntax for a full explanation of the order in which Logo evaluates its commands and how to change it.

Logo infix operators are :

/	=	>
>=	<	<=
-	*	+

Other mathematical operations include:

ABS	LOGAND	QUOTIENT
ARCTAN	LOGNOT	RANDOM
BASE	LOGOR	REMAINDER
COS	LOGXOR	RERANDOM
EXPN	LSH	ROUND
IBASE	PI	SIN
INT		SQRT
LOG	PRODUCT	SUM
LOG10		

Logo Data

Logo is designed to manipulate a wide variety of information or data beyond turtles and numbers. This includes text, numbers, text and numbers mixed together, structured and unstructured information. Information in Logo is recognized as one of four types: words, lists, property lists, and arrays. Logo has primitives to make it easy to create, store, recall, and manipulate each type of information.

Words

In Logo, a word is any group of characters, including letters, numbers, and punctuation marks, that does not contain a space and is not a primitive or procedure. In Logo, a word is indicated when a group of characters is preceded by quotation marks.

Numbers in Logo are a special class of word which have special primitives to deal with them. All word primitives work with numbers but numerical operators only work with numbers. Numbers may be expressed with a quotation mark at the beginning like the general class of Logo words, but Logo recognizes a number as both a number and a word whether or not it has a quotation mark at the front.

Names of all procedures in Logo, including primitive procedures, are words, although numbers cannot be used as the name of procedures.

Examples of Logo words include:

```
? WORD? "HELLO
Result: TRUE
? WORD? "PLATE.LUNCH
Result: TRUE
? WORD? 123
Result: TRUE
? WORD? "123GO
Result: TRUE
? WORD? "DRAW
Result: TRUE
? WORD? "HELP!
Result: TRUE
? _
```

An empty word, which is one set of quotation marks (") is a word which contains no elements.

```
? WORD? "
Result: TRUE
? COUNT "
Result: 0
? _
```

Lists

A list is information in Logo enclosed in square brackets. Lists can contain numbers, words, other lists or any combination. Each item inside the square bracket of a list is an element of the list. Each list is a single object in Logo, although there is no limit to the number of elements in a single list. Examples of lists include:

```
? LIST? [1 2 3 GO!]
Result: TRUE
? LIST? [[RED BLUE] [ORANGE GREEN ] [BLACK YELLOW]]
Result: TRUE
? LIST? [PD FD 50 RT 90]
Result: TRUE
? _
```

An empty list ([]) is a list with no elements.

```
? LIST? []
Result: TRUE
? COUNT []
Result: 0
? _
```

Additional commands which are used with words and lists include:

ASCII	FPUT	NAME
BUTFIRST	FROMMEMBER	NAMES
BUTLAST	ITEM	NAME?
BUTMEMBER	LAST	NUMBER?
CHAR	LIST	SENTENCE
		THING
EMPTY?	LPUT	WORD
EQUAL?	MAKE	
FIRST	MEMBER?	

Property Lists

Property lists are a special type of Logo list that allow assignment of multiple values to a single object in Logo. A property list is made up of an object, which must be a Logo word, and one or more property pairs associated with that object. The first element of a property pair is the name of the property. The second element is the property value.

Property lists are in the form:

```
name [property 1 value 1 property 2 value 2 ...]
```

The name of a property list must be a non-numerical word. The properties and their values can be words or lists. A property list could look like:

```
DOG [BREED SHELTIE HEIGHT [14 INCHES] COLOR [SABLE]]
```

Logo has special primitives to deal with property lists. In addition, many primitives that manipulate general lists can also manipulate property lists. The special property list primitives include:

GPROP	PPROP	PROPERTIES
PLIST	PPROPS	REMPROP

Arrays

Arrays are a special type of Logo object designed for efficient storage of structured information. Arrays are most often thought of as tabular forms of information with multiple rows and columns. Arrays are established by indicating the number of rows and columns or dimensions. Information can be stored in each position of the array that can be identified by locating its position on each dimension.

A calendar is a common form of array. Information written on a calendar for any particular date can be found by locating the date according to its three dimensions: year, month, and day. Each date has space to store the activities for that day.

Because of the structured nature of arrays, it can be easier to retrieve and use some types of information than if it were stored in lists or property lists. Logo has special primitives to create arrays of the dimensions you specify and enter, retrieve, and manipulate the information in them. Numbers, words, or lists can be stored in Logo arrays.

A BYTEARRAY is a special class of Logo arrays that can be used to store numerical information. Sets of information in which each element is a number between 0 and 255 can be stored more efficiently (which means it requires less of the computer's memory) in a BYTEARRAY than in a general array.

Because of the special nature of arrays, Logo has special primitives to deal with them:

AGET	ARRAYDIMS	BYTEARRAY?
ARRAY	ASET	FILLARRAY
ARRAY?	BYTEARRAY	LISTARRAY

Sounds and Multimedia

Most personal computers on which Logo is used have a speaker and are capable of generating sounds. Logo utilizes this capability, making sound an additional object which Logo can manipulate. You can specify the frequency and duration of a sound you want the computer to make or use common musical notation, allowing you to use Logo to play music on your computer. Logo's sound primitives are:

PLAY TONE
MCI MCI?

Especially the MCI command allows you to access the Windows mutimedia engine.

Names

Names, or variables, are Logo identifiers that store information. The information they store are Logo objects such as numbers, words, lists, property lists, or arrays. Think of Logo names as containers which hold the values or objects assigned to them. Names are always preceded by a colon (:), called dots in Logo parlance. The dots in front of a Logo name inform Logo to use the object named by the variable rather than the name itself.

There are three types of variables in Logo: local, public, and global. Local variables are created by a single procedure and exist only when that procedure is running. Public variables are created within a procedure and are only available to that procedure and any procedures that procedure calls. Global variables are created with the MAKE or NAME primitive and are available in Logo workspace to be recalled and used at any time by any procedure or any Logo operation.

Here is an example of using MAKE to assign a name to a word and to a list and then using the names with the Logo primitive PRINT.

```
? MAKE "GREETING "HELLO
? MAKE "SALUTATION [HOW ARE YOU?]
? PRINT :GREETING
HELLO
? PRINT :SALUTATION
HOW ARE YOU?
? _
```

Inputs and Outputs

Many Logo commands, including primitives and user-defined procedures, require information to operate and/or produce information when they conclude operation. Such information is one or more Logo object or the name that identifies the Logo object(s).

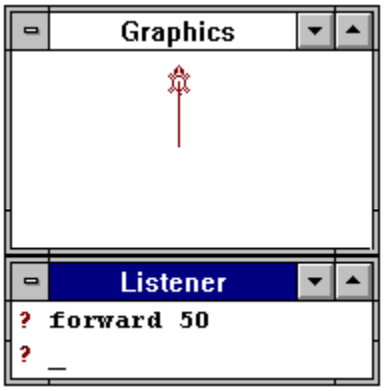
Information that a command requires before it can operate is called **input**. Information that a command produces after it operates is called **output**. Many commands that control the turtle's movement require input specifying where the turtle is to move. For example, FORWARD and BACK require a number specifying how far the turtle is to move. LEFT and RIGHT require a number specifying how many degrees the turtle is to turn.

Other commands produce or output information about the turtle. GETXY outputs a list of the turtle's current x and y coordinates. HEADING outputs a number specifying the turtle's current heading in degrees.

By using Logo objects as inputs to primitives and procedures, you can change or manipulate turtles, numbers, words, lists, property lists, and arrays. For example, when you give the turtle command FORWARD a valid number as an input, the turtle moves forward the specified number of turtle steps. Since the output of one procedure can be used as the input of another procedure, there is no limit to how you can use Logo to create and show graphics, store, manipulate, and display data, and play music.

The following commands are all equivalent in that they cause the turtle to move forward 50 turtle steps:

```
? FORWARD 50
? MAKE "DIST 50 FORWARD :DIST
? FORWARD SQRT 2500
? FORWARD FIRST [50 100 150]
? _
```



Logo Language Syntax

Logo syntax is similar to that of written language in that spaces are the most common way to separate one element from another. When you type a Logo line, each individual element is separated from the one before and after by one or more spaces. When Logo reads the line after you press the ENTER key, it identifies each individual element as a primitive, procedure, number, word, list, property list, or array and operates according to set rules for responding to each command and object.

Parsing

The process Logo uses to interpret and carry out a line of commands is called parsing. As a general rule, Logo reads elements in a line one at a time from left to right and processes them in that order. Most Logo primitives and all user-defined procedures are used by invoking the procedure name and then the inputs to the procedure to conform to left to right parsing (which is also the way we read). Logo identifies a command, checks its memory for the number of inputs required by the command, and then proceeds from left to right across the line to identify the input.

Arithmetic infix operators are the major exception to this rule. Infix operators (+, -, *, /, <, >, <=, >=, =) can come between their inputs (although they can also precede their inputs). In that case, Logo looks on either side of an infix operator to identify its inputs.

Delimiters

The basic way to indicate the beginning and end of each element in a Logo command line is with a space. The space is the basic **delimiter** in Logo, or the means used to indicate the beginning or end of a Logo element.

Logo recognizes other characters as delimiters as well. These include the infix operators (+, -, *, /, <, >, <=, >=, =) as well as quotation marks (") which identify words, colons (:) which identify names, square brackets ([]) which identify lists, and parentheses (()) which group elements together. Spaces are optional before and after these characters in a Logo line. For example, Logo will understand if you type:

```
? IF(3+5)=(4*2)PRINT"RIGHT!
RIGHT!
? _
```

Parentheses

Parentheses are used in Logo to group objects together differently than Logo would ordinarily. When parentheses are used with arithmetic operators, calculations are performed in a different order from the standard mathematical hierarchy. For example:

```
? 16 - 4 / 2
Result: 14
? (16 - 4) / 2
Result: 6
? _
```

Parentheses can also be used with primitives and user-defined procedures that accept a varying number of inputs. If you give more inputs to a procedure than Logo expects, Logo will stop because it doesn't know what to do with the extra input. For example, Logo expects only one input with PRINT. If you type:

```
? PRINT "HI "THERE
```

Logo prints HI and then attempts to interpret the next input, THERE. Since you have not told Logo what to do with the input word THERE, Logo outputs it as a result:

```
Result: THERE
```

and stops.

PRINT can take a variable number of inputs if it and its inputs are inside parentheses. For example:

```
? (PRINT "HI "THERE)
HI THERE
? _
```

This use of parentheses is valid for Logo primitives and user-defined procedures that are specified to use optional variables. See the section on Optional Variables for information on creating procedures that can have variable inputs.

Punctuation

It is always good practice to **close** parentheses (which group Logo elements) and brackets [which identify Logo lists] by providing a right parenthesis or bracket for each left parenthesis or bracket on a Logo line. As a convenience, when you end a Logo line by pressing ENTER, any open brackets or parentheses are closed. For example:

```
? REPEAT 3 [(PRINT [GOOD MORNING] [FRIEND!
```

This command is the same as:

```
? REPEAT 3 [(PRINT [GOOD MORNING] [FRIEND!])]
```

Occasionally you may want Logo delimiter characters to act as normal characters rather than assume their special meaning. For example:

```
? PRINT "HOWDY-DOODY
DOODY is not a Logo procedure.
? _
```

In this case, Logo is interpreting the dash as a subtraction operator and cannot find any numbers to subtract. Use the backslash key (\) to indicate to Logo that the following character does not have its special Logo meaning.

```
? PRINT "HOWDY\-DOODY
HOWDY-DOODY
? _
```

The backslash **quotes** the character following it, removing any special meaning in Logo. The system name :DELIMITERindicates the character that can be used to quote a group of characters. When Logo starts up the group delimiter is the vertical bar (|).

```
? PRINT "*/+#&
Bad number syntax.
? PRINT "|*/+#&|
```

```
*/+#&
? _
```

In a manner similar to parentheses and brackets, Logo will close an open group delimiter character when you press ENTER.

Logo Programming

Programming your computer is giving it instructions to take some sort of action. Logo commands act on Logo objects to turn your computer into an appliance you can use to explore and create. With Logo you can move the turtle(s) to create graphics images on the computer screen, print the results of mathematical calculations, manipulate information, play music, or any combination of these elements.

Logo primitives, or built-in commands, are easy to understand and can perform many functions. The power of Logo comes from its ability to let you create your own commands and to incorporate those commands as an integral part of the language. By creating your own commands, or procedures, you expand the "vocabulary" of Logo and give your computer the capability to do what you want it to in your own words.

 Creating Procedures

 Defining Events

 Inputs to Procedures

 Multitasking

 Program Structure

 Interaction

 Recursion

 Streams

 Timing

 Debugging

 Flow Control

 Accessing DLLs

Creating Procedures

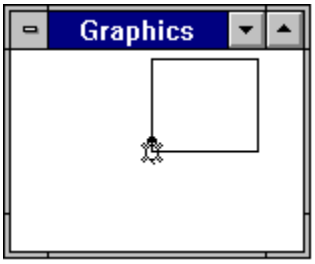
To create a Logo procedure, you must first choose a name for the procedure. A procedure name can be any Logo word except for the name of a primitive procedure or a number. It cannot contain any spaces.

To signal to Logo that you want to create a new procedure, type the word TO followed by the name you have chosen for the procedure. Logo enters **quick define** mode, which allows you to type all of the instructions you want the procedure to contain.

Quick define mode is indicated by the greater than prompt (>) as distinguished from the normal question mark (?) prompt that indicates Logo is waiting for a command. When defining a procedure in quick define mode, Logo stores the commands you type as part of the procedure when you press ENTER rather than carrying them out.

Type the commands you want to become part of the procedure one line at a time, pressing ENTER at the end of each line. After you have entered all the instructions you want included in the procedure, type the word END on a line by itself. Logo will respond by saying that the procedure is defined. Now the procedure you created is an integral part of Logo. Until you exit Logo or turn off your computer, every time you type the name of the procedure you created, Logo will carry out the instructions you included in that procedure. For example:

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



You can also create procedures and/or change procedures in a Logo Editor. The editor provides more extensive editing capabilities while writing procedures than quick define mode. See the section on Editing in the Reference for further information.

Inputs to Procedures

Like primitives, procedures can require information or input in the form of Logo objects. This allows a single procedure to operate in different ways depending on what information is supplied. For example, the Logo primitive FORWARD requires a number as input to specify how far to move the turtle.

You can re-write the SQUARE procedure illustrated above to also require an input to specify how large a square to draw. Do so by inserting a variable name in the title of the procedure and substituting the variable name for the distance the turtle is to travel to draw the side of the square. Remember that in Logo a variable name begins with a colon (:) or dots.

```
? TO SQUARE :SIDE
>    REPEAT 4 [FD :SIDE RT 90]
> END
SQUARE defined
? SQUARE
SQUARE needs more input(s).
? SQUARE 100
```

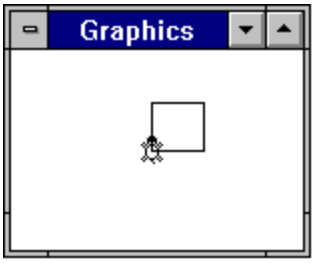
When you insert a variable name in the title of a Logo procedure, you indicate to Logo that information must be supplied along with the procedure name when the procedure is invoked. The first SQUARE procedure illustrated above only draws squares of size 100. The second SQUARE procedure draws squares of any size, but you must specify the size when you call the procedure.

Optional Inputs

You can also create Logo procedures whose inputs are optional, meaning that the procedure will operate with a **default** or pre-specified value if none is given or with the supplied value if one is available. Procedures with optional variables have a specific form:

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

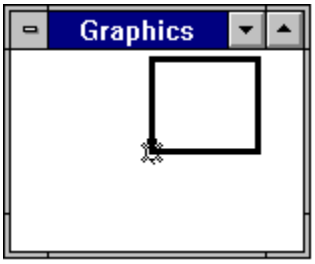
```
? (SQUARE 50)
```



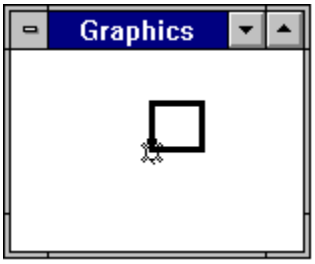
Optional variables in procedures and their default values are enclosed in brackets and made part of the title line in the procedure definition. If the procedure is called with no argument, it operates with the default value of the variable. When a value for the optional variable is supplied, the procedure name and its input(s) must be enclosed in parentheses to indicate to Logo that optional information is included.

If a procedure has both required and optional variables, the optional variables must be specified in the title line after the required variables.

```
? TO SQUARE :PEN.WIDTH [:SIDE 100]
>    SETWIDTH :PEN.WIDTH
>    REPEAT 4 [FD :SIDE RT 90]
> END
SQUARE defined
? SQUARE 3
```




```
? (SQUARE 3 50)
```



If you want to specify the number of default arguments, add this number to the list of arguments. This requires the user to supply the specified number of arguments, unless parentheses are used.

```
? TO POWER :M [:N 2] 2
>    LOCAL "RES
>    MAKE "RES :M
>    REPEAT :N - 1 [MAKE "RES :RES * :M]
>    OUTPUT :RES
> END
POWER defined.
? POWER 2 3
Result: 8
? POWER 2
The procedure POWER needs more input(s).
? (POWER 2)
Result: 4
? _
```

List arguments

You can also write procedures which accept any number of inputs. When such a procedure is called, all inputs are passed to the procedure in the form of a Logo list which it can evaluate.
to create such a procedure, the name of the list enclosed in brackets should follow the name of the procedure. The following procedure prints its arguments one by one:

```
? TO PRINT.INPUTS [:LIST]
>    WHILE [NOT EMPTY? :LIST] [PR FIRST :LIST MAKE "LIST BF :LIST]
> END
PRINT.INPUTS defined.
? PRINT.INPUTS 1 2 3
1
2
3
? _
```

You can also specify the number of required arguments. This is convenient since Logo would add everything following the name of the procedure to the list passed to the procedure as input.

```
? PRINT.INPUTS 1 2 PRINT "FINISHED 3
FINISHED
1
2

3
? _
```

In this example Logo evaluates all arguments on the input line following the procedure name, since `PRINT.INPUTS` does not have a required number of arguments. First `PRINT` was evaluated, printing the word `FINISHED`. After this, the input list for `PRINT.INPUTS` was constructed, leaving an empty word between the numbers 2 and 3 because `PRINT` does not output anything. Finally, this list was printed by `PRINT.INPUTS`. The following example illustrates how to specify the number of inputs.

```
? TO PRINT.INPUTS [:LIST] 2
>    WHILE [NOT EMPTY? :LIST] [PR FIRST :LIST MAKE "LIST BF :LIST]
> END
PRINT.INPUTS redefined.
? PRINT.INPUTS 1 2 PRINT "FINISHED 3
1
2
FINISHED
Result: 3
? PRINT.INPUTS 1
The procedure PRINT.INPUTS needs more input(s).
? _
```

Program Structure

When a Logo procedure is defined, it becomes an integral part of the language and functions just like a Logo primitive. This makes it possible for one procedure to include another procedure as a command within it.

It is common Logo practice to break a programming project into small parts and write a procedure to accomplish each task in the project. Then an overall procedure can be used that calls each of the other procedures in its turn to accomplish the project.

Two types of Logo programming styles have emerged. One is the **top-down** approach which means looking at an overall programming project, deciding what the parts are, and then writing procedures to accomplish each of the specified parts. The other is the **bottom-up** approach which means experimenting with various parts of the project by writing procedures to accomplish them and then putting all the pieces together at the end to accomplish the project.

Both styles of program structure are discussed in the Tutorial and the many books that have been written about Logo. Both are equally valid and the fact that both can be successful illustrates the flexibility of Logo to adapt to individual styles of thinking.

 Recursion

 Timing

 Flow Control

 Defining Events

 Multitasking

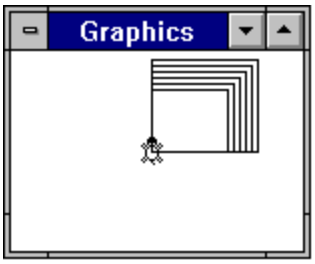
 Interaction

Recursion

A special type of Logo program structure is called **recursion** or the inclusion of a procedure inside itself. Once a Logo procedure is defined, it can be used as a Logo command like any other Logo command. This means that a Logo procedure can call itself. When used with variables, this practice, called recursion, makes it easy in Logo to create complex and fascinating results from very simple programs.

The `SQUARE` program uses an optional input to determine the size of the square it draws. One procedure can make squares of any size, depending on the value of the `:SIZE` variable. You can combine the flexibility of the `SQUARE` procedure with Logo's recursion capability to draw squares of many sizes with one command. To do so, you can use the `SQUARE` procedure over and over, giving it a new value for `:SIZE` each time. For example:

```
? TO SQUARE :SIDE
>    REPEAT 4 [FD :SIDE RT 90]
>    SQUARE :SIDE + 10
> END
SQUARE defined
? SQUARE 50
```



This new `SQUARE` procedure first draws a square of size 50, which is the initial input you gave it. It then calls itself and draws a square of size 60 or size `:SIDE + 10`. It repeats this process endlessly by drawing ever larger squares until the program is stopped with `Control-G` or the  `Stop` button.

Timing

Logo commands (or any computer instruction) are carried out sequentially, or one after another. As a general rule, commands on a Logo instruction line are invoked from left to right. (See the section on Parsing for more information.)

When Logo encounters a procedure name in a line of instructions, it carries out the commands in that procedure sequentially. Logo starts with the first line of the procedure, invoking the commands from left to right. It then proceeds to the next line and so forth until the `END` of the procedure is reached.

Flow Control

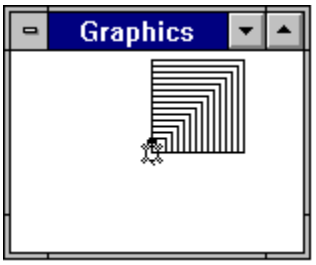
This process of sequential operation is called program flow. Once Logo begins to execute a procedure, the control of the computer passes from the keyboard to the procedure. Logo provides commands that allow you to control the flow of the computer's operation from within Logo procedures. You can use IF and related primitives to test conditions in the Logo environment and pass control to one procedure or another depending on the condition being met. You can let the user determine how the program is to proceed by considering input from the keyboard to determine which procedure to call next. Your Logo program can examine information in disk files and operate in various manners according to what is encountered. The ability to control the flow and timing of your computer is a powerful aspect of Logo.

The following primitives and system names allow you to control the flow of your procedures:

AND	IFTRUE	THEN
CATCH	LABEL	THROW
CONTINUE	NOT	TOPLEVEL
ELSE	OR	TRACE
ERROR	PAUSE	TRACE.LEVEL
FALSE	REPEAT	TRON
FOR	RUN	TROFF
GO	SINGLE.STEP	TRUE
IF	STOP	WAIT
IFFALSE	TEST	WHILE

The recursive SQUARE procedure above continued operation endlessly until it was interrupted with certain combinations of keys that tell the computer to stop whatever it was doing (Control-G or the Stop button). It might be useful to have the procedure stop and return control to the user after it has drawn the largest square that will fit on the screen. The following example shows a SQUARE procedure that will stop after it has drawn a square of size 200:

```
? TO SQUARE :SIDE
>    IF :SIZE > 200 THEN STOP
>    REPEAT 4 [FD :SIDE RT 90]
> END
SQUARE defined
? SQUARE 10
```



SQUARE draws squares as it did before, but each time it calls itself it examines the value of :SIDE to see if it is greater than 200. When the value of :SIDE exceeds 200 the procedure stops and returns control to the user, as indicated by the question mark (?) prompt indicating that Logo is waiting for further instruction.

Defining Events

The DEFEVENT command allows you to tie the operation of a Logo procedure to the occurrence of a particular event, including the activation of the Control-G key combination, or to operate on a scheduled regular basis. This allows you to handle the program user's attempt to break a program in the way you like or to cause your Logo procedures to operate or output on a regular basis without having to repeatedly call them.

There are two events defined which you can define. The BREAK event occurs every time the Control-G key is being pressed. Normally, a built-in event procedure will turn off the timer ticks (if a timer has been activated with the TIMER command) and return you to toplevel. You can modify this behavior by defining an event procedure for the BREAK event. Your definition should always provide the option of returning to toplevel. The TIMER event is activated with the TIMER command. This command starts a timer which, when elapsed, generates a TIMER event which can be used to activate a Logo procedure. Both the BREAK and the TIMER events do not generate any inputs for the event handling procedures.

Multitasking

Logo has the ability to run one or more procedures in the "background" or while you issue Logo commands at toplevel. This allows you to continue programming in Logo or do other activities while your specified Logo procedure executes. The `LAUNCH` command starts a Logo procedure in the background and then returns to toplevel to await your next instruction. The `HALT` command stops procedures that are running in the background. Note, however, that the procedures you `LAUNCH` should be kept as small and fast as possible, because all Logo programs stop running during the execution of a background routine. Also, the Listener Window does not record any input while a background procedure is running.

Pressing `Control-G` halts any background procedures. You will have to `LAUNCH` them again after pressing `Control-G`.

Interaction

The normal Logo state is the command line where Logo displays a question mark (?) prompt and waits for you to type a command and press ENTER. When you press ENTER, Logo executes commands on the line. This Logo state is called **toplevel**.

When Logo executes a procedure, control of the program flow passes from toplevel to the procedure. As you saw in the recursive SQUARE procedures above, it is possible to write procedures that never return control to toplevel until they are interrupted by Control-G or the  Stop button. You can write procedures that continue operation until certain conditions are met and then return control to toplevel.

It is also possible to write Logo procedures that maintain flow control even though they ask the user to enter information from the keyboard. The READ... series of primitives can be used to obtain information from the keyboard without returning flow control to toplevel. You can alter the SQUARE procedure to draw squares of the size specified from the keyboard:

```
? TO SQUARE
>    PRINT [PLEASE ENTER A NUMBER BETWEEN 1 AND 200.]
>    MAKE "SIDE READ
>    REPEAT 4 [FD :SIDE RT 90]
>    SQUARE
> END
SQUARE defined
? _
```

The SQUARE procedure will continue to run until you type Control-G to return flow control to toplevel.

Streams

The normal interaction between a person and a computer is for the person to enter information from the keyboard and for the computer to display information on the computer screen. The information that is entered or input is called the input stream and the information that is displayed or output is called the output stream. When Logo starts, the input stream originates from the keyboard and travels to the computer's processor as you type characters. The output stream originates from the computer and is directed to the screen for you to read.

You can use Logo to control the source of input and destination of output as well as its flow. Input and output streams allow you to use the same Logo commands and operations that you use between your keyboard and computer screen to communicate with other devices. For example, you can input characters from the keyboard, a disk file, or a telephone modem and direct the output to another device such as the video screen, a printer, a disk file, or a telephone modem.

STANDARD.INPUT and STANDARD.OUTPUT

Two pre-defined names, STANDARD.INPUT and STANDARD.OUTPUT, control input and output streams in Logo. The values of these names determine the source of the input stream and the destination of the output stream. When Logo starts, the value of both variables is 0, meaning that input comes from the keyboard and output is directed to the computer screen. By changing the value of these names you can re-direct the source of Logo input or the destination of its output.

Other primitives and system names that work with STANDARD.INPUT and STANDARD.OUTPUT to control streams in Logo include:

CLEARINPUT	GETBYTE.NO.ECHO	READCHAR		
CLOSE	OPEN	READLINE		
COPYOFF	PEEKBYTE	READLIST		
COPYON	PRINT	READQUOTE		
CREATE	PRINTLINE	SHOW		
EOF	PUTBYTE	UNGETBYTE		
GETBYTE	READ	.READ	.WRITE	.SEEK

Debugging

Sometimes when you run a Logo procedure its outcome is unexpected and not what you really wanted it to do. The computer is a very literal machine and does exactly what you tell it to do since it does not know what you mean for it to do. Sometimes you may not always say exactly what you mean and in Logo this shows up in procedures that do not do what you want.

These unexpected occurrences are known as **bugs** and are common in computer programming of all types. Because of the procedural nature of Logo and its ability to break a project into small parts and write a procedure to perform each part, bugs are sometimes easier to find, isolate, and eliminate than in some other computer languages.

Nevertheless, it can sometimes still be difficult to figure out why your Logo programs are not acting as you expected. Logo provides several types of **debugging** tools to help you eliminate bugs or unexpected consequences from your programs.

PAUSE

A simple means of debugging is to pause while a procedure is executing and examine what it has done so far. The Logo command PAUSE will temporarily halt execution of a procedure and return control to the keyboard without returning to toplevel. By inserting a PAUSE into a procedure, you can suspend the execution of the procedure so you can examine what it has done so far.

A PAUSE in a procedure is indicated by the pause mode prompt:

```
PAUSE> _
```

This indicates that flow control is maintained by the procedure and that you have not returned to toplevel. Logo will, however, execute any commands typed at the keyboard while in pause mode, allowing you to examine the value of variables, ascertain the position of the turtle(s), etc.

You can also click the Pause button  in the button bar or select the Pause command from the Debug Menu any time to temporarily pause a procedure. You can also pause a procedure by pressing Control-Z.

Type CONTINUE or CO to exit pause mode and return flow control to the procedure which will continue executing until another PAUSE, STOP, or END is encountered.

Alternatively, you can click the Continue button  in the button bar or select the Continue command from the Debug Menu.

By typing CONTINUE with a number as an argument and putting both in parentheses, the procedure will not pause again until it has encountered PAUSE the number of times indicated by the argument to CONTINUE. This allows you to pass through several steps of a procedure before pausing again which can speed up debugging.

Type TOPLEVEL or press Control-G while in pause mode to return to TOPLEVEL without continuing execution of the procedure.

SINGLE.STEP

SINGLE.STEP is a special case of PAUSE. SINGLE.STEP is a predefined Logo name whose value is FALSE when Logo starts. By typing

```
MAKE "SINGLE.STEP "TRUE
```

you automatically insert a PAUSE after every command of every procedure. Logo will execute each command one at a time (a single step) and put you in single step pause mode after each one. Single step pause mode is indicated by the prompt:

```
SINGLE.STEP> _
```

As in pause mode, you can type commands at the keyboard. Logo will tell you what it is doing between each single step, indicating what is being evaluated and output.

You can also turn single stepping on and off by selecting the Single Step command on the Debug Menu. This command acts like a toggle; selecting it will either turn single stepping on or off.

TRACE and TRACE.LEVEL

Tracing is following your procedure step by step to see exactly at what point its behavior departs from what you want it to do. TRACE is a pre-defined Logo name whose value is FALSE when Logo starts. By typing

```
MAKE "TRACE "TRUE
```

Logo will automatically print information in the Trace window about its internal operations for you to monitor. This information includes what is being evaluated and output as Logo executes each command in a procedure.

The pre-defined name TRACE.LEVEL determines the detail of the information that TRACE provides. :TRACE.LEVEL can be set at level 1, 2, or 3. The level of :TRACE.LEVEL determines the amount of detail Logo reports about its internal operations. The higher the level, the more detail that is reported. When Logo starts, the value of :TRACE.LEVEL is 2.

The **Global trace...** command in the **Debug** Menu lets you select directly which trace level you want.

TRON and TROFF

The TRON command is a special type of tracing. TRON accepts a Logo name, the name of a property list, or the name of a procedure as an argument. When TRON is used with a Logo name, the value of that name is displayed in the Names window. Each time the value changes, the new value appears in the Names window. The values of property lists whose name is used as the argument to TRON appear in the Properties window. You can edit the value of the name or property list directly inside the Names or Properties window.

Operation of any procedure being traced by using its name as an argument with the TRON command or by setting :TRACE to "TRUE is reported in the Trace window.

The TROFF command turns off debugging for thespecified name or procedure.

Both TRON and TROFF can be used with no or with multiple inputs if the commands and their inputs are surrounded with parentheses. Using TRON with no inputs enables tracing for all Logo names. TROFF with no inputs completely disables debugging.

The TRON and TROFF command do not interact with the built-in TRACE and TRACE.LEVEL variables. Since TRACE and TRACE.LEVEL operate globally in Logo, TRON and TROFF allow more selective

monitoring of your program operation.

Debug Buttons

You can also enable and disable debugging for selected names, property lists, or procedures by clicking one of the following buttons on the button bar:

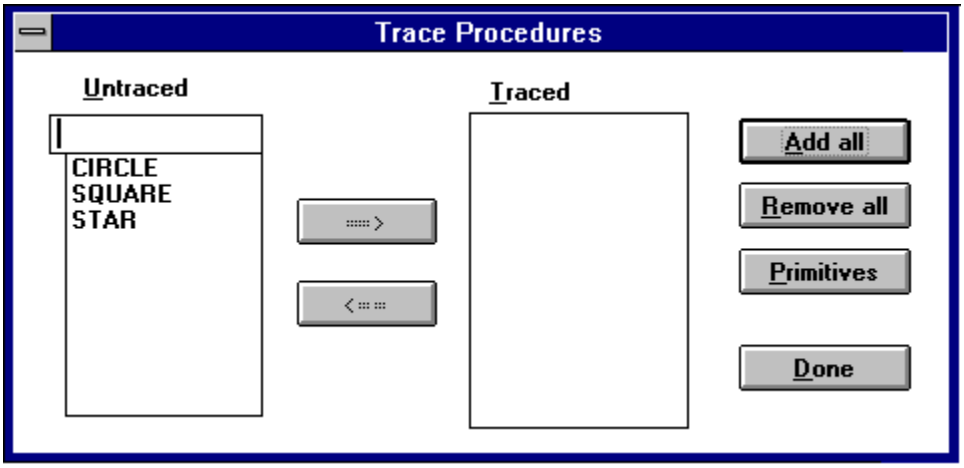
 Enable or disable tracing of selected procedures. The **Trace procedures** button is equivalent to the menu command **Debug/Procedures...**

 Enable or disable the display of the contents of selected names. The **Trace names** button is equivalent to the menu command **Debug/Names...**

 Enable or disable the display of the property list of selected names. The **Trace properties** button is equivalent to the menu command **Debug/Properties...**

Debug Dialog

When you choose to trace procedures, names, or property lists with either the buttons or the menu commands, the Debug Dialog appears. This allows you to select which procedures, names, or property lists you want to trace. All of the available procedures, names, or property lists are listed in the left window of the Debug Dialog. Select the ones you want to trace in the left window and click the button to move it to the right window which lists the procedures, names, or property lists currently being traced. The reverse process eliminates a procedure, name, or property list from being traced. When you close the Debug Dialog and run your Logo programs, the appropriate trace window automatically opens, showing the output or value of the procedure(s), name(s), or property list(s) you specified that you wanted to trace.



Accessing DLLs

Logo provides several commands to ease the direct access to Windows. One of the most interesting features of Logo is its ability to call any routine contained in a Windows DLL. This makes it possible to call virtually any Windows routine, but it also opens the path to writing own routines for Logo, thus enhancing the overall functionality of Logo. The `.WINDOWS` command converts it inputs to character strings and numbers and passes these values to a routine located within a DLL. In addition to the inputs, you will have to supply the name of the routine to be called. Additionally, you may supply the name of the DLL where the routine is located. Logo recognizes all entry points located in the standard Windows 3.0 and 3.1 DLLs, so there is no need to supply these names.

Besides `.WINDOWS`, Logo offers access to its internal window handles and device contexts as well as to the Windows messaging system. The `.HINST` command outputs the instance handle of Logo. The `.HWND` command outputs the handle of any Logo window, and the `.GETDC` and `.FREEDC` commands provide access to the Device Context for the Graphics Window. The `.MESSAGE` and `.WNDPROC` commands allow you to intercept Windows messages, and the `.WINDOWPOINT` and `.TURTLEPOINT` commands help you in converting between window and turtle coordinates.

Colors

Logo takes advantage of the computer's ability to display graphics and text in various colors. Use of color can give your Logo programs pizzazz and dramatically enhance Logo graphics with little or no additional complication to the Logo procedures that draw them. Contextual text colors help you structure your procedures and immediately identify typing mistakes.

Text Colors

The Listener Window and edit windows automatically display Logo text in colors according to the types of words you type. Logo provides a different color for each type of Logo element, including primitives, procedures, names, numbers and delimiters. This helps you organize your Logo commands and immediately identify typing mistakes. The colors for the different elements can be set and changes in the **Text Colors...** dialog in the **Options** menu.

Colors displayed in the Listener Window can also be controlled by the TEXTBG, TEXTFG, and SETATTR commands. These commands control the color of the text Logo uses for printing and the color of the background on which it appears.

Graphics Colors

Logo graphics can utilize up to 256 different colors depending on the video card in your computer. The SETPC command controls the color of the pen with which the turtle draws. The color of the turtle itself reflects its current pen color.

To select a pen color, you may also click the SETPC button  on the button bar or select the **Turtle/Pen color...** menu command.

The SETBG command determines the background color of the Graphics Window in which the turtle draws.

To select a background color, you can also click the SETBG button  on the button bar or select the menu command **Turtle/Background color...**

Palettes

The colors that the turtle(s) can use to draw are organized into four sets called palettes. These palettes are similar to an artist's palette in that each has a set of 256 different colors representing the available pen and background colors when the palette is active. (If you use Windows in 16-color mode the 256 colors are 16 repetitions of the 16 available colors.) Use the SETPALLET command to change from one palette to another. The four palettes are numbered from 0 to 3.

When you switch from one palette to another, all of the colors displayed on the screen also switch. For example, a line drawn in color 1 of palette 0 switches to color 1 of palette 1 when you switch from palette 0 to palette 1.

Windows is capable of displaying as many colors as the graphics card can support. If Windows is running in 256-color mode, Logo uses a 256-color palette as the default palette for the display device. On a 16-color display, the Logo color palette holds the 16 available colors.

Designing Colors and Palettes

You can design individual colors in a palette by specifying the amount of red, green, and blue tint in each by using the SETCOLOR command.

Combining the ability to switch palettes with the ability to design colors can produce spectacular effects. For example, you can design a colorful graphic with colors from palette 0. Switch to palette 1 and make each color black (`SETCOLOR n [0 0 0]`). As you change each color number, each region of the screen drawn in that color disappears. When all colors have been changed, the screen is blank. Switch back to palette 0 and watch the graphic dramatically reappear.

Customizing Your Logo Environment

Logo has many commands that allow you to adapt it to both your particular computer equipment and to your personal preferences. You can change Logo's appearance and operation as you wish either temporarily from within Logo or store your preferences to record them permanently. Once your preferences are in a disk file, you can adjust your Logo loading procedure to adapt Logo to your wishes every time it loads.

Personalizing Logo

There are many commands from within Logo that change its appearance and operation. If you discover you are repeatedly changing certain aspects of Logo to suit your preferences, you may want to record these changes in a file so that Logo incorporates them automatically for you.

For example, numerical precision, the Logo prompt, and the effect of the function keys are all system names that can be changed as you wish. There are commands to determine graphics scaling, the default number of turtles, colors, and case sensitivity among others.

