HyperLogo™

A Scripting Language for HyperStudio

Mike Westerfield

Byte Works®, Inc.
4700 Irving Blvd. NW, Suite 207
Albuquerque, NM 87114
(505) 898-8183

Master Set 1.0.0.0

Table of Contents

Chapter 1 – Getting the Most from HyperLogo 1
  What's in This Book 1
  Backing Up HyperLogo 2
  Installing HyperLogo 3
  Giving Stacks to Friends and Customers 4
    Distributing HyperLogo With HyperStudio 5
    Distributing HyperLogo Without HyperStudio 5
    Copyright Information 5
  Getting Answers to Questions 6
  Finding Out More About Logo 6

Chapter 2 – Getting Started with HyperLogo 9
  Your First Script 9
    Changing A Script 11
  Using HyperLogo to Learn a Little Logo 11
    Text Windows 11
    Turtle Windows 12
  Simple Logo Commands 13
    Introducing the Logo Turtle... 13
    Drawing Lines with the Turtle 13
    Clearing the Screen 14
    Turning the Turtle 14
    Creating Procedures with TO 15
  The Logo Workspace 17
    Listing Procedures in the Workspace 17
    Editing Procedures with Edit Windows 18
    Using TO in the Script Window 19
    Removing Procedures from the Workspace 20
  Variables 20
    Creating Variables with MAKE 20
    Quotes and Colons and Nothing, Oh My! 21
    Printing a Variable 22
    Logo Numbers Come in Two Flavors 22
    Word Variables 22
    Lists 23
    Logo Variables Do It All 23
    What About Arrays? 25
    Variables as Parameters 25
  Math With Logo 27
    Addition, Subtraction, Multiplication and Division 27
    Negative Numbers 29
    Functions 30
    Reals and Integers 30

Chapter 3 – 3D Pictures with HyperLogo 31
- The Cubes Program 31
- How the 3D Display Works 32
  - Realistic 3D Pictures 33
- The 3D Turtle 34
  - Rotating Out of the Screen 34
  - Rolling the Turtle 34
  - Some 3D Shapes to Play With 35
- 3D Without the Glasses (The Chemistry Card) 37
  - Trying the Chemistry Card 38
  - How the Program Works 39

Chapter 4 – Making Movies with HyperLogo 43
- Making Movies is Easy 43
  - A Finished Movie 43
  - Creating Your First Movie 43
  - Playing the Movie 45
  - Movie Options 46
- Playing a Movie On a Card 47
- Movies and Other Programs 47
- Movie Samples 47
  - The Cube Movie 48
  - Spinning Octahedrons 48
  - Spinning Molecules 49

Chapter 5 – Talking Logo 51
- What You Need 51
- A Quick Demonstration 51
  - Why Not Use a Recording? 51
- Making Logo Talk 52
  - Teaching Logo to Pronounce Words 52
  - Male and Female Voices 52
  - Speed, Volume and Pitch 53

Chapter 6 – Controlling HyperStudio from HyperLogo 55
- How HyperLogo and HyperStudio Work Together 55
- Using READWORD For Text Input 55
  - Changing the Prompt 55
  - Changing the Dialog's Position and Size 56
  - An Example of READWORD in Action 56
- Reading Text from a Text Item 57
  - The Callback Library 58

Chapter 7 – Desktop Interface Reference 61
- Apple Menu 61

About HyperLogo... 61
Desk Accessories 61
File Menu 61
New Text Window 61
New Edit Window 62
Script Window 62
New Turtle Window 62
New Movie Window 63
Open... 64
Close 64
Save 65
Save As... 65
Page Setup... 65
Print... 65
Quit 65
Edit 66
Undo 66
Cut 66
Copy 66
Paste 66
Clear 67
Select All 67
Movie 67
Add Frame After 67
Insert Frame Before 67
Delete Frame 67
Movie Options... 68
Windows 69

Chapter 8 – HyperLogo Language Reference 71
Command Descriptions 71
Finding Commands 71
Procedures 72
COPYDEF 72
DEFINE 72
DEFINEDP 73
PRIMITIVEP 73
TEXT 73
TO 74
Variables 75
LOCAL 75
MAKE 76
NAME 76
NAMEP 76
THING 76

Words and Lists 77
Numbers and Words 77
ASCII 78
BEFOREP 78
BUTFIRST 78
BUTLAST 79
CHAR 79
COUNT 80
EMPTYP 80
EQUALP 80
FIRST 81
FLOATP 81
FPUT 81
INTEGERP 82
ITEM 82
LAST 82
LIST 83
LISTP 83
LOWERCASE 83
LPUT 84
MEMBER 84
MEMBERP 84
NUMBERP 85
PARSE 85
SENTENCE 85
UPPERCASE 86
WORD 86
WORDP 87
Property Lists 87
GPROP 88
PLIST 88
PPROP 88
REMPROP 89
SETPLIST 89
Numbers and Arithmetic 89
Expressions 90
Special Rules for / 91
Special Rules for - 92
ABS 93
ARCCOS 93
ARCSIN 93
ARCTAN 94
ARCTAN2 94
COS 94
DIFFERENCE 95

EXP	95
FLOAT	95
FORM	96
INT	96
INTQUOTIENT	97
LN	97
POWER	97
PRODUCT	97
QUOTIENT	98
RANDOM	98
REMAINDER	98
RERANDOM	99
ROUND	99
SIN	99
SQRT	100
SUM	100
TAN	100
Flow of Control	101
CATCH	101
DOUNTIL	102
ERROR	102
GO	103
IF	103
IFFALSE	104
IFTRUE	104
LABEL	105
OUTPUT	105
REPEAT	105
RUN	106
STOP	106
TEST	106
THROW	107
TOPLEVEL	107
WAIT	107
WHILE	108
Logical Operators	108
AND	108
NOT	109
OR	109
Input and Output	109
BUTTONP	109
KEYP	110
MOUSE	110
PRINT	110
READCHAR	111

READCHARS 111
READLIST 111
READWORD 112
SETRWPROMPT 112
SETRWRECT 112
SHOW 113
TEXTIO 113
TOOT 113
TURTLEIO 114
TYPE 115
Disk Commands 116
File Names 116
ALLOPEN 117
CATALOG 117
CLOSE 118
CLOSEALL 118
CREATEFOLDER 119
DIR 119
ERASEFILE 119
FILELEN 119
FILEP 120
LOAD 120
ONLINE 120
OPEN 121
POFILE 121
PREFIX 121
READER 122
READPOS 122
RENAME 122
SAVE 123
SETPREFIX 123
SETREAD 124
SETREADPOS 124
SETWRITE 124
SETWRITEPOS 125
WRITEPOS 125
WRITER 125
Turtle Graphics 126
Turtle Positions 126
Turtle Heading 126
3D Turtle 127
Turtle Colors in 2D 128
Turtle Colors in 3D 129
BACK 129
BACKGROUND 129

CLEAN 130
CLEARSCREEN 130
CLEARSCREEN3D 130
DOT 131
DOTP 131
FENCE 132
FILL 132
FORWARD 133
HEADING 133
HIDETURTLE 134
HOME 134
LEFT 134
PEN 135
PENCOLOR 135
PENDOWN 135
PENERASE 136
PENREVERSE 136
PENUP 136
POS 137
RIGHT 137
ROLLLEFT 137
ROLLRIGHT 138
ROTATEIN 138
ROTATEOUT 139
SCRUNCH 139
SETBG 139
SETHEADING 140
SETHEADING3D 140
SETPC 140
SETPENSIZE 141
SETPOS 141
SETSCRUNCH 141
SETX 142
SETY 142
SETZ 142
SHOWNP 143
SHOWTURTLE 143
SHOWTURTLE3D 144
TOWARDS 144
WINDOW 144
WRAP 145
XCOR 145
YCOR 145
ZCOR 146

Other Drawing Commands 146
Rectangles 146
The Rectangle-based Drawing Commands 147
ERASEARC 149
ERASEOVAL 149
ERASERECT 149
ERASERRECT 149
FRAMEARC 149
FRAMEOVAL 150
FRAMERECT 150
FRAMERRECT 150
INVERTARC 150
INVERTOVAL 150
INVERTRECT 151
INVERTRRECT 151
PAINTARC 151
PAINTOVAL 151
PAINTRECT 152
PAINTRRECT 152
Movies 152
ADDFRAME 152
CLOSEMOVIE 152
DELETEFRAME 153
INSERTFRAME 153
LASTFRAME 154
NEXTFRAME 154
OPENMOVIE 154
PLAY 155
Fonts 156
GETFONTINFO 156
SETBACKCOLOR 156
SETFONTFAMILY 157
SETFONTSIZE 157
SETFONTSTYLE 158
SETFORECOLOR 159
TEXTWIDTH 159
Speech 159
DICT 160
ERDICT 160
PHONETIC 161
PITCH 161
SAY 161
SPEED 162
TONE 162
VOICE 162

VOLUME 163

Workspace Management 163

BURY 163

BURYALL 164

BURYNAME 164

BURYPLIST 164

EDIT 165

EDITS 165

ERALL 165

ERASE 166

ERN 166

ERNS 166

ERPROPS 167

ERPS 167

NODES 167

PO 168

POALL 168

PON 168

PONS 169

POPS 169

POT 169

POTS 170

PPS 170

RECYCLE 170

UNBURY 171

UNBURYALL 171

UNBURYNAME 171

UNBURYPLIST 172

Callbacks 172

CALLBACK 172

CBDISPOSEPARAMH 173

CBGETPARAMINT 173

CBGETPARAMPOINT 173

CBGETPARAMRECT 173

CBGETPARAMSTR 174

CBGETPARAMSTRH 174

CBSETCMD 174

CBSETPARAMINT 174

CBSETPARAMPOINT 175

CBSETPARAMRECT 175

CBSETPARAMSTR 175

CBSETPARAMSTRH 175

Chapter 9 – The CallBack Library 177
Callbacks 177
CALLNBA 177
DRAWTOOFFSCREEN 177
DRAWTOSCREEN 178
FINDTEXT 178
GETCURRENTSCORE 178
GETFIELDTEXT 178
HIDEITEM 179
MOVENEXT 179
MOVEPREV 179
MOVETOCARD 179
MOVETOFIRST 180
MOVETOLAST 180
REDRAWCARD 180
SETBKGDDIRTY 180
SETFIELDTEXT 180
SETITEMRECT 181
SHOWITEM 181

Appendix A – Logo Command Summary 183

Index 189

Chapter 1 – Getting the Most from HyperLogo

What's in This Book

Like most books that come with a program, this one serves three purposes.

- This book shows you how to install HyperLogo.
- This book shows you how to use HyperLogo.
- This book includes a reference section that is a catalog of all of the commands in HyperLogo.

This section gives you a quick overview of the book, as well as a detailed overview of this chapter. If you read this section carefully, you'll learn what you can safely skip, and what you really need to read. If you're an experienced computer user, especially if you already know Logo, you'll end up skipping a lot.

This book is divided into three major sections.

- This chapter tells you how to install 3D Logo, where to go to find others who are using 3D Logo, and how to get help if you need it.
- Chapters 2 through 6 are a tutorial introduction to the Logo language in general, and HyperLogo in particular. Learning by example and exploration, you'll write real, working Logo programs. You'll learn how the Logo language works, and how you can use it to draw pictures, create 3D images, and even manipulate HyperStudio.
- Chapters 7, 8 and 9 are the reference manual for HyperLogo. These chapters are a dictionary of the commands you can use in HyperLogo, many of which are not used in the introductory chapters. There are sample subroutines, programs, or short commands you can type that show how each command is used.

This chapter helps you set up HyperLogo. You'll learn how to install HyperLogo, how to back up the disks, what to do if you have questions or problems, and where to go to find out more about Logo. I'd suggest skimming most of the chapter. Here's a quick synopsis so you know what you can skip and what you need to read:

Backing Up HyperLogo
If you're an experienced computer user, you already know how to copy floppy disks to make a backup, and even why you should make a backup. If not, be sure you read this section carefully.

Installing HyperLogo
Before you can use HyperLogo, you need to install it in HyperStudio. This section tells you how.

Giving Stacks to Friends and Customers

HyperLogo is a copyrighted program, so you can't just give it away. On the other hand, you'll want to show others your stacks, and maybe even sell them. This section tells you how to do that legally, and it doesn't even cost you anything!

Getting Answers to Questions

If you have questions or problems, you may want to talk with us or to other people who are using HyperLogo. This section tells you how to get help and where to swap ideas.

Finding Out More About Logo

Logo is a popular language, but most bookstores don't carry Logo books. This section lists a few, and tells you how to find more.

Backing Up HyperLogo

HyperLogo comes on one floppy disk. Floppy disks are a very reliable way to store information, but accidents do happen. If your floppy disk gets too hot, or gets too close to a magnet, the program might be partially erased – and if that happens, it needs to be re-recorded. It's also possible you could loose a disk.

Of course, if you need to re-record a disk, you need to have more than one copy. That's what backups are for.

We suggest making one backup copy of your floppy disks. You should also have a working copy, generally the one on your hard disk.

If you have a disk copy program you already know how to use, go ahead. The HyperLogo disk is not copy protected, so you can copy them with any copy program.

Here's how to make backup copies from the Finder. The Finder is the program supplied by Apple Computer that shows disks, lets you copy files, and lets you run programs.

- Get into Apple's Finder. One way to do that is to boot from the System Disk that came with your computer.
- Insert a blank floppy disk into the computer. The Finder will ask you to initialize the disk. Initialize it, using the default name of Untitled.
- Eject the disk by pulling down the Disk menu and selecting Eject. The disk has to be selected before you can eject it. If it doesn't eject, click one time on the disk icon to select the disk, then try again.
- Insert the HyperLogo disk into the disk drive.
- Drag the disk icon for the HyperLogo disk to the disk you just initialized. The Finder will double-check before copying the disk, and will ask you to swap the disks a few times. (Of course, if you have two 3.5 inch floppy disk drives, you can leave both disks in drives and avoid the swapping.)

- Click on the name of the new disk Type HyperLogo (the name of the disk you copied), since the name of the disk can be important. Press RETURN to finish naming the disk.
- Eject the new disk by dragging it to the trash can.
- Set the write-protect tab so the disk can't be accidentally erased, and store the disk in a safe place.

Installing HyperLogo

HyperLogo is a new scripting language for HyperStudio. Before you can use HyperLogo, you need to install the language in your copy of HyperStudio. This is a little different from NBAs and the like, which can go in a separate file and get copied into stacks where they are used – scripting languages, like HyperLogo and SimpleScript, really must be installed in HyperStudio itself.

Installing HyperLogo is really very easy. In a moment, you'll run a simple program that will do all of the work. Before running the installer, though, let's take a moment to see what the installer will really do.

HyperStudio is also two different programs. The program you run to create new stacks, and the program you probably run almost all of the time to run stacks, is called HyperStudio. When you install HyperLogo, you'll copy a small communications program in HyperStudio, and copy a larger file that contains most of HyperLogo into a folder in the same location as HyperStudio. The folder is called HyperByteWorks, and the file inside that contains most of HyperLogo is called HyperLogo. You only have to install HyperLogo once, but if you copy HyperStudio to another disk, you need to be sure to copy this folder and file, too.

There is also a run-time version of HyperStudio called HS.Sys16. That's the one you distribute with your stacks so people who don't have HyperStudio can use your stacks. If your stacks use HyperLogo, you'll also have to install the run-time version of HyperLogo in HS.Sys16. Once again, this will create a folder called HyperByteWorks, but this time, the installer copies a file called Logo.Runtime into the folder. If you copy HS.Sys16, be sure and copy the folder and the file Logo.Runtime, too.

Finally, there are some samples on the HyperLogo disk. The installer can move these to your hard disk for you, or you can copy them with the copy program you normally use.

To install HyperLogo, insert the HyperLogo disk and run the program HyperInstaller. You'll see a dialog with four installer options on the left; they are:

Install HyperLogo

This installs a small communications program in the file HyperStudio, creates a folder called HyperByteWorks, and copies HyperLogo into the folder. Before using this installer option, be sure the folder to the right shows the file HyperStudio.

Install HyperLogo Runtime

This installs a small communications program in the file HS.Sys16, creates a folder called HyperByteWorks, and copies Logo.Runtime into the folder. Before using this installer option, be sure the folder to the right shows the file HS.Sys16.

Install HyperLogo Samples
This copies the HyperLogo demonstration stack, a movie, and the CallBacks file to the folder you pick from the list to the right.

Install Talking Tools
If you have the program Talking Tools from the Byte Works, Inc., HyperLogo can talk. This option copies four tools to your system folder, so it doesn't matter what folder is selected from the right list.

Select the things you want to install, then find the folder where you want to install them from the file list to the right, and click the Install button. Everything else is automatic.

▲ **Warning** 3D Logo requires System Disk 6.0 or later. If you have an older version of Apple's operating system, you will need to update your operating system.

If you don't have the latest System Disk, and would like to get a copy, check first with your local dealer, your user's group, and any online services you have access to. You may be able to get a copy of the latest System Disks free. If all else fails, or if you want to be sure you're getting the complete set of disks and release notes, you can buy them from Resource Central.

Resource Central
6339 West 110th
Overland Park, KS 66211
(913) 469-6502 ▲

Giving Stacks to Friends and Customers

HyperLogo is a copyrighted program, so you can't just give copies away. Like HyperStudio, HyperLogo comes with a run-time version that contains enough of HyperLogo to run your stacks, but doesn't contain the parts that let you create new stacks or modify existing ones. While the run-time module for HyperLogo is copyrighted, too, you don't have to pay anything to distribute it. There are just a few guidelines you need to follow.

There are two ways you can distribute the HyperLogo run-time package. If you have licensed the HyperStudio run-time program from Roger Wagner Publishing, Inc., you can install HyperLogo and distribute a complete, ready-to-run system. If you expect your customer to already have the run-time version of HyperStudio, you can give them a special version of our installer, along with the files needed to install the HyperLogo run-time package.

Distributing HyperLogo With HyperStudio

When you license HyperStudio, you normally distribute a file called HS.Sys16. To add HyperLogo, be sure you've installed the HyperLogo run-time version in this file using the installer script "Install HyperLogo Runtime." Instead of just copying HS.Sys16, you need to copy HS.Sys16, create a folder called HyperByteWorks, and copy the file Logo.Runtime into this folder.

Distributing HyperLogo Without HyperStudio

When you distribute the HyperLogo run-time system without HyperStudio, you need to pass along the installer and another file. Here's the files you need:

```
RTInstaller
RTFolder:
    RTLoader
    Logo.Runtime
```

- RTInstaller is a short version of the installer you used to install HyperLogo. It installs the run-time version of HyperLogo in HS.Sys16. Since there is only one thing to install, there are no check boxes – the user selects the folder containing HS.Sys16 from the file list and clicks Install.
- RTFolder is a folder that contains the two files used by the installer.
- RTLoader is the small communications program that is installed in HS.Sys16.
- Logo.Runtime is the mail part of HyperLogo that gets copied into the file HyperByteWorks.

The folder RTFolder has to be in the same folder as RTInstaller, but you can put this collection of files on any disk you like, even buried inside folders. Once the user installs HyperLogo, they can run your HyperLogo stacks from HS.Sys16.

Copyright Information

When you distribute your stacks, you need to include our copyright notice somewhere. If you have a copyright notice of your own, put ours in the same place. If not, put our copyright notice in an obvious place, like the documentation or read-me file that goes with the stack, or on the title card for the stack itself. The copyright notice should read:

HyperLogo

HyperLogo is a copyrighted program of Byte Works, Inc. licensed to <your company name or your name> to distribute for use only in combination with <the name of your stack>. HyperLogo may not be used for any purpose except to run this stack.

HyperLogo Copyright 1993, Byte Works, Inc. All Rights Reserved.

If your stack is written in another language, please translate the notice to the appropriate language.

Inserting this copyright notice is the only action you need to take. There are no licensing fees to pay.

Getting Answers to Questions

Eventually, you'll probably want to talk with others about Logo.

If you need technical assistance – anything from a bad disk to not understanding the manual to reporting a bug – you should contact the publisher and ask for technical assistance. The publisher occasionally adds new technical support channels or changes hours or phone numbers. For the latest times, numbers, online services and mailing address, read the file Tech.Support, which you can find on the HyperLogo disk.

If you would like to talk to others about Logo in general or HyperLogo in particular, we suggest one of the major online services, like GEnie or America Online. You'll probably find a topic to discuss HyperLogo already there, and can join in the discussion. As this manual goes to press, the publisher sponsors discussions on America Online and GEnie. You can join the discussion on America Online using keyword ByteWorks. From GEnie, use A2Pro, then look in category 36 of the bulletin board.

Finding Out More About Logo

HyperLogo is a scripting language for HyperStudio, but the language itself, Logo, is a popular computer language in its own right. The various books on the Logo language contain all sorts of useful information, programming tips, and source code that you can use in HyperStudio with little or no change. Unfortunately, very few bookstores actually have Logo books on their shelves. That doesn't mean there aren't any good ones around, but you have to ask for them. For a complete list of Logo books, ask any book seller for the Subject volume of Books in Print, and look under Logo. There are several columns of Logo books listed there. Most book sellers will be happy to order almost any of the books you find there.

Here's a few of the more prominent Logo books.

Exploring Language with Logo
E. Paul Goldenberg and Wallace Feurzeig
The MIT Press, 1987
This book is about human languages. You'll learn about the Logo language as you write programs that analyze English – and even programs that write original poems and prose.

Computer Science LOGO Style: Introduction to Programming
Brian Harvey
The MIT Press
This is the most comprehensive course on Logo programming I am aware of. In fact, it's a three volume set. Unlike many Logo books, this one doesn't concentrate on simple graphics for grade school children. This is a full-blown programming course that will teach you quite a lot about Logo, programming, and computers.

Chapter 2 – Getting Started with HyperLogo

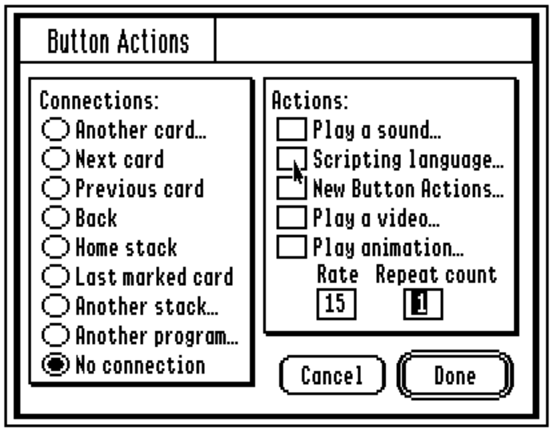
This chapter introduces the basic concepts of the Logo language and HyperStudio scripting using example scripts. You'll start by creating simple scripts that draw lines on the screen, then gradually build on this as you learn about Logo procedures, variables, and lists. Along the way, you'll learn about the HyperLogo environment as you explore the text window, learn to use edit windows, and draw in the turtle window.

As you work through this chapter, one of the things you'll notice is that all of the examples are basically just running Logo programs from a button. Naturally, you'll want to use HyperLogo to control HyperStudio, too – moving to other cards, for example. That topic is important enough that it has it's own chapter. This chapter teaches you enough Logo to get started with the language. Chapter 6 shows you how to use Logo to control HyperStudio.

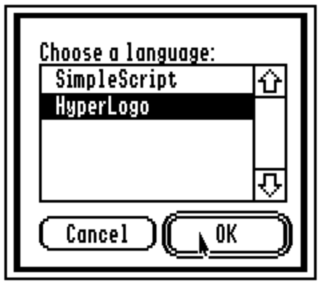
Your First Script

For your very first excursion into scripting, we'll start with something very simple. In this example, we'll create a button that will draw a box when you click on the button.

Start HyperStudio, and create a new stack. Create a new button. The name, shape, and position of the button really don't matter much, although you might want to move the button away from the center of the card. After you pick the position of the button, you'll see the familiar Button Actions dialog:



Click on Scripting Language..., and you'll get a second dialog asking you to pick a language. If you don't get this dialog, you didn't install HyperLogo. It that case, go back to Chapter 1 and follow the instructions on installing HyperLogo.



Select HyperLogo and click OK, and you will enter the HyperLogo scripting environment. In a very real way, HyperLogo is a complete program that happens to be imbedded inside of HyperStudio. It has it's own menu bar and it's own windows. You can create new windows while you are inside HyperLogo, using them as you create and test scripts for HyperStudio.

A script window is opened automatically when you enter HyperLogo. Whatever you type in this script window will be executed when you click on the button, back in HyperStudio. For our example, we'll draw a square roughly in the middle of the card. To draw the lines, we'll move the Logo turtle. Type this line in the scripting window:

```
REPEAT 4 [FORWARD 30 RIGHT 90]
```

Be sure to type it exactly as you see it. The spaces aren't too important, as long as you have at least one wherever you see a space in the example, but everything must be on a single line.

Note You may be wondering exactly what these commands do, and exactly what the rules for creating the line are. That's fair. There's a lot to learn about Logo, though, and you sort of need to know it all just to get started. We'll cover all of these ideas gradually in this chapter. For now, type the line like you see it and keep in mind that your questions will get answered!

The whole purpose of HyperLogo is to create button scripts, so the script window stays open until you leave HyperLogo. In fact, one way to get out of HyperLogo now that you've created the script is to close the script window. Another way is to pull down the File menu and Quit, just like you would get out of HyperStudio. This time, though, you just leave HyperLogo, and go back to HyperStudio.

Either way, since you changed the script, you'll get a chance to cancel the changes or accept them. Of course, you want to accept them. Once you're back to HyperStudio, finish creating your button, adding any sounds or other effects you want to attach to the button. Using a script is just one option, after all. You can still do other things with the same button.

Now that you have a button, try it out! When you press the button, you'll see a small square in the middle of the screen.

Changing A Script

Once you start using HyperLogo a lot, you'll find yourself changing scripts almost constantly. Tweaking a program – making a small change here, and a small change there – is a time-honored way to get a program from a rough idea to exactly what you want. There are two ways to change a button script once you have one. The first, and most obvious, is to edit the button with the button tool, the same way you would edit the button to change a sound, or some other characteristic. Once you're at the Button Actions dialog, you click twice on the Use HyperLogo check box. The second time you click, you'll end up back in the HyperLogo scripting environment.

There's a much easier way to get to the script for a quick change, though. Hold down the command key (the one with the open-apple and expanded clover) and click on the button. Instead of carrying out the button actions, you'll go straight to HyperLogo. Make a change – say, change the number 30 to a 50 to change the size of the square – and close the script window. Click on the button again, and you'll see the result of your new script.

Using HyperLogo to Learn a Little Logo

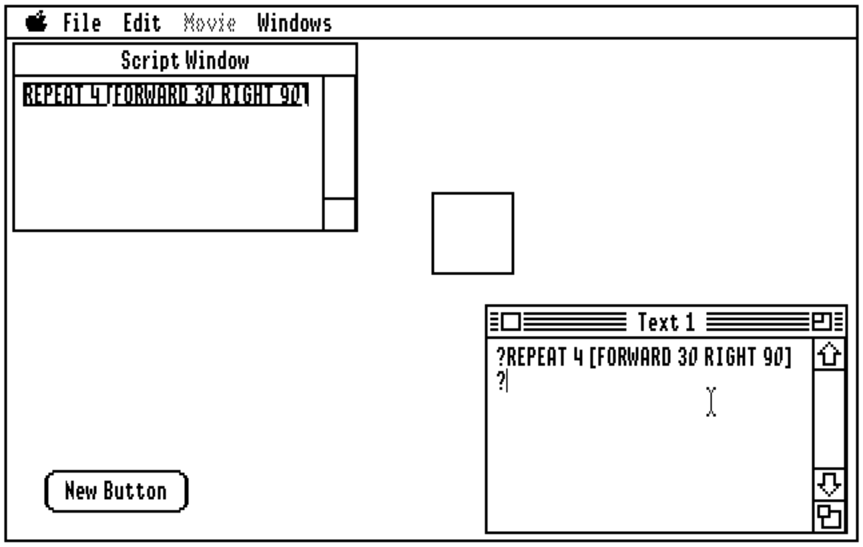
The real purpose of this chapter is to teach you enough about the Logo language so you can write simple scripts, explore more advanced ideas in disk magazines, use examples from online services, and use the reference section of this book to explore other commands. We could continually flip back and forth between HyperStudio and HyperLogo, changing a line and trying it by clicking on a button, but that would take a lot of time. For the rest of this chapter, we'll stay inside the HyperLogo environment.

Text Windows

Get into HyperLogo, either by creating a new button or editing the one you just made. Pull down the File menu and pick New Text Window. You'll get a new window, Text 1, that looks a lot like your script window. There is a major difference, though. To see what it is, start by resizing your windows so you can see the HyperStudio card, then type the same line you used to draw a square and press RETURN:

```
REPEAT 4 [FORWARD 30 RIGHT 90]
```

This time, the square is drawn right away.



```
REPEAT 4 [FORWARD 30 RIGHT 90]
```

```
?REPEAT 4 [FORWARD 30 RIGHT 90]
?
```

Text windows are like script windows that do what you say immediately, rather than waiting for you to get back to HyperStudio and click on the button. We'll use text windows in this chapter to give you a quick, easy way to play with some Logo commands. You'll end up using text windows yourself as you write scripts to experiment with various commands, trying them quickly to see what they do.

Turtle Windows

Logo is a lot more than a graphics language, but drawing with the turtle is an easy, fun way to explore the Logo language. When you're using all of those graphics commands, though, you may want to play with graphics effects without changing the card. Then, too, the card gets refreshed occasionally, and there's no way to see what gets drawn behind a button or one of Logo's windows. Turtle windows solve these problems.

Pull down the File menu and pick New Turtle Window. You'll get a new window on the left half of the screen. You can move or resize this window, of course. Also, if you draw something behind another window, you can select the turtle window to bring it to front, and the things you drew are refreshed so you can see them.

In the rest of this chapter, I'll describe things as if you're typing commands in a text window, and drawing in a turtle window. Of course, all of the commands you see will work just fine in a script, too. Be sure and try a few of them in scripts as you go along.

Simple Logo Commands

Introducing the Logo Turtle...

The triangle you see in the turtle window is the turtle. You don't normally see it on the HyperStudio card, although you can turn it on with the command

```
SHOWTURTLE
```

The turtle is a computer model of the first graphics turtle, which was a real robot that crawled around on the floor, dragging a pen behind it to draw on a big sheet of paper. As you move and turn the turtle, you'll see the triangle move and turn, too.

Tip The turtle is a great way to see what's happening on the screen, but is can slow things down a bit, too. The reason is that Logo is doing a lot more work when it moves the turtle than it has to do just to draw lines. Every time the turtle moves, Logo has to draw six lines – three to erase the old turtle, and three to draw the new one. You can draw pictures a lot faster by hiding the turtle before you start drawing using `HIDETURTLE`, then showing the turtle when you're finished with `SHOWTURTLE`. You can find complete description for these commands, along with two-letter abbreviations, in Chapter 8.

Drawing Lines with the Turtle

Let's see how the turtle works right away. Type

```
FORWARD 30
```

The FORWARD command tells the turtle to go forward 30 steps. In HyperLogo, each step is one pixel long when you move vertically, and two pixels long when you move horizontally. As the turtle moves, it draws a line.

Tip In all of the examples in this book, Logo commands are shown using uppercase letters. You don't have to type them that way; we're just using uppercase letters to make it easier to tell when we're talking about a Logo command.

If you've looked ahead at any of the Logo sample programs in this book, they probably look pretty cryptic. That's strange, in a way, because Logo generally uses easy to understand words like FORWARD for the commands. A lot of commands are used so often, though, that typing them out gets to be a drag, not to mention a waste of space and typing time. To save time, the most common Logo commands have two-letter abbreviations. For example, FD is the abbreviation for FORWARD, and

```
FD 30
```

does exactly the same thing as

```
FORWARD 30
```

Clearing the Screen

If you've tried the abbreviation for `FORWARD`, the turtle may have already marched off of the screen. Any time you want to start over with a fresh screen, type the command

```
CLEARSCREEN
```

This erases everything in the screen and moves the turtle back to its starting position. Most of the time, you'll want to use the abbreviation `CS`.

