# Computer Science Logo Style

Second Edition

VOLUME 2 ADVANCED TECHNIQUES

Brian Harvey

# Computer Science Logo Style
# Advanced Techniques

Brian Harvey

# Computer Science Logo Style

SECOND EDITION

## *Volume 2*
## Advanced Techniques

The MIT Press
Cambridge, Massachusetts
London, England

For information on program diskettes for PC and Macintosh, please contact the Marketing Department, The MIT Press, 55 Hayward Street, Cambridge, Massachusetts, 02142.



This book was typeset in the Baskerville typeface.

The cover art is an untitled mixed media acrylic monotype by San Francisco artist Jon Rife, copyright © 1996 by Jon Rife and reproduced by permission of the artist.

Library of Congress Cataloging-in-Publication Data

Harvey, Brian, 1949–
Computer Science Logo Style / Brian Harvey. — 2nd ed.
p. cm.
Includes indexes.
Contents: v. 1. Symbolic computing. — v. 2. Advanced techniques — v. 3. Beyond programming.
ISBN 0–262–58151–5 (set : pbk. : alk. paper). — ISBN 0–262–58148–5 (v. 1 : pbk. : alk. paper). — ISBN 0–262–58149–3 (v. 2 : pbk. : alk. paper). — ISBN 0–262–58150–7 (v. 3 : pbk. : alk. paper)
1. Electronic digital computers–Programming. 2. LOGO (Computer programming language) I. Title.
QA76.6.H385 1997
005.13′3—dc20 96–35371
CIP

# Contents

## Appendices

# Preface

This is the second volume of *Computer Science Logo Style,* a three-volume series that uses the Logo programming language as the medium for a presentation of a range of topics in computer science. The main audience I had in mind for these books was high school students, but it's turned out that they have also been used in teacher training, and to some extent by independent adult learners.

In the first edition, the first volume was a complete Logo tutorial, explaining all of the features of the language; the second volume was entirely devoted to programming projects. (The third volume, then and now, is a sampler of topics from undergraduate computer science courses.) My idea was that students would spend their first year in an intensive programming course, and would then pursue their own programming projects on an independent study basis, using my projects as examples.

As it turned out, people found the first volume both too hard and too easy. It was too hard because it arrived too soon at the more advanced and complicated features of Logo; it was too easy because the actual programming examples were all short enough to fit on a page. Such tiny examples didn't help the learner extrapolate to the design of a program that could actually do something interesting. This deficiency may have encouraged some readers to conclude that Logo is just a toy, and that serious projects should be done in a "serious" language such as Pascal or C++.

In this second edition I've rearranged things. The first volume now teaches only the core features of Logo, the ones every programmer must understand; it also includes three of the projects that were originally in the second volume. This volume is now a more advanced programming text; it alternates tutorial chapters on advanced language features with example projects that demonstrate those features.

The project chapters serve two purposes at once. First, each project is an example of something you might actually want to do. The emphasis is on getting the computer

to do something fun and interesting. Each of the projects in this book is here because I thought I'd enjoy writing it myself, not because it fit some subtle pedagogic purpose. The projects are offered as case studies, as examples to inspire your own creative efforts.

At the same time, I *am* a teacher, and in this book I'm trying to teach some ideas about programming technique and programming style. Often there is an easy way and a hard way to achieve a certain result, and you're better off if you know the easy way. Nobody has a complete list of such techniques; you'll be learning new ones for as long as you maintain your interest in computer programming. The ones I discuss in this book are the ones that came up in these particular projects. Ideally, as your teacher, I would look over your shoulder while you're working, and I'd tell you about the techniques that apply to *your* projects. I can't do that in a book, and so instead I'm presenting some projects of my own and discussing them as I would discuss your projects if I knew you personally.

With one exception, each example chapter comes after a tutorial chapter that has introduced a new Logo programming technique, and that technique is used in the project. (The exception, the pattern matching project, is an advanced programming technique in its own right, and is used in a later project.) But the technique from the previous chapter is rarely the most important aspect of the project! Each project exhibits many different techniques, and the project chapters describe some of them.

This book does not make much explicit reference to the first volume, but to understand the discussion here, you should be familiar with the ideas presented in Volume 1: evaluation, procedures, locality, iteration, recursion, mapping, predicates, operations, and so on.

Teaching and learning, by the way, don't necessarily imply a classroom in a school. I like to imagine you curled up with this book in front of your home computer, playing around with one of these projects just for the fun of it. Pretend I'm a friend or relative who happens to be a professional computer scientist. On the other hand, if you *are* reading this for a course in a school, you have the advantage of a living teacher who can provide the kind of individual attention to your specific projects that I can't. There are advantages and disadvantages either way.

## About the Projects

Although I now have the projects linked with tutorial chapters, in the first edition I organized them into five categories, based not on the programming techniques used but rather on the purposes of the programs. The projects reflect aspects of my own

character: I came to computers by way of an early interest in mathematics; my computing background is in artificial intelligence and in systems programming; I tend to think in words, not in pictures. I think it may give the collection of projects a more coherent feel if I explain the categories in which they were written, even though the book is no longer organized around those categories.

The first is cryptography. One of the first books I can remember buying, as a child in elementary school, was about secret codes. Besides the universal appeal of knowing a secret, cryptography was interesting to me because it's a *mathematical* sort of puzzle, like those logic problems about who lives in the yellow house. The Cryptographer's Helper project in this volume includes a very small effort at artificial intelligence: the program makes some guesses, on its own, to start solving a cryptogram. The Playfair Cipher project, now moved to the first volume, deals with a more complicated technique for encoding a message, but it doesn't try to break such a code.

The second category is games. I'm not a video game enthusiast; hand-eye coordination isn't my strong point. (I never really learned to ride a bicycle!) Anyway, writing video game programs depends too much on the particular hardware of your computer, so I can't do it in this general book.* Instead I've written two simple strategy games. In the first volume is a program that plays tic-tac-toe. This game is extremely trivial for a human being, but it's surprisingly hard to formulate strategy rules that are simple and precise enough to embody in a computer program. Also, it's an opportunity to throw in a little bit of graphics programming, to draw the board and fill it with Xs and Os. In this volume is a program that deals out a hand of solitaire and maintains the display of the layout as you play the hand. Before I wrote this program, I had been feeling bored and lonely for an extended period, and I was wasting a lot of time playing solitaire myself. I figured it would be more productive to write a computer program!

The third category is mathematics. I once spent some time working as a systems programmer at a computer music research center in Paris, and this volume includes a project about Fourier analysis, the mathematical basis of computer music. The project demonstrates graphically how a complex waveform, representing the texture of a sound, can be built up from much simpler elements. In the first volume is a program to solve the kind of problem, often found on IQ tests, in which you are given pitchers of certain sizes and asked to use them to measure a given amount of water by pouring back and forth. This project illustrates the idea of searching through a "solution space" of possible pouring steps.

---

* You can find a video game that I wrote in the collection *LogoWorks: Challenging Programs in Logo*, edited by Solomon, Minsky, and Harvey (McGraw-Hill, 1985).

The fourth category is that of utility programs. This is actually the area of programming I know best: writing things that are not complete applications in themselves, but rather tools to help in the creation of even larger projects. For the second edition I've replaced the original projects in this category with two new ones. The project Finding File Differences is a utility program that can be used to compare two versions of a file to see what's changed from the old one to the new one. Then there is a compiler for the BASIC programming language; besides illustrating the idea of program as data—the compiler generates new Logo procedures to carry out the instructions in a BASIC program—this project may help to prepare the reader for the more complicated Pascal compiler in the third volume.

The fifth category is pattern matching. This category combines my interests in systems programming and artificial intelligence. The first project is a tool, like the ones in the utilities category, but it's a tool designed specifically for artificial intelligence applications: a pattern matcher. This program compares a particular list with a general template, or pattern. Instead of checking for exact equality like `equalp`, the pattern matcher checks for a kind of "fill in the blanks" partial equality. The second project in this category uses the pattern matcher to implement `doctor`, another famous artificial intelligence program that simulates a conversation with the user.

## About This Series

*Computer Science Logo Style* is intended to bring to the hobbyist audience a particular point of view about computer science: the artificial intelligence view. This way of looking at computers is quite different from the more usual software engineering approach. In that approach, you are always dealing with a very well-defined problem, and are looking for the best way to solve it. Software engineers like to start with a formal problem statement, and then design a computer program to fit. They believe that the design process should be *top-down*; you should start with the overall structure and work down to the details. Their preferred programming language is Pascal.

In artificial intelligence, the problems are not usually so well defined. Starting with a vague problem statement like "develop a good strategy for playing chess," AI programmers can't begin with a rigid program specification. Instead, they build *tools:* program fragments that can be pieced together to form larger programs. The programming process involves writing code, testing, coming up with new ideas, and modifying the program interactively. This process is encouraged by an interactive language like Lisp or Logo.

Computer programming is a great intellectual hobby; it provides the same opportunity for creative, concrete work in mathematical thinking that drama or creative writing does for verbal thinking. A learner can have years of intellectual adventure just learning to write better and better programs. Finally, though, there may come a time when the learner gets bored with just writing more and more programs, and seeks a deeper understanding of the issues behind this practical work. The third volume of this series, *Beyond Programming,* addresses the needs of these learners by introducing them to some of the elements of university-level computer science, still in the context of Logo programming.

## How to Read This Book

You should have each program actually available to you on a computer as you read about it. These programs are available on diskette from the MIT Press, or can be downloaded from the Internet. Details are in the first volume.

There are many dialects of Logo; this book uses Berkeley Logo, a free version available for PC, Macintosh, and Unix systems. The more fundamental Logo techniques used in the first volume are more or less standard among Logo implementations, but some of the advanced techniques in this volume are unique to Berkeley Logo. It, too, is on the diskette and the Internet.

The programs you see here are essentially the programs I wrote as I was trying to get each project to work. I didn't start with a particular programming style in mind and then invent an example to illustrate the style. It's not always obvious what is the "correct" style for a given problem; sometimes one way is much easier to understand, for example, while a different solution may run much more efficiently. The comments in each chapter sometimes suggest alternative ways in which I might have written some piece of the program. I try to explain why I chose the style I did, although sometimes the real explanation is simply that that's the first thing I thought of. I've modified almost all of these programs for the second edition, and some of the chapters explain my second thoughts.

Each example chapter begins with an explanation of what the project is all about. Remember that these projects were meant to be interesting in themselves, not just as vehicles for a discussion of programming techniques! The discussion in each chapter ends with a return to the purpose of the project, with suggestions for how that purpose might be extended. One source of ideas for projects of your own is to extend someone else's work, and one important purpose of this book is to give you ideas for such starting points. In between comes a technical discussion of the programming techniques used.

What I do *not* provide, generally, is a guided tour of every procedure. One of the things you should learn from this book is the ability to read a long program on your own. You should recognize some of the typical categories of procedures, like ones that apply a given command to each member of a list. In the discussions, rather than explain every detail, I try to focus your attention on the parts of the program that seem to illuminate some more general technical issue. A complete listing of the program is at the end of each example chapter.