INIT.LGO

When Logo loads, it automatically looks for a file named INIT.LGO. If that file is present in the Logo startup directory , Logo loads the file and executes any commands that it contains. If you have any commands you would like Logo to execute every time it loads, you can record them in the INIT.LGO file. For example, if you want Logo text to appear as white text on black background rather than the default black text on white background, you can install the command `SETATTR 15` in the INIT.LGO file. You can also include any frequently used utility procedures in INIT.LGO and they are automatically defined in your workspace when Logo loads. An INIT.LGO file can be created with the Logo editor.

Startup Files

A startup file is similar to INIT.LGO, but its application is slightly more specific. While Logo loads INIT.LGO automatically if it is present, startup files are loaded only when you specify. You can use a startup file to set up an environment that you use sometimes but not every time you use Logo.

Any Logo text file can be used as a startup file. To load a startup file when Logo loads, create a command line within the Program Manager with the name of the startup file as a parameter.

Identifying Logo Files

You can also use the File Manager to start Logo. To do this inform the File Manager that all files with the .LGO extension belong to Logo. You can do this by editing the file WIN.INI in the Windows directory. In the section **[Extensions]** add the following entry:

```
lgo=C:\WINLOGO\LOGO.EXE ^.lgo
```

If Logo is in a different directory than C:\WINLOGO, you should substitute this directory name in the command above. This command tells File manager to start Logo every time a file with the extension .LGO is double-clicked. With Windows 3.1, run the following command from the Program Manager:

```
REGEDIT C:\WINLOGO\LOGO.REG
```

This records all necessary information for File Manager in the Windows registration database. To start Logo, however, you should include the Logo program path in your PATH variable.

LOGO.INI

If Logo finds the file LOGO.INI in the Windows startup directory (usually C:\WINDOWS), it will execute the commands contained in this file. These commands cover all settings for Logo which are made via menu selections and dialogs. The contents of LOGO.INI are divided into sections which are described below.

[Desktop] This section contains common elements for the application display.

ButtonBar=TRUE This elements controls the display of the button bar. Set by the **Options/Button bar** menu command.

WindowTitle=id,previd,state,start_x,start_y,width,height

These entries describe the location of each window within Logo. They are written on selection of the **Window/Save layout** menu command. The fields have the following meaning:

WindowTitle	the window title.
id	an internal ID number.
previd	the ID number of the window which is covered by this window.
state	the window state: 0-normal, 1-iconized, 2-maximized.
start_x,start_y	the upper left corner of the window relative to the parent window.
width,height	the window size.

[Editor] This section controls the appearance of the listener and editor windows.

ClearText=FALSE If enabled, this will change the behavior of the listener window. If any previous input line is edited, all lines below the input line are erased before the command is executed. Set by the **Options/Environment** menu command.

TabStops=5 This item sets the tab stops for the listener and editor windows. Set by the **Options/Environment** menu command.

Font=System The name of the text font. Set by the **Options/Text font** menu command.

Size=10 The size of the text font. Set by the **Options/Text font** menu command.

Bold=FALSE If set to TRUE, the font will be displayed in **bold**. Set by the **Options/Text font** menu command.

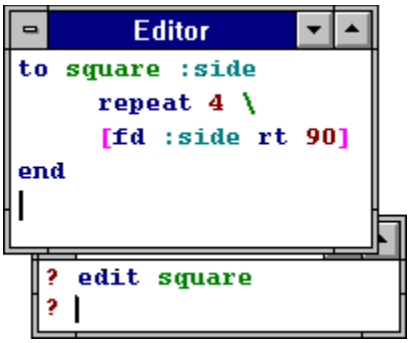
Italic=FALSE If set to TRUE, the font will be displayed in *italics*. Set by the **Options/Text font** menu command.

Underline=FALSE If set to TRUE, the font will be displayed underlined. Set by the **Options/Text font** menu command.

[Colors] This section will contains the color selection for syntax highlighting as selected by the **Options/Text colors** dialog. All color numbers correspond to the Logo system colors. If any of these colors are changed by the `SETCOLOR` command, this will also change the syntax highlighting colors.

Highlight=TRUE If set to FALSE, syntax highlighting is eliminated.

Background=15	The text background. This color assignment can also be changed with the `TEXTBG` and `SETATTR` commands.
Output=0	The color for all system output. This color is also used if no syntax highlighting is desired. This color assignment can also be changed with the `TEXTFG` and `SETATTR` commands.
Word=2	All text not recognized by Logo as a primitive.
Primitive=1	All Logo primitives.
Procedure=9	User-defined procedure names.
Name=3	All Logo names, *i.e.* text strings starting with a colon.
Number=4	All words recognized as numbers.
String=12	All strings starting with a double quote.
Separator=13	All special characters recognized by Logo as separators, like parentheses, brackets, *etc*.
Comment=8	Comments starting with a semicolon.
[Language]	Settings for the Logo kernel. All options are set by the **Options/Environment** dialog.
Atoms=2000	The number of atoms (symbols and numbers).
Nodes=5000	The number of list nodes.
Playback=64	The size of the picture playback buffer, in kilobytes. Decrease this buffer size if memory is a scarce resource.
Init=INIT.LGO	The name of the startup file loaded automatically when Logo starts.
Locals=LOCAL	This items controls the scope of the variables declared as LOCAL. The default setting is LOCAL, which means that these variables are invisible for all subprocedures. Set it to PUBLIC if local variables are to be visible in subprocedures.
[Printer]	This section contains the individual settings for the printer.
AdjustToFit=TRUE	If this element is TRUE, the Logo picture will be scaled to fit on the page. If it is FALSE, the picture will be drawn in exactly the size as it appears on screen. Set by the check box **Adjust graphics to fit page** in the **File/Print** menu command.
[Debug]	This section contains debugging switches.
StackTrace=FALSE	If enabled, the stack trace window will initially displayed. Set by the **Option/Stack Trace** menu command.

Opens a new, empty editor window. Same as the 

Edit button or the EDIT command.

Opens a new editor window and loads the contents of a file.

Loads a file from the disk into the currently selected window. If the Listener or an editor is the currently selected window, a text file is loaded. If the Graphics window is the currently selected window, a graphics image is loaded. Same as the Load button or the `LOAD` and `LOADPIC` commands.

Saves the contents of the currently selected window. If the Listener or and editor is the currently selected window, a text file is saved. If the Graphics window is the currently selected window, a graphics file is saved. Same as the  Save button or the SAVE and SAVEPIC commands.

Saves the contents of the currently selected window under a new name. If the Listener or and editor is the currently selected window, a text file is saved. If the Graphics window is the currently selected window, a graphics file is saved. Same as the  Save button or the SAVE and SAVEPIC commands.

Prints the contents of the current window. Same as the  Print button.

Opens a dialog box which lets you select and customize your printer. Same as the SETPRINTER command.

Closes the Logo workspace and terminates the execution of Logo. Same as the BYE command.

Moves the contents of the current window to the clipboard. In editor windows and the Listener, only selected text is moved.

Copies the contents of the current window to the clipboard. In editor windows and the Listener, only selected text is copied.

Pastes the contents of the clipboard into the current window. Graphics may only be pasted to the Graphics window, while text may only be pasted to a text window.

Inserts the contents of a file into an editor window at the cursor location.

Writes any selected and highlighted text into a file.

Selects and highlights all text within an editor window.

Clear the contents of an editor window or the Listener.

Opens a dialog box where you can enter text to be searched for.

Opens a dialog box where you can specify one text string to be replaced by a second specified text string.

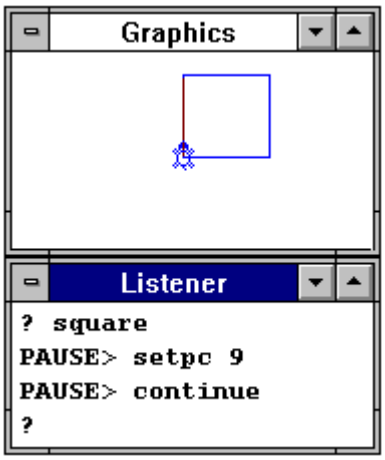
Repeats the last find or replace operation.

Defines the contents of the editor window in the Logo workspace and returns to the Listener.

Saves the contents of the editor window in a file, defines the contents in the Logo workspace and returns to the Listener.

Returns from an editor window to the Listener without defining its contents in the Logo workspace.

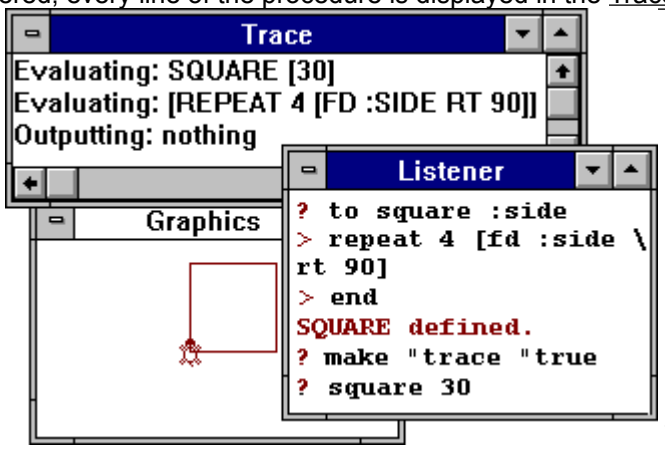
Causes Logo to break a running program, stop all background procedures and timer events and to return to toplevel. Same as the  Break button.



Temporarily halts the execution of a program. Same as the Pause button or the PAUSE command.

Resumes the execution of a paused program. Same as the  Continue button or the CONTINUE command.

Turns single stepping on and off. Same as setting the system name SINGLE.STEP to TRUE or FALSE.

Opens a dialog box where you can select the procedures you want to trace. Every time the procedure is entered, every line of the procedure is displayed in the Trace window as is is being executed. Same as the 

Trace Proces button.

Opens a dialog box where you can select the names you want to trace. The name is displayed in the Names window. By double-clicking the name and its value, you can alter its contents on the fly. Same as the  Trace Names button.

Opens a dialog box where you can select the properties you want to trace. The names of selected properties are displayed in the Properties window. You can select the name and alter its value directly while Logo is running. Same as the  Trace Property button.

Opens a submenu where you can select the trace level you want to see. This command enables global tracing and corresponds to the system names `TRACE` and `TRACE.LEVEL`.

Erases the contents of the Trace window.

Disables global tracing. Same effect as setting the system name `TRACE` to `FALSE`.

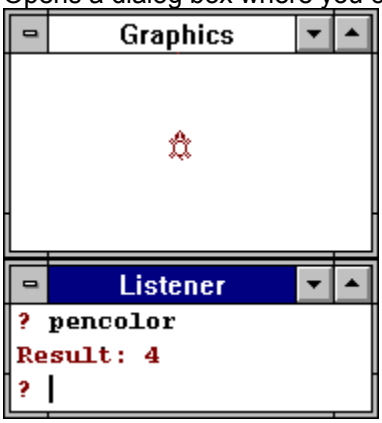
Enables global tracing of user-defines procedures. Same as setting the system name `TRACE` to `TRUE` and the system name `TRACE.LEVEL` to 2.

Enables global tracing of built-in commands. Same as setting the system name `TRACE` to `TRUE` and the system name `TRACE.LEVEL` to 1.

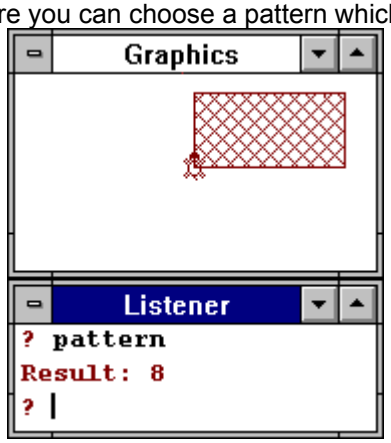
Enables global tracing of both built-in commands and user-defined procedures. Same as setting the system name `TRACE` to `TRUE` and the system name `TRACE.LEVEL` to 3.

Opens a dialog box where you can choose the background color for the Graphics window. Same as the Background button or the `(SETBG)` command.

Opens a dialog box where you can choose the pen color for the turtle(s). Same as the

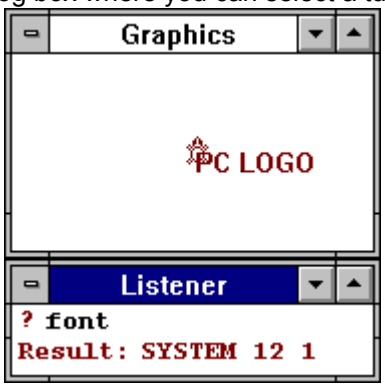


Pen button or the `(SETPC)` command.

Opens a dialog box where you can choose a pattern which the turtles use for subsequent filling operations. Same as the 

Pattern button or the `(SETPATTERN)` command.

Opens a dialog box where you can select a turtle font which is used during the `TURTLETEXT` command.

Same as the 

Turtle Font button or the `(SETFONT)` command.

Opens or closes the Stack window. This window displays the hierarchy of your running procedures.

Displays or hides the Button bar.

Opens a dialog box where you can customize your Logo programming environment.

Opens a dialog box where you can select the font which you want to use in the editor windows and the Listener.

Opens a dialog box where you can choose the color settings used to highlight the various elements of the Logo language.

Cascades all windows.

Tiles all windows.

Moves the icons of all minimized windows to the bottom of the workspace.

Restores the standard layout for the Graphics and Listener windows, where the Graphics window occupies the top 2/3 of the workspace, while the Listener occupies the bottom 1/3. Same as the Standard Layout button or the SPLITSCREEN command.

Saves the current window layout to disk.

Restores a previously saved window layout from disk.

Activates the Listener.

Activates the Stack window.

Activates the Trace window.

Activates the Names window.

Activates the Properties window.

Activates the Graphics window.

Displays the main help index.

Explains the last error.

Finds help for a specific keyword.

Help for the MCI Multimedia command line interpreter.

Provides a complete on-line explanation of how to use the on-line Help system.

Displays the version and copyright notice.

Loads a file from the disk into the currently selected window. If the Listener or an editor is the currently selected window, a text file is loaded. If the Graphics window is the currently selected window, a graphics image is loaded. Same as the **File/Load** menu command or the `LOAD` and `LOADPIC` commands.

Saves the contents of the currently selected window. If the Listener or an editor is the currently selected window, a text file is saved. If the Graphics window is the currently selected window, a graphics file is saved. Same as the **File/Save** menu command or the `SAVE` and `SAVEPIC` commands.

Prints the contents of the current window. Same as the **File/Print** menu command.

Opens a new editor window. Same as the **File/New** menu command or the `EDIT` command.

Opens a dialog box where you can choose the background color for the Graphics window. Same as the **Turtle/Background color...** menu command or the `(SETBG)` command.

Opens a dialog box where you can choose the pen color for the turtle(s). Same as the **Turtle/Pen color...** menu command or the `(SETPC)` command.

Opens a dialog box where you can choose a pattern which the turtles use for subsequent `FILL` commands. Same as the **Turtle/Fill pattern...** menu command or the `(SETPATTERN)` command.

Opens a dialog box where you can select a turtle font which is used during the TURTLETEXT command. Same as the **Turtle/Turtle font...** menu command or the (SETFONT) command.

Opens a dialog box where you can select the procedure(s) you want to trace. Each line of the selected procedure(s0 is displayed in the Trace window as it is being executed. Same as the **Debug/Procedures...** menu command.

Opens a dialog box where you can select the names you want to trace. The name and its value are displayed in the Names window. By selecting the name you can change its value while your procedure is running. Same as the **Debug/Names...** menu command.

Opens a dialog box where you can select the properties you want to trace. The name is displayed in the Properties window. By the name and its properties, you can change the values while your procedure is running. Same as the **Debug/Properties...** menu command.

Resumes execution of a paused program. Same as the **Debug/Continue** menu command or the `CONTINUE` command.

Temporarily halts execution of a program. Same as the **Debug/Pause** menu command or the PAUSE command.

Causes Logo to break a running program, stop all background procedures and timer events and return to toplevel. Same as the **Debug/Break** menu command.

Restores the standard layout for the Graphics and Listener windows, where the Graphics window occupies the top 2/3 of the workspace, while the Listener occupies the bottom 1/3. Same as the **Window/Standard layout** menu command or the `SPLITSCREEN` command.

Opens the help system and displays the Help Index. Same as the **Help/Index** menu command or the HELP command.

Error Messages

Ambiguous filename not allowed.
Attempt to draw a dot outside the screen.
Attempt to move turtle nnnn outside the fence.
Bad number syntax.
Cannot access the clipboard.
Cannot allocate requested amount of memory.
Can't find catch for "symbol".
Cannot initialize properly.
Cannot print window contents.
Cannot start timer.
Cannot tell turtle n; there are only x turtles defined.
Division by zero.
FATAL ERROR: Unable to obtain device context.
FATAL ERROR: Garbage collection failed.
File "name" is too large to fit into buffer.
File "name" not found.
File stream nnnn not open.
Input/output (I/O) error.
Internal error: "text"
Math overflow.
MCI error.
"name" is already in use. Try a different name.
"name" is not a Logo procedure.
"name" is not a Logo name.
"name" needs more input(s).
No more file structures for OPEN or CREATE.
No printer defined.
Out of atom space.
Out of memory.
Printer not ready.
Service not available in Windows 3.0.
The procedure "name" does not like "value" as input.
The procedure "name" needs ... as its ... input.
Too many recursive procedure calls.
Turtle(s) nnnn must be inside window.
Unable to load picture "name".
Unable to save options.
"value" is not a valid input for "variable".
You don't say what to do with the output of "name".

FATAL ERROR: Unable to obtain device context.

This fatal error occurs only when Windows runs out of system resources. Close all applications and restart Windows.

FATAL ERROR: Garbage collection failed.

There was an internal error in the garbage collector. Try to isolate the error and contact Harvard Associates, Inc. for help.

Cannot allocate requested amount of memory.

There is not enough memory for Logo to initialize properly. Close one or more applications, check your free system resources and restart PC Logo for Windows again.

Cannot initialize properly.

An error occured during the initialization of PC Logo for Windows.

Out of atom space.

Logo does not have any more memory in which it can store numbers and names. To create additional atom space, you must erase variables that are currently in workspace.

Out of memory.

You have run out of workspace memory. Erase names and procedures to free additional memory for workspace.

Too many recursive procedure calls.

There is a limit to the number of times a procedure can call itself unless you use tail recursion which has no limit. To avoid this error, rewrite the procedure(s) to use tail recursion.

Ambiguous filename not allowed.

Special DOS characters * and ? are not allowed in file names.

File "name" not found.

The file you requested to load was not found in the current directory on the current disk.

Input/output (I/O) error.

There is a problem in communication between the computer and peripheral devices such as the disk drive or printer.

File stream nnnn not open.

The file stream you specified has not been opened. See OPEN or CREATE for more information.

No more file structures for OPEN or CREATE.

You can only `OPEN` or `CREATE` 15 file streams at once. You must `CLOSE` a file stream to `OPEN` or `CREATE` another.

"name" is already in use. Try a different name.

You cannot give a procedure the same name as a primitive.

Bad number syntax.

The number you have entered is not a valid number.

You don't say what to do with the output of "name".

You have used a command or procedure that outputs a result, but have not assigned the result to a variable.

The procedure "name" does not like "value" as input.

The primitive or procedure you have used requires a different type of input than you have supplied.

"value" is not a valid input for "variable".

You have tried to assign an invalid value to a system variable.

"name" is not a Logo procedure.

The name you have used is not a Logo procedure. If the name you have used is a variable, you must use proper syntax to indicate that to Logo.

"name" is not a Logo name.

You have tried to use a Logo variable that has not been created or is not available to the procedure which used it. Use MAKE to create global variables that are available to all procedures.

"name" needs more input(s).

The command or procedure you called needs more input(s) than you have supplied.

Can't find catch for "symbol".

You have used the `THROW` command without a corresponding `CATCH` clause.

Math overflow.

Your computations are either too large or too small to be stored correctly by Logo.

Division by zero.

You have used 0 as a divisor in your mathematical calculations.

Internal error: "text"

Logo has encountered an unexpected internal condition which keeps it from continuing. If possible, please isolate the steps that caused this error and report them to Harvard Associates, Inc. along with the error message displayed. You should reboot the computer after this error.

The procedure "name" needs ... as its input.

The command you used requires a different value than you supplied as its input.

Attempt to move turtle nnn outside the fence.

In FENCE mode the turtle cannot be moved off the screen. See also WINDOW and WRAP modes.

Turtle(s) nnnn must be inside window.

You have moved one or more turtles off the screen while in `WINDOW` mode and then tried to switch to `WRAP` or `FENCE` mode. Move all the turtles onto the screen before switching modes.

Cannot tell turtle n; there are only x turtles defined.

You tried to activate a turtle with a number greater than the number of turtles defined. Use `SETTURTLES` to increase the number of turtles.

Attempt to draw a dot outside the screen.

A dot may only be drawn within the dimensions of the computer screen.

Unable to load picture "name"

The picture file which you tried to load cannot be displayed on this screen or has a file format incompatible with PC Logo for Windows.

No printer defined.

There is no printer defined. Before printing, use the Windows System Control utility to define a printer.

Cannot print window contents.

The contents of the selected window cannot be printed with the selected printer.

Printer not ready.

The selected printer is not ready. Please verify that the printer is connected, on line, and ready with paper.

Unable to save options.

There was a write error when Logo tried to save the current options into the `LOGO.INI` file.

File "name" is too large to fit into buffer.

The edit buffer is too small to hold the entire file. Split the file before attempting to load it for editing.

Cannot access the clipboard.

There was an internal error while copying to or pasting from the clipboard.

Cannot start timer.

An attempt to start a timer failed because all Windows timers were busy.

Service not available in Windows 3.0.

Certain advanced features like the `MCI` command are only available in Windows versions 3.10 and up.

MCI error: "text"

When using the MCI command, the MCI command interpreter returned the specified error message.

Alphabetical list of commands

ABS Outputs the absolute value of its input
AGET Accesses an array element
AND Performs a logical AND on its inputs
APPLY Applies a function to a list
ARCTAN Outputs the arc tangent of its input
ARRAY Defines an array
ARRAY? Checks for its input being an array
ARRAYDIMS Outputs the array dimensions of its input
ASCII Outputs the ASCII number code of its input
ASET Stores a value in an array
ASK Applies a run list to specific turtles

BACK Moves the turtles backwards
BACKGROUND Outputs the background color
BASE System variable; sets the conversion base for numerical output
BGPATTERN Outputs the background pattern number
BURIEDNAMES Outputs a list of all buried names
BURIEDPROCS Outputs a list of all buried procedures
BURIEDPROPS Outputs a list of all buried properties
BURY Buries an object
BURYALL Buries every object
BURYNAME Buries a name
BURYPROC Buries a procedure
BURYPROP Buries a property
BUTFIRST Outputs all but the first element of a list or a word
BUTLAST Outputs all but the last element of a list or a word
BUTMEMBER Removes a member from a list or a word
BUTTON? Checks if a mouse button is pressed
BYE Leaves PC Logo for Windows
BYTEARRAY Defines an array of byte values
BYTEARRAY? Checks for its input being a byte array

CASE Enables case conversion
CASE? Checks for case conversion
CATCH Catches any error or `THROW` command
CHAR Outputs the character for its ASCII input
CLEAN Cleans the graphics screen
CLEARINPUT Clears the keyboard buffer
CLEARSCREEN Clears the Graphics window
CLEARTEXT Clears the Listener window
CLOSE Closes a file
COLOR Outputs the color code of a specific color
CONST Defines a text constant
CONTENTS Outputs a list of all objects
CONTINUE Continues after a `PAUSE`
COPYDEF Copies a procedure definition
COPYOFF Turns off the Listener window protocol
COPYON Turns on the Listener window protocol

COS	Outputs the cosine of its input
COUNT	Counts the size of its input
CREATE	Creates a file
CURDIR	Outputs the current working directory
DATE	Outputs the current date
DEFEVENT	Defines an event handler
DEFINE	Defines a procedure
DEFINED?	Checks for a procedure being defined
DELETE	Deletes a file
DELIMITER	System variable; sets the string delimiter
DIRECTORY	Outputs a list of files
DISK	Outputs the current disk
DOT	Draws a dot
DOTCOLOR	Outputs the color of a dot
DRAW	Resets all turtles and the Graphics window
EACH	Sequentially executes a command list for all active turtles
EDIT	Opens an edit window
EDN	Opens an edit window and fills it with all defined Logo names
ELSE	Part of the `IF` command
EMPTY?	Checks for its input being an empty list
END	Ends a procedure definition
EQUAL?	Checks for its inputs being equal
ERASE	Erases procedure definitions, names and properties
ERC	Erases all constants
ERN	Erases names
ERROR	System variable that contains the code of the last error
EVAL	Evaluates its input
EVENT	Generates an event
EXPN	Outputs the *e* value of its input
EXTENT	Outputs the extents of the Graphics window
FENCE	Sets the Graphics window to fence mode
FILE.INFO	Outputs information about a file
FILE?	Verifies the presence of the specified file
FILL	Fills an area in the Graphics window
FILLARRAY	Fills an array with values
FIRST	Outputs the first element of a list or a word
FKEY.n	System variables `FKEY.2` to `FKEY.10` contain strings assigned to function keys
FONT	Outputs the current turtle font
FONTS	Outputs a list of available fonts
FOR	Begins a `for...next` loop
FORWARD	Moves the turtles forward
FPUT	Adds the specified element to the beginning of a list or word
.FREEDC	Releases a previously obtained device context
FROMMEMBER	Outputs a list where the first elements are removed
FULLSCREEN	Maximizes the Graphics window
GETATTR	Outputs the current color attributes of the Listener window
GETBYTE	Outputs the next byte of the input stream
GETBYTE.NO.ECHO	Outputs the next byte of the input stream without echo
.GETDC	Obtains a device context for use within Windows
GETMODE	Outputs the number of the current screen mode
GETPALLET	Outputs the current palette number

GETXY	Outputs the coordinates of the first active turtle
GO	Transfers execution control to a LABEL command
GPROP	Outputs a property
HALT	Halts a background procedure
HEADING	Outputs the heading of the first active turtle
HELP	Opens the help system
.HINST	Outputs the instance handle for PC Logo for Windows
HIDETURTLE	Hides all turtles
HOME	Moves all turtles home
.HWND	Outputs the window handle of the PC Logo for Windows frame window
IBASE	System variable specifying the base in which numbers are input
IF	Conditional execution
IFFALSE	Runs a run list if the preceding TEST command yields FALSE
IFTRUE	Runs a run list if the preceding TEST command yields TRUE
IGNORE	Ignores the output of a command
INT	Truncates its input to an integer
ITEM	Returns a specific element of a list or a word
KEY?	Checks for the presence of input in the keyboard buffer
LABEL	Defines a branch label
LAST	Returns the last element of a list or a word
LAUNCH	Launches a background procedure
LEFT	Turns the turtles left
LIST	Creates a list
LIST?	Verifies that its input is a list
LISTARRAY	Converts an array into a list
LOAD	Loads procedure defintions, names and constants into workspace
LOADPIC	Loads a picture into the Graphics window
LOADSNAP	Loads a snapped image or a bitmap file
LOCAL	Defines a local variable
LOG	Outputs the logarithm of its input
LOG10	Outputs the base 10 logarithm of its input
LOGAND	Performs a bitwise AND on its inputs
LOGNOT	Performs a bitwise NOT on its input
LOGOR	Performs a bitwise OR on its inputs
LOGXOR	Performs a bitwise XOR on its inputs
LPUT	Adds an element to the end of a word or list
LSH	Performs a bitwise shift operation on its input
MAKE	Assigns a value to a name
MCI	Provides access to the Windows 3.1 multimedia extensions
MCI?	Checks for the presence of the Windows 3.1 multimedia extensions
MEMBER?	Checks for an element being a member of the specified word or list
.MESSAGE	Establishes a message processing procedure
MOUSE	Outputs the current mouse coordinates
.MOUSEON	Checks for mouse presence
MOUSESHAPE	Outputs the shape of the mouse cursor
NAME	Assigns a value to a name
NAME?	Verifies that its input is a Logo name
NOCASE	Turns off case conversion
NODES	Outputs the number of available list elements

NOT	Performs a logical NOT on its input
NUMBER?	Checks if its input is a number
OPEN	Opens a file
OR	Performs a logical OR on its inputs
ORIGIN	Outputs the coordinate system origin for the current turtle(s)
OUTPUT	Outputs a value from a procedure
PATTERN	Returns the fill pattern number of the first active turtle
PAUSE	Pauses the execution of a procedure
PEEKBYTE	Returns the next byte of an input stream without reading it
PEN	Outputs the pen state of the first active turtle
PENCOLOR	Outputs the pen color of the first active turtle
PENDOWN	Sets the pen down
PENERASE	Sets the pen color to the background color
PENREVERSE	Sets the pen to invert the colors under the pen
PENUP	Lifts the pen up
PI	Outputs
PICK	Randomly picks an element
PLAY	Plays notes or a sound file
PLIST	Outputs the proerty list of its input
POC	Prints all text constants
PONS	Prints all names
POPLS	Prints all property lists
POPS	Prints all procedures
POTS	Prints the names of all procedures
PPROP	Stores a property
PPROPS	Stores a property list
PRECISION	System variable; sets the numerical output precision
PRINT	Prints its inputs
PRINTER	Outputs information about the printer
PRINTLINE	Prints numerical input as ASCII
PRINTOUT	Prints procedures, names, properties and constants
PRINTSCREEN	Prints the Graphics window
PROCLIST	Outputs a list of all user-defined procedures
PRODUCT	Multiplies its inputs
PROMPT	System variable; sets the prompt string
PRTRACE	Prints to the Trace window
PUBLIC	Defines a public variable
PUTBYTE	Writes a byte to the output stream
QUOTIENT	Divides its inputs
RANDOM	Outputs a random number
READ	Reads a Logo object
.READ	Reads binary or text data from a file
READCHAR	Reads a character
READLINE	Reads a line as a list of numbers
READLIST	Reads a Logo list
READQUOTE	Reads a line as a Logo word
RECYCLE	Performs a garbage collection
REMAINDER	Outputs the remainder of its inputs
REMPROP	Removes a property
RENAME	Renames a file
REPEAT	Repeats execution of a run list the specified number of times
RERANDOM	Re-initializes the random number generator

RIGHT	Turns the turtles right
ROUND	Round its input to the nearest integer
RUN	Executes a list
SAVE	Saves the workspace into a file
SAVEPIC	Saves the contents of the Graphics window
SAVESNAP	Saves a snapped image to disk
SCREENFACTS	Outputs information about the Listener and Graphics windows
.SEEK	Moves the read/write pointer in a file
SENTENCE	Combines its inputs
SETATTR	Sets the colors attributes of the Listener window
SETBG	Sets the background color of the Graphics window
SETBGPATTERN	Sets the background pattern of the Graphics window
SETCOLOR	Defines a color
SETCURDIR	Defines the current working directory
SETDISK	Changes the current disk
SETEXTENT	Changes the logical size of the Graphics window
SETFONT	Changes the turtle font
SETHEADING	Sets the heading of all turtles
SETMOUSESHAPE	Defines the shape of the mouse cursor
SETORIGIN	Defines the coordinate system origin for the current turtle(s)
SETPALLET	Changes the current palette
SETPATTERN	Sets the fill pattern
SETPC	Sets the pen color
SETPEN	Sets pen color and pen state
SETPRINTER	Defines a printer
SETSHAPE	Defines the turtle shape
SETSPEED	Sets the turtle speed
SETTURTLEFACTS	Sets turtle attributes
SETTURTLES	Defines the number of turtles
SETWIDTH	Sets the pen width
SETWINSIZE	Sets the physical size of the Graphics window
SETX	Sets the X coordinate for all turtles
SETXY	Sets both the X and the Y coordinates for all turtles
SETY	Sets the Y coordinate for all turtles
SHAPE	Outputs the shape of the first active turtle
SHOW	Prints its inputs
SHOWN?	Checks the visible state of the first active turtle
SHOWTURTLE	Displays all turtles
SIN	Outputs the sine of its input
SINGLE.STEP	System variable which invokes single stepping
SNAP	Saves an area of the Graphics window
SNAPSIZE	Outputs the size of a snapped image
SPEED	Outputs the turtle speed
SPLITSCREEN	Sets the standard window layout
SQRT	Returns the square root of its input
STAMP	Draws a previously saved screen area
STAMPOVAL	Draws a circle or ellipse
STAMPRECT	Draws a rectangle
STANDARD.INPUT	System variable which inputs channel number
STANDARD.OUTPUT	System variable which outputs channel number
STOP	Halts execution of a procedure
SUBDIR	Outputs a list of subdirectories
SUM	Adds its inputs
TAB	System variable which sets the tab stop position

TELL	Activates turtles
TELLALL	Activates a range of turtles
TEST	Tests its input
TEXT	Outputs the list representation of a procedure
TEXTARRAY	Converts a byte array into a text string
TEXTBG	Outputs the background color of the Listener window
TEXTFG	Outputs the foreground color of the Listener window
TEXTSCREEN	Maximizes the Listener window
THEN	Part of the `IF` command
THING	Outputs the value associated with its input
THROW	Throws an object to a `CATCH` command
TIME	Outputs the current time
TIMER	Starts a timer
TO	Starts the definition of a procedure
TONE	Sounds a tone
TOPLEVEL	Returns to toplevel
TOWARDS	Outputs the heading to a given coordinate
TRACE	System variable; enables procedure tracing
TRACE.LEVEL	System variable; defines the level of tracing information
TRACED	Outputs the list of traced objects
TROFF	Turns on tracing for specific objects
TRON	Turns off tracing for specific objects
TURTLEFACTS	Outputs information about the first active turtle
.TURTLEPOINT	Converts window coordinates to turtle coordinates
TURTLES	Outputs the number of available turtles
TURTLETEXT	Prints its inputs at the current turtle location
TYPE	Prints its inputs
UNBURY	Unburies the specified object
UNBURYALL	Unburies all objects
UNBURYNAME	Unburies the specified name
UNBURYPROC	Unburies the specified procedure
UNBURYPROP	Unburies the specified property
UNGETBYTE	Pushes a byte back into the input stream
VERSION	Outputs the version of PC Logo for Windows
WAIT	Waits a specified time
WHILE	Repeats execution of a run list while the specified condition is true
WHO	Outputs the number of the first active turtle
WIDTH	Outputs the pen width of the first active turtle
WINDOW	Sets the Graphics window to window mode
.WINDOWPOINT	Converts turtle coordinates to window coordinates
.WINDOWS	Calls the Windows API
.WINDOWSL	Calls the Windows API
WINSIZE	Outputs the physical size of the Graphics window
WINVER	Outputs the version of the Windows operating environment
.WNDPROC	Calls the default message handler for PC Logo for Windows
WORD	Creates a word
WORD?	Verfies that its input is a word
WRAP	Sets the Graphics window to wrap mode
.WRITE	Writes binary or text data into a file
XCOR	Outputs the `X` coordinate of the first active turtle
YCOR	Outputs the `Y` coordinate of the first active turtle

`/`	Infix operator; division
`==`	Infix operator; equal to
`>`	Infix operator; greater than
`>=`	Infix operator; greater than or equal to
`<`	Infix operator; less than
`<=`	Infix operator; less than or equal to
`-`	Infix operator; subtraction
`*`	Infix operator, multiplication
`+`	Infix operator; addition
`--`	Start of a comment

; semi-colon

Syntax

```
; comment
```

Explanation

The semicolon causes Logo to ignore all characters from the right of the semicolon to the end of the line (the carriage return). The semicolon is useful for making comments within procedures or Logo files.

The semicolon can be used to write, load, and save comments only in the editor. To save space, Logo removes anything following a semicolon when loaded into toplevel. If a file containing comments is loaded at toplevel, semicolons and comments are not displayed.

Examples

```
? ;THE FOLLOWING PROCEDURE DRAWS A SQUARE
? TO SQUARE
>    REPEAT 4 [FD 40 RT 90]
> END
SQUARE defined
? TO TURN
>    DRAW
>    FD 30 ;TURTLE MOVES 30 STEPS FORWARD
>    RT 85 ;TURTLE TURNS RIGHT 85 DEGREES
>    FD 30 ;TURTLE MOVES 30 STEPS FORWARD
> END
DRAW defined
? PO TURN
TO TURN
   DRAW
   FD 30
   RT 85
   FD 30
END
? _
```

/ (division)

Syntax

```
number / number
```

Explanation

/ outputs the result of the first input divided by the second. / can also be used as a prefix operation, like **QUOTIENT**.

Example

```
? 8 / 2
Result: 4
? _
```

= (equals)

Syntax

```
object = object
```

Explanation

= outputs `TRUE` if its two inputs are equal. Its inputs may be numbers, words, or lists. = can also be used as a prefix operation with two inputs.

Examples

```
? 6 = 6
Result: TRUE
? 6 = 66
Result: FALSE
? "AZURE = "AZURE
Result = TRUE
? [SPRING GREEN] = [SPRING GREEN]
Result: TRUE
? _
```

Note that if one input is a word and the other input is a list, they are not equal even if their contents are the same.

```
? "AZURE = [AZURE]
Result = FALSE
? _
```

> (greater than)

Syntax

```
object > object
```

Explanation

> outputs TRUE if the first input is greater than the second input; otherwise, > outputs FALSE. > may also be used as a prefix operation with two inputs.

Note that letters can be compared with > as well as numbers. The letter A has the lowest value and Z the highest.

Examples

```
? 8 > 3
Result: TRUE
? 3 > 8
Result: FALSE
? 3 > 3
Result: FALSE
? "A > "F
Result: FALSE
? "RED > "BLUE
Result: TRUE
? "RED > "ROSE
Result: FALSE
? "ROSE > "RED
Result: TRUE
? _
```

>= (greater than or equal to)

Syntax

```
object >= object
```

Explanation

>= outputs TRUE if the first input is greater than or equal to the second input; otherwise, >= outputs FALSE. >= may also be used as a prefix operation with two inputs.

Note that letters can be compared with >= as well as numbers. The letter A has the lowest value and Z the highest.

Examples

```
? 8 >= 3
Result: TRUE
? 3 >= 8
Result: FALSE
? 3 >= 3
Result: TRUE
? "F >= "A
Result: TRUE
? "GREEN >= "BLUE
Result: TRUE
? "BLUE >= "BROWN
Result: FALSE
? _
```

< (less than)

Syntax

```
object < object
```

Explanation

< outputs TRUE if the first input is less than the second input; otherwise, < outputs FALSE. < may also be used as a prefix operation with two inputs.

Note that letters can be compared with < as well as numbers. The letter A has the lowest value and Z the highest.

Examples

```
? 3 < 8
Result: TRUE
? 8 < 3
Result: FALSE
? 3 < 3
Result: FALSE
? "A < "F
Result: TRUE
? "APPLE < "BANANA
Result: TRUE
? "BANANA < "BASEBALL
Result: TRUE
? _
```

<= (less than or equal to)

Syntax

```
object <= object
```

Explanation

<= outputs TRUE if the first input is less than or equal to the second input; otherwise, <= outputs FALSE. <= may also be used as a prefix operation with two inputs.