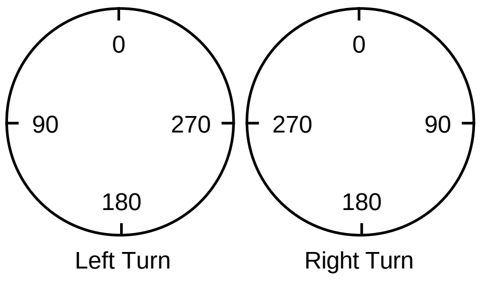
Turning the Turtle

You need to turn the turtle to draw anything interesting. You can turn the turtle left with the `LEFT` command, or right with the `RIGHT` command. These commands have abbreviations, too. They are `LT` for `LEFT`, and `RT` for `RIGHT`.

The turtle turns in degrees, so turning the turtle right 90 degrees is like making a right-hand turn with a car. No matter which way the turtle starts,

RIGHT 90

makes a right-hand turn. Turning 180 degrees in either direction turns the turtle around.



Logo's `REPEAT` command lets you tell Logo to do more than one thing at a time. The line

```
REPEAT 5 [FORWARD 30 RIGHT 144]
```

tells Logo to repeat the commands in the brackets five times. Logo knows there are two commands inside the brackets because it knows `FORWARD` only needs one number for an input, so the word right after 30 has to be a new command.

It turns out that the way you type this line is important. The `REPEAT` command is a single line, with no RETURN keys anywhere in the line. If the text window isn't wide enough to display a line, the line will get wrapped in the text window, but you can't type a RETURN key to break the line up yourself.

If you haven't already tried the `REPEAT` command, type the line now. If the turtle isn't on the screen, use `CLEARSCREEN` first. Logo will draw a small star.

Exploring Black on white is fine, but you can use color, too. Try `SETPC` followed some number from 0 to 15, then draw some lines, like this:

```
SETPC 4
REPEAT 6 [FD 20 RT 60]
```

You'll find more information about color in Chapter 8. You might want to try the `SETBG` command, too, to change the background color.

Creating Procedures with TO

Logo programs are usually written as a series of very short procedures. There are several ways to create procedures. We'll start with a simple, interactive command called TO.

TO teaches Logo to do something new. The procedure you create will look and work just like the built in commands you've used so far. Of course, you have to give the command a name. The TO command expects you to type the name of the procedure you are creating right after TO.

To see how this works, let's create a procedure to draw a square. Start with the TO command, like this:

```
TO Square
```

So far, Logo always shows a ? character when you can type a command in the text window. Once you type TO, though, you start entering a procedure. Logo switches to a > character to remind you that you are typing in a procedure, now. As long as the > prompt is shown, the commands you type aren't executed, like they have been so far. Instead, Logo saves these commands and executes them when you type the name of the procedure.

Our sample procedure will draw a square, so you can type the same line you used earlier to draw a square, or you can switch to the abbreviations, like this:

```
REPEAT 4 [FD 20 RT 90]
```

When you finish typing all of the commands for the procedure, finish it off by typing

```
END
```

This tells Logo you are finished entering the procedure.

Running your procedure is as easy as using any other Logo command. To draw a square, type

```
Square
```

Logo procedures can call the built-in procedures, or the ones you define. Once you enter a procedure, anyone can call it. Here's two classic Logo procedures that show how this works. Type them in and try both `Flag` and `Flower`.

```
TO Flag
FD 20
SQUARE
PENUP
BACK 20
PENDOWN
END

TO Flower
REPEAT 10 [FLAG LEFT 36]
END
```

There are a few new commands in `Flag`. `PENUP` raises the turtle pen, so it doesn't draw a line as it moves, and `PENDOWN` puts the pen down so it will start drawing lines again. `BACK` is the opposite of `FORWARD`: the turtle moves backward. All of these commands have abbreviations. You can find a list of them in Chapter 8. It's a pretty good idea to start exploring that chapter now, in fact, reading about new commands as we try them, and scanning the chapter for similar commands you might like to try.

The Logo Workspace

So far, you've created three procedures. Now try this: Close the text window completely. You'll be asked if you want to save the window to disk; say no. Now open a new text window by pulling down the File menu and picking New Text Window. Finally, type

```
CS
FLOWER
```

The point of this little exercise is to introduce the concept of the Logo workspace. The text window you use to type command is just a scratch pad. You can use several of them, close them, delete all of the text in the windows, or even select lines from the text window and execute the lines a second time.

When you create a procedure, though, it's independent of the text window. Procedures are stored in Logo's workspace, where they can be used from any text window. The workspace doesn't go away when you leave HyperLogo, either. The procedures you put in the workspace can be used by any script in the same stack. They only go away when you deliberately delete them with a Logo command, or when you leave the stack and come back to it.

Listing Procedures in the Workspace

Of course, after a while you might loose track of just what is in the workspace.

```
POALL
```

prints everything in the workspace, including variables and property lists – two topics we haven't talked about, yet. You can also print just one procedure with the PO command:

```
PO "Flower
```

These commands show the procedure just like you entered it. You can quickly change the procedure with a few mouse clicks and keystrokes. Let's say you want to create a slightly larger square. Start by typing

```
PO "Square
```

This gives you a listing that looks like this:

```
TO Square
REPEAT 4 [FD 20 RT 90]
END
```

To change the size of the square, change the 20 to a 30. When you are through, select all three lines and press the return key. When you enter a procedure with the same name like this, it replaces the old procedure. This is a quick way to make short changes to a procedure.

Printing the entire workspace prints a lot of stuff, when you may just want to get a quick list of the procedures that are in the workspace. The `POTS` command lists just the name of the procedures.

Some of these names seem pretty cryptic at first. Think of them as "Print Out" something. In the case of `POTS`, think "Print Out TitleS."

Exploring There are several varieties of the `PO` command. You can find a complete listing in the reference section. A few you might want to try right away are `POT`, which prints a single procedure title, and `POPS`, which prints just the procedures in the workspace. True, so far all we know how to put in the workspace is a procedure, but that will change shortly!

Editing Procedures with Edit Windows

Editing a procedure in the middle of the text window is a neat, fast trick, and it works especially well when you've just entered a short procedure and want to try a quick change or two. Sometimes, though, you will want to edit a procedure with a separate window. The slickest way to do that is with the `EDIT` or `EDITS` command; they open up a separate kind of window called an edit window.

To see how edit windows work, edit the `Square` procedure with this command:

```
EDIT "Square
```

The `EDIT` command opens a new window with `Square` in the window. Edit windows are different from text windows: When you type a line in a text window and press RETURN, the command is executed right away. When you type a command in an Edit window and press return, you just change what's in the edit window.

Change the length of the sides of the square from 30 (or 20, if you didn't make the last change) to 40. Now select the original text window by dragging the edit window to the side and clicking on the text window. If you try the procedure, you'll find that the size of the square has changed. Any time an edit window is the front window and you close it or select a different window, Logo enters everything in the text window into the workspace.

Tip What if you have made several changes in an edit window and don't *want* to enter them in the workspace? In that case, delete everything in the edit window and *then* close it.

In our sample procedures, you actually have two procedures, `Square` and `Flag`, which are closely related. When you change the size of one, you probably want to change the size of the other. `EDIT` can edit both procedures at once, in the same edit window, by putting the names of the procedures in a **list**. Lists are something we'll talk about more later. So far, you've used **words** to name a procedure. Words are the equivalent of strings in other languages, but they look a little strange, since there is no closing quote mark. In a list, you type the names without quote marks, and put all of the names inside brackets, like this:

```
EDIT [Square Flag]
```

`EDITS` doesn't need any names. It's the plural form of `EDIT`. The `EDITS` command opens an edit window that shows everything in the workspace.

Exploring Actually, you can hide procedures from EDITS, and from all of the other commands that work on "everything" in the workspace. Logo has a series of bury commands that hide stuff from normal view, and some unbury commands that retrieve the things you have buried. Try BURY and UNBURY; you'll find descriptions and examples in the reference chapter.

Using TO in the Script Window

You can use the TO command from the script window, too. Just be sure to finish what you start, making sure there is an END statement to finish off each of your procedures.

By now, you have three different ways to create a procedure. You can create a procedure interactively in a text window, change a procedure or even type in a new one in an edit window, or type a procedure directly into a script window. So what's the difference? To a large extent, there isn't one. In every case, the procedure gets saved in the workspace, where it can be used by any button in your stack. The reason there are so many different ways to create a procedure is just for convenience.

In general, I would recommend developing procedures with a combination of text windows and edit windows. Use text windows to create very short procedures and to test your longer procedures. Use edit windows to edit procedures you've already created, and to create long procedures.

Once you've developed a procedure, print it in either a text or edit window, copy it into the scrap using the Copy command, and paste it into your script. In most cases, it's best to put the procedure in the same button script where it is used. There are two pitfalls to be careful of, though.

First, creating a procedure does take time. In most cases, the amount of time it takes to create a procedure just doesn't matter – a few hundredths of a second to define a procedure isn't going to be noticed when you push a button. In some time-critical cases, though, you might want to create an auto-activate button that is executed as soon as you arrive at the card, and declare your procedures there.

Second, you might need a procedure from more than one script. Pasting the procedure into every script that needs it works, but it takes time and space. Then, too, if you need to change the procedure, you have to hunt it down in all of the different places you saved it and change it in each place. If you need a procedure in more than one script, it's time for an auto-activate button.

Exploring If you've never used an auto-activate button, you're in for a treat. These buttons do things automatically, even when the user doesn't click on anything. Check out your HyperStudio manual for details.

Removing Procedures from the Workspace

There comes a time when you might want to clean up your workspace. After all, the procedures do take up space. Then again, you might have tried a few things that didn't work out, and you might want to get rid of the failures so you don't see them every time you list all of the procedures in your workspace.

ERASE erases a procedure from the workspace. You can erase a single procedure or a list of procedures. Here's two samples you can try; they will erase the three procedures you've created so far:

```
ERASE "Square
ERASE [Flower Flag]
```

Exploring There are several other erase commands that work just a bit differently. They are all grouped together in the reference chapter. Two you might find interesting right away are ERALL and ERPS.

Variables

Creating Variables with MAKE

Logo's MAKE command is used both to create a new variable and to assign a value to a variable. To create a variable called size and save, say, the size of a square, you could use

```
MAKE "size 20
```

The first thing MAKE looks for is the name of a variable. With the single exception of the TO command, anytime you want to give a name to Logo, you use a single quote before the name. Right after the name is the value you want to stuff into the variable.

When you want to use the value stuffed inside of a variable, use a colon before the name. Here's `Square`, reworked so it uses `size` to figure out how big the square should be.

```
TO Square
REPEAT 4 [FD :size RT 90]
END
```

You can put this variable to use right away to create a cascade of ever larger squares.

```
MAKE "size 10
REPEAT 8 [Square MAKE "size :size + 10]
```

Quotes and Colons and Nothing, Oh My!

Take a close look at that last `MAKE` command. It's still setting the variable `size`; you can see the name with the quote mark right after the name of the `MAKE` command. The next part is `:size + 10`, which `MAKE` treats as a single lump, adding 10 to whatever used to be in the variable `size`.

Using quotes in one place and colons in another probably seems awkward, and it is, at first. The way to keep the two punctuation marks straight is to remember that `"size` is the *name* of the variable, while `:size` is the *value* of the variable. `Size` with nothing else is a *procedure*; Logo will look for a procedure called `size`, run it if one is found, and use the value `size` returns.

This triple use of a single name is both the bane of people who are just starting to use Logo, and one of the most powerful features of the language. After all, if there wasn't some reason for all these punctuation marks, Logo would have just left them out, like most other languages. The reason is tied to the fact that Logo programs can change themselves as they run, adding new procedures or using variables to do tricks that would be very hard to duplicate in a traditional programming language. For example, you could us a variable value for the first parameter to `MAKE`, like this:

```
MAKE :variable 20
```

This single command can initialize any number of different variables. What Logo actually does is look inside the variable `name` for the value, which in this case is the name of the variable to set!

If all of this seems a little strange at first, don't feel left out. It seems a little odd to most people the first time they see it, especially to people who are used to other computer languages, like BASIC or C. The important thing, for now, is just to remember that *quotes mean names*, and *colons mean values*. As you learn more about Logo and start to explore some of its power, you'll gradually pick up a lot of tricks that make this oddity one of the most powerful features of the Logo language.

Printing a Variable

There are several printing commands that let you look at the value of a variable. One of them is `SHOW`. Here's how you can look at the value of `size` using the `SHOW` command:

```
SHOW :size
```

Just for the sake of experimenting, try the same command with a quote mark:

```
SHOW "size
```

This time, `SHOW` shows the name of the variable, which is, after all, what you asked it to do.

Exploring There are several ways to print a value, and each has it's own unique features. Two other printing commands you may want to use from time to time are `PRINT` and `TYPE`.

Logo Numbers Come in Two Flavors

Like most computer languages, Logo uses two different kinds of numbers. Whole numbers, like 4, -16 and 10000, are called integers. Numbers with a fraction part, like 4.27 or 3.14159, are called floating-point numbers. You can use either kind of number in Logo, and you can mix them any way you like.

Word Variables

You can save words in a number, too. Logo words are the equivalent of strings in many other computer languages, but they work a little differently. There is no closing quote on a Logo word. A word ends when Logo finds the first punctuation mark, usually a space. To imbed a punctuation mark in a word, use a \ character right before the punctuation mark. Here's an example that stuffs a complete word into a variable, then prints the value.

```
MAKE "Hi "Hello,\ world.
SHOW :Hi
```

Besides spaces, all of these characters will stop a word:

```
[ ] ( ) = < > + - *
```

You have to put the \ character before any of these characters to stuff them into a word. To put a \ character in a string, use two in a row, like this:

```
MAKE "str "Here's\ how\ you\ put\ in\ a\ backslash:\ \\
```

Lists

The last kind of Logo value that you can stuff into a variable is the list. Even if you're a very experienced programmer, you may not have used a language that handles lists. Lists are enormously important in artificial intelligence languages like LISP (LISt Processing) and Logo, but are completely missing from languages like C and Pascal.

A list is just a series of values. You put all of the values in square brackets to form the list. About the only difference between stuffing values into a list and putting a single value in a variable is that you don't need a quote before words.

```
MAKE "Hi [Here's an example of a list of words.]
PRINT :Hi
```

You can put any kind of variable you want in a list, including another list. You can even mix different kinds of variables in a list.

```
MAKE "Sample [1 3.14159 [List in a list] word]
```

Lists are very important in Logo, but unless you've used LISP, you've probably never seen this sort of data before. The best way to learn about lists is to use them in a variety of ways. You'll see lists used quite a few times in several different and very powerful ways throughout the rest of the introductory chapters.

Exploring Manipulating lists – pulling them apart, putting them together, searching a list, and so forth – is what makes Logo a powerful artificial intelligence language, not just a kid's graphics language. In fact, our Logo was tested in part by working problems from a LISP artificial intelligence textbook!

You'll see examples of lists and commands that manipulate lists and words scattered throughout this book. It's worth looking over the various commands that manipulate lists now, though, too get a preview of some of the commands you can use. Check out "Words and Lists" in Chapter 8 for other list manipulation commands, plus a lot of examples.

Logo Variables Do It All

So far, you've stuffed integers, floating-point numbers, words and lists into variables. How does Logo know what a variable is supposed to hold? After all, anyone who has used BASIC or Pascal or C knows that stuffing the wrong kind of value into a variable either won't work at all, or can cause serious problems.

Well, the answer is that Logo doesn't really care. Logo variables are general purpose buckets. You can stuff whatever you want into a variable. In fact, a variable can hold a number on one line, a word on the next, and a list on yet another line, and it will work just fine!

```
MAKE "Var 14
PRINT :Var
MAKE "Var "XIV
PRINT :Var
MAKE "Var [1 2 3]
PRINT :Var
```

Logo even takes this mix-and-match approach to variables one step further: technically, there really are only two kinds of variables in Logo. Logo uses lists and words, and that's it. Numbers are just a special case of a word, with some special properties. Later, you'll find some Logo commands that work on words, and every one of them will work on numbers, too!

Let's look at an example to see how this happens. Try this:

```
MAKE "a 01
MAKE "b "02
```

The first line creates a variable called A and stuffs the integer 1 into the variable. Next, the word "02 is stuffed in the variable B. This is important: B really does have a word, not an integer. To see what the difference is, try these print commands:

```
PRINT :a
PRINT :b
```

The value of 01 is 1, so Logo prints 1. The value of "02, though, is a word, not a number, and you get both characters.

Since a number is just a special case of a word, you can use words in mathematical expressions, as long as the word translates into a number. This really will work, and will print 3:

```
PRINT :a + :b
```

So words can be used as numbers. The opposite is also true. `FIRST` is a command that returns the first character from a word (or the first item from a list).

```
PRINT FIRST "ABC
```

will print A. But numbers are just a special case of words, and

```
    PRINT FIRST 3.14159
```

prints 3!

Of course, numbers really are stored a different way than words. If they weren't Logo programs would be very, very slow. There are a few places where the difference is important, like when a number has extra digits.

```
    PRINT FIRST 001
```

will print 1, not 0.

Exploring There are times when you need to know just what a variable contains before you try to do something with it. Is a variable a number? Is it a list? Since variables can hold anything, you need some way to find out. You'll find a series of predicate commands – commands that end in a P – that can look inside a variable to see just what's there. They are `LISTP`, `WORDP`, `FLOATP` and `INTEGERP`. You can find examples in Chapter 8.

What About Arrays?

If you've used another computer language before, like BASIC or Pascal, you've probably used arrays to hold a fixed number of values. Maybe you used an array to work with a matrix for a math class, or to hold a series of names for a simple database.

Well, Logo doesn't have arrays. In fact, it doesn't need them. Lists do the job quite well. Just because you can mix the kinds of values you stuff in a list doesn't mean you have to mix them!

In Logo, an array is a list. An array of arrays is a list of lists.

Variables as Parameters

Traditionally, Logo programs are written as a series of short procedures that are debugged as they are developed. In these short procedures, the most important kind of variable isn't usually the ones that are create with the `MAKE` command. The most commonly use variables are actually parameters to procedures.

There are several differences between parameters and the variables you have created with `MAKE`. The most obvious is the way the variable is created in the first place. To create a parameter, you put the variable name on the `TO` line you use to create the procedure, like this:

```
TO Square :length
```

Once you've created a parameter, you can use the value or even change the value just like you did with variables created with `MAKE`. Here's a complete version of `Flag` and `Square` that use parameters to set the length of the lines:

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

TO Flag :length
FD :length
Square :length
PU
BK :length
PD
END
```

You can see how to pass a parameter by looking at how `Flag` calls `Square`. Here's another example, this time drawing a flag:

```
Flag 25
```

The second big difference between parameters and variables created with `MAKE` are where they can be used. In very technical books you'll see this called the **scope** of the variable. If you've been following along for the last few pages, you've created several variables with `MAKE`, like `HI`, `A`, and `B`. These variables are global; they're stuffed into the workspace just like procedures. From the workspace, they can be read or changed from any script in your stack. There's even a set of commands to look at the values and erase them, just like for procedures. Try that now:

```
PONS
```

You can think of `PONS` as Print Out NameS. It shows all of the variables in the workspace, along with their values. One of the things you didn't see, though, was a variable called `length`. That's because parameters are created when the procedure starts to run, and go away as soon as the procedure is finished. They can only be used and set from inside the procedure, too. `Flag` and `Square` both have parameters called `length`, but these are two separate variables. For example, you can't use this shortcut:

This Does Not Work!

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

```
TO Flag :length
FD :length
Square
PU
BK :length
PD
END
```

The reason this doesn't work is that `length` is only available inside of `Flag`. Even though `Square` is called from `Flag`, it can't use `Flag`'s variables.

Exploring When you create a variable inside of a procedure with `MAKE`, the variable is still entered into the workspace. You can create something called a local variable, though, using `LOCAL`. Local variables exist only inside of the procedure, just like a parameter.

Math With Logo

You've already seen some simple math operations in Logo. In the next few paragraphs, we'll take a quick look at how Logo handles math operations. There aren't many surprises here, so this section will be fairly quick. If you are completely new to computer languages, this may go a little to quickly. If that's the case for you, there are several things you can do to find out more about how Logo handles math operations:

- Chapter 8 of this manual has a more in-depth description of Logo's math capabilities. Look in the section "Numbers and Arithmetic."
- Beginner's books on Logo all deal with math operations on one level or another. Look under "Logo" in the subject listing of Books in Print at any large bookstore for a long list of Logo books you can choose from. (Even though you're using HyperStudio, these books will still work. HyperLogo is a complete Logo that just happens to be a scripting language, too!)
- Probably the best idea is just to try some things as you read this section. The worst thing that will happen if you make a mistake is Logo will tell you what the mistake was – all in all, not a bad way to learn!

Addition, Subtraction, Multiplication and Division

Any Logo command that will take a number will also take a mathematical expression. For example, `FORWARD` will take a constant, like

```
FORWARD 40
```

or an expression, like

```
FORWARD 20 + 20
```

Anywhere you can use a number, a variable will do, too, as long as the variable contains a number or a word that can be converted to a number. You'll see expressions like this a lot:

```
MAKE "Distance 20
FORWARD :Distance + 20
```

Logo supports the four standard math operations you see on just about any calculator. The operations are:

`a + b`	Add `a` and `b` together.
`a - b`	Subtract `b` from `a`.
`a * b`	Multiply `a` and `b`.
`a / b`	Divide `a` by `b`.

▲ **Warning** Unlike the other math operators, the `/` character isn't a separator. That has a very important consequence: When you type something like `:A+:B`, Logo knows you are adding the value of :A and :B together, since `+` is a separator. When you type `:A/:B`, though, Logo treats the `/` character as a part of the name of the variable.

The proper way to type an expression that uses a division operator is to put spaces on both sides of the `/` character. ▲

Of course, you can mix these operations, putting more than one in a single expression. The only thing you have to be careful of is what order the operations are done in. For example, what happens when you type

```
FORWARD 1 + 2 * 3
```

There are two ways to handle the math in this case. Both make sense, and both are used in various computer languages. The way most algebra students learn to handle this expression is to do the multiplication first, so the answer is 7. It's also possible to do the operations left to right, which would move the turtle forward 9. Logo uses the same conventions as algebra, so this example would actually move the turtle forward 7 units.

So what if you want to change the order of operations? Logo uses parenthesis, just like you use in algebra. The only difference is that in algebra, you can leave out a multiplication sign just before or after a parenthesis, so 3(1+2) is pretty common. Logo makes you put the multiplication sign in.

Here's our example, using parenthesis to change the order so the turtle moves forward 9 units:

```
FORWARD ( 1 + 2 ) * 3
```

Negative Numbers

Numbers can be positive or negative, of course. It's perfectly natural, and works just fine, to move backward this way:

```
FORWARD -10
```

You can also put a minus sign in front of a variable, like this:

```
FORWARD -:distance
```

Either way, the effect is the same as subtracting the number from zero. There is a problem with the unary subtraction operator, as this operation is known. To see what the problem is, let's have Logo echo a simple list of two numbers back to us. When you type

```
PRINT [1 2]
```

Logo echoes back with

```
1
2
```

That's natural enough. Now try

```
PRINT [1-2]
```

Be sure you type the line with no spaces on either side of the `-` character. Logo decides this is a subtraction operation, and says

```
-1
```

Now try

```
PRINT [1 -2]
```

making sure you put a space before the `-` character. This time Logo treats the `-` operator as a minus sign, and responds with

```
1
-2
```

This isn't what you would expect if you've used other computer languages. Logo is a list processing language, though, and it really did need some rule to tell the difference between subtraction and negating a number. The actual rules used are a little complicated, but you're safe with a simple rule of thumb: Always put a space before a minus sign, but never put one before a subtraction operation. The complete set of rules is in the section "Expressions" in Chapter 8.

Functions

The last thing you can put in a Logo procedure is a function. A function is a procedure that returns a result. For mathematical expressions, of course, the result has to be a number.

There are a lot of built-in Logo functions. One example is `SQRT`, which takes the square root of a number. Here's a simple example that uses the `SQRT` function to find the length of the hypotenuse of a right triangle:

```
PRINT SQRT :A * :A + :B * :B
```

Exploring The section "Expressions" in Chapter 8 lists all of the functions that are built into Logo. If you want to write your own Logo functions, check out the `OUTPUT` command.

Reals and Integers

Back in the section "Logo Variables Do It All," you learned that there are really two kinds of numbers in Logo. The first kind are whole numbers, which are called integers in computer circles. The other kind are real numbers, which can have a fractional part.

So which do you use?

The answer is pretty neat: Don't worry about it. Logo figures things out for itself, and uses whichever kind of number makes sense. In those rare cases when some Logo function needs a real number, and you give it an integer, Logo just converts the number. If a function needs an integer, and you supply a real number, Logo rounds the real number to the closest integer.

There are only three places where you need to be aware of what sort of number you are using:

- When you print a number, you may want a little control over the format.
- If you are doing operations on very large integers, you might end up with an integer that is too large for Logo to handle. You can avoid a math error by converting the integers to real numbers. (The largest integer HyperLogo can handle is 2147483648.)
- When you are doing complicated drawings, Logo can turn the turtle faster if you use integers for angles instead of real numbers.

In all three cases, you can use Logo's `INT` and `FLOAT` functions to convert back and forth from integer to real numbers.

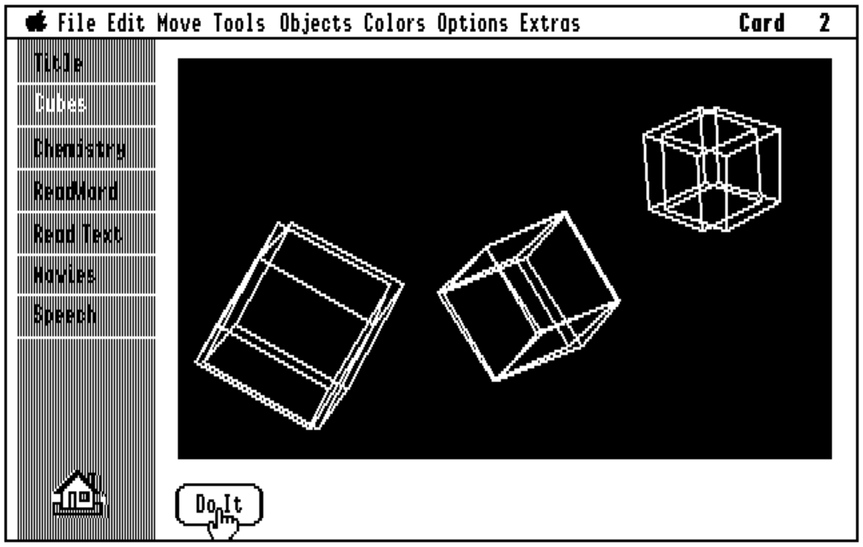
Chapter 3 – 3D Pictures with HyperLogo

The Cubes Program

This chapter shows you how to create your own 3D pictures with HyperLogo. Before we get too far into the chapter, though, let's take a look at an example. Start by opening the stack HyperLogo.Demo. When you installed the samples back in Chapter 1, this is one of the things you installed. If not, you can find the stack on the HyperLogo disk. This stack has a number of demonstration scripts we'll use in this chapter and the next one. You'll start with a title card. Each of the demonstrations is on a separate card, with the name of the demonstration in a button on the button bar to the left of the screen. Move to the Cubes demonstration by clicking on the Cubes button.

It's time to put on the 3D glasses! Make sure the red lens is on the left. It also helps to dim any bright lights, although normal room lighting won't destroy the effect. Now click on the Do It button.

You'll see three cubes on your screen, like this, although yours will be jumping out of the screen instead of laying flat on the page.



Problems? There are two common reasons why the picture might not look three-dimensional:

- You must use a color monitor. The 3D turtle has a lot of uses besides drawing true 3D pictures, but to see the 3D pictures, you must use a color monitor.
- If the drawing just doesn't look three-dimensional – if it looks like red and blue lines, for example – give it some time. Try playing with the room lighting, too. We've found that, roughly, the older the person, the longer it takes to relax and see the 3D picture. Eventually, you should see gray or purple lines on a black background.

How the 3D Display Works

You may already see how the 3D display works, but let's stop and talk about it for a moment. Some of the ideas will help you create realistic 3D pictures, even if you already know the basic concepts.

Hold your finger up at arm's length in front of a background that is at least a few feet away. Cover your left eye and look at your finger, but notice where it is against the background. Now, without moving your finger, uncover your left eye and cover the right one. Your finger will jump to the right against the background.

Your brain is a powerful image processor. It's so powerful, in fact, that you don't notice how you see three-dimensions unless you really stop to think about what is happening and use tricks like covering one eye or the other. Basically, though, you are seeing two pictures at a time, each from a slightly different vantage point. Your brain automatically combines these images to tell you how far away things are.

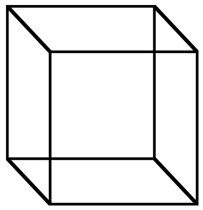
Of course, there are limits. If something is so far away that it doesn't shift against the background, you can't really tell anything except that it's not close. People with one eye, or with severely limited vision in one eye, can't tell how far away things are–they have no depth perception.

Logo is using a pair of special glasses to give your brain two pictures of the same scene from slightly different viewpoints. Since you only have one computer screen, there's obviously a trick involved. Logo draws one picture using red and one using blue. The red picture comes through the red lens just fine, but is blocked by the blue lens, so your brain sees this as the left-eye view. The blue picture is the right-eye view.

Note HyperLogo is using color to split the images between your left and right eye, which means you can only draw black-and-white pictures in three dimensions with HyperLogo. There are other ways to do the same trick, of course. At some theme parks, they use polarizing lenses and two projectors to get the same effect, but because they are using polarizers, the theme parks can still show color. So why doesn't HyperLogo do this? Basically because you can't display two polarized images on a standard computer monitor, and we didn't think you'd want to spend tens of thousands of dollars for a special monitor!

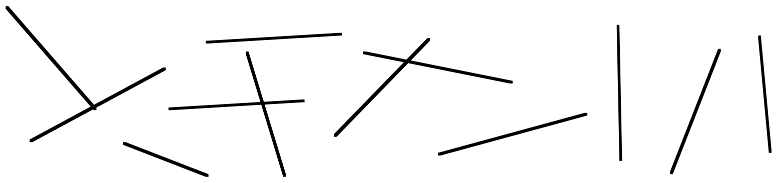
Realistic 3D Pictures

It may take a little practice, but the cubes demonstration should look pretty realistic. In fact, some of our younger testers even reach out to grab the cubes! Some of the simple lines you'll be drawing as you explore the 3D turtle won't look like they are really three dimensional, though. The reason is simple. Your brain is so used to processing three dimensional, solid images like the ones you see in the real word every day that it desperately wants to see the cube as a three dimensional object. You can even see a cube with a simple line drawing, like this one:



You can't tell which is the front and which is the back, and in fact, many people can mentally flip the front and back of the cube as they look at it. With the cube, then, your brain really wants to see three dimensions, and Logo just gives it a helpful nudge in the right direction.

Now look at this set of lines – the same number as in the cube:



This time, there is no order, and your brain isn't picking out an image. This picture doesn't look three dimensional at all. If you draw single lines which aren't connected to anything in HyperLogo, your brain has a hard time using the depth information in the picture. You may see a jumble of red and blue lines instead of a three dimensional set of lines. You might even get a little disoriented, since your brain doesn't understand why some of the lines appear in one eye and not the other.

The secret to drawing realistic pictures with HyperLogo is really very simple: Try to help the brain by giving it a picture of solid objects or lines that connect. Don't try to fool the brain by drawing points or lines that your brain rarely, if ever, sees floating in air in real life.

The 3D Turtle

Creating 3D pictures with HyperLogo is really very simple. The first step is to clear the screen and start out in the 3D viewing mode. In the turtle window, you can do both with `CLEARSCREEN3D`. `CLEARSCREEN3D` works pretty much like the `CLEARSCREEN` command you learned in Chapter 2, but in sets up a 3D screen instead of a 2D screen. The turtle moves to the center of the screen, pointed up. You've already used the abbreviation for this command, which is `CS3D`.

The other way to switch to a 3D turtle is with `SHOWTURTLE3D`, abbreviated as `ST3D`. You can switch back to a 2D turtle with `SHOWTURTLE`. Normally this isn't very important, although you might want to do some tricks by mixing 2D and 3D drawings. It's still an important point, though, since you might be in the habit of using `HIDETURTLE` and `SHOWTURTLE` to speed up your drawings. When you're using the 3D turtle, you need to use `SHOWTURTLE3D` instead of `SHOWTURTLE`.

In general, use `CLEARSCREEN3D` in the turtle window, and `SHOWTURTLE3D` in a script. If you use `CLEARSCREEN3D` in a script, the whole card will get painted black. That may be OK, depending on what you want to do with the card, but most of the time you'll just paint the part of the card where you'll be drawing the 3D picture black.

Rotating Out of the Screen

You can turn the 3D turtle left and right with the `LEFT` and `RIGHT` commands, just like you did with the 2D turtle. In fact, everything you did with the 2D turtle will still work with the 3D turtle. You can even set the pen color – but the result with the 3D turtle is black or white, not the colors you had with the 2D turtle. As long as you only use 2D turtle commands, you're locked in the screen, and all of the lines will be gray (or purple, without the glasses).

Two new rotate commands free your pedestrian turtle, so it can take to the sky. `ROTATEOUT` (abbreviated `RO`) turns the turtle up, rotating it out of the screen. `ROTATEIN` (abbreviated `RI`) turns the turtle down. You can put these to work right away, drawing a line that moves out of the screen. Of course, like we mentioned in the last section, a single line doesn't look very three dimensional, but if you look at it carefully enough, you should see the line coming out of the screen.