The programs in this book are copyright, but you can use, copy, and redistribute them freely; the exact terms are given in the GNU General Public License, which is distributed with the programs and is printed in the first volume of this series. Essentially, the only restriction is that you can't use these programs as the basis for your own commercial programs; if you extend these projects, you can only distribute your extensions on the same free terms. Share ideas, don't hoard them!

# Acknowledgments

Cynthia Solomon and Margaret Minsky are the people who got me started at the enterprise of developing exemplary Logo projects. People in the Logo community had been talking for many years about the need for an advanced Logo project book, but nobody got around to it until 1982 when Atari had all the money in the world and used some of it to establish a Corporate Research Department. Cynthia was in charge of the Atari research lab in Cambridge, where many MIT old-timers were gathered. She and Margaret decided that this was the time for the project book. I was one of several people they recruited to contribute projects. The result of that effort is called *LogoWorks: Challenging Programs in Logo* (McGraw-Hill, 1985).

This book is somewhat different from *LogoWorks* in that it's part of a series, so I can make assumptions here about what the reader already knows from having read the first volume. Still, I've benefited greatly from what I learned from Cynthia and Margaret about how to explain the structure of a large programming project.

The people who have read and commented on early drafts of this book include Hal Abelson, Alison Birch, Sharon Yoder, Mike Clancy, Jim Davis, Batya Friedman, Paul Goldenberg, Margaret Minsky, and Cynthia Solomon. As for the first volume, I am particularly indebted to Hal and Paul for their strong encouragement and their deep insights into issues both in computer science and in education. Matthew Wright reviewed some chapters for the second edition.

Berkeley Logo, the interpreter used in this edition, is a collective effort of many people, both at Berkeley and across the Internet. My main debt in that project is to three former students: Dan van Blerkom, Michael Katz, and Doug Orleans. At the risk of missing someone, I also want to acknowledge substantial contributions by Freeman Deutsch, Khang Dao, Fred Gilham, Yehuda Katz, George Mills, and Randy Sargent.

# Computer Science Logo Style
# Advanced Techniques

# 1 Data Files

Program file for this chapter: `format`

The programming techniques that you learned in the first volume of this series are all you need to express any computation. That is, given any question that a computer program can answer, you can write the program in Logo using those techniques. Also, those techniques can be used, with few if any changes in notation, in any implementation of Logo. However, saying that a problem can be solved using certain tools doesn't mean that it can be solved in the most convenient way. In this volume the overall goal is to expand your repertoire of Logo techniques, so that you'll find it easier to deal with more difficult problems. Some of the techniques here are unique to Berkeley Logo; others exist in other dialects, but in significantly different forms.

Probably the most glaring omission in the first volume is that we made no provision for saving information from one session to the next. (You do know how to save a Logo workspace, but that's too all-or-nothing to be very useful. You'd like to be able to save specific kinds of information, and perhaps use that information in some program outside of Logo.) In this chapter we'll explore the use of *data files* in Logo programs.

There isn't much in the way of truly new ideas here. There are a few new primitives and a few grubby details about how files are named in your particular computer, but for the most part you won't have to change the way you think about the programming process. My plan for this chapter is to give a quick summary of the vocabulary you'll need, and spend most of the chapter on a practical programming project that will show you the sort of thing you can accomplish easily in Logo.

## Reader and Writer

We've been reading and writing data all along. We've been reading from the keyboard, with operations like `readlist` and `readchar`, and we've been writing to your screen, with commands like `print` and `type`.

The goal now is to read and write the same data, but from and to other devices. This includes files on a hard disk or a diskette, but also things like printers or TV cameras if you have them. The same procedures that read the keyboard and write the screen can be used for these other devices as well. The trick is to divert the attention of those procedures to someplace else.

The part of the Logo interpreter that reads characters for `readlist` and `readchar` is called the *reader;* the part that handles `print` and its friends is the *writer.* The commands `setread` and `setwrite` tell the reader and the writer, respectively, what file or device to use. The input to either command is the name of a file or device. The format of that name will vary from one operating system to another, so you should look it up in your computer's reference manual. Generally it will be the same format that you (I assume) have already been using as input to the `save` and `load` commands.

If you invoke `setread` with the empty list as input, it tells the reader to read from the keyboard. If you give `setwrite` the empty list as input, it tells the writer to write to the screen. In other words the empty list "turns off" whatever file or device you may have been using and returns to Logo's usual style of interaction.

You can switch the attention of the reader or the writer among several files in rotation without "losing your place" in each one. You must *open* a file when you want to begin reading or writing it before you can use it as input to `setread` or `setwrite`. You do this with the `openread` or `openwrite` command.* Once a file is opened, you can `setread` or `setwrite` to it, read or write some data, then switch to a different file for a while, and then continue where you left off. When you're finished using the file, you must `close` it.

Some operating systems allow access to devices like printers using the same programming interface that works for files. In those systems, you can `setwrite` to a printer just as you can to a disk file. The format of the input to `setwrite` may be different (a device name instead of a file name), but there is no conceptual difference.

* `Openwrite` creates a new, empty file, replacing any file that might previously have existed with the same name. Berkeley Logo also provides `openupdate`, which opens an existing file for both reading and writing simultaneously, and `openappend`, which opens an existing file for writing, putting the newly written data after the old contents of the file. I won't use those in this book, though.

## End of File

When reading information from a file, the problem arises of what happens when there is no more left to read. How does a program know it's reached the end of the file?

Berkeley Logo provides two ways to answer this question. If the structure of your program makes it convenient to test for the end of the file *before* attempting to read more information from the file, you can use the predicate `eofp`, which takes no inputs, and returns `true` if the file currently being read is at its end. (If Logo is reading from the keyboard, then `eofp` always returns `false`.)

In some cases it may be more convenient to try to read from the file, and then later test whether there was really any information available to read. To make this possible, the reading operations output an empty datum when there is nothing left to read, but of the opposite type from their usual output. In other words `readlist`, which usually outputs a list, outputs an empty *word* to indicate the end of a file. `Readchar`, which normally outputs a word, outputs an empty *list* when there are no more characters to be read. You can use `wordp` or `listp`, therefore, to check for the end of the file.

Here's an example. `Extract` is a command that takes two inputs, a word and a filename. Its effect is to print every line in that file that contains the chosen word. For example, you might have a file in which each line contains someone's name and telephone number; you could use this procedure to find a particular person in the file.

```
to extract :word :file
openread :file
setread :file
extract1 :word
setread []
close :file
end

to extract1 :word
local "line
if eofp [stop]
make "line readlist
if memberp :word :line [print :line]
extract1 :word
end

? extract "brian "phonelist
Brian Harvey 555-2368
Brian Silverman 555-5274
```

Notice that the program restores reading from the keyboard when it's done reading the file. In the example I'm assuming that `phonelist` is the name of a file you've created earlier, with a Logo program or with your favorite text editor outside of Logo.

## Case Sensitivity

In this example, I used the word `brian`, in all lower case letters, as the input to `extract`, whereas the data file contained the word `Brian` with an initial upper case or capital letter. You can control whether or not Logo considers those two words equal by changing the value of the variable `caseignoredp`. If this variable has the value `true`, as it does by default, then `equalp` and `memberp` consider upper and lower case letters the same. But if you say

```
make "caseignoredp "false
```

then upper and lower case letters will not be equal. (This variable does *not* affect Logo's understanding of the names of procedures and variables, in which case is always ignored. The words `print` and `PRINT` always name the same procedure, for example.)

## Dribble Files

Not everything Logo prints goes through the writer. Error messages and trace output always go to the screen, not into a file. The idea is that even when you're using files, you're still programming interactively, and those messages are part of the programming process rather than part of the result of your program.

Sometimes, though, you want to capture in a file *everything* that happens while you're using Logo. Some programming teachers, for instance, like to look over their students' shoulders but can't look at everyone at once. If you record everything you do, your teacher can print out the record, take it home, and study it overnight. The formal name for this kind of record is a *transcript file,* but it's more popularly known as a *dribble file.* (The metaphor is that there's a leak in the pipe between the computer and the screen and some of the data dribbles out into the file.)

The `dribble` command takes a file name as input and starts dribbling into that file. The `nodribble` command, with no input, turns off dribbling. Information is sent to the dribble file *in addition to* being printed on your screen, or written in a file by the writer. Compare this with the effect of `setwrite`, which tells Logo to print into a file *instead of* onto the screen.

If you want to keep a transcript of a programming session, remember that much of your interaction with Logo happens in the Logo editor and that that kind of interaction can't be recorded in a dribble file. So you might want to make it a habit to `po` the procedures you've edited, each time you leave the editor.

## A Text Formatter

Okay, it's time for the practical project I promised you. Probably the most useful "real" program you can find for a home computer is a word processor. There are two parts to a word processing package: a text editor and a formatter. The editor is the part of the system that lets you type in your document, correct errors, and make additions and deletions later. The formatter is the part that takes what you type and turns it into beautiful printed pages with even margins and so on. (In most word processors, these two parts are integrated, so that every character you type makes an immediate change in the beautifully formatted document. But in principle the two tasks are separable.)

I'm going to write a text formatter. I assume that you have some way to put text into a file. (In some versions of Logo the same editor that you use for procedures can also edit text files. Otherwise you probably have a separate program that edits files, or else you can write one in Logo!) The formatter will read lines from a file, fill and justify paragraphs, and print the result. (To *fill* text means to fit as many words as possible into each printed line. To *justify* the text is to insert extra spaces between words so that both margins line up.) You can see how the formatter will work by examining the example on the following pages. I've shown both what's in the file and what my program prints.

For the most part the formatter just copies words from one file to another, filling and justifying as it goes. A blank line in the file indicates a break between paragraphs. The program skips a line between paragraphs and indents the first line of the new paragraph. It's possible to control the formatter's work by including *formatting commands* in the file. These are the lines that start with asterisks in the example. For example, the line that says

```
* nofill
```

means, "From now on, stop filling paragraphs. Instead, each line in the input file should be one line in the printed result." The `yesfill` command returns to normal paragraph style.*

* I'd have liked to call the command `fill`, as it would be in a commercial word processing program, but unfortunately that's the name of a primitive procedure in Logo.

When I wrote the first edition of this book in 1984, I said that the study of computer programming was intellectually rewarding for young children in elementary school, and for computer science majors in college, but that high school students and adults studying on their own generally had an intellectually barren diet, full of technical details of some particular computer brand.

At about the same time I wrote those words, the College Board was introducing an Advanced Placement exam in computer science. Since then, the AP course has become popular, and similar official or semi-official computer science curricula have been adopted in other countries as well. Meanwhile, the computers available to ordinary people have become large enough and powerful enough to run serious programming languages, breaking the monopoly of BASIC.
* nofill
I think that there shall never exist
a poem as lovely as a tree-structured list.
* yesfill
So, the good news is that intellectually serious computer science is within the reach of just about everyone. The bad news is that the curricula tend to be imitations of what is taught to beginning undergraduate computer science majors, and I think that's too rigid a starting point for independent learners, and especially for teenagers.