Note that letters can be compared with <= as well as numbers. The letter A has the lowest value and Z the highest.

Examples

```
? 3 <= 8
Result: TRUE
? 8 <= 3
Result: FALSE
? 3 <= 3
Result: TRUE
? "A <= "F
Result: TRUE
? "BLUE <= "GREEN
Result: TRUE
? "BLUE <= "BROWN
Result: TRUE
? "BROWN <= "BLUE
Result: FALSE
? _
```

- (minus)

Syntax

```
number - number
- number number
(- number number number ...)
```

Explanation

- outputs the difference between the first and second inputs. - can also be used to indicate a negative number. A negative number consists of a minus sign followed by the number with no space in between.

- can be used as a prefix operation with two inputs.

Examples

```
? 7 - 5
Result: 2
? 5 - 7
Result: -2
? _
```

The minus sign can be used in several ways: as an infix operation, as a prefix operation, and to indicate a negative number.

```
? -14 - 5 ; negative 14 minus 5
Result: -19
? -14 - -5 ; negative 14 minus negative 5
Result: -9
? - 14 5 ; prefix operation: 14 minus 5
Result: 9
? _
```

* (multiplication)

Syntax

```
number * number
* number number
(* number number number ...)
```

Explanation

* outputs the product of the two inputs. * can also be used as a prefix operation with two inputs, like PRODUCT.

Examples

```
? 3 * 8
Result: 24
? -3 * 8
Result: -24
? .3 * 8
Result: 2.40
? * 2 6
Result: 12
? (* 2 3 4)
Result: 24
? _
```

+ (addition)

Syntax

```
number + number
+ number number
(+ number number number ...)
```

Explanation

+ outputs the sum of its two inputs. + may also be used as a prefix operation with two inputs, like SUM.

Examples

```
? 5 + 7
Result: 12
? 8 + -3
Result: 5
? 5 + 7 + 4
Result: 16
? _
```

The following example uses + as a prefix operation:

```
? + 5 7
Result: 12
? (+ 6 5 4 3 2 1)
Result: 21
? _
```

.HINST

Syntax

```
.HINST
```

Explanation

`.HINST` outputs the *instance handle* of PC Logo for Windows. This is a value needed as a parameter for Windows API calls.

See also `.WINDOWS`.

Example

The following procedure loads a string with a given integer resource ID from the resource part of the PC Logo for Windows executable file.

```
TO LOADSTRING :N
   LOCAL "ARRAY
   MAKE "ARRAY BYTEARRAY 128
   IF (.WINDOWS "LoadString .HINST :N :ARRAY 128) = 0 \
       THEN OUTPUT "FALSE \
       ELSE OUTPUT TEXTARRAY :ARRAY
END
```

```
? LOADSTRING 1001
Result: Opens an editor window
? _
```

.HWND

Syntax

```
.HWND
(.HWND name)
```

Explanation

`.HWND` outputs the *window handle* of the PC Logo for Windows main window. This value is needed as a parameter for many Windows API calls. You can obtain the window handle of any window of PC Logo for Windows by supplying its title as an optional input. If the window cannot be found, 0 is output.

See also `.HINST` and `.WINDOWS`.

Example

The following procedure outputs the window title of any window. To output the title of a window, you must supply the window handle for the window.

```
TO GETWINTEXT :HANDLE
   (LOCAL "ARRAY "DUMMY)
   MAKE "ARRAY BYTEARRAY 128
   MAKE "DUMMY (.WINDOWS "GetWindowText :HANDLE :ARRAY 128)
   OUTPUT TEXTARRAY :ARRAY
END
```

```
? GETWINTEXT .HWND
Result: PC Logo for Windows
? _
```

.GETDC

Syntax

```
.GETDC windowhandle
```

Explanation

.GETDC allows use of the Windows graphics engine to draw in a PC Logo for Windows window. The required input is the handle for the window which is to be used for output. .GETDC outputs a number which can be used as a device context required by the Windows kernel as a parameter for further drawing operations. It is essential to release this device context number with the .FREEDC command, since the number of device contexts which Windows can manage is limited. Using .GETDC without a corresponding .FREEDC command may cause Windows to crash.

See also .TURTLEPOINT, .WINDOWPOINT and .FREEDC.

Example

```
? MAKE "MY.DC .GETDC .HWND
? :MY.DC
Result: 2966
? .FREEDC .HWND
? _
```

.FREEDC

Syntax

```
.FREEDC windowhandle
```

Explanation

.FREEDC allows use of the Windows graphics engine to draw in a PC Logo for Windows window.The required input is the handle for the window which was used as input for a previous .GETDC command. .FREEDC releases any device context previously obtained by the .GETDC command and returns it to Windows. It is essential to release this device context number with the .FREEDC command, since the number of device contexts which Windows can manage is limited. Using .GETDC without a corresponding .FREEDC command may cause Windows to crash.

See also .TURTLEPOINT, .WINDOWPOINT and .GETDC.

Example

```
? MAKE "MY.DC .GETDC .HWND
? :MY.DC
Result: 2966
? .FREEDC .HWND
? _
```

.MESSAGE

Syntax

```
.MESSAGE messagenumber procname
```

Explanation

The .MESSAGE command allows you to create a Logo procedure which responds to a Windows message. The Windows operating system communicates with PC Logo by sending messages. When you select a menu item, Windows sends a message which contains a code meaning "a menu item was selected" along with the number of the selected item. The inputs to .MESSAGE are the number which your procedure should react to and the name of your procedure.

The procedure receives a list of three numbers as input. The first is the message number, the second is the contents of the *wParam* parameter, while the third is the contents of the *lParam* parameter.

If you define a message handling procedure, all messages arriving with the number you requested are routed to your specifed procedure. Other messages should be forwarded to PC Logo itself by using the .WNDPROC command.

You can turn off the message handler by calling the .MESSAGE command with the message number and FALSE as its second input.

A thorough knowledge of the Windows API and its messaging system is needed to safely respond to messages. It is easy to crash PC Logo or Windows with a wrong message handler.

See also .WNDPROC.

Example

The following procedure responds to the menu item **"Help/Last error"**. When this menu item is selected, the message "HELP / LAST ERROR SELECTED" is displayed. All other menu selections are routed to PC Logo so PC Logo remains fully functional.

```
TO HELP_HANDLER :ARGS
   IF (ITEM 2 :ARGS) = 1602 \
       THEN PR [HELP / LAST ERROR SELECTED] \
       ELSE .WNDPROC :ARGS
END
```

```
? .MESSAGE 273 "HELP_HANDLER
? _
```

.MOUSEON

Syntax

.MOUSEON

Explanation

.MOUSEON outputs TRUE if a mouse is present and FALSE if not.

See also BUTTON? and MOUSE.

Example

```
? .MOUSEON
Result: TRUE
? _
```

.READ

Syntax

```
.READ streamnumber bytearray size
(.READ streamnumber bytearray)
```

Explanation

`.READ` transfers data from a file into a `BYTEARRAY`. The first input is the stream number and the second input is the `BYTEARRAY` where the data is to be stored. The third input is the number of bytes to transfer. If the third input is missing, the `BYTEARRAY` is filled completely.

The output of `.READ` is the number of bytes transferred. If no bytes are transferred because the end of the file is reached, the output is the value `"EOF`.

If the file is open in normal mode, a maximum of one line, terminated by the line feed character, is transferred. The last byte is appended with a byte of value 0. You can then use the `TEXTARRAY` command to convert the contents of the `BYTEARRAY` to a Logo word. If the file is open in binary mode, however, the data is transferred unchanged, including all carriage return and line feed characters.

See also `OPEN`, `.SEEK` and `.WRITE`.

Example

The following example opens a file for reading and reads the first 128 bytes in binary mode.

```
TO READ.128.BYTES :NAME
    MAKE "STREAM (OPEN :NAME "RB)
    MAKE "DATA BYTEARRAY 128
    (.READ :STREAM :DATA)
    CLOSE :STREAM
END
```

.SEEK

Syntax

```
.SEEK streamnumber offset
(.SEEK streamnumber offset mode)
(.SEEK streamnumber)
```

Explanation

`.SEEK` positions the read/write pointer of the specified stream to the numerical byte position given as its second input. The output of the `.SEEK` command is the current stream position. Note that it is possible to position beyond the end of a file.

If `.SEEK` is supplied with an optional third input, this input affects the way the pointer is positioned:

0 Position = offset; equivalent to `.SEEK streamnumber offset`
1 Position = Current position + offset
2 Position = End-of-file + offset

Using `.SEEK` with the stream number only as input yields the current position of the stream pointer.

See also `.READ` and `.WRITE`.

Example

The following example opens a file for reading and writing, seeks to position 32 and writes the letter "X at that position.

```
TO WRITE.X.AT.32 :NAME
    MAKE "STREAM (OPEN :NAME "RW)
    MAKE "DATA BYTEARRAY 1
    ASET :DATA 0 #H58
    .SEEK :STREAM 32
    .WRITE :STREAM :DATA 1
    CLOSE :STREAM
END
```

.TURTLEPOINT

Syntax

```
.TURTLEPOINT [xvalue yvalue]
```

Explanation

`.TURTLEPOINT` allows use of the Windows graphics engine to draw in the Graphics window. Its input is a list of two integers describing a dot inside the Graphics window in pixel coordinates, where [0 0] is the upper left corner of the Graphics window. The output of `.TURTLEPOINT` is a list of two integers describing the same point in turtle coordinates with respect to the current window extent and coordinate system origin.

See also `.WINDOWPOINT`, `.GETDC` and `.FREEDC`.

Examples

```
? .TURTLEPOINT [0 0] ; upper left corner
Result: [-200 100]
? .WINDOWPOINT [-200 100]
Result: [0 0]
? _
```

.WINDOWPOINT

Syntax

```
.WINDOWPOINT [xvalue yvalue]
```

Explanation

`.WINDOWPOINT` allows use of the Windows graphics engine to draw in the Graphics window. Its input is a list of two integers describing a dot inside the Graphics window in turtle coordinates with respect to the current window extent and coordinate system origin. The output of `.WINDOWPOINT` is a list of two integers describing the same point in pixel coordinates, where [0 0] is the upper left corner of the Graphics window.

See also `.TURTLEPOINT`, `.GETDC` and `.FREEDC`.

Examples

```
? .TURTLEPOINT [0 0] ; upper left corner
Result: [-200 100]
? .WINDOWPOINT [-200 100]
Result: [0 0]
? _
```

.WINDOWS

Syntax

```
.WINDOWS "function-name
.WINDOWS [function-name DLL-name]
(.WINDOWS "function-name arguments)
.WINDOWSL "function-name
.WINDOWSL [function-name DLL-name]
(.WINDOWSL "function-name arguments)
```

Explanation

`.WINDOWS` provides a limited interface to the Windows API. Its first input is either the name of the Windows API function or a list of two elements. The first element of that list is the function name and the second element is the name of the DLL where the function is located. Optional function parameters may be supplied. They will be pushed on the stack in Pascal order. Numbers are pushed as 16-bit integers, so if the function requires a 32-bit argument, you are required to supply two numbers. Symbols are pushed as ASCIIZ strings. Lists are converted to an ASCIIZ string. A bytearray may be used as a buffer for return values or for structures. When you supply a bytearray, the address of its data area is pushed. The output is the 16-bit integer outcome of the Windows API function. For converting output in a bytearray into a Logo word, use `TEXTARRAY`.

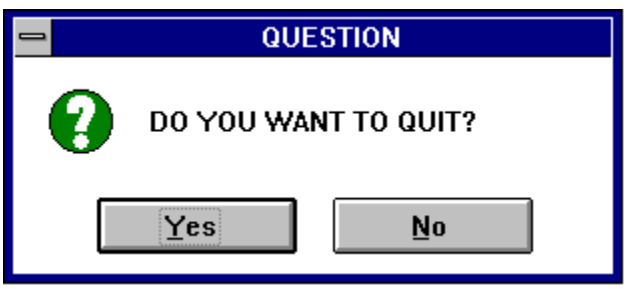
You can use the `.WINDOWSL` command to obtain return values of 32 bit integers. This command requires a thorough knowledge of Windows API calls. Supplying wrong parameters to the function can crash Windows.

Example

The following procedure displays a message box showing the text you supply as input along with `YES` and `NO` buttons. Depending on which button you select, the procedure outputs `"TRUE` or `"FALSE`. To display the message box, the Windows API function `MessageBox` is used.

```
TO QUERY :MSG
   LOCAL "ANSWER
   MAKE "ANSWER (.WINDOWS "MESSAGEBOX 0 :MSG "QUESTION 36)
   IF :ANSWER = 6 THEN OUTPUT "TRUE
   OUTPUT "FALSE
END
```

```
? QUERY [DO YOU WANT TO QUIT?]
Result: FALSE
? _
```



.WNDPROC

Syntax

```
.WNDPROC message-list
```

Explanation

`.WNDPROC` calls the Windows message callback procedure attached to the main window of the Logo programming environment. Its input is a list of three numbers corresponding to the message's *nMsg*, *wParam* and *lParam* parameters, respectively.

`.WNDPROC` is used with a message handling event procedure installed with the `.MESSAGE` command. When this procedure does not process the message and its contents, it forwards the message to the Logo programming environment to be handled there.

Example

The following procedure responds to the menu item **"Help/Last error"**. When this menu item is selected, the message `HELP / LAST ERROR SELECTED` is displayed. All other menu selections are routed to PC Logo so PC Logo remains fully functional.

```
TO HELP_HANDLER :ARGS
   IF (ITEM 2 :ARGS) = 1602 \
       THEN PR [HELP / LAST ERROR SELECTED] \
       ELSE .WNDPROC :ARGS
END
```

```
? .MESSAGE 273 "HELP_HANDLER
? _
```

.WRITE

Syntax

```
.WRITE streamnumber bytearray size
(.WRITE streamnumber bytearray)
```

Explanation

`.WRITE` transfers data from a `BYTEARRAY` into a file. The first input denotes the stream number, while the second input is the `BYTEARRAY` where the data to be written is stored. The third input is the number of bytes to transfer. If the third input is missing, the `BYTEARRAY` is written completely.

The output of `.WRITE` is the number of bytes transferred. If an error occurs during the write, the output is the value `"EOF`.

If the file is open in normal mode, all line feed characters in the `BYTEARRAY` are converted to carriage return/line feed pairs. If the file is open in binary mode, however, all data is transferred unchanged.

See also `OPEN`, `.SEEK` and `.READ`.

Example

The following example opens a file for reading and fills the first 128 bytes with the ASCII equivalents of the values 0 to 127.

```
TO WRITE.128.BYTES :NAME
    MAKE "STREAM (OPEN :NAME "WB)
    MAKE "DATA BYTEARRAY 128
    FOR "I 0 127 [ASET :DATA :I :I]
    (.WRITE :STREAM :DATA)
    CLOSE :STREAM
END
```

ABS

Syntax

```
ABS number
```

Explanation

`ABS` outputs the absolute value of its input.

Examples

```
? ABS -30
Result: 30
? ABS 30
Result: 30
? ABS -3 + -4
Result: 7
? _
```

AGET

Syntax

```
AGET array number or list
```

Explanation

`AGET` outputs the value in a specific location of an array as specified by its inputs. The first element of its input is the array to be accessed, while the second element describes the array element to be obtained. For one-dimensional arrays, the second element may be a number starting from 0; for multi-dimensional arrays, the input element is a list of numbers, where each number stands for one dimension. Bytearrays always output numbers between 0 and 255, while arrays output the value stored in the accessed element. If no value is stored, bytearrays output 0 while arrays output an empty list [].

Examples

```
? MAKE "A ARRAY [2 2]
? AGET :A [1 0]
Result: []
? ASET :A [1 1] [SQUARE 4]
? AGET :A [1 1]
Result: [SQUARE 4]
? MAKE "B BYTEARRAY [2 2]
? AGET :B [1 0]
Result: 0
? ASET :B [1 1] 10
? AGET :B [1 1]
Result: 10
? _
```

AND

Syntax

```
AND object1 object2
(AND object1 object2 object3 . . .)
(AND object)
```

Explanation

AND accepts one or more inputs which must be either TRUE or FALSE. AND outputs TRUE if all of its inputs are true; otherwise, it outputs FALSE.

Examples

```
? AND "TRUE "TRUE
Result: TRUE
? AND "TRUE "FALSE
Result: FALSE
? (AND "TRUE "FALSE "TRUE)
Result: FALSE
? _
```

AND can be used in conjunction with **IF. . .THEN** statements:

```
? IF AND (3=3) (2=2) THEN PRINT "YES
YES
? _
```

APPLY

Syntax

```
APPLY procedure list
```

Explanation

`APPLY` runs the procedure given as its first input with the arguments supplied as the second input. The output of `APPLY` is the output of the procedure.

Examples

```
? APPLY "FPUT ["HELLO [BILL]]
Result: [HELLO BILL]
? _
```

ARCTAN

Syntax

```
ARCTAN number
```

Explanation

`ARCTAN` outputs the arctangent (inverse tangent) of the input.

Examples

```
? ARCTAN 1
Result: 45.00
? ARCTAN 0
Result: 0.00
? _
```

ARRAY

Syntax

```
ARRAY number or list
(ARRAY number or list list)
```

Explanation

ARRAY creates an array of the size specified by its input. If the input is a number, a one-dimensional array of the specified size is created. Its elements may be accessed by numbers between 0 and the input size less 1.

If the input to ARRAY is a list, the list describes the array. Each number in the list corresponds to the size of one dimension of the array. In order to access an element of a multi-dimensional array, a list of numbers must be supplied to the AGET and ASET primitives, where each number must be in the range from 0 to the number supplied for the corresponding dimension less 1.

A list may be supplied as an optional second element if ARRAY and all its arguments are enclosed in parentheses. The contents of this list are the initial values of the array elements.

See also ARRAYDIMS, BYTEARRAY, FILLARRAY, and LISTARRAY.

Examples

```
? MAKE "A ARRAY [2 2]
? ASET :A [1 0] [HELLO WORLD]
? AGET :A [1 0]
Result: [HELLO WORLD]
? AGET :A [0 0]
Result: []
? AGET :A [2 0]
The procedure AGET does not like 2 as an input.
? :A
Result: {ARRAY}
? _
```

ARRAY?

Syntax

```
ARRAY? word
```

Explanation

ARRAY? outputs TRUE if its input is the name of an array or bytearray. Otherwise it outputs FALSE.

See also BYTEARRAY?, LIST?, NAME?, NUMBER?N, and WORD?.

Examples

```
? MAKE "A ARRAY 5
? ARRAY? :A
Result: TRUE
? _
```

ARRAYDIMS

Syntax

```
ARRAYDIMS array
```

Explanation

ARRAYDIMS outputs a list of numbers describing the dimensions of the array named in its input. This list matches the input to the ARRAY primitive when the array was created.

See also ARRAY?.

Examples

```
? MAKE "A ARRAY [2 3 4]
? ARRAYDIMS :A
Result: [2 3 4]
? _
```

ASCII

Syntax

```
ASCII character
```

Explanation

`ASCII` outputs the American Standard Code for Information Interchange (ASCII) value of its input. If its input is a word, `ASCII` outputs the ASCII value of the first character in the word. `ASCII` outputs an integer between 0 and 255. The input must contain at least one character. The character can be a letter, number or special character.

To output the character corresponding to an ASCII code input, use `CHAR`.

Examples

```
? ASCII "A
Result: 65
? ASCII "1
Result: 49
? NOCASE
? ASCII "a
Result: 97
? ASCII "/
Result: 47
? _
```

The American Standard Code for Information Interchange (ASCII) is a standard code for representing numbers, letters and symbols. The IBM Personal Computer has many unique characters that are also represented by ASCII codes.

An ASCII file is a text file where the characters are represented in ASCII codes. PC Logo saves its files as text or ASCII files.

ASET

Syntax

```
ASET array number or list value
```

Explanation

`ASET` stores a value into a specific element of an array or bytearray. `ASET` requires three inputs: the first input is the name of the array, the second input describes the array element where the value is to be stored, and the third input is the value itself.

For one-dimensional arrays, the second input must be a number. For multi-dimensional arrays, the second input is a list of numbers, where each number stands for one dimension.

Bytearrays accept numbers between 0 and 255 as values, while arrays accept any Logo object as values.

Examples

```
? MAKE "A ARRAY [2 2]
? ASET :A [1 0] "HELLO
? AGET :A [1 0]
Result: HELLO
? MAKE "BA BYTEARRAY 5
? ASET "BA 1 255
? ASET "BA 2 "HELLO
The procedure ASET does not like HELLO as its input.
? _
```

ASK

Syntax

```
ASK number or list list
```

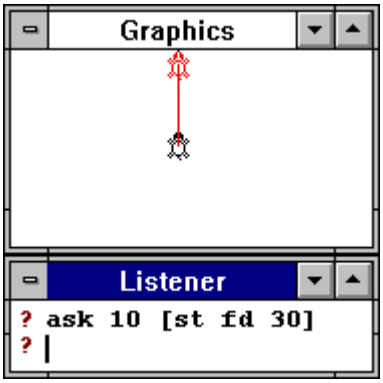
Explanation

ASK causes the turtles named in its first argument to execute the commands in its second argument. ASK makes it possible to send commands to a turtle that is not currently active without making it one of the active turtles.

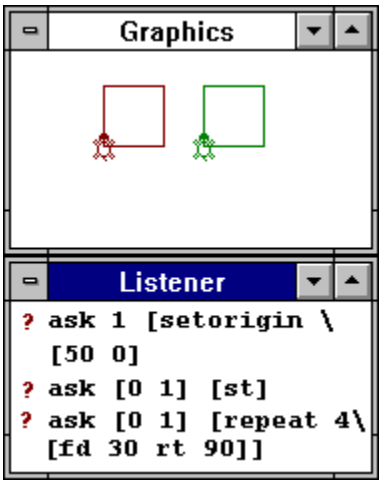
The first argument to ask can be either a single turtle number or a list of turtle numbers. To ASK a turtle it must be defined with the SETTURTLES command.

See also EACH, TELL, TURTLES, and WHO.

Examples



Turtle 10 moves forward 30 turtle steps.



Turtles 0 and 1 draw squares.

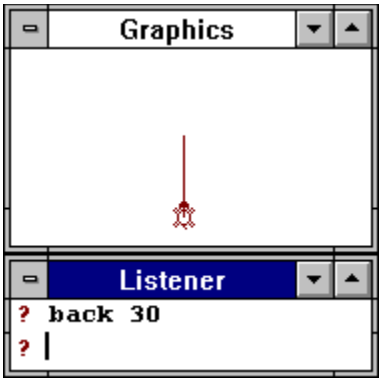
BACK (BK)

Syntax

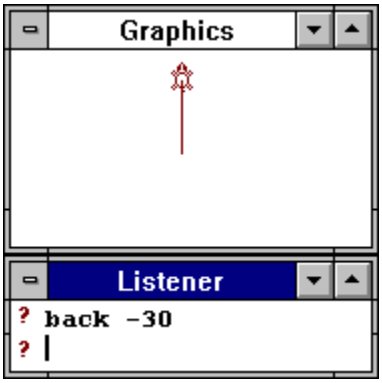
```
BACK number
BK number
```

`BACK` moves the turtle backwards the distance of its input. The heading of the turtle does not change. `BACK` moves the turtle in the opposite direction of `FORWARD`.

Examples



Turtle moves **backwards** 30 steps.



Turtle moves **forward** 30 steps.

BACKGROUND (BG)

Syntax

```
BACKGROUND
BG
```

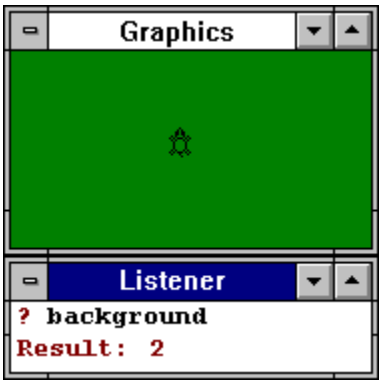
Explanation

`BACKGROUND` outputs a number which represents the current background color of the graphics screen.

0 Black	**8 Dark Grey**
1 Blue	**9 Light Blue**
2 Green	**10 Light Green**
3 Cyan	**11 Light Cyan**
4 Red	**12 Light Red**
5 Violet	**13 Magenta**
6 Brown	**14 Yellow**
7 Light Grey	15 White

To set the background color of the graphics screen, use `SETBG`.

Example



BASE

Syntax

```
MAKE "BASE number
```

Explanation

BASE is a pre-defined name which determines the base in which numbers are output by Logo. BASE requires an integer between 2 and 16 as its input.

The base in which numbers are input into Logo is separately controlled by the system name IBASE.

Examples

```
? MAKE "BASE 2
? PRINT 10
1010
? _
```

10 in base 10 is 01010 in base 2.

```
? MAKE "BASE 16
? PRINT 10
0A
? _
```

10 in base 10 is 0A in base 16.

BGPATTERN

Syntax

```
BGPATTERN
```

Explanation

`BGPATTERN` outputs a number which represents the current background pattern of the graphics screen.

To set the background pattern of the graphics screen, use `SETBGPATTERN`.

The available background patterns are:

1  **2**

 **3**

 **4**

 **5**

6  **7**

 **8**

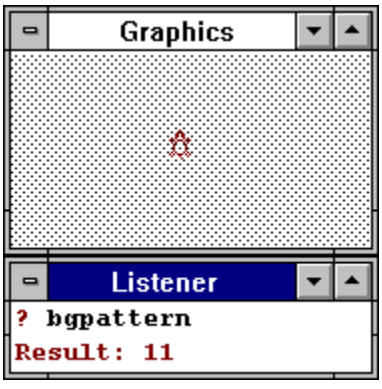
 **9**

 **10**

11 

If `SETBGPATTERN` was called with a user defined pattern (a list of eight numbers between 0 and 255), `BGPATTERN` outputs this list.

Example



BURIEDNAMES

Syntax

```
BURIEDNAMES
```

Explanation

BURIEDNAMES outputs a list of all currently buried names. Use BURYNAME to bury defined names in workspace. Use UNBURYNAME to restore buried names to the general workspace.

See also BURIEDPROCS, BURIEDPROPS, BURY, BURYALL, BURYPROC, BURYPROP, UNBURY, UNBURYALL, UNBURYPROC, and UNBURYPROP.

Examples

```
? MAKE "A 123
? MAKE "B 456
? PONS
A is 123
B is 456
? BURY [A B]
? PONS
? BURIEDNAMES
Result: [A B]
? _
```

BURIEDPROCS

Syntax

```
BURIEDPROCS
```

Explanation

BURIEDPROCS outputs a list of all currently buried procedures. Use BURYPROC to bury defined procedures in the workspace. Use UNBURYPROC to restore buried procedures to the general workspace.

See also BURIEDNAMES, BURIEDPROPS, BURY, BURYALL, BURYNAME, BURYPROP, UNBURY, UNBURYALL, UNBURYNAME, and UNBURYPROP.

Example

```
? TO SAY.HELLO
> PR "HELLO
> END
SAY.HELLO defined.
? POTS
TO SAY.HELLO
? BURY "SAY.HELLO
? POTS
? BURIEDPROCS
Result: [SAY.HELLO]
? _
```

BURIEDPROPS

Syntax

BURIEDPROPS

Explanation

BURIEDPROPS outputs a list of all currently buried variables with property lists. Use BURYPROP to bury currently defined property lists in the workspace. Use UNBURYPROP to restore buried property lists to the general workspace.

See also BURIEDNAMES, BURIEDPROCS, BURY, BURYALL, BURYNAME, BURYPROP, UNBURY, UNBURYALL, UNBURYNAME, and UNBURYPROP.

Examples

```
? PPROP "CAPITAL "NEBRASKA "LINCOLN
? PPROP "CAPITAL "KENTUCKY "FRANKFURT
? POPLS
CAPITAL is [NEBRASKA LINCOLN KENTUCKY FRANKFURT]
? BURYPROP "CAPITAL
? POPLS
? BURIEDPROPS
Result: [CAPITAL]
? _
```

BURY

Syntax

```
BURY word or list
```

Explanation

BURY makes all procedures, names, and property lists in its input invisible in the workspace. All "buried" Logo objects act as primitives. They are not included in lists of procedures, names, or property lists output by POTS, PONS, POPLS, or PRINTOUT. They cannot be ERASEd, EDITed, or SAVEd as part of the workspace. Unlike primitives, buried procedures, names, and property lists are lost when Logo is exited.

Use UNBURY to restore buried procedures, names, and property lists to the general workspace.

See also BURIEDNAMES, BURIEDPROCS, BURIEDPROPS, BURYALL, BURYNAME, BURYPROC, UNBURYALL, UNBURYNAME, UNBURYPROC, and UNBURYPROP.

Examples

```
? MAKE "A 123
? MAKE "B 456
? TO MYPROC
> PRINT [THIS IS MY PROCEDURE.]
> END
MYPROC defined
? BURY [A B MYPROC]
? PONS
? POTS
? _
```

BURYALL

Syntax

BURYALL

Explanation

BURYALL buries all user-defined procedures, names, and property lists. All "buried" Logo objects act as primitives. They are not included in lists of procedures, names, or property lists output by POTS, PONS, POPLS, or PRINTOUT. They cannot be ERASEd, EDITed, or SAVEd as part of the workspace. Unlike primitives, buried procedures, names, and property lists are lost when Logo is exited.

Use UNBURYALL to restore all buried procedures, names, and property lists to the general workspace. Use BURY and UNBURY to bury and restore specific procedures, names, and property lists.

See also BURIEDNAMES, BURIEDPROCS, BURIEDPROPS, BURYALL, BURYNAME, BURYPROC, BURYPROP, UNBURYNAME, UNBURYPROC, and UNBURYPROP.

Examples

```
? MAKE "A 123
? MAKE "B 456
? TO SAY.HELLO
> PR "HELLO
> END
SAY.HELLO defined.
? BURYALL
? PONS
? POPS
? _
```

BURYNAME

Syntax

```
BURYNAME word or list
```

Explanation

BURYNAME buries all user-defined name(s) in its input. Buried names do not appear in lists created by PONS and are not ERASEd, EDITed, or SAVEd.

Use UNBURYNAME to restore buried name(s) to the general workspace. Use BURIEDNAMES for a list of what name(s) are buried.

See also BURIEDPROCS, BURIEDPROPS, BURY, BURYALL, BURYPROC, BURYPROP, UNBURY, UNBURYNAME, UNBURYPROC, and UNBURYPROP.

Examples

```
? MAKE "A 123
? MAKE "B 456
? BURYNAME "A
? PONS
B is 456
? _
```

BURYPROC

Syntax

```
BURYPROC word or list
```

Explanation

BURYPROC buries all user-defined procedure(s) in its input. Buried procedures do not appear in lists created by POTS or POPS and are not ERASEd, EDITed, or SAVEd.

Use UNBURYPROC to restore buried procedure(s) to the general workspace. Use BURIEDPROCS for a list of what procedure(s) are buried.

See also BURIEDNAMES, BURIEDPROPS, BURY, BURYALL, BURYNAME, BURYPROP, UNBURY, UNBURYNAME, UNBURYPROC, and UNBURYPROP.

Example:

```
? TO SAY.HELLO
> PR "HELLO
> END
SAY.HELLO defined.
? BURYPROC "SAY.HELLO
? POPS
? _
```

BURYPROP

Syntax

```
BURYPROP word|list
```

Explanation

BURYPROP buries all user-defined property list(s) in its input. Buried property lists do not appear in lists created by POPLS and are not ERASEd, EDITed, or SAVEd.

Use UNBURYPROP to restore buried property list(s) to the general workspace. Use BURIEDPROPS for a list of what property list(s) are buried.

See also BURIEDNAMES, BURIEDPROCS, BURY, BURYALL, BURYNAME BURYPROC, UNBURY, UNBURYNAME, and UNBURYPROC.

Example

```
? PPROP "CAPITAL "TEXAS "AUSTIN
? PPROP "CAPITAL "GEORGIA "ATLANTA
? BURYPROP "CAPITAL
? POPLS
? _
```

BUTFIRST (BF)

Syntax

```
BUTFIRST word or list
BF word or list
```

Explanation

BUTFIRST outputs all but the first element of its input. If its input is a list of words, BUTFIRST outputs a list of words containing all but the first word. If its input is a word or number, BUTFIRST outputs all the characters of the word or number except the first character.

See also BUTLAST, FIRST, and LAST.

Examples

```
? BUTFIRST [MARY HAD A LITTLE LAMB]
Result: [HAD A LITTLE LAMB]
? BUTFIRST "WHEAT
Result: HEAT
? BUTFIRST [WHEAT]
Result: []
? BUTFIRST 2135
Result: 135
? BUTFIRST [[JANUARY FEBRUARY][MARCH APRIL] [MAY JUNE]]
Result: [[MARCH APRIL] [MAY JUNE]]
? _
```

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

```
TO SHRINKWORD :WORD
   IF EMPTY? :WORD THEN STOP
   PR :WORD
   SHRINKWORD BUTFIRST :WORD
END
```

```
? SHRINKWORD "WHEAT
WHEAT
HEAT
EAT
AT
T
? _
```

BUTLAST (BL)

Syntax

```
BUTLAST word or list
BL word or list
```

Explanation

BUTLAST outputs all but the last element of its input. If the input is a list of words, BUTLAST outputs a list containing all the words but the last word. If the input is a word or number, BUTLAST outputs all the characters of the word or number except the last character.

See also BUTFIRST, FIRST, and LAST.

Examples

```
? BUTLAST [MARY HAD A LITTLE LAMB]
Result: [MARY HAD A LITTLE]
? BUTLAST "OSTRICH
Result: OSTRIC
? BUTLAST 3265
Result: 326
? BUTLAST [[SNAKES SLITHER][RABBITS HOP] [MICE SCURRY]]
Result:[[SNAKES SLITHER] [RABBITS HOP]]
? _
```

The procedure below makes a plural word or list of words into singular form.

```
TO SINGULAR :WORD
   IF EMPTY? :WORD THEN STOP
   PRINT BUTLAST LAST :WORD
   SINGULAR BUTLAST :WORD
END
```

```
? SINGULAR [CATS]
CAT
? SINGULAR [BOOKS TOOLS FLOWERS EYES RUNS]
RUN
EYE
FLOWER
TOOL
BOOK
? _
```

BUTMEMBER (BM)

Syntax

```
BUTMEMBER word1 or list1 word2 or list2
BM word1 or list1 word2 or list2
```

Explanation

BUTMEMBER outputs a word or list consisting of its second input with all occurences of its first input removed. It the second input is a word, the first input must also be a word. If the second input is a list, the first input can be either a word or list.

See also BUTFIRST and BUTLAST.

Examples

```
? BUTMEMBER "AM [HI I AM FRED]
Result: [HI I FRED]
? BM "D "ABCDABCDABCD
Result: ABCABCABC
? BUTMEMBER 22 [11 22 33 44 55]
Result: [11 33 44 55]
? BUTMEMBER [JANUARY 1] [[JANUARY 1][JULY 4][DECEMBER 25]]
Result: [[JULY 4][DECEMBER 25]]
? _
```

BUTTON?

Syntax

```
BUTTON? 1, 2 or 3
```

Explanation

BUTTON? outputs the state of the left or right mouse button. The number given as its input is 1 for the left button, 2 for the right button or 3 for the middle button. The output is TRUE when the button is depressed, or FALSE when the button is not depressed or when no mouse is found.

See also MOUSE and .MOUSEON.

Example

```
TO CHECK.MOUSE
   IF BUTTON? 1 THEN PRINT [LEFT BUTTON WAS PRESSED.]
   IF BUTTON? 2 THEN PRINT [RIGHT BUTTON WAS PRESSED.]
   CHECK.MOUSE
END
```

Use Control G to stop this procedure.

BYE

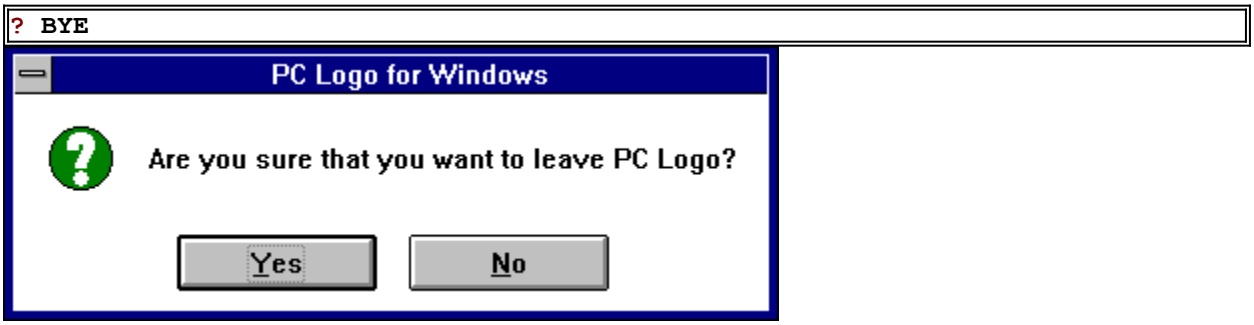
Syntax

```
BYE
QUIT
EXIT
```

Explanation

`BYE` clears Logo and the Logo workspace from the computer's memory and returns to the Program Manager.

Example



BYTEARRAY

Syntax

```
BYTEARRAY number or list
(BYTEARRAY number or list list)
```

Explanation

BYTEARRAY creates a byte array which is a special array in which numbers between 0 and 255 and be stored. If BYTEARRAY has a number as an input, a one-dimensional array of the specified size is created. The elements of the array may be accessed by numbers between 0 and the specified size less 1. Each element of a onedimensional byte array corresponds to one byte in memory; therefore, a byte array may also be used to transfer data to Windows API calls. The TEXTARRAY command can be used to convert the contents of a onedimensional byte array to a Logo word.

If BYTEARRAY has a list of numbers as input, each number in the list corresponds to the size of one dimension of the array. To access an element of a multi-dimensional byte array, a list of numbers must be supplied to the AGET and ASET primitives with each number in the range from 0 to the number supplied for the corresponding dimension less 1.

If a list is supplied as an optional second input to BYTEARRAY, the contents of this list are used to initialize the array elements.

See also ARRAY and BYTEARRAY?.

Example

```
? MAKE "A BYTEARRAY [2 2]
? ASET :A [1 0] [HELLO WORLD]
The procedure ASET does not like [HELLO WORLD] as an input.
? ASET :A [1 0] 255
? AGET :A [1 0]
Result: 255
? AGET :A [2 0]
The procedure AGET does not like 2 as an input.
? :A
Result: {ARRAY}
? _
```

BYTEARRAY?

Syntax

```
BYTEARRAY? word
```

Explanation

`BYTEARRAY?` outputs `TRUE` if its input is the name of a bytearray. Otherwise it outputs `FALSE`.

See also `ARRAY?`, `LIST?`, `NAME?`, `NUMBER?`, and `WORD?`.

Example

```
? MAKE "A BYTEARRAY 5
? ARRAY? :A
Result: FALSE
? BYTEARRAY? :A
Result: TRUE
? _
```

CASE

Syntax

CASE

Explanation

CASE causes Logo to convert all alphabetic input to upper case. CASE is an abbreviation for CASE CONVERSION. What you type in lower case characters is automatically converted to upper case. In CASE, all Logo output displays in upper case.