```
RO 45
FD 50
```

Rolling the Turtle

Imagine the turtle as a bird, or a plane – or maybe just a really strange turtle with wings. Flying around in the air, you can point the turtle in any direction with the four turning commands you already know. There is one more way to rotate the turtle, though, and that's to roll the turtle. Rolling the turtle doesn't change the direction it's pointed. Rolling the turtle is like a bird or plane dipping one wing while raising the other, spiraling so the nose stays pointed straight ahead. `ROLLRIGHT` (abbreviated `RLR`) rolls the turtle so it's right edge goes down and the left edge rolls up, turning in a clockwise direction compared to the way the turtle is pointed. `ROLLLEFT` (abbreviated `RLL`) rolls the turtle the other direction.

Some 3D Shapes to Play With

To see how these commands work, let's dissect the Cube procedure from the Cubes sample you ran at the start of the chapter. Open the script for the Do It button, and look for the Cube procedure.

The Cube procedure has a single parameter, the size of the cube. It starts like this:

```
TO Cube :size
```

A cube is basically a 3D version of a square, so that's how we'll draw it. The first step is to draw a square with posts pointed up at each corner. To draw a square, you use a command like this:

```
REPEAT 4 [FD :size RT 90]
```

To put a post pointed straight up from where the turtle is, you rotate out 90 degrees, draw the line, back up to the starting place, and rotate in 90 degrees so the turtle ends up right back where it started. The Logo commands to draw a post are

```
RO 90
FD :size
PU
BK :size
PD
RI 90
```

Putting these two ideas together, here's the line that draws the bottom of the cube and the four corner posts:

```
REPEAT 4 [FD :size RO 90 FD :size PU BK :size PD RI 90 RT 90]
```

The next step is to move to the top of the starting post. It works pretty much the same way as drawing a post, but you don't draw the line or back up after you get to the top.

```
PU
RO 90
FD :size
RI 90
PD
```

Next, you draw a square to form the top of the cube.

```
REPEAT 4 [FD :size RT 90]
```

The last step is to move the turtle back to it's starting point. This step doesn't draw anything, but it makes it easier to use the Cube procedure to draw more complicated pictures. By putting the turtle back where we found it, we don't have to worry about what the procedure might do to the position or orientation of the turtle when we use it to draw a cube.

```
PU
RO 90
BK :size
RI 90
PD
END
```

With a little work, you can see how to create the first useful addition to Cube. By taking three inputs instead of one, and using them for the depth, width, and height of the cube, you can draw a box – and with a little imagination, that box could be a building in a city, the body of a simple car, or the top of a table in a room.

```
   To Post :height
   RO 90
   FD :height
   PU
   BK :height
   PD
   END

   TO Box :width :depth :height
   REPEAT 2 [FD :depth Post :height RT 90 FD :width Post :height
RT 90]
   PU
   RO 90
   FD :height
   RI 90
   PD
   REPEAT 2 [FD :depth RT 90 FD :width RT 90]
   PU
   RO 90
   BK :height
   RI 90
   PD
   END
```

This one may stretch your imagination a bit. Octahedron draws an eight sided figure in space, where each of the eight sides are a triangle with all of the angles 60 degrees. You could draw it as a series of triangles, but this way is a little faster. This procedure draws the octahedron as three squares connected at the corners. It's a fun shape to play with in three dimensions.

```
TO Octahedron :size
RT 45
Top :size
LT 45
FD :size
RT 90 + 45
Top :size
LT 45
REPEAT 3 [FD :size RT 90]
END

TO Top :size
RO 45
REPEAT 4 [FD :size RI 90]
RI 45
END
```

3D Without the Glasses (The Chemistry Card)

If you think 3D shapes are just a gimmick or a kid's toy, this section should change your mind. There are a lot of very practical uses for the 3D turtle, and this section will show you one of them.

Tip This is a pretty complicated example. If you've programmed in other computer languages, you'll probably work right through this section without a hitch. If you have never programmed before, though, or if you were never very comfortable programming in other languages, this example may be a bit deep.

If you decide this example is a little beyond where you're at, there are two ways you can approach this section.

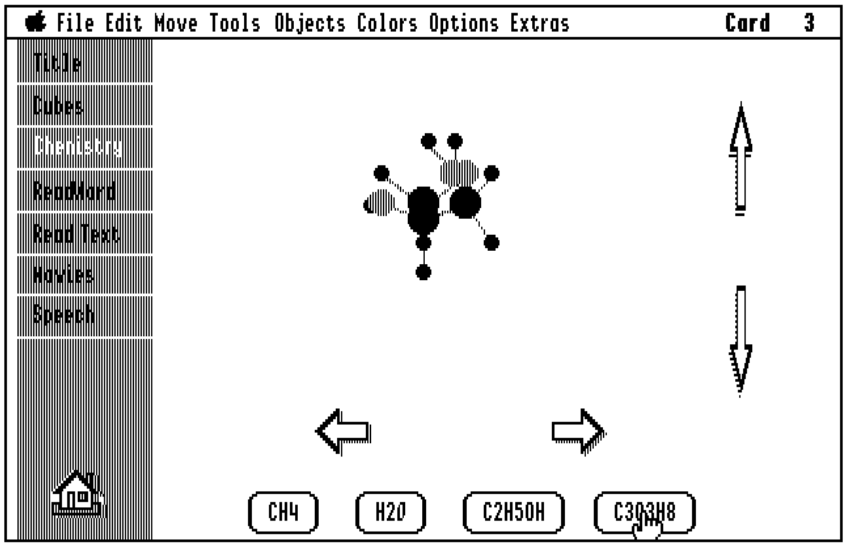
First, you can slow down, take the section very slowly, and try a lot of things as you go along. Read the reference section on each new command carefully. Read the introductory sections in the reference manual about lists. Try a lot of things as you see new ideas and commands. You might spend several hours, or even a weekend or two, just on this example, but you'll learn a lot about programming in general and Logo in particular.

The other thing to do is just read through the section, doing exactly what the instructions say. You'll still see some neat demonstrations, and learn a few things along the way. Later, when you're more experienced, you can come back to this section and try again.

Trying the Chemistry Card

From the HyperLogo Demo stack, click the Chemistry button. You'll see a series of molecules along the bottom. Click on the one labeled C3O3H8. What you see is a true 3D rendering of $C_3O_3H_8$. In plain English, it's a particle of a liquid chemists call glycerol.

When you draw this on your own computer, one of the things you'll notice right away is that the picture is in color. It's not using the 3D glasses, but the program that draws the molecule does use the 3D turtle. In fact, without the 3D turtle, it would be a *lot* harder to draw this picture.



When you drew cubes and octahedrons in 3D, changing the direction of the turtle rotated the object the turtle drew. The same thing happens with this demonstration. If you want to get a better view of the overlapping oxygen atoms (the red ones), just rotate the turtle and try again. There are some rotation buttons along the right and bottom (they have arrow icons) that will rotate the molecule.

A quick look at the script shows that the buttons are just changing a heading variable and calling a subroutine to redraw the current molecule.

There are a total of four molecules to pick from. CH_4, methane, is the first one you saw when you got to the card.

How the Program Works

While we won't go through this program line by line, there are some interesting things you can learn about Logo, scripts, and lists from the program, so we'll look at a few procedures.

Logo uses lists a lot. Because of this, one of the things Logo can do easier than a lot of languages is to treat the contents of a variable as a program. That's an important part of creating this molecule with a short Logo program.

The chemistry program actually uses a separate procedure for each atom. For example, an oxygen atom is a list with O, for oxygen, as the first thing in the list, and two other lists for the two other atoms. The reason there are two connections is that oxygen can form a chemical bond to two other atoms. Hydrogen is an H with a single list, since hydrogen only forms one chemical bond. Starting with the oxygen atom, water (H_2O) looks like this in Logo:

```
[O [H []] [H []]]
```

Each of the hydrogen atoms has an empty list. That's because the atom the hydrogen bonds to has already been taken care of, and we don't need to draw it twice. If we started with one of the hydrogen atoms, instead of the oxygen atoms, we would write

```
[H [O [H []] []]
```

This time, oxygen uses one empty list, since it's already connected to the hydrogen.

Exploring

With this in mind, entering new molecules is easy. The chemistry program only has carbon (C), oxygen (O) and hydrogen (H), and it only handles single bonds, but there are a lot of molecules that will work even with just these three atoms.

The first step in creating a new molecule button is to take a look at the ones that are already on the card. Editing one, you can see that the molecule buttons set the variable Current to a new molecule list, then draw the molecule by calling Update. All you have to do to create a new molecule is to make up a new button, set the molecule list to represent your molecule, and call Update in your own button.

You can find some sample molecules in a chemistry book.

The procedure that eventually gets called to draw an oxygen atom takes two parameters, one for each atom that attaches to the oxygen atom.

```
   TO O :b1 :b2
```

Next, the procedure draws the atom connected by the first chemical bond. Here's the whole line that does the work:

```
   IF NOT EMPTYP :b1 [SETPC :lcolor BK :sep RUN :b1 PU FD :sep PD]
```

The first part of this line is checking to see if the parameter is the empty list. The `IF` statement evaluates a condition. In this case the condition is `NOT EMPTYP :b1`. `EMPTYP` is a procedure that checks to see if a value is an empty list. In this case, `EMPTYP` is checking to see if the parameter `:b1` is an empty list. If it is, then `NOT EMPTYP :b1` will be false, and the `IF` statement will skip ahead to the next line in the procedure. If the parameter isn't the empty list, `NOT EMPTYP :b1` will be true, and Logo will do the stuff in the brackets.

Inside the brackets, most of the commands should look pretty familiar. Taking out one of the commands for a while, you get

```
   SETPC :lcolor BK :sep PU FD :sep PD
```

These commands just draw the line that connects the atoms. Lcolor is a variable that holds the color used to draw the lines, and sep is a variable that determines how far away the atoms will be – their separation, which is where the variable name comes from.

The really neat part of the line is `RUN :b1`. This runs the contents of the list you pass the oxygen atom as if it were a program instead of a list in a variable. In our water molecule example, what it runs is

```
   H []
```

This calls another procedure, `H`, to draw the hydrogen atom. It could just as easily call some other atom which was hooked to still other atoms. You could even write a program that creates the molecules on the fly, then draw the molecule by running the list you build in the program!

The next line of the oxygen procedure draws the second atom, rotating 107° first, since atoms connected to an oxygen molecule are separated by about that angle.

```
   IF NOT EMPTYP :b2 [RT 107 SETPC :lcolor FD :sep RUN :b2 PU BK
:sep PD LT 107]
```

Finally, just before it leaves, the procedure calls another procedure to draw a ball with a radius of 10 pixels (20 pixels across) and a color of 2.

```
   AddBall 2 10
   END
```

There are some tricks in the program. The biggest problem in drawing the molecule isn't handling all of the rotations to connect the atoms. In fact, in Logo, the only real problem is making sure the atoms that are closest are drawn last, so they are on top of the ones that are farther away. The chemistry program handles this problem by building a list of the atoms as the original molecule is traced. Each time `AddBall` is called, the position, color and size of the ball is stored in the list. Once the lines connecting the atoms are drawn, and all of the atoms in the molecules have been traced, the program goes back and looks for the atom that is farthest away, and draws it first. It then moves up to the next farthest away, drawing that atoms, and so on until all of the atoms are drawn. If you would like to explore this part of the program, look at `DrawAtoms` and `DrawFarthest` in the chemistry program.

The other trick that's used is defining the procedures in the first place, and drawing the methane molecule you see when you go to the card. This card has an auto-activate button in the upper right corner of the card. This button defines all of the procedures used by the other buttons, sets the initial molecule to methane, sets the initial rotation angles, and then draws the starting molecule. The rest of the buttons just change the molecule or rotation angles, then redraw the molecule. Almost all of the work is hidden in the auto-activate button.

You can look at the script several different ways. The easiest is to hold down the command key and click near the top right corner of the card, even though it doesn't look like anything is there.

Exploring The chemistry program is pretty simple, but it's also easy to expand.

For example, adding new atoms is easy. Chemically, sulfur (S) works pretty much like oxygen. To add sulfur to the program's bag of tricks, start with oxygen, but change the name to S and make the atom yellow.

The biggest limitation of the chemistry program for real molecules is that it doesn't handle double bonds. A double bond happens when, for example, an oxygen atom hooks up with another oxygen to form a molecule, and both of the chemical bonds are used. You can handle this easily with a separate procedure, perhaps called O" for oxygen with a double bond. In this case, the procedure would only need one parameter, like hydrogen, since it will only connect with one other atom.

Chapter 4 – Making Movies with HyperLogo

Making Movies is Easy

Making movies is one of the easiest things you can do with HyperLogo. From classroom demonstrations to visualizing an object to grabbing bragging rights at the local computer user's group, making movies is a great way to go.

A Finished Movie

Let's start by looking at a very simple movie in the HyperLogo.Demo stack. Click on the Movies button and then press Play. You'll see a very simple animation of a flag.

In a way, movies are like animations in HyperStudio, but there are some major differences. The biggest difference is that animations show an object moving across the card. In fact, the movie you just ran could have been done with an animation. The differences are that movies can easily change the shape of what you see from frame to frame, and it's easy to create movies from shapes in Logo.

Creating Your First Movie

We're going to learn about movies in HyperLogo the easy way: by making one! Get into HyperLogo by editing a button script or creating a new button script, then open a text window. Clean out the workspace with

```
ERALL
```

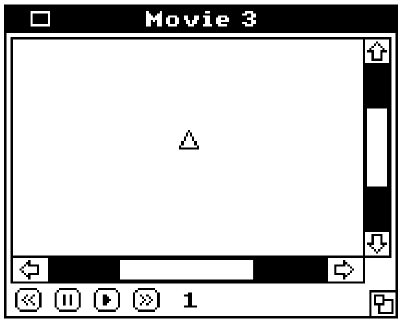
Open a turtle window and enter the Flag procedure from Chapter 2. Actually, this one is a little simpler.

```
TO Flag :s
FD :s
REPEAT 4 [FD :s RT 90]
BK :s
END
```

Try the flag routine to make sure it works:

```
FLAG 20
```

Now close the turtle window and open a movie window. To open a movie window, pull down the File menu and select New Movie Window.



What you get looks a little intimidating at first, but all a movie window really is is a fancy turtle window. You can draw in the movie window the same way you draw in a turtle window. Give is a try by typing

```
SETPC 4
FLAG 20
```

Let's stop for a moment and think about how movies work in a movie theater. What you are really seeing on the screen is a series of still pictures, shown one after the other. If they are shown quickly enough, and if the difference between each picture is small and continuous, your brain tells you you're seeing motion. That's the way the movie window works, so what we need to do is create a few more frames.

To add the next frame, select the movie window, then pull down the Movie menu and select Add Frame After. The flag you just drew disappears! If you look down at the bottom of the window, though, you'll see a 2. That tells you you're on frame 2, which doesn't have a drawing, yet. Click on the « button, and you'll move back to frame 1, where the flag is still safe and sound. Click on the » button, and you'll be back at the new frame.

The movie we're going to create is a pretty simple one, with the flag twirling around in a circle. To draw the second frame, type

```
LEFT 45
FLAG 20
```

Add one more frame, but notice where the turtle is. It's still rotated 45° to the left. That's an important characteristic of the movie window we'll use to our advantage, now, as we add the next six frames of the movie.

```
REPEAT 6 [FLAG 20 LEFT 45 ADDFRAME]
```

Our first movie is almost complete, but there is one last step. Our repeat loop left us with a blank frame at the end of the movie. To get rid of it, you can either type

```
DELETEFRAME
```

or you can select the movie window, pull down the Movie menu, and select Delete Frame.

Playing the Movie

The two controls we haven't used are play and stop. To play your movie, select the movie window, then click on the play button, ▶. To stop the movie, click on the stop button, ⏸.

Exploring This movie only has 8 frames, so it's a little jumpy. You can create a movie with more frames, and get smoother motion, by using `360 / :frames` for the rotation angle, where `frames` is the number of frames you will use. The movie will draw a lot faster if you stick to integer angles and hide the turtle first.

For example, here's the commands to create a movie with 18 frames:

```
HIDETURTLE
SETPC 4
REPEAT 18 [LT 20 FLAG 20 ADDFRAME]
DELETEFRAME
```

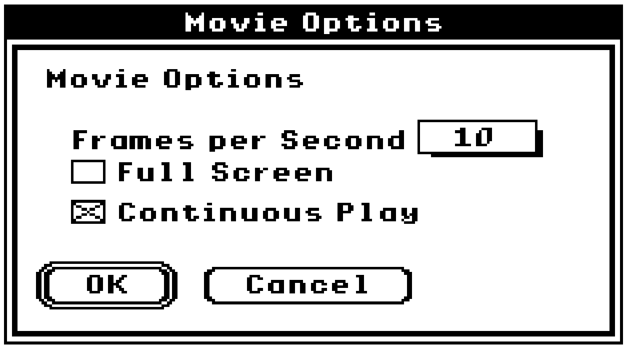
Of course, you'll want to do this in a new movie window. Be sure and close your old movie window first, too.

At some point, you'll run out of memory as you try to add more frames. Movies take lots and lots of memory. That's why it's a good idea to close all of your movie windows before you start a new one.

To get an idea just how many frames you can create with the memory you have, keep cranking the number of frames up in this example until you get an out of memory error.

Movie Options

Click on your movie window, then pull down the Movie window and select Movie Options... You'll see a dialog like this one.



Frames per Second 10
☐ Full Screen
☒ Continuous Play

Frames per Second controls the speed of the movie. You pick the number of frames you want the computer to show each second. The movie player can go as fast as 30 frames per second, which is the same speed your television uses. To get a full 30 frames per second, though, you'll have to keep the movie window small, like it starts, and you'll have to have an accelerator card. If the computer isn't fast enough to draw the frames as fast as you want, Logo will skip frames to keep the movie playing at the correct overall rate.

The big problem with a fast frame rate isn't the speed of the computer, though, it's the amount of memory you have. If your goal is a 3 second movie, and you're using the default rate of 10 frames per second, you'll need 30 frames – about 1 megabyte worth of pictures. If you pick 30 frames per second, to get the same 3 seconds, you'll need three times as much memory.

Below 10 frames per second, most pictures look so jumpy that they look more like a fast slide show than a movie. That's OK, though – these slow rates are handy when you're fine tuning a complicated movie.

When you're playing a finished creation, you might not want to see all of the Logo windows in the background. Try picking Full Screen, then click OK, and finally, click the play button in the movie window. What you get is a full screen movie.

Of course, the Stop button is gone, so there's no obvious way to stop the movie. Holding down the open-apple key and pressing period (.) does the trick, though.

The last option is Continuos Play. When this option is selected, the movie starts playing again as soon as it is finished. This is a great way to get a lot more time from a simple movie. When you first ran the flag movie, for example, you probably didn't notice that it took less than one second to run! You watched a continuos movie until you got tired of it, then went back to the book.

Playing a Movie On a Card

Once you've created a movie, you'll want to play it from a button on a card. The first step is to save the movie you've created to disk. Click on the movie window and save you're flag movie to a file called Flag.

Back in the script, you need to do two things to play a movie. The first is to open the movie, which loads the movie into memory and tells HyperLogo about the movie. Here's the command to load your flag movie:

```
OPENMOVIE "Flag [-50 50 50 -50] [-40 40] 1
```

The first parameter is the name of the movie. Generally, you'll want to keep your movies in the same folder as the stack that uses them, and use the file name for the movie. If you know how to use GS/OS path names, though, you can. With GS/OS path names, you can put the movies just about anywhere.

The next parameter is a rectangle that tells what part of the movie you want to see. Your movie window is actually as big as the computer screen. Using turtle coordinates, it stretches from -160 on the left to 160 on the right, and from 100 at the top to -100 at the bottom. In general, you just want to play a small part of the movie. The list contains four values that pick the left, top, right and bottom sides of a rectangle from the original movie window. In our case, the flag was 40 units long, but we're adding a little extra area for the cases when the flag is tipped a bit, and extends just outside of a box extending 40 units in each direction from the center.

The next list tells Logo where to put the movie. It's still in turtle units, with 0, 0 at the center of your card. The first value sets the left edge for the movie, and the second value sets the top edge.

The last value is a flag. If you us 0, HyperLogo plays the movie once, then stops. That's how the American Flag movie from the demonstration stack works. In this case, we used 1, so HyperLogo plays the movie over and over. You stop the movie by clicking anywhere with the mouse, or by pressing any key.

Movies and Other Programs

You can load and save movies just like you can load and save the contents of any other Logo window. Logo stores movies on your disk using the popular Painworks animation format. That means you can load and play Logo movies with just about any movie player, and you can load and play a lot of the movies you'll find on online services in Logo.

Movie Samples

You already know everything you need to know to create movies with Logo. Before moving on, though, let's take a look at a few other examples that show how versatile movies can be.

▲ **Warning** Some of these movies use a lot of memory. You may need to change the number of frames they create to get the movies to work on your computer. ▲

The Cube Movie

The cubes movie shows two cubes rotating at different speeds. The cubes are in three dimensions, too. To create the movie, open a movie window, then load the file CubeMovie from your samples folder. Select all of the text and press RETURN to enter the procedures in your workspace, then type

```
CubeMovie
```

This movie shows you one way to handle multiple objects in the same movie. In this example, the movie is completely created with one of the two cubes. The program then backs up to the start of the movie and moves through the frames again, drawing the second cube. That way, you only have to keep track of one object at a time.

This movie also shows how well movies and the 3D display work together. Movies or 3D separately are powerful tools for showing an object in space, but when they are combined, the effect is a lot more realistic. Once you add motion, even lines and points get easier to see in three dimensions.

Spinning Octahedrons

This movie shows two octahedrons. It uses the same technique as the cube movie to show two different objects in the same movie. This movie shows two new kinds of motion, though.

To create the movie, create a new movie window, load OctaMovie from your samples folder, and type

```
OctaMovie
```

All of the movies you've created before this one are made by rotating an object around some point, so the object sits at one place in space and turns. The larger of the two octahedrons in this movie does the same thing, but it's on a tether, so it's rotating in a circle around a center point it never touches. It's as if you spun the object from an invisible string.

The second octahedron doesn't spin at all. Instead, it moved through the picture, diving through the circle created by the first octahedron. It shows a good way to handle motion, starting from a point off of the screen, moving through the field of view, then vanishing off the other end of the screen. (If you make the movie display too large, you can see the entire path – the idea is to make the movie window big enough to see the circle formed by the first octahedron, but no bigger.)

Spinning Molecules

This time, you'll create a movie from an older example. Start by closing all of the windows except the script window and the text window you are typing in, and clear the workspace with

```
ERALL
```

Open a new movie window, then load the chemistry file with

```
LOAD "Chemistry
```

This file is the original set of Logo procedures that were used to create the Chemistry card in the HyperLogo.Demo stack. Select all of the text and enter the procedures into the workspace by pressing RETURN.

With a complicated molecule like $C_3O_3H_8$, it's tough to see the exact relationship of the different atoms. A movie makes it easy. Here's all you need to create a movie of the molecule:

```
RT 30
REPEAT 18 [HT Draw "C3O3H8 RLL 20 ADDFRAME]
DELETEFRAME
```

Exploring You may already see some of the possibilities for movies. One is to show chemical reactions. Instead of rotating a molecule in the middle of the screen, you can move two molecules from off of the screen to the center, flash a red screen, then move the new molecules that are created in the reaction off of the screen.

Chapter 5 – Talking Logo

What You Need

If you have our Talking Tools software package, HyperLogo can talk. Anything you can type, Logo can read back to you using standard English pronunciation. If you don't have Talking Tools, HyperLogo ignores all of the commands in this chapter. It won't do any harm to use them, but the commands won't do anything, either.

If you haven't installed talking tools, one way to install them is to run HyperInstaller from Logo and pick the script "Install Talking Tools".

A Quick Demonstration

Open the HyperLogo.Demo stack and click "Say It." HyperLogo will read whatever is in the text box just above the button. You aren't likely to get the voice mixed up with a human voice, but it is clear and easy to understand.

There are several options you can use to control the quality of the voice. The one that makes the most difference is switching between a male and female voice. The first voice you heard was the male voice. Click on "Female Voice" and try again, and you'll hear a distinctly feminine voice.

Why Not Use a Recording?

There is one very important thing to remember about Talking Tools. The tools are really converting text into sound, not using a recorded voice. HyperStudio can record your voice and play it back, and if you need to use a short, preset sequence of words, that's the way to do it. A recording of your own voice will sound a lot better than Talking Tools.

There are two big disadvantages to recording your voice, though. (Besides stage fright, that is.) First, recorded voices take up a lot of room on disk and a lot of room in memory. Once Talking Tools are loaded, the only additional memory you need is the memory to hold the text. That isn't much. For example, to say "I can talk!" Talking Tools needs 11 bytes of memory to store the text. (If you don't know what a byte is, don't worry. Think of them as buckets. The issue is the relative number used by the two methods.) If you record your voice at a decent rate of, say, 10 kilohertz, you will use about 8000 bytes to read the same text!

The other big disadvantage to a recording is that it's fixed. You can't change what is said without making a new recording. In a talking storybook, for example, you could ask the user for a name, and insert the name in the story. Talking Tools can read the name, but with a prerecorded voice, you can't do the same thing.

Making Logo Talk

Logo's SAY command will take any word and say it. It's really that simple.

```
   SAY "I\ am\ completely\ operational,\ and\ all\ my\ circuits\
are\ functioning\ perfectly.
```

Teaching Logo to Pronounce Words

The voice sounds mechanical, but most of the words are pronounced clearly. There is one exception, though. When the computer reads "circuits", it sounds more like it is spelled – serkewits. A pair of commands sets this straight. PHONETIC takes a word as input and converts it to special phonetic symbols. You really don't need to worry about the phonetic symbols much, since you will almost always feed the output from PHONETIC straight to the DICT command, which is a phonetic dictionary. The DICT command takes two words as input. The first is the word you want to teach Logo to say, and the second word is the correct phonetic spelling for the word. The practical way to put these commands to use is to spell the word the way it sounds, using the PHONETIC command to convert this to phonetics, then pass the result straight to DICT.

Here's a line that will teach Logo to say circuits properly.

```
   DICT "circuits PHONETIC "serkits
```

Try this, then ask Logo to read the original line again.

Exploring While you probably don't need to know about the phonetic symbols used by Logo, you might enjoy learning about them. Using phonetic symbols, you can control the way Logo talks in very subtle ways. The phonetic symbols used by Logo are described in the Talking Tools manual.

Male and Female Voices

By default, Logo uses a male computer voice. It's easy enough to switch genders:

```
   VOICE "female
   SAY "My\ circuits\ are\ doing\ fine,\ too!
   VOICE "male
   SAY "I\ noticed.
```

Speed, Volume and Pitch

Three other commands fine tune the voice. For all three commands, you can use a value of 1 to 9, and the default is 5. If you use a number outside of this range, Logo pins the value to 1 or 9, whichever is closest.

`SPEED` controls how fast Logo reads the words. The larger the number, the faster Logo talks.