See, the wonderful thing about computer programming is that it's fun, perhaps not for everyone, but for very many people. There aren't many mathematical activities that appeal so spontaneously. Kids get caught up in the excitement of programming, in the same way that other kids (or maybe the same ones) get caught up in acting, in sports, in journalism (provided the paper isn't run by teachers), or in ham radio. If schools get too serious about computer science, that spontaneous excitement can be lost. I once heard a high school teacher say proudly that kids used to hang out in his computer lab at all hours, but since they introduced the computer science curriculum, the kids don't want to program so much because they've learned that programming is just a means to the end of understanding the curriculum. No! The ideas of computer science are a means to the end of getting computers to do what you want.
*skip 4
*make "nofilltab 15
*nofill
Computer
Science
Apprenticeship
*yesfill
*make "spacing 2
My goal in this series of books is to make the goals and methods of a serious computer scientist accessible, at an introductory level, to people who are interested in computer programming but are not computer science majors. If you're an adult or teenaged hobbyist, or a teacher who wants to use the computer as an educational tool, you're definitely part of this audience. I've taught these ideas to teachers and to high school students. What I enjoy most is teaching high school freshmen who bring a love of programming into the class with them--the ones who are always tugging at my arm to tell me what they found in the latest Byte.

formatter input file

formatted output

When I wrote the first edition of this book in 1984, I said that the study of computer programming was intellectually rewarding for young children in elementary school, and for computer science majors in college, but that high school students and adults studying on their own generally had an intellectually barren diet, full of technical details of some particular computer brand.

At about the same time I wrote those words, the College Board was introducing an Advanced Placement exam in computer science. Since then, the AP course has become popular, and similar official or semi-official computer science curricula have been adopted in other countries as well. Meanwhile, the computers available to ordinary people have become large enough and powerful enough to run serious programming languages, breaking the monopoly of BASIC.

I think that there shall never exist
a poem as lovely as a tree-structured list.

So, the good news is that intellectually serious computer science is within the reach of just about everyone. The bad news is that the curricula tend to be imitations of what is taught to beginning undergraduate computer science majors, and I think that's too rigid a starting point for independent learners, and especially for teenagers.

See, the wonderful thing about computer programming is that it's fun, perhaps not for everyone, but for very many people. There aren't many mathematical activities that appeal so spontaneously. Kids get caught up in the excitement of programming, in the same way that other kids (or maybe the same ones) get caught up in acting, in sports, in journalism (provided the paper isn't run by teachers), or in ham radio. If schools get too serious about computer science, that spontaneous excitement can be lost. I once heard a high school teacher say proudly that kids used to hang out in his computer lab at all hours, but since they introduced the computer science curriculum, the kids don't want to program so much because they've learned that programming is just a means to the end of understanding the curriculum. No! The ideas of computer science are a means to the end of getting computers to do what you want.

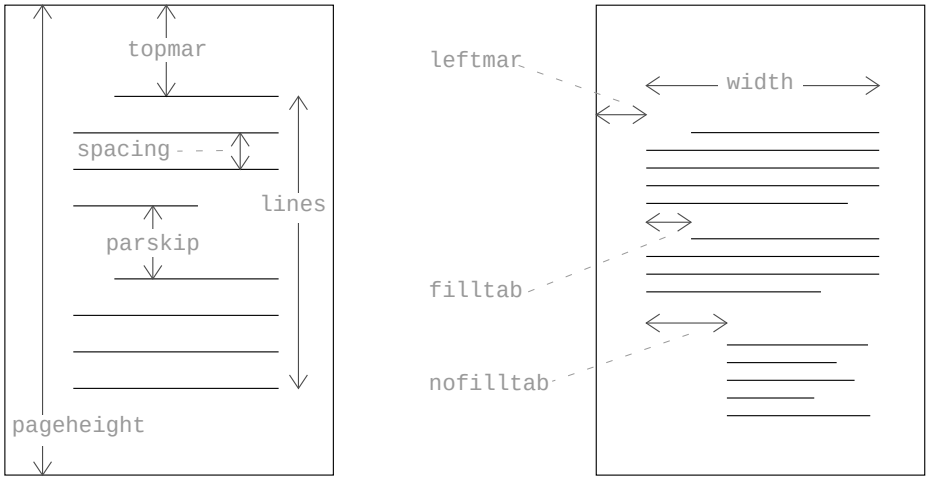
## Computer Science Apprenticeship

My goal in this series of books is to make the goals and methods of a serious computer scientist accessible, at an introductory level, to people who are interested in computer programming but are not computer science majors. If you're an adult or teenaged hobbyist, or a teacher who wants to use the computer as an educational tool, you're definitely part of this audience. I've taught these ideas to teachers and to high school students. What I enjoy most is teaching high school freshmen who bring a love of programming into the class with them--the ones who are always tugging at my arm to tell me what they found in the latest Byte.

To run the program, invoke the `format` command. This command takes two inputs: the name of a file to read and the name of a file to write. The latter might be the name of the printer if your operating system allows it.

## Page Geometry

The program uses several global variables to determine the layout of a printed page. Vertical measurements are in vertical lines (6 per inch for most computer printers); horizontal measurements are in characters (10 per inch is common, although there is more variation in this unit). The program assumes fixed-width characters; a more professional program would handle variable-width character fonts, but the added complexity wouldn't help you learn the things I'm most interested in now.




| | |
|---|---|
| `pageheight` | Height of the entire sheet of paper, including margins. |
| `topmar` | Number of lines of margin at the top of each page. |
| `lines` | Number of lines to be printed on each page. |
| `parskip` | Number of blank lines between paragraphs. |
| `spacing` | 1 for single spaced printing, 2 for double spaced, etc. |
| `leftmar` | Number of characters of margin at the left of the page. |
| `width` | Number of characters to print on each line. |
| `filltab` | Number of characters to indent the first line of a paragraph. |
| `nofilltab` | Number of characters to indent each `nofill` line. |

The formatter recognizes formatting commands, in the file it's reading, to change the values of these variables. By a strange coincidence these formatting commands look similar to the Logo commands to set a variable. In the sample file, for instance, the formatting command

```
*make "spacing 2
```

is used to start double spacing.

## The Program

Here are the procedures that make up the formatter.

```
to format :from :to
openread :from
openwrite :to
setread :from
setwrite :to
init.vars
loop
setread []
setwrite []
close :from
close :to
end

to init.vars
make "pageheight 66
make "topmar 6
make "lines 54
make "leftmar 7
make "width 65
make "filltab 5
make "nofilltab 0
make "parskip 1
make "spacing 1
make "started "false
make "filling "true
make "printed 0
make "inline []
end
```

```
to loop
forever [if process nextword [stop]]
end

;; Add a word to the output file, starting a new line if necessary

to process :word
if listp :word [output "true]
if not :started [start]
if (:linecount+1+count :word) > :width [putline]
addword :word
output "false
end

to addword :word
if not emptyp :line [make "linecount :linecount+1]
make "line lput :word :line
make "linecount :linecount+count :word
end

to putline
repeat :leftmar+:indent [type "| |]
putwords :line ((count :line)-1) (:width-:linecount)
newline
skip :spacing
end

to putwords :line :spaces :filler
local "perword
if emptyp :line [stop]
type first :line
make "perword ifelse :spaces > 0 [int ((:filler+:spaces-1)/:spaces)] [0]
if :filler > 0 [repeat :perword [type "| |]]
type "| |
putwords (butfirst :line) (:spaces-1) (:filler-:perword)
end
```

```
;; Get the next input word, reading a new line if necessary

to nextword
if not emptyp :inline [output extract.word]
if not :filling [break]
make "inline readword
if listp :inline [break output []]
if emptyp :inline [break output nextword]
if equalp first :inline "|*| ~
   [run butfirst :inline
    make "inline "]
make "inline skipspaces :inline
output nextword
end

to extract.word
local "result
make "result firstword :inline
make "inline skipfirst :inline
output :result
end

to firstword :word
if emptyp :word [output "]
if equalp first :word "| | [output "]
output word (first :word) (firstword butfirst :word)
end

to skipfirst :word
if emptyp :word [output "]
if equalp first :word "| | [output skipspaces :word]
output skipfirst butfirst :word
end

to skipspaces :word
if emptyp :word [output "]
if equalp first :word "| | [output skipspaces butfirst :word]
output :word
end
```

```
;; Formatting helpers

to start
make "started "true
repeat :topmar [print []]
newindent
end

to newindent
newline
make "indent ifelse :filling [:filltab] [:nofilltab]
make "linecount :indent
end

to newline
make "line []
make "indent 0
make "linecount 0
end

to break
if emptyp :line [stop]
make "linecount :width
putline
newindent
if :filling [skip :parskip]
end

;; Formatting commands to be invoked by the user

to skip :howmany
break
repeat :howmany [print []]
make "printed :printed+:howmany
if :printed < :lines [stop]
repeat :pageheight-:printed [print []]
make "printed 0
end

to nofill
break
make "filling "false
newindent
end
```

```
to yesfill
break
if not :filling [skip :parskip]
make "filling "true
newindent
end
```

To help you understand this program, you should start by imagining that the text file contains one big paragraph with no formatting commands. For each word in the file, `loop` invokes `nextword` to read the word and `process` to process it. Just take `nextword` on faith for now and look at `process`. The third and fourth instruction lines are the interesting ones. The third line asks whether adding this word to the partially filled print line will overflow its width. If so, `process` invokes `putline` to print that line and start a new one. Then, in either case, `process` invokes `addword` to add the word to the print line it's accumulating. `Addword` puts the word at the end of the line and also adds its length to `:linecount`, the number of characters in the line. If this isn't the first word of a new line, then it must also add another character to `:linecount` to take account of the space between words.

`Putline` is essentially just a fancy `print` command. The complication comes in because the program is trying to justify the line by adding spaces where needed between words. To do this, it has to `type` the line a word at a time; that's the task of `putwords`. In that procedure, `:spaces` is the number of spaces between words not yet printed; in other words it's the number of positions into which extra spaces can be shoved. (The idea is to spread out the necessary spaces as evenly as possible.) `:Filler` is the total number of extra spaces we need to insert; `:perword` is the number that should be inserted after the particular word we're typing right now. (When I started writing `putline` and `putwords`, I thought that I could just calculate `:perword` once for each line. But if the number of extra spaces we want to insert is not a multiple of the number of positions available, then the number of extra spaces may not be equal for every word in the line.)

That's pretty much the whole story about the printing part of the program. The reading part is handled by `nextword`. It reads a line at a time into the variable `inline`. `Nextword` uses the Logo primitive `readword` to read a line, rather than the usual `readlist`, to avoid Logo's usual special handling of parentheses and brackets. `Readword` outputs a word containing all of the characters on the line that it reads, even if the line includes spaces, which would ordinarily separate words. Therefore, the program must divide the long word output by `readword` into ordinary words; that's the job of `extract.word` and its subprocedures `firstword`, `skipword`, and `skipspaces`.