When Logo loads, its default state is CASE.

To make Logo read input and display output in both upper and lower case, use NOCASE. Use CASE? to see which mode Logo is in.

Example

```
? PRINT [THREE BLIND MICE]
THREE BLIND MICE
? print [Three Blind Mice]
THREE BLIND MICE
? NOCASE
? print [Three Blind Mice]
print is not a Logo procedure.
? PRINT [Three Blind Mice]
Three blind mice
? _
```

CASE?

Syntax

CASE?

Explanation

CASE? outputs CASE or NOCASE to reflect the current case state of Logo.

Example

```
? CASE
? CASE?
Result: CASE
? NOCASE
? CASE?
Result: NOCASE
? _
```

CATCH

Syntax

```
CATCH word instructionlist
```

Explanation

CATCH runs the instructions in its second input. If Logo encounters a THROW statement with the same word argument, control returns to CATCH.

There are two special uses of CATCH. If the first input to CATCH is TRUE, any THROW statement will be caught. Also, CATCH "ERROR catches an error that would otherwise print a Logo message and return to toplevel. In this case, the built-in variable ERROR contains a hint about which error occured.

Examples

The following example asks you to type a name. If you type a number instead, the program prints a message and continues.

```
TO NAMIT
   CATCH "NOTNAME [NAMIT1 STOP]
   NAMIT
END

TO NAMIT1
   PRINT [PLEASE TYPE A NAME]
   MAKE "NAME READ
   IF NUMBER? :NAME \
      [PRINT [THAT'S A NUMBER, NOT A NAME] THROW "NOTNAME]
   PRINT (SE :NAME [IS A GOOD NAME])
END
```

```
? NAMIT
PLEASE TYPE A NAME
? KURT
KURT IS A GOOD NAME
PLEASE TYPE A NAME
? 5
THAT'S A NUMBER NOT A NAME
? _
```

Type Control-G to return to toplevel.

In the following example, AVOID.INTERRUPTIONS runs the commands you type. If an error occurs, Logo prints

```
THAT'S NOT A LOGO COMMAND
```

and continues executing the procedure instead of printing the usual Logo message and terminating the procedure by returning to toplevel.

```
TO AVOID.INTERRUPTIONS
   CATCH "ERROR [AVOID.INTERRUPTIONS1]
   PRINT [THAT'S NOT A LOGO COMMAND]
```

```
   AVOID.INTERRUPTIONS
END

TO AVOID.INTERRUPTIONS1
   RUN READLIST
   AVOID.INTERRUPTIONS1
END
```

```
? AVOID.INTERRUPTIONS
? PRINT [THIS IS RIGHT]
THIS IS RIGHT
? PRINT THIS IS RIGHT
THAT'S NOT A LOGO COMMAND
? _
```

Type `TOPLEVEL` or Control-G to return to toplevel.

CHAR

Syntax

```
CHAR number
```

Explanation

`CHAR` outputs the character whose ASCII code is the input. The input number can be from 0 through 255.

To output the ASCII code corresponding to an input character, use `ASCII`.

Examples

```
? CHAR 65
Result: A
? CHAR 47
Result: /
? CHAR 49
Result: 1
? CHAR 97
Result: a
? _
```

CLEAN

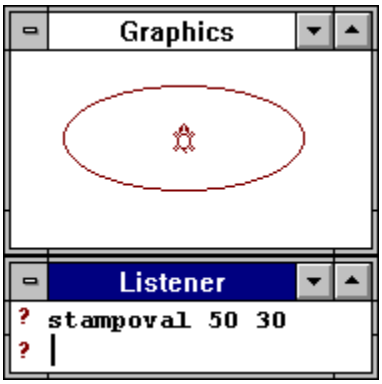
Syntax

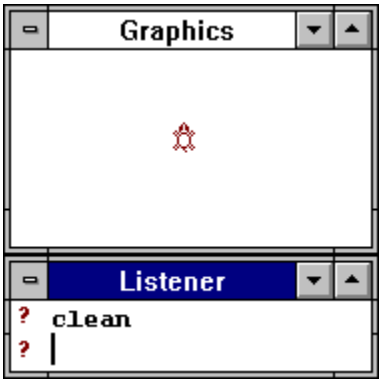
```
CLEAN
```

Explanation

`CLEAN` erases the graphics screen but does not affect the heading or position of the turtle. See also `CLEARSCREEN` and `DRAW`.

Example





CLEARINPUT

Syntax

`CLEARINPUT`

Explanation

`CLEARINPUT` clears all input from the current input stream. For input stream 0 (keyboard), `CLEARINPUT` discards any keys the user types.

Example

```
? PRINT [WAITING...] WAIT 200 PR KEY?
WAITING...(User types J.) TRUE
? J
? PR[WAITING...] WAIT 200 CLEARINPUT PR KEY?
WAITING...(User types a key.) FALSE
? _
```

CLEARSCREEN (CS)

Syntax

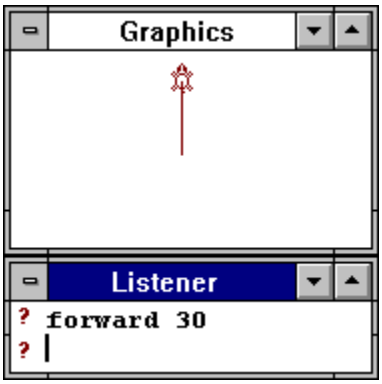
```
CLEARSCREEN
CS
```

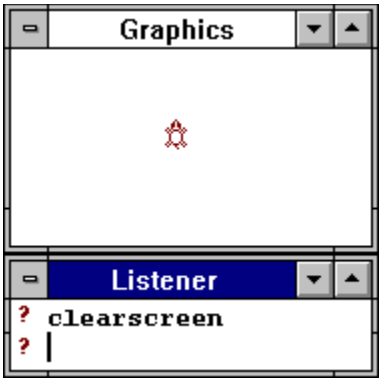
Explanation

`CLEARSCREEN` erases the graphics screen, returns the turtle to the center of the screen, and sets the turtle's heading to 0. `CLEARSCREEN` does not affect the pen state or screen colors.

See also `CLEAN` and `DRAW`.

Example





CLEARTEXT (CT)

Syntax

CLEARTEXT

Explanation

CLEARTEXT clears all text and places the cursor in the upper left corner of the Listener window.

Example

```
? REPEAT 5 [PR [GOOD EVENING]
GOOD EVENING
GOOD EVENING
GOOD EVENING
GOOD EVENING
GOOD EVENING
? CLEARTEXT
? _
```

CLOSE

Syntax

```
CLOSE streamnumber
(CLOSE)
```

Explanation

CLOSE concludes any pending operations with the stream number specified and releases the stream for reuse. If the stream number has not been opened, an error will occur.

CLOSE is necessary for all file output operations, since data is automatically buffered in memory and not sent to the disk drive until a large quantity is ready to go. If an output stream is not closed, you may lose data.

If CLOSE is used without any inputs, all open data streams are closed and I/O will reverts to the console and keyboard. Only the copy stream opened with the COPYON command remains open.

See also OPEN and CREATE.

Example

The following procedure creates a disk file and prints an address into it.

```
TO ADDRESS.LIST
   MAKE "STANDARD.OUTPUT CREATE "ADDRESS
   PRINT [JOHN Q. PUBLIC]
   PRINT [ANYTOWN, U.S.A.]
   CLOSE :STANDARD.OUTPUT
   MAKE "STANDARD.OUTPUT 0
END
```

To view the file, enter the editor and load the file **ADDRESS**.

COLOR

Syntax

```
COLOR number
```

Explanation

COLOR returns the color setting for the given color number in the current palette. The color setting is a list of three values between 0 and 255. The first value stands for Red, the second value for Green and the third value for Blue.

Windows will try to support every color request made by any program. In 16 color modes, however, all available colors are used up by Windows itself for its array of reserved colors. Therefore, any additional color will be generated as a dithered color. A dithered color is a color made up by painting small patterns of pixels put together by the available colors on the display. Since a dithered color is not a true color, Windows us unable to use these colors for drawing lines. If a request is made to draw a line in a dithered color, the true color that most closely matches the desired color will be used.

On displays with 256 colors and more, Windows has many more chances to match a color request to a true color. If the requested color is unavailable in the current color set used by the display, an existing but unused entry will be altered to reflect the new true color. If all entries are used up, one of the existing entries will be replaced with the requested color, thus slightly altering the color stored there before.

Logo has a total of four palettes available, where each palette contains 256 colors. In 16-color modes, the 16 available colors are repeated all over each palette until each palette is filled. If you want to define a color which does not belong to the set of colors defined, Logo wil lmatch your requested color to the nearest color available, since Logo does not want to disturb Windows' own coloring. When Windows makes 256 colors avilable, Logo will always try to satisfy your color request as good as possible. The color palette will be initialized to the palette Windows generates for 256-color VGA displays.

To change a color, use SETCOLOR. See also PENCOLOR and SETPC.

Examples

The following example assumes that you use color 4 (red) for Logo output, and how to assign different colors to the text that Logo prints.

```
? COLOR 4
Result: [128 0 0]
? SETCOLOR 4 [0 255 0]
Result: [168 0 0]
? _
```

CONST

Syntax

```
CONST name expansion
```

Explanation

The CONST command defines an abbreviation for a certain Logo name. Its first input is the abbreviation, while the second input is the text string which is to replace the name every time it is encountered. If the second input is a list, its elements are formed into a text string much like the TYPE command does. Every time the name is typed, it will be replaced by its text string. Remember that this replacement is a textual replacement; it takes place before the text enters Logo at all.

When finding a constant name within a constant text, Logo will attempt to replace this name as well. This could lead to a deadlock situation when you define a constant which contains its own name like CONST "HELLO "HELLO. Therefore, Logo limits these replacements to a level of 16.

The CONST command is especially useful for defining replacements for otherwise meaningless numbers. Also, often used commands may replaced by an abbreviation.

See also POC.

Example

```
? CONST "NAME "MICHAEL
? CONST "HELLO [OH HI, NAME]
? POC
NAME is MICHAEL
HELLO is OH HI, NAME
? PR [HELLO]
OH HI, MICHAEL
? _
```

CONTENTS

Syntax

CONTENTS

Explanation

CONTENTS outputs a list of all objects currently in the Logo system that occupy name space: names, variables, procedure names, words used in procedures, and all other text either entered or loaded with Logo.

Example

```
? PR ITEM 30 CONTENTS
BASE
? _
```

CONTINUE (CO)

Syntax

```
CONTINUE
(CONTINUE number)
CO
(CO number)
```

Explanation

`CONTINUE` resumes a procedure that has been temporarily halted with `PAUSE` or Control-Z. For debugging purposes, you can supply an optional additional number. This number tells Logo in single step mode how many step to execute before pausing again.

Example

```
TO PRINTSIT
   REPEAT 3 [PR [I WANT]]
   PAUSE
   REPEAT 5 [PR "MONEY]
END
```

```
? PRINTSIT
I WANT
I WANT
I WANT
PAUSE> CONTINUE
MONEY
MONEY
MONEY
MONEY
MONEY
? _
```

COPYDEF

Syntax

```
COPYDEF name name
```

Explanation

COPYDEF copies the definition of its second input to its first input. The second input to COPYDEF must be a name of either a procedure or a primitive.

If you use COPYDEF to redefine a primitive with a new meaning, you will lose the old primitive. For example: COPYDEF "BK "FD redefines BK to move the turtle forward. You must restart Logo to regain the original definition of BK.

Examples

```
? TO MYNAME
>    PRINT [ORLANDO]
> END
MYNAME defined.
? COPYDEF "YRNAME "MYNAME
? PO MYNAME YRNAME
TO MYNAME
   PRINT [ORLANDO]
END
TO YRNAME
   PRINT [ORLANDO]
END
? MYNAME
ORLANDO
? YRNAME
ORLANDO
? _
```

COPYOFF

Syntax

```
COPYOFF
```

Explanation

`COPYOFF` turns off the copy, or "dribble," stream, if one has been turned on with `COPYON`.

Example

```
? COPYON "TRYOUT.LGO
? PR [THIS SESSION IS BEING RECORDED]
THIS SESSION IS BEING RECORDED
? COPYOFF
? _
```

Now, the file TRYOUT.LGO on disk contains the following:

```
? PR [THIS SESSION IS BEING RECORDED]
THIS SESSION IS BEING RECORDED
? COPYOFF
```

COPYON

Syntax

```
COPYON filename.ext
```

Explanation

COPYON copies, or "dribbles," all text interaction between user and computer to the filename indicated by its input. COPYON copies information until COPYOFF is entered.

When a file created with COPYON is closed, it may be manipulated like any other Logo file: it can be loaded, viewed, edited, and resaved.

Example

```
? COPYON "TRYOUT.LGO
? PR [THIS SESSION IS BEING RECORDED]
THIS SESSION IS BEING RECORDED
? COPYOFF
? _
```

Now, the file TRYOUT.LGO on disk contains the following:

```
? PR [THIS SESSION IS BEING RECORDED]
THIS SESSION IS BEING RECORDED
? COPYOFF
```

COS

Syntax

```
COS number
```

Explanation

COS outputs the cosine of its input, a number of degrees. Remember that COS x = adjacent / hypotenuse.

See also ARCTAN and SIN.

Examples

```
? COS 0
Result: 1
? COS 90
Result: 0
? COS 270
Result: 0
? _
```

The following procedure defines the tangent function:

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

COUNT

Syntax

```
COUNT word or list
```

Explanation

COUNT outputs the number of elements in its input. If its input is a word, COUNT outputs the number of characters. If the input is a list, COUNT outputs the number of elements in the list.

Examples

```
? COUNT "ELEMENTARY
Result: 10
? COUNT [ELEMENTARY]
Result: 1
? COUNT [[MT. WASHINGTON] [MT. RAINIER] [MAUNA LOA]]
Result: 3
? _
```

CREATE

Syntax

```
CREATE filename
```

Explanation

CREATE deletes the DOS file with the name specified by its input. A new, empty file is prepared for output and its assigned Logo stream number is output. Data may then be written to the stream using the PRINT, TYPE and other Logo stream output primitives by making STANDARD.OUTPUT the stream number.

If a DOS device name is specified, CREATE functions as OPEN.

Examples

The following procedure prints titles of user-defined procedures in workspace into a disk file.

```
TO POTS.FILE
   MAKE "STANDARD.OUTPUT CREATE "PROCS
   POTS
   CLOSE :STANDARD.OUTPUT
   MAKE "STANDARD.OUTPUT 0
END
```

CURDIR

Syntax

```
CURDIR
```

Explanation

`CURDIR` outputs the current directory.

See also `SETCURDIR`, `DISK` and `SETDISK`.

Examples

The following two procedures switch to a new directory and switch back to the old directory.

```
TO CHDIR :NAME
    MAKE "OLD.DIR CURDIR
    SETCURDIR :NAME
END

TO CHBACK
    SETCURDIR :OLD.DIR
END
```

DATE

Syntax

```
DATE
```

Explanation

`DATE` outputs the current date as a three-element list in the form [day month year].

Example

```
? DATE
Result: [30 3 1993]
? _
```

DEFEVENT

Syntax

```
DEFEVENT eventname eventprocedure
```

Explanation

The DEFEVENT command attaches a Logo procedure to a specific event. Whenever this event occurs, the attached Logo procedure will be called. Its first input is a name which is the name of the event, while the second input is the name of the procedure which is to be attached to that event.

Currently, there are two events defined which you can overload. The BREAK event occurs every time the Control-G key is being pressed. Normally, a built-in event procedure will turn off the timer ticks (if a timer has been activated with the TIMER command) and return you to toplevel. You can modify this behaviour by defining an event procedure for the BREAK event. This procedure should, however, provide the option of returning to toplevel. If you omit this option, pressing Control-G will not return you to toplevel or even break any existing procedures. Also, you might want to turn off the timer during the event processing. The TIMER event is activated with the TIMER command. This command starts a timer which, when elapsed, generates a TIMER event which can be used to activate a Logo procedure. Both the BREAK and the TIMER events do not generate any inputs for the event handling procedures.

See also EVENT and TIMER.

Example

The following procedure is a replacement for the built-in BREAK handler. It will ask for confirmation of the break request before returning to toplevel.

```
TO MY.OWN.BREAK
   LOCAL "ANSWER
   PR "BREAK!
   TYPE [RETURN TO TOPLEVEL? |(Y/N)|]
   MAKE "ANSWER RC
   (PR)
   IF :ANSWER = "Y THEN \
      IGNORE TIMER "FALSE \
      (HALT) \
      TOPLEVEL
END
```

```
? DEFEVENT "BREAK "MY.OWN.BREAK
? _
```

DEFINE

Syntax

```
DEFINE name instructionlist
```

Explanation

`DEFINE` names a new procedure with the name of its first input. The second input to `DEFINE` determines the definition of the procedure.

Variable(s) in the title line must be the first element(s) of the list of instructions, with no dots (:) before their name(s). If there are no variables, the first element must be the empty list.

Each remaining element in the list of instructions is a list which consists of one line of the procedure definition. The list of instructions is written in the same form as the output of `TEXT`.

`END` must not be included in the list of instructions, as it is not part of the definition.

Examples

```
? DEFINE "SQUARE [[] [REPEAT 4 [FD 100 RT 90]]]
? PO SQUARE
TO SQUARE
   REPEAT 4 [FD 40 RT 90]
END
? DEFINE "HEX [[LENGTH] [REPEAT 6 [FD :LENGTH RT 60]]]
? PO HEX
TO HEX :LENGTH
   REPEAT 6 [FD :LENGTH RT 60]
END
? _
```

DEFINED?

Syntax

```
DEFINED? name
```

Explanation

`DEFINED?` outputs `TRUE` if the input is a name of a primitive procedure or a user-defined procedure; otherwise; it outputs `FALSE`.

Examples

```
? DEFINED? "FORWARD
Result: TRUE
? DEFINED? "BLIMP
Result: FALSE
? TO MYPROC
> PRINT [THIS IS MY PROCEDURE.]
> END
MYPROC defined
? IF DEFINED? "MYPROC THEN MYPROC
THIS IS MY PROCEDURE.
? _
```

DELETE

Syntax

```
DELETE filename.ext
```

Explanation

`DELETE` removes the file specified by its input from the disk. If the file is successfully deleted, `DELETE` outputs `TRUE`; otherwise, it outputs `FALSE`.

To remove a file from a disk in the drive which is not selected, precede the file name with the drive specifier, backslash (\), and a colon.

Examples

```
? DELETE "KNIFE.LGO
Result: TRUE
? MEMBER? "KNIFE.LGO DIR
Result: FALSE
? DISK
Result: C
? DELETE "B\:SWORD.LGO
Result: TRUE
? MEMBER? "SWORD.LGO (DIR B\:)
Result: FALSE
? _
```

```
DELETE "????.LGO
```

erases all files with four-character names with the file extension .LGO and outputs `TRUE` if the files are successfully erased.

```
DELETE "\*.LGO
```

erases all files with names of any length with the file extension .LGO and outputs `TRUE` if they are successfully erased.

DELIMITER

Syntax

```
MAKE "DELIMITER character
```

Explanation

The pre-defined variable `DELIMITER` contains the character used to delimit characters which otherwise would have a special meaning in Logo. This delimiter character is predefined to the vertical bar "|". To use the `DELIMITER`, place the `DELIMITER` character before and after the string of characters to be interpreted literally. Delimited characters are treated as written, no matter what `CASE` state Logo is in.

Examples

```
? :DELIMITER
Result: |
? PR "|**** Give me an answer: |
**** Give me an answer:
? MAKE "DELIMITER "\*
? PR "*Hello, Bill*
Hello, Bill
? _
```

DIRECTORY (DIR)

Syntax

```
DIRECTORY
DIR
(DIRECTORY word)
(DIR word)
```

Explanation

DIRECTORY outputs a list of file names on the disk in the currently selected disk drive.

If DIRECTORY is used with an input, it outputs the file names specified by its input. A drive specifier may be used to access a disk drive which is not currently selected. A ? may be used to match a single character except a period and a * may be used to match a group of characters not including a period.

See also SUBDIR.

Examples:

```
? DIRECTORY
Result: [TEST.LGO TEXT.LGO JUNK.LGO LOGO.EXE LOGO.HLP]
? (DIRECTORY "\*.LGO)
Result: [TEST.LGO TEXT.LGO JUNK.LGO]
? DISK
Result: C
? (DIRECTORY "B\:)
Result: [HOUSE.LGO TROT.LGO HEAD.LGO]
? _
```

DISK

Syntax

`DISK`

Explanation

`DISK` outputs the name of the default disk drive. When Logo starts up, the default drive is the current DOS default. All disk operations are performed on this drive unless otherwise specified with `SETDISK`.

Examples

```
? DISK
Result: C
? SETDISK "B
? DISK
Result: B
? SETDISK "X
The procedure SETDISK does not like Y as input.
? _
```

DOT

Syntax

```
DOT [xcoordinate ycoordinate]
```

Explanation

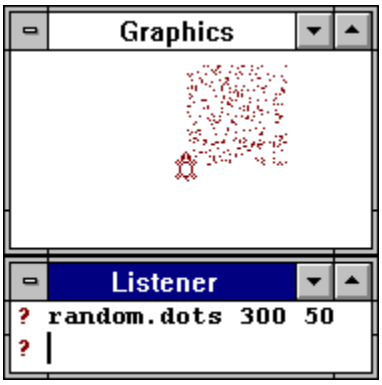
DOT prints a dot at the point determined by its inputs. The color of the dot is printed in the current pen color.

DOT requires a list of two numbers as its input. Since DOT does not evaluate the contents of its input list, the list must contain two numbers. Use SENTENCE to compose the list as an argument for DOT if using variables for the X and Y coordinates.

See also SETXY and GETXY.

Examples

```
TO RANDOM.DOTS :COUNT :AREA
  REPEAT :COUNT [DOT SE RANDOM :AREA RANDOM :AREA]
END
```



This procedure call causes 300 dots to appear randomly in the quadrant defined from 0 to 50 turtle steps in both the X and Y directions.

DOTCOLOR

Syntax

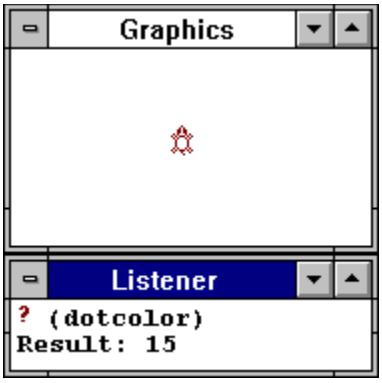
```
DOTCOLOR [xcoordinate ycoordinate]
(DOTCOLOR)
```

Explanation

`DOTCOLOR` returns the color of the pixel identified by its argument. The color is identified by a number from 0 to 15 or 255 corresponding to the `BACKGROUND` or `PENCOLOR`.

`(DOTCOLOR)` with no arguments enclosed in parentheses returns the color of the pixel under the turtle.

Examples



DRAW

Syntax

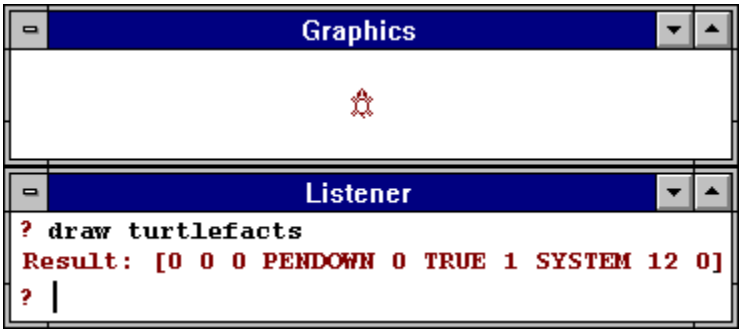
```
DRAW
```

Explanation

`DRAW` prepares the graphics screen for drawing by doing the following:

- Clears the screen.
- Homes the turtle.
- Shows the turtle.
- Puts the pen down.
- Sets the pen color to color 0.
- Sets the background color to color 15.
- Sets the turtle's pen width to 1.
- Resets the turtle font to the system font
- Sets to `WRAP` mode.

Example



EACH

Syntax

```
EACH list
```

Explanation

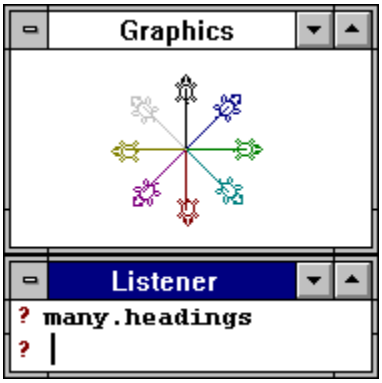
EACH causes each of the currently active turtles to execute the commands contained in its argument sequentially. This allows each of several turtles to be given a variable input or to be addressed by WHO or its current number.

See also ASK, SETTURTLES, TELL, and WHO.

Example

```
TO MANY.HEADINGS
  TELL [0 1 2 3 4 5 6 7]
  PENDOWN ST
  EACH [SETH 45 * WHO SETPC WHO FD 30]
END
```

This procedure causes eight turtles to change their color to the same value as their turtle number. They then move apart in different directions.



EDIT

Syntax

```
EDIT
EDIT name
EDIT name1 name2 name3 . . .
EDIT ALL
EDIT NAMES
EDIT PROCEDURES
EDIT CONSTANTS
```

Explanation

`EDIT` enters the Logo editor and opens an edit window.

If no input is specified with `EDIT`, the last edit window which was used will be reopened. If a name or names are specified with `EDIT`, the contents of the edit window are the definitions of the specified procedures. If the procedures have not yet been defined, the editor contains the line `TO name` and `END`.

`EDIT ALL` makes the contents of the editor all the procedures, names, and property lists that exist in the workspace. `EDIT NAMES` makes the contents of the editor all the user-defined names in the workspace (names defined with `MAKE`). `EDN` is equivalent to `EDIT NAMES`. `EDIT PROCEDURES` makes the contents of the editor all the user- defined procedures in the workspace. `EDIT CONSTANTS` edits all the currently defined constants.

`EDIT` belongs to the commands which do not evaluate their input. You can supply the inputs without quoting them. If you, however, wannt an input to be evaluated, you can surround it with brackets.

Example

EDN

Syntax

```
EDN
EDN name
EDN name1 name2 name3 . . .
```

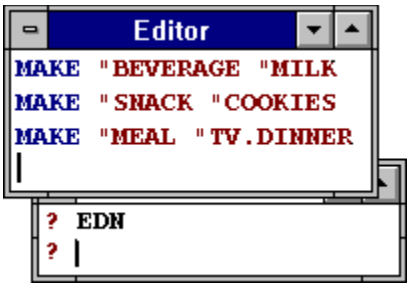
Explanation

`EDN` enters the editor and makes its contents the variable name(s) or names specified by its input. `EDN` will accept any number of inputs. `EDN` without inputs puts all user-defined names in workspace in the editor. `EDN` without inputs is equivalent to `EDIT NAMES`.

`EDN` belongs to the commands which do not evaluate their input. You can supply the inputs without quoting them. If you, however, wannt an input to be evaluated, you can surround it with brackets.

Examples

```
? MAKE "BEVERAGE "MILK
? MAKE "SNACK "COOKIES
? MAKE "MEAL "TV.DINNER
? EDN
```



ELSE

Syntax

```
IF statement THEN instructionlist ELSE instructionlist
```

Explanation

ELSE runs its input if the conditional statement which precedes it is FALSE. ELSE is used in conjunction with IF. See also IF and THEN.

Examples

The following procedure lets you guess the animal you supply as input.

```
TO GUESS :ANIMAL
PR [WHAT ANIMAL AM I THINKING OF?]
MAKE "CHOICE FIRST READLIST
IF :ANIMAL = :CHOICE THEN PR[CORRECT ANSWER] GUESS :ANIMAL
END
```

```
? GUESS "CAT
WHAT ANIMAL AM I THINKING OF?
? DOG
TRY AGAIN
? CAT
CORRECT ANSWER
? _
```

EMPTY?

Syntax

```
EMPTY? word or list
```

Explanation

`EMPTY?` outputs `TRUE` if the input is the empty word (") or the empty list ([]); otherwise, it outputs `FALSE`.

Examples

```
? EMPTY? [3 26 56]
Result: FALSE
? MAKE "FRUIT "
? EMPTY? :FRUIT
Result: TRUE
? MAKE "FRUIT [PAPAYA]
? EMPTY? :FRUIT
Result: FALSE
? EMPTY? BUTFIRST :FRUIT
Result: TRUE
? _
```

END

Syntax

```
END
```

Explanation

`END` terminates a procedure definition. The word `END` should be the last line of a procedure and should appear on a line by itself.

Example

```
? TO SQUARE
> REPEAT 4 [FD 50 RT 90]
> END
SQUARE is defined.
? _
```

EQUAL?

Syntax

```
EQUAL? word1 or list1 word2 or list2
```

Explanation

`EQUAL?` outputs `TRUE` if its first two inputs are equal numbers or identical words or lists; otherwise, it outputs `FALSE`. Equivalent to `=`.

Examples

```
? EQUAL? 6 6
Result: TRUE
? EQUAL? [6] [6]
Result: TRUE
? EQUAL? 6 [6]
Result: FALSE
? EQUAL? "BLUE "BLUE
Result: TRUE
? _
```

ERASE (ER)

Syntax

```
ERASE ALL
ERASE procname
ERASE procname1 procname2 procname3 . . .
ERASE NAMES
ERASE PROCEDURES
ERASE PROPERTIES
ERASE CONSTANTS
```

Explanation

ERASE removes the definition of its input from the workspace. The input to ERASE must be a procedure name or names. To remove individual variables from the workspace, use ERN.

ERASE NAMES removes all variables from the workspace. ERASE PROCEDURES removes all procedures from the workspace. ERASE PROPERTIES removes all property lists from the workspace. ERASE CONSTANTS removes all constants from the workspace. ERASE ALL removes all procedures and variables from the workspace.

Examples

```
? ERASE SQUARE
? ERASE SQUARE CIRCLE TRIANGLE
? ERASE ALL
? _
```

ERC

Syntax

ERC

Explanation

The ERC command erases all defined constants. You cannot erase individual constants, since every constant name you would enter on the command line would immediately be replaced by its text string.

This command is an abbreviation for the command ERASE CONSTANTS.

Example

```
? CONST "NAME "MICHAEL
? CONST "HELLO [OH HI, NAME]
? POC
NAME is MICHAEL
HELLO is OH HI, NAME
? ERC
? POC
? _
```

ERN

Syntax

```
ERN
ERN name
ERN name1 name2 name3 . . .
```

Explanation

ERN removes the variable(s) specified by its input from the Logo workspace.

ERN used without any inputs removes all variables from the workspace. See also ERASE, BURIEDNAMES, BURYNAME, and UNBURYNAME.

ERASE belongs to the commands which do not evaluate their input. You can supply the inputs without quoting them. If you, however, want an input to be evaluated, you can surround it with brackets.

Examples

```
? PONS
COLOR is BLUE
NUMBER is 2
JOHN is JANE
? ERN COLOR
? ERN NUMBER (FIRST [JOHN SMITH])
? PONS
? _
```

ERROR

Syntax

```
CATCH "ERROR
```

Explanation

ERROR is the error object used by error handling mechanism. See also CATCH and THROW.

If you catch an error, the system variable ERROR contains a short word describing the error which occured:

"NODE	Out of list space.
"ATOM	Out of atom space.
"MEM	Out of memory.
"STCK	Too many recursive procedure calls.
"AFN	Ambiguous filename not allowed.
"FILE	File xxxx not found.
"IO	Input/output (I/O) error.
"OPEN	File stream xxxx not open.
"STRU	No more file structures for OPEN or CREATE.
"USED	xxxx is already in use. Try a different name.
"HASH	Bad number syntax.
"OUT	You don't say what to do with the output of xxxx.
"INP	The procedure xxxx does not like yyyy as input.
"VAL	xxxx is not a valid input for yyyy.
"PROC	xxxx is not a Logo procedure.
"NAME	xxxx is not a Logo name.
"MORE	xxxx needs more input(s).
"CTCH	Can't find catch for xxxx
"BUFF	Unable to create buffer for picture.
"MATH	Math overflow.
"DIV0	Division by zero.
"INT	Internal error: xxxx
"MOVE	Attempt to move turtle nnnn outside the fence.
"WIN	Turtle(s) nnnn must be inside window.
"DOT	Attempt to draw a dot outside the screen.
"PCX	Unable to load picture xxxx
"TELL	Cannot TELL turtle xxxx; there are only nnnn turtles...
"PRN	No printer defined.
"RDY	Printer not ready.
"CLIP	Clipboard error.
"TICK	Cannot create timer.
"VER	Service not available.
"MCI	MCI command interface error.

Note that the variable ERROR always contains the last error which occured, unless you explicitly assign a different value to it.

Example

```
? HELLO
HELLO is not a Logo procedure.
? CATCH "ERROR [HELLO]
? :ERROR
Result: PROC
? _
```

EVAL

Syntax

```
EVAL list
```

Explanation

EVAL evaluates its input like the RUN command. Unlike RUN, however, the outputs of each command are collected into a list. This command is very handy when variables in a list have to be replaced with their values.

Example:

```
? MAKE "X 100 MAKE "Y 50
? GETXY
Result: [0 0]
? SETXY EVAL [:X :Y]
? GETXY
Result: [100 50]
? _
```

EVENT

Syntax

```
EVENT eventname
(EVENT eventname parameter)
```

Explanation

The `EVENT` command invokes an event. Its input is the name of the event to be invoked. If the command is enclosed in parentheses, an optional second input is forwarded to the event handling procedure.

This command may be used to invoke an event handling procedure tied to the given event.

Example

This command is the equivalent of pressing `Control-G`.

```
EVENT "BREAK
```

EXPN

Syntax

```
EXPN number
```

Explanation

`EXPN` calculates the natural base e (2.7183. . .) raised to the power specified by its input.

Examples

```
? EXPN 3
Result: 20.09
? EXPN 0
Result: 1
? EXPN 10
Result: 22026.46
? EXPN -1
Result: 0.37
? _
```

The following procedure outputs the number B raised to the power of E.

```
TO POWER :B :E
   OP EXPN (:E * LOG :B)
END
```

```
? POWER 2 8
Result: 256
? _
```

EXTENT

Syntax

EXTENT

Explanation

EXTENT outputs the current X and Y extents of the graphics window as a list. If one turtle step corresponds to one screen pixel, the extents are both Zero.

For a detailed explanation of the window extent, see SETEXTENT.

Example

```
? EXTENT
Result: [0 0]
? _
```

FENCE

Syntax

`FENCE`

Explanation

`FENCE` prevents the turtle from moving beyond the edge of the graphics window. If you try to move the turtle off the window, it does not move and Logo displays a message.

See also `WRAP` and `WINDOW`.

Example

FILL

Syntax

```
FILL
(FILL number)
```

Explanation

FILL fills an area on the graphics screen. The area is FILLed with the current pen mode, color and pattern. FILL starts at the current turtle position and stops at a closed border of the current pen color.

If FILL has an argument, FILL stops at a closed border of the pen color given as input. FILL and its argument must be enclosed in parentheses.

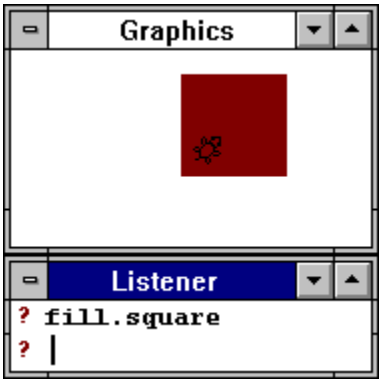
If the turtle's pen state is PENDOWN, FILL colors the area with the current pen color. If the pen state is PENERASE, FILL colors the area with the current background color, thus erasing the form. If the pen state is PENREVERSE, FILL colors the area with the complement of the current area color; for example, white turns to black, and green to red. If the turtle's pen state is PENUP, FILL has no effect.

If the turtle is on a horizontal or vertical line, C changes the color of the line and other connected vertical and horizontal lines to the current pen color. If the line is slanted, C has no effect.

Example

```
TO FILL.SQUARE
   REPEAT 4 [FD 40 RT 90]
   PENUP RT 45 FD 20 PENDOWN
   FILL
   SETPC 0
END
```

This procedure first draws a square, then the turtle is moved inside this square and it is filled. To illustrate the position of the turtle after the fill, the pen color is set to 0.



FILE.INFO

Syntax

```
FILE.INFO filename.ext
```

Explanation

FILE.INFO outputs an eight element list containing the following information about the specified file:

[Attributes Size Year Month Day Hour Minute Second]

See also DIRECTORY.

Example

```
? FILE.INFO "TUNA.LGO
Result: [32 256 1993 2 14 12 25 50]
? _
```

The file TUNA.LGO is 256 bytes in size and was last modified on February 14, 1993 at 12:25:50 p.m.

FILE?

Syntax

```
FILE? filename.ext
```

Explanation

FILE? outputs TRUE if the file specified by its input exists on the disk; otherwise, it outputs FALSE.

Use a drive specifier to access the disk that is not currently selected. Since a colon is a delimiter in Logo, it must be preceded by a backslash (\) to be read correctly.

See also DIRECTORY, FILE.INFO, LOAD, and SAVE.

Example

```
? SAVE "SESSION1
Saving workspace in file SESSION1.LGO
Result: TRUE
? FILE? "SESSION1.LGO
Result: TRUE
? FILE? "SESSION2.LGO
Result: FALSE
? _
```

FILLARRAY

Syntax

```
FILLARRAY array list
```

Explanation

`FILLARRAY` initialized the array or bytearray named in its first input with the data in the list given as its second input. If the list is non-structured, the array is filled sequentially regardless of its dimensions. If the list is structured, the array is filled according to the structure of the list and its dimensions.

Example

```
? MAKE "A ARRAY [2 2]
? FILLARRAY "A [WORD.1 WORD.2 WORD.3 WORD.4]
? LISTARRAY "A
Result: [[WORD.1 WORD.2][WORD.3 WORD.4]]
? _
```

FIRST

Syntax

```
FIRST word or list
```

Explanation

`FIRST` outputs the first element of its input. If the input is a word, `FIRST` outputs the first character. If the input is a list, `FIRST` outputs the first element of that list.

See also `BUTFIRST`, `BUTLAST`, and `LAST`.

Examples

```
? FIRST "TOADSTOOL
Result: T
? FIRST [TABLE CHAIR STOOL]
Result: TABLE
? FIRST [[FEBRUARY 14][APRIL 26][JUNE 19]]
Result: [FEBRUARY 14]
? _
? TO INITIALS :LIST
>    IF :LIST = [] STOP
>    PR FIRST FIRST :LIST
>    INITIALS BF :LIST
> END
INITIALS defined.
? INITIALS[INTERNATIONAL BUSINESS MACHINES PERSONAL COMPUTER]
I
B
M
P
C
? _
```

FKEY.n

Syntax

```
MAKE "FKEY.n word or list
```

Explanation

The `FKEY.n` variables are built-in system variables which correspond to the function keys F1 through F10. `n` may be one of the values 1 to 10. Each of the variables may be assigned a string which is stored into the keyboard buffer and executed like keyboard input when the coressponding function key is pressed together with the `Shift` key.