```
SPEED 9
SAY "That\ was\ fast!
```

`PITCH` changes the pitch. A higher number raises the pitch, making the voice sound a little higher. Raising the pitch of the male voice, for example, makes it sound a little more like a female voice.

`VOLUME` controls how loud the voice is. With a higher volume, Logo shouts louder, and with a lower one, the voice is quieter.

Chapter 6 – Controlling HyperStudio from HyperLogo

How HyperLogo and HyperStudio Work Together

The button scripts we've looked at so far all have one thing in common: they only do one thing. You push the button, and the script takes some predetermined action. True, the chemistry example took this idea to an extreme by tying several different buttons together, communicating with variables to they all did something different to the environment, but each button still only did one very specific, predetermined thing.

This chapter shows you how to go beyond predetermined actions, and also how to cause Logo to make changes in the stack itself. You'll learn how to read text with a dialog from Logo, and how to make callbacks. You'll see a working example of a callback that reads a text field. Other callbacks, which are just as easy to use, will let you set text in a text field, move from card to card, hide or show objects, and so forth. In fact, you've already used one card that read a text field. The Talking Tools demonstration literally read the text, too, reading it out loud!

Using READWORD For Text Input

The quickest and easiest way to get information into a stack is with `READWORD`. If you're working from a text window, `READWORD` stops until you type a line of text, and converts what you type into a Logo word. When you use `READWORD` from a script, though, it works a little differently. HyperLogo realizes that there isn't a text window to read a line of text, so it brings up a dialog instead. It's a simple dialog – you get a ? character as a prompt, an EditLine item to enter the text, and an OK button to click on when you're done. All of this is pretty standard fare in any Apple IIGS program. Whatever you type into the EditLine item is converted to a Logo word, and returned as the result to `READWORD`. From there, you can use the word itself, or break the word up with `PARSE` and use Logo's powerful list manipulation commands to work with the text.

Exploring You can use the `READWORD` dialog from a text window, too, but it takes a little extra work. You have to use the `TURTLEIO` command to let Logo know you want to use the dialog instead of reading text from the text window.

Changing the Prompt

A question mark for a prompt? Well, that works – but it's pretty lame. `SETRWPROMPT` let's you change the prompt, though. It takes a single word as input, and uses that word as a prompt.

Keep in mind that we're talking about a Logo word here, not an English word. A Logo word is any number of characters, and can contain a bunch of English words. In most cases, you'll use a full English sentence, but stuff it into a single Logo word, like this:

```
SetRWPrompt "How\ old\ are\ you?
```

The prompt you set with SETRWPROMPT will be used the next time you call READWORD, then READWORD reverts to the default question mark prompt. That's no problem if you use the two calls as a pair. In most cases, you'll set the prompt with SETRWPROMPT right before you call READWORD.

△ **Important** If the dialog isn't wide enough to display the entire prompt, it's chopped until it fits. △

Changing the Dialog's Position and Size

You can change the position and size of the dialog, too. SETRWRECT uses a rectangle to set the position and size of the dialog. You pass a rectangle, which is a list of four coordinates giving the dimensions of a box on the screen. You can make the dialog any size you like, but it will look best if you make the height of the rectangle 63.

Like all rectangles in Logo, you use turtle coordinates. The list is in this order: [left top right bottom].

An Example of READWORD in Action

Putting all of this together, here's a set of three Logo commands that set up a READWORD dialog:

```
SetRWPrompt "Hi!\ \ What's\ your\ name?
SetRWRect [-136 54 140 -9]
Make "theName ReadWord
```

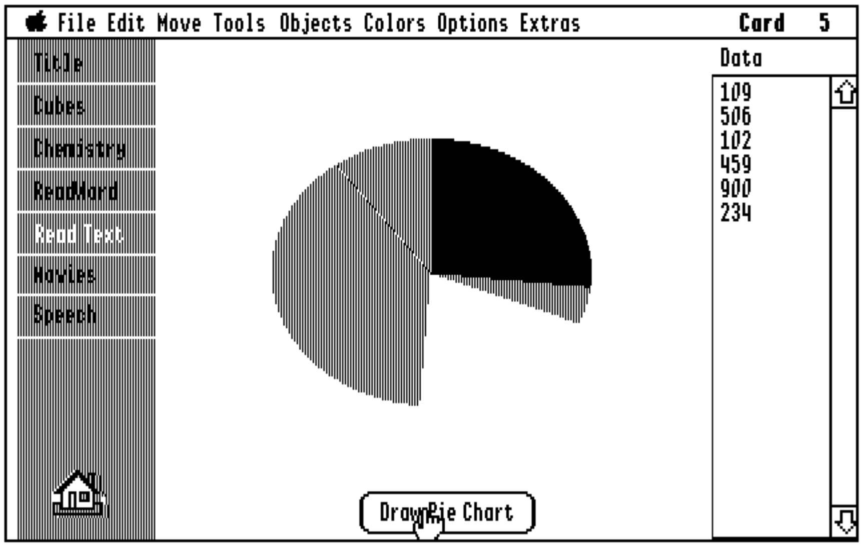
To see these lines in action, bring up the HyperLogo Demo stack and try the ReadWord card. These lines are taken directly from the script for that card. The script itself is in an auto-activate button located at the top right of the card. To edit the script, hold down the command key and click in the top right corner of the card.

Reading Text from a Text Item

Up until this point, everything we've done from Logo could have been done from a stand-alone version of Logo. In fact, all of these things are perfectly possible from 3D Logo. The only difference between the two versions of Logo is that we've executed our Logo programs by clicking on a button.

We can move beyond traditional Logo programs with callbacks. Callbacks are subroutines inside of HyperStudio itself that we can call from Logo. Each callback tells HyperStudio to do something, like move to another card or hide a button.

Let's start by looking at a sample card that shows one of the most common uses for a callback: reading the text from a text item. Run the HyperLogo Demo stack, and move to the Read Text card. When you press the Draw Pie Chart button, you'll see a pie chart like this one:



```
109
506
102
459
900
234
```

Well, that's pretty, but we could do that before. The difference is that HyperLogo got the information for the pie chart from the text item on the right side of the window. Change one of the numbers – or even add or remove one – and draw the pie chart again. The new pie chart plots whatever values you enter.

The call to read the text from the text item is really pretty simple. It's called `GetFieldText`. It returns a Logo word containing all of the characters in a text field. Of course, a card can have more than one text field, so you need some way of telling `GetFieldText` just which field to get the text from. `GetFieldText` takes two parameters. The first is the name of the card, or the empty list if you want to read a field from the current card. The second input is the name of the text field.

Naming cards and text fields is something you may not have done before, since the main reason you'd want to name one is to give HyperLogo some way of telling which card or text item you want to look at. It's not hard to give a card or field a name, though.

To name a card, pull down HyperStudio's Objects menu and select Card Info... One of the fields in the card information dialog is called Card name. Type the name in this EditLine field, just like you'll type it in HyperLogo. In our example, though, we just used the empty set (which is two brackets with nothing inside, like this: []), since we were reading text from a text field on the current card.

The text field itself needs a name, too. There's a field for the name of the text item in the dialog you use to create it or change it's characteristics.

If you edit the script that draws the pie chart, you'll find the call to `GetFieldText` down near the end. Here's the line:

```
PieChart [-60 50 60 -50] Parse GetFieldText [] "Data
```

Besides the call to `GetFieldText`, there are two interesting things about this line. `PARSE` is one of Logo's list manipulation commands, and it makes the job of using the text from a text field very easy. `PARSE` breaks the text from the text field up into individual Logo words, returning a list with all of the words. The other neat feature of this line is the `PieChart` procedure, which is defined in the script itself. If you ever need to draw a pie chart yourself, you can copy all of the procedures from this button script into your own stack, instead of doing all of the work to create a pie chart for yourself.

The Callback Library

I glossed over one very important detail when we looked at the pie chart example. `GetFieldText` isn't built into the Logo language. It's a procedure that uses some very low-level Logo calls that make HyperStudio callbacks. The low-level calls that are built into HyperStudio are designed for advanced programmers who know exactly what they are doing.

Fortunately, you don't have to know how to use the low-level callback commands to talk to HyperStudio. Chapter 9 lists the names and parameters for a series of procedures that make all of the common callbacks. These have been written for you. All of the procedures listed in Chapter 9 are in a file called CallBacks. This file is in the Samples folder on the HyperLogo disk. If you used the installer to install the samples, you'll find it in the same folder as the HyperLogo.Demo stack, too. The easiest way to use them is to open the CallBack file from HyperLogo, copy any of the procedures you need, and paste them into your script. That's how the pie chart card was created. You can find the `GetFieldText` procedure at the top of the script.

Exploring `GetFieldText` is just one of the many callbacks listed in Chapter 9. It's just as easy to set the text in a text field with `SetFieldText`. It's even easier to move to another card. You can hide items or move them around, too.

Even if you aren't going to take the time to learn all of the Logo language commands, be sure you browse through the callback chapter to see what's available. It's the callbacks that let you read and set HyperStudio's fields and cards, and that's the heart and soul of scripting.

Chapter 7 – Desktop Interface Reference

This chapter describes the desktop interface used by HyperLogo. Each of the menu commands is described. See Chapter 8 for a detailed description of the Logo language.

After an initial discussion of how you get to Logo, this chapter is organized by menu, with each command from each menu listed in order under the menu heading.

There are five kinds of windows used by Logo. The use of all of the windows except the script window is described under the various New commands. The new commands, in turn, are described under the File menu. The script window is described in the section "Script Window," right after edit windows.

In several places, this documentation refers you to Apple's System Disk manuals. These are the manuals that came with your computer. Many introductory Apple IIGS books cover the same material.

Apple Menu

About HyperLogo...

The About alert shows the copyright notice and version number for HyperLogo. The version number is something you should check, especially if you are about to call for assistance with HyperLogo.

Desk Accessories

Any remaining items in this menu are desk accessories. You can find a general description of desk accessories in Apple's System Disk manuals. For specific information about a particular desk accessory, see the documentation that comes with the desk accessory.

File Menu

New Text Window

The New Text Window command opens a new text window. A text window is a standard edit window, as described in Apple's System Disk manual, with one twist: The RETURN key has special meaning. When you press the RETURN key, Logo executes the line the insertion point is on, writing any output at the end of the window. If you select some text, then press RETURN, Logo executes all of the text, even if you have selected several lines.

With that exception, text windows are just TextEdit windows. You can scroll them, change the size of the window, select text, or edit text using the standard editing commands described in Apple's System Disk manual.

When you save a text window, Logo saves the file as an Apple IIGS language file. It uses a file type of $B0 (SRC) and an auxiliary file type of $0112 (Byte Works Logo).

Logo opens a text window any time you use the Open command to open any form of ASCII file.

Tip Text windows are a great way to explore the Logo language. You can try things quickly, even developing procedures interactively, without changing a script button to see each new change. When you're finished, you can apply the things you learn to the script window, or leave procedures you developed in the workspace and call them from a script.

New Edit Window

Edit windows work pretty much like text windows created with the New command. There are two differences.

First, the RETURN key is not treated in a special way. The RETURN key simply splits the line, moving to the start of the new line.

You'll see the second difference when you close the edit window or select some other window. At that time, Logo runs all of the commands in the window.

Generally, you'll open an edit window from a text window with Logo's `EDIT` or `EDITS` command. Once the window is open, you can make changes to the procedures or variables you are editing, then switch back to the text window to test the changes.

Script Window

Script windows work pretty much like edit windows. The difference is that each script window is attached to a specific button, and you get one automatically when you enter Logo. When you leave Logo, if you've changed anything in the script window, you get a chance to save the button's new script.

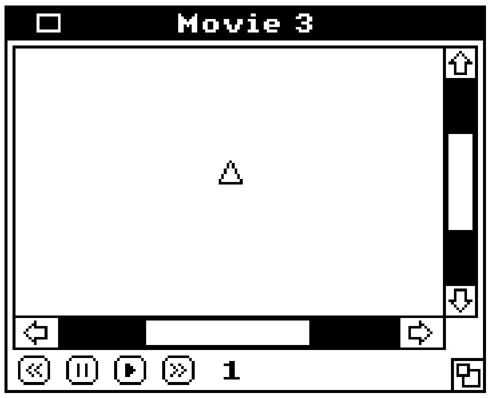
New Turtle Window

This command opens a new, blank turtle window. The turtle window is the same size, and starts off in the same position, as the card. You can use the turtle window to try your procedures that will draw something. This doesn't mess up the card, even temporarily. In addition, if you happen to cover up part of a turtle window with one of the other Logo windows, Logo can properly refresh what you have drawn in the turtle window – something that doesn't happen on the card.

When you save a turtle window, Logo uses a graphics format supported by most paint programs. Logo can open any Apple Preferred format picture, which you can use as a background for your Logo drawings.

New Movie Window

This command opens a new movie window, like this one:



The drawing area (the rectangle above and to the left of the scroll bars) looks and works exactly like a turtle drawing window. All of the Logo drawing commands work the same way.

The total size of the drawing area is one computer screen. A computer screen is 200 pixels tall and 640 pixels wide. With the default scrunch setting, this gives a range of turtle coordinates from -160 to 160 horizontally, and 100 to -100 vertically. The scroll bars can be used to display different parts of the screen, and the grow box can be used to enlarge the drawing area.

Note A full screen display is also available for the movie window, but only when you are playing a movie. See "Movie Options...", later in this chapter, for details.

The four buttons along the bottom of the window are used to move from one frame to another, play a movie, and stop a movie.

⏪ Last Frame	Move to the frame before the one that is displayed. If the movie window is showing the first frame, this button is ignored.
⏸ Stop	If a movie is playing, this button stops the movie. If no movie is playing, this button is ignored.

▶ Play	Start playing a movie. Once a movie starts, the only command the program will accept is to stop the movie. You can stop the movie by pressing the stop button or by pressing open-apple period (.).
≫ Next Frame	Move to the frame after the one being displayed. If the movie window is showing the last frame of the movie, this button is ignored.

The number to the right of the buttons is the frame number. The first frame is frame number 1. The frames are numbered sequentially.

There are several Logo commands which can add or delete frames, or mimic the operation of the buttons in the movie window. See "Movies" in Chapter 8 for details.

The Movies menu has four commands to add and delete frames and set various options that control how a movie is displayed. See "Movies", later in this chapter, for details.

Open...

The Open command brings up a standard Apple open dialog. The open dialog is described in Apple's System Disk manual.

Logo can open three kinds of files.

Pictures	Apple preferred format pictures (file type $C0, auxiliary file type $0002) are opened as turtle windows.
Animations	Paintworks animation files (file type $C2, auxiliary file type $0000) are opened as movie windows.
Text	Text Files (file type $04) and program source files (file type $B0) are opened as text windows.

Logo won't let you make a mistake and open a file it can't handle. The only time you need to worry about the actual file type is when you are creating a file from another program that you want to load into Logo.

Close

This command closes the front window. It works with all Logo windows and most windows opened by desk accessories.

△ **Important** Closing the script window is equivalent to using the Quit command. When you close the script window, Logo will start closing all of the windows that are open. It will, of course, give you a chance to save any changes made in a window before the window is closed, and you can cancel the operation at that point. △

Save

Saves the contents of the front window. If the front window has never been saved to disk, this command works exactly like the Save As... command.

Save As...

This command brings up a standard Apple save dialog, which will let you pick a location and file name to use to save a file. Apple's save dialog is described in Apple's System Disk manual.

There are five kinds of windows in Logo, and each is saved in a slightly different way.

Text Window	Text windows are saved as Logo program source files. The file type is $B0, and the auxiliary file type is $0112.
Edit Window	Edit windows are saved as Logo program source files, too. The file type is $B0, and the auxiliary file type is $0112.
Script Window	Script windows are also saved as Logo program source files. The file type is $B0, and the auxiliary file type is $0112.
Turtle Window	Turtle windows are saved as Apple Preferred picture images. The file type is $C0, and the auxiliary file type is $0002.
Movie Window	Movie windows are saved as Paintworks animation files. The file type is $C2, and the auxiliary file type is $0000.

Page Setup...

This command opens a printer page setup dialog. Logo supports any printer driver that conforms to Apple's standards. Apple's printer drivers are described in Apple's System Disk manual; for other printer drivers, see the documentation that came with the driver.

Print...

This command opens a print dialog. Logo supports any printer driver that conforms to Apple's standards. Apple's printer drivers are described in Apple's System Disk manual; for other printer drivers, see the documentation that came with the driver.

Quit

This command leaves HyperLogo. Before leaving, all windows are closed. If any window has changed since it was last saved to disk (or since it was created, if it has never been saved to disk), you will get a chance to save the file, not save the file, or stop shutting down the program.

When you quit from HyperLogo, you'll go back to HyperStudio, where you can click on the button you just worked on to execute all of the commands in the script window.

Edit

Undo

Undo exists to support desk accessories. This command is not used by Logo.

Cut

Cut is used by desk accessories and by the text, edit and script windows in Logo.

In Logo's text, edit and script windows, Cut does nothing if there is no text selected. If text is selected, Cut removes the text from the window, leaving the insertion point at the spot the text was removed from. The text is placed in the scrap. From there, you can use the Paste command to copy the text back into a Logo window, or you can use a scrapbook desk accessory to move the text to another program.

Copy

Copy is used by desk accessories and by the text, edit and script windows in Logo.

In Logo's text, edit and script windows, Copy does nothing if there is no text selected. If text is selected, the text is placed in the scrap. From there, you can use the Paste command to copy the text back into a Logo window, or you can use a scrapbook desk accessory to move the text to another program.

Paste

Paste is used by desk accessories and by the text, edit and script windows in Logo.

In Logo's text, edit and script windows, Paste starts by deleting any selected text, as if the Clear command were used. Next, Paste copies any text in the scrap into the window. The text is copied, not removed, from the scrap, so you can paste the same text several times in a window, or you can paste the text into several different windows.

Logo's Copy and Cut commands change the scrap. Desk accessories can also change the scrap.

Clear

Clear is used by desk accessories and by the text, edit and script windows in Logo.

In Logo's text, edit and script windows, Clear does nothing if there is no text selected. If text is selected, Clear removes the text from the window, leaving the insertion point at the spot the text was removed from.

Select All

This command is used with Logo's text, edit and script windows. It selects all of the text in the window.

Movie

The movie menu is only active when a Logo movie window is the front window.

Add Frame After

A new, blank frame is added after the frame that is currently displayed, then the frame is advanced so you are viewing the new frame.

Logo's `ADDFRAME` command does the same thing as this menu command.

Insert Frame Before

A new, blank frame is added before the frame that is currently displayed. The frame counter does not change, so you are left viewing the new frame.

Logo's `INSERTFRAME` command does the same thing as this menu command.

Delete Frame

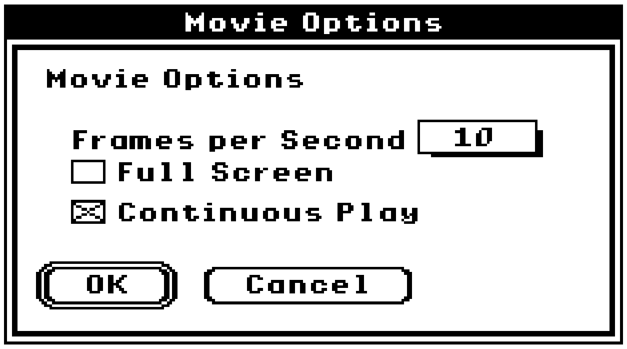
The visible frame is deleted from the movie.

This command is not available if the movie only has one frame.

Logo's `DELETEFRAME` command does the same thing as this menu command.

Movie Options...

This command brings up a dialog. You can use the dialog to select options that effect the way the current movie is played. The changes only effect the frontmost movie window.


Movie Options

Movie Options

Frames per Second 10
☐ Full Screen
☒ Continuous Play

Frames per Second 10

This pop-up menu lets you pick from a variety of preset play rates. You can play a movie as fast as 30 frames per second or as slow as one frame per second.

30 Frames per second is the same rate as a television set and most computer monitors. In some cases, the computer won't be able to keep up with this display rate, especially if your computer does not have an accelerator card, or if you are using full screen display and the picture changes radically between frames. Long movies with high speeds also take lots of memory.

The slowest frame rate of 1 frame per second is useful for checking your animations to see where the problems are.

For most cases, you should pick a frame rate from 7.5 to 15 frames per second.

☐ Full Screen

If this option is selected, movies will use the entire screen, including the area normally used by the menu bar. To stop a full screen movie, hold down the open-apple key and press the period key (.).

☒ Continuous Play

If this option is selected, the movie will play continuously. Once the movie finishes, it loops back to the first frame and keeps playing. Carefully designed movies that loop back on themselves, like the rotation movies created in Chapter 4, can play continuously and look very good, yet use very little memory.

Windows

All of your open Logo windows are listed in this menu. You can select a window, even if you can't see it, by picking the window from the list.

The window list also shows two other pieces of information. The active window – the one that will be saved, and if it's a text or edit window, the one that will get the results of your typing – has a check mark beside it. The current drawing window – the one Logo will draw in when you use any drawing commands – has a diamond.

Note The current drawing window is always the last drawing window that was a front window. So, to change the drawing window when you have more than one open turtle window or movie window, select the window you want to draw to. Selecting a text or edit window will not change the current drawing window.

If there is no active drawing window, HyperLogo draws to the HyperStudio card.

Chapter 8 – HyperLogo Language Reference

Command Descriptions

This chapter is a technical reference to the Logo language. Each command is covered. The commands start with a description of the command that includes the name of the command, along with any parameters. Anything you have to type exactly as shown will be shown in uppercase letters. In the spots where you can substitute a value, you'll find a descriptive parameter in lowercase letters. If there are any punctuation marks, like [or] characters, they must be typed exactly as you see them.

Here's a typical example:

```
BUTLAST object
BL object
```

In this case, there are two names for the command, something that is very common in Logo. `BUTLAST` is a longer, descriptive name that tells you pretty much what the command does: it returns everything but the last element of a list, or all of the characters except the last one from a word. `BUTLAST` is used a lot, though, so to save typing, you can also use `BL`.

You'll find a detailed description of the command right after this header. The command description will tell you exactly what the command does, and what limitations it has.

If the command description doesn't show an example, the last thing in the entry for the command is a code snippet. The idea is to show you at least one example of the command in an actual line of Logo. This is usually just a single line of Logo, although a few useful Logo procedures are sprinkled here and there in the code snippets.

Finding Commands

There are three ways to find a command.

This chapter is organized by category, with commands for a category listed alphabetically within the category. That's a handy way to list the commands when you're looking for a command to do something, but you're not quite sure what the command will look like. For example, if you're looking for a drawing command, you can look in the turtle graphics section.

Scanning through the whole chapter is a little tedious if you have some idea what command you are looking for, but need to be reminded of the name. You might also remember the name, but want to check on exactly what the parameters are. Appendix A lists just the command headers, while the table of contents has an outline of this chapter by command and category.

Finally, if you know the name of a command, and you're looking for the complete description, look in the index. All of the commands are listed there, and the starting page number for the technical description is shown in bold. Some commands are used in the tutorial section, too; you might find some useful tips there that aren't obvious from a technical description.

Procedures

COPYDEF

```
COPYDEF old-name new-name
```

`COPYDEF` creates a copy of a procedure. The new procedure has the same parameters and statements as the old procedure, it just has a new name.

Snippet

```
COPYDEF "Square "Polygon
```

DEFINE

```
DEFINE name list
```

`DEFINE` creates a new procedure. Once a procedure is created, it really doesn't matter whether it was created with `DEFINE` or with `TO`. The difference is that `DEFINE` works better when you want to create a procedure from inside another procedure, while `TO` works better if you are typing in a procedure in a text or edit window.

`name` is the name of the procedure.

`list` is a list of lists. The first element of the list is a list of parameters. If there are no parameters, you can use the empty list. The rest of the lists are the lines that make up the procedure.

For example, to create the procedure

```
TO Flag :size
FD :size
REPEAT 4 [FD :size RT 90]
PU
BK :size
PD
END
```

using `DEFINE` , you would use

```
   DEFINE "Flag [[size] [FD :size] [REPEAT 4 [FD :size RT 90]]
[PU] [BK :size] [PD]]
```

DEFINEDP

```
DEFINEDP name
```

`DEFINEDP` returns `TRUE` if name is the name of a procedure created with `TO` or `DEFINE`, and `FALSE` if it is not.

Snippet

```
IF NOT DEFINEDP "Flag [DefineFlag]
```

PRIMITIVEP

```
PRIMITIVEP name
```

`PRIMITIVEP` returns `TRUE` if `name` is the built-in procedures, and `FALSE` if it is not.

Note **Primitive** is the technical name for one of the procedures that are built into the Logo language. `PRIMITIVEP` gets it's name from this technical term.

Snippet

```
IF PRIMITIVEP :command [PR [Can't TO that!]]
```

TEXT

```
TEXT name
```

`TEXT` returns the procedure `name` as a list. The list uses the same format as the input list for `DEFINE`.

Snippet

```
PR TEXT "Square
```

TO

```
TO name parm1 parm2 ...
```

TO is a special command that lets you type in new procedures. The first line is a description of the procedure. It gives the name of the procedure, and lists the parameters. You don't have to include any parameters, but there is no limit to the number of parameters, either.

Here's an example of TO. This line defined a procedure called Greet that has a single parameter.

```
to Greet :who
```

The lines that actually get executed come right after the line with TO. END marks the end of the procedure. Here's what you would type to finish up the Greet procedure:

```
SHOW "Hi
SHOW :who
END
```

Once the procedure is in, you can use it over and over with different parameters, like this:

```
Greet "Fred
Greet "Sam
```

Parameters are local variables, but other than that they work just like any other variable. When you call the procedure, the value you type after the name of the procedure is assigned to the parameter. It works as if the procedure started out with the lines

```
LOCAL "who
MAKE "who "Fred
```

Other than available memory, there is no limit to the number of procedures you can define. There are also no restrictions on the way you can use your procedures. They work just like the built-in procedures, like SHOW. Of course, you can call your procedures from inside another procedure or from any button script, and you can even have a procedure call itself.

Note You can define a procedure in several different places. You can use TO to define a procedure from within a script, you can use a text window and enter the procedure interactively, or you can use an edit window to create the procedure. In all cases, the procedure is placed in a workspace that is used throughout your stack. Once a procedure is in the workspace, it can be used from any button script.

There is one restriction you have to keep in mind when you pick a name for your procedure: It can't have the same name as one of the built-in procedures. Other than that, anything Logo will accept as a word works as a procedure name.

Variables

LOCAL

```
LOCAL name
```

`LOCAL` creates a local variable. Local variables exist only inside of a procedure, and vanish as soon as the procedure finishes executing, but with that exception, they work just like any other variable.

Using local variables makes it easier to create procedures you can reuse, moving them from one stack to another, since they don't interact with the rest of your buttons in unexpected ways. Ideally, your procedures should only use local variables and parameters. If you follow that rule, the only problem you have to worry about when you move one of your procedures into a new stack is whether you already have a procedure by the same name.

Once you create a local variable, commands the deal with variables will find your local variable first. For example, let's say you create a global variable called Fred, and identify Fred as a dog.

```
MAKE "Fred "dog
```

Now suppose you run this procedure:

```
TO try
LOCAL "Fred
MAKE "Fred "hound
SHOW "Fred
END
```

Before the procedure runs, `SHOW :Fred` gives `dog` as a response. Inside the procedure, a new variable, also called Fred, is created. It's the new, local variable that gets set to `hound`, and that's what will be printed by the `SHOW` command. When the procedure finishes, though, the local variable vanishes, and you'll see `dog` again if you run `SHOW :Fred`.

MAKE

```
MAKE name object
```

MAKE sets the value of the variable name to object. If the variable already exists, MAKE simply changes the value of the existing variable. If the variable doesn't exist, MAKE will create a new, global variable.

Snippet

```
MAKE "Colors [red orange yellow green blue violet]
```

NAME

```
NAME name object
```

NAME is another version of the MAKE command. The only difference is the value comes first, followed by the name of the variable.

Snippet

```
NAME [tall blonde] "Igor
```

NAMEP

```
NAMEP name
```

NAMEP returns TRUE if name is the name of a variable, and FALSE if it isn't.

Snippet

```
IF NOT NAMEP "Spot [MAKE "Spot "dog]
```

THING

```
THING variable
```

THING returns the value of name. It's completely equivalent to :variable. In fact, :variable is really just a shorter way of saying THING "variable.

Snippet

```
PR THING "Spot
```

Words and Lists

Numbers and Words

A lot of the commands in this section are used to pull apart words or put them together. In other languages, these would be called string manipulation commands. Logo shares a rather unusual ability with other artificial intelligence languages like LISP: A number is actually just a special kind of word. All of the commands that work on words will also work on numbers.

To see how this works, let's look at a couple of examples. Let's say you print a word, like this:

```
SHOW "010.0
```

Since there is a quote mark right before the characters, Logo treats this as a word, and prints `010.0`. If you take the quote mark off, though, Logo converts the value to a more efficient form, but you don't get exactly what you put in, either. When you try

```
SHOW 010.0
```

Logo responds with `10`.

When you use a number with a command that works on words, Logo does the same sort of conversion. For example,

```
FIRST "010.0
```

returns the word `0` (not the number!), while

```
FIRST 010.0
```

returns the word `1`, since the number is converted to 10 before `FIRST` gets a chance to work on the number.

Incidentally, you can also use words as input to calls that expect numbers; just make sure the string is a legal number before you use it as a parameter.

ASCII

```
ASCII word
```

`ASCII` returns the ASCII character code for the first letter in `word`.
Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
PR ASCII "a
```

BEFOREP

```
BEFOREP word1 word2
```

`BEFOREP` returns `TRUE` if `word1` is before `word2`, and `FALSE` if it isn't.

The strings are compared character by character. One character is less than another if the ASCII character code for the character is less than the ASCII character code for the other character. For the most part, that means one character is less than the other if it comes first in alphabetical order, but all uppercase letters are less than lowercase letters.

If one word is shorter than the other, but the two words match up to the end of the shorter word, the shorter word is less than the longer one.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Tip You can use `BEFOREP` to sort words, but usually you want to sort words in dictionary order, ignoring the case. An easy way to sort the words is to use `LOWERCASE` or `UPPERCASE` to convert both words to the same case, then use `BEFOREP` to compare the words, like this:

```
   BEFOREP   LOWERCASE   :word1   LOWERCASE
:word2
```

BUTFIRST

```
BUTFIRST object
BF object
```

`BUTFIRST` returns all of the object except the first element.

In the case of a list, `BUTFIRST` returns a list that has all of the elements from the input list except the first element. It is an error to use `BUTFIRST` on an empty list.

If `object` is a word, `BUTFIRST` returns the same word with the first character removed. If there is only one character, `BUTFIRST` returns a word with no characters. It is an error to give `BUTFIRST` a word with no characters as input.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
MAKE "Rest BUTFIRST :Cities
```

BUTLAST

```
BUTLAST object
BL object
```

`BUTLAST` returns all of the object except the last element.

In the case of a list, `BUTLAST` returns a list that has all of the elements from the input list except the last element. It is an error to use `BUTLAST` on an empty list.

If `object` is a word, `BUTLAST` returns the same word with the last character removed. If there is only one character, `BUTLAST` returns a word with no characters. It is an error to give `BUTLAST` a word with no characters as input.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
MAKE "Singular BUTLAST "words
```

CHAR

```
CHAR number
```

`CHAR` converts a number into the equivalent ASCII character. Compare this with the `ASCII` function, which converts characters to numbers.

Snippet

```
TO NextCh :ch
OP CHAR 1 + ASCII :ch
END
```

COUNT

```
COUNT object
```

COUNT returns the number of elements in object. For a list, this is the number of items in the list. For a word, this is the number of characters in the word.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
PR COUNT :Members
```

EMPTYP

```
EMPTYP object
```

EMPTYP returns TRUE if object is the empty list or a word with no characters, and FALSE if one of these conditions is not met.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
WHILE NOT EMPTYP :obj [TYPE LAST :obj MAKE "obj BL :obj]
```

EQUALP

```
EQUALP object1 object2
```

EQUALP returns TRUE if object1 and object2 are equal, and FALSE if they are not.

For numbers, the two objects are equal if they are the same number. If one of the objects is an expression, the expression is evaluated first, then compared to the other object.

For words, the two objects are equal if the words are identical. Letter case does matter, so "DAVE is not equal to "Dave.

For lists, the two objects are equal if the lists are equivalent. That means that each list has to contain exactly the same number of objects, and each object in the list has to be equal to the corresponding object in the other list.

Snippet

```
IF EQUALP "black :color [SETPC 0]
```

FIRST

```
FIRST object
```

FIRST returns the first element of object.

In the case of a list, FIRST returns the first element of the list. Unlike BUTFIRST and BUTLAST, FIRST doesn't return a list, unless, of course, the first element of object happens to be a list. It is an error to use FIRST on an empty list.

If object is a word, FIRST returns a word consisting of the first character in object. It is an error to give FIRST a word with no characters as input.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
PR FIRST [John Q. Doe]
```

FLOATP

```
FLOATP object
```

FLOATP returns TRUE if object is a floating-point number, and FALSE if it is anything else.

Snippet

```
IF NOT FLOATP :number [PR [I expected a price, like 4.35]]
```

FPUT

```
FPUT object list
```

FPUT places object at the start of list, and returns the new list.

LIST and LPUT can also be used to create or add to lists.

Snippet

```
FPUT "Fred [Sam Susan Karen]
```

INTEGERP

```
INTEGERP object
```

INTEGERP returns TRUE if object is an integer number, and FALSE if it is anything else.

Snippet

```
IF NOT INTEGERP :count [PR [I need a number!]]
```

ITEM

```
ITEM integer object
```

ITEM returns an element from object. integer tells ITEM which element to return. The items are numbered from 1 to the number of items in the object. integer must be greater than or equal to 1.

In the case of a list, ITEM returns an element of the list. Unlike BUTFIRST and BUTLAST, ITEM doesn't return a list, unless, of course, the element happens to be a list. If integer is larger than the number of elements in the list, ITEM will return an empty list.

If object is a word, ITEM returns a character from object. If integer is larger than the number of characters in the word, ITEM returns an empty string.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
MAKE "word ITEM 1 + :RANDOM 100 :HangmanWords
```

LAST

```
LAST object
```

LAST returns the last element of object.