Each time `nextword` is invoked, it removes one word from the line and outputs that word. When `:inline` is empty, `nextword` reads a new line from the file. There

are four possibilities: First, the end of the file may be reached. `Listp` tests for this; if so, `nextword` outputs an empty list. Second, the new line can be empty, indicating a paragraph break. In this case `nextword` invokes `break` and reads another line. Third, the new line can be a formatting command, starting with an asterisk. In this case `nextword` just `run`s the line, minus the asterisk, and reads another line. Fourth, the line can be an ordinary text line, in which case `nextword` goes back to extracting words from the line.

In most programming languages, most of the effort in writing a formatter like this would be in recognizing and evaluating the formatting commands. I hope you appreciate how much Logo's ability to `run` instructions found in a file simplifies this task! The danger in this technique is that an invalid instruction in the input file will crash the formatting program, giving a Logo error message. (This is especially bad because after the error message we are left with a half-written output file still open.) I'd like to "catch" errors while running the user's instructions; you'll see how to do that in Chapter 3.

The rest of the program is just a bunch of detail. The `skip` command is written to be used both by the formatting program itself and as a formatting command, as in the example I showed earlier. As an exercise in understanding program structure, notice that `skip` invokes `break` and `break` invokes `skip`; then explain why they don't just keep invoking each other forever, like a recursive procedure without a stop rule.

Another slightly tricky part to understand is the variable `started` and the procedure `start`. `Start` is invoked by `process`, but only once, before processing the very first word of the text. Ensuring the "only once" is the sole purpose of `started`, a variable that initially contains `false` and is changed to `true` by `start`. Instead, why don't I just invoke `start` from `format` before calling `loop`? The answer is that this technique allows the file to start with an instruction like

```
*make "topmar 10
```

Any such instructions will be evaluated *before* processing the first text word. If `start` were invoked by `format`, the top margin would be skipped before this instruction had a chance to set `:topmar`.

## Improving the Formatter

Actually, using `make` as a formatting command is a little schlock—not what I'd call good "human engineering." If you wanted to make a million dollars selling this program, you'd add several little procedures like this:

```
to topmar :lines
make "topmar :lines
end
```

Like `nofill` and `yesfill`, these procedures would be used only as formatting commands, not as part of the formatter itself.

The program leaves out a lot of things you'd like to be able to do. You should be able to number pages automatically in the top or bottom margins. (That's a pretty easy modification; most of the work would be in `skip`.) You'd like to be able to center lines on the page for chapter headings. If your printer can underline or use different type faces, you'll want a way to control those things with formatting commands.*

Still, this is a usable program carrying out a real task. It takes 19 Logo procedures averaging 7 lines each. This would be a much harder project in most languages. What makes it so manageable in Logo? First, *modularity*. A small procedure for each task makes the overall program easier to understand than it would be if it were all in one piece. Second, Logo's data types, words and lists, are well suited to this problem. Third, Logo's control mechanisms, especially recursive operations and `run`, have the needed flexibility.

* If you're *really* ambitious, you could try teaching the program about footnotes!

# 2 Example: Finding File Differences

Program file for this chapter: `diff`

As an example of a practical program that manipulates data files, this chapter is about comparing two similar files to find the differences between them. This program is most often useful in comparing a current version of a file with an earlier version to see what has changed. On the next page is an example of two input files and the program's output. The output shows only those lines that differ between the two files, indicating changed lines, insertions, and deletions. When several consecutive lines are different, they are grouped together as a single reported change. (To demonstrate all of the program's capabilities, I've used short input files with more lines different than identical, and so the program's report is longer than the input files themselves. In a more realistic example, the input files might be hundreds of lines long, with only a few differences, so it would be easier to read the program's output than to examine the files directly.)

I've called this program `diff` because it was inspired by a similar program of that name in the Unix operating system. The format of the report that my `diff` generates is similar to that of the Unix version. In particular, I've followed the Unix `diff` convention that when a line from one of the input files appears in the report, it is marked by either a "<" character if it's from the first file or a ">" character if it's from the second file.

The numbers in the lines that begin with `CHANGE`, `INSERT`, or `DELETE` are line numbers, counting from one, in each of the two files. For example, the line

```
CHANGE 6-8 6-7
```

indicates that lines 6 through 8 in the first file were replaced by lines 6 through 7 in the second file. (The program considers a change to be finished when it finds two consecutive identical lines in the two files. In this case, lines 9 and 10 of the first file are identical to lines 8 and 9 of the second file.)

**Input Files**

Text1

```
My goal in this series of books
is to make the goals and methods
of a serious computer scientist
accessible, at an introductory
level, to people who are
interested in computer
programming but are not computer
science majors.
If you're an
adult or teenaged hobbyist,
or a teacher who wants to use the
computer as an educational tool,
you're definitely part of this
audience.
```

Text2

```
My goal in this series of books
is to make the goals and methods
of a mad computer scientist
accessible, at an introductory
level, to people who are
interested in playing computer
games.
If you're an
adult or teenaged hobbyist,
you're definitely part of this
audience.
And I hope you appreciate
the privilege!
```

**Output File**

```
DIFF results:
< File 1 = Text1
> File 2 = Text2
==========
CHANGE 3-3 3-3
< of a serious computer scientist
-----
> of a mad computer scientist
==========
CHANGE 6-8 6-7
< interested in computer
< programming but are not computer
< science majors.
-----
> interested in playing computer
> games.
==========
DELETE 11-12 10
< or a teacher who wants to use the
< computer as an educational tool,
==========
INSERT 15 12-13
> and I hope you appreciate
> the privilege!
==========
```

The `diff` procedure takes three inputs. The first two are names of the two input files; the third is either a name for an output file or an empty list, in which case the program's results are printed on the screen. For example, to see the results of my sample run, I'd say

```
diff "Text1 "Text2 []
```

I picked this project partly because it requires switching between two input files, so you can see how the program uses `setread` repeatedly.

## Program Overview

`Diff` reads lines from the two input files in alternation. As long as the corresponding lines are equal, the program just moves on to the next pair of lines. (Procedure `diff.same` handles this process.) When a difference is found, the program's operation becomes more complicated. It must remember all the lines that it reads from both files until it finds two consecutive equal pairs of lines. (Procedures `diff.differ` and `diff.found` do this.)

Life would be simple if the differences between the two files were only changes within a line, without adding or removing entire lines. (This would be a realistic assumption if, for example, the second file had been created by applying a spelling correction program to the first file. Individual words would then be different, but each line of the second file would correspond to one line of the first.) In that case, the structure of `diff.differ` could be similar to that of `diff.same`: Read a line from each file, compare the two, and report the pairs that are different.

But in practice, a change to a paragraph may make the file longer or shorter. It may turn out, as in my sample run, that three lines from the first file correspond to only two lines from the second one. If that's the case, then there's no guarantee that the equal lines that mark the end of a change will be at the same line *numbers* in the two files. (In the sample, line 9 of the first file matches line 8 of the second.) Whenever the program reads a line from one file, therefore, it must compare that line to *every* line that it's read from the other file since the two started being different. Therefore, `diff.differ` must *remember* all of the lines that it reads from both files.

Finally, when two pairs of equal lines are found, the program must report the difference that it's detected. That's the job of procedure `report`. Once the change has been reported, the program continues in `diff.same` until another difference is found.

The program finishes its work when the ends of both input files have been reached.

## The File Information Block Abstract Data Type

For each of the two input files, the program must remember several kinds of information. The `report` procedure must know which is file number 1 and which file number 2, in order to print the lines with the correct starting character. The name of each file is needed as the input to `setread`. The current line number is needed in order to report the location within each file of a changed section. As I've just explained, there is a collection of *pending* lines during the examination of a change; we'll see shortly that another collection of *saved* lines is used for another purpose.

To keep all the information for a file collected together, `diff` uses an abstract data type called a "file information block," or FIB, that is implemented as an array with five members. The array is made by a constructor procedure `makefile`, and there are selectors for four of the five components: `which`, `filename`, `linenum`, and `lines`. For the fifth component, the saved lines, instead of a selector for the entire collection the program uses a selector `popsaved` that outputs a single line each time it's invoked. (This will make more sense when you read about saved lines in the next section.)

The procedures within this program use these two FIBs as inputs instead of just the filenames. To read from one of the files, for example, the program will say

```
setread filename :fib1
```

## Saving and Re-Reading Input Lines

One further detail complicates the program. Suppose that a change is found in which the two groups of differing lines are of different lengths. For example, suppose three lines in the first file turn into six lines in the second file, like this:

```
Original line 1                 Original line 1
Original line 2                 Changed line 2
Original line 3                 Changed line 3
Original line 4                 New line 3.1
Original line 5                 New line 3.2
Original line 6                 New line 3.3
Original line 7                 Changed line 4
Original line 8                 Original line 5
Original line 9                 Original line 6
```

The program has been reading lines alternately from the two files. It has just read the line saying "Original line 6" from the second file, and that's the second consecutive match with a line previously read from the first file. So the program is ready to report a change from lines 2–4 of the first file to lines 2–7 of the second.

The trouble is that the program has already read three lines of the first file (the last three lines shown above) that have to be compared to lines that haven't yet been read from the second file. Suppose that the files continue as follows:

```
Original line 10                    Original line 7
```

We can't just say, "Okay, we've found the end of a difference, so now we can go back to `diff.same` and read lines from the two files." If we did that, we'd read "Original line 10" from file 1, but "Original line 7" from file 2, and we'd think there is a difference when really the two files are in agreement.

To solve this problem we must arrange for `diff.same` to *re-read* the three unused lines from file 1. Logo allows a programmer to re-read part of a file by changing the current *position* within the file (this ability is called *random access*), but in this program I found it easier to *buffer* the lines by saving them in a list and then, the next time the program wants to read a line from the file, using one of the saved lines instead.

```
to readline :fib
if savedp :fib [output popsaved :fib]
setread filename :fib
output readword
end
```

The first instruction of this procedure says, "If there are any saved lines for this file, remove the first one from the list and output it." Otherwise, if there are no saved lines, then the procedure directs the Logo reader to the desired file (using `setread`) and uses `readword` to read a line. Because `popsaved` removes a line from the list of saved lines, eventually the saved lines will be used up and then the program will continue reading from the actual file.

## Skipping Equal Lines

Here is the procedure that skips over identical pairs of lines:

```
to diff.same :fib1 :fib2
local [line1 line2]
do.while [make "line1 getline :fib1
          make "line2 getline :fib2
          if and listp :line1 listp :line2 [stop]    ; Both files ended.
] [equalp :line1 :line2]
addline :fib1 :line1                                 ; Difference found.
addline :fib2 :line2
diff.differ :fib1 :fib2
end
```

```
to getline :fib
nextlinenum :fib
output readline :fib
end
```

Most of the names you don't recognize are selectors and *mutators* for the FIB abstract data type. (A mutator is a procedure that changes the value of an existing datum, such as `setitem` for arrays.) One new Berkeley Logo primitive used here is `do.while`. It takes two inputs, an instruction list and an expression whose value is `true` or `false`. `Do.while` first carries out the instructions in the first input. Then it evaluates the predicate expression. If it's `true`, then `do.while` repeats the process, carrying out the instructions and evaluating the predicate, until the predicate becomes `false`. In this case, the idea is "Keep reading lines as long as `:line1` and `:line2` are equal."