Whan PC Logo starts up, the following text strings are assigned to the function key variables:

FKEY.1	HELP	(cannot be changed)
FKEY.2	SPLITSCREEN	
FKEY.3	TEXTSCREEN	
FKEY.4	FULLSCREEN	
FKEY.5	LOAD "	
FKEY.6	SAVE "	
FKEY.7	LOADPIC "	
FKEY.8	SAVEPIC "	
FKEY.9	CLEARTEXT	
FKEY.10	EDIT	

FONT

Syntax

```
FONT
```

Explanation

The `FONT` command outputs the current font of the first active turtle.

Example

FONTS

Syntax

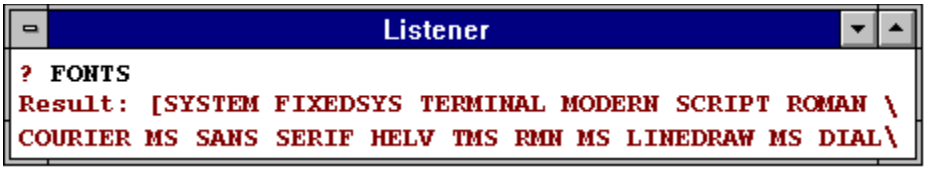
```
FONTS
```

Explanation

The `FONTS` command outputs a list of all available fonts which can be used for output by means of the `TURTLETEXT` command.

See also `FONT`, `SETFONT` and C.

Example



FOR

Syntax

```
FOR word number number list
(FOR word number number list number)
```

Explanation

FOR lets you execute a list of Logo commands a given number of times. Inputs to FOR are a control variable, a beginning value, an ending value, and a list of Logo commands to be executed. FOR assigns the beginning value to the variable, executes the run list, and then increments the variable by one. This process is repeated until the value of the variable equals the ending value.

The variable increment step can be changed to a number other than one by listing the increment as a fifth input to FOR and enclosing FOR and all its inputs in parentheses.

See also REPEAT.

Examples

```
? FOR "I 1 4 [PRINT :I]
1
2
3
4
? (FOR "I 1 4 [PRINT :I] 2)
1
3
? _
```

FORWARD (FD)

Syntax

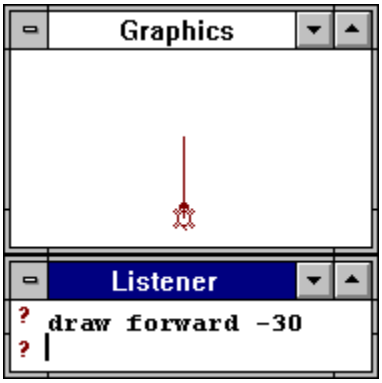
```
FORWARD number
FD number
```

Explanation

`FORWARD` moves the turtle forward the distance specified by its input. See also `BACK`.

Examples

The turtle moves **forward** 30 steps.



The turtle moves **backwards** 30 steps.

FPUT

Syntax

```
FPUT word1 or list 1 word2 or list2
```

Explanation

`FPUT` outputs an object which is created by putting the first input at the beginning of the second input.

If the first input is a list, the second must be a list. If both inputs are words, `FPUT` outputs a word.

See also `LIST`, `LPUT`, `SENTENCE`, and `WORD`.

Examples

```
? FPUT "A "BC
Result: ABC
? FPUT 1 23
Result: 123
? FPUT "A [GREEN CHEVY]
Result: [A GREEN CHEVY]
? FPUT [NORTH DAKOTA] [NEW HAMPSHIRE]
Result: [[NORTH DAKOTA] NEW HAMPSHIRE]
? FPUT [NORTH DAKOTA] "UTAH
The procedure FPUT needs a list as its second input.
? _
```

FROMMEMBER (FM)

Syntax

```
FROMMEMBER word1 or list1 word2 or list2
FM word1 or list1 word2 or list2
```

Explanation

`FROMMEMBER` outputs a word or list consisting of its second input with all elements removed up until the first occurence of its first input. If the second input is a word, the first input must also be a word. If the second input is a list, the first input can be either a word or list.

Examples

```
? FROMMEMBER "B "ABC
Result: BC
? FROMMEMBER 3 [1 2 3 4 5]
Result: [3 4 5]
? FROMMEMBER "HAT "MANHATTAN
Result: HATTAN
? FM "CHARLIE [ARCHIE BETSY CHARLIE DINAH EDWARD FRANCIS]
Result: [CHARLIE DINAH EDWARD FRANCIS]
? _
```

FULLSCREEN (FS)

Syntax

```
FULLSCREEN
```

Explanation

`FULLSCREEN` brings the graphics window to full size, thus hiding any other window.

See also `TEXTSCREEN` and `SPLITSCREEN`.

GETATTR

Syntax

GETATTR

Explanation

GETATTR outputs the attribute most recently set with the SETATTR primitive. When Logo loads,the default attribute is 240.

See also TEXTBG and TEXTFG.

Examples

```
? GETATTR
Result: 240
? SETATTR 242
? GETATTR
Result: 3
? _
```

GETBYTE

Syntax

```
GETBYTE
```

Explanation

GETBYTE outputs the ASCII value of the first character in the input stream. If no character is waiting to be read, GETBYTE waits for input from the keyboard.

See EOF, GETBYTE.NO.ECHO, and PUTBYTE.

Examples

The following procedure prints all input typed until the Enter key is pressed.

```
TO WAIT.FOR.ENTER.KEY
   TEST GETBYTE = 13
   IFFALSE [WAIT.FOR.ENTER.KEY]
END
```

```
? WAIT.FOR.ENTER.KEY
I WILL PRESS THE ENTER KEY NOW
```

GETBYTE.NO.ECHO

Syntax

```
GETBYTE.NO.ECHO
```

Explanation

GETBYTE.NO.ECHO outputs the ASCII value of the first character in the input stream, but does not print it on the screen. If no character is waiting to be read, GETBYTE.NO.ECHO waits for input from the keyboard.

See EOF, GETBYTE, and PUTBYTE.

Example

The procedure below reads in characters without echoing them to the screen and then prints out the next character in the ASCII set. It ends on Q (which is ASCII 81).

```
TO CONFUSE
   LABEL "TOP
   MAKE "A GETBYTE.NO.ECHO
   IF :A = 81 THEN STOP
   PUTBYTE :A + 1
   GO "TOP
END
```

GETMODE

Syntax

```
GETMODE
```

Explanation

`GETMODE` outputs the number of the current screen mode. In Windows, this value will be meaningless.

GETPALLET

Syntax

```
GETPALLET
```

Explanation

The number of the current pallet is returned. The default value of the pallet when Logo loads is 0. There are four different pallets of 256 pen colors each available.

To change the pallet, use SETPALLET.

See also COLOR for a detailed explanation of how to work with colors, PENCOLOR, SETCOLOR, and SETPC.

Example

```
? GETPALLET
Result: 0
? _
```

GETXY

Syntax

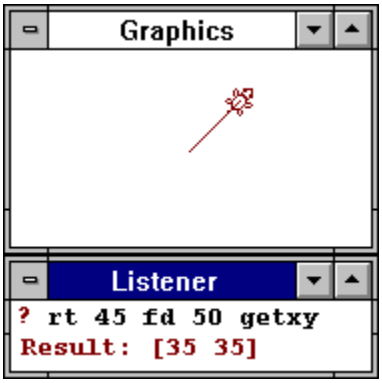
`GETXY`

Explanation

`GETXY` outputs a list consisting of the x and y coordinates of the turtle.

See also `SETX`, `SETXY`, `SETY`, `XCOR`, and `YCOR`.

Example



GO

Syntax

```
GO object
```

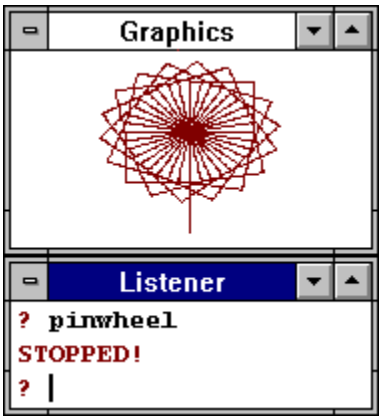
Explanation

GO transfers the flow within a procedure to the line immediately following its corresponding LABEL command. GO and its corresponding LABEL must reside within the same procedure.

Example

```
TO PINWHEEL
   FD 100
   LABEL "LOOP
   REPEAT 4 [FD 50 RT 90]
   RT 20
   GO "LOOP
END
```

Use Control-G or Control-Break to stop this procedure.



GPROP

Syntax

```
GPROP name propertyname
```

Explanation

GPROP gets the property value of a name that has been assigned a property with PPROP. If the property list does not exist, GPROP outputs the empty list.

See also PLIST, PPROP, PPROPS and REMPROP.

Examples

```
? PPROP "CAR "TIRES 4
? PPROP "CAR "DOORS 2
? GPROP "CAR "TIRES
Result: 4
? GPROP "CAR "DOORS
Result: 2
? GPROP "CAR "MPG
Result: []
? _
```

HALT

Syntax

```
HALT name
(HALT name name name ...)
(HALT)
```

Explanation

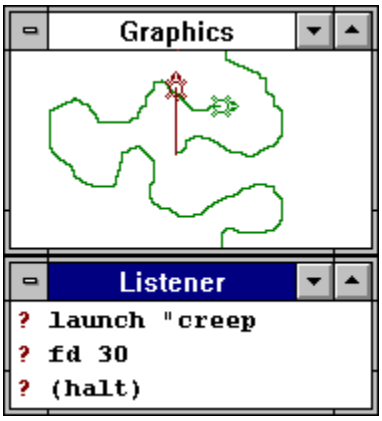
`HALT` stops the execution of any running background procedure. It input is the name of the procedure to be stopped. Optionally, you can supply more than one name. If `HALT` is used without inputs, all running background procedures will be stopped.

See also `LAUNCH`.

Example

The procedure below will send turtle 1 to a random location on the screen every time it is called. If this procedure is installed as a background procedure, the turtle will creep along the screen while letting you enter commands and other procedures.

```
TO CREEP
   LOCAL "TELL.LIST
   MAKE "TELL.LIST WHO
   TELL 1
   SETPC 2 ST
   SETH HEADING + (RANDOM 60) - 30
   FORWARD RANDOM 20
   TELL :TELL.LIST
END
```



HEADING

Syntax

```
HEADING
```

Explanation

`HEADING` outputs the turtle's heading, an integer from 0 to 359. Straight up is 0, to the right is 90, down is 180, and to the left is 270.

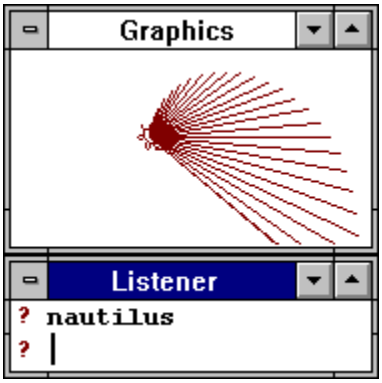
Use `SETHEADING` to set the turtle's heading.

`DRAW` and `CLEARSCREEN` set the turtle's heading to 0.

Example

```
TO NAUTILUS
    RT 5
    FORWARD HEADING
    BACK HEADING
    IF HEADING < 130 THEN NAUTILUS
END
```

This procedure slowly moves the turtle around, drawing longer lines as the heading increases.



HELP

Syntax

```
HELP primitive name
HELP
```

Explanation

`HELP` exits Logo and brings the Logo Help system on the screen. The Logo Help system contains definitions and examples of all Logo primitives and system names. If `HELP` is used with no arguments, it takes you to the beginning of the Logo Help system. If `HELP` is used with a primitive name as an argument, it takes you to the definition and example of that primitive.

You can also invoke help for a specific keyword by moveing the cursor of the word and pressing the F1 function key.

HIDETURTLE (HT)

Syntax

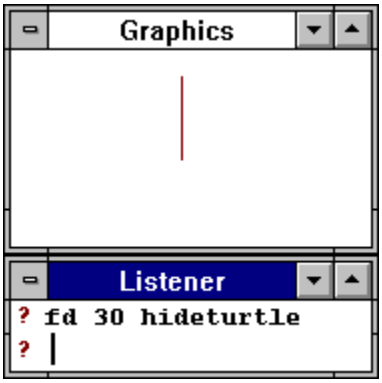
```
HIDETURTLE
HT
```

Explanation

`HIDETURTLE` makes the current turtle(s) disappear from the screen. The turtle continues to draw, but cannot be seen. The turtle draws much faster when it is hidden.

Also see `SHOWTURTLE` and `SHOWN?`

Example



HOME

Syntax

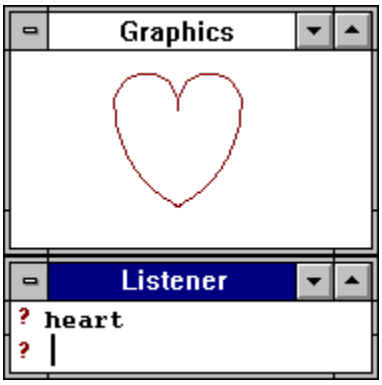
```
HOME
```

Explanation

`HOME` moves the turtle to the center of the screen, points the turtle straight up (`HEADING` 0), but does not clear the graphics screen or alter the pen state.

Example

```
TO HEART
    REPEAT 10 [FD 5 RT 18]
    REPEAT 22 [FD 3 RT 3]
    PU HOME PD
    REPEAT 10 [FD 5 LT 18]
    REPEAT 22 [FD 3 LT 3]
    HT
END
```



IBASE

Syntax

```
MAKE "IBASE number
```

Explanation

IBASE is a pre-defined name which sets the base in which numbers are input to PC Logo. IBASE requires an integer between 2 and 16 as its input.

The base in which numbers are output by Logo is separately controlled by the system name BASE.

Examples

```
? MAKE "IBASE 16
? PRINT 10
16                          (10 in base 16 is 16 in base 10.)
? MAKE "IBASE 2
? PRINT 10
2                           (10 in base 2 is 2 in base 10.)
? _
```

IF

Syntax

```
IF statement [then-instructions] [else-instructions]
IF statement THEN instructions ELSE instructions
```

Explanation

`IF` runs the instructions if the result of the conditional statement is `TRUE`. If the conditional statement is `FALSE`, nothing is done unless `ELSE` follows the instructions.

See also `IFFALSE` and `IFTRUE`.

Examples

```
? TO TRY :NUMBER
>    IF :NUMBER >100 THEN PR [THE NUMBER IS TOO BIG] STOP
>    FD :NUMBER
>    LT 90
> END
TRY defined.
? TRY 85
? TRY 105
THE NUMBER IS TOO BIG
? _
```

IFFALSE (IFF)

Syntax

```
IFFALSE instructionlist
IFF instructionlist
```

Explanation

IFFALSE runs the instruction list if the most recent TEST operation is FALSE. If the TEST operation is TRUE, IFFALSE does nothing.

See also TEST and IFTRUE.

Examples

```
? TEST 5 = 6
? IFFALSE [PR [NOPE, IT AIN'T]]
NOPE, IT AIN'T
? TEST 5 = 5
? IFFALSE [PR [NOPE, IT AIN'T]]
? _
```

IFTRUE (IFT)

Syntax

```
IFTRUE instructionlist
IFT instructionlist
```

Explanation

IFTRUE runs the instruction list if the most recent TEST operation is TRUE. If the TEST operation is FALSE, IFTRUE does nothing. See also IFFALSE.

Examples

```
? TEST 5 = 5
? IFTRUE [PR [YES, IT IS]
YES, IT IS
? TEST 5 = 6
? IFTRUE [PR [YES, IT IS]
? _
```

IGNORE

Syntax

```
IGNORE procedure
```

Explanation

`IGNORE` simply "swallows" any output created by its input, a Logo procedure. `IGNORE` is very handy if the output of any procedure is not needed.

Example

```
? IGNORE INT 50.5
? _
```

INT

Syntax

```
INT number
```

Explanation

`INT` outputs the integer portion of its input by removing the decimal portion, if any. No rounding occurs.

See also `ROUND`.

Examples

```
? INT 2.345
Result: 2
? INT 2.789
Result: 2
? INT 57.999
Result: 57
? _
```

ITEM

Syntax

```
ITEM number word or list
```

Explanation

`ITEM` outputs the nth element from the second input where n is the first input and the second input is a number, word, or list.

See also `MEMBER?`.

Examples

```
? ITEM 3 "CAT
Result: T
? ITEM 2 753
Result: 5
? ITEM 3 [IN AT ON]
Result: ON
? _
```

KEY?

Syntax

`KEY?`

Explanation

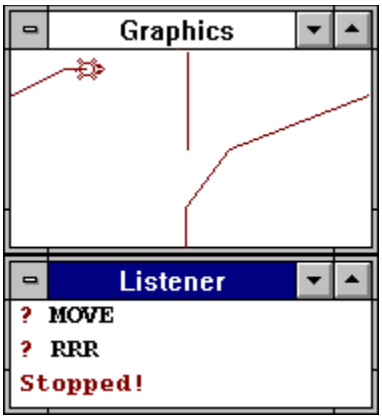
`KEY?` outputs `TRUE` if a character is available from the keyboard; otherwise, it outputs `FALSE`.

Example

The procedures below moves the turtle forward until the R or L keys are pressed to turn the turtle.

```
TO MOVE
   FD 5
   IF KEY? THEN COMMAND
   MOVE
END

TO COMMAND
   MAKE "CHOICE READCHAR
   IF :CHOICE = "R THEN RT 30
   IF :CHOICE = "L THEN LT 30
END
```


Graphics
Listener
? MOVE
? RRR
Stopped!

LABEL

Syntax

```
LABEL object
```

Explanation

`LABEL` marks the beginning of a GO-LABEL loop. Its input must match the input of the corresponding `GO` command. Used in conjunction with `GO`.

Example

```
TO PINWHEEL
   FD 100
   LABEL "LOOP
   REPEAT 4 [FD 50 RT 90]
   RT 20
   GO "LOOP
END
```

To stop this procedure, use Control-G.

LAST

Syntax

`LAST word` or `list`

Explanation

`LAST` outputs the last element of its input. If the input is a word or number, `LAST` outputs the last character of the word or number. If the input is a list, `LAST` outputs the last element of the list.

See also `BUTFIRST`, `BUTLAST`, and `FIRST`.

Examples

```
? LAST 987
Result: 7
? LAST "MOUSE
Result: E
? LAST [EENIE MEENIE MYNIE MO]
Result: MO
? _
```

LAUNCH

Syntax

```
LAUNCH name
(LAUNCH name name name ...)
```

Explanation

`LAUNCH` launches a Logo procedure to be run in the background. This procedure runs simultaneously with other Logo procedures. Since a background procedure is executed at the end of the execution of every Logo statement, these procedures should be kept as small as possible.

Background procedures may unexpectedly alter the value of any Logo variable.

See also `HALT`.

Example

The procedure below sends turtle 1 to a random location on the screen every time it is called. If this procedure is installed as a background procedure, the turtle creeps across the screen while letting you enter commands and other procedures.

```
TO CREEP
   LOCAL "TELL.LIST
   MAKE "TELL.LIST WHO
   TELL 1
   SETPC 2 ST
   SETH HEADING + (RANDOM 60) - 30
   FORWARD RANDOM 20
   TELL :TELL.LIST
END
```

LEFT (LT)

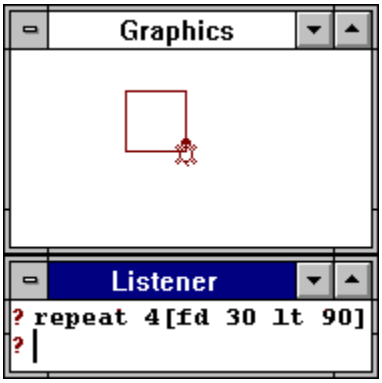
Syntax

```
LEFT number
LT number
```

Explanation

`LEFT` rotates the turtle left (counterclockwise) the number of degrees specified in its input. See also `RIGHT`.

Example



LIST

Syntax

```
LIST word1 or list1 word2 or list2
(LIST word1 or list1 word2 or list2 word3 or list3 . . .)
(LIST word or list)
```

Explanation

LIST outputs a list composed of its inputs. The inputs to LIST can be either words or lists. If the inputs to LIST are themselves lists, LIST preserves them as lists.

LIST expects two inputs, but can accept more if it and all its inputs are enclosed in parentheses.

See also FPUT, LPUT, and SENTENCE.

Examples

```
? LIST "NORTH "CAROLINA
Result: [NORTH CAROLINA]
? LIST "DECEMBER 25
Result: [DECEMBER 25]
? LIST [TO BE] [OR NOT TO BE]
Result: [[TO BE] [OR NOT TO BE]]
? (LIST "JULY 4 [INDEPENDENCE DAY])
Result: [JULY 4 [INDEPENDENCE DAY]]
? _
```

LIST?

Syntax

```
LIST? object
```

Explanation

`LIST?` outputs `TRUE` if its input is a list; otherwise, it outputs `FALSE`.

See also `NUMBER?` and `WORD?`.

Examples

```
? LIST? [GREEN BLUE]
Result: TRUE
? LIST? "GREEN
Result: FALSE
? LIST? []
Result: TRUE
? LIST? 34
Result: FALSE
? LIST? SENTENCE "ROCKY "ROAD
Result: TRUE
? _
```

LISTARRAY

Syntax

```
LISTARRAY array
```

Explanation

LISTARRAY outputs a list whose structure resembles the dimensional structure of the array or bytearray named in its input.

See also ARRAY, BYTEARRAY, FILLARRAY, and FILLARRAY.

Example

```
? MAKE "A BYTEARRAY [2 2]
? ASET :A [0 1] 25
? ASET :A [1 1] 50
? LISTARRAY :A
Result: [[0 25][0 50]]
? _
```

LOAD

Syntax

```
LOAD filename
LOAD filename.ext
(LOAD)
```

Explanation

LOAD transfers the contents of the file specified by its input from the disk to the workspace. The entire file is treated as though it were typed from the keyboard. LOAD outputs TRUE if the file is successfully loaded; otherwise, it outputs FALSE.

Note that the file still exists on the disk; only a copy of it has been transferred to the workspace.

If no file name extension is specified, LOAD loads the file filename.LGO. To load a file that has no extension, a period is necessary after the filename.

If LOAD is used without any inputs, a dialog box will pop up, letting you select the file to be loaded. The dialog box also pops up if the file name contains any wild card characters like * or ?. In this case, the contents of the dialog box are preset to the file name specification.

Drive specifiers (A:, B:, etc.) can be used with LOAD, but all colons must be preceeded with backslash (\). If a drive specifier is not used, LOAD loads a file from the currently selected drive.

Function key F9 is equivalent to LOAD when Logo is first started.

See also LOADPIC, SAVE, and SAVEPIC.

Examples

```
? LOAD "SHAPES
Loading from file SHAPES.LGO
SQUARE is defined.
CIRCLE is defined.
TRIANGLE is defined.
Result: TRUE
? LOAD "B\:ALPHABET
Loading File: ALPHABET.LGO
A is defined.
B is defined.
C is defined.
Result: TRUE
? _
```

If LOAD is used within a procedure, its output (TRUE or FALSE) must be redirected so it will not print on the screen. The procedure below prints text, loads a file, and assigns its output as the value of STUFF so that TRUE or FALSE will not display. Compare it to the result of the preceding example.

```
? TO HIDELOAD
>    MAKE "STUFF LOAD "ALPHABET
? END
HIDELOAD defined.
```

```
? HIDELOAD
Loading File: ALPHABET.LGO
A is defined.
B is defined.
C is defined.
? _
```

To hide individual procedure names as a file loads, see the LOADIT procedure in the explanation of OPEN.

LOADPIC

Syntax

```
LOADPIC filename
LOADPIC filename.ext
(LOADPIC filename "FALSE)
(LOADPIC)
```

Explanation

LOADPIC loads the file filename.PCX from the disk to the graphics screen. LOADPIC clears the current screen to display the picture file. Note that the file still exists on the disk; only a copy of it has been transferred to the workspace.

Drive specifiers (A:, B:, etc.) can be used with LOADPIC, all colons must be preceeded with backslash (\). If a drive specifier is not used, LOADPIC loads a file from the currently selected drive. Function key F7 is equivalent to LOADPIC when Logo starts.

LOADPIC can load any PCX format file. LOADPIC also loads Windows bitmaps (.BMP) and Windows Placeable Metafiles (.WMF).

By default, the picture is loaded and stretched to fit into the window. If the optional third input FALSE is supplied, the picture is not stretched and the graphics window adjusts to fit the size of the picture.

If LOADPIC is used without any inputs, a dialog box pops up, letting you choose a file to load. The dialog box also pops up if the file name contains any wild card characters like * or ?. In this case, the contents of the dialog box are preset to the file name specification.

See also LOAD, SAVE, and SAVEPIC.

Example

LOADSNAP

Syntax

```
LOADSNAP filename
LOADSNAP filename.ext
(LOADSNAP)
```

Explanation

`LOADSNAP` loads the file `filename.PCX` from the disk into a bit map. Note that the file still exists on the disk; only a copy of it has been transferred to the workspace.

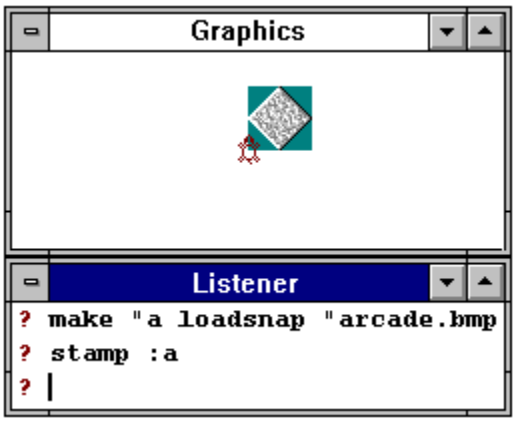
Drive specifiers (`A:`, `B:`, etc.) can be used with `LOADSNAP`. All colons used in a drive name must be preceded with backslash (\). If a drive is not specified, `LOADSNAP` loads the file from the currently selected drive.

`LOADSNAP` can load any PCX format file. `LOADSNAP` also loads Windows bitmap (`.BMP`) files.

If `LOADSNAP` is enclosed in parentheses without inputs or if the file name contains any wildcard characters (`*`, `?`, etc.), a dialog box appears which lets you specify the file to load.

See also SNAP, SNAPSIZE, SAVESNAP and STAMP.

Example



LOCAL

Syntax

```
LOCAL name
(LOCAL name1 name2 . . .)
```

Explanation

LOCAL defines its input as a local variable whose value affects only the procedure in which it is called. The variable's previous value (if any) is saved at the beginning of the procedure where it is redefined and restored at the end of the procedure. The variable is only available within the procedure in which it is defined.

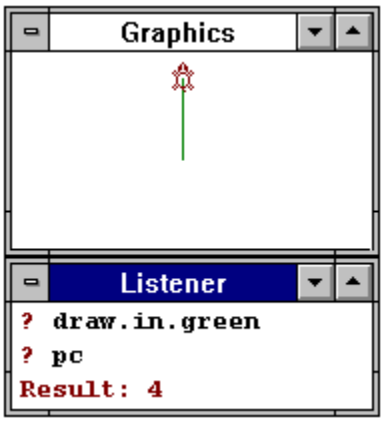
Use PUBLIC to define a variable available to the procedure in which it is defined and any procedures that procedure calls. To define a global variable, use MAKE without LOCAL.

Examples

```
? MAKE "SKY "BLUE
? MAKE "GRASS "GREEN
? PONS
SKY is BLUE
GRASS is GREEN
? TO CHICKEN.LITTLE
> LOCAL "SKY
> MAKE "SKY "FALLING
> (PRINT [THE SKY IS] :SKY)
> END
CHICKEN.LITTLE defined
? CHICKEN.LITTLE
THE SKY IS FALLING
? PONS
SKY is BLUE
GRASS is GREEN
? _
```

The procedure below saves the current drawing color, draws a green line and then restores the current drawing color again.

```
TO DRAW.IN.GREEN
   LOCAL "CURRENT.COLOR
   MAKE "CURRENT.COLOR PC
   SETPC 2
   FD 30
   SETPC :CURRENT.COLOR
END
```


Graphics
Listener
? draw.in.green
? pc
Result: 4

LOG

Syntax

```
LOG number
```

Explanation

`LOG` outputs the natural logarithm of its input.

See also `LOG10`.

Examples

```
? LOG 10
Result: 2.3
? LOG 1
Result: 0
? LOG 2.7183
Result: 1
? _
```

LOG10

Syntax

```
LOG10 number
```

Explanation

`LOG10` outputs the base 10 logarithm of its input.

See also `LOG`.

Examples

```
? LOG10 1
Result: 0
? LOG10 1000
Result: 3
? LOG10 0.001
Result: -3.00
? _
```

LOGAND

Syntax

```
LOGAND integer1 integer2
```

Explanation

LOGAND outputs the bitwise logical AND of its two inputs. Each input is expressed internally as a sixteen digit binary number. A logical AND operation is performed on the pair of binary digits (bits) in each position, resulting in a sixteen bit integer.

The logical AND operation is defined on the binary digits 0 and 1 as follows:

LOGAND 0 0 = 0
LOGAND 1 0 = 0
LOGAND 0 1 = 0
LOGAND 1 1 = 1

See also LOGNOT, LOGOR, LOGXOR.

Examples

```
? LOGAND 2 1
Result: 0
? _
```

2 in base 10 is 10 in base 2; 1 in base 10 is 01 in base 2. In the 1's place, LOGAND 0 1 = 0. In the 2's place, LOGAND 1 0 = 0. Thus, 00 base 2 is obtained. 00 base in 2 is 0 in base 10.

```
? LOGAND 2 3
Result: 2
? _
```

2 in base 10 is 10 in base 2; 3 in base 10 is 11 in base 2. In the 1's place, LOGAND 0 1 = 0. In the 2's place, LOGAND 1 1 = 1. Thus, 10 base 2 is obtained. 10 in base 2 is 2 in base 10.

LOGNOT

Syntax

```
LOGNOT integer
```

Explanation

LOGNOT outputs the bitwise logical complement of its input, replacing all 1's with 0's and all 0's with 1's. Since integers are stored in the computer as 16 base 2 digits long, all the leading 0's turn into 1's.

See also LOGAND, LOGOR, and LOGXOR.

Example

```
? MAKE "BASE 2
? 21
Result: 010101
? LOGNOT 21
Result: 0111111111101010
? _
```

LOGOR

Syntax

```
LOGOR integer1 integer2
```

Explanation

LOGOR outputs the bitwise logical OR of its two inputs. Each input is expressed internally as a sixteen digit binary number. A logical OR operation is performed on the pair of binary digits in each position, resulting in a sixteen bit integer.

The logical OR operation is defined on the binary digits 0 and 1 as follows:

LOGOR 0 0 = 0
LOGOR 1 0 = 1
LOGOR 0 1 = 1
LOGOR 1 1 = 1

See also LOGAND, LOGNOT, and LOGXOR.

Examples

```
? LOGOR 2 1
Result: 3
? _
```

2 in base 10 is 10 in base 2; 1 in base 10 is 01 in base 2. In the 1's place, LOGOR 0 1 = 1. In the 2's place, LOGOR 1 0 = 1. Thus, 11 base 2 is obtained. 11 in base 2 is 3 in base 10.

```
? LOGOR 2 3
Result: 3
? _
```

2 in base 10 is 10 in base 2; 3 in base 10 is 11 in base 2. In the 1's place, LOGOR 0 1 = 1. In the 2's place, LOGOR 1 1 = 1. Thus, 11 base 2 is obtained. 11 base 2 is 3 in base 10.

LOGXOR

Syntax

```
LOGXOR integer1 integer2
```

Explanation

LOGXOR outputs the bitwise logical XOR of its two inputs. Each input is expressed internally as a sixteen digit binary number. A logical XOR operation is performed on the pair of binary digits in each position, resulting in a sixteen bit integer.

The logical XOR operation is defined on the binary digits 0 and 1 as follows:

LOGXOR 0 0 = 0
LOGXOR 1 0 = 1
LOGXOR 0 1 = 1
LOGXOR 1 1 = 0

See also LOGAND, LOGNOT, and LOGOR.

Examples

```
? LOGXOR 2 1
Result: 3
? _
```

2 in base 10 is 10 in base 2; 1 in base 10 is 01 in base 2. In the 1's place, LOGXOR 0 1 = 1. In the 2's place, LOGXOR 1 0 = 1. Thus, 11 base 2 is obtained. 11 in base 2 is 3 in base 10.

```
? LOGXOR 2 3
Result: 1
? _
```

2 in base 10 is 10 in base 2; 3 in base 10 is 11 in base 2. In the 1's place, LOGXOR 0 1 = 1. In the 2's place, LOGXOR 1 1 = 0. Thus, 01 base 2 is obtained. 01 in base 2 is 1 in base 10.

LPUT

Syntax

`LPUT word1` or `list1 word2` or `list2`

Explanation

`LPUT` outputs a new object which is created by placing the first input at the end of the second input.

If the first input is a list, the second cannot be a word. If both inputs are words, `LPUT` will output a word.

See also `FPUT`, `LIST`, `SENTENCE`, and `WORD`.

Examples

```
? LPUT "ISSIPPI "MISS
Result: MISSISSIPPI
? LPUT [COLORADO] [MISS]
Result: [MISS [COLORADO]]
? LPUT [GREEN] "BLUE
The procedure LPUT needs a list as its second input.
? LPUT FIRST [X Y Z] [A B C D]
Result: [A B C D X]
? _
```

LSH

Syntax

```
LSH integer integer
```

Explanation

`LSH` outputs the first input logically shifted the number of bit positions specified by the second input. If the second input is positive, the logical shift is to the right. If the second input is negative, the logical shift is to the left.

Examples

```
? LSH 2 1
Result: 1
? _
```

2 in base 10 is 10 in base 2. 10 shifted one bit right is 1; 1 in base 2 is 1 in base 10.

```
? LSH 2 -1
Result: 4
? _
```

2 in base 10 is 10 in base 2. 10 shifted one bit left is 100; 100 in base 2 is 4 in base 10.

MAKE

Syntax

```
MAKE name object
```

Explanation

MAKE defines a variable using the name of the first input and assigns the second input as the value of that variable.

Once you have created the variable, you can get its contents by using :name. Think of the colon (:) as "the value of name."

To keep a variable local to the procedure in which MAKE is used, see LOCAL or PUBLIC. See also NAME and THING.

Examples

```
? MAKE "NUMBER 73
? :NUMBER
Result: 73
? MAKE "COLOR "MAGENTA
? :COLOR
Result: MAGENTA
? MAKE "CHOICE FIRST [A B C D]
? :CHOICE
Result: A
? TO NAME.A.TREE
>   PR [WHAT IS THE LATIN NAME OF THAT TREE?]
>   MAKE "ANSWER READLIST
>   PR SENTENCE :ANSWER [HAS RED LEAVES]
> END
NAME.A.TREE defined.
? NAME.A.TREE
WHAT IS THE LATIN NAME OF THAT TREE?
? ACER ROBUSTUS
ACER ROBUSTUS HAS RED LEAVES
? _
```

MCI

Syntax

```
MCI command
MCI command command ...)
```

Explanation

MCI provides an interface to the Windows 3.1 multimedia extensions. MCI works much like the TYPE command. Instead of printing its inputs, however, the resulting text is sent to the MCI command line interface and executed by Windows.

Some of the MCI commands return values. Logo collects these values into a list which is the output of this command. The default output value is the empty list [].

The Multimedia Command Interface help can be invoked with the **Help/MCI commands...** menu command.

See also MCI?.

Examples

The following command sequence opens the Windows sound file DING.WAV and plays it. Note that the PLAY command also is able to play sound files.

```
? MCI [OPEN |C:\WINDOWS\DING.WAV| TYPE WAVEAUDIO ALIAS DINGSOUND]
Result: [1]
? MCI [SEEK DINGSOUND TO START]
Result: []
? MCI [PLAY DINGSOUND WAIT]
Result: []
? MCI [CLOSE DINGSOUND]
Result: []
? _
```

MCI?

Syntax

`MCI?`

Explanation

`MCI?` outputs `TRUE` if Logo detects the Windows 3.1 multimedia extensions, otherwise it outputs `FALSE`.

See also `MCI`.

Example

```
? MCI?
Result: TRUE
? _
```

MEMBER?

Syntax

```
MEMBER? object1 object2
```

Explanation

`MEMBER?` outputs `TRUE` if the first input is an element of the second input; otherwise, it outputs `FALSE`.

Examples

```
? MEMBER? "A [A B C]
Result: TRUE
? MEMBER? "A [X Y Z]
Result: FALSE
? MEMBER? "A "CAT
Result: TRUE
? MEMBER? "SALE [BARN SALE]
Result: TRUE
? MEMBER? "SALE [[BARN SALE]]
Result: FALSE
? MEMBER? 2 14236
Result: TRUE
? _
```

MOUSE

Syntax

```
MOUSE
```

Explanation

`MOUSE` outputs the current mouse coordinates in a list of two numbers representing the x coordinate and the y coordinate. If no mouse is present, the coordinates are always [0 0].

See also `BUTTON?` and `.MOUSEON`.

Example

The following procedure uses the mouse to move the turtle. When the left mouse button is held down, the turtle will draw as it moves. Press the right mouse button to end the procedure.

```
TO CHASE
   LABEL "L
   IF BUTTON? 1 THEN PD ELSE PU
   SETXY MOUSE
   IF NOT BUTTON? 2 THEN GO "L
END
```

MOUSESHAPE

Syntax

```
MOUSESHAPE
```

Explanation

MOUSESHAPE outputs the number of the current mouse shape as defined with the SETMOUSESHAPE command. On startup, the mouse shape is 1.

The available mouse shapes are:


1


2


3


4


5


6


7


8


9


10


11

12


13


14


15


16


17

18


19


20


21


22

Example

NAME

Syntax

```
NAME object name
```

Explanation

NAME defines a variable using the name of the second input and assigns the first input as the value of that variable.

Once you have created the variable, you can get its contents by using :name. Think of the colon (:) as "the value of name."

NAME is equivalent to MAKE except that inputs are in reverse order.

To keep a variable local to the procedure in which NAME or MAKE is used, see LOCAL and PUBLIC.

Examples

```
? NAME 73 "NUMBER
? :NUMBER
Result: 73
? NAME "MAGENTA "COLOR
? :COLOR
Result: MAGENTA
? NAME FIRST [A B C D] "CHOICE
? :CHOICE
Result: A
? _
```

NAME?

Syntax

```
NAME? name
```

Explanation

`NAME?` outputs `TRUE` if the input is a name of a variable; otherwise; it outputs `FALSE`.

See also `LIST?`, `NUMBER?`, and `WORD?`.