In the case of a list, LAST returns the last element of the list. Unlike BUTFIRST and BUTLAST, LAST doesn't return a list, unless, of course, the last element of object happens to be a list. It is an error to use LAST on an empty list.

If object is a word, LAST returns a word consisting of the last character in object. It is an error to give LAST a word with no characters as input.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
   TO Reverse :list
   LOCAL "new
   MAKE "new []
   WHILE NOT EMPTYP :list [MAKE "new FPUT LAST :list :new MAKE
"list BUTLAST :list]
   OUTPUT :new
   END
```

LIST

```
LIST object1 object2
(LIST object1 object2 object3 ... )
```

`LIST` creates a list made up of the input objects. When you use the parenthesized form, you can give `LIST` any number of inputs, including a single input.

Compare `LIST` with `SENTENCE`, which also builds a list, but uses the *contents* of the objects instead of the objects themselves.

Snippet

```
   MAKE "pets (LIST Fred Sam Psi)
```

LISTP

```
LISTP object
```

`LISTP` returns `TRUE` if `object` is a list, and `FALSE` if it is not a list.

Snippet

```
   IF LISTP :parameter [ProcessList :parameter]
```

LOWERCASE

```
LOWERCASE word
```

`LOWERCASE` returns the input word after converting all of the uppercase letters in the word to the equivalent lowercase letters.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
IF EQUALP LOWERCASE :animal "dog [PR "Bark!]
```

LPUT

```
LPUT object list
```

`LPUT` places `object` at the end of `list`, and returns the new list. `LIST` and `FPUT` can also be used to create or add to lists.

Snippet

```
MAKE "kids LPUT :who :kids
```

MEMBER

```
MEMBER object1 object2
```

`MEMBER` returns the part of `object2` that starts with `object1`.

If `object2` is a list, `MEMBER` checks each element of the list to see if it is equal to `object1`. If `MEMBER` finds a match, it returns a list that starts with the matching member, and contains all of the elements of `object2` from that element on. If there is no match, `MEMBER` returns the empty list.

If `object2` is a word, then `object1` must also be a word. In this case, `MEMBER` searches `object2` for a sequence of characters that matches `object1`. If it finds a match, `MEMBER` returns a word that starts with the first matching character from `object2`, and contains that character and all of the characters that follow it. If `MEMBER` does not find a match, it returns an empty word.

Snippet

```
IF NOT EMPTYP MEMBER :who :membership [PR [He is a member]]
```

MEMBERP

```
MEMBERP object1 object2
```

`MEMBERP` returns `TRUE` if `object1` is a member of `object2`, and `FALSE` if it is not a member of `object2`.

If `object2` is a list, `object1` is a member of `object2` if at least one of the elements of `object2` is equal to `object1`. The comparison is made exactly as if `EQUALP` were used to compare `object1` to each element of `object2`.

If `object2` is a word, then `object1` must also be a word. In this case, `object1` is a member of `object2` if `object1` appears somewhere in `object2`.

Snippet

```
IF NOT MEMBERP :who :membership [PR [He is a member]]
```

NUMBERP

```
NUMBERP object
```

`NUMBERP` returns `TRUE` if `object` is a number, and `FALSE` if it is not a number.

Snippet

```
IF NOT NUMBERP :value [PR [Please give me a number]]
```

PARSE

```
PARSE word
```

`PARSE` takes a word and breaks it up into a list, returning the list.

Snippet

```
MAKE "characters PARSE :who
```

SENTENCE

```
SENTENCE object1 object2
SE object1 object2
(SENTENCE object1 object2 object3 ... )
(SE object1 object2 object3 ... )
```

`SENTENCE` creates a list made up of the contents of the input objects. When you use the parenthesized form, you can give `SENTENCE` any number of inputs, including a single input.

If the objects are words or numbers, SENTENCE works just like its cousin, LIST. The difference between the two commands is how they handle objects that are lists. In this case, LIST puts the list itself into the output list, while SENTENCE puts the elements from the list into the output list. For example,

```
SHOW SENTENCE 1 [2 3 4]
```

prints `[1 2 3 4]`, but

```
SHOW LIST 1 [2 3 4]
```

prints `[1 [2 3 4]]`.

UPPERCASE

```
UPPERCASE word
```

UPPERCASE returns the input word after converting all of the lowercase letters in the word to the equivalent uppercase letters.

Numbers are words, too. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
TO Compare :word1 :word2
OP EQUALP UPPERCASE :word1 UPPERCASE :word2
END
```

WORD

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

WORD creates a word by combining all of the input words. It is an error if one of the inputs is a list.

Numbers are treated as words. See "Numbers and Words," earlier in this chapter, for details.

Snippet

```
TO Plural :word
OUTPUT WORD :word "s
END
```

WORDP

```
WORDP object
```

WORDP returns TRUE if object is a word, and FALSE if it is not a word.

Logo treats numbers as a special kind of word, so WORDP returns TRUE if object is a number, too. To see if an object is a word that is not a number, you have to make sure that WORDP returns TRUE and that NUMBERP returns FALSE.

Snippet

```
TO OnlyWordP :Object
IF NUMBERP :Object [OUTPUT "FALSE]
OUTPUT WORDP :Object
END
```

Property Lists

You can use property lists to organize information. A property list contains a series of names and properties. The names are used to find a property within the property list.

For example, you might want to keep track of the students in a class. One way to do this is with a group of property lists, one per student. Within the property list, you can keep track of various information about the students.

Here's how you can build a property list sing PPROP.

```
PPROP "Susan "who [Susan Patricia Westerfield]
PPROP "Susan "parents [Mike & Patty]
PPROP "Susan "phone [123 4567]
PPROP "Susan "transportation [bus route 1]
PPROP "Susan "likes [art science math]
PPROP "Susan "activities [swimming gymnastics]
MAKE "class LPUT :class "Susan
```

You can put all of the property list names in another list, then use the various list commands and property list commands to search and change the lists.

The obvious way to organize property lists is with the same names for each of the property lists you create, but you don't have to do it that way. In our example, you'd probably want to have a name for every student, but activities might be a special property you only create when you happen to know them.

GPROP

```
GPROP name property
```

`GPROP` gets a property from a property list. `name` is the name of the property list to check, while `property` is the property to look for. `GPROP` returns the value associated with `property`.

If there is no property list named `name`, or if the property list doesn't have a property named `property`, `GPROP` returns the empty list.

<u>Snippet</u>

```
SHOW GPROP "Susan "likes
```

PLIST

```
PLIST name
```

Returns the property list `name`. The property list is a list of property names and property values, with the names right after the value. For the example at the start of this section,

```
PLIST "Susan
```

would return a list that starts off

```
[name [Susan Patricia Westerfield] "parents [Mike & Patty] ...
```

PPROP

```
PPROP name property value
```

`PPROP` places a property and value in a property list. `name` is the property list `PPROP` will modify. `property` is the name of the property `PPROP` will set, while `value` is the value for the property.

If there isn't a property list called `name`, `PPROP` creates one.

If there is already a property in the property list with the name `property`, `PPROP` changes the value to `value`. If there isn't a property in the property list called `property`, `PPROP` adds a new property and value at the end of the property list.

<u>Snippet</u>

```
PPROP "Susan "activities [swimming gymnastics]
```

REMPROP

```
REMPROP name property
```

`REMPROP` removes `property` and it's value from the property list `name`. If `property` is the last property in the property list, the entire property list is removed from the workspace.

Snippet

```
REMPROP "Susan "likes
```

SETPLIST

```
SETPLIST name list
```

`SETPLIST` replaces all of the properties in the property list `name` with the properties in the property list `list`.

`list` is a list with an even number of elements. Counting from 1, the first element and all other odd numbered elements must be non-numeric words. These are the names for the properties in the property list. The entries in the list right after each property name is the property value for that name.

Snippet

```
SETPLIST "Frank PLIST "Defaults
```

Numbers and Arithmetic

One of the three forms for an object is a number. A Logo number is pretty much the same thing as what you think of as a number. Numbers can be whole numbers (integers in computer terms), like 1 or 47, or decimal numbers, like 4.5.

Logo uses two different ways to store numbers. Integers are whole numbers, like 4 or 1993. Integers have a definite range. Logo can handle integers from -2147483648 to 2147483647. If you type a whole number that is too big or too small to be an integer, Logo automatically converts the number to a floating-point number.

Floating-point numbers can have a fraction part, like 4.5 or 3.14159. They can also use scientific notation to represent very large or very small numbers, like 3e99. Floating-point numbers can range from 2.3e-308 to 1.7e+308.

Expressions

The obvious way to write a number object is just to type the number, but there's actually another way. Instead of writing the number, you can calculate it using Logo's math operations. Keep in mind that a number can literally be a value, like 4 or 3.14159, or it could be a number returned by some Logo function, or even a value returned by a function you write.

operator	use
-	When - appears right before a number, like -4, it changes the sign of the number.
+	The + operator is used between two numbers. It adds the two numbers together, returning the result.
-	When - appears after a number, Logo expects to find a second number right after the - operator. The right-hand number is subtracted from the left-hand number. For example, 3 - 7 returns -4.
*	The * operator is used between two numbers. It multiplies the two numbers, returning the result. For example, 4 * 5 is 20.
/	The / operator is used between two numbers. The number on the left is divided by the number on the right. For example, 5 / 4 returns 1.2, and 100 / 10 returns 10.

There are three other operations you can use in an expression that don't return a number; instead, they return either `TRUE` or `FALSE`. Each of these three comparison operators works on two numbers, comparing the number to the left of the operator to the number to the right of the operator.

operation	meaning
a < b	Returns `TRUE` of a is less than b, and `FALSE` if it is not.
a > b	Returns `TRUE` of a is greater than b, and `FALSE` if it is not.
a = b	Returns `TRUE` of a is equal to b, and `FALSE` if it is not.

Note Unlike the other math operators, = can be used on objects that are lists or words as well as on objects that are numbers. In fact,

```
object1 = object2
```

is just another way of writing

```
EQUALP object1 object2
```

The arithmetic operations use operator precedence to decide how calculations will be done. Operator precedence is just a fancy way of saying that some of the operations are done first, no matter what order they appear in. For example, when you write 1 + 2 * 3 in math class, you generally assume that the multiplication will be done first, then the addition, so the answer is 7.

Logo works the same way. Here's a table that shows the operator precedence from highest (the operations that will be done first) to lowest. In the case of a tie, the operations are done from left to right.

operation	notes
`-`	This is the unary subtraction, as in -4.
`*` `/`	
`+` `-`	This is the subtraction operator, as in 2-4.
`<` `>` `=`	

You can use parenthesis to force Logo to do calculations in a particular order. For example, 1 + 2 * 3 is 7, but (1 + 2) * 3 is 9. Anything inside of a set of parenthesis is calculated completely before any operation from outside the parenthesis is done.

Anything that isn't in the table, like the built in math function `SIN` or one of your own functions, has a lower priority. Because of this rule

```
1 + SQRT 25 + 11
```

works like

```
1 + SQRT (25 + 11)
```

returning 7, not 17.

Special Rules for /

Most of the math operators are delimiters, too. Delimiters are special characters that Logo treats as the start of a new symbol. For example, if you type

```
:value*4
```

Logo knows that you want to multiply the contents of the variable `value` by 4. The reason is that `*` is a delimiter, so Logo treats it as the start of a new symbol, and treats the character that comes after it as the start of yet another symbol. The way Logo reads the line, you might as well have typed

```
:value * 4
```

In fact, that's what Logo will print when it lists a program that contains your original line.

The division operator, `/`, isn't a delimiter. Because of this, you need to make sure you put spaces around this character when you want it to be treated like a math operation. When you type

```
:value/4
```

Logo things you want to know the contents of a variable named `value/4`. When you type

```
:value / 4
```

Logo returns the contents of the variable `value` divided by 4.

There is a simple rule of thumb that will always keep you out of trouble: Always put a space on both sides of any math operator that requires two inputs, one on either side of the operator.

Special Rules for -

The character - is used for two completely different purposes. Because Logo is a list processing language, it's easy to get the purposes messed up. For example, is [2-5] a list with two elements, 2 and -5, or an expression that gives -3? That's very important when you're using a command like

```
SETPOS [20 -50]
```

Here are the rules Logo uses to decide what a - character really means:

- If a - character comes right before a number, and has any delimiter except a) character right before the - character (a space counts), the - character is treated like a negative sign for the number. For example,

  ```
  LAST [20 -50]
  ```

  is -50.

- If a - character comes right after a numeric expression, and the first rule doesn't apply, it's treated as a subtraction operator. For example, both of the - characters in

  ```
  SETPOS [XCOR - 10 YCOR-10]
  ```

  are subtractions.

- If the first rule doesn't apply, and the - character does not come after a numeric expression, it's a unary subtraction. For example,

  ```
  SETPOS [- XCOR YCOR]
  ```

  has a unary subtraction.

Trying to follow these rules literally isn't very easy. The easy way to stay out of trouble is to remember to always put spaces on both sides of a subtraction operation, and to always put a space before, but not after, a minus sign. In addition, don't use minus signs in front of functions – use an expression, as in

```
SETPOS [0 - XCOR 0 - YCOR]
```

ABS

`ABS value`

`ABS` returns the absolute value of the argument. For example, `ABS -3` returns 3, while `ABS 10` returns 10.

ARCCOS

`ARCCOS value`

`ARCCOS` returns the angle whose cosine is `value`. `value` must be a number, or some combination of Logo statements that return a number. The angle is returned in degrees.

Snippet

```
MAKE "angle ARCCOS :adjacent / :hypotenuse
```

ARCSIN

`ARCSIN value`

`ARCSIN` returns the angle whose sine is `value`. `value` must be a number, or some combination of Logo statements that return a number. The angle is returned in degrees.

Snippet

```
MAKE "angle ARCSIN :opposite / :hypotenuse
```

ARCTAN

```
ARCTAN value
```

ARCTAN returns the angle whose tangent is value. value must be a number, or some combination of Logo statements that return a number. The angle that is returned is in degrees, and will be in the range 270 to 359 or 0 to 90.

Snippet

```
MAKE "angle ARCTAN :opposite / :adjacent
```

ARCTAN2

```
ARCTAN2 x y
```

ARCTAN2 returns the angle from a vertical line through the point [0 0] to [x y]. The angle that is returned will be greater than or equal to 0, and less than 360. The angle works just like turtle graphics, with 0 degrees being up, 90 degrees to the right, and so forth.

Snippet

```
MAKE "angle ARCTAN2 :adjacent :opposite
```

COS

```
COS angle
```

COS returns the cosine of the angle. angle must be a number, or some combination of Logo statements that return a number. Use degrees for the angle, just as you do with the Turtle Graphics commands.

▲ **Warning** If you use extremely large angles, the number you get back isn't very accurate. If you're doing a lot of math to calculate the angle, try and keep the result as close to the range of 0 to 359 degrees as possible. ▲

Snippet

```
MAKE "adjacent :hypotenuse * COS :angle
```

DIFFERENCE

```
DIFFERENCE value1 value2
```

`DIFFERENCE` returns `value1 - value2`. The inputs must both be numbers or statements that return numbers.

Snippet

```
PR DIFFERENCE :money :price
```

EXP

```
EXP number
```

`EXP` returns the exponent of `number`.

Snippet

```
TO PWR10 :value
OP EXP :value * LN 10.0
END
```

FLOAT

```
FLOAT number
```

`FLOAT` converts a numeric value to a floating-point value.

Tip The main reason for using `FLOAT` is to convert an integer into a floating-point number, which changes the way Logo does some math operations. For example, let's say you want to multiply two large numbers, like

```
SHOW 1000000 * 1000000
```

The answer is 1000000000000, but the largest integer Logo can handle is 2147483647. The largest floating-point number Logo can handle is a log bigger, though; it's about 1.7e308. By converting one of the integer inputs to a floating-point number, you tell Logo to do the multiplication using floating-point math, and return the result as a floating-point number. When you type

```
SHOW 1000000 * FLOAT 1000000
```

Logo gives you 1.000000000000000e+12. The number is in scientific notation, but it is the right number.

FORM

```
FORM number width digits
```

FORM formats a number, returning a word.

number is the number to format.

field is the field width. The word that FORM returns will have this many characters. If the number is so small that it doesn't need this many characters, blanks will be placed at the start of the word to force it to be width characters long. If width is too small, the number will be returned using the smallest number of characters possible. width must be a number from 1 to 128.

precision is the number of digits after the decimal point. It must be a number from 0 to 6. If you use 0, the number will be printed as an integer.

Some numbers are too large or too small to represent as integers or as numbers with a decimal point. If the absolute value of the number is smaller than 0.000001 or larger than 999999.0, the number will be printed using scientific notation.

Snippet

```
TO TYPE$ :amount
TYPE FORM :amount 1 2
END
```

INT

```
INT number
```

INT returns the integer part of a number. To get the integer part of a number, all of the digits to the right of the decimal place are removed.

Snippet

```
RIGHT INT :angle
```

INTQUOTIENT

```
INTQUOTIENT numerator denominator
```

`INTQUOTIENT` divides two integers, returning an integer result.
If `denominator` is zero, you will get a divide by zero error.
You can use real numbers for the inputs, but if you do, the inputs are converted to integers as if the `INT` function were used before the numbers are divided.

Snippet

```
(TYPE "You\ each\ get\  INTQUOTIENT :candy :kids "pieces.)
```

LN

```
LN number
```

`LN` returns the natural logarithm of `number`.

Snippet

```
PR EXP LN 3 + LN 4
```

POWER

```
POWER x y
```

`POWER` returns `x` raised to the power `y`. `y` must be greater than or equal to 0.

Snippet

```
PR :price * POWER 1.0 + :interest :months
```

PRODUCT

```
PRODUCT number1 number2
(PRODUCT number1 number2 number3 ... )
```

`PRODUCT` multiplies all of the input numbers, returning the result. If you use the second form, with parenthesis, it's possible to give `PRODUCT` a single number. In that case, `PRODUCT` returns the input number.

Snippet

```
(TYPE "The\ volume\ is\  (PRODUCT :length :width :height))
```

QUOTIENT

```
QUOTIENT numerator denominator
```

`QUOTIENT` divides two real numbers, returning a real number as the result. The inputs can be integers, but they will be converted to real numbers before the numbers are divided.

If `denominator` is zero, you will get a divide by zero error.

Snippet

```
MAKE "length QUOTIENT :area :height
```

RANDOM

```
RANDOM number
```

`RANDOM` returns a random integer that is greater than or equal to zero and less than `number`. See `RERANDOM` for a way to generate the same sequence of random numbers a second time.

Snippet

```
TO PICK :object
OP ITEM 1 + COUNT :object :object
END
```

REMAINDER

`REMAINDER numerator denominator`

`REMAINDER` returns the remainder from an integer division. The rules used to divide the numbers are the same as for `INTQUOTIENT`, but `REMAINDER` returns the remainder instead of the result of the division.

Snippet

```
MAKE "remaining REMAINDER :things :people
```

RERANDOM

`RERANDOM`

Computer generated random numbers aren't really random, of course. They are actually very specific numbers that are generated so they seem to be random. Random number generators start with a number called the seed. Logo uses the computer's clock to form a seed for its random number generator. The `RERANDOM` function will restart the random number generator, using the same value that was used for the very first call to `RANDOM`. When you do this, you will get the same sequence of random numbers again. Since Logo uses the clock to start the random number generator in the first place, though, the sequence will be different if you quit Logo and start again.

Snippet

```
PlayGame
RERANDOM
PlayGame
```

ROUND

`ROUND number`

`ROUND` converts real numbers to integers, returning the integer that is closest to the real number. Compare this with the `INTEGER` function, which returns the real number after removing the fraction part.

Snippet

```
TO ROUND$ :amount
OP QUOTIENT ROUND :amount * 100 100
END
```

SIN

```
SIN angle
```

`SIN` returns the sine of the angle. `angle` must be a number, or some combination of Logo statements that return a number. Use degrees for the angle, just as you do with the Turtle Graphics commands.

▲ **Warning** If you use extremely large angles, the number you get back isn't very accurate. If you're doing a lot of math to calculate the angle, try and keep the result as close to the range of 0 to 359 degrees as possible. ▲

Snippet

```
MAKE "opposite :hypotenuse * COS :angle
```

SQRT

```
SQRT value
```

`SQRT` returns the square root of `value`. `value` must be a number or a statement that returns a number.

The square root is the number that, when multiplied by itself, gives `value`. You can't take the square root of a negative number; if you try, you'll get an error.

Snippet

```
MAKE "length SQRT X * X + Y * Y
```

SUM

```
SUM number1 number2
(SUM number1 number2 number3 ... )
```

`SUM` adds all of the input numbers, returning the result. If you use the second form, with parenthesis, it's possible to give `SUM` a single number. In that case, `SUM` returns the input number.

Snippet

```
PR (SUM 1 2 3 4 5)
```

TAN

```
TAN angle
```

TAN returns the tangent of the angle. angle must be a number, or some combination of Logo statements that return a number. Use degrees for the angle, just as you do with the Turtle Graphics commands.

▲ **Warning** If you use extremely large angles, the number you get back isn't very accurate. If you're doing a lot of math to calculate the angle, try and keep the result as close to the range of 0 to 359 degrees as possible. ▲

Snippet

```
MAKE "opposite :adjacent * TAN :angle
```

Flow of Control

CATCH

```
CATCH word list
```

CATCH runs list, just as if you used the RUN command. While the commands are running, though, you can use the THROW command. The THROW command jumps back to the CATCH command, stopping the execution of list. If CATCH is used in a procedure, the program will continue with the first statement after CATCH.

You can have more than one CATCH active at a time, so you need some way to tell the THROW command which CATCH command you want to go to. word is a label. The THROW command uses a label, too. Logo matches the THROW to the CATCH with the same label.

It's perfectly legal to THROW from some other procedure, which makes CATCH a perfect way to handle errors. For example, you can use a command like

```
CATCH "oops [DoButtonAction1]
```

to run a procedure called DoButtonAction1. From inside DoButtonAction1, or any procedure DoButtonAction1 calls, you can use a THROW command to gracefully stop the script if you find an error, like this:

```
IF errorDetected [THROW "oops]
```

There is one special `CATCH` label. If you use the label `"ERROR`, your `CATCH` statement will catch Logo errors, like divide by zero. If you catch the error, the script won't stop; it just jumps back to your `CATCH` statement and lets you handle the problem. You can use the `ERROR` statement to find out what error was flagged.

DOUNTIL

```
DOUNTIL list condition
```

`DOUNTIL` is a loop statement, sort of like `REPEAT`. `DOUNTIL` executes the statements in `list` as long as `condition` evaluates to `FALSE`. The instructions in `list` will always be evaluated at least once, since `condition` isn't checked until after `list` is executed.

Snippet

```
DOUNTIL [SETPOS MOUSE] NOT BUTTONP
```

ERROR

```
ERROR
```

`ERROR` is used to find out what error has occurred. When you use `CATCH` to catch errors, and Logo finds an error, it records the error number. `ERROR` returns a list; the first element of that list is the number of the last error caught by a `CATCH` statement. If there haven't been any errors caught, the error number will be zero.

Here's a list of the errors Logo can return. There are some gaps in the error numbers; that's to try and keep the error numbers in sync with other popular Logo implementations.

error	message
6	`obj` is a primitive
7	Can't find the label `obj`
9	`obj` is undefined
13	Can't divide by 0"
19	Too few items in `name`
20	Too many files are open
21	Can't find catch for `obj`
23	Out of space
25	`obj` is not TRUE or FALSE
29	Not enough inputs to `name`
30	Too many inputs to `name`
33	Can only do that in a procedure
34	Turtle out of bounds
35	I don't know how to `name`
38	You didn't say what to do with `obj`
41	`name` doesn't like `obj` as input

45	`name` is not open
57	Can't write to `obj`
58	Can't read from `obj`
59	A sound channel could not be started
60	Not enough memory to run Logo
61	Property lists must have an even number of elements
62	The directory `name` is not valid, or is not available
63	Disk I/O error
64	POFILE can only print text files
65	`name` wants a reader file
66	`name` needs a text file
67	`name` could not find a window
68	`name` could not find the window `obj`
69	Out of stack space
71	You used a TO without an END
72	Can't use `name` in a script

Snippet

```
IF NOT ERROR = 0 [PR [Abnormal termination]]
```

GO

```
GO label
```

`GO` jumps to the `LABEL` statement with the same label. `label` is a word. There must be a `LABEL` statement with a matching label in the same procedure as the `GO` statement.

△ **Important** You can only use `GO` from inside of a procedure. It can't be used from inside a button script. △

Snippet

```
GO "Fish
```

IF

```
IF condition trueList
IF condition trueList falseList
```

`IF` is used to pick between alternatives. `condition` is an expression that returns either `TRUE` or `FALSE`.

If `condition` is `TRUE`, `trueList` is executed, just as if you used the `RUN` command. `trueList` must be a list.

If `condition` is `FALSE`, and the second form of the `IF` statement is used, `falseList` is executed. Just as with `trueList`, `falseList` must be a list, and will be executed using the `RUN` command.

If `condition` is `FALSE` and the first form of the `IF` statement is used, the command does nothing.

The `IF` statement is a little unusual, since it can be either a command or a function. The `IF` statement returns whatever the list returns when it is executed. If the list doesn't return anything, or if you use the first form of the `IF` statement and condition is `FALSE`, if is a command. If one of the lists is executed and returns a value, `IF` returns the same value.

<u>Snippet</u>

```
PRINT IF :x < 0 [SQRT - :x] [SQRT :x]
```

IFFALSE

```
IFFALSE list
IFF list
```

If the last `TEST` statement was `FALSE`, `IFFALSE` runs `list`. If no `TEST` statement has been used in the current procedure, or if the last one returned `TRUE`, `IFFALSE` does nothing.

`IFFALSE` can only be used inside of a procedure.

`list` must not return a result. `IFFALSE` is a command, and does not return a result.

<u>Snippet</u>

```
IFFALSE [PR [Sorry\, that's not right.]
```

IFTRUE

```
IFTRUE list
IFT list
```

If the last `TEST` statement was `TRUE`, `IFTRUE` runs `list`. If no `TEST` statement has been used in the current procedure, or if the last one returned `FALSE`, `IFTRUE` does nothing.

`IFTRUE` can only be used inside of a procedure.

`list` must not return a result. `IFTRUE` is a command, and does not return a result.

Snippet

```
IFTRUE [PR [That's right!]]
```

LABEL

```
LABEL label
```

`LABEL` is used to create destinations for `GO` statements. If your program happens to execute a LABEL statement, it just gets skipped.

`label` must be a word, and it has to be a word constant. For example, you cannot call a function that returns the word.

Snippet

```
LABEL "Fish
```

OUTPUT

```
OUTPUT object
OP object
```

`OUTPUT` returns from the current procedure, returning a value in the process. When you leave a procedure with `OUTPUT`, your procedure becomes a function; the result is `object`.

`OUTPUT` can only be used inside of a procedure.

See `STOP` for a way to return from a procedure without returning a value.

Snippet

```
TO ! :value
LOCAL "x
MAKE "x 1
WHILE :value > 1 [MAKE "x :x * :value MAKE "value :value - 1]
OP :x
END
```

REPEAT

```
REPEAT count list
```

`REPEAT` runs `list` `count` times.

Snippet

```
REPEAT 4 [FD 20 RT 90]
```

RUN

```
RUN list
```

`RUN` runs `list` just as if it were typed from the input line. If `list` is an operation, `RUN` writes the result to the screen.

The `RUN` command gives Logo one of it's most powerful features: with some care, you can create a Logo program from inside your own Logo program by forming a list. Once the list has been created, you can run the list from inside your program.

Snippet

```
TO Apply :command :list
IF EMPTYP :list [STOP]
RUN LIST :command FIRST :list
Apply :command BUTFIRST :list
END
```

STOP

```
STOP
```

`STOP` returns from the current procedure. When you leave a procedure with `STOP`, your procedure acts like a command.

You don't need to use `STOP` to return from a procedure if you are already an the end of the procedure. The procedure stops automatically when it runs out of statements.

`STOP` can only be used inside of a procedure.

See `OUTPUT` for a way to return a value when you return from a procedure.

Snippet

```
IF :done [STOP]
```

TEST

```
TEST condition
```

`TEST` evaluates `condition`, setting a flag that is used later by `IFTRUE` and `IFFALSE` statements. Each procedure has its own test flag, so using `TEST` in one procedure doesn't effect the way other procedures work.

`condition` must return either `TRUE` or `FALSE`.

Snippet

```
TEST EQUALP :reply :answer
```

THROW

```
THROW word
```

`THROW` jumps back to the `CATCH` statement whose label is word. It is an error if there is no `CATCH` statement to catch this throw.

See the description of the `CATCH` command for details on how `THROW` is used.

Snippet

```
THROW "Exit
```

TOPLEVEL

```
TOPLEVEL
```

`TOPLEVEL` stops a program. If it is executed while a button script is running, the script stops. If it is executed while you are using the console, whatever is running is stopped and you can type a new command.

Snippet

```
IF ErrorFound [TOPLEVEL]
```

WAIT

```
WAIT value
```

WAIT waits for value 1/60ths of a second. You can use WAIT to pause in a program. While WAIT isn't accurate enough to use as a clock, it isn't effected by the machine you are using – it will wait just as long on an accelerated Apple IIGS as on a standard model.

Snippet

```
WAIT 60
```

WHILE

```
WHILE condition list
```

WHILE is a loop statement, sort of like REPEAT. The difference is that REPEAT executes a list for a specific number of times, but WHILE executes a list as long as condition is TRUE. As soon as condition is not TRUE, WHILE stops.

What actually happens is this: WHILE starts by evaluating condition. If the result is TRUE, the statements in list are executed, and the WHILE statement starts over. If condition is not TRUE, WHILE stops immediately and moves on to the next statement.

Since the condition is checked right away, a while loop doesn't necessarily execute the statements in list at all. If condition is not TRUE the first time it is evaluated, all WHILE does is make sure list is, in fact, a list.

Snippet

```
WHILE BUTTONP [SETPOS MOUSE]
```

Logical Operators

AND

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

AND returns TRUE if all of the input objects are TRUE, and FALSE in any one of the input objects is FALSE. All of the inputs must be TRUE or FALSE.

Snippet

```
TO Range :start :value :end
OP :start < :value AND :value < :start
END
```

NOT

```
NOT object
```

`NOT` returns `TRUE` if `object` is `FALSE`, and `FALSE` if `object` is `TRUE`. `object` must be `TRUE` or `FALSE`.

Snippet

```
PR NOT :a < :b
```

OR

```
OR object1 object2
(OR object1 object2 object3 ...)
```

`OR` returns `TRUE` if at least one of the input objects are `TRUE`, and `FALSE` if all of the input objects are `FALSE`. All of the inputs must be `TRUE` or `FALSE`.

Snippet

```
PR :a < :b OR :a = :b
```

Input and Output

The input and output commands are used to communicate with the outside world.

BUTTONP

```
BUTTONP
```

BUTTONP returns TRUE if the mouse button is down, and FALSE if it is not being pressed.

Snippet

```
WHILE BUTTONP [SETPOS MOUSE]
```

KEYP

```
KEYP
```

KEYP returns TRUE if a key is available from the current input device (usually the keyboard), and FALSE if not.

Snippet

```
IF KEYP [DoCommand]
```

MOUSE

```
MOUSE
```

MOUSE returns a two-element list. The first element is the horizontal mouse position, using Turtle Graphics coordinates. The second element of the list is the vertical mouse position.

Snippet

```
WHILE BUTTONP [SETPOS MOUSE]
```

PRINT

```
PRINT object
(PRINT object1 object2 object3 ...)
PR object
(PR object1 object2 object3 ...)
```

PRINT prints the input objects, printing each one on a separate line. It follows the last object with a carriage return, too, so anything you print with another command will start on a fresh line.

PRINT does not print brackets if the object is a list.

Note SHOW and TYPE are similar to SHOW, but work in slightly different ways.

Snippet