`Getline` reads a line, either from the file or from the saved lines, and also adds one to the current line number by invoking `nextlinenum`:

```
to nextlinenum :fib
setitem 3 :fib (item 3 :fib)+1
end
```

This is a typical mutator; I won't show the others until the complete program listing at the end of the chapter.

## Comparing and Remembering Unequal Lines

```
to diff.differ :fib1 :fib2
local "line
make "line readline :fib1
addline :fib1 :line
ifelse memberp :line lines :fib2 ~
       [diff.found :fib1 :fib2] ~
       [diff.differ :fib2 :fib1]
end
```

`Diff.differ` reads a line (perhaps a saved line) from one of the files, adds it to the collection of pending lines (not saved lines!) for that file, then looks to see whether a line equal to this one is pending in the other file. If so, then we may have found the end of the changed area, and `diff.found` is called to make sure there is a second pair of equal lines following these two. If not, we must read a line from the other file; this is accomplished by a recursive call to `diff.differ` with the two inputs in reversed order.

What was `:fib1` this time will be `:fib2` in the recursive call, and vice versa. (This is why the FIB data type must include a record of which is the original file 1 and file 2.)

The reason that `diff.differ` uses `readline` rather than `getline` to read from the input files is that it doesn't advance the line number. When dealing with a difference between the files, we are keeping a range of lines from each file, not just a single line. The line number that the program keeps in the FIB is that of the *first* different line; the line number of the last different line will be computed by the `report` procedure later.

```
to diff.found :fib1 :fib2
local "lines
make "lines member2 (last butlast lines :fib1) ~
                    (last lines :fib1) ~
                    (lines :fib2)
ifelse emptyp :lines ~
       [diff.differ :fib2 :fib1] ~
       [report :fib1 :fib2 (butlast butlast lines :fib1)
               (firstn (lines :fib2) (count lines :fib2)-(count :lines))]
end
```

`Diff.found` is called when the last line read from file 1 matches some line pending from file 2. Its job is to find out whether the last *two* lines from file 1 match two consecutive lines from file 2. Most of the work is done by the straightforward helper procedure `member2`, which works this way:

```
> show member2 "and "joy [she's my pride and joy etcetera]
[and joy etcetera]

> show member2 "pride "joy [she's my pride and joy etcetera]
[]
```

If the first two inputs are consecutive members of the third, then `member2` outputs the portion of its third input starting from the point at which the first input was found. If not, then `member2` outputs the empty list.

If `member2`'s output is empty, we continue reading lines from the two files by invoking `diff.differ`. If not, then we've found the end of a change, and we invoke `report` to print the results. The first two inputs to `report` are the two files; the third and fourth are the corresponding sets of unequal lines. The unequal lines from file 1 are all but the last two, the ones we just matched; the unequal lines from file 2 are all but the ones that `member2` output. Helper procedure `firstn` is used to select those lines.

## Reporting a Difference

The `report` procedure is somewhat lengthy, but mostly because differences in which one of the sets of lines is empty are reported specially (as an insertion or a deletion, rather than as a change).

```
to report :fib1 :fib2 :lines1 :lines2
local [end1 end2 dashes]
if equalp (which :fib1) 2 [report :fib2 :fib1 :lines2 :lines1 stop]
print "==========
make "end1 (linenum :fib1)+(count :lines1)-1
make "end2 (linenum :fib2)+(count :lines2)-1
make "dashes "false
ifelse :end1 < (linenum :fib1) [
    print (sentence "INSERT :end1+1 (word (linenum :fib2) "- :end2))
] [ifelse :end2 < (linenum :fib2) [
    print (sentence "DELETE (word (linenum :fib1) "- :end1) :end2+1)
] [
    print (sentence "CHANGE (word (linenum :fib1) "- :end1)
                           (word (linenum :fib2) "- :end2))
    make "dashes "true
]]
process :fib1 "|< | :lines1 :end1
if :dashes [print "-----]
process :fib2 "|> | :lines2 :end2
diff.same :fib1 :fib2
end

to process :fib :prompt :lines :end
foreach :lines [type :prompt print ?]
savelines :fib butfirst butfirst chop :lines (lines :fib)
setlines :fib []
setlinenum :fib :end+2
end
```

Here's how to read `report`: The first step is to ensure that the files are in the proper order, so that `:fib1` is file number 1. (If not, `report` invokes itself with its inputs reordered.) The next step is to compute the ending line number for each changed section; it's the starting line number (found in the file data structure) plus the number of unmatched lines, minus one. `Report` then prints a header, choosing `INSERT`, `DELETE`, or `CHANGE` as appropriate. Finally, it invokes `process` once for each file.

`Process` prints the unmatched lines, with the appropriate file indicator (< or >). Then it takes whatever pending lines were not included in the unmatched group and

transfers them to the saved lines, so that they will be read again. (As a slight efficiency improvement, `process` skips over the two lines that we know matched two lines in the other file; there's no need to read those again.) The set of pending lines is made empty, since no file difference is pending. Finally, the line number in the file structure is increased to match the position following the two lines that ended the difference.

If `process` confuses you, look back at the example I gave earlier, when I first talked about saving and re-reading lines. In that example, the lines from "Original line 7" to "Original line 9" in the first file are the ones that must be moved from the list of pending lines to the list of saved lines. (No lines will be moved in the second file, since that one had the longer set of lines in this difference, six lines instead of three.)

By the way, in the places where the program adds or subtracts one or two in a line number calculation, I didn't work those out in advance. I wrote the program without them, looked at the wrong results, and then figured out how to correct them!

## Program Listing

I've discussed the most important parts of this program, but not all of the helper procedures. If you want to understand the program fully, you can read this complete listing:

```
to diff :file1 :file2 :output
local "caseignoredp
make "caseignoredp "false
openread :file1
openread :file2
if not emptyp :output [openwrite :output]
setwrite :output
print [DIFF results:]
print sentence [< File 1 =] :file1
print sentence [> File 2 =] :file2
diff.same (makefile 1 :file1) (makefile 2 :file2)
print "==========
setread []
setwrite []
close :file1
close :file2
if not emptyp :output [close :output]
end
```

```
;; Skip over identical lines in the two files.

to diff.same :fib1 :fib2
local [line1 line2]
do.while [make "line1 getline :fib1
          make "line2 getline :fib2
          if and listp :line1 listp :line2 [stop]    ; Both files ended.
] [equalp :line1 :line2]
addline :fib1 :line1                                 ; Difference found.
addline :fib2 :line2
diff.differ :fib1 :fib2
end

;; Remember differing lines while looking for ones that match.

to diff.differ :fib1 :fib2
local "line
make "line readline :fib1
addline :fib1 :line
ifelse memberp :line lines :fib2 ~
       [diff.found :fib1 :fib2] ~
       [diff.differ :fib2 :fib1]
end

to diff.found :fib1 :fib2
local "lines
make "lines member2 (last butlast lines :fib1) ~
                    (last lines :fib1) ~
                    (lines :fib2)
ifelse emptyp :lines ~
       [diff.differ :fib2 :fib1] ~
       [report :fib1 :fib2 (butlast butlast lines :fib1)
             (firstn (lines :fib2) (count lines :fib2)-(count :lines))]
end

to member2 :line1 :line2 :lines
if emptyp butfirst :lines [output []]
if and equalp :line1 first :lines equalp :line2 first butfirst :lines ~
   [output :lines]
output member2 :line1 :line2 butfirst :lines
end
```

```
to firstn :stuff :number
if :number = 0 [output []]
output fput (first :stuff) (firstn butfirst :stuff :number-1)
end

;; Read from file or from saved lines.

to getline :fib
nextlinenum :fib
output readline :fib
end

to readline :fib
if savedp :fib [output popsaved :fib]
setread filename :fib
output readword
end

;; Matching lines found, now report the differences.

to report :fib1 :fib2 :lines1 :lines2
local [end1 end2 dashes]
if equalp (which :fib1) 2 [report :fib2 :fib1 :lines2 :lines1 stop]
print "==========
make "end1 (linenum :fib1)+(count :lines1)-1
make "end2 (linenum :fib2)+(count :lines2)-1
make "dashes "false
ifelse :end1 < (linenum :fib1) [
    print (sentence "INSERT :end1+1 (word (linenum :fib2) "- :end2))
] [ifelse :end2 < (linenum :fib2) [
    print (sentence "DELETE (word (linenum :fib1) "- :end1) :end2+1)
] [
    print (sentence "CHANGE (word (linenum :fib1) "- :end1)
                            (word (linenum :fib2) "- :end2))
    make "dashes "true
]]
process :fib1 "|< | :lines1 :end1
if :dashes [print "-----]
process :fib2 "|> | :lines2 :end2
diff.same :fib1 :fib2
end
```

```
to process :fib :prompt :lines :end
foreach :lines [type :prompt print ?]
savelines :fib butfirst butfirst chop :lines (lines :fib)
setlines :fib []
setlinenum :fib :end+2
end

to chop :counter :stuff
if emptyp :counter [output :stuff]
output chop butfirst :counter butfirst :stuff
end

;; Constructor, selectors, and mutators for File Information Block (FIB)
;; Five elements: file number, file name, line number,
;; differing lines, and saved lines for re-reading.

to makefile :number :name
local "file
make "file array 5               ; Items 4 and 5 will be empty lists.
setitem 1 :file :number
setitem 2 :file :name
setitem 3 :file 0
output :file
end

to which :fib
output item 1 :fib
end

to filename :fib
output item 2 :fib
end

to linenum :fib
output item 3 :fib
end

to nextlinenum :fib
setitem 3 :fib (item 3 :fib)+1
end

to setlinenum :fib :value
setitem 3 :fib :value
end
```

```
to addline :fib :line
setitem 4 :fib (lput :line item 4 :fib)
end

to setlines :fib :value
setitem 4 :fib :value
end

to lines :fib
output item 4 :fib
end

to savelines :fib :value
setitem 5 :fib (sentence :value item 5 :fib)
end

to savedp :fib
output not emptyp item 5 :fib
end

to popsaved :fib
local "result
make "result first item 5 :fib
setitem 5 :fib (butfirst item 5 :fib)
output :result
end
```

# 3 Nonlocal Exit

This chapter is about the commands `catch` and `throw`. These commands work together as a kind of super-`stop` command, which you can use to stop several levels of procedure invocation at once.