Examples

```
? NAME? "ANIMAL
Result: FALSE
? MAKE "ANIMAL "CAT
? NAME? "ANIMAL
Result: TRUE
? _
```

NOCASE

Syntax

NOCASE

Explanation

NOCASE causes Logo to become case sensitive, so that Logo differentiates between upper case and lower case letters. NOCASE is an abbreviation for NO CASE CONVERSION. What you type in lower case characters is interpreted by Logo as different from upper case.

When Logo loads, its default state is CASE, where Logo internally converts lower case letters to their upper case equivalent. The current case status of Logo is returned by CASE?.

Examples

```
? CASE?
Result: CASE
? print [Your friends are here]
YOUR FRIENDS ARE HERE
? nocase
? print [Your friends are here]
print is not a Logo procedure.
? PRINT [Your friends are here]
Your friends are here
? CASE
? print [Your friends are here]
YOUR FRIENDS ARE HERE
? _
```

NODES

Syntax

NODES

Explanation

NODES outputs the total number of free nodes in memory. NODES provides an estimate of how much space is left to name variables and write and run procedures.

Example

```
? NODES
Result: 2020
? _
```

NOT

Syntax

```
NOT object1
```

Explanation

`NOT` outputs `TRUE` if its input is false; otherwise, it outputs `FALSE`.

Examples

```
? NOT "FALSE
Result: TRUE
? NOT "TRUE
Result: FALSE
? NOT NUMBER? "A
Result: TRUE
? IF NOT (3 = 3) THEN PRINT "YES
? _
```

NUMBER?

Syntax

```
NUMBER? object
```

Explanation

NUMBER? outputs TRUE if its input is a number; otherwise, it outputs FALSE.

See also LIST?, NAME?, and WORD.

Examples

```
? NUMBER? 41
Result: TRUE
? NUMBER? [41]
Result: FALSE
? NUMBER? FIRST [41]
Result: TRUE
? NUMBER? 4.1
Result: TRUE
? _
```

OPEN

Syntax

```
OPEN filename
(OPEN filename mode)
```

Explanation

OPEN prepares for input or output the DOS file or device specified by its input, and then outputs its assigned Logo stream number. Data may then be read using READ, READCHAR, READLINE, READLIST, READQUOTE and other Logo primitives by making STANDARD.INPUT the stream number. If the specified file or device does not exist, OPEN outputs FALSE.

Because a colon (:) is a delimiter, it must be preceded with \ to be read correctly by Logo. See also CLOSE and CREATE.

Legal MS-DOS device names include:

CON\:	Keyboard and screen
AUX\: or COM1\:	First asynchronous communications adapter
COM2\:	Second asynchronous communications adapter
LPT1\: or PRN\:	First parallel printer
LPT2\:	Second parallel printer
LPT3\:	Third parallel printer
NUL\:	Null device

OPEN may optionally be supplied with a third input which describes the open mode. This is a string consisting of one or more characters. The flowing modes are supported:

"R	The file is opened for reading only. If the file does not exist, an error is generated.
"W	The file is opened for writing only. If the file does not exist, it is created. Any data in the file is overwritten.
"RW	The file is opened for both reading and writing. If the file does not exist, it is created.
"A	The file is opened for writing only. If the file does not exist, it is created. The data written is appended to the end of the file

Optionally, this string may be followed by the letter **"B** which marks the file as binary. Normally, reading stops at the end of a line feed character, and line feed characters written are translated into CR/LF character pairs. If the file is opened in binary mode all data is transferred untranslated.

The commands .READ, .WRITE and .SEEK are available for low level I/O.

Examples

```
? OPEN "PRN\:
Result: 1
? _
```

The following procedure prevents procedure names from displaying on the screen when Logo loads the file PICTURES. Instead, it redirects the Logo output stream to a null device.

```
TO LOADIT
   MAKE "STANDARD.OUTPUT OPEN "NUL\:
   MAKE "LOADED? LOAD "PICS.LGO
```

```
    MAKE "STANDARD.OUTPUT 0
  END
```

```
? LOAD "PICS.LGO
Loading File: PICS.LGO
CIRCLE is defined.
SQUARE is defined.
TRIANGLE is defined.
? LOADIT
PICS.LGO
? _
```

OR

Syntax

```
OR object1 object2
(OR object1 object2 object3 . . .)
(OR object1)
```

Explanation

OR outputs FALSE if all of its inputs are false; otherwise, it outputs TRUE. OR accepts one or more inputs which must be either TRUE or FALSE.

Examples

```
? OR "TRUE "TRUE
Result: TRUE
? OR "TRUE "FALSE
Result: TRUE
? OR "FALSE "FALSE
Result: FALSE
? (OR "FALSE "TRUE "FALSE)
Result: TRUE
? IF OR (2=3) (3=3) PRINT "YES
YES
? _
```

ORIGIN

Syntax

```
ORIGIN
```

Explanation

The `ORIGIN` command outputs the coordinate system origin of the first active turtle in the form of a list with two integers, the first being the X value and the second being the Y value. The coordinates output by `ORIGIN` are relative to the standard turtle coordinate system, where [0 0] is the center of the window.

See also `SETORIGIN`.

Example

OUTPUT (OP)

Syntax

```
OUTPUT object
OP object
```

Explanation

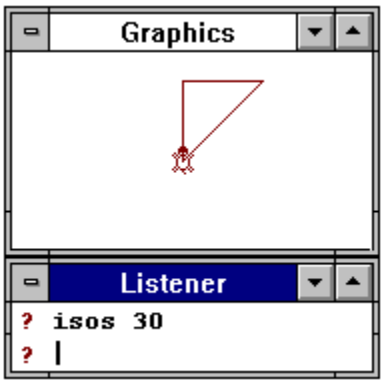
`OUTPUT` makes its input the output of the procedure. `OUTPUT` can only be used within a procedure. After the object of `OUTPUT` is run, control returns to the calling procedure or to toplevel.

Example

`HYPOT` calculates the hypotenuse of an isosceles right triangle with the Pythagorean Theorem (C*C = A*A + B*B) and uses `OUTPUT` to send that value to the `ISOS` procedure.

```
TO HYPOT :SIDE
   OUTPUT SQRT (2 * (:SIDE * :SIDE))
END

TO ISOS :SIDE
   FD :SIDE
   RT 90
   FD :SIDE
   RT 135
   FD HYPOT :SIDE
   RT 135
END
```


Graphics
Listener
? isos 30
?

PATTERN

Syntax

```
PATTERN
```

Explanation

`PATTERN` outputs a number which represents the current fill pattern of the graphics screen.

To set the fill pattern of the graphics screen, use `SETPATTERN`.

The available fill patterns are:

1  **2**

 **3**

 **4**

 **5**

6  **7**

 **8**

 **9**

 **10**

11 

If `SETPATTERN` was called with a user defined pattern (a list of eight numbers between 0 and 255), `PATTERN` outputs this list.

Example

PAUSE

Syntax

```
PAUSE
```

Explanation

PAUSE temporarily halts the execution of a procedure. PAUSE makes it possible to add commands to an ongoing procedure or check the execution of a procedure. It is also possible to enter the editor while in a pause to change the procedure.

When Logo pauses, it prints PAUSE and the prompt changes to the word PAUSE followed by a greater than sign (PAUSE >). The value of the PAUSE prompt is stored in the system variable name :PAUSE and can be changed.

To resume execution of the procedure, type CO or CONTINUE. To return to toplevel, type TOPLEVEL or Control-G.

PAUSE may be used only within a procedure. To pause a procedure that is running at toplevel, use Control-Z.

Example

```
TO SQUARE
   DRAW
   FORWARD 60
   RT 90
   PAUSE
   REPEAT 3 [FD 60 RT 90]
END
```

PEEKBYTE

Syntax

```
PEEKBYTE
```

Explanation

`PEEKBYTE` outputs the `ASCII` value of the next character waiting on the input stream. The character is left in the input stream for subsequent `READ`, `READCHAR`, `READLINE`, `READLIST`, `READQUOTE`, or other input primitives.

Example

The following procedure prints the text of a file in Logo without loading the file into workspace.

```
TO TYPE.FILE :FILE
   LOCAL "CHANNEL
   MAKE "CHANNEL OPEN :FILE
   IF :CHANNEL = "FALSE (PR :FILE [NOT FOUND.]) STOP
   MAKE "STANDARD.INPUT :CHANNEL
   NOCASE
   WHILE [NOT PEEKBYTE = :EOF] [PRINT READQUOTE] CASE
   CLOSE "STANDARD.INPUT
   MAKE "STANDARD.INPUT 0
END
```

PEN

Syntax

PEN

Explanation

PEN outputs the pen mode of the current turtle. Available modes are PENDOWN, PENUP, PENERASE, and PENREVERSE.

The output of PEN is the same as the input to SETPEN.

Example

```
? DRAW
? PEN
Result: PENDOWN
? PENREVERSE
? PEN
Result: PENREVERSE
? _
```

PENCOLOR (PC)

Syntax

```
PENCOLOR
PC
```

Explanation

`PENCOLOR` outputs the current pen color. Use `SETPC` to alter the drawing color.

Example

PENDOWN (PD)

Syntax

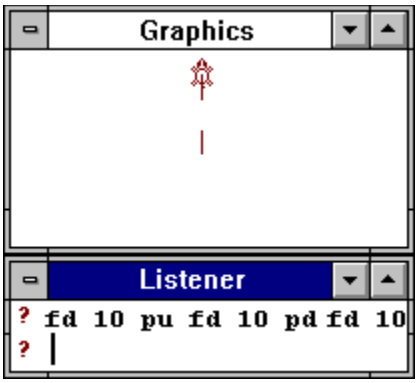
```
PENDOWN
PD
```

Explanation

`PENDOWN` puts the turtle's pen down and draws a line when the turtle moves. Used in conjunction with `PENUP`.

`DRAW` puts the pen down. See also `PENERASE` and `PENREVERSE`.

Example



PENERASE (PE)

Syntax

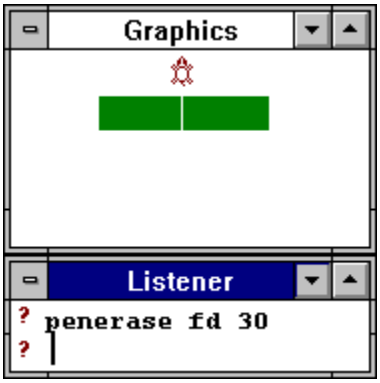
```
PENERASE
PE
```

Explanation

`PENERASE` turns the turtle's pen into an eraser. When the turtle moves, it appears to erase by drawing in the current background color.

To stop `PENERASE`, use `PENDOWN`, `PENUP` or `SETPEN`.

Example



PENREVERSE (PX)

Syntax

```
PENREVERSE
PX
```

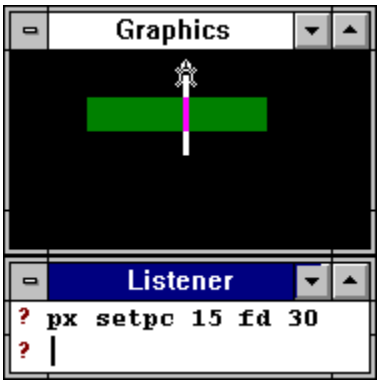
Explanation

`PENREVERSE` reverses the pen and background color when the turtle moves, drawing where there are no lines and erasing previously drawn lines.

The combination creates many possible effects. The exact appearance of the screen depends on the pen and background colors.

`DRAW` puts the pen down. See also `PENDOWN`, `PENUP` and `PENERASE`.

Example



PENUP (PU)

Syntax

```
PENUP
PU
```

Explanation

`PENUP` puts the turtle's pen up. When the turtle moves, it does not draw a line. Used in conjunction with `PENDOWN`.

See also `PENERASE` and `PENREVERSE`.

Example

PI

Syntax

`PI`

Explanation

`PI` outputs the value of . The number of digits in `PI` is determined by the current value of `PRECISION`.

Example

```
? MAKE "PRECISION 2
? PI
Result: 3.14
? _
```

PICK

Syntax

`PICK list` or `word`

Explanation

`PICK` picks a randomly selected element from either a list or a word.

Example

```
? MAKE "MUSIC [JAZZ POP ROCK CLASSIC]
? PICK :MUSIC
Result: ROCK
? PICK :MUSIC
Result: JAZZ
? _
```

PLAY

Syntax

```
PLAY list
PLAY filename
```

Explanation

PLAY causes the computer's speaker to play musical notes as specified by its input list. PLAY will accept lists as input that contain special musical commands, such as the note and octave to play, and the length of time to sound the note. The following elements can be included in a list that is input to PLAY.

Musical Notes: A B C D E F G P (for pause)
Note prefixes: An integer immediately preceding a note determines its duration. 1 is a whole note, 2 is a half note, 4 is a quarter note, 8 is an eighth note, etc.
Note suffixes: # following a note indicates it is a sharp note, b following a note indicates it is a flat note; . following a note extends its duration to 3/2 time
Octaves: O followed by an integer between 0 and 6 sets the octave which is preset to 3. O# raises one octave, Ob lowers one octave
Tempo: T followed by an integer sets the tempo in units of quarter notes per minute. T is preset to 120.
Note length: L followed by an integer sets the default note length if no duration is specified with the individual note. L is preset to 4 (for quarter notes).
Staccato: S followed by an integer between 0 and 100 sets the staccato ratio as a percentage. S0 is a perfect legato. S is preset to 10.
Reset: R resets note length, octave, tempo, and staccato to their preset values. Changes in these values are preserved during multiple calls to PLAY unless they are changed again or reset to their preset values.
Chords: A list within the PLAY list is treated as a chord. Intervals and length specifiers are ignored on the first notes in the chord list; only the last note is played in the given length. The first notes are played quickly. PLAY [C [C E G O# C] Ob E] will play a C, a chord ending with a high C and an E.

If PLAY is used with a word as input, Logo assumes a file name for a waveform file. If the Windows 3.1 multimedia extensions are present, Logo plays this file.

Examples

```
? PLAY [2A# 2P Ob Ab]
? _
```

sounds an A-sharp half note, pauses for the duration of a half note, lowers by one octave, and sounds an A-flat quarter note.

```
? PLAY "|C:\WINDOWS\DING.WAV|
?
```

The Windows waveform file DING.WAV is played.

PLIST

Syntax

```
PLIST name
```

Explanation

PLIST outputs the property list associated with its input. The property list is a list of a property name or names paired with its property value or values. PLIST stands for "Property List."

See also GPROP, PPROP, PPROPS, POPLS and REMPROP.

Examples

```
? PPROP "SHOES "SIZE 6
? PPROP "HAT "COLOR "BROWN
? PLIST "SHOES
Result: [SIZE 6]
? PLIST "HAT
Result: [COLOR BROWN]
? _
```

POC

Syntax

POC

Explanation

POC prints all user-defined constants. POC is equivalent to PRINTOUT CONSTANTS.

See also PONS, POPS, POPLS, and POTS.

Example

```
? CONST "TWO 2
? CONST "THREE 3
? POC
TWO is 2
THREE is 3
? PR TWO * THREE
6
? _
```

PONS

Syntax

PONS

Explanation

PONS prints all user-defined names and their values. PONS is equivalent to PRINTOUT NAMES.

See also POC, POPS, POPLS, and POTS.

Example

```
? MAKE "SKY "BLUE
? MAKE "GRASS "GREEN
? PONS
GRASS is GREEN
SKY is BLUE
? _
```

POPLS

Syntax

POPLS

Explanation

POPLS prints out all defined property lists. POPLS is equivalent to PRINTOUT PROPERTIES.

See also POC, PONS, POPS, and POTS.

Example

```
? PPROP "JOE "SEX "MALE
? PPROP "JOE "AGE 24
? POPLS
JOE is [SEX MALE AGE 24]
? _
```

POPS

Syntax

POPS

Explanation

POPS prints all user-defined procedures and their definitions. POPS is equivalent to PRINTOUT PROCEDURES. To print out only procedure titles, use POTS.

See also POC, PONS and POPLS.

Example

```
? TO LINE
>    FD 30
> END
LINE defined.
? TO CIRCLE
>   REPEAT 360 [FD 1 RT 1]
> END
CIRCLE defined.
? POPS
TO CIRCLE
   REPEAT 360 [FD 1 RT 1]
END
TO LINE
   FD 30
END
? _
```

POTS

Syntax

POTS

Explanation

POTS prints the titles of all user-defined procedures. POTS stands for "Print Out Titles" and is an abbreviation for PRINTOUT TITLES

To print out procedure definitions as well as titles, use POPS. To print out procedure titles in list format, use PROCLIST.

See also PONS and POPLS.

Example

```
? TO LINE :LENGTH
>    FD :LENGTH
> END
LINE defined.
? TO CIRCLE
>   REPEAT 360 [FD 1 RT 1]
> END
CIRCLE defined.
? POTS
TO LINE :LENGTH
TO CIRCLE
? _
```

PPROP

Syntax

```
PPROP name propertyname object
```

Explanation

PPROP assigns a property pair to a Logo name. PPROP takes three inputs: the first, the name with which a property list should be associated; the second, the property name; the third, the property value.

The first input to PPROP must be a word; the second and third inputs can be either a word or a list. PPROP stands for "Put Property."

A property pair consists of a property name and its value.

See also GPROP, PLIST, POPLS, PPROPS, and REMPROP.

Examples

```
? PPROP "MUSIC "COMPOSER "STRAVINSKY
? PPROP "MUSIC "COMPOSITION "PETROUCHKA
? PLIST "MUSIC
Result: [COMPOSITION PETROUCHKA COMPOSER STRAVINSKY]
? _
```

PPROPS

Syntax

```
PPROPS name list
```

Explanation

PPROPS is a handy method to store multiple properties into a property list. The second input to PPROPS is a list of property pairs. The first word is the property, the second value is the property value for that property. The list must be of even length.

See also GPROP, POPLS, PPROP, and REMPROP.

Example

```
? PPROPS "JOE [SEX MALE AGE 24]
? GPROP "JOE "AGE
Result: 24
? GPROP "JOE "SEX
Result: MALE
? _
```

PRECISION

Syntax

```
MAKE "PRECISION number
```

Explanation

`PRECISION` is a pre-defined name that sets the number of decimal places displayed in Logo calculations. The default value of `PRECISION` when Logo is loaded is 2, the maximum allowed number is 6.

Examples

```
? PI
Result: 3.14
? MAKE "PRECISION 6
? PI
Result: 3.141593
? _
```

PRINT (PR)

Syntax

```
PRINT object
(PRINT object1 object2 . . .)
PR object
(PR object1 object2 . . .)
```

Explanation

`PRINT` prints its inputs to the output stream andd adds a carriage return. If the input is a list, `PRINT` removes the brackets.

See also `TYPE` and `SHOW`.

Examples:

```
? PRINT "HELLO
HELLO
? PRINT [HI HOW ARE YOU?]
HI HOW ARE YOU?
? (PRINT "TWO "WORDS)
TWO WORDS
? _
```

PRINTER

Syntax

PRINTER

Explanation

PRINTER outputs a list of four values:

- the name of the printer
- the port to which it is connected
- the paper width
- the paper length

See also PRINTSCREEN and SETPRINTER.

Example

```
? PRINTER
Result: [HP Laserjet III LPT1: 1200 1580]
? _
```

PRINTLINE

Syntax

```
PRINTLINE integerlist
```

Explanation

PRINTLINE prints to the output stream the ASCII characters corresponding to the elements of its input list. A carriage return is not inserted, so if the output of PRINTLINE is printed on the screen, the prompt appears after the last character of the input list.

To output the ASCII codes for alphabetic characters, use READLINE.

Examples

```
? PRINTLINE [71 65 82 66 76 69]
GARBLE? PRINTLINE [84 87 79 32 66 65 84 83]
TWO BATS? PRINTLINE [116 119 111 32 98 97 116 11 5]
two bats? _
```

PRINTOUT (PO)

Syntax

```
PRINTOUT procname
PRINTOUT ALL
PRINTOUT NAMES
PRINTOUT PROCEDURES
PRINTOUT PROPERTIES
PRINTOUT CONSTANTS
PRINTOUT TITLES
PO procname
PO ALL
PO NAMES
PO PROCEDURES
PO CONSTANTS
PO TITLES
```

Explanation

PRINTOUT prints the names, definitions, and values specified by its input.

PRINTOUT ALL prints out all procedure titles, definitions, variable names and values, and property lists. PRINTOUT ALL can be abbreviated to PO ALL.

PRINTOUT NAMES prints out all user-defined variable names and values. PRINTOUT NAMES can be abbreviated to PO NAMES or PONS.

PRINTOUT PROCEDURES prints out all user-defined procedure titles and definitions. PRINTOUT PROCEDURES can be abbreviated to PO PROCEDURES or POPS.

PRINTOUT CONSTANTS prints out all user-defined constants. PRINTOUT CONSTANTS can be abbreviated to PO CONSTANTS or POC.

PRINTOUT TITLES prints out all user-defined procedure titles. PRINTOUT TITLES can be abbreviated to PO TITLES or POTS.

Examples

```
? TO LINE
>   FD 40
> END
LINE defined.
? TO CIRCLE
>   REPEAT 360 [FD 1 RT 1]
> END
CIRCLE defined.
? MAKE "SKY "BLUE
? MAKE "GRASS "GREEN
? PPROP "KATE "EYES "BROWN
? PPROP "KURT "EYES "HAZEL
? PO CIRCLE
TO CIRCLE
   REPEAT 360 [FD 1 RT 1]
END
```

```
? PO ALL
TO CIRCLE
   REPEAT 360 [FD 1 RT 1]
END

TO LINE
   FD 40
END

GRASS is GREEN
SKY is BLUE
PPROP "KURT "EYES "HAZEL
PPROP "KATE "EYES "BROWN
? _
```

PRINTSCREEN (PS)

Syntax

```
PRINTSCREEN
PS
```

Explanation

The PRINTSCREEN command prints the graphics window. PRINTSCREEN outputs TRUE if the screen is successfully printed; otherwise, it outputs FALSE.

See also PRINTER, and SETPRINTER.

PROCLIST

Syntax

PROCLIST

Explanation

PROCLIST outputs a list of all user-defined procedures currently in the workspace.

See also POPS, POTS, and PRINTOUT.

Example

```
? TO LINE :LENGTH
>    FD :LENGTH
> END
LINE defined.
? TO CIRCLE
>   REPEAT 360 [FD 1 RT 1]
> END
CIRCLE defined.
? POTS
TO LINE :LENGTH
TO CIRCLE
? PROCLIST
Result: [LINE CIRCLE]
? _
```

PRODUCT

Syntax

```
PRODUCT number number
```

Explanation

`PRODUCT` outputs the product of its inputs. `PRODUCT` expects two inputs, but will accept more if it and all its inputs are enclosed in parentheses.

`PRODUCT` is equivalent to `*`.

Examples

```
? PRODUCT 2 3
Result: 6
? PRODUCT 4 -1.2
Result: -4.80
? PRODUCT -.5 -1.5
Result: 0.75
? (PRODUCT 2 3 4 5)
Result: 120
? TO CUBE :NUM
>    PRINT (PRODUCT :NUM :NUM :NUM)
? END
CUBE defined.
? CUBE 3
27
? _
```

PROMPT

Syntax

```
MAKE "PROMPT object
```

Explanation

PROMPT is a pre-defined name whose value is the Logo prompt. PROMPT can be changed to any Logo object using the MAKE statement. When Logo starts up, the default value of PROMPT is a question mark, followed by a space.

Remember that if you change the prompt into a character that has a special Logo meaning, it must be preceded by a backslash (\).

Example

```
? MAKE "PROMPT "\*
*PR [THIS IS A NEW PROMPT]
THIS IS A NEW PROMPT
*MAKE "PROMPT "|? |
? _
```

PRTRACE

Syntax

```
PRTRACE object
(PRTRACE object1 object2 . . .)
```

Explanation

PRTRACE prints its inputs to the trace window and adds a carriage return. If the input is a list, PRTRACE removes the brackets. PRTRACE is handy for adding debugging messages during program development.

See also PRINT, TYPE and SHOW.

Examples:

```
? PRTRACE "HELLO
? PRTRACE [HI HOW ARE YOU?]
? _
```

PUBLIC

Syntax

```
PUBLIC name
(PUBLIC name name ...)
```

Explanation

`PUBLIC` defines its input as a local variable which is invisible in the global workspace. Unlike the `LOCAL` command which is specific to only one procedure, a variable defined as `PUBLIC` is available for all procedures called by the procedure where the variable is defined.

See also `LOCAL` and `MAKE`.

Examples

```
? TO PROC.A
> LOCAL "LOCAL.VAR MAKE "LOCAL.VAR "LOCAL
> PUBLIC "PUB.VAR MAKE "PUB.VAR "PUBLIC
> PR [WITHIN PROCEDURE A]
> PONS
> PROC.B
> END
PROC.A defined.
? TO PROC.B
> PR [WITHIN PROCEDURE B]
> PONS
> END
PROC.B defined.
? PROC.A
WITHIN PROCEDURE A
LOCAL.VAR is LOCAL
PUB.VAR is PUBLIC
WITHIN PROCEDURE B
PUB.VAR is PUBLIC
? _
```

PUTBYTE

Syntax

```
PUTBYTE number
```

Explanation

PUTBYTE prints to the output stream the character corresponding to its ASCII input. The input number can be from 0 through 255. PUTBYTE does not insert a carriage return after printing its output.

See also GETBYTE, GETBYTE.NO.ECHO, PEEKBYTE, PRINTLINE, and READLINE.

Examples

```
? PUTBYTE 65
A? PUTBYTE 97
a?
? PRINT [MANY          SPACES]
MANY SPACES
? TYPE "MANY REPEAT 10 [PUTBYTE 32] PR "SPACES
MANY          SPACES
? _
```

QUOTIENT

Syntax

```
QUOTIENT number number
```

Explanation

`QUOTIENT` outputs the result of dividing the first input by the second input.

See also `/` and `REMAINDER`.

Examples

```
? QUOTIENT 10 5
Result: 2
? QUOTIENT 10 3
Result: 3.33
? QUOTIENT 10 30
Result: 3.33E-01
? QUOTIENT -10 3
Result: -3.33
? _
```

RANDOM

Syntax

```
RANDOM number
```

Explanation

RANDOM outputs a randomly selected number from 1 through its input. The output can only be a positive integer. For example:

RANDOM 5

could output 1, 2, 3, 4, or 5. See also RERANDOM.

Examples

```
? RANDOM 4
Result: 2
? RANDOM 4
Result: 3
? RANDOM 10
Result: 7
? _
```

READ

Syntax

READ

Explanation

READ outputs the first object from the input stream. If no object is waiting to be read, READ waits for input from the keyboard. READ outputs EOF if the end of file is reached.

See also READCHAR, READLINE, READLIST and READQUOTE.

Example

```
? TO GREETING
>    PR [WHAT'S YOUR NAME?]
>    MAKE "RESPONSE READ
>    PR (SE [HI THERE,] :RESPONSE "\!)
> END
GREETING defined.
? GREETING
WHAT'S YOUR NAME?
? ELEANOR
HI THERE, ELEANOR !
? GREETING
WHAT'S YOUR NAME?
? ELEANOR RIGBY
HI THERE, ELEANOR !
RIGBY is not a Logo procedure.
? _
```

READCHAR (RC)

Syntax

```
READCHAR
RC
```

Explanation

`READCHAR` outputs the first character from the input stream. If no character is waiting to be read, `READCHAR` waits for input from the keyboard.

See also `READ`, `READLINE`, `READLIST`, and `READQUOTE`.

Example

The procedure below can be used to pause between a series of procedures, such as a game, and wait for user input.

```
TO HOLDUP
   PR [PRESS Y IF YOU ARE READY TO CONTINUE]
   IF READCHAR = "Y PLAYGAME
   HOLDUP
END
```

READLINE

Syntax

```
READLINE
```

Explanation

READLINE outputs the next line (up to a carriage return) from the input stream as a list of ASCII characters. If no line is waiting to be read, READLINE waits for input from the keyboard.

To output characters from the corresponding ASCII codes, use PRINTLINE.

See also READ, READCHAR, READLIST, and READQUOTE.

Examples

```
? READLINE
? ASCII
Result: [65 83 67 73 73]
? READLINE
? SHOPPING CARTS
Result:[83 72 79 80 80 73 78 71 32 67 65 82 84 83]
? NOCASE
? READLINE
? shopping carts
Result:[115 104 111 112 112 105 110 10 3 32 99 97 114 116 115]
? _
```

READLIST (RL)

Syntax

```
READLIST
RL
```

Explanation

READLIST outputs in the form of a list the next line (up to a carriage return) from the input stream. If no line is waiting to be read, READLIST waits for input from the keyboard.

See also READ, READCHAR, READLINE and READQUOTE.

Example

```
? TO ASKIT
>    PR [WHAT ARE YOUR FAVORITE FOODS?]
>    MAKE "FOODS READLIST
>    PR [I KNOW A RESTAURANT WHERE THEY SERVE]
>    PR :FOODS
>    PR [THAT WOULD MAKE YOUR MOUTH WATER.]
> END
ASKIT defined.
? ASKIT
WHAT ARE YOUR FAVORITE FOODS?
? JAM, PEACHES, AND BEEF JERKY
I KNOW A RESTAURANT WHERE THEY SERVE
JAM, PEACHES, AND BEEF JERKY
THAT WOULD MAKE YOUR MOUTH WATER.
? _
```

READQUOTE (RQ)

Syntax

```
READQUOTE
RQ
```

Explanation

READQUOTE outputs the next line (up to a carriage return) from the input stream as a single Logo name. If no line is waiting to be read, READQUOTE waits for input from the keyboard.

READQUOTE is useful to define names that contain delimiters or characters that would otherwise need to be quoted with \.

See also READ, READCHAR, READLINE and READLIST.

Example

```
? MAKE "X READQUOTE
? IBM-PC
? PRINT :X
IBM-PC
? MAKE "Y READLIST
? IBM-PC
? PRINT :Y
IBM - PC
? _
```

RECYCLE

Syntax

```
RECYCLE
(RECYCLE "TRUE)
```

Explanation

RECYCLE causes Logo to perform a garbage collection which frees memory by clearing information that is no longer in use. Unless RECYCLE is used, Logo normally performs a garbage collection when memory is no longer available. When programming time-dependent procedures, such as a melody, use RECYCLE at a point in the procedures that a garbage collection delay will not interfere with your program. Otherwise Logo carries out a garbage collection whenever one is necessary. It takes a second or longer to perform a garbage collection which can cause delays in running programs.

At toplevel, RECYCLE outputs the number of free nodes. When RECYCLE is given the argument TRUE and both it and its argument are enclosed in parentheses, recycle outputs the number of free slots for symbols and numbers, the number of free list elements and the available memory, both locally and globally. See also NODES.

Example

```
? RECYCLE
Result: 4998
? (RECYCLE "TRUE)
Recycle #3
Symbols and numbers: 3587
List elements: 5884
Workspace memory: 200014 bytes
Global heap: 2599552 bytes
Result: 1884
? _
```

REMAINDER

Syntax

```
REMAINDER number number
```

Explanation

REMAINDER outputs an integer which is the remainder of dividing the first input by the second. See also / and QUOTIENT.

Examples

```
? REMAINDER 6 3
Result: 0
? REMAINDER 159 2
Result: 1
? REMAINDER 689 468
Result: 221
? _
```

REMPROP

Syntax

```
REMPROP name propertyname
```

Explanation

REMPROP removes a property and its value from the name with which it is associated. REMPROP stands for "Remove Property."

ERASE erases everything in memory, including all property lists.

See also GPROP, PLIST, POPLS, and PPROP.

Example

```
? PLIST "ANIMALS
Result: [FURRY FOX FLAT FLOUNDER]
? REMPROP "ANIMALS "FURRY
? PLIST "ANIMALS
Result: [FLAT FLOUNDER]
? _
```

RENAME

Syntax

```
RENAME filename1.ext filename2
```

Explanation

`RENAME` changes the name of the file specified by its first input to its second input. If the file specified does not exist or if the first and second inputs are the same, `RENAME` outputs `FALSE`, otherwise, it outputs `TRUE`.

Examples

```
? SAVE "LALA
Saving workspace in file LALA.LGO
Result: TRUE
? RENAME "LALA.LGO "LULU
Result: TRUE
? RENAME "LALA.LGO "LOLO
Result: FALSE
? _
```

RERANDOM

Syntax

```
RERANDOM number
(RERANDOM)
```

Explanation

RERANDOM makes RANDOM output identical sequences of numbers. Once you use RERANDOM, it will output the same sequence of random numbers as the first time. Its input is the seed to the random number generator. A different seed yields different random numbers. To achieve truly random numbers, use the current time as a seed. When RERANDOM is enclosed in parentheses without an argument, it uses the current time as its seed.

Example

```
? RERANDOM ((item 1 time) * 3600) + ((item 2 time) * 60) + (item 3 time)
? REPEAT 4 [PR RANDOM 5]
3
2
3
1
? _
```

REPEAT

Syntax

```
REPEAT number list
```

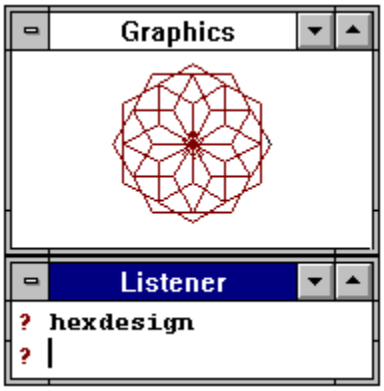
Explanation

REPEAT runs the list of instructions in the second input the number of times indicated by its first input. The number input to REPEAT can be any positive number. If the number is not an integer, its fractional portion is ignored. REPEAT commands can be nested, or placed inside other REPEAT commands. See also FOR.

Examples

```
? REPEAT 5[PR[I WILL NOT BITE MY NAILS]]
I WILL NOT BITE MY NAILS
I WILL NOT BITE MY NAILS
I WILL NOT BITE MY NAILS
I WILL NOT BITE MY NAILS
I WILL NOT BITE MY NAILS
? _
```

```
TO HEXDESIGN
   REPEAT 12[REPEAT 6 [FD 40 LT 60] RT 30]
END
```



RIGHT (RT)

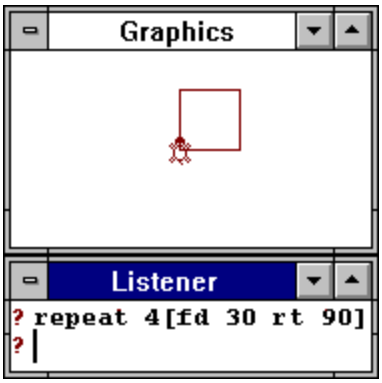
Syntax

```
RIGHT number
RT number
```

Explanation

`RIGHT` rotates the turtle right (clockwise) the number of degrees specified by its input. See also `LEFT`.

Example



ROUND

Syntax

```
ROUND number
```

Explanation

`ROUND` outputs the number rounded to the nearest integer.

See also `INT`.

Examples

```
? ROUND 1.45
Result: 1
? ROUND 1.50
Result: 2
? ROUND 57.99
Result: 58
? _
```

RUN

Syntax

```
RUN list
```

Explanation

`RUN` runs its input as if it were typed in directly. `RUN` outputs whatever its list of instructions outputs.

Example

```
? TO CALCULATOR
>    LOCAL "LIST
>    TYPE [WHAT CALCULATION DO YOU WISH]
>    MAKE "LIST READLIST
>    IF EMPTY? :LIST THEN STOP
>    (PR "RESULT "\= RUN :LIST)
>    CALCULATOR
> END
? CALCULATOR
WHAT CALCULATION DO YOU WISH? 5 + 7
RESULT = 12
WHAT CALCULATION DO YOU WISH? 12 / 4
RESULT = 3
WHAT CALCULATION DO YOU WISH? _
```

SAVE

Syntax

```
SAVE filename
SAVE filename.ext
(SAVE)
```

Explanation

SAVE saves the contents of the workspace to a file on the disk. This includes all defined procedures and names. SAVE outputs TRUE if the file is successfully saved; otherwise, it outputs FALSE.

If no file name extension is specified, SAVE saves the file with the name filename.LGO. To save a file that has no extension, a period is necessary after the filename.

Drive specifiers (A:, B:, etc.) can be used with SAVE, all colons must be preceeded with backslash (\). If a drive specifier is not used, SAVE saves a file to the currently selected drive. Function key F10 is equivalent to SAVE when Logo starts.

If SAVE is used without inputs or if the supplied file name contains wild card characters like ? or *, a dialog box will pop up, letting you select a file name.

See also LOAD, LOADPIC, SAVEPIC and SETDISK.

Examples

```
? SAVE "SHAPES
Saving workspace in file SHAPES.LGO
Result: TRUE
? SAVE "B\:SHAPES
Saving workspace in file SHAPES.LGO
Result: TRUE
? _
```

SAVEPIC

Syntax

```
SAVEPIC filename
SAVEPIC filename.BMP
SAVEPIC filename.WMF
```

Explanation

SAVEPIC saves data on the graphics screen directly to the disk. SAVEPIC outputs TRUE if the file is successfully saved; otherwise, it outputs FALSE. SAVEPIC saves graphics images in the PCX format. They can be viewed with LOADPIC in Logo or with any other program that uses the PCX format. When you use SAVEPIC, the file name will be appended with the extension .PCX unless you indicate otherwise.

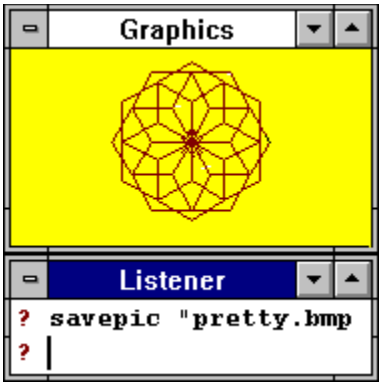
Drive specifiers (A:, B:, etc.) can be used with SAVEPIC, but all colons must be preceeded with backslash (\). If a drive specifier is not used, SAVEPIC saves the file to the currently selected drive. Function key F8 is equivalent to SAVEPIC when Logo starts.

If SAVE is used without inputs or if the supplied file name contains wild card characters like ? or *, a dialog box will pop up, letting you select a file name.

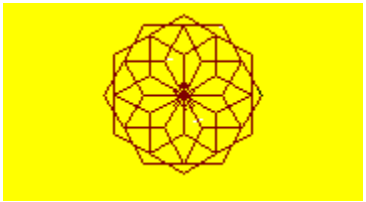
Other valid file formats are BMP (Windows bitmap) and WMF (Windows Placeable Metafile).

See also SAVE and LOAD.

Example



The saved image, PRETTY.BMP, looks like this:



SAVESNAP

Syntax

```
SAVESNAP bitmap filename
SAVESNAP bitmap filename.BMP
SAVESNAP bitmap filename.PCX
(SAVESNAP bitmap)
```

Explanation

SAVESNAP saves a bit map file directly on the disk. SAVESNAP outputs TRUE if the file is successfully saved; otherwise, it outputs FALSE. SAVESNAP saves graphics images in PCX format. They can be viewed with LOADSNAP or LOADPIC. SAVESNAP appends .PCX to your filename unless you specify otherwise.