```
PRINT [Hello\, world.]
```

READCHAR

```
READCHAR
RC
```

READCHAR returns a key from the current input device, usually the keyboard. If a key has not been pressed, READCHAR waits for a keypress before returning. The key that is pressed is not printed.

Tip You can use KEYP to see if a key is available.

Snippet

```
MAKE "ch READCHAR
```

READCHARS

```
READCHARS number
RCS number
```

READCHARS reads number characters from the keyboard. It returns all of the characters in a word.

Snippet

```
MAKE "zipcode READCHARS 5
```

READLIST

```
READLIST
RL
```

READLIST reads a line from the input device (usually the keyboard) and converts the input into a list. READLIST returns the list. It's equivalent to PARSE READWORD.

Snippet

```
MAKE "address READLIST
```

READWORD

```
READWORD
```

READWORD reads a line from the current input file, returning the line it reads as a word. Normally that's the keyboard, but you can change where the lines are read from using SETREAD.

From a script, READWORD will being up a dialog, returning whatever the user types as a Logo word. You can control the size and position of the dialog with SETRWRECT. The dialog will default to a ? for a prompt. You can change this default prompt using SETRWPROMPT.

From a text window, READWORD normally reads text from the window itself. You can use TURTLEIO to tell READWORD to use a dialog from the text window, too.

Snippet

```
MAKE "answer READWORD
```

SETRWPROMPT

```
SETRWPROMPT word
```

SETRWPROMPT sets the prompt string for a READWORD dialog. The prompt will be used by the next call to READWORD, but will revert to the default prompt of a single ? character for subsequent calls.

Snippet

```
SetRWPrompt "Hi!\ \ What's\ your\ name?
Make "theName ReadWord
```

SETRWRECT

```
SETRWRECT rect
```

SETRWRECT sets the size and position for a READWORD dialog. The size and position will be used by the next call to READWORD, but will revert to the default size and position for subsequent calls.

See "Rectangles," later in this chapter, for a complete description of Logo rectangles.

Snippet

```
SetRWRect [-100 32 100 -32]
Make "theName ReadWord
```

SHOW

```
SHOW object
```

`SHOW` prints `object`, followed by a carriage return. If `object` is a list, `SHOW` will print brackets around the list.

You can use this command to write to a file; see `SETWRITE`.

Note `PRINT` and `TYPE` are similar to `SHOW`, but work in slightly different ways.

Snippet

```
SHOW :value
```

TEXTIO

```
TEXTIO
```

`TEXTIO` tells Logo to write to the current text window. That's where it normally writes anyway, so you won't need this command unless you've already used `TURTLEIO` to tell Logo to write to the current turtle port.

Snippet

```
TEXTIO
```

TOOT

```
TOOT frequency duration
```

`TOOT` plays a note.

The note will last for `duration` 1/60ths of a second.

`frequency` is the MIDI frequency for the note. It must be a number from 1 to 127.

	A	A#	B	C	C#	D	D#	E	F	F#	G	G#
Octave 1					1	2	3	4	5	6	7	8
Octave 2	9	10	11	12	13	14	15	16	17	18	19	20
Octave 3	21	22	23	24	25	26	27	28	29	30	31	32
Octave 4	33	34	35	36	37	38	39	40	41	42	43	44
Octave 5	45	46	47	48	49	50	51	52	53	54	55	56
Octave 6	57	58	59	60	61	62	63	64	65	66	67	68
Octave 7	69	70	71	72	73	74	75	76	77	78	79	80
Octave 8	81	82	83	84	85	86	87	88	89	90	91	92
Octave 9	93	94	95	96	97	98	99	100	101	102	103	104
Octave 10	105	106	107	108	109	110	111	112	113	114	115	116
Octave 11	117	118	119	120	121	122	123	124	125	126	127	

MIDI `frequency` Values and the Musical Scale

Snippet

```
TOOT 60 60
```

TURTLEIO

```
TURTLEIO
```

`TURTLEIO` tells Logo to write to the current turtle window. Once you use this command, Logo draws text in the turtle window instead of writing it in the text window.

Note Anything in a script, or in a procedure called by a script, is always written to the card. In that case, you don't need `TURTLEIO`. The only time you need to use `TURTLEIO` is when you are trying things from a text window, and want the text to be written to the card or to a turtle window.

Logo switches back to the text window as soon as it finishes running the command you type.

The obvious use of `TURTLEIO` is to write text onto a card, but there are other benefits, too. When you draw text in the turtle window or to a card, you get a lot more control over the way the text is drawn. You can change the font, the size of the letters, the style used to write the letters, the color, and even draw letters in 3D. See "Fonts" for more information about drawing text and the commands you use to control it.

△ **Important** The fact that Logo switches back to the text screen when it is waiting for you to type a command can cause some surprises. The biggest is that typing

```
TURTLEIO
PR "Hello
```

from a text window doesn't write "Hello" to the card or turtle window; it is written in the text window. The reason, of course, is that Logo switched back to the text screen after you typed TURTLEIO, while it was waiting for you to type the next command. To write to the turtle window, either put both commands in a procedure, so they will be executed before Logo needs more input, or type them on one line, like this:

```
turtleio  pr "Hello △
```

TURTLEIO changes the way Logo reads text, too. Once you switch to graphics input and output, reading a line with READLIST or READWORD doesn't wait for you to type something in a text window. Instead, Logo brings up a dialog box with a line edit box where you can type text and an OK button. You type whatever you want in the dialog, then press OK – the input command gets the characters, just as if you had typed them from a text window.

The keyboard input commands (READCHAR and READCHARS) are not effected by this command. They still read characters directly from the keyboard, never displaying the character they read.

TYPE

```
TYPE object
(TYPE object1 object2 object3)
```

TYPE prints the objects. It does not print brackets if the object is a list, and it does not print a carriage return after printing the object. If you use TYPE to print more than one object, they will be jammed together on a single line.

Note SHOW and TYPE are similar to SHOW, but work in slightly different ways.

<u>Snippet</u>

```
TYPE 1 + 3
```

Disk Commands

File Names

Several of the commands in this section use file names. File names are words that describe the name of a file and where it is located on a disk. There are a lot of different ways to give a file name, and some seem a little complicated the first time you see them. We'll cover two of the most common, and easiest to understand, kinds of file names here. You can find more detailed information about file names from a variety of sources, including the technical reference material Apple publishes for programmers.

GS/OS is a very flexible operating system that supports more than one disk format. Each of the different disk formats have slightly different rules for what is allowed and what is not allowed in a file name. We'll describe the most common disk format, ProDOS, here. The rules for ProDOS names will work on all of the other commonly used disk formats, too.

Every file on your disk has a name. The file name is what you see under the icon for a file when you look at the file from the Finder. The file name is used to identify a file uniquely, just like your name identifies you. You can use up to 15 characters in a file name. File names must start with a letter. After this starting letter, you can use letters, numbers, or the period.

As you know, you can use either uppercase or lowercase letters in a file name, but what you might not know is that, while your computer remembers the case of the letters you use, it doesn't treat them as different in any way. In programming terms, file names are case insensitive.

▲ **Warning** The fact that file names are case insensitive is very important. For example, let's say you save your workspace with this command:

```
SAVE "mystuff
```

If you go to another stack, and save its workspace with this command:

```
SAVE "MYSTUFF
```

you might think that you were saving the information to a new file, but that isn't the case. The names may look different on the disk, but your computer treats them as the same name, wiping out your original file. ▲

If there are two people named Susan in a classroom, you can usually tell them apart by using their last name. You can have two files on your computer with the same name, too, as long as they are in different folders. To tell the computer which file you mean, you can use a full path name, which works sort of like a person's last name.

The full path name starts with the name of the disk the file is on. Next comes the folder the file is in (if it's in one). The disk name, folder names, and the name of the file are all separated with colon (':') characters. You also use a colon before the name of the disk itself.

For example, to save a file called `mystuff` to a disk named `mydisk`, you use the command

```
SAVE ":mydisk:mystuff
```

To put a different file called `mystuff` on the same disk, but this time inside of a folder named `myfolder`, use the command

```
SAVE ":mydisk:myfolder:mystuff
```

If the file goes in a folder that is inside of another folder, just tack all of the folder names together, separated by colons, in the same order you would open them from the Finder.

ALLOPEN

```
ALLOPEN
```

`ALLOPEN` returns a list of all of the open files. If there are no open files, `ALLOPEN` returns the empty list.

Snippet

```
ALLOPEN
```

CATALOG

```
CATALOG
```

`CATALOG` lists all of the files in the current folder. To the left of each file name are two fields; the number of blocks the file use on the disk, and the type of the file.

```
407     ---         Logo3D.Sys16
75      ---         Installer
1       DIR         Scripts
1       DIR         Samples
1       ---         Finder.Root
1       ---         Finder.Data

Blocks Used: 553   Blocks Free: 1047
```

The file type is one of six things:

file type	meaning
DIR	A directory (folder).
TXT	A text file you can open and read with the Open command from the File menu.
ANI	An animation file, or movie. You can open and play these movies from Logo.
PNT	A picture file that you can open; it will become a turtle window.
LOG	A Logo program. You can open this file with the Open command to edit it, or execute the file as a program.
---	Any file Logo can't use is shown with dashes for the file type.

Catalog also shows the total number of blocks on the disk and the number of free blocks left on the disk. On most disks, a block is 512 bytes, but due to the way files are stored on disk, the number of blocks the file uses doesn't always have an exact relationship to the number of bytes in the file.

Snippet

```
CATALOG
```

CLOSE

```
CLOSE name
```

`CLOSE` closes a file that was opened with `OPEN`. It is an error to close a file that is not open.

Snippet

```
CLOSE "myfile
```

CLOSEALL

```
CLOSEALL
```

`CLOSEALL` closes all files that are open. You can use `CLOSEALL` even if no files are open.

Snippet

```
CLOSEALL
```

CREATEFOLDER

`CREATEFOLDER name`

`CREATEFOLDER` makes a new directory (folder).

Snippet

```
CREATEFOLDER "myfolder
```

DIR

`DIR`

`DIR` returns a list of the names of all of the files in the current folder.
See `SETPREFIX` for a way to change the current folder.

Snippet

```
PR DIR
```

ERASEFILE

`ERASEFILE file`

`ERASEFILE` removes a file from a disk. Once a file is erased, you can't recover it. The space used by the file is available for other files.

Some programs can lock files, protecting them from `ERASEFILE`.

You can erase a directory (folder) with `ERASEFILE`, but only if all of the files inside of the directory have already been erased.

Snippet

```
ERASEFILE "myfile
```

FILELEN

```
FILELEN name
```

`FILELEN` returns the length of a file in bytes. (Loosely speaking, one byte is one character.) The file must be open before you can use `FILELEN` to get the length.

Snippet

```
FILELEN "myfile
```

FILEP

```
FILEP file
```

`FILEP` checks to see if a file exists. If the file exists, `FILEP` returns TRUE; if the file does not exist, `FILEP` returns FALSE.

Snippet

```
IF FILEP "temp [ERASEFILE "temp]
```

LOAD

```
LOAD file
```

`LOAD` loads a file from your disk. The file itself must be a text file. Logo reads and executes the contents of the file just as if you had typed the file from a text window.

The most common way to use `LOAD` is to save and load files and variables from your workspace. For example, you could save your workspace, then load the procedures again the next time you use Logo. Once you have saved a file, you can even edit it with a text editor and load the results – or create a file of procedures and variable from scratch.

See "File Names," earlier in this chapter, for information about legal file names.

Snippet

```
LOAD "Cubes
```

ONLINE

```
ONLINE
```

`ONLINE` shows the name of all disks that Logo can use.

You can see what files are on the disk by using `SETPREFIX` to set the current folder to the disk, then using either `DIR` or `CATALOG`.

Snippet

```
ONLINE
```

OPEN

```
OPEN name
```

`OPEN` opens a file for reading and writing. If the file doesn't already exist, `OPEN` creates a new, empty file.

You must open a file before using the file input and file output procedures described in this section. In general, you will follow `OPEN` with a call to `SETREAD` or `SETWRITE`.

Snippet

```
OPEN "myfile
```

POFILE

```
POFILE name
```

`POFILE` prints a file. The file must be a text file. (Text files have a GS/OS file type of TXT or SRC. Logo saves programs using SRC files.)

Snippet

```
POFILE "myfile
```

PREFIX

PREFIX

PREFIX returns the default prefix.

When you use a partial path name of file name, Logo forms a full path name by attaching your partial file name to the default prefix. For example, if the prefix is :MYDISK:MYFOLDER:, and you erase the file MYFILE with the command

```
ERASEFILE myfile
```

Logo forms the full path name :MYDISK:MYFOLDER:MYFILE, and deletes this file.

You can set the default prefix with the SETPREFIX command.

Snippet

```
MAKE "old PREFIX
```

READER

READER

READER returns the name of the file that is open for reading. If there isn't an open reader file, READER returns the empty list.

You can open a file for reading with SETREAD.

Snippet

```
PR READER
```

READPOS

READPOS

READPOS returns the number of bytes that have already been read from a file.

Snippet

```
SETREADPOS READPOS + 10
```

RENAME

```
RENAME oldname newname
```

`RENAME` changes the name of a file, folder or disk. The name is changed from `oldname` to `newname`.

Snippet

```
RENAME "myfile "yourfile
```

SAVE

```
SAVE file
```

`SAVE` saves a file to your disk. It writes all of the variables, property lists and procedures to a disk file just as if you had used the `POALL` command and captured the results in the file – which, as a matter of fact, is actually what happens. The file is a text file with no owner, which means you can load it with pretty much any program that can read a text file.

See "file Names," earlier in this chapter, for information about legal file names.

Snippet

```
SAVE "Cubes
```

SETPREFIX

```
SETPREFIX prefix
```

`SETPREFIX` changes the default prefix to `prefix`.

When you use a partial path name of file name, Logo forms a full path name by attaching your partial file name to the default prefix. For example, if the prefix is :MYDISK:MYFOLDER:, and you erase the file MYFILE with the command

```
ERASEFILE myfile
```

Logo forms the full path name :MYDISK:MYFOLDER:MYFILE, and deletes this file.

When you use `SETPREFIX` to change the prefix, you can use either a full path name (one that starts with a colon and the name of a disk) or a partial path name. If you use a partial path name, it is added to the current prefix to form the new default prefix. This is a good way to burrow deeper into a directory structure. For example, let's say you put in the disk :MYDISK:, and set the default prefix to the new disk and catalog the disk with the commands

```
SETPREFIX ":mydisk:
   CATALOG
```

If you see a folder called MYFOLDER, you could use a full path name to set the default prefix to MYFOLDER with the command

```
   SETPREFIX ":mydisk:myfolder:
```

but you could save some typing by using a partial path name:

```
   SETPREFIX "myfolder
```

SETREAD

```
SETREAD name
```

SETREAD makes a file the reader. You must open a file before you can use SETREAD to make the file the reader.

Once the file is the reader, READWORD, READCHAR, READCHARS and READLIST will all read from the file.

To stop reading from the reader file and switch back to reading the keyboard, either close the file or set the reader to the empty list, like this:

```
   SETREAD []
```

You can also set the reader to a different open file. When you switch back, input will pick up from the spot it was at when you switched the reader.

SETREADPOS

```
SETREADPOS position
```

SETREADPOS sets the position in the current reader file. You can use SETREADPOS to jump around in an open file, reading first one part, then another.

It is an error if the position is past the end of the file, or if there is no open reader.

Snippet

```
   SETREADPOS :record * :length
```

SETWRITE

```
SETWRITE name
```

`SETWRITE` makes a file the writer. You must open a file before you can use `SETWRITE` to make the file the writer.

Once the file is the writer, `SHOW`, `PRINT` and `TYPE` will all write to the file.

To stop writing to the writer file and switch back to writing to the text window, either close the file or set the writer to the empty list, like this:

```
SETWRITE []
```

You can also set the writer to a different open file. When you switch back, you'll start writing at the spot it you were writing to before you switched the writer.

SETWRITEPOS

```
SETWRITEPOS position
```

`SETWRITEPOS` sets the position in the current writer file. You can use `SETWRITEPOS` to jump around in an open file, writing to first one part, then another. This is one way to create a random access database, which lets you write a record that has been changed, even if it is in the middle of a file.

It is an error if the position is past the end of the file, or if there is no open writer.

<u>Snippet</u>

```
SETWRITEPOS :record * :length
```

WRITEPOS

```
WRITEPOS
```

`WRITEPOS` returns the number of bytes that have already been written to a file. This isn't necessarily the total number of bytes in the file, since `SETWRITEPOS` can be used to jump around in the file. To find the total number of bytes in the file, use `FILELEN`.

<u>Snippet</u>

```
MAKE "record WRITEPOS / :length
```

WRITER

```
WRITER
```

`WRITER` returns the name of the file that is open for writing. If there isn't an open writer file, `WRITER` returns the empty list.

You can open a file for reading with `SETWRITE`.

<u>Snippet</u>

```
PR WRITER
```

Turtle Graphics

Turtle Positions

Turtle graphics uses a standard Cartesian coordinate system. `[0 0]` is at the center of a card, movie window, or turtle window. The first coordinate, X, increases as you move to the right, and decreases as you move to the left. The second coordinate, Y, increases as you move up, and decreases as you move down. Unless you use a scrunch factor, one unit is the same as one screen pixel when you are moving vertically, and two units when you move horizontally.

You can also use a third dimension, Z. The Z coordinate is 0 in the plane of the display screen, and increases as you move from the screen towards your eye. Negative Z coordinates move into the computer display. See "3D Turtle", a little later in this chapter, for more information about the 3D display systems.

Turtle Heading

The turtle is always pointed in some direction, and when you use `FORWARD` or `BACK` to move the turtle, it moves off in the direction it is pointed. The turtle uses compass headings for direction. Zero degrees is up, pointed towards the top of the screen. The angles move clockwise, so 90 degrees points to the right, 180 degrees points down, and 270 degrees points left.

If you give the turtle a heading that isn't between 0 and 360 degrees, the heading is adjusted. For example, if you use `SETHEADING` to set the heading to -90 degrees, then look at the heading with `HEADING`, you'll get 270. For the most part, you can just let the turtle adjust the angles for itself, but you do need to be careful if you're working with very large numbers. The larger the number, the less accurate it will be when it's adjusted.

When you use the 3D turtle, the normal 2D heading works the same way, but it's called a longitude angle instead of a heading. A second angle, latitude, points the turtle out of the screen or into the screen; and a third angle, roll, rotates the turtle back and forth.

With the second angle, the turtle heading works like longitude and latitude angles on a globe, with the turtle in the middle of the globe, the equator lying on the screen, and the north pole towards you. Zero degrees latitude keeps the turtle in the screen plane, where it will work just like a 2D turtle. Ninety degrees points the turtle straight up, towards your eye and towards the north pole. Minus 90 degrees points the turtle straight down, into the screen and towards the south pole.

The third angle rolls the turtle back and forth. Think about how `LEFT` and `RIGHT` work with the normal, two-dimensional turtle. When you say `RIGHT 90`, the turtle doesn't point to the right, it moves 90 degrees further to the right from whatever its current heading happens to be. Floating in space, the turtle will still rotate to its own right. Now think about the turtle sitting on the screen, just like a normal 2D Logo turtle, and pointed straight up. `RIGHT 90` works just like it always did. Now imagine that the turtle is starting out pointed straight up again, but this time you say `ROLLLEFT 90`, then `RIGHT 90`. This time the turtle is pointed straight up.

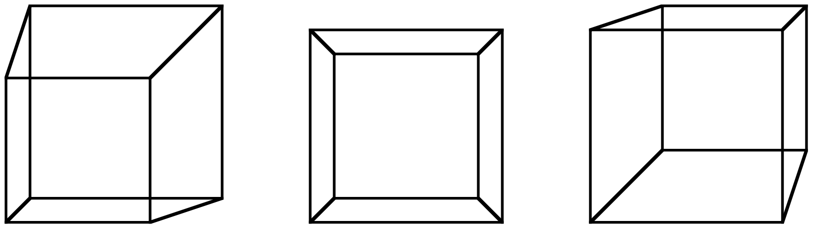
A lot of folks used to laugh at fighter pilots, who always talk with their hands, but you should try the same trick. If you're having any trouble seeing what is happening with the headings, read the last paragraph again, but this time, rotate your hand.

3D Turtle

The last two sections mentioned that HyperLogo supports a 3D turtle, rather than the standard 2D turtle you find in most Logos. "Turtle Positions" explained that the Z coordinate is zero at the screen level, with positive values moving out of the screen and negative values moving into the screen. "Turtle Headings" explained that the heading works like latitude and longitude, with the turtle at the middle of the globe, and a roll angle.

Of course, as you can easily see, your computer can't really draw in three dimensions... or can it? Actually, HyperLogo lets you draw in three dimensions two different ways.

You've seen three-dimensional objects displayed in two dimensions all your life, and the result can be very realistic. Television screens, pictures in books, photographs, and some engineering drawings are just a few examples of effective ways to put a three dimensional image on a flat display. Basically, these three dimensional display systems show you an image as if you closed one eye. In this book, we'll call that a projection display. If you draw a cube using the projection display, the back edges will look a little smaller than the front edges, like this:


Projections of a Cube

Graphics books call this a wire frame drawing. It's sort of like a three-dimensional version of a simple line drawing, with each edge represented by a wire instead of a line. With a wire frame drawing, you see all of the outlines, even the ones that would be invisible if the object were solid.

This sort of projection is about as good as it gets with a standard, flat display. You can improve on the drawing to get a more realistic picture, of course, but it's still a flat projection of a three dimensional scene, not a true three dimensional display. To get three dimensions, each of your eyes has to see a slightly different picture – the left eye needs to see a scene from an point about two to two and a half inches away from the scene your right eye sees. (The exact distance depends on the spacing between your eyes. For adults, it's close to two and a half inches; for second grade kids, it's pretty close to two inches.) Your brain uses the two different images to figure out how far away the object you are looking at happens to be.

One way to trick your brain into thinking it is seeing a three dimensional object is to show it two different pictures, one in each eye, each drawn from a slightly different viewpoint. That's exactly what HyperLogo does. When you use the stereoscopic display system, moving the turtle actually draws two lines. When you draw a complete picture, you get two complete drawings, one for the left eye and one for the right eye. The picture your left eye should see is drawn in red, and the picture for the right eye is drawn in blue, and both are drawn against a black background. If you're not wearing the special 3D glasses, the result is, well, strange. In really doesn't look like much. When you put the glasses on, though, the effect is very realistic. In fact, it's so realistic that one of my second grade beta testers actually reached out to touch the drawing! She giggled when she grabbed air about four inches in front of the computer screen.

Use the `CLEARSCREEN3D` command to turn on the stereoscopic drawing mode. `CLEARSCREEN3D` changes the background to black, clears the screen, and puts the turtle in the middle of the screen pointed up. All of the turtle graphics commands except `FILL` and `DOTP` will start working in three dimensions. You can use `SHOWTURTLE3D` to turn on the 3D turtle without clearing the screen to black, but the effect is still a lot more realistic when you draw on a black background.

Some of the normal turtle graphics commands work a little differently, since they have a new dimension and two new angles. The commands that work differently are `DOT`, `HEADING`, `POS` and `SETPOS`. The other commands support the stereoscopic display, but you use them the same way you do for a two dimensional turtle. For example, `FORWARD` still moves forward, but forward might be straight out of the screen with a three dimensional turtle.

You can also draw wire frame drawings using the projection display system. There are a couple of reasons you might want to use the projection display instead of the stereoscopic display. First, the stereoscopic display only works in black and white, since it is using color to split the screen display between your two eyes. For a color drawing, you should stick with the projection display. The other reason to use the projection display is when you don't want to use the 3D glasses, or when some people who are looking over your shoulder don't have the glasses.

The projection display turns on automatically any time you use a command with a Z coordinate or set a heading that isn't in the screen plane, and `CLEARSCREEN3D` or `SHOWTURTLE3D` haven't already been used to turn on the stereoscopic display. The commands that can turn on the projection screen are `DOT`, `ROTATEIN`, `ROTATEOUT`, `ROLLLEFT`, `ROLLRIGHT`, `SETHEADING3D`, `SETPOS` and `SETZ`.

Turtle Colors in 2D

Several of the turtle graphics commands support color. These commands use a number to pick a color, or to tell you what color something is.

There are actually only four pure colors, but they are combined in a special way when pixels of differing colors appear next to each other. This effect, called dithering, gives a total of 16 colors – but there are some limits.

First, if a line is too thin, you may not get the color you want. Normally, the Logo pen is 2 pixels wide. That will give you all 16 colors, but if you have too many lines too close together, you may end up with fringing effects that spoil the colors. To avoid these fringing effects, stick to the pure colors – color numbers 0, 3, 10 and 15.

Assuming the objects you are drawing are not too narrow, and assuming you haven't changed the color palette from the default palette, the colors are:

number	color	number	color
0	black	8	green
1	dark blue	9	blue green
2	olive	10	spring green
3	gray	11	light green
4	red	12	dark gray
5	purple	13	periwinkle blue
6	orange	14	yellow
7	peach	15	white

Turtle Colors in 3D

When you are using the stereoscopic display system, you don't have the same range of colors you get with the 2D turtle. Instead, the colors are mapped into gray scales.

With the 3D display, there are only two colors. They are 0 (black) and 1 (white).

It's not an error to use a color number for a color that doesn't exist. When you do, Logo automatically maps the color number you give into one that's available. Even numbers are drawn as black, while odd ones are drawn as white.

BACK

```
BACK distance
BK distance
```

The turtle moves back `distance` pixels. If the pen is down, a line is drawn from the old turtle location to the new turtle location.

Snippet

```
BACK 40
```

BACKGROUND

```
BACKGROUND
BG
```

Returns a number from 0 to 15; this is the current background color.

`SETBG` sets the background color.

See "Turtle Colors in 2D," earlier in this chapter, for a list of the turtle colors.

Snippet

```
MAKE "oldcolor BACKGROUND
```

CLEAN

```
CLEAN
```

Erases the card or turtle window, painting the entire window with the background color. The turtle's position and heading do not change.

Snippet

```
CLEAN
```

CLEARSCREEN

```
CLEARSCREEN
CS
```

Clears the card or turtle window, puts the turtle at [0 0], and sets the turtle heading to 0.
See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
CLEARSCREEN
```

CLEARSCREEN3D

```
CLEARSCREEN3D
CS3D
```

Clears the card or turtle window, puts the turtle at [0 0 0], and sets the turtle heading to [0 0 0]. The background color is set to black before clearing the screen. Once this command is used, all turtle drawing commands will create stereo images that appear as true 3D images when you look at the screen with the special 3D glasses.
See "3D Turtle", earlier in this chapter, for details on the 3D turtle.

Note `CLEARSCREEN3D` turns on the stereoscopic 3D display system. Anything you draw using this display model will look crummy without 3D glasses, but will display as true 3D with the glasses.

Snippet

```
CS3D
```

DOT

```
DOT [x y]
DOT [x y z]
```

Draws a dot at `[x y]`. This is a very simple drawing command, and isn't really even related to turtle graphics. The dot is drawn using the turtle's pen color, but the turtle itself does not move.

You can also draw a dot in either three dimensional display system. If you are using a 3D display and give a Z coordinate, the coordinate you give will be used. If you are using a 3D display system and don't give a Z coordinate, `DOT` uses a Z value of 0, drawing the dot in the computer screen plane.

See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Note If you give a Z coordinate, and aren't already using one of the 3D display systems, the `DOT` command will turn on the projection 3D display system.

Snippet

```
DOT [10 10]
```

DOTP

```
DOTP [x y]
```

Outputs `TRUE` if there is a dot at [x y], and `FALSE` if there is not.

Technically, `DOTP` returns `TRUE` if the pixel is not the same color as the background.

See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Note `DOTP` ignores the third dimension. It can tell you if a screen pixel is on or off, but it can't look into three dimensions to see if a point in space is on or off.

This makes a little more sense if you think about how `DOTP` works. It actually looks at the screen to see if a pixel is on or off, and what color it is. With a 3D display, though, each dot on the screen represents a line running from your eye through the screen pixel, and `DOTP` has no idea which point on that line is turned on.

Snippet

```
IF DOTP [:x :y] [MAKE "collision "TRUE]
```

FENCE

```
FENCE
```

Confines the turtle to the card or turtle window. Once this command is used, if the turtle tries to move out of the drawing area, an error is flagged and the turtle is not moved.

See `WRAP`, which wraps the turtle back onto the card when it leaves; and `WINDOW`, which allows the turtle to roam off of the card.

Snippet

```
FENCE
```

FILL

```
FILL
```

Fills an area with the current pen color. The area can be as small as a single pixel, or as large as the display area, and can be very irregular in shape. The area that is painted is made up of the point at the turtle position and all of the adjacent points that are the same color. The area stops when it hits points of a different color, although it can flow around obstacles.

Note `FILL` works in two dimensions, but not in three. It always fills an area on the screen.

Snippet

```
TO Triangle :size
REPEAT 3 [FD :size RT 120]
PU
RT 30
FD 10
PD
FILL
BK 10
LT 30
END
```

FORWARD

```
FORWARD distance
FD distance
```

The turtle moves forward `distance` pixels. If the pen is down, a line is drawn from the old turtle location to the new turtle location.

Snippet

```
REPEAT 5 [FD 30 RT 144]
```

HEADING

```
HEADING
```

Outputs the turtle's heading as a number. The heading is given in degrees, and will always be an integer greater than or equal to 0, and less than 360.

If you are using either 3D display system, `HEADING` returns a list of three numbers. The first is the standard, 2D heading, which is the longitude angle in the 3D coordinate systems. The second angle is the latitude angle. The third is the roll angle.

See "Turtle Headings", earlier in this chapter, for more information on turtle headings.

Snippet

```
MAKE "oldheading HEADING
```

HIDETURTLE

```
HIDETURTLE
HT
```

Makes the turtle invisible. When the turtle is invisible, the triangle doesn't show up on the screen, but otherwise things work just like they do when the turtle is visible.

Drawing the turtle takes time, so you might want to use this command just before a long series of complicated drawing commands. Depending on what you are drawing, you may see quite a speed improvement.

`SHOWTURTLE` or `SHOWTURTLE3D` will make the turtle visible, again.

Snippet

```
HT Draw ST
```

HOME

```
HOME
```

Moves the turtle to [0 0] and sets the turtle's heading to 0. If the pen is down, a line is drawn from the starting position to [0 0].

See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
HOME
```

LEFT

```
LEFT angle
LT angle
```

Turns the turtle left `angle` degrees.

See "Turtle Headings", earlier in this chapter, for more information on turtle headings.

Snippet

```
REPEAT 6 [FD 30 LT 60]
```

PEN

```
PEN
```

Returns `PENDOWN`, `PENUP`, `PENERASE` or `PENREVERSE`, indicating the state of the turtle pen.

Snippet

```
MAKE "oldpen PEN
```

PENCOLOR

```
PENCOLOR
```

Returns a number from 0 to 7; this is the current pen color.
`SETPC` sets the pen color.
See "Turtle Colors in 2D," earlier in this chapter, for a list of the turtle colors.

Snippet

```
SETPC PENCOLOR + 1
```

PENDOWN

```
PENDOWN
PD
```

Puts the pen down. When the pen is down, moving the turtle draws a line from the old turtle position to the new turtle position.
See `PENUP`, `PENERASE` and `PENREVERSE` for other pen effects.

Snippet

```
PENDOWN
```

PENERASE

```
PENERASE
PE
```

Turns the pen into an eraser. When the pen is erasing, moving the turtle draws a line in the background color, erasing any drawings that are under the line.

See `PENDOWN`, `PENUP` and `PENREVERSE` for other pen effects.

Snippet

```
PENERASE
```

PENREVERSE

```
PENREVERSE
PX
```

`PENREVERSE` is sort of like `PENDOWN`, since lines are drawn when the turtle is moved, but the line is drawn by reversing pixels. For example, of you draw over an area that is part black and part white, the places that are white will become black, and the places that are black will become white.

`PENREVERSE` is very handy for animation. If you draw an object with `PENREVERSE`, then draw it again in the same spot, the object is erased, no matter how complicated the background is.

With a colored pen or colored background, predicting the colors that will show up is pretty tough, but you can still erase any object, even a colored one, by drawing it twice.

See `PENDOWN`, `PENERASE` and `PENUP` for other pen effects.

Snippet

```
Square :40
PX
Square :40
```

PENUP

```
PENUP
PU
```

`PENUP` lifts the turtle pen. You can still move the turtle, but it won't draw a line.

See `PENDOWN`, `PENERASE` and `PENREVERSE` for other pen effects.

Snippet

```
REPEAT 20 [FD 2 PU FD 2 PD]
```

POS

```
POS
```

Returns the current position of the turtle. The position is returned as a list, with the X coordinate as the first element and the Y coordinate as the second element. For example, the home position is returned as [0 0].

If you are using either 3D coordinate system, POS returns a list if three numbers, instead of two numbers. The third value is the Z coordinate.

See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
MAKE "old POS
```

RIGHT

```
RIGHT angle
RT angle
```

Turns the turtle right `angle` degrees.