## Quiz Program Revisited

In Chapter 4 of the first volume, which was about predicates, I posed the problem of a quiz program that would allow three tries to answer each question. Here is the method I used then:

```
to ask.thrice :question :answer
repeat 3 [if ask.once :question :answer [stop]]
print sentence [The answer is] :answer
end

to ask.once :question :answer
print :question
if equalp readlist :answer [print [Right!] output "true]
print [Sorry, that's wrong.]
output "false
end
```

I remarked that `ask.once` acts like a command, in that it has an effect (printing stuff), but it's also an operation, which outputs `true` or `false`. What it *really* wants to do is not output a value but instead be able to stop not only itself but also its calling procedure `ask.thrice`. Here is another version that allows just that:

```
to qa :question :answer
catch "correct [ask.thrice :question :answer]
end

to ask.thrice :question :answer
repeat 3 [ask.once :question :answer]
print sentence [The answer is] :answer
end

to ask.once :question :answer
print :question
if equalp readlist :answer [print [Right!] throw "correct]
print [Sorry, that's wrong.]
end
```

To understand this group of procedures, start with `ask.thrice` and suppose the player keeps getting the wrong answer. Both `ask.once` and `ask.thrice` are straightforward commands; the `repeat` instruction in `ask.thrice` is simpler than it was in the other version.

Now what if the person answers correctly? `Ask.once` then evaluates the instruction

```
throw "correct
```

`Throw` is a command that requires one input, which must be a word, called a "tag." The effect of `throw` is to stop the current procedure, like `stop`, and to keep stopping higher-level procedures until it reaches an active `catch` whose first input is the same as the input to `throw`.

If that sounds confusing, don't give up; it's because I haven't explained `catch` and you have to understand them together. The description of `catch` is deceptively simple: `Catch` is a command that requires two inputs. The first must be a word (called the "catch tag"), the second a list of Logo instructions. The effect of `catch` is the same as that of `run`—it evaluates the instructions in the list. `Catch` pays no attention to its first input. That input is there only for the benefit of `throw`.

In this example program `qa` invokes `catch`; `catch` invokes `ask.thrice`, which invokes `repeat`, which invokes `ask.once`. To understand how `throw` works, you have to remember that primitive procedures are just as much procedures as user-defined ones. That's something we're sometimes lax about. A couple of paragraphs ago, I said that `ask.once` evaluates the instruction

```
throw "correct
```

if the player answers correctly. That wasn't really true. The truth is that `ask.once` evaluates the instruction

```
if equalp readlist :answer [print [Right!] throw "correct]
```

by invoking `if`. It is the procedure `if` that actually evaluates the instruction that invokes `throw`. I made a bit of a fuss about this fine point when we first met `if`, but I've been looser about it since then. Now, though, we need to go back to thinking precisely. The point is that there is a `catch` procedure in the collection of active procedures (`qa`, `catch`, `ask.thrice`, and so on) at the time `throw` is invoked.

(In Chapter 9 of the first volume, I made the point that primitives count as active procedures and that `stop` stops the lowest-level invocation of a user-defined procedure. I said that it would be silly for `stop` to stop only the `if` that invoked it, but that you could imagine `stop` stopping a `repeat`. I gave

```
repeat 100 [print "hello if equalp random 5 0 [stop]]
```

as an example of something that doesn't work. But we can make it work this way:

```
catch "done [repeat 100 [print "hello
                         if equalp random 5 0 [throw "done]]]
```

The `throw` stops the `if`, the `repeat`, and the `catch`. Here's a little quiz for you: Why don't I say that the `throw` stops the `equalp`?)

## Nonlocal Exit and Modularity

`Throw` is called a "nonlocal exit" because it stops not only the (user-defined) procedure in which it is used but also possibly some number of superprocedures of that one. Therefore, it has an effect on the program as a whole that's analogous to the effect of changing the value of a variable that is not local to the procedure doing the changing. If you see a `make` command used in some procedure, and the variable whose name is the first input isn't local to the same procedure, it becomes much harder to understand what that procedure is really doing. You can't just read that procedure in isolation; you have to think about all its superprocedures too. That's why I've been discouraging you from using global variables.

`Throw` is an offense against modularity in the same way. If I gave you `ask.once` to read, without having shown you the rest of the program, you'd have trouble understanding

it. The point may not seem so important when you're reading the small example programs in this book, but when you are working on large projects, with 30 or 300 procedures in them, it becomes much more important.

If I were going to use `catch` and `throw` in this quiz project, one thing I might do is rename `ask.thrice` and `ask.once` as `qa1` and `qa2`. These names would make it clear that the three procedures are meant to work together and indicate which is a subprocedure of which. That name change would help a reader of the program. (Remember that `qa` and its friends are not the whole project; they're all subprocedures of a higher-level `quiz` procedure. So grouping them with similar names really does distinguish them from something else.)

## Nonlocal Output

Consider this procedure that takes a list of numbers as its input and computes the product of all the numbers:

```
to multiply :list
if emptyp :list [output 1]
output (first :list) * (multiply butfirst :list)
end
```

Suppose that we intend to use this procedure with very large lists of numbers, and we have reason to believe that many of the lists will include a zero element. If any number in the list is zero, then the product of the entire list must be zero; we can save time by giving an output of zero as soon as we discover this:

```
to multiply :list
if emptyp :list [output 1]
if equalp first :list 0 [output 0]
output (first :list) * (multiply butfirst :list)
end
```

This is an improvement, but not enough of one. To see why, look at this trace of a typical invocation:

```
? trace "multiply
? print multiply [4 5 6 0 1 2 3]
( multiply [4 5 6 0 1 2 3] )
 ( multiply [5 6 0 1 2 3] )
  ( multiply [6 0 1 2 3] )
   ( multiply [0 1 2 3] )
   multiply outputs 0
  multiply outputs 0
 multiply outputs 0
multiply outputs 0
0
```

Each of the last three lines indicates an invocation of `multiply` in which the zero output by a lower level is multiplied by a number seen earlier in the list: first 6, then 5, then 4. It would be even better to avoid those extra multiplications:

```
to multiply :list
output catch "zero [mul1 :list]
end

to mul1 :list
if emptyp :list [output 1]
if equalp first :list 0 [(throw "zero 0)]
output (first :list) * (mul1 butfirst :list)
end

? trace [multiply mul1]
? print multiply [4 5 6 0 1 2 3]
( multiply [4 5 6 0 1 2 3] )
 ( mul1 [4 5 6 0 1 2 3] )
  ( mul1 [5 6 0 1 2 3] )
   ( mul1 [6 0 1 2 3] )
    ( mul1 [0 1 2 3] )
multiply outputs 0
0
```

This time, as soon as `mul1` sees a zero in the list, it arranges for an immediate return to `multiply`, without completing the other three pending invocations of `mul1`.

In the definition of `mul1`, the parentheses around the invocation of `throw` are required, because in this situation we are giving `throw` an optional second input. When given a second input, `throw` acts as a super-`output` instead of as a super-`stop`. That is, `throw` finds the nearest enclosing matching `catch`, as usual, but arranges that that matching `catch` outputs a value, namely the second input to `throw`. In this example, the word `zero` is the catch tag, and the number `0` is the output value.

The same trick that I've used here for efficiency reasons can also be used to protect against the possibility of invalid input data. This time, suppose that we want to multiply a list of numbers, but we suspect that occasionally the user of the program might accidentally supply an input list that includes a non-numeric member. A small modification will prevent a Logo error message:

```
to multiply :list
output catch "early [mul1 :list]
end

to mul1 :list
if emptyp :list [output 1]
if not numberp first :list [(throw "early "non-number)]
if equalp first :list 0 [(throw "early 0)]
output (first :list) * (mul1 butfirst :list)
end

? print multiply [781 105 87 foo 24 13 6]
non-number
```

I've changed the catch tag, even though Logo wouldn't care, because using the word `zero` as the tag is misleading now that it also serves the purpose of catching non-numeric data.

## Catching Errors

On the other hand, if we don't expect to see invalid data very often, then checking every list member to make sure it's a number is needlessly time-consuming; also, this "defensive" test makes the program structure more complicated and therefore harder for people to read. Instead, I'd like to be able to multiply the list members, and let Logo worry about possible non-numeric input. Here's how:

```
to multiply :list
catch "error [output mul1 :list]
output "non-number
end

to mul1 :list
if emptyp :list [output 1]
output (first :list) * (mul1 butfirst :list)
end
```

```
? print multiply [3 4 5]
60
? print multiply [3 four 5]
non-number
```

To understand how this works, you must know what Logo does when some primitive procedure (such as `*` in this example) complains about an error. The Logo error handler automatically carries out the instruction

```
throw "error
```

If this `throw` "unwinds" the active procedures all the way to top level without finding a corresponding `catch`, then Logo prints the error message. If you do catch the error, no message is printed.

If you are paused (see Chapter 15 of the first volume), the situation is a little more complicated. Imagine that there is a procedure called `pause.loop` that reads and evaluates the instructions you type while paused. The implicit `throw` on an error can be caught by a `catch` that is invoked "below" that interactive level. That is, during the pause you can invoke a procedure that catches errors. But if you don't do that, `pause.loop` will catch the error and print the appropriate message. (You understand, I hope, that this is an imaginary procedure. I've just given it a name to make the point that the interactive instruction evaluator that is operating during a pause is midway through the collection of active procedures starting with the top-level one and ending with the one that caused the error.) What all this means, more loosely, is that an error during a pause can't get you all the way back to top level, but only to where you were paused.

You should beware of the fact that stopping a program by typing control-C or command-period, depending on the type of computer you're using, is handled as if it were an error. That is, it can be caught. So if you write a program that catches errors and never stops, you're in trouble. You may have to turn the computer off and start over again to escape!

If you use the `item` primitive to ask for more items than are in the list, it's an error. Here are two versions of `item` that output the empty list instead:

```
to safe.item1 :number :list
if :number < (1+count :list) [output item :number :list]
output []
end
```

```
to safe.item2 :number :list
catch "error [output item :number :list]
output []
end
```

The first version explicitly checks, before invoking `item`, to make sure the item number is small enough. The second version goes ahead and invokes `item` without checking, but it arranges to catch any error that happens. If there is no error, the `output` ends the running of the procedure. If we get to the next instruction line, we know there must have been an error. The second version of the procedure is a bit faster because it doesn't have to do all that arithmetic before trying `item`. Also, the first version only tests for one possible error; it will still bomb out, for example, if given a negative item number. The second version is safe against *any* bad input.

This technique works well if the instruction list `output`s or `stop`s. But what if we want to do something like

```
catch "error [make "variable item 7 :list]
```

and we want to put something special in the variable if there is an error? In this example, the procedure will continue to its next instruction whether or not an error was caught. We need a way to ask Logo about any error that might have happened. For this purpose we use the operation `error`. This operation takes no inputs. It outputs a list with information about the most recently caught error. If no error has been caught, it outputs the empty list. Otherwise it outputs a list of four members: a numeric error code, the text of the error message that would otherwise have been printed, the name of the procedure in which the error happened, and the instruction line that was being evaluated.

```
to sample
catch "error [print :nonexistent]
show error
end

? sample
[11 [nonexistent has no value] sample
    [catch "error [print :nonexistent]]]
```

But for now all that matters is that the output will be nonempty if an error was caught. So I can say

```
catch "error [make "variable item 7 :list]
if not emptyp error [make "variable []]
```

This will put an empty list into the variable if there is an error in the first line.