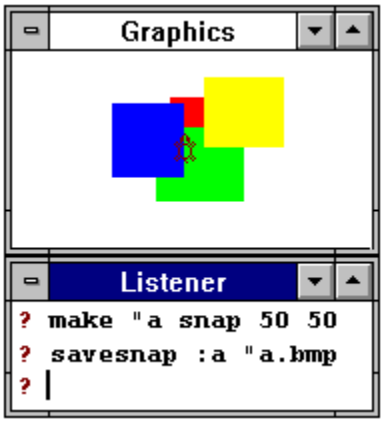
Drive specifiers (A:, B:, etc.) can be used with SAVESNAP. All colons used in a drive name must be preceded with backslash (\). If a drive is not specified, SAVESNAP saves the file on the currently selected drive.

If SAVESNAP is enclosed in parentheses without inputs or if the file name contains any wildcard characters (*, ?, etc.), a dialog box appears which lets you specify the file to save.

SAVESNAP can also save .BMP (Windows bitmap) format files.

See also SNAP, SNAPSIZE, LOADSNAP and STAMP.

Example



The SNAPped image looks like this:

SCREENFACTS (SF)

Syntax

SCREENFACTS

Explanation

SCREENFACTS outputs a list containing information on the screen state:

- the background color
- one of the following keywords:
  TEXTSCREEN if the Listener window is maximized
  FULLSCREEN if the Graphics window is maximized
  SPLITSCREEN if neither is maximized.
- the number of lines that fits into the Listener window
- the window mode: WRAP, FENCE or WINDOW
- the aspect ratio of the graphics windows. This value is always 1.
- the video mode
- the current horizontal and vertical turtle extents

Example

```
? SCREENFACTS
Result: [15 [] 6 WRAP 1 84 [316 118]]
? _
```

SENTENCE (SE)

Syntax

```
SENTENCE object1 object2
SE object1 object2
(SENTENCE object1 object2 object3 . . .)
```

Explanation

SENTENCE outputs a list made up of its inputs. SENTENCE expects two inputs, but will accept more if it and all of its inputs are enclosed in parentheses.

If the inputs to SENTENCE are lists, their brackets are removed and combined into one list.

See also LIST, FPUT, LPUT, and WORD.

Examples

```
? SENTENCE "CRUELLEST "MONTH
Result: [CRUELLEST MONTH]
? SENTENCE [CRUELLEST] [MONTH]
Result: [CRUELLEST MONTH]
? SENTENCE [APRIL IS THE][CRUELLEST MONTH]
Result: [APRIL IS THE CRUELLEST MONTH]
? (SENTENCE  "APRIL "IS "THE "CRUELLEST "MONTH)
Result: [APRIL IS THE CRUELLEST MONTH]
? _
```

SETATTR

Syntax

```
SETATTR number
```

Explanation

SETATTR sets the color of the text and background in the Listener window according to its input. The input to SETATTR is based on the formula:

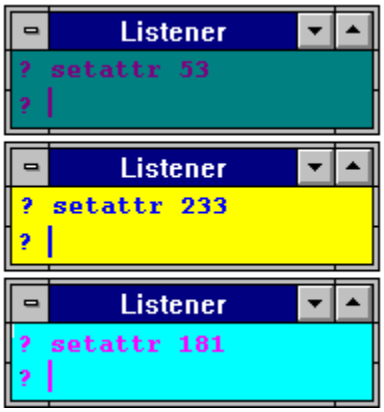
```
(background color) * 16 + foreground color.
```

When Logo loads, the default attribute is 240, which corresponds to a background color of 15 and a foreground color of 0.

Note that only the foreground color of the text that Logo outputs is changed. If you work with syntax highlighting, all the special colors which highlight your text do not change.

See also GETATTR, TEXTBG and TEXTFG.

Examples



SETBG

Syntax

```
SETBG number
(SETBG)
```

Explanation

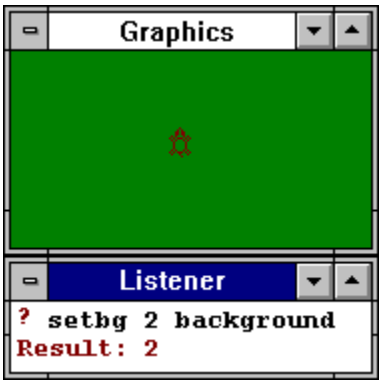
SETBG sets the background color of the graphics screen the color specified by its input. The number is an index into the current palette. The first 16 numbers are predefined as follows:

0 Black	8 Dark Grey
1 Blue	9 Light Blue
2 Green	10 Light Green
3 Cyan	11 Light Cyan
4 Red	12 Light Red
5 Violet	13 Magenta
6 Brown	14 Yellow
7 Light Grey	15 White

If SETBG is used without any inputs, a dialog box pops up, allowing you to pick a background color.

To output the number of the current background color, use BACKGROUND. The current background color displays in the current background pattern. See also SETBGPATTERN and BGPATTERN.

Example



SETBGPATTERN

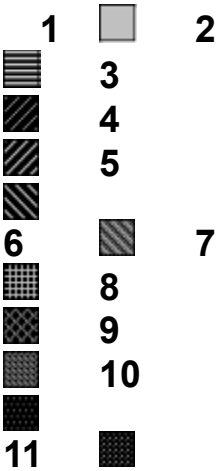
Syntax

```
SETBGPATTERN number
```

Explanation

SETBGPATTERN changes the background pattern to the pattern specified by its input. SETBGPATTERN accepts inputs from 1-12 corresponding to the available background patterns. The pattern appears in the current background color.

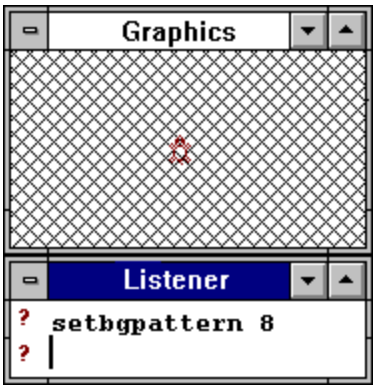
The available background patterns are:



If you use SETBGPATTERN with a list of 8 numbers in the range from 0 to 255, these eight bytes are treated as an 8x8 bit pattern.

See also PATTERN

Example



The background displays pattern 8.

SETCOLOR

Syntax

```
SETCOLOR number list
```

Explanation

The color setting for the given color number in the current pallet is changed. This setting is a list of three values between 0 and 255, where the first value stands for red, the second value for green and the third value for blue.

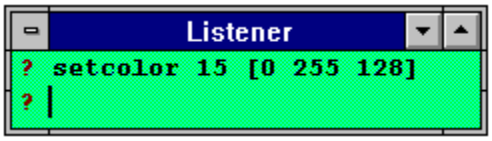
Depending on the graphics card used, the color values are interpreted differently. On an EGA card, only the two most significant bits are interpreted. This makes it possible to create a total of 64 colors. On a VGA card, the six leftmost bits are interpreted, which gives a total of 262,144 colors. Each pallet holds 255 entries, resulting in valid color numbers between 0 to 255.

The color values are interpreted modulo 256, which means that a color value of 257 is interpreted as 1.

See also COLOR, PENCOLOR, and SETPC.

Example

This example redefines color 15 in the current pallet to a bright green.



SETCURDIR

Syntax

```
SETCURDIR name
```

Explanation

`SETCURDIR` changes the current directory to the directory given as its input. To change the current disk drive, use `SETDISK`.

See also `CURDIR`, `DISK` and `SETDISK`.

Example

The following two procedures switch to a new directory and switch back to the old directory.

```
TO CHDIR :NAME
    MAKE "OLD.DIR CURDIR
    SETCURDIR :NAME
END

TO CHBACK
    SETCURDIR :OLD.DIR
END
```

SETDISK

Syntax

```
SETDISK character
```

Explanation

SETDISK sets the default disk drive to the specified name. To output the current default drive, use DISK.

Example

```
? disk
Result: C
? SETDISK "B
? DISK
Result: B
? _
```

SETEXTENT

Syntax

```
SETEXTENT number
(SETEXTENT number number)
SETEXTENT 0
SETEXTENR "PRINTER
```

Explanation

SETEXTENT defines the coordinate mapping mode in terms of turtle steps. The default is an extent value of zero which means the coordinate system has an aspect ratio of 1:1 regardless of the window size. Painting outside the window in WINDOW mode is possible and the drawings become visible when the window is made larger.

If a different extent is defined, the coordinate system is mapped to the size given regardless of the window size. If an extent value of 200 is set, the coordinates range from -200 to +200. Scaling is performed automatically to fit within the window. If the window is resized, the drawing automatically adjusts to fit within the new size according to the value given to SETEXTENT.

If two inputs are given to SETEXTENT and both the command and its inputs are enclosed in parentheses, then the first input is the extent for the X axis while the second input is the extent for the Y axis. (SETEXTENT) or SETEXTENT 0 restores an extent of 1:1. SETEXTENT "PRINTER sets both the extent and the window format to the current printer paper size.

See also EXTENT.

Example

```
? SETEXTENT "PRINTER
? EXTENT
Result: 1169 1691
? _
```

SETFONT

Syntax

```
SETFONT name size attributes
(SETFONT)
```

Explanation

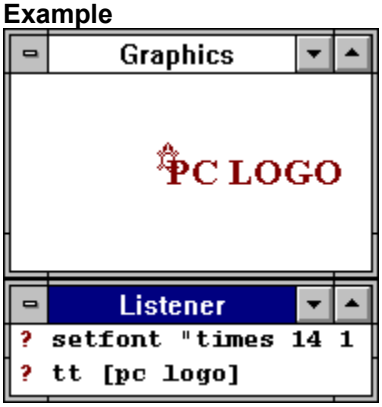
SETFONT defines the turtle font. The first input is the font name. This font should be present in the system. If the font is not present, Windows selects a similar font for you. The second input is the font size, given in points, while the third is a combination of the following attributes:

1	bold
2	*italic*
4	underlined
8	~~strike-out~~

If SETFONT is used without parameters and is enclosed with parentheses, a dialog box appears where you can select a font.

See also FONT, FONTS and TURTLETEXT.

Example



SETHEADING (SETH)

Syntax

```
SETHEADING number
SETH number
```

Explanation

`SETHEADING` turns the turtle to the degree position specified by its input. Positive numbers turn the turtle clockwise.

`SETHEADING` turns the turtle according to the direction of the screen and not the current heading of the turtle. `SETHEADING 0` always heads the turtle straight up despite whatever direction it is pointing.

To output the turtle's heading, use `HEADING`.

Example

```
TO NAUTILUS
    RT 5
    FORWARD HEADING
    BACK HEADING
    IF HEADING < 130 THEN NAUTILUS
END
```

This procedure slowly moves the turtle around, drawing longer lines as the heading increases.

SETMOUSESHAPE

Syntax

```
SETMOUSESHAPE number
(SETMOUSESHAPE)
```

Explanation

`SETMOUSESHAPE` changes the shape of the mouse cursor when it is within the Graphics window. Its input is a number representing one of 23 available mouse shapes. When Logo starts, the mouse shape is 1. If `SETMOUSESHAPE` enclosed in parentheses without inputs, shape 1 is used.

Use `MOUSESHAPE` to output the number of the current mouse shape.

The available mouse shapes are:


1 **2**


3

4


5


6 **7**


8


9


10


11 **12**


13


14


15


16 **17**


18


19


20

21 **22**

23

Example

SETORIGIN

Syntax

```
SETORIGIN [xvalue yvalue]
```

Explanation

The `SETORIGIN` command allows the coordinate system origin point to be set for active turtles. Normally, the coordinate system origin (location [0 0]) is in the center of the window. This origin may be changed for all turtles or individually for any turtle. Input for `SETORIGIN` is a list of two integers, the first being the X value and the second being the Y value. The coordinates are always relative to the standard turtle coordinate system, where [0 0] is the center of the window.

See also `ORIGIN`.

Example

SETPALLET

Syntax

```
SETPALLET number
```

Explanation

SETPALLET changes the current pallet to the given number. There are four pallets available, so SETPALLET will accept a number between 0 and 3. Each pallet has 256 pen colors. Each of the pen colors in each of the pallets can be separately defined with the SETCOLOR command.

SETPALLET changes the colors of all the turtle drawings on the screen at once. By defining different pen colors in different pallets, SETPALLET can be used to change the color of turtle drawings rapidly and dramatically.

When Logo starts up, the pen colors in each of the palletes is the same. See also COLOR, GETPALLET, PEN, PENCOLOR, and SETPC.

SETPATTERN

Syntax

`SETPATTERN number` or `list`

Explanation

`SETPATTERN` select a fill pattern for fill operations. The `PATTERN` command outputs the number of the currently selected fill pattern.

The available fill patterns are:

1  **2**

 **3**

 **4**

 **5**

6  **7**

 **8**

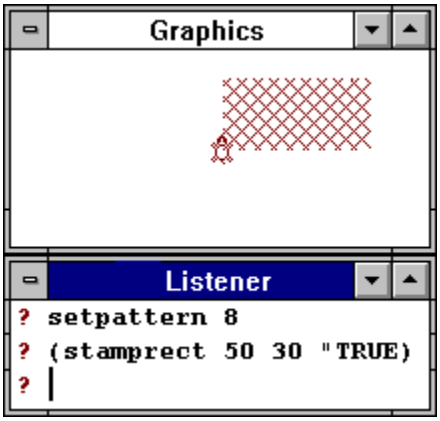
 **9**

 **10**

11 

If `SETPATTERN` is called with a list of eight numbers between 0 and 255, this list is treated as an 8x8 bit pattern to used for fill operations.

Example



SETPC

Syntax

```
SETPC number
(SETPC)
```

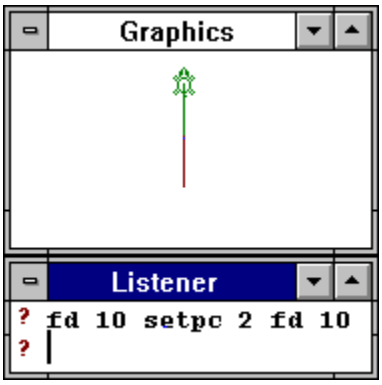
Explanation

`SETPC` selects a pen color for all active turtles. The color number is an index into the current pallet. The first 16 colors are predefined as follows:

0 Black	**8 Dark Grey**
1 Blue	**9 Light Blue**
2 Green	**10 Light Green**
3 Cyan	**11 Light Cyan**
4 Red	**12 Light Red**
5 Violet	**13 Magenta**
6 Brown	**14 Yellow**
7 Light Grey	15 White

If `SETPC` is used without inputs, a dialog box pops up, allowing you to pick a color. Use `PENCOLOR` to obtain the current pen color.

Example



SETPEN

Syntax

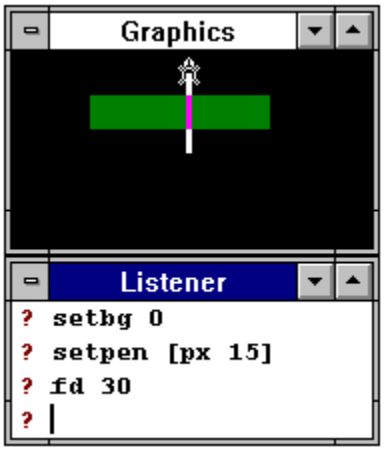
```
SETPEN [penstate pencolor]
```

Explanation

SETPEN changes the state of the turtle's pen and the pen color as specified by its input list. The first element of the list can be PENUP, PENDOWN, PENERASE, or PENREVERSE. The second element is a number from 0 to 255 which specifies the pen color.

Use PEN to output the current turtle's pen state and PENCOLOR to output the current turtle's pen color. See also SETPC.

Example



SETPRINTER

Syntax

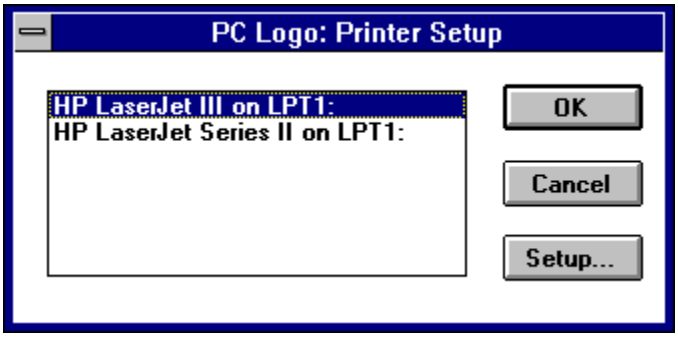
```
SETPRINTER
```

Explanation

`SETPRINTER` causes a dialog box to appear in which you can select the printer you are using with Logo.

Example

```
? SETPRINTER
```



SETSHAPE

Syntax

```
SETSHAPE list
(SETSHAPE list list list list)
(SETSHAPE)
```

Explanation

SETSHAPE redefines the shape of the active turtle(s). Each turtle has four definable shapes, representing headings in ranges of 22.5 degrees. (The computer automatically generates the remaining shapes necessary to position the turtle(s) in a full 360 degree range.) Each of the lists given as input to SETSHAPE defines the turtle's shape through four ranges from HEADING 0 to HEADING 90 in that order.

If SETSHAPE is used with no arguments and enclosed in parentheses, the original turtle shape is restored for all active turtle(s).

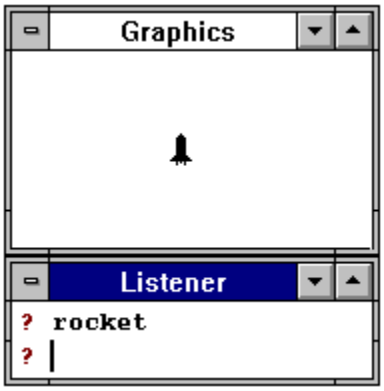
Each list to define a turtle shape contains up to 32 numbers in the range of 0 to 255.

See also SHAPE.

Example

The following procedure changes the turtle shape to a rocket.

```
TO ROCKET
    SETSHAPE [1 0 3 128 7 192 7 192 7 192 7 192 7 192 7 \
              192 7 192 7 192 15 224 31 240 63 248 5 64 4 64]
END
```



SETSPEED

Syntax

```
SETSPEED number
```

Explanation

SETSPEED determines the speed at which the turtle(s) moves on the screen. SETSPEED accepts a number from .1 to 1 as an input. A speed of 1 is the fastest speed and .1 is the slowest. When Logo starts, the turtle speed is 1.

See SPEED.

SETTURTLEFACTS (SETTF)

Syntax

```
SETTURTLEFACTS list
SETTF list
```

Explanation

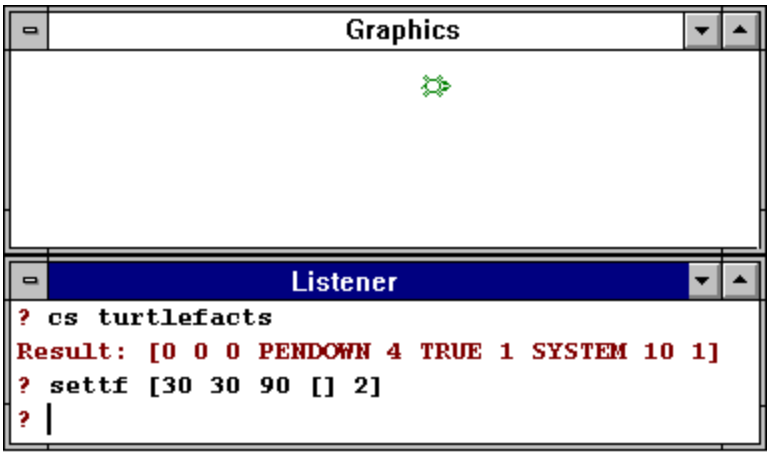
`SETTURTLEFACTS (SETTF)` changes the settings of the active turtles to the values given in the list. The list holds the following elements:

1. The X coordinate.
2. The Y coordinate.
3. The heading in degrees.
4. The pen mode (`PENDOWN`, `PENUP`, `PENERASE`, `PENREVERSE`).
5. The pen color.
6. `TRUE` if the turtle is to be made visible, `FALSE` otherwise.
7. The line width.
8. The turtle font.
9. The font size.
10. The font attributes.

Each element in the list is interpreted according to its position. If one element is the empty list [], this element is ignored and the current settings of the turtles regarding that element remain.

`TURTLEFACTS` returns a list of the current situation of the first active turtle. The list is in the same format as that required for `SETTURTLEFACTS`.

Example



The turtle moves to position [30 30] with a `HEADING` of 90. The pen color changes to green. All other factors stay the same.

SETTURTLES

Syntax

```
SETTURTLES number
```

Explanation

SETTURTLES defines the total number of turtles available. They range from 0 to the input of SETTURTLES minus 1. For example, SETTURTLES 16 will create the turtles [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]. Logo starts up with 16 turtles defined and turtle 0 activated.

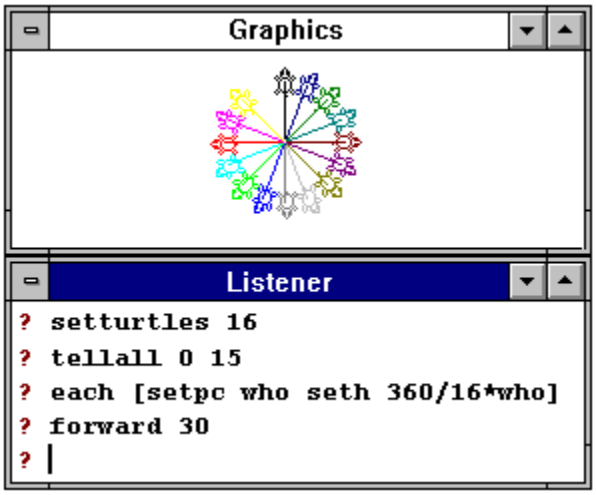
SETTURTLES creates turtles with the following characteristics:

Position:	Home, heading 0
Color:	0 (black)
Line width:	1
Text size:	1
Pattern:	solid

SETTURTLES accepts any number between 1 and 32767 as input. The number of turtles is limited by the memory of the computer system on which Logo is operating. TURTLES returns the number of turtles that have been defined with SETTURTLES.

Use the TELL, ASK, EACH, and WHO commands to access multiple turtles.

Example



SETWIDTH

Syntax

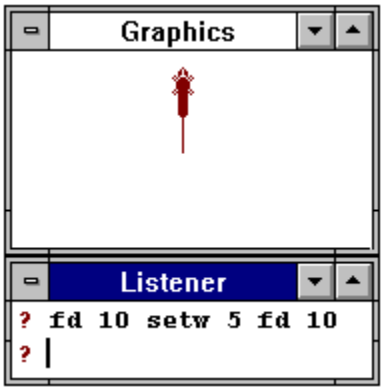
```
SETWIDTH number
```

Explanation

`SETWIDTH` defines the width of the line drawn by all active turtles. `SETWIDTH` will take a number between 1 and 999 as input. `WIDTH` returns the current line width.

See also `PENDOWN`, `PENREVERSE`, and `PENERASE`.

Example



SETWINSIZE

Syntax

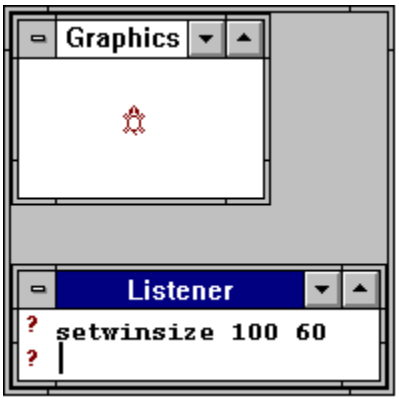
```
SETWINSIZE number number
(SETWINSIZE number number number)
```

Explanation

`SETWINSIZE` sets the size of the Graphics window. Its first input is the width of the window in screen pixels, while the second input is the window height. Use `WINSIZE` to retrieve the current window size.

A third optional input can be supplied to `SETWINSIZE` if `SETWINSIZE` and all its inputs are enclosed in parentheses. The third input is the number of the window as listed in the Window menu.

Example



SETX

Syntax

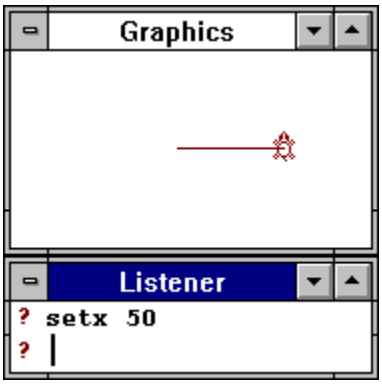
```
SETX xcoordinate
```

Explanation

SETX moves the turtle horizontally to the point specified by the input number. SETX does not affect the turtle's heading or its Y coordinate.

See also GETXY, SETXY, SETY, XCOR, and YCOR.

Example



SETXY

Syntax

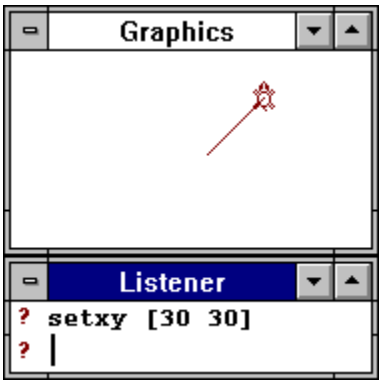
```
SETXY [xcoordinate ycoordinate]
```

Explanation

`SETXY` moves the turtle to the point specified by its input list. The first element is the X coordinate (horizontal); the second, the Y coordinate (vertical).

To output the X and Y coordinates of the turtle, use `GETXY`. See also `SETX` and `SETY`, `XCOR`, and `YCOR`.

Example



SETY

Syntax

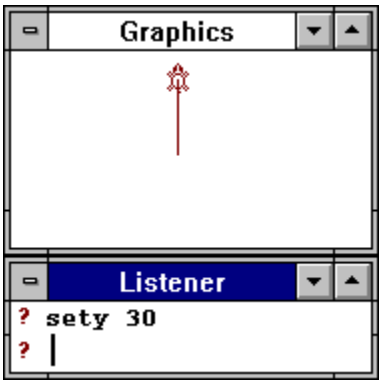
```
SETY ycoordinate
```

Explanation

`SETY` moves the turtle vertically to the point specified by the input number. `SETY` does not affect the turtle's heading or its X coordinate.

See also `GETXY`, `SETX`, `SETXY`, `XCOR`, and `YCOR`.

Example



SHAPE

Syntax

```
SHAPE
(SHAPE number)
```

Explanation

SHAPE outputs the first defined shape for the first active turtle in the form of a list of 32 elements in the range 0 to 255. Since a turtle shape is composed of four different shapes, each representing a HEADING range of 22.5 degrees, a number in the range of 1 to 4 can be used as an optional input. The SHAPE of the turtle(s) can be changed with the SETSHAPE command.

Example

```
? DRAW
? SHAPE
Result: [1 0 2 128 6 192 6 192 20 80 43 168 20 80 8 32 8 32 \
8 32 8 32 20 80 43 168 17 16 0 128 0 0]
? _
```

SHOW

Syntax

```
SHOW object
(SHOW object1 object2 . . .)
```

Explanation

SHOW prints its input to the output stream and inserts a carriage return. SHOW leaves list brackets intact and prints them. Unless the output stream has been redirected, SHOW normally prints its inputs on the computer screen and moves the cursor to the beginning of the next line.

See also PRINT and TYPE.

Examples

```
? SHOW "HELLO
HELLO
? SHOW [NICE DAY]
[NICE DAY]
? SHOW [[LIST] [OF] [LISTS]]
[[LIST] [OF] [LISTS]]
? PRINT [[LIST] [OF] [LISTS]]
[LIST] [OF] [LISTS]
? (SHOW "TWO "WORDS)
TWO WORDS
? _
```

SHOWN?

Syntax

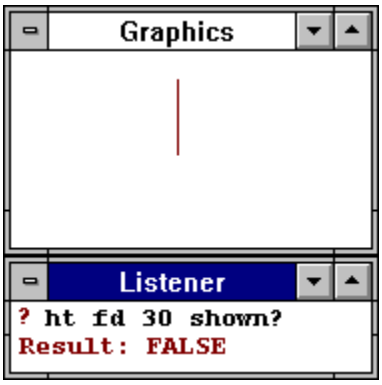
SHOWN?

Explanation

SHOWN? outputs TRUE if the turtle is currently displayed on the graphics screen; otherwise, it outputs FALSE.

See also HIDETURTLE, SHOWTURTLE, and TURTLEFACTS.

Example



SHOWTURTLE (ST)

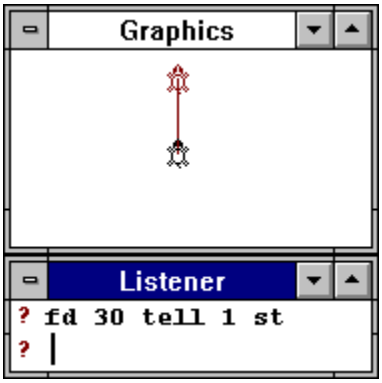
Syntax

```
SHOWTURTLE
ST
```

Explanation

`SHOWTURTLE` makes the turtle shape visible. To make the turtle invisible, use `HIDETURTLE`. See also `SHOWN?` and `TURTLEFACTS`.

Example



Turtle 0 (red) moves forward 30 turtle steps, then turtle 1 (black) is made visible.

SIN

Syntax

```
SIN number
```

Explanation

`SIN` outputs the sine of its input, which is the number of degrees in an angle. Remember that `SIN` x =opposite/hypotenuse.

See also `ARCTAN` and `COS`.

Examples

```
? SIN 30
Result: .5
? SIN 90
Result: 1
? _
```

SINGLE.STEP

Syntax

```
MAKE "SINGLE.STEP "TRUE
```

Explanation

SINGLE.STEP is a pre-defined name that allows monitoring of procedure or command line execution. SINGLE.STEP displays each step of a line or procedure before execution and pauses until you type CONTINUE or CO.

SINGLE.STEP allows the user to test the expression with different variables or perform other computations during the pause.

To return to toplevel before the entire procedure has run, type Control-G or TOPLEVEL during the pause. To turn off SINGLE.STEP, type:

```
MAKE "SINGLE.STEP "FALSE
```

See also PAUSE and TRACE.

SNAP

Syntax

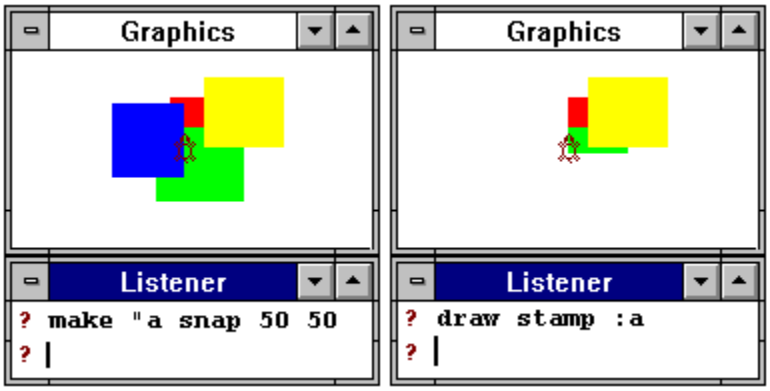
```
SNAP xcoordinate ycoordinate
```

Explanation

SNAP stores a region of the screen into a bit map. The turtle position marks the lower left corner, while the X and Y inputs describe the size of the image to be SNAPped. SNAP outputs a bit map object which can be saved, loaded or STAMPed.

See also STAMP, SAVESNAP, LOADSNAP and SNAPSIZE.

Example



SNAPSIZE

Syntax

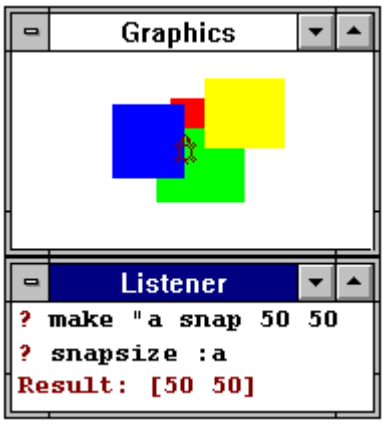
```
SNAPSIZE bitmap
```

Explanation

SNAPSIZE outputs the size of a previously SNAPped bit map as a list of two elements. The first element is the width of the bit map in screen pixels, while the second element is the height of the bitmap in screen pixels.

See also SNAP, STAMP, LOADSNAP and SAVESNAP.

Example



SPEED

Syntax

SPEED

Explanation

SPEED outputs the current speed at which the turtle(s) moves on the screen. When Logo starts, the turtle speed is 1. Use SETSPEED to change the speed.

Example

```
? SPEED
Result: 1
? _
```

SPLITSCREEN (SS)

Syntax

```
SPLITSCREEN
SS
```

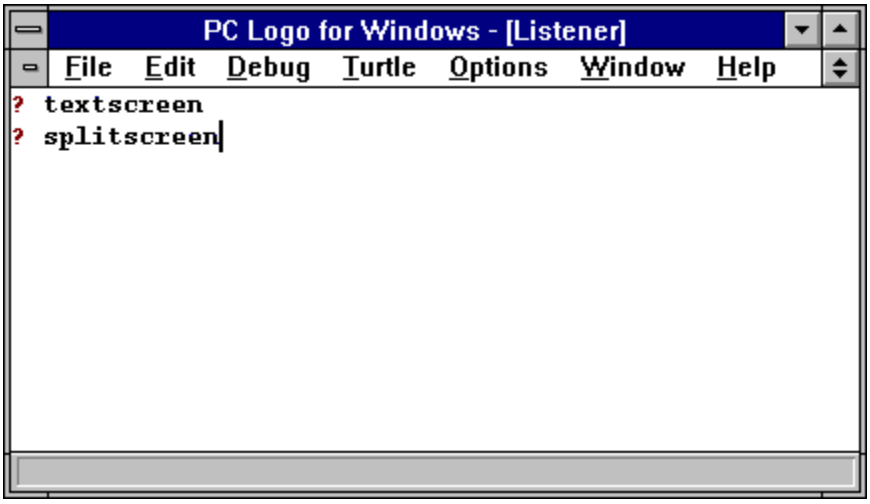
Explanation

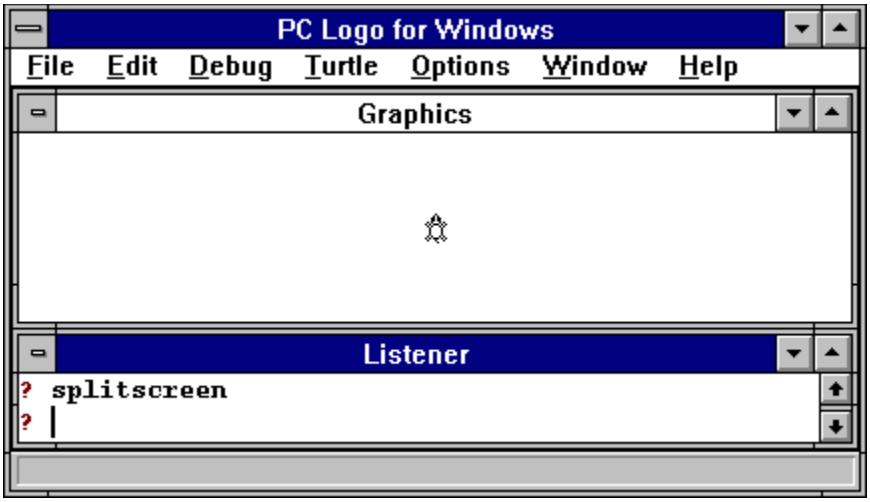
`SPLITSCREEN` restores the default window layout with the Graphics window occupying the upper two thirds of the screen and the Listener window occupying the bottom third of the screen.

The menu command **Window/Standard layout** or the `Control-L` key have the same effect as `SPLITSCREEN`.

See also `FULLSCREEN` and `TEXTSCREEN`.

Example





SQRT

Syntax

```
SQRT number
```

Explanation

`SQRT` outputs the square root of its input. The input number must be a positive number.

Examples

```
? SQRT 25
Result: 5
? SQRT 121
Result: 11
? SQRT 492
Result: 22.18
? SQRT -1
The procedure SQRT does not like -1 as input.
? _
```

STAMP

Syntax

```
STAMP bitmap
(STAMP bitmap xcoordinate ycoordinate)
```

Explanation

`STAMP` displays a bit map created by the `SNAP` command at the location(s) of the active turtle(s). The turtle location(s) becomes the lower left corner of the image. If you enclose the `STAMP` command in parentheses, you can supply a width and a height for the bit map, both in turtle steps. If you do so, the bit map is adjusted to display within the specified area.

See also `SNAP`, `SNAPSIZE`, `LOADSNAP` and `SAVESNAP`.

Example

STAMPOVAL

Syntax

```
STAMPOVAL number number
(STAMPOVAL number number "TRUE)
```

Explanation

STAMPOVAL draws an oval around the current turtle(s) with a radius of the number of turtle steps in its first input in the horizontal direction and the number of turtle steps in its second input in the vertical direction. STAMPOVAL draws a circle if the two inputs are equal.

If STAMPOVAL, its inputs, and the value TRUE are all enclosed in parentheses, the oval drawn is filled with the current pattern in the current pen color.

See also STAMPRECT.

Example

STAMPRECT

Syntax

```
STAMPRECT number number
(STAMPRECT number number "TRUE)
```

Explanation

`STAMPRECT` draws a rectangle with a vertical length of the number of turtle steps defined by its first input and a horizontal length defined by its second input. The rectangle is drawn with the current turtle position in the lower left corner. `STAMPRECT` draws a square if the two inputs are equal.

If `STAMPRECT`, its inputs, and the value `TRUE` are all enclosed in parentheses, the rectangle drawn is filled with the current pattern in the current pen color.

See also `STAMPOVAL`.

Example

STANDARD.INPUT

Syntax

```
MAKE "STANDARD.INPUT streamnumber
```

Explanation

STANDARD.INPUT is a pre-defined name which controls the source of the Logo input stream. When Logo starts up, the default value of STANDARD.INPUT is 0, which means that all input into Logo is read from the keyboard.

To change the source of the input stream to another device such as a disk file, the device must be opened or created to prepare it for input, and STANDARD.INPUT assigned a new value.

To redirect Logo's output stream, use STANDARD.OUTPUT. See also OPEN and CREATE.

Example

The following procedures print the contents of a file to the screen.

```
TO ECHO :FILE
   IF NOT FILE? :FILE (PR :FILE[DOES NOT EXIST]) STOP
   MAKE "OLDSTREAM :STANDARD.INPUT
   MAKE "STANDARD.INPUT OPEN :FILE
   ECHO.CHARS
   CLOSE :STANDARD.INPUT
   MAKE "STANDARD.INPUT :OLDSTREAM
END

TO ECHO.CHARS
   MAKE "CHAR RC
   IF :CHAR = "EOF STOP
   TYPE :CHAR
   ECHO.CHARS
END
```

STANDARD.OUTPUT

Syntax

```
MAKE "STANDARD.OUTPUT number
```

Explanation

STANDARD.OUTPUT is a pre-defined name which controls the destination of the Logo output stream. When Logo starts up, the default value of STANDARD.OUTPUT is 0, which means that output from Logo is displayed on the screen.