See "Turtle Headings", earlier in this chapter, for more information on turtle headings.

Snippet

```
REPEAT 4 [FD 30 RT 90]
```

ROLLLEFT

```
ROLLLEFT angle
RLL angle
```

Rolls the turtle left (counterclockwise) `angle` degrees.

See "3D Turtle", earlier in this chapter, for more information on the 3D turtle.

Note If you aren't already using one of the 3D display systems, `ROLLLEFT` will turn on the projection 3D display system.

Snippet

```
REPEAT 4 [Square 40 RLL 90]
```

ROLLRIGHT

```
ROLLRIGHT angle
RLR angle
```

Rolls the turtle right (clockwise) `angle` degrees.
See "3D Turtle", earlier in this chapter, for more information on the 3D turtle.

Note If you aren't already using one of the 3D display systems, `ROLLRIGHT` will turn on the projection 3D display system.

Snippet

```
REPEAT 18 [Draw :H2O RLR 20 ADDFRAME]
```

ROTATEIN

```
ROTATEIN angle
RI angle
```

Turns the turtle into the screen (down) `angle` degrees. Like `LEFT` and `RIGHT`, `ROTATEIN` is based on the current turtle heading, so it you turn the turtle in far enough, it will rotate completely around.
See "3D Turtle", earlier in this chapter, for more information on the 3D turtle.

Note If you aren't already using one of the 3D display systems, `ROTATEIN` will turn on the projection 3D display system.

Snippet

```
REPEAT 4 [FD 30 RI 90]
```

ROTATEOUT

```
ROTATEOUT angle
RO angle
```

Turns the turtle out of the screen (up) `angle` degrees. Like `LEFT` and `RIGHT`, `ROTATEOUT` is based on the current turtle heading, so it you turn the turtle out far enough, it will rotate completely around.

See "3D Turtle", earlier in this chapter, for more information on the 3D turtle.

Note If you aren't already using one of the 3D display systems, `ROTATEOUT` will turn on the projection 3D display system.

Snippet

```
REPEAT 4 [FD 40 RO 90]
```

SCRUNCH

```
SCRUNCH
```

`SCRUNCH` returns the current scrunch factor. See `SETSCRUNCH` for a description of the scrunch factor.

Snippet

```
PR SCRUNCH
```

SETBG

```
SETBG color
```

Sets the background color. `CLEAN` and `CLEARSCREEN` fill the card with the background color, and `PENERASE` causes the turtle to draw lines in the background color. When you start drawing, the background color is white.

See "Turtle Colors in 2D," earlier in this chapter, for a list of the turtle colors.

Snippet

```
SETBG 4
```

SETHEADING

```
SETHEADING angle
SETH angle
```

Changes the turtle's heading to `angle` degrees.
See "Turtle Headings", earlier in this chapter, for more information on turtle headings.

Snippet

```
SETHEADING 45
```

SETHEADING3D

```
SETHEADING3D [longitude latitude roll]
SETH3D [longitude latitude roll]
```

Changes the turtle's heading. The heading works like longitude and latitude on a globe. The first angle, `longitude`, sets the turtle heading in the screen's view plane. It works the same way as the turtle heading you set with the 2D turtle's `SETHEADING` command. The second angle, `latitude`, sets the up-and-down orientation of the turtle. Zero degrees is in the screen plane, just like it is for a 2D turtle. Ninety degrees points straight up, out of the screen. Minus ninety degrees points straight down, into the screen. The last angle rolls the turtle counterclockwise or clockwise around the heading. Zero degrees is in the screen plane. Ninety degrees rolls the turtle clockwise, so his right legs of the turtle are pointed down. Minus ninety degrees rolls the turtle counterclockwise, so his left legs are pointed straight down.
See "3D Turtle", earlier in this chapter, for more information on the 3D turtle.

Note If you aren't already using one of the 3D display systems, `SETHEADING3D` will turn on the projection 3D display system.

Snippet

```
SETHEADING [20 45 45]
```

SETPC

```
SETPC color
```

Sets the pen color. Once you change the pen color, all of the turtle drawing commands will draw lines using the new color. `DOT` also uses the pen color to decide what color dots should be.

See "Turtle Colors in 2D," earlier in this chapter, for a list of the turtle colors.

Snippet

```
SETPC 4
```

SETPENSIZE

```
SETPENSIZE x y
```

SETPENSIZE changes the width and height of lines drawn by the turtle. The default pen size is y = 1, x = 1.

Snippet

```
SETPENSIZE 4 4
```

SETPOS

```
SETPOS [x y]
SETPOS [x y z]
```

Moves the turtle to `[x y]` or `[x y z]`, drawing a line if the pen is down.

If you are using either 3D display system, and only pass a two-element list, `SETPOS` sets the Z coordinate to 0.

See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Note If you give a z coordinate, and you aren't already using one of the 3D display systems, the `SETPOS` command will turn on the projection 3D display system.

Snippet

```
SETPOS :oldpos
```

SETSCRUNCH

```
SETSCRUNCH scrunch
```

Sets the scrunch factor to `scrunch`.

The scrunch factor is used to adjust the horizontal size of pictures compared to the vertical size. For example, if squares don't look square, you can use the scrunch factor to widen your display or make your display narrower.

When you're using HyperLogo, the default scrunch factor is 2. For perfect pictures, the scrunch factor should be about 2.4.

Snippet

```
SETSCRUNCH 2.4
```

SETX

```
SETX x
```

Moves the turtle horizontally to `x`, without changing the turtle's Y position. If the pen is down, a line is drawn. The turtle's heading does not change.

See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
SETX XCOR + 10
```

SETY

```
SETY y
```

Moves the turtle vertically to `y`, without changing the turtle's X position. If the pen is down, a line is drawn. The turtle's heading does not change.

See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
SETY -YCOR
```

SETZ

```
SETZ z
```

Moves the turtle into or out of the screen to `z`, without changing the turtle's X or Y position. If the pen is down, a line is drawn. The turtle's heading does not change.

See "3D Turtle", earlier in this chapter, for details on the 3D turtle.

Note If you aren't already using one of the 3D display systems, `SETZ` will turn on the projection 3D display system.

Snippet

```
SETZ 45
```

SHOWNP

```
SHOWNP
```

Returns `TRUE` if the turtle is visible, and `FALSE` if it is not visible.
`SHOWTURTLE`, `SHOWTURTLE3D` and `HIDETURTLE` are used to make the turtle visible and invisible.

Snippet

```
IF SHOWNP [HT Draw ST] [Draw]
```

SHOWTURTLE

```
SHOWTURTLE
ST
```

Makes the turtle visible.
`HIDETURTLE` will make the turtle invisible.

△ **Important** If you are using the stereoscopic 3D display system, `SHOWTURTLE` turns it off. Be sure to use `SHOWTURTLE3D` when you are using the stereoscopic 3D display. △

Snippet

```
HT Draw ST
```

SHOWTURTLE3D

```
SHOWTURTLE3D
ST3D
```

Makes the turtle visible, and turns on the 3D stereoscopic display mode. Once this command is used, all turtle drawing commands will create stereo images that appear as true 3D images when you look at the screen with the special 3D glasses.

See "3D Turtle", earlier in this chapter, for details on the 3D turtle.

Note `SHOWTURTLE3D` turns on the stereoscopic 3D display system. Anything you draw using this display model will look crummy without 3D glasses, but will display as true 3D with the glasses.

Snippet

```
ST3D
```

TOWARDS

```
TOWARDS [x y]
```

Returns a heading that would make the turtle turn towards the point `[x y]`. If you pass this value to `SETHEADING` and draw a line that is long enough, it will pass through `[x y]`.

See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
MAKE "angle TOWARDS [40 25]
```

WINDOW

```
WINDOW
```

Allows the turtle to roam off of the card. All of the drawing commands will still work, but the results won't be visible. This is the default drawing mode.

See `WRAP`, which wraps the turtle back onto the card when it leaves; and `FENCE`, which restricts the turtle to the card.

Snippet

```
WINDOW
```

WRAP

`WRAP`

Confines the turtle to a card. If the turtle is off of the card when the command is used, it is moved to `[0 0]`. Once this command is used, if the turtle tries to move off of the screen, it wraps back onto the card. For example, if you set the heading to 90 and draw off of the right edge of the card, the turtle will wrap to the left edge and keep going.

See `FENCE`, which restricts the turtle to the card, and `WINDOW`; which allows the turtle to roam off of the card.

Snippet

```
WRAP
SETHEADING 30
FD 10000
```

XCOR

`XCOR`

Returns the current X position of the turtle.
See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
MAKE "x XCOR
```

YCOR

`YCOR`

Returns the current Y position of the turtle.
See "Turtle Positions", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
SETPOS [10 YCOR]
```

ZCOR

```
ZCOR
```

Returns the current Z position of the turtle. If you are using a 2D turtle, `ZCOR` always returns zero.

See "Turtle Positions" and "3D Turtle", earlier in this chapter, for details on the turtle coordinate system.

Snippet

```
SETZ -ZCOR
```

Other Drawing Commands

Rectangles

All of the commands you see in this section are based on rectangles, rather than the position of the turtle on the screen. A rectangle is defined by a list, where the list contains the left, top, right and bottom coordinates of the rectangle. The coordinates used match the ones used by the turtle.

For example, to quickly fill a rectangle with black, you could use the commands

```
SETPC 0
PAINTRECT [:left :top :right :bottom]
```

There are two reasons for these commands. Logo's turtle is a great way to draw lines, but it doesn't work well when you want to fill a large area with a color. These commands are very fast compared to filling the same area with the turtle. The second reason to use these commands is that drawing curves and circles in Logo is slow, and the results aren't always very pleasing. The arc and oval drawing commands you see here do the job quickly and well.

The Rectangle-based Drawing Commands

The easiest way to understand these commands isn't to read the descriptions of each command as an individual entity. It's a lot easier to understand these sixteen commands as four different ways to draw four different kinds of shape, for a total of sixteen commands.

The four different ways to draw are:

erase	Erasing an area works like drawing with the turtle when the pen is in erase mode. All of the pixels in the shape are set to the current background color.
frame	Framing a shape outlines the shape. It's like tracing the shape using turtle drawing commands. The size of the lines used to outline the shape match the turtle – you can change the size of the lines with the turtle's `SETPENSIZE` command. The turtle's color determines the color of the outline, too.
invert	Inverting a shape works as if you drew every pixel in the shape with the turtle with a pen color of white and a pen mode of reverse. All black pixels become white, for example, and all white pixels become black. Exactly what happens to the colored pixels is a little hard to predict without a complicated color map. The overall effect is very useful, though. If you invert the same shape twice, you end up with the same picture you started with. It's a great way to flash and area of the screen!
paint	Painting a shape fills all of the pixels in the shape with the current turtle pen color.

The four different shapes are:

rect	A rect is a rectangle. It's just a box.
oval	An oval is a squashed circle that fits inside of a box defined by a rectangle. Of course, if the rectangle is a square, the oval isn't squashed at all, and you get a circle.
arc	An arc is a slice out of an oval. The slice starts with an angle called `start`, which is measured in degrees, starting at the top of the oval and turning clockwise – just like a turtle heading. A second angle, `angle`, is the size of the slice. It's given in degrees, too.
rrect	This is a rounded rectangle. It's a rectangle with the corners rounded off. The rounded part of the corner is formed by slicing an oval into four chunks, and using one chunk of the oval for each corner. Rounded rectangles come with a `height` and `width` parameter; these are the height and width of the oval that gets cut up to form the corners.

Here's a pair of small procedures that shows what all of these commands actually do on a screen. To see the shapes, type in the procedures and then run `DemoRects`, either from a button's script or from a text window.

```
TO DemoRects
SETBG 15
CS
PU
SETPOS [-50 -50]
```

```
   PD
   SETPC 0
   HT
   REPEAT 25 [FD 100 RT 90 FD 2 RT 90 FD 100 LT 90 FD 2 LT 90]
   MAKE "r [-45 45 -30 30]
   ERASEARC :r 45 225
   AdjustRect
   ERASEOVAL :r
   AdjustRect
   ERASERECT :r
   AdjustRect
   ERASERRECT :r 10 10
   MAKE "r [-45 20 -30 5]
   FRAMEARC :r 45 225
   AdjustRect
   FRAMEOVAL :r
   AdjustRect
   FRAMERECT :r
   AdjustRect
   FRAMERRECT :r 10 10
   MAKE "r [-45 -5 -30 -20]
   INVERTARC :r 45 225
   AdjustRect
   INVERTOVAL :r
   AdjustRect
   INVERTRECT :r
   AdjustRect
   INVERTRRECT :r 10 10
   MAKE "r [-45 -30 -30 -45]
   PAINTARC :r 45 225
   AdjustRect
   PAINTOVAL :r
   AdjustRect
   PAINTRECT :r
   AdjustRect
   PAINTRRECT :r 10 10
   END

   TO AdjustRect
   MAKE "r (LIST 25 + ITEM 1 :r ITEM 2 :r 25 + ITEM 3 :r ITEM 4
:r)
   END
```

ERASEARC

```
ERASEARC rect start angle
```

Erases an area on the screen. The area is an arc with the center at the middle of `rect`. The arc starts `start` degrees from the vertical, and is `angle` degrees wide.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

ERASEOVAL

```
ERASEOVAL rect
```

Erases an area on the screen. The area is an oval with edges that touch the edge of the rectangle `rect`.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

ERASERECT

```
ERASERECT rect
```

Erases all of the pixels inside of `rect`.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

ERASERRECT

```
ERASERRECT rect width height
```

Erases an area on the screen. The area is rectangle formed by `rect`, but the corners of the rectangle are rounded by a quarter-oval. `Height` is the height of the complete oval that is cut up to form the corners, while `width` is the width of the oval.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

FRAMEARC

```
FRAMEARC rect start angle
```

Draws the curved outline for an arc. The arc's center is in the middle of `rect`. The arc starts `start` degrees from the vertical, and is `angle` degrees wide. The outline uses the same pen size and color as turtle lines.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

FRAMEOVAL

```
FRAMEOVAL rect
```

Outlines the oval formed by `rect`. The outline uses the same pen size and color as turtle lines.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

FRAMERECT

```
FRAMERECT rect
```

Draws an outline around the rectangle defined by `rect`. The outline uses the same pen size and color as turtle lines.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

FRAMERRECT

```
FRAMERRECT rect width height
```

Draws a rectangle in the area defined by `rect`, but the corners of the rectangle are rounded by a quarter-oval. `Height` is the height of the complete oval that is cut up to form the corners, while `width` is the width of the oval. The outline uses the same pen size and color as turtle lines.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

INVERTARC

```
INVERTARC rect start angle
```

Reverses the color of all of the pixels in an area. The area is an arc with the center at the middle of `rect`. The arc starts `start` degrees from the vertical, and is `angle` degrees wide.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

INVERTOVAL

```
INVERTOVAL rect
```

Reverses the color of all of the pixels in an area. The area is an oval with edges that touch the edge of the rectangle `rect`.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

INVERTRECT

```
INVERTRECT rect
```

Reverses the color of all of the pixels inside `rect`.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

INVERTRRECT

```
INVERTRRECT rect width height
```

Reverses the color of all of the pixels in an area. The area is rectangle formed by `rect`, but the corners of the rectangle are rounded by a quarter-oval. `Height` is the height of the complete oval that is cut up to form the corners, while `width` is the width of the oval.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

PAINTARC

```
PAINTARC rect start angle
```

Paints all of the pixels in area on the screen, using the current turtle pen color. The area is an arc with the center at the middle of `rect`. The arc starts `start` degrees from the vertical, and is `angle` degrees wide.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

PAINTOVAL

```
PAINTOVAL rect
```

Paints all of the pixels in area on the screen, using the current turtle pen color. The area is an oval with edges that touch the edge of the rectangle `rect`.

See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

PAINTRECT

```
PAINTRECT rect
```

Paints all of the pixels in `rect`, using the current turtle pen color.
See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

PAINTRRECT

```
PAINTRRECT rect width height
```

Paints all of the pixels in area on the screen, using the current turtle pen color. The area is rectangle formed by `rect`, but the corners of the rectangle are rounded by a quarter-oval. `Height` is the height of the complete oval that is cut up to form the corners, while `width` is the width of the oval.
See "The Rectangle-based Drawing Commands," earlier in this section, for an example.

Movies

ADDFRAME

```
ADDFRAME
```

If the current drawing window is a movie window, `ADDFRAME` adds a new frame after the frame that is visible and displays the new frame. The position of the turtle does not change, but the new frame is cleared as if `CLEAN` were used.
If the current drawing window is not a movie window, `ADDFRAME` does nothing.

Snippet

```
REPEAT 18 [Flag 30 LT 20 ADDFRAME]
```

CLOSEMOVIE

```
CLOSEMOVIE
```

CLOSEMOVIE closes the last movie opened by OPENMOVIE. If there are no open movies, CLOSEMOVIE does nothing.

Closing a movie when you are finished with it is not always necessary, since HyperLogo will close it automatically when you edit a script or leave the stack. Closing the movie does free up any memory used by the movie, though.

Snippet

```
CLOSEMOVIE
```

DELETEFRAME

```
DELETEFRAME
```

If the current drawing window is a movie window, and if there are more than one frames in the movie, DELETEFRAME deletes the visible frame and leaves the display on the frame after the one that is visible. If the frame that is deleted is the last frame, the frame before is displayed.

If the current drawing window is not a movie window, or if there is only one frame in the movie, DELETEFRAME does nothing.

Snippet

```
REPEAT 18 [Flag 30 LT 20 ADDFRAME]
DELETEFRAME
```

INSERTFRAME

```
INSERTFRAME
```

If the current drawing window is a movie window, INSERTFRAME adds a new frame before the frame that is visible and displays the new frame. The position of the turtle does not change, but the new frame is cleared as if CLEAN were used.

If the current drawing window is not a movie window, INSERTFRAME does nothing.

Snippet

```
REPEAT 18 [INSERTFRAME Flag 30 RT 20]
```

LASTFRAME

`LASTFRAME`

If the current drawing window is a movie window, and if the visible frame is not the first frame in the movie, `LASTFRAME` moves to the frame before the visible frame.

If the current drawing window is not a movie window, or if the first frame is being displayed, `LASTFRAME` does nothing.

Snippet

```
TO TwoFlagMovie
REPEAT 18 [Flag 30 LT 20 ADDFRAME]
REPEAT 18 [LASTFRAME]
PU
SETPOS [50 50]
PD
REPEAT 18 [Flag 10 LT 40 NEXTFRAME]
DELETEFRAME
END
```

NEXTFRAME

`NEXTFRAME`

If the current drawing window is a movie window, and if the visible frame is not the last frame in the movie, `NEXTFRAME` moves to the frame after the visible frame.

If the current drawing window is not a movie window, or if the last frame is being displayed, `NEXTFRAME` does nothing.

Snippet

```
NEXTFRAME
```

OPENMOVIE

```
OPENMOVIE name clip position flags
```

`OPENMOVIE` opens a movie file, closing any previously opened movie file in the process. Once the movie file is open, `PLAY` will play the movie.

`Name` is the file name for the movie file. See "File Names," earlier in this chapter, for information about file names.

`Clip` is a rectangle used to pick the part of the movie to play. Each movie extends from -160 to 160 in the horizontal direction, and 100 to -100 in the vertical direction. Only the part of the movie enclosed in this rectangle is actually played by the `PLAY` command, though. See "Rectangles," earlier in this chapter, for a description of rectangles.

`Position` determines where the movie will be located on the card. The card itself uses turtle coordinates, with [0 0] in the middle of the card. `Position` is the location of the top, left corner of the movie. It is a list of two values; the first is the horizontal turtle coordinate, which is the left edge of the movie, and the second is the vertical turtle coordinate, which is the top of the movie.

`Flags` should be either 0 or 1. If `flags` is 0, the movie will play one time, from start to finish, and stop. The last frame will remain visible. If `flags` is 1, the movie will cycle repeatedly until a key is pressed or the mouse button is pushed.

Snippet

```
OPENMOVIE "Flag [-50 50 50 -50] [-40 40] 1
```

PLAY

```
PLAY
```

While you are editing scripts in HyperLogo, if the current drawing window is a movie window, PLAY plays the movie. The PLAY command works exactly as if the play button on the movie window were pressed.

While a script is running, `PLAY` plays the last movie opened with OPENMOVIE.

If the current drawing window is not a movie window, or if the command is used in a script and there is no open movie, `PLAY` does nothing.

Snippet

```
TwoFlagMovie
PLAY
```

Fonts

GETFONTINFO

`GETFONTINFO`

`GETFONTINFO` returns a list of three integers. In the order they appear in the list, the values represent:

ascent	This is the distance from the baseline of the character to the top of the character. For capital letters like K, this is also the height of the letter.
descent	This is the distance from the baseline of the character to the bottom of characters that go below the baseline, like "y" and "g".
leading	This is the recommended space between two lines.

Use the `TEXTWIDTH` command to find the width of a line of text.

Note Leading is pronounced like the metal lead, not like "lead a horse." It comes from the not so distant past, when strips of lead were inserted between moveable lead type to increase the space between lines.

Tip Adding ascent and descent gives the height of a line of text. If you add leading, too, you get the distance between two lines of text.

<u>Snippet</u>

```
MAKE "info GETFONTINFO
MAKE "lineSize FIRST :info + ITEM 2 :info + LAST :info
```

SETBACKCOLOR

```
SETBACKCOLOR color
```

SETBACKCOLOR changes the color of the background for text. The colors you can use are limited to black, white, purple and green.

The background for the text includes a rectangle that surrounds the character.

Snippet

```
SETBACKCOLOR 3
```

SETFONTFAMILY

```
SETFONTFAMILY family
```

SETFONTFAMILY changes the font family, which controls the shape of the letters. You can use pretty much any number. If it doesn't match a font in your font folder, Logo picks the closest font it can find.

Here are some of the font numbers you can pick from:

number	name
0	System font (this is the default font)
2	New York
3	Geneva
4	Monaco
5	Venice
6	London
7	Athens
9	Toronto
11	Cairo
20	Times
21	Helvetica
22	Courier
23	Symbol
-2	Shaston

Snippet

```
SETFONTFAMILY 20
```

SETFONTSIZE

```
SETFONTSIZE size
```

`SETFONTSIZE` changes the size of the letters that are drawn in the turtle window. Font sizes can range from 1 to 255. In general, start with a font size of 12 or 14, then change the size based on what you see.

Tip Some font sizes look better with a particular font than others. That's because the letters are drawn like pictures, but there isn't a separate picture of each letter saved for all 255 possible sizes. The most common predrawn sizes are 9, 10, 12, 14, 16, 24, and 36, although you'll find a number of fonts that don't have all of these sizes, and a few that have more. If you ask for a size that isn't available, Logo does its best to create a font by enlarging or shrinking one of the font sizes that are recorded as pictures. If you don't like the result, try increasing or decreasing the font size.

Snippet

```
SETFONTSIZE 24
```

SETFONTSTYLE

```
SETFONTSTYLE style
```

`SETFONTSTYLE` changes the way letters are drawn to the screen. There are five different basic styles. Each style has a number; setting the font style to that number picks the style.

number	style
0	plain text
1	**bold**
2	*italic*
4	underline
8	outline
16	shadow

"Plain text" just means you aren't using a special style. All of the other styles can be combined with each other. To combine two or more styles, add all of the style numbers together. For example

```
    SETFONTSTYLE 3
```

gives ***bold italic*** text.

Note Some fonts don't support all of these styles. If you can't get the style you want, try switching the font family.

SETFORECOLOR

```
SETFORECOLOR color
```

SETFORECOLOR changes the color of the text drawn in the turtle window. The colors you can use are limited to black, white, purple and green.

Snippet

```
    SETFORECOLOR 3
```

TEXTWIDTH

```
TEXTWIDTH word
```

TEXTWIDTH returns the width of a string. GETFONTINFO returns the height.

Snippet

```
    TO Center :text :width
    LOCAL "x
    MAKE "x XCOR
    PU
    SETX :x + (:width - TEXTWIDTH :text) / 2
    PD
    TYPE :text
    PU
    SETX :x
    PD
    END
```

Speech

If you have the speech tools from First Byte installed, Logo can talk. Getting Logo to talk is actually very simple; you just tell it to say a string. There are a lot of options, though. You can control whether Logo uses a male or female voice, how fast Logo talks, and a number of other variables that control the sound of the speech.

Note You must have the First Byte speech tools installed to use these commands. If these tools are not available, these commands don't do anything.

It might seem strange not to get an error, but there is a reason for this. If you want to write a program that uses optional speech, you don't want the program to fail when a friend who doesn't have the speech tools runs the program. If the speech tools flagged an error, the program wouldn't work right unless the speech tools were installed. This way, if speech isn't available, the program just ignores the speech commands and moves on.

You can add speech to Logo on the Apple IIGS by buying and installing *Talking Tools* from the Byte Works.

DICT

```
DICT word phonetic
```

Logo does a pretty good job of reading English words and converting them into speech. There are a lot of exceptional words in English that aren't pronounced the way they are spelled, though, and Logo needs a little help with these words. `DICT`, combined with `PHONETIC`, gives you an easy way to teach Logo how to say words that aren't pronounced the way they are spelled. Use `PHONETIC` to convert a word that sounds correct into a phonetic string, then give the result, along with the actual spelling of the word, to `DICT`. From that point on, Logo will pronounce the word correctly.

Snippet

```
DICT "Kansas PHONETIC "kansus
```

ERDICT

```
ERDICT
```

`ERDICT` erases all of the words in the dictionary. You put words into the dictionary with `DICT`.

Snippet

```
ERDICT
```

PHONETIC

```
PHONETIC word
```

`PHONETIC` converts `word` to a string of phonetic characters. In most cases, you'll use the result to enter the correct pronunciation for a word into the dictionary using the `DICT` command.

Snippet

```
DICT "Toto PHONETIC towtow
```

PITCH

```
PITCH value
```

`PITCH` controls the overall frequency of the voice. You can use any number from 0 to 9; if you use a number outside of this range, `PITCH` adjusts the number to the correct range.

The starting value for `PITCH` is 5. Higher numbers give a higher pitched voice, which sounds a little squeakier. Low pitch numbers give a lower, rumbling voice.

Snippet

```
PITCH 7
```

SAY

```
SAY word
```

`SAY` reads `word`.

▲ **Warning** You should make sure the string isn't too long. Only the first 255 characters or so are actually read. ▲

Snippet

```
SAY "We're\ not\ in\ Kansas\ anymore\,\ Toto.
```

SPEED

```
SPEED value
```

SPEED controls how fast Logo talks. You can use any number from 0 to 9; if you use a number outside of this range, SPEED adjusts the number to the correct range.

The starting value for SPEED is 5. Higher numbers cause Logo to talk faster, while lower numbers cause Logo to talk slower.

Snippet

```
SPEED 7
```

TONE

```
TONE tone
```

TONE adjusts the quality of the voice. You can use two parameters for tone, bass or treble. In both cases, the input is a word, and the case doesn't matter. (BASS works just as well as bass.)

Snippet

```
TONE "bass
```

VOICE

```
VOICE gender
```

VOICE changes the gender of Logo. When Logo starts, it uses a male voice.