You can only invoke `error` once for each caught error. If you invoke `error` a second time, it will output the empty list. That's so that you don't get confused by trying to catch an error twice and having an error actually happen the first time but not the second time. If you'll need to refer to the contents of the `error` list more than once, put it in a variable.

Just in case you've previously caught an error without invoking `error`, it's a good idea to use the instruction

```
ignore error
```

before catching an error and invoking `error` to test whether or not the error occurred. `Ignore` is a Berkeley Logo primitive that takes one input and does nothing with it; the sole purpose of the instruction is to "use up" any earlier caught error so that the next invocation of `error` will return an empty list if no error is caught this time.

## Ending It All

You can stop all active procedures and return to top level by evaluating the instruction

```
throw "toplevel
```

This is a special kind of `throw` that can't be caught.

You've seen this instruction before, in the first volume, where I mentioned it as a way to get out of a pause. That's where it's most useful. Before you use it in a procedure, though, you should be sure that you *really* want to stop everything. For example, suppose you're writing a game program. If the player gets zapped by an evil Whatzit, he's dead and the game is over. So you write

```
to zap.player
print [You're dead!]
throw "toplevel
end
```

because `zap.player` might be invoked several levels deep, but you want to stop everything. But one day you decide to take three different games you've written and combine them into a single program:

```
to play
local "gamename
print [You can play wumpus, dungeon, or rummy.]
print [Which do you want?]
make "gamename first rl
if :gamename = "wumpus [wumpus]
if :gamename = "dungeon [dungeon]
if :gamename = "rummy [rummy]
if not memberp :gamename [wumpus dungeon rummy] [print [No such game!]]
play
end
```

Now your game is no longer the top-level procedure. `play` wants to keep going after a game is over. By throwing to toplevel in the game program, you make that impossible.

# 4 Example: Solitaire

Program file for this chapter: `solitaire`

This program deals out a hand of solitaire and maintains a picture of the card layout as you play the game by entering commands to move cards. It doesn't try to provide help with strategy, but it does know the rules for legal moves.

This chapter follows Chapter 3 because the solitaire program uses `catch` and `throw` for three kinds of nonlocal exit. The program is an infinite loop that plays games repeatedly, so there is an exit command that is implemented as a `throw`. Each game is itself an infinite loop, processing user commands repeatedly until either the game is won or the user asks to start a new game. The command to start a new game is also implemented as a `throw`. Finally, if the program detects an error in a user command, such as asking to move a card that isn't playable, the program rings a bell and `throw`s back to the command-reading loop.

```
to solitaire
...initialization...
catch "exit [forever [onegame]]
end

to onegame
...initialization...
catch "endgame [forever [catch "bell [parsecmd]]]
end
```

## The User Interface

But what I actually find most interesting about this program is the way in which it interacts with the user. By now, most people have seen computer solitaire programs in which the cards are drawn graphically on the screen, and the user moves cards by dragging with a

mouse. (A program of that kind is included with Microsoft Windows, and versions are also available for most other computer systems.) The advantage of the mouse interface is that it's very easy to learn. Once you've seen how dragging an object with the mouse works in a painting program or a word processor, it's immediately obvious how to drag cards in the solitaire program, without reading an instruction manual.

This Logo solitaire program doesn't use a mouse. Instead, you move cards with keyboard commands. Most of the time it takes a single keystroke to tell the program which card to move, and where to move it. The trouble is that you have to learn the command keys! Given the choice, I think that most people would rather start playing right away with a mouse-driven program than take the time to learn to use mine. But I actually find the Logo program *easier* to use. Typing a single key is faster and easier on the wrist than moving the mouse to where the card is, holding down the mouse button, moving the mouse to where you want to put the card, and then releasing the button.

There's no question that mouse-based graphical user interfaces have vastly increased the acceptance and use of computers by people who are not technical experts. And I was happy to have a mouse-based drawing program to produce many of the illustrations in these books! But I did the word processing with a keyboard-controlled text editor; I find it easier to use and more flexible than the mouse-based word processors. Maybe it's just incipient old age, but I'm still a holdout against the idea that *everything* is better done with a mouse.

Play several games using this program, and several using a mouse-based solitaire program, and see what you think.

## The Game of Solitaire

On the next page is a picture of a solitaire game in progress.

In the center of the picture are seven *stacks* of cards. Each stack may include some *hidden* cards and some *shown* cards. The hidden cards, if any, are beneath the shown cards. If there are any cards at all in a stack, at least one must be shown. Cards that are not part of this layout are held in the *hand* and dealt from the hand onto the *pile*; the cards in the hand are hidden, while the top card of the pile is visible. At the top of the picture are four more piles of cards, one for each suit; I'll call these piles "the *top*" so that I can reserve the name *pile* for the one at the bottom.

Here is how the same layout would be represented by the program:

```
                AS      2D

                _       _       _       _
KS      7D      KD      3H      9H      5S      JH
QH              QS              8S      4D
JS              JD                      3S
10D             10S
9C              9D
8H              8C
7S              7H
                6S
                5H
                4S
                3D
                2C
                                        3       5D
```

Shown cards are represented on the screen by the rank and suit of the card. Several cards may be shown in each stack, while only one card is shown in the pile, and one of each suit in the top. Each stack has a dash at the top of its display if there are any hidden cards in that stack; the hand is represented by the number of cards in it.

In playing solitaire it's important to distinguish black cards from red cards, so the program does its best to present the color information to you. The facilities for

color display vary tremendously between computer models, both in what capabilities are available and in the means by which a program can use them. Berkeley Logo sacrifices versatility for uniformity; there is a `standout` primitive operation that can be used to print text in "reverse video," whichever of black-on-white and white-on-black isn't the usual presentation.

```
? print (word "c (standout "red) "it)
credit
```

The `solitaire` program displays red cards in normal text and black cards in reverse video. The DOS version normally displays white text on a black background, while the Macintosh version normally displays black text on a white background, so the effect looks different on each kind of computer.

There are many variations in the rules of solitaire, so I should describe in detail the version this program knows. In the initial layout, there are seven stacks. The first stack (on the left) has one shown card. The second has one shown and one hidden. The third has one shown and two hidden. Each stack has one more hidden card than the one before it, so the seventh stack, at the right, has one shown card and six hidden cards. There are 28 cards altogether on the board; the remaining 24 cards are in the hand.

Here are the legal moves:

1. Three cards at a time may be dealt from the hand to the pile. The cards are turned face up, so that the last one dealt is shown. If there are fewer than three cards in the hand, however many cards are left may be dealt in this way. If there are no cards in the hand at all, the entire pile may be picked up and turned upside down, so that they return to the hand in the same order they were in at the beginning.
2. The top card of the pile, or the topmost card of any stack, may be moved to the top if (a) it is an ace, *or* (b) the card of the same suit and the immediately preceding rank is visible at the top. For example, the four of clubs can be played onto the three of clubs at the top.
3. The top card of the pile, or any shown card in any stack, may be moved onto a stack if the topmost card of that stack is (a) of the opposite color, *and* (b) of the immediately following rank as the card you are moving. For example, the four of clubs can be played onto the five of hearts or the five of diamonds on a stack.
4. When a card is moved onto a stack, it is placed so that it does not completely cover any other shown cards on that stack. Any such shown cards remain shown.

5. When moving a shown card from a stack, any other cards that are above it (partly covering it, because they were moved onto it earlier) must be moved along with it.
6. When all shown cards are removed from a stack, the topmost hidden card is turned over so that it becomes a shown card. If there are no hidden cards in that stack, the stack becomes empty. (At the beginning of the game, there are no empty stacks.)
7. Any king that is the top card of the pile, or a shown card in any stack, may be moved onto an empty stack.
8. The game is won if all cards are moved to the top. The game is lost if there are no legal moves and not all cards are moved to the top.

I've expressed these rules in more formal language than would usually be used. Card players have shorthand ways of speaking, like "play up in the same suit at the top" or "play down in the opposite color on the stacks." I wanted to be very precise in stating the rules because each part of a rule must be reflected in the computer program. Even so, I've left out some details. For example, my list of rules talks about concepts like "suit" and "rank" without defining them. I haven't specified the rank order, namely ace-low. (That is, ace comes before two, not after king.) What other details, if any, have I forgotten?

## Running the Program

To use the program, invoke the command `solitaire` with no inputs. (Being as I am a lazy typist, I've also defined an abbreviation `s` for this command.) The program prints an initial screenful of instructions, and then repeatedly deals solitaire hands until you give the exit command. Here are the instructions:

```
Welcome to solitaire

Here are the commands you can type:
    + =  Deal three cards onto pile
    P    Play top card from pile
    R    Redisplay the board
    ?    Retype these instructions
    card Play that card
    M    Move same card again
    W    Play up as much as possible (Win)
    G    Give up (start a new game)
    X    Exit to Logo
```

```
A card consists of a rank:
   A 2 3 4 5 6 7 8 9 10 J Q K  or T for 10
followed by a suit:
   H S D C
or followed by . to play all possible suits up

If you make a mistake, hit delete or backspace.

To move an entire stack,
   hit the shifted stack number:
     ! @ # $ % ^ & for stacks
     1 2 3 4 5 6 7
```

My goal in designing the "human interface" for this program was that most moves should require typing only a single character. My idea is that the most common moves are to play a card from the pile and to move an entire stack (that is, the entire shown part of a stack) at once. There are one-character commands for all these. If you want to move only part of a stack, then you must type the name of the card, in the form `8S` for the eight of spades.

As it turns out, in this case what's easy for the user is also easiest for the program. When you refer to a card by its position, it's easy for the program to look up which card you mean. For example, when you say `P` to play the top card from the pile, it's easy for the program to find out what card that is. But when you specify a card by typing its rank and suit, it's harder for the program to know *where* that card is. The program must check the pile and all the stacks to see if your card is a member of each one. So the program runs faster if you use the single-keystroke commands.

The instructions don't say how to let the program know where you want to move the chosen card *onto*. The reason is that in most cases there is only one possible place, and the program finds that place itself. (This is the most complicated part of the program.) Sometimes the chosen card can be moved to two different places. If so, the program picks a stack to move the card onto, and if you don't like the program's choice, you can type `M` to move the same card again until it ends up where you wanted it.

The program makes no effort to help you with strategic decisions. For example, some people like to play cards to the top as soon as possible, while other people prefer to keep cards visible on the stacks as long as possible. Such choices are up to you. Also, the program does not detect losing the game. (Detecting winning is easy—all four tops have kings showing—but you haven't lost the game until no further moves are possible, which is harder for the program to figure out.) When you decide the game is over, you just type `G` to start another game.

## Program Structure

There are about 60 procedures in this program. These procedures can be roughly divided into several purposes:

- initialization
- reading and interpreting keyboard commands
- finding the chosen card in the layout
- finding where the chosen card can move
- moving the card
- displaying the card layout
- miscellaneous user commands
- data abstraction

In the procedures that move cards, the most interesting part of the program, a few important variables are used to communicate what moves should be made:

`card` The card that the user asked to move.

`cards` All the cards that must be moved. (There may be more than one if the requested card is in the middle of a stack.)

`where` The location (before moving) of the chosen card.

`onto` A list of all possible locations to which the card can be moved.

As we'll see later in more detail, the card locations in `:where` and `:onto` are represented in the form of Logo instructions that can be `run` to perform the desired move.

The overall program structure is two nested loops. The top-level procedure `solitaire` repeatedly invokes `onegame`. As its name suggests, each invocation of `onegame` plays one solitaire game. It shuffles and deals the cards, then repeatedly invokes `parsecmd`, which reads a character from the keyboard and chooses the appropriate procedure to carry out the user's command.

(Most user commands require only one character. The situation is a little more complicated if the user types the name of a card, such as `10H` for the ten of hearts, as a command. `Parsecmd` actually treats this as three separate commands. The `1` and the `0` merely record the card's rank in a variable named `digit`. When `parsecmd` sees the letter `H`, which selects the card's suit, it invokes `play.by.name`, which combines the remembered rank with the just-typed suit to determine the desired card.)

## Initialization

Both `solitaire` and `onegame` include initialization instructions. That's because some actions are only required once, such as computing the 52 names of cards in the deck, while others are required for each game, such as shuffling those cards.

Many initialization actions use the Berkeley Logo primitive command `localmake`, which is an abbreviation for a `local` command followed by a `make` command. The program uses no global variables, although the variables that are local to these top-level procedures are available to any procedure within the solitaire program.

For most purposes, the most convenient representation of the deck of cards is as a list. That's because what the program most often does with the deck is to deal a card from it to somewhere else. If the deck is represented as a list in the variable `hand`, then dealing a card is roughly equivalent to these instructions:

```
do.something.with first :hand
make "hand butfirst :hand
```

A list is convenient because `butfirst` can be used to remove a card from the deck. It turns out, however, that *shuffling* the deck is easiest if it's represented as an array. That's because the technique used to shuffle the deck is to exchange pairs of cards repeatedly. In the first step, we swap the 52nd card of the deck with a randomly chosen card (perhaps itself). The newly chosen last card is now exempt from further exchanges. In the second step, the 51st card of the deck is swapped with some card in the remainder of the deck, and so on, for 51 steps. The `setitem` primitive makes it easy to change the value of a member partway through an array. If the deck were represented as a list, each exchange would require making a (slightly changed) copy of the entire list.

The solution to this problem is that both representations, list and array, are used in the program. The `solitaire` procedure creates an array containing the 52 cards. For each game, `onegame` invokes `shuffle`, which shuffles the cards in the array and then uses the primitive `arraytolist` to output a list containing the cards in their new order. That list is used by the other parts of the program.

```
to shuffle :len :array
if :len=0 [output arraytolist :array]
localmake "choice random :len
localmake "temp item :choice :array
setitem :choice :array (item :len-1 :array)
setitem :len-1 :array :temp
output shuffle :len-1 :array
end
```

## Data Abstraction

As in most large programs, the solitaire program uses selectors like `first` and `last` for several different purposes in different contexts. To make the program easier to read and maintain, more meaningful names are used in each context.

For example, cards are represented in the program as words containing the rank and the suit, so the word `8C` represents the eight of clubs. To find the rank of a card, the program must take the `butlast` of the word, and to find the suit, it must take the `last` of the word. (Why not use `first` instead of `butlast` to get the rank? Because if the card happens to be a ten, there are two digits in its rank. The suit is always a single character.) Instead of using these primitive selectors directly, I've defined synonyms:

```
to rank :card
output butlast :card
end

to suit :card
output last :card
end
```

When considering playing a card onto a stack, the program does not have to know the precise suit of the card, but must know whether it's red or black:

```
to redp :card
output memberp (suit :card) :reds
end
```

One complication in dealing with cards is that the program wants to use a card's rank in two different ways. For user interaction (reading commands and displaying cards on the screen) the ranks should be represented using the names for aces and picture cards (`A`, `J`, `Q`, and `K`). But for comparison purposes (such as deciding whether a card can be played on top of another card), it's more convenient to represent all ranks as numbers: 1 for ace, 11 for jack, 12 for queen, and 13 for king. A conversion function `ranknum` makes this possible:

```
to ranknum :rank
if emptyp :rank [output 0]
if numberp :rank [output :rank]
if :rank = "A [output 1]
if :rank = "J [output 11]
if :rank = "Q [output 12]
if :rank = "K [output 13]
end
```

(When would a rank be empty? The `emptyp` test is useful in the case of deciding whether a card can be played onto an empty "top." In general, the only card that can be played onto a top is the rank after the one that's already visible there; for example, if a five is showing, then a six can be played. Treating an empty top as having a rank of zero means that the following rank, an ace, is permitted, just as the rules require.)

In an actual solitaire game, a top is a pile of several cards of the same suit, with an ace on the bottom and other cards over it in sequence. But in the program, there is no need to represent any but the topmost card, since the lower cards have no further role in the game. In this program, the tops are represented by four variables `toph`, `tops`, `topd`, and `topc`. (The last letter indicates the suit.) The value of each variable is the empty word if that top is empty, or the rank of the topmost card if not. Instead of using these variables directly, the program uses data abstraction procedures `top` and `settop` to examine and modify the values:

```
to top :suit
output thing word "top :suit
end

to settop :suit :value
make (word "top :suit) :value
end
```

For example, part of the initialization in `onegame` is to make all four tops empty:

```
foreach :suits [settop ? "]
```

## Stacks

A *stack* (also called a *pushdown list*) is a data structure that is used to remember things and recall them later. A stack uses the rule "Last In, First Out." That is, when you take something out of a stack, the one you get is the one you put in most recently. The name "stack" is based on the metaphor of the spring-loaded stack of trays you find in a self-service cafeteria. You add a tray to the stack by pushing down the trays that were already there, adding the new tray at the top of the pile. When you remove a tray, you take the one at the top of the pile—the one most recently added to the stack.

A pile of cards in a solitaire game works just like a pile of trays in a cafeteria. You add cards to the top of the pile, and you remove cards from the top of the pile. I've used the name "stack" for some of the piles of cards in this project partly because those groups of cards are represented in the program by stacks in the technical sense.

Berkeley Logo provides primitive procedures `push` and `pop` to implement stacks. Each stack is represented as a list. To push something onto the stack, Logo uses `fput`; to pop something off the stack, it uses `first`. (Actually, it's slightly more complicated, as you'll see in a moment. But this is essentially true.) For example, each of the seven numbered card stacks in the solitaire layout is represented by two lists, one for the shown cards and one for the hidden cards. The lists for the third stack are kept in variables named `shown3` and `hidden3`. To *push* a new card onto the hidden stack without using the `push` primitive, you could say

```
make "hidden3 fput :card :hidden3
```

To *pop* a card from that stack, you'd say

```
make "card first :hidden3
make "hidden3 butfirst :hidden3
```

In this case, the first instruction reads the top of the stack, while the second removes that entry from the stack.

Berkeley Logo provides `push` and `pop` as a data abstraction mechanism. `Push` is a command that takes two inputs. The first input is a word, the *name* of a stack. The second input is any Logo datum. `Pop` is an operation with one input, the name of a stack. Its output is the first datum on the stack. It also has the effect of removing that datum from the stack. Instead of the instructions above, you can say

```
push "hidden3 :card
make "card pop "hidden3
```

If Berkeley Logo didn't already provide these procedures, it would be easy to write them:

```
to push :stack :thing
make :stack fput :thing (thing :stack)
end

to pop :stack
local "result
make "result first thing :stack
make :stack butfirst thing :stack
output :result
end
```

Within the definition of push, the expression :stack represents the name of the stack, while the expression thing :stack represents the stack itself. The make instruction is an example of indirect assignment; it does not give a new value to the variable stack but rather to the variable whose name is contained in stack.

Pop is an unusual Logo procedure in that it's an operation that also has an effect. Most operations don't have effects. They compute some value, but they don't make any permanent change in the state of the computer. Another way of looking at this is to say that for most operations, if you apply the same operation to the same inputs repeatedly, you'll get the same result every time.

```
? make "cards [AH 5C 10S]
? print first :cards
AH
? print first :cards
AH
? print first :cards
AH
```

But if you apply pop to the same input repeatedly, you'll get a different output each time.

```
? print pop "cards
AH
? print pop "cards
5C
? print pop "cards
10S
```

The combination of output and effect in pop is a powerful technique, but a potentially confusing one. It's important for anyone who tries to read this program to be aware that pop has an effect. Fortunately, the concept of a stack is a standard, well-known convention in computer science, and the names push and pop are the traditional ones for this purpose, so pop is somewhat self-documenting.

Before a stack can be used, it must be initialized. Generally a stack starts out with no data in it. That is, it's initially an empty list. This initialization could be done with an explicit make instruction, but instead I invented a procedure for the purpose:

```
to setempty :stack
make :stack []
end
```

I think this makes the program slightly more elegant.

It is an error to try to pop more items from a stack than you've pushed onto it. If you try to do this, you'll get an error message something like

```
First doesn't like [] as input
```

Often the logic of a program ensures automatically that you never try to overpop a stack. But in the solitaire program I sometimes have to check for this possibility explicitly, with an instruction like

```
if not emptyp :hidden3 [make "card pop "hidden3]
```

I've been using the name `hidden3` as an example in this discussion, typing `"hidden3` when the name of the stack was needed or `:hidden3` when its value was needed. In fact, such names do not appear explicitly in the program. There are no instructions that are directed exclusively to the third stack. Instead, stack instructions are applied either to all seven stacks or to a stack chosen by the user through keyboard commands. The name of a stack must be *computed* using an expression like

```
word "hidden :num
```

The contents of the stack would be examined by applying `thing` to that expression. To make the program cleaner I created procedures to generate these variable names.

```
to shown :num
output word "shown :num
end

to hidden :num
output word "hidden :num
end
```

Remember that these operations output the *name* of a stack variable, not the contents of a stack. So, for example, you can use them in instructions like these:

```
push (shown 5) :card
make "card pop shown 5
setempty shown 5
```

There are only a few places in the program where a procedure needs to refer to the entire contents of a stack, rather than just pushing or popping a single datum at a time. (One

such place, for example, is `remshown`, which has the job of removing perhaps several cards from a stack.) In those places, there is an explicit use of `thing` to examine the contents of a stack selected by `shown` or `hidden`. An expression that occurred often in the program was

```
emptyp thing shown :num
```

to see if a stack is empty; I cleaned up these expressions somewhat by inventing a special procedure for this test.

```
to stackemptyp :name
output emptyp thing :name
end
```

This is used in an expression like

```
stackemptyp shown :num
```

Note that when a stack *is* mentioned explicitly by name in the program, like `:hand` or `:pile`, it is tested for emptiness with the ordinary `emptyp`. In this case the colon abbreviates the invocation of `thing`; for the `shown` or `hidden` names, `stackemptyp` abbreviates the invocation of `thing`.

One small detail that's easy to miss is that in a non-computer game of solitaire, when a