To change the destination of the output stream to another device such as a printer, the device must be opened to prepare it for output, and STANDARD.OUTPUT assigned a new value.

To change the source of Logo's input stream, use STANDARD.INPUT.

See also OPEN, CLOSE and CREATE.

Example

The following procedures redirect the output stream to the printer. Note that when the output stream is redirected to another device, it no longer appears on the screen.

Since a colon (:) is a delimiter, it must be quoted with \.

```
TO OUTPUT.TO.PRINTER
   MAKE "STANDARD.OUTPUT OPEN "PRN\:
END

TO OUTPUT.TO.SCREEN.AGAIN
   TEST :STANDARD.OUTPUT = 0
   IFF [CLOSE :STANDARD.OUTPUT]
   MAKE "STANDARD.OUTPUT 0
END
```

STOP

Syntax

```
STOP
```

Explanation

`STOP` makes Logo halt execution of the current procedure and return to the calling procedure. If there is no calling procedure, Logo returns to `TOPLEVEL`.

Example

```
TO GUESS
   TYPE [HOW MANY SYMPHONIES DID BEETHOVEN COMPOSE]
   MAKE "GUESS READ
   MAKE "NUMBER 9
   IF :NUMBER = :GUESS \
       [PR [THAT'S RIGHT\!]
       [PR [NOT QUITE. GUESS AGAIN.]]
   GUESS
END
```

```
? GUESS
HOW MANY SYMPHONIES DID BEETHOVEN COMPOSE? ? 3
NOT QUITE. GUESS AGAIN.
HOW MANY SYMPHONIES DID BEETHOVEN COMPOSE? ? 9
THAT'S RIGHT!
? _
```

SUBDIR

Syntax

```
SUBDIR
(SUBDIR word)
```

Explanation

`SUBDIR` outputs a list of subdirectory names on the disk in the currently selected disk drive.

If `SUBDIR` is used with an input, it outputs the subdirectory names specified by its input. A drive specifier may be used to access a disk drive which is not currently selected. A ? may be used to match a single character except a period and a * may be used to match a group of characters not including a period.

See also `DIRECTORY`.

Examples:

```
? SUBDIR
Result: [DOS WINDOWS WINLOGO]
? (DIRECTORY "W\*)
Result: [WINDOWS WINLOGO]
? _
```

SUM

Syntax

```
SUM number1 number2
(SUM number1 number2 number3 . . .)
```

Explanation

`SUM` outputs the result of adding its inputs. `SUM` expects exactly two inputs, but will accept more if it and its inputs are enclosed within parentheses.

`SUM` is equivalent to the infix operator `+`.

Examples

```
? SUM 3 6
Result: 9
? SUM 3.2 6.4
Result: 9.6
? (SUM 3.2 6.4 1)
Result: 10.6
? _
```

TAB

Syntax

```
MAKE "TAB number
```

Explanation

The system variable `TAB` sets the tab stop width used in the Editor and Listener windows. This width is preset to 5 at startup.

Example

```
? :TAB
Result: 5
? PO CIRCLE
TO CIRCLE
     REPEAT 360 [FD 1 RT 1]
END
? MAKE "TAB 10
? PO CIRCLE
TO CIRCLE
          REPEAT 360 [FD 1 RT 1]
END
? _
```

TELL

Syntax

`TELL number` or `list`

Explanation

`TELL` activates the turtles that respond to turtle commands. A single number as an argument to `TELL` activates that single turtle. A list of numbers activates all the turtles named in the list. Use `TELLALL` to activate a range of turtles.

Before turtles can be activated, they must be defined by the `SETTURTLES` command. The default number of turtles available when Logo starts is 16. (turtle 0 through turtle 15)

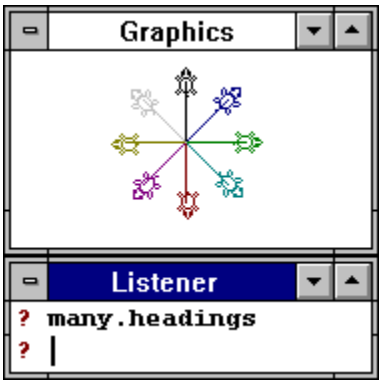
After startup or after execution of a `SETTURTLES` command, turtle 0 becomes the active turtle.

See also `ASK`, `EACH`, `TURTLES`, and `WHO`.

Example

```
TO MANY.HEADINGS
    TELL [0 1 2 3 4 5 6 7]
    PENDOWN ST
    EACH [SETH 45 * WHO SETPC WHO FD 30]
END
```

This procedure causes eight turtles to change theirs color to the same value as their turtle number. They then move apart in different directions.



TELLALL

Syntax

```
TELLALL number number
```

Explanation

TELLALL activates a range of turtles to respond to turtle commands. TELLALL takes two numbers as arguments. The first number is the number of the first turtle in the range to be activated. The second number is the last turtle in the range to be activated. Use TELL to activate a single turtle or a list of turtles.

Before turtles can be activated, they must be defined by the SETTURTLES command. The default number of turtles available when Logo starts is 16. (turtle 0 through turtle 15)

After startup or after execution of a SETTURTLES command, turtle 0 becomes the active turtle.

See also ASK, EACH, TURTLES, and WHO.

Example

```
TO MANY.HEADINGS
    TELLALL 0 7
    PENDOWN ST
    EACH [SETH 45 * WHO SETPC WHO FD 30]
END
```

This procedure causes eight turtles to change theirs color to the same value as their turtle number. They then move apart in different directions.

TEST

Syntax

```
TEST statement
```

Explanation

`TEST` determines whether its input is `TRUE` or `FALSE` and remembers it for later use in an `IFTRUE` or `IFFALSE` statement.

Example

```
TO GUESSNUM
     MAKE "NUM RANDOM 10
     PR [I'M THINKING OF A NUMBER BETWEEN 1 AND 10.]
     LABEL "LOOP
     TYPE [CAN YOU GUESS IT]
     MAKE "GUESS READ
     TEST (:NUM = :GUESS)
     IFTRUE [PR [GOOD GUESS!] STOP]
     IFFALSE [PR [NO, TRY AGAIN.]]
     GO "LOOP
END
```

```
? GUESSNUM
I'M THINKING OF A NUMBER BETWEEN 1 AND 10.
CAN YOU GUESS IT? 3
NO, TRY AGAIN.
CAN YOU GUESS IT? 7
GOOD GUESS!
? _
```

TEXT

Syntax

```
TEXT procname
```

Explanation

TEXT outputs the definition of the procedure named in its input. The form of the output is a list.

The first element of the list is any variable(s) defined in the title line of the procedure. If there are no variables, the first element is the empty list ([]). Each remaining element is a list which consists of one line of the procedure definition.

The output of TEXT is in the same form as the required input for DEFINE. See also PRINTOUT, POPS, and POTS.

Example

```
? TO PENTA :SIDE
>   SETPC 2
>   REPEAT 5 [FD :SIDE RT 72]
> END
PENTA defined.
? TEXT "PENTA
Result: [[SIDE][SETPC 2][REPEAT 5 [FD :SIDE RT 72]]]
? _
```

TEXTARRAY

Syntax

```
TEXTARRAY bytearray
```

Explanation

`TEXTARRAY` converts the contents of a bytearray into a Logo word. The bytearray must only contain readable characters. The conversion stops when Logo detects an empty byte with a zero value.

Example

```
? MAKE "TEXT BYTEARRAY 20
? ASET :TEXT 0 72
? ASET :TEXT 1 69
? ASET :TEXT 2 76
? ASET :TEXT 3 76
? ASET :TEXT 4 79
? TEXTARRAY :TEXT
Result: HELLO
? _
```

TEXTBG

Syntax

```
TEXTBG number
```

Explanation

TEXTBG changes the background color of the Listener window to the color indicated by its input. Any number from 0 through 15 is an acceptable input to TEXTBG. Inputs of 8 through 15 cause the text to blink. Control-L redisplays the Listener window in the selected background color after using the TEXTBG command.

To change the color of the text itself, use TEXTFG. See also GETATTR and SETATTR.

Example

TEXTFG

Syntax

```
TEXTFG number
```

Explanation

`TEXTFG` changes the color of the text in the Listener window to the color indicated by its input. Any number from 0 through 15 is an acceptable input to `TEXTFG`.

Note that only the foreground color of the text that Logo outputs is changed; if you work with syntax highlighting, all the special colors which highlight your text do not change.

To change the background color of the text screen, use `TEXTBG`. See also `GETATTR` and `SETATTR`.

Example

TEXTSCREEN (TS)

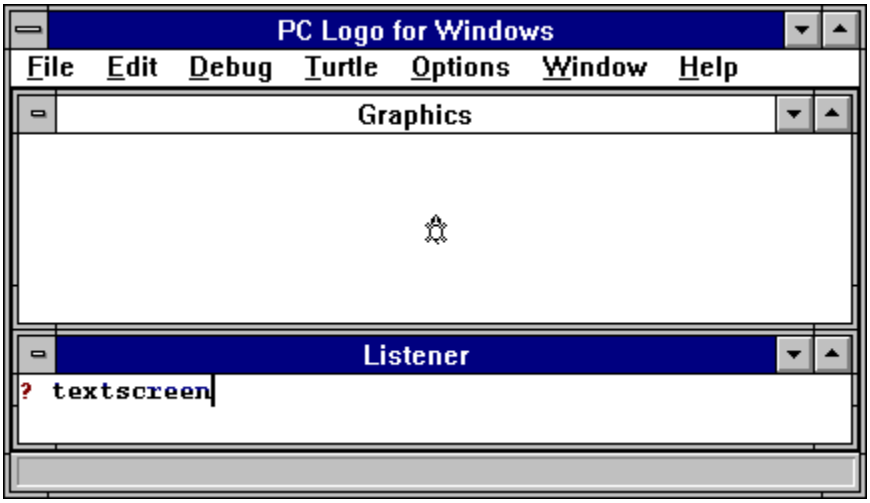
Syntax

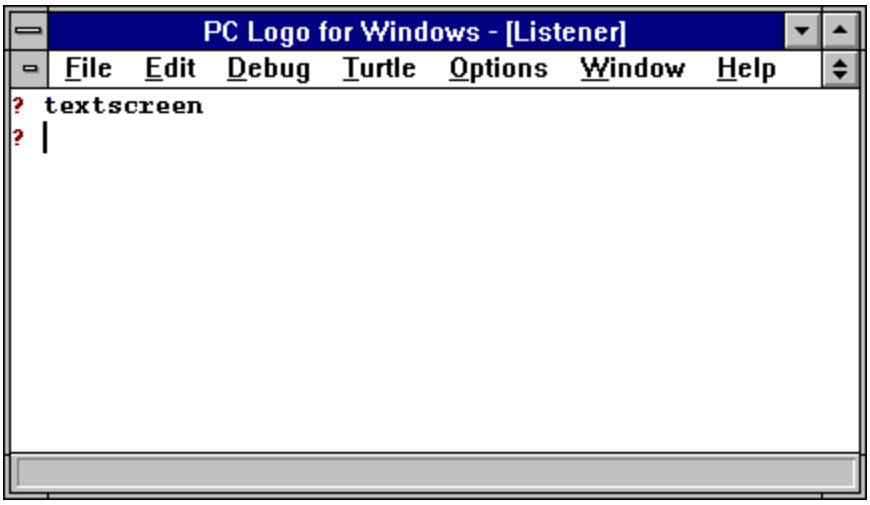
```
TEXTSCREEN
TS
```

Explanation

`TEXTSCREEN` maximizes the Listener window, thus hiding all other windows. See also `SPLITSCREEN` and `FULLSCREEN`.

Example





THEN

Syntax

```
IF conditional THEN instructionlist
```

Explanation

THEN denotes the operational clause in an IF...THEN statement. If the conditional input to IF is TRUE, then the Logo instruction list following THEN is executed. If the conditional is FALSE, then the Logo instruction list following THEN is not executed. If there is an ELSE clause, the Logo instruction list following ELSE is executed.

See also IFFALSE and IFTRUE.

Example

```
? TO ASK.OPINION
>    PRINT [DO YOU THINK LOGO IS FUN?]
>    MAKE "OPINION FIRST READLIST
>    IF :OPINION = "YES THEN PRINT[I THINK SO TOO!]
> END
ASK.OPINION defined.
? ASK.OPINION
DO YOU THINK LOGO IS FUN?
? YES
I THINK SO TOO!
? _
```

THING

Syntax

```
THING word
```

Explanation

THING outputs the value associated with the variable named in the input. THING is the Logo primitive that does the same job as : (dots). It can be used to give a variable a second level of evaluation.

Examples

```
? MAKE "COLOR "BLUE
? MAKE "BLUE "AQUAMARINE
? THING "COLOR
Result: BLUE
? THING :COLOR
Result: AQUAMARINE
? THING "BLUE
Result: AQUAMARINE
? THING :BLUE
AQUAMARINE is not a Logo name.
? _
```

THROW

Syntax

```
THROW word
```

Explanation

THROW returns control to the CATCH statement with a matching first input, or to the CATCH TRUE statement if no matching CATCH statement is found.

Examples:

The following example asks you to type a name. If you type a number instead, the program prints a message and continues.

```
TO NAMIT
   CATCH "NOTNAME [NAMIT1 STOP]
   NAMIT
END

TO NAMIT1
   PRINT [PLEASE TYPE A NAME]
   MAKE "NAME READ
   IF NUMBER? :NAME \
      [PRINT [THAT'S A NUMBER, NOT A NAME] THROW "NOTNAME]
   PRINT (SE :NAME [IS A GOOD NAME])
END
```

```
? NAMIT
PLEASE TYPE A NAME
? KURT
KURT IS A GOOD NAME
PLEASE TYPE A NAME
? 5
THAT'S A NUMBER NOT A NAME
? _
```

Type Control-G to return to toplevel.

In the following example, AVOID.INTERRUPTIONS runs the commands you type. If an error occurs, Logo prints

```
THAT'S NOT A LOGO COMMAND
```

and continues executing the procedure instead of printing the usual Logo message and terminating the procedure by returning to toplevel.

```
TO AVOID.INTERRUPTIONS
   CATCH "ERROR [AVOID.INTERRUPTIONS1]
   PRINT [THAT'S NOT A LOGO COMMAND]
   AVOID.INTERRUPTIONS
END
```

```
TO AVOID.INTERRUPTIONS1
   RUN READLIST
   AVOID.INTERRUPTIONS1
END
```

```
? AVOID.INTERRUPTIONS
? PRINT [THIS IS RIGHT]
THIS IS RIGHT
? PRINT THIS IS RIGHT
THAT'S NOT A LOGO COMMAND
? _
```

Type `TOPLEVEL` or Control-G to return to toplevel.

TIME

Syntax

TIME

Explanation

TIME outputs the current time as a list of three numbers in the form [hour minute second]. The hours are in 24-hour format. See also DATE.

Example

```
? TO SECONDS.SINCE.MIDNIGHT
>    LOCAL "T
>    MAKE "T TIME
>    OUTPUT ((FIRST :T) * 3600) + ((ITEM 2 :T) * 60) + LAST :T
> END
SECONDS.SINCE.MIDNIGHT defined.
? TIME
Result: [20 28 54]
? SECONDS.SINCE.MIDNIGHT
Result: 73735
? _
```

TIMER

Syntax

```
TIMER number
TIMER "FALSE
```

Explanation

`TIMER` starts a timer. Its input is the timer tick interval in 1/100 seconds. Each time the timer ticks, a `TIMER` event is generated. A Logo procedure tied to this `TIMER` event by `DEFEVENT`, is called at regular intervals.

If you use `FALSE` as input for the `TIMER` command, the timer stops. The timer also stops if you press `Control-G`. The built-in event handling procedure for the `BREAK` event stops the timer before returning to toplevel.

The output is the previous value given as input for the `TIMER` command.

Example

In the following example, the procedure `TICK` is called once every second.

```
? TO TICK
>    PRINT [TICK TICK TICK]
> END
TICK defined.
? DEFEVENT "TIMER "TICK
? TIMER 100
Result: FALSE
? TICK TICK TICK
TICK TICK TICK
TICK TICK TICK
TICK TICK TICK
```

TO

Syntax

```
TO procname
```

Explanation

TO is the first word of a procedure definition. When typed at toplevel, TO allows you to write a new procedure without entering the editor. When you type TO and a procedure name, the ? prompt will change to a special prompt, >, to let you know you are no longer at toplevel. This special prompt is stored in the system variable :TO and can be changed.

Type in the procedure. Before you press the Enter key after each line, check for typing errors. Unlike the screen editor, where you can move the cursor through the entire procedure to make corrections, TO allows only line editing commands. Once the Enter key has been pressed, no changes can be made in that line before defining the procedure.

Type END on a separate line and press the Enter key to inform Logo that your procedure is ready to be defined. Once the procedure has been defined, you can use it in the same way as a procedure defined in the editor. The procedure can be executed at toplevel or brought into the editor for further changes.

In the editor, TO procedurename and END are also necessary to indicate to Logo the beginning and end of a procedure.

Example

```
? TO SAY.HELLO :NAME
> (PRINT "HELLO :NAME)
> END
SAY.HELLO defined
? SAY.HELLO "BILL
HELLO BILL
? _
```

TONE

Syntax

```
TONE frequency duration
```

Explanation

`TONE` plays a musical tone whose pitch in Hertz is the first input. Each unit of the second input, duration, is 1/18.2 of a second.

The following table correlates tones with their frequency in Hertz. 440.00 A is the tuning note; 261.63 C is the same pitch as middle C on a piano. Higher or lower notes can be approximated by either doubling or halving the frequency of the corresponding note in the nearest octave.

Note	Frequency	Note	Frequency	Note	Frequency
C	130.81	C	261.63	C	523.25
C#	138.59	C#	277.18	C#	554.37
D	146.83	D	293.66	D	587.33
D#	155.56	D#	311.13	D#	622.25
E	164.81	E	329.63	E	659.26
F	174.61	F	349.23	F	698.46
F#	185.00	F#	370.00	F#	739.99
G	196.00	G	392.00	G	783.99
G#	207.65	G#	415.30	G#	830.61
A	220.00	A	440.00	A	880.00
A#	233.08	A#	466.18	A#	932.33
B	246.94	B	493.88	B	987.77
C	1046.50				

Example

```
? TONE 440 18.2
```

produces the note A for one second.

```
TO TRILL
   REPEAT 10 [TONE 440 1 TONE 493.88 1]
END
```

produces a short trill.

TOPLEVEL

Syntax

```
TOPLEVEL
```

Explanation

`TOPLEVEL` stops execution of a procedure and returns Logo to toplevel, the command mode. `TOPLEVEL` is the primitive to use in a procedure to perform the same function that Control-G does from the keyboard.

Note that `TOPLEVEL` is different from `STOP` in that control is not returned to any calling procedure.

Example

The procedure below can be used as a subprocedure of a game program. If the player wants to end the game, the procedure returns to toplevel.

```
TO ENDALL
   PR [DO YOU WISH TO CONTINUE?]
   PR [PLEASE TYPE YES OR NO]
   MAKE "ANSWER READLIST
   IF :ANSWER = [YES] THEN GAME
   IF :ANSWER = [NO] THEN PR[THAT'S ALL FOR THIS GAME.] TOPLEVEL ENDALL
END
```

TOWARDS

Syntax

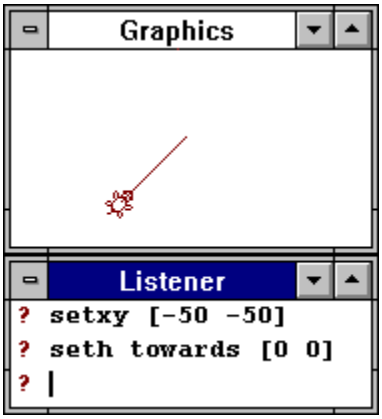
```
TOWARDS [xcoordinate ycoordinate]
```

Explanation

`TOWARDS` outputs a number which is the heading necessary for the turtle to rotate from its present position to the new position indicated by its input list.

`SETHEADING TOWARDS [xcoordinate ycoordinate]` heads the turtle in the direction of its input list.

Example



TRACE

Syntax

```
MAKE "TRACE "TRUE
```

Explanation

TRACE is a pre-defined name that allows monitoring of procedure or command line execution. TRACE displays each step of a procedure in the trace window without pausing as it is run. TRACE displays the current step being evaluated as well as the procedure name, if any, line, and output.

There are three TRACE levels. Level 1 lists entry and exit from user functions; level 2 lists entry and exit from user and system functions other than arithmetic and relational comparisons; level 3 lists all evaluation calls, user and system functions including arithmetic and relational comparisons.

Level 2 is the default level of TRACE.

MAKE "TRACE.LEVEL integer from one to three sets the active level when tracing is enabled.

To pause during a trace, use Control-Z. To abort a lengthy TRACE, use Control-G. To turn off TRACE, type:

```
MAKE "TRACE "FALSE
```

See also SINGLE.STEP, TRON and TROFF.

Example

TRACE.LEVEL

Syntax

MAKE "TRACE.LEVEL integer from 1 to 3

Explanation

The value of TRACE.LEVEL controls the level of detail at which TRACE operates. There are three TRACE levels. Level 1 lists entry and exit from user functions. Level 2 lists entry and exit from user and system functions other than arithmetic and relational comparisons. Level 3 lists all evaluation calls, user and system functions including arithmetic and relational comparisons.

See also SINGLE.STEP.

TRACED

Syntax

TRACED

Explanation

TRACED outputs a list of all procedures currently being traced with the TRON command.

Example

```
? TRON "CIRCLE
? TRACED
Result: [CIRCLE]
? _
```

TROFF

Syntax

```
TROFF name
(TROFF name name ...)
(TROFF)
```

Explanation

`TROFF` turns off tracing for the names given as its input. If `TROFF` is used without any inputs, tracing will be turned off for all names.

Example

```
? TRON "CIRCLE
? TRACED
Result: [CIRCLE]
? (TROFF)
? TRACED
Result: []
? _
```

TRON

Syntax

```
TRON name
(TRON name name ...)
(TRON)
```

Explanation

TRON turns on tracing for the selected names. If tracing is enabled for a particular name, all calls to a procedure with that name are displayed in the trace window. If the name is a built-in command, only its input and output are displayed. If the name is a user-defined procedure, each step of the procedure is displayed. Furthermore, all assignments of a value to the name is displayed in the Names window, and all properties are displayed in the Properties window.

If TRON is used without any inputs, tracing is enabled for every Logo name including built-in names nd user-defined names.

Use TROFF to disable tracing for any name.

Use the menu commands **Debug/Procedures...**, **Debug/Names...** and **Debug/Properties...** to selectively enable tracing.

Example

```
? TRON "CIRCLE
? TRACED
Result: [CIRCLE]
? (TROFF)
? TRACED
Result: []
? _
```

TURTLEFACTS (TF)

Syntax

```
TURTLEFACTS
TF
```

Explanation

TURTLEFACTS outputs a list of the settings of the first of the currently active turtles. The list holds the following elements:

1. The X coordinate.
2. The Y coordinate.
3. The heading in degrees.
4. The pen mode (PENDOWN, PENUP, PENERASE, PENREVERSE).
5. The pen color.
6. TRUE if the turtle is to be made visible, FALSE otherwise.
7. The line width.
8. The TURTLETEXT font name.
9. The TURTLETEXT font size.
10. The TURTLETEXT font attributes.

SETTURTLEFACTS can be used to change the turtle settings using the same list.

Example

```
? DRAW
? TURTLEFACTS
Result: [0 0 90 PENDOWN 4 TRUE 1 SYSTEM 10 1]
? _
```

TURTLES

Syntax

```
TURTLES
```

Explanation

The `TURTLES` command outputs the total number of available turtles.

Example

```
? TURTLES
Result: 16
? _
```

TURTLETEXT (TT)

Syntax

`TURTLETEXT word` or `list`
`TT word` or `list`

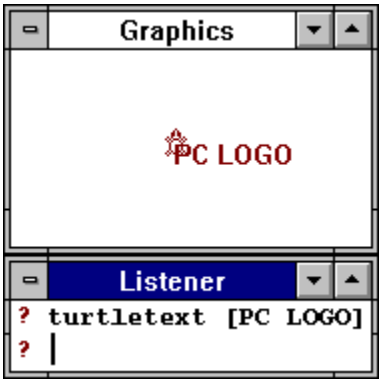
Explanation

`TURTLETEXT` prints its input on the Graphics window at the position of the current turtle(s).

`TURTLETEXT` prints in the pen color of the current turtle(s). `TURTLETEXT` has no effect if the turtle's pen is up.

See also `FONT`, `SETFONT` and `FONTS`.

Example



TYPE

Syntax

```
TYPE object
(TYPE object1 object2 . . .)
```

Explanation

TYPE prints its inputs to the output stream without inserting a carriage return. If the input is a list, TYPE removes the brackets. Normally, TYPE prints its inputs on the screen, and the prompt appears after the last character printed.

See also PRINT and SHOW.

Examples

```
? TYPE "HELLO
HELLO? TYPE [HI HOW ARE YOU?]
HI HOW ARE YOU? (TYPE "TWO "WORDS)
TWO WORDS? _
```

UNBURY

Syntax

UNBURY word or list

Explanation

UNBURY returns the object(s) in its argument to the general Logo workspace. UNBURY operates on procedures, names, and/or property lists previously buried with the BURY, BURYNAME, BURYPROC, or BURYPROP commands.

See also BURIEDNAMES, BURIEDPROCS, BURIEDPROPS, UNBURYALL, UNBURYNAME, UNBURYPROC, and UNBURYPROP.

Example:

```
? MAKE "A 123
? MAKE "B 456
? BURY [A B]
? PONS
? UNBURY "A
? PONS
A is 123
? _
```

UNBURYALL

Syntax

UNBURYALL

Explanation

UNBURYALL returns all previously buried Logo objects, including procedures, names, and property lists to the general Logo workspace. UNBURY operates on procedures, names, and/or property lists previously buried with the BURY, BURYNAME, BURYPROC, or BURYPROP commands.

See also BURIEDNAMES, BURIEDPROCS, BURIEDPROPS, UNBURY, UNBURYNAME, UNBURYPROC, and UNBURYPROP.

Example

```
? MAKE "A 123
? MAKE "B 456
? TO SAY.HELLO
> PR "HELLO
> END
SAY.HELLO defined.
? BURYALL
? PONS
? POTS
? UNBURYALL
? PONS
A is 123
B is 456
? POTS
TO SAY.HELLO
? _
```

UNBURYNAME

Syntax

`UNBURYNAME word` or `list`

Explanation

`UNBURYNAME` returns all previously buried Logo names to the general Logo workspace. `UNBURYNAME` operates on names previously buried with the `BURY`, `BURYALL`, or `BURYNAME` commands.

See also `BURIEDNAMES`, `BURIEDPROCS`, `BURIEDPROPS`, `BURYPROC`, `BURYPROP`, `UNBURY`, `UNBURYALL`, `UNBURYPROC`, and `UNBURYPROP`.

Example

```
? MAKE "A 123
? MAKE "B 456
? BURYNAME "A
? PONS
B is 456
? UNBURYNAME "A
? PONS
A is 123
B is 456
? _
```

UNBURYPROC

Syntax

UNBURYPROC word or list

Explanation

UNBURYPROC returns all previously buried Logo procedures to the general Logo workspace. UNBURYPROC operates on procedures previously buried with the BURY, BURYALL, or BURYPROC commands.

See also BURIEDNAMES, BURIEDPROCS, BURIEDPROPS, BURYNAME BURYPROP, UNBURY, UNBURYALL, UNBURYNAME, UNBURYPROC, and UNBURYPROP.

Example

```
? TO SAY.HELLO
> PR "HELLO
> END
SAY.HELLO defined.
? BURYPROC "SAY.HELLO
? POTS
? UNBURYPROC "SAY.HELLO
? POTS
TO SAY.HELLO
? _
```

UNBURYPROP

Syntax

`UNBURYPROP word` or `list`

Explanation

`UNBURYPROP` returns all previously buried Logo property lists to the general Logo workspace. `UNBURYPROP` operates on procedures previously buried with the `BURY`, `BURYALL`, or `BURYPROP` commands.

See also `BURIEDNAMES`, `BURIEDPROCS`, `BURIEDPROPS`, `BURYNAME` `BURYPROC`, `UNBURY`, `UNBURYALL`, `UNBURYNAME`, `UNBURYPROC`, and `UNBURYPROP`.

Example

```
? PPROP "CAPITAL "MONTANA "HELENA
? BURYPROP "CAPITAL
? PLIST "CAPITAL
Result: []
? UNBURYPROP "CAPITAL
? PLIST "CAPITAL
Result: [MONTANA HELENA]
? _
```

UNGETBYTE

Syntax

```
UNGETBYTE number
```

Explanation

UNGETBYTE pushes the ASCII character corresponding to its input onto the input stream so that the next character input primitive will pick up the character.

Only one character can be pushed back at a time, so the character must be removed from the input stream before UNGETBYTE can be used again.

See also GETBYTE, GETBYTE.NO.ECHO, PEEKBYTE, and PUTBYTE.

Example

The following procedure performs the same function as the SKIP.EMPTY.LINES example in the PEEKBYTE entry. The procedure removes empty lines from the input stream and sends all other information to the output stream.

```
TO SKIP.EMPTY.LINES
   MAKE "FIRST.CHAR.ON.LINE GETBYTE
   TEST :FIRST.CHAR.ON.LINE = 13
   IFF [UNGETBYTE :FIRST.CHAR.ON.LINE PR READLINE] SKIP.EMPTY.LINES
END
```

VERSION (VER)

Syntax

```
VERSION
VER
```

Explanation

`VERSION` outputs information about the version of PC Logo for Windows.

Example

```
? VERSION
Result: 1.00.00 01May93
? _
```

WAIT

Syntax

```
WAIT number
```

Explanation

`WAIT` inserts a pause before the next instruction is run. The length of the pause is the input to `WAIT` times 1/100 of a second.

Example

The following procedure will print `HOORAY` 20 times, pausing for a second between each `HOORAY`.

```
TO CHEER
   REPEAT 20 [PRINT "HOORAY WAIT 100]
END
```

WHILE

Syntax

```
WHILE test list run list
```

Explanation

WHILE evaluates its first input and runs the Logo command(s) in its second input if the value of the first input is TRUE. WHILE will continue this process until the value of the first input is FALSE.

See also FOR, IF, IFTRUE, and IFFALSE.

Example

```
? MAKE "X 1
? WHILE [:X < 5] [PRINT :X MAKE "X :X + 1]
1
2
3
4
? _
```

WHO

Syntax

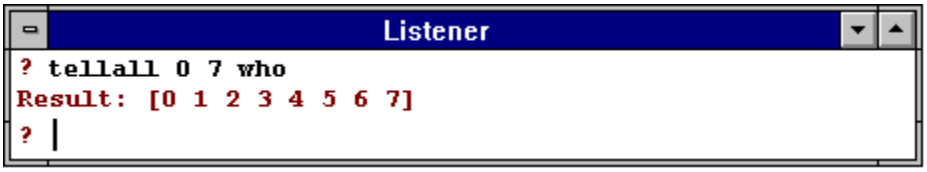
```
WHO
```

Explanation

`WHO` outputs the list of currently active turtles which are determined by the `TELL` command.

See also `ASK`, `EACH`, `SETTURTLES`, `TELLALL` and `TURTLES`.

Example



WIDTH

Syntax

`WIDTH`

Explanation

`WIDTH` outputs the pen width of the first active turtle. The pen width can be a number between 1 and 999. `SETWIDTH` sets the pen width.

If more than one turtle is active, the pen width of these turtles may be interrogated with `WIDTH` and the `ASK` or `EACH` commands.

Example

```
? DRAW
? WIDTH
Result: 1
? TELL [0 1 2 3 4]
? SETWIDTH 5
? EACH [PRINT WIDTH]
5
5
5
5
5
? _
```

WINDOW

Syntax

```
WINDOW
```

Explanation

`WINDOW` removes the boundaries from the turtle's field of movement. If the turtle moves beyond the borders of the graphics window, it continues to move, but cannot be seen. The graphics window becomes a small window overlooking the plane on which the turtle can travel.

The range of movement for the turtle when `WINDOW` is in use is -32,768 to 32,767.

See also `FENCE` and `WRAP`.

Example

The turtle has moved off the window.

WINSIZE

Syntax

```
WINSIZE
(WINSIZE number)
```

Explanation

WINSIZE outputs the size of the Graphics window as a list with two elements. The first element is the width in pixels and the second is the height in pixels.

An optional input can be supplied to WINSIZE if WINSIZE and its inputs are enclosed in parentheses. This input identifies the window by the number listed in the Window menu.

See also SETWINSIZE, EXTENT and SETEXTENT.

Example

```
? WINSIZE
Result: [320 198]
? _
```

WINVER

Syntax

```
WINVER
```

Explanation

`WINVER` outputs the version of the Windows operating environment as a list with two elements. The first element is the version number and the second element is the revision number.

Example

```
? WINVER
Result: [3 10]
? _
```

WORD

Syntax

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

Explanation

WORD outputs a word made up of its inputs. WORD expects two inputs, but will accept more if it and its inputs are enclosed in parentheses.

See also SENTENCE and WORD?.

Examples

```
? WORD "RI "BALD
Result: RIBALD
? WORD 34 56
Result: 3456
? (WORD BF "WAGE FIRST "WAGE "AY)
Result: AGEWAY
? _
```

WORD?

Syntax

```
WORD? object
```

Explanation

`WORD?` outputs `TRUE` if its input is a word; otherwise, it outputs `FALSE`.

See also `LIST?`, `NAME`, `NUMBER?` and `WORD`.

Examples

```
? WORD? "HOUSE
Result: TRUE
? WORD? [HOUSE]
Result: FALSE
? WORD? 1234
Result: TRUE
? WORD? [G B H]
Result: FALSE
? WORD? ITEM 2 [G B H]
Result: TRUE
? _
```

WRAP

Syntax

```
WRAP
```

Explanation

`WRAP` makes the turtle remain inside the graphics window no matter how large a movement command is given. Any time the turtle moves off the window borders, it wraps around the window and reappears on the opposite edge.

When Logo starts up, the default window state is `WRAP`.

See also `FENCE` and `WINDOW`.

Example

The turtle wraps around every time it reaches the edges of the graphics window.

XCOR

Syntax

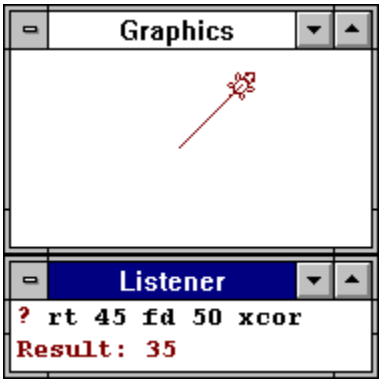
`XCOR`

Explanation

`XCOR` outputs the X coordinate of the turtle's position on the screen.

See also `GETXY`, `SETX`, `SETXY`, and `YCOR`.

Example



YCOR

Syntax

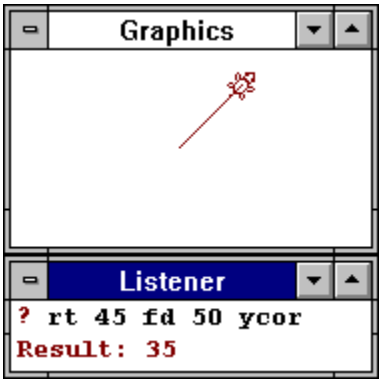
```
YCOR
```

Explanation

`YCOR` outputs the Y coordinate of the turtle's position on the screen.

See also `GETXY`, `SETXY`, `SETY`, and `XCOR`.

Example



ALL

`ALL` is a special word used as an input for `EDIT`, `ERASE`, and `PRINTOUT` which includes all procedures, names, and property lists currently in Logo's workspace.

BREAK

BREAK is a reserved word, used as an input for the DEFEVENT command. It allows you to define a procedure which will be called whenever the Control-G key is pressed. The built-in Logo procedure stops all background procedures as well as the timer event procedure and returns to toplevel.

You should redefine the BREAK procedure with great care. If you do not issue a TOPLEVEL command in your procedure, your Logo programs cannot be interrupted.

Example

The following procedure may act as a replacement procedure for the built-in break procedure. Before returning to toplevel, however, you are asked whether you want to break at all. Note that all background tasks are halted with the HALT command.

```
TO MY.OWN.BREAK
   LOCAL "ANSWER
   PR "BREAK!
   TYPE [RETURN TO TOPLEVEL? |(Y/N)|]
   MAKE "ANSWER RC
   (PR)
   IF :ANSWER = "Y THEN (HALT) IGNORE TIMER "FALSE TOPLEVEL
END
```

CONSTANTS

`CONSTANTS` is a reserved word and can be used together with the commands `PRINTOUT`, `EDIT` or `ERASE`. It allows you to access all defined constants at once.

EOF

EOF is a pre-defined name indicating that the end of file has been reached on the current input stream. All stream input primitives, such as READ, READCHAR, READLINE, READLIST, or READQUOTE, output :EOF if the stream from which they are reading reaches the end of a file. If an attempt is made to read characters past the end of a file, an error message displays. The default value of "EOF is EOF.

Example

The procedure below reads every character from a file the end of file marker is reached. Thus, the file is checked for bad sectors or other hardware problems.

```
TO SCAN :FILE
   IF NOT FILE? :FILE THEN (PRINT :FILE [DOES NOT EXIST]) STOP
   MAKE "OLDSTREAM :STANDARD.INPUT
   MAKE "STANDARD.INPUT OPEN :FILE
   SCAN.CHARS
   CLOSE :STANDARD.INPUT
   MAKE "STANDARD.INPUT :OLDSTREAM
END

TO SCAN.CHARS
   IF EQUAL? RC :EOF THEN STOP
   SCAN.CHARS
END
```

FALSE

`FALSE` object used as input to `IF`, `AND`, `NOT`, `OR`, `TEST` and many other commands. Its value is `FALSE`.

NAMES

`NAMES` is a special word used as an input for `EDIT`, `ERASE`, and `PRINTOUT` which includes all defined variables currently in Logo's workspace.

PROPERTIES

`PROPERTIES` is a special Logo word used as an input to `EDIT`, `ERASE`, and `PRINTOUT`. `PRINTOUT PROPERTIES` lists all currently defined property lists in the Logo workspace.

PROCEDURES

`PROCEDURES` is a special word used as an input for `EDIT`, `ERASE`, and `PRINTOUT` which includes all user-defined procedures currently in Logo workspace.

TITLES

`TITLES` is a special word used as an input for the `PRINTOUT` command. It lists all procedure titles. The equivalent of `PRINTOUT TITLES` is `POTS`.

TRUE

`TRUE` object used as input to `IF`, `AND`, `NOT`, `OR`, `TEST` and many other commands. Its value is `TRUE`.