```
VOICE "female
```

changes to a female voice, and

```
VOICE "male
```

changes back to the male voice.

The parameter is not case sensitive. MALE works as well as male.

VOLUME

```
VOLUME value
```

VOLUME makes Logo talk louder or quieter. You can use any number from 0 to 9; if you use a number outside of this range, VOLUME adjusts the number to the correct range.

The starting value for VOLUME is 5. Higher numbers cause Logo to talk louder, while lower numbers cause Logo to talk quieter.

Snippet

```
VOLUME 9
```

Workspace Management

BURY

```
BURY word
BURY list
```

BURY buries a procedure or a list of procedures. You can still use a buried procedure, calling it just like you did before it was buried, but the calls that effect all of the procedures in the workspace don't effect buried procedures.

The calls that see unburied procedures, but don't effect procedures once they are buried, are ERALL, ERPS, POALL, POPS, POTS and SAVE.

You can still print or erase a buried procedure, but you have to use one of the commands that requires a specific procedure name as input.

Note BURY only works with your own procedures. You can't bury a primitive.

Snippet

```
BURY "Flag
```

BURYALL

```
BURYALL
```

BURYALL buries all of the procedures, variables and property lists in the workspace.

Snippet

```
BURYALL
```

BURYNAME

```
BURYNAME word
BURYNAME list
```

BURYNAME buries a global variable or a list of global variables. You can still use a buried variable, accessing it with the colon operator or setting the value with MAKE, just like you did before it was buried, but the calls that effect all of the global variables in the workspace don't effect buried variables.

The calls that see unburied variables, but don't effect variables once they are buried, are ERALL, ERNS, POALL, PONS and SAVE.

Snippet

```
BURYNAME [x y z]
```

BURYPLIST

```
BURYPLIST word
BURYPLIST list
```

BURYPLIST buries a property list or a list of property lists. You can still use a buried property list, just like you did before it was buried, but the calls that effect all of the property lists in the workspace don't effect buried property lists.

The calls that see unburied property lists, but don't effect property lists once they are buried, are ERALL, ERPROPS, POALL, PPS and SAVE.

Snippet

```
BURYPLIST "Susan
```

EDIT

```
EDIT word
EDIT list
```

`EDIT` opens a new edit window. You can give `EDIT` a single name or a list of names. Each of the names must be the name of an unburied procedure, variable or property list in the workspace. The new window will show all of the objects you listed as a parameter to `EDIT`.

See "New Edit Window" in Chapter 7 for a technical description of edit windows. Chapter 2 introduces edit windows in a tutorial.

△ **Important** EDIT is only available from a text window, or from a procedure executed from a text window. Do not use this command from a script. △

Snippet

```
EDIT "flag
```

EDITS

```
EDITS
```

`EDITS` opens a new edit window. The edit window will contain all of the unburied procedures, variables and property lists currently in the workspace.

See "New Edit Window" in Chapter 7 for a technical description of edit windows. Chapter 2 introduces edit windows in a tutorial.

△ **Important** EDITS is only available from a text window, or from a procedure executed from a text window. Do not use this command from a script. △

Snippet

```
EDITS
```

ERALL

```
ERALL
```

Erases all of the global variables, property lists and procedures in the workspace. ERALL does the same thing as these three commands:

```
ERNS
ERPROPS
ERPS
```

Snippet

```
ERALL
```

ERASE

```
ERASE name
ERASE list
```

Erases the named procedures from the workspace. You can use a single name, as in

```
ERASE "Fractal
```

or a list of names, like

```
ERASE [Oval Square Star]
```

ERN

```
ERN name
ERN list
```

Erases the named variables from the workspace. This completely deletes both the variable itself and any value it might have. You can use a single name, as in

```
ERN "Fred
```

or a list of names, like

```
ERN [Fred Sam Psi]
```

ERNS

```
ERNS
```

Erases all of the variables in the workspace.

Snippet

```
ERNS
```

ERPROPS

```
ERPROPS
```

Erases all property lists from the workspace.

Tip There isn't a command to erase an individual property list, but you can get rid of a property list by using REMPROP to remove all of the properties from the property list.

Snippet

```
ERPROPS
```

ERPS

```
ERPS
```

Erases all procedures from the workspace.

Snippet

```
ERPS
```

NODES

```
NODES
```

Displays the number of free nodes.

This is the number of nodes actually available in the free node pool. To find out how many nodes are actually available, use `RECYCLE` first to collect all of the free nodes into the free node pool.

Snippet

```
NODES
```

PO

```
PO name
PO list
```

Prints the definitions for all of the named procedures. You can use a single name, as in

```
PO "Fractal
```

or a list of names, like

```
PO [Oval Square Star]
```

POALL

```
POALL
```

Prints all of the global variables, property lists and procedures in the workspace. `POALL` does the same thing as these three commands:

```
POPS
PPS
PO
```

Snippet

```
POALL
```

PON

```
PON name
PON list
```

Prints the definitions for the named variables. You can use a single name, as in

```
PON "Fred
```

or a list of names, like

```
PON [Fred Sam Psi]
```

PONS

```
PONS list
```

Prints the definitions for all of the global variables. This is a quick way to see what variables are defined, and what the current values are.

Snippet

```
PONS
```

POPS

```
POPS
```

Prints the definitions for all of the procedures in the workspace.

Snippet

```
POPS
```

POT

```
POT name
POT list
```

Prints the header line for all of the named procedures. You can use a single name, as in

```
POT "Fractal
```

or a list of names, like

```
POT [Oval Square Star]
```

The header line is the line that starts with TO and shows the procedure name and parameters.

POTS

```
POTS
```

Prints the header line (the one that starts with TO and shows the procedure name and parameters) for all of the procedures in the workspace.

Snippet

```
POTS
```

PPS

```
PPS
```

Prints all of the property lists in the workspace.

Snippet

```
PPS
```

RECYCLE

```
RECYCLE
```

`RECYCLE` does garbage collection. Garbage collection scans all of the nodes in the node space, freeing up any nodes that aren't being used.

To understand this command, and why it is useful, you have to understand a little about the way Logo actually works. As you type commands, and as the program runs, Logo is constantly grabbing chunks of memory called nodes. These nodes are used for temporary values, for building lists, for creating strings, and for function return values. If Logo needs a node, but all of them have been used, it does garbage collection. Basically, garbage collection is when Logo takes a moment to clean up after itself, looking at each of the nodes in the node space to see if it's really being used, or was just used and thrown away. The nodes that aren't being used are collected so they can be used again.

The problem with garbage collection is that it takes time, and you don't want it to happen in the middle of a time-critical part of your program. `RECYCLE` forces Logo to do garbage collection, even if it still has some free nodes. That delays the time until Logo will need to do garbage collection automatically.

Don't overuse `RECYCLE`! This call takes time, and if you use `RECYCLE` to often, the whole program will seem slow.

Snippet

```
RECYCLE
```

UNBURY

```
UNBURY word
UNBURY list
```

UNBURY unburies a procedure or a list of procedures.

Note See BURY for more information about buried procedures.

<u>Snippet</u>

```
UNBURY "Flag
```

UNBURYALL

```
UNBURYALL
```

UNBURYALL unburies all of the buried procedures, variables and property lists in the workspace.

<u>Snippet</u>

```
UNBURYALL
```

UNBURYNAME

```
UNBURYNAME word
UNBURYNAME list
```

UNBURYNAME unburies a buried variable or a list of variables.

Note See BURYNAME for more information about buried variables.

<u>Snippet</u>

```
UNBURYNAME [x y z]
```

UNBURYPLIST

```
UNBURYPLIST word
UNBURYPLIST list
```

`UNBURYPLIST` unburies a buried property list or a list of property lists.

Note See `BURYPLIST` for more information about buried property lists.

Snippet

```
UNBURYPLIST "Susan
```

Callbacks

Callbacks are the communication path between HyperStudio and HyperLogo. They are used whenever you want Logo to tell HyperStudio to do something. For example, of you want to move to another card from a Logo script, you will end up using a callback – even though the callback might be hidden inside another Logo command or procedure.

The commands in this section give you a very direct way to make callbacks to HyperStudio. In most cases, you won't use these commands directly. The best way to make callbacks is to use one of the library procedures that you can find in the file CallBacks . You can either open this file and paste the callbacks you need into scripts, or simply open the file into an edit window, then close the window. Using the edit window will enter all of the library procedures into the workspace for your stack. They'll take up a bit of room, but you won't have to worry about whether a particular callback has been loaded.

The callbacks in the CallBacks file are listed in Chapter 9.

CALLBACK

```
CALLBACK
```

Makes a callback to HyperStudio.

▲ **Warning** Callbacks are low-level calls to HyperStudio. You must set up the parameters using the other calls in this section before you use this call. If you make a mistake, the results can be dramatic! In extreme cases, you might have to reboot your computer. ▲

CBDISPOSEPARAMH

`CBDISPOSEPARAMH number`

Some callbacks return a handle. If you won't be using the contents of handle, use this call to dispose of the memory used by the handle.

CBGETPARAMINT

`CBGETPARAMINT number`

Some callbacks return an integer value in one of the five callback parameter variables. You can read the returned value with this call. `Number` is the number of the callback parameter to read; they are numbered 1 to 5.

Tip Some of the callbacks return handles. You can safely read a handle with this call, so long as the only thing you use the handle for is to pass the value, unchanged, as the parameter to another callback.

CBGETPARAMPOINT

`CBGETPARAMPOINT number`

Some callbacks return a pointer to a point in one of the five callback parameter variables. This call converts the point to a list. The coordinates are converted to turtle coordinates. The list has the X coordinate first, followed by the Y coordinate, just like the list returned by `POS`.

CBGETPARAMRECT

`CBGETPARAMRECT number`

Some callbacks return a pointer to a rectangle in one of the five callback parameter variables. This call converts the rectangle to a list. The coordinates are converted to turtle coordinates. The rectangle list uses the same order and coordinate system as the other Logo commands that use rectangles, like `PAINTRECT`.

CBGETPARAMSTR

`CBGETPARAMSTR number`

Some callbacks return a pointer to a string value in one of the five callback parameter variables. You can read the string with this call, which returns the string as a Logo word. `Number` is the number of the callback parameter to read; they are numbered 1 to 5.

CBGETPARAMSTRH

`CBGETPARAMSTRH number`

Some callbacks return the handle of a string value in one of the five callback parameter variables. You can read the string with this call, which returns the string as a Logo word. `Number` is the number of the callback parameter to read; they are numbered 1 to 5.

Note This call disposes of the handle after the text has been converted to a Logo string, so you do not need to call `CBDISPOSEPARAMH` to dispose of the handle.

CBSETCMD

`CMSETCMD command`

Sets the callback command number to `command`. You must use this call before every call to `CALLBACK` to tell HyperStudio which callback you are about to do.

CBSETPARAMINT

`CBSETPARAMINT number value`

Sets one of the five callback parameters to an integer.

`number` is the number of the HyperStudio callback parameter to set. The parameters are numbered 1 to 5.

`value` is the value to set the parameter to.

CBSETPARAMPOINT

`CBSETPARAMPOINT number point`

Sets one of the five callback parameters to a point.

`number` is the number of the HyperStudio callback parameter to set. The parameters are numbered 1 to 5.

`point` is a list, given in turtle coordinates with the X coordinate first, and the Y coordinate second. This is the same format that is used by the turtle graphics commands, like `POS`.

CBSETPARAMRECT

`CBSETPARAMRECT number rect`

Sets one of the five callback parameters to a rectangle.

`number` is the number of the HyperStudio callback parameter to set. The parameters are numbered 1 to 5.

`rect` is a list, containing the four coordinates that define the rectangle. It uses the same format as the other Logo graphics commands that use rectangles, like `PAINTRECT`.

CBSETPARAMSTR

`CBSETPARAMSTR number value`

Sets one of the five callback parameters to a string.

`number` is the number of the HyperStudio callback parameter to set. The parameters are numbered 1 to 5.

`value` is the logo word to pass as the parameter.

△ **Important** Some callbacks ask for a string pointer parameter, while others ask for a string handle. Use this call for callbacks that expect a pointer, and `CBSETPARAMSTRH` for those that expect a string handle. △

CBSETPARAMSTRH

`CBSETPARAMSTRH number value`

Sets one of the five callback parameters to a string handle.

`number` is the number of the HyperStudio callback parameter to set. The parameters are numbered 1 to 5.

`value` is the logo word to pass as the parameter.

△ **Important** Some callbacks ask for a string pointer parameter, while others ask for a string handle. Use this call for callbacks that expect a handle, and `CBSETPARAMSTR` for those that expect a string pointer. △

Chapter 9 – The CallBack Library

Callbacks

The procedures you see in this chapter are defined like Logo commands, but they are actually library procedures. Before you can use these procedures, you need to copy the procedure from the CallBacks file, located in your HyperStudio folder.

CALLNBA

```
CALLNBA :NBAname :param
```

Calls a new button action (NBA).

`NBAname` is the name of the NBA to call.

`param` is a word that is passed to the NBA. Whether a parameter is used at all, and what the string means if one is use, is up to the individual NBA. Some NBAs can use a string parameter, and some can't.

DRAWTOOFFSCREEN

```
DRAWTOOFFSCREEN
```

Tells HyperStudio to put all of the stuff HyperLogo draws on the card, instead of drawing it on the visible screen.

Normally, when you draw with HyperLogo, the things you draw disappear as soon as the card needs to be drawn again – either because you covered up part of the card and uncovered it again, or because you flipped to another card and came back. With this call, you can tell HyperStudio to save what you draw. Anything you draw after making this call will be saved as part of the permanent background for the card, just as if you had painted the card with the paint tools.

There are three other calls you should make in any script that uses this call.

As soon as you finish drawing the things you want to become a part of the card, call `DRAWTOSCREEN` to tell HyperStudio to put anything else in the visible window, instead of on the card.

Next, call `SETBKGDDIRTY` to tell HyperStudio the background for the card has changed.

Finally, call `REDRAWCARD` to redraw the card you are looking at. None of the things you draw will show up until you make this call.

DRAWTOSCREEN

```
DRAWTOSCREEN
```

Tells HyperStudio to put all of the stuff HyperLogo draws on the visible screen. This is what HyperStudio normally does. The only reason to use this callback is to reverse the effect of `DRAWTOOFFSCREEN`.

FINDTEXT

```
FINDTEXT :text :flags
```

Searches text fields for a string. If the string is found, the card containing the text item is displayed, and the text is selected.

`text` is the Logo word to search for.

`flags` controls exactly how the search is performed. You can combine characteristics from the flags table by adding all of the values for the characteristics you want.

value	characteristic
1	Case sensitive search. With this flag set, "Fred" and "fred" won't match.
2	Search fields that are marked read only.
4	Search fields that can be edited.
8	If this flag is set, once the search reaches the end of the stack, it starts over at the beginning.

GETCURRENTSCORE

```
GETCURRENTSCORE
```

HyperStudio buttons have a built-in testing ability. `GETCURRENTSCORE` lets you check the status of the testing functions. It returns a list with two elements.

The first object in the list is the number of correct choices.

The second object in the list is the total number of choices made. You can get the number of incorrect answers by subtracting the first value.

GETFIELDTEXT

```
GETFIELDTEXT :card :itm
```

Returns all of the text in a field. The text is returned as a Logo word; you can break it up easily with `PARSE`.

`card` is the card containing the item. Use the empty list, [], for the current card.
`itm` is the name of the item.

HIDEITEM

```
HIDEITEM :card :itm :type
```

Hides an item. A hidden object isn't visible, and can't be selected.
`card` is the card containing the item. Use the empty list, [], for the current card.
`itm` is the name of the item to hide.
`type` is the kind of item to hide. It must be one of:

type	kind of object
0	button
1	text field
2	graphic object

MOVENEXT

```
MOVENEXT
```

Moves to the card after the one currently displayed.

MOVEPREV

```
MOVEPREV
```

Moves to the card before the one currently displayed.

MOVETOCARD

```
MOVETOCARD :name
```

Moves to the card whose name is `name`.

MOVETOFIRST

`MOVETOFIRST`

Moves to the first card in the stack.

MOVETOLAST

`MOVETOLAST`

Moves to the last card in the stack.

REDRAWCARD

`REDRAWCARD`

Redraws the current card.

SETBKGDDIRTY

`SETBKGDDIRTY`

Tells HyperStudio that the permanent background for the card has changed. This call is usually used in conjunction with `DRAWTOOFFSCREEN`, which makes the things you draw with HyperLogo a permanent part of the card, just as if they were drawn with the paint tool. See `DRAWTOOFFSCREEN` for a complete description.

SETFIELDTEXT

`SETFIELDTEXT :card :itm :text`

Sets the text for a text item. The text should be supplied as a Logo word.
`card` is the card containing the item. Use the empty list, [], for the current card.
`itm` is the name of the item.
`text` is the text to set. All of the text in the text item is replaced with this text.

SETITEMRECT

```
SETITEMRECT :card :itm :type :rect
```

Sets the rectangle for an item. The rectangle for an item controls both its size and position. The rectangle is given in turtle coordinates, as described in Chapter 8.

`card` is the card containing the item. Use the empty list, [], for the current card.

`itm` is the name of the item.

`type` is the kind of item. It must be one of:

type	kind of object
0	button
1	text field
2	graphic object

`rect` is the rectangle for the item.

SHOWITEM

```
SHOWITEM :card :itm :type
```

Shows an item that has been hidden with `HIDEITEM`.

`card` is the card containing the item. Use the empty list, [], for the current card.

`itm` is the name of the item to show.

`type` is the kind of item to show. It must be one of:

type	kind of object
0	button
1	text field
2	graphic object

Appendix A – Logo Command Summary

Procedures

```
COPYDEF old-name new-name
DEFINE name list
DEFINEDP name
PRIMITIVEP name
TEXT name
TO name parm1 parm2 ...
```

Variables

```
LOCAL name
MAKE name object
NAME name object
NAMEP name
THING variable
```

Words and Lists

```
ASCII word
BEFOREP word1 word2
BUTFIRST object
   BF object
BUTLAST object
   BL object
CHAR number
COUNT object
EMPTYP object
EQUALP object1 object2
FIRST object
FLOATP object
FPUT object list
INTEGERP object
ITEM integer object
LAST object
LIST object1 object2
   (LIST object1 object2 object3
     ... )
LISTP object
LOWERCASE word
LPUT object list
MEMBER object1 object2
MEMBERP object1 object2
NUMBERP object
PARSE word
SENTENCE object1 object2
   SE object1 object2
   (SENTENCE object1 object2
     object3 ... )
   (SE object1 object2 object3
     ... )
UPPERCASE word
WORD word1 word2
   (WORD word1 word2 word3 ... )
WORDP object
```

Property Lists

```
GPROP name property
PLIST name
PPROP name property value
REMPROP name property
SETPLIST name list
```

Numbers and Arithmetic

```
ABS value
ARCCOS value
ARCSIN value
ARCTAN value
ARCTAN2 x y
COS angle
DIFFERENCE value1 value2
EXP number
FLOAT number
FORM number width digits
INT number
INTQUOTIENT numerator
  denominator
LN number
POWER x y
PRODUCT number1 number2
   (PRODUCT number1 number2
     number3 ... )
QUOTIENT numerator denominator
RANDOM number
REMAINDER numerator denominator
RERANDOM
ROUND number
SIN angle
SQRT value
```

```
SUM number1 number2
   (SUM number1 number2 number3
     ... )
TAN angle
```

Flow of Control

```
CATCH word list
DOUNTIL list condition
ERROR
GO label
IF condition trueList
   IF condition trueList
     falseList
IFFALSE list
   IFF list
IFTRUE list
   IFT list
LABEL label
OUTPUT object
   OP object
REPEAT count list
RUN list
STOP
TEST condition
THROW word
TOPLEVEL
WAIT value
WHILE condition list
```

Logical Operators

```
AND object1 object2
   (AND object1 object2 object3
     ...)
NOT object
OR object1 object2
   (OR object1 object2 object3
     ...)
```

Input and Output

```
BUTTONP
KEYP
MOUSE
PRINT object
   (PRINT object1 object2
object3 ...)
   PR object
   (PR object1 object2 object3
     ...)
READCHAR
   RC
READCHARS number
   RCS number
READLIST
   RL
READWORD
SETRWPROMPT word
SETRWRECT rect
SHOW object
TEXTIO
TOOT frequency duration
TURTLEIO
TYPE object
   (TYPE object1 object2
     object3)
```

Disk Commands

```
ALLOPEN
CATALOG
CLOSE name
CLOSEALL
CREATEFOLDER name
DIR
ERASEFILE file
FILELEN name
FILEP file
LOAD file
ONLINE
OPEN name
POFILE name
PREFIX
READER
READPOS
RENAME oldname newname
SAVE file
SETPREFIX prefix
SETREAD name
SETREADPOS position
SETWRITE name
SETWRITEPOS position
WRITEPOS
```

```
WRITER
```

Turtle Graphics

```
BACK distance
   BK distance
BACKGROUND
   BG
CLEAN
CLEARSCREEN
   CS
CLEARSCREEN3D
   CS3D
DOT [x y]
   DOT [x y z]
DOTP [x y]
FENCE
FILL
FORWARD distance
   FD distance
HEADING
HIDETURTLE
   HT
HOME
LEFT angle
   LT angle
PEN
PENCOLOR
PENDOWN
   PD
PENERASE
   PE
PENREVERSE
   PX
PENUP
   PU
POS
RIGHT angle
   RT angle
ROLLLEFT angle
   RLL angle
ROLLRIGHT angle
   RLR angle
ROTATEIN angle
   RI angle
ROTATEOUT angle
   RO angle
SCRUNCH
SETBG color
SETHEADING angle
   SETH angle
SETHEADING3D [longitude latitude
  roll]
   SETH3D [longitude latitude
     roll]
SETPC color
SETPENSIZE x y
SETPOS [x y]
   SETPOS [x y z]
SETSCRUNCH scrunch
SETX x
SETY y
SETZ z
SHOWNP
SHOWTURTLE
   ST
SHOWTURTLE3D
   ST3D
TOWARDS [x y]
WINDOW
WRAP
XCOR
YCOR
ZCOR
```

Other Drawing Commands

```
ERASEARC rect start angle
ERASEOVAL rect
ERASERECT rect
ERASERRECT rect width height
FRAMEARC rect start angle
FRAMEOVAL rect
FRAMERECT rect
FRAMERRECT rect width height
INVERTARC rect start angle
INVERTOVAL rect
INVERTRECT rect
INVERTRRECT rect width height
PAINTARC rect start angle
PAINTOVAL rect
PAINTRECT rect
```

PAINTRRECT rect width height

Movies
ADDFRAME
CLOSEMOVIE
DELETEFRAME
INSERTFRAME
LASTFRAME
NEXTFRAME
OPENMOVIE name clip position flags
PLAY

Fonts
GETFONTINFO
SETBACKCOLOR color
SETFONTFAMILY family
SETFONTSIZE size
SETFONTSTYLE style
SETFORECOLOR color
TEXTWIDTH word

Speech
DICT word phonetic
ERDICT
PHONETIC word
PITCH value
SAY word
SPEED value
TONE tone
VOICE gender
VOLUME value

Workspace Management
```
BURY word
   BURY list
BURYALL
BURYNAME word
   BURYNAME list
BURYPLIST word
   BURYPLIST list
EDIT word
   EDIT list
EDITS
ERALL
ERASE name
   ERASE list
ERN name
   ERN list
ERNS
ERPROPS
ERPS
NODES
PO name
   PO list
POALL
PON name
   PON list
PONS list
POPS
POT name
   POT list
POTS
PPS
RECYCLE
UNBURY word
   UNBURY list
UNBURYALL
UNBURYNAME word
   UNBURYNAME list
UNBURYPLIST word
   UNBURYPLIST list
```

Callbacks
CALLBACK
CBDISPOSEPARAMH number
CBGETPARAMINT number
CBGETPARAMPOINT number
CBGETPARAMRECT number
CBGETPARAMSTR number
CBGETPARAMSTRH number
CMSETCMD command
CBSETPARAMINT number value
CBSETPARAMPOINT number point
CBSETPARAMRECT number rect
CBSETPARAMSTR number value
CBSETPARAMSTRH number value

Callbacks
CALLNBA :NBAname :param

```
DRAWTOOFFSCREEN
DRAWTOSCREEN
FINDTEXT :text :flags
GETCURRENTSCORE
GETFIELDTEXT :card :itm
HIDEITEM :card :itm :type
MOVENEXT
MOVEPREV
MOVETOCARD :name
MOVETOFIRST
MOVETOLAST
REDRAWCARD
SETBKGDDIRTY
SETFIELDTEXT :card :itm :text
SETITEMRECT :card :itm :type
  :rect
SHOWITEM :card :itm :type
```

special characters

* character 90
+ character 90
- character 29, 90, 92
/ character 90, 91
< character 90
= character 90
> character 90
\ character 22

numbers

3D display 31-41
  colors 129
  projection 127, 131, 138, 139, 140, 141, 143
  stereoscopic 128, 130, 143, 144
3D turtle (see turtle heading)

A

ABS **93**
ADDFRAME 44, 67, **152**
addition 28, 90, 100
ALLOPEN **117**
AND **108**
Apple's System Disk manuals 61
ARCCOS **93**
ARCSIN **93**
ARCTAN **94**
ARCTAN2 **94**
arrays 25
artificial intelligence 23
ASCII **78**
ASCII character codes 78, 79, 83, 86

B

BACK 16, **129**
BACKGROUND **129**
backups 2
BEFOREP **78**
BF **78**
BG **129**
BK **129**
BL **79**
BURY 19, **163**
BURYALL **164**
BURYNAME **164**
BURYPLIST **164**
BUTFIRST **78**
BUTLAST **79**
button 74
BUTTONP **109**

C

CALLBACK **172**
callbacks 57, 58, 172-176, 177-181
CALLNBA **177**
case sensitivity 13, 78, 80, 116, 162
CATALOG **117**
CATCH **101**, 102, 107
CBDISPOSEPARAMH **173**
CBGETPARAMINT **173**
CBGETPARAMPOINT **173**
CBGETPARAMRECT **173**
CBGETPARAMSTR **174**
CBGETPARAMSTRH **174**
CBSETPARAMINT **174**
CBSETPARAMPOINT **175**
CBSETPARAMRECT **175**
CBSETPARAMSTR **175**
CBSETPARAMSTRH **175**
CHAR **79**
CLEAN **130**
CLEARSCREEN 14, **130**
CLEARSCREEN3D 34, 128, **130**
CLOSE **118**
CLOSEALL **118**
CLOSEMOVIE **152**
CMSETCMD **174**
colon 21
colors (see turtle colors)
COPYDEF **72**
copyright 5
COS **94**
COUNT **80**
CREATEFOLDER **119**

CS 14, **130**
CS3D **130**

D

DEFINE **72**, 73
DEFINEDP **73**
DELETEFRAME 45, 67, **153**
delimiters 22, 91
DICT 52, **160**, 161
DIFFERENCE **95**
DIR **119**
disks 116-125
division 28, 90, 91, 97, 98
DOT 128, **131**
DOTP 128, **131**
DOUNTIL **102**
DRAWTOOFFSCREEN **177**
DRAWTOSCREEN **178**

E

EDIT 18, 62, **165**
edit window 18, 19, 62, 65
EDITS 18, 19, 62, **165**
EMPTYP 40, **80**
END 16
EQUALP **80**
ERALL 43, 163, 164, **165**
ERASE 20, **166**
ERASEARC **149**
ERASEFILE **119**
ERASEOVAL **149**
ERASERECT **149**
ERASERRECT **149**
ERDICT **160**
ERN **166**
ERNS 164, **166**
ERPROPS 164, **167**
ERPS 163, **167**
ERROR **102**
errors 101, 102
EXP **95**
expressions 90-93
    functions 91
    operator precedence 90

F

factorial 105
FD 14, **133**
FENCE **132**
file names 116
FILELEN **119**
FILEP **120**
files 117-125
FILL 128, **132**
FINDTEXT **178**
FIRST 24, **81**
FLOAT 30, **95**
floating-point (see numbers)
FLOATP 25, **81**
folders 117, 119, 120, 121
fonts 156-159
FORM **96**
FORWARD 13, **133**
FPUT **81**
FRAMEARC **149**
FRAMEOVAL **150**
FRAMERECT **150**
FRAMERRECT **150**
functions 30, 105

G

garbage collection 170
GETCURRENTSCORE **178**
GetFieldText 57, 58, **178**
GETFONTINFO **156**
GO **103**, 105
GPROP **88**
graphics 126, 152

H

HEADING **133**
HIDEITEM **179**
HIDETURTLE 13, **134**
HOME **134**
HT **134**

HyperLogo run-time 3, 4-6

I

IF 40, **103**
IFF **104**
IFFALSE **104**
IFT **104**
IFTRUE **104**
INSERTFRAME 67, **153**
installation
  hard disk 3
INT 30, **96**
INTEGERP 25, **82**
integers (see numbers)
INTQUOTIENT **97**
INVERTARC **150**
INVERTOVAL **150**
INVERTRECT **151**
INVERTRRECT **151**
ITEM **82**

K

keyboard 110, 111, 112, 114
KEYP **110**, 111

L

LABEL 103, **105**
LAST **82**
LASTFRAME **154**
LEFT 14, **134**
licensing HyperLogo 5
LISP 23
LIST **83**
LISTP 25, **83**
lists 23, 39, 77-87
LN **97**
LOAD **120**
LOCAL 27, **75**
local variables 75
LOWERCASE **83**
LPUT **84**
LT 14, **134**

MAKE 20, 27, **76**
math 27
MEMBER **84**
MEMBERP **84**
MIDI 113
mouse 109, **110**
MOVENEXT **179**
MOVEPREV **179**
MOVETOCARD **179**
MOVETOFIRST **180**
MOVETOLAST **180**
movie window 44, 63, 65, 69
movies 43, 49, 67-69, 152-155
multiplication 28, 90, 97
music 113

N

NAME **76**
NAMEP **76**
NEXTFRAME **154**
NODES **167**
NOT 40, **109**
NUMBERP **85**
numbers 24, 27-30, 77, 85, 89-101
  comparing 90
  floating-point 22, 30, 81, 89, 95, 99
  formatting 96
  integers 22, 30, 82, 89, 96, 97, 99
  negative 29, 90
  random 98, 99
  scientific notation 89

O

ONLINE **120**
OP **105**
OPEN **121**
OPENMOVIE name clip position flags **154**
operator precedence 28, 90
OR **109**
OUTPUT **105**, 110, 113, 114, 115

P

PAINTARC **151**
PAINTOVAL **151**
PAINTRECT **152**
PAINTRRECT **152**
parameters 72, 74
PARSE 55, 58, **85**
PD **135**
PE **136**
PEN **135**
PENCOLOR **135**
PENDOWN 16, 135
PENERASE 135, **136**
PENREVERSE 135, **136**
PENUP 16, 135, **136**
PHONETIC 52, 160, **161**
pie chart 57
PITCH 53, **161**
PLAY **155**
PLIST **88**
PO 17, **168**
POALL 17, 163, 164, **168**
POFILE **121**
PON **168**
PONS 26, 164, **169**
POPS 163, **169**
POS **137**
POT **169**
POTS 18, 163, **170**
POWER **97**
PPROP **88**
PPS 164, **170**
PR **110**
PREFIX **121**
PRIMITIVEP **73**
PRINT 23, **110**
printing 65
procedures 15, 39-40, 72-75, 101, 105, 163-171
    parameters 25
PRODUCT **97**
property lists 87-89, 164-168, 170-172
PU **136**
punctuation 22
PX **136**

Q

quote mark 21
QUOTIENT **98**

R

RANDOM **98**
random numbers 98, 99
RC **111**
RCS **111**
READCHAR **111**, 115
READCHARS **111**, 115
READER **122**
READLIST **111**, 115
READPOS **122**
READWORD 55, 56, **112**, 115
rectangles 146
RECYCLE **170**
REDRAWCARD **180**
REMAINDER **98**
REMPROP **89**
RENAME **122**
REPEAT 15, **105**
RERANDOM **99**
RI 34, **138**
RIGHT 14, **137**
RL **111**
RLL 35, **137**
RLR 35, **138**
RO 34, **139**
ROLLLEFT 35, 128, **137**
ROLLRIGHT 35, 128, **138**
ROTATEIN 34, 128, **138**
ROTATEOUT 34, 128, **139**
ROUND **99**
RT 14, **137**
RUN **106**

S

SAVE **123**, 163, 164
SAY 52, **161**
script 74
script window 10, 11, 19, 62, 65

SCRUNCH **139**
scrunch factor 139, 141
SE **85**
SENTENCE **85**
SETBACKCOLOR **156**
SETBG 15, **139**
SETBKGDDIRTY **180**
SETFIELDTEXT **180**
SETFONTFAMILY **157**
SETFONTSIZE **157**
SETFONTSTYLE **158**
SETFORECOLOR **159**
SETH **140**
SETH3D **140**
SETHEADING **140**
SETHEADING3D 128, **140**
SETITEMRECT **181**
SETPC 15, **140**
SETPENSIZE **141**
SETPLIST **89**
SETPOS 128, **141**
SETPREFIX **123**
SETREAD **124**
SETREADPOS **124**
SETRWPROMPT 55, 56
SETRWPROMPT word **112**
SETRWRECT 56
SETRWRECT rect **112**
SETSCRUNCH **141**
SETWRITE **124**
SETWRITEPOS **125**
SETX **142**
SETY **142**
SETZ 128, **142**
SHOW 22, **113**
SHOWITEM **181**
SHOWNP **143**
SHOWTURTLE 13, 34, **143**
SHOWTURTLE3D 34, 128, **144**
SIN **99**
speech 4, 51-53, 159-163
SPEED 53, **162**
SQRT **100**
ST **143**
ST3D 34, **144**
STOP **106**
strings (see words)
subtraction 28, 90, 92, 95
SUM **100**

T

Talking Tools 4, 160
TAN **100**
TEST 104, **106**
TEXT **73**
text item 57
text window 11, 17, 19, 61, 65
TEXTIO **113**
TEXTWIDTH **159**
THING **76**
THROW **107**
TO 15, 19, 25, 72, 73, **74**, 75
TONE **162**
TOOT **113**
TOPLEVEL **107**
TOWARDS **144**
turtle
    colors 128, 129, 135, 139, 140
    colors, 3D 129
    heading 126, 133, 134, 137-140, 144
    position 126, 131, 134, 137, 141, 142, 145, 146
turtle graphics , 146
turtle window 12, 62, 65
turtle windows 69
TURTLEIO 55, **114**
TYPE **115**

U

unary subtraction 29, 90, 92
UNBURY 19, **171**
UNBURYALL **171**
UNBURYNAME **171**
UNBURYPLIST **172**
UPPERCASE **86**

V

variables 20-27, 28, 75-77, 164-169, 171
    local 75
version number 61
VOICE 52, **162**
VOLUME 53, **163**

W

WAIT **107**
WHILE **108**
WINDOW **144**
WORD **86**
WORDP 25, **87**
words 22, 24, 77-87
    comparing 78, 80, 84, 86, 90
workspace 17, 18, 19, 74, 163-172
WRAP **145**
WRITEPOS **125**
WRITER **125**

X

XCOR **145**

Y

YCOR **145**

Z

ZCOR **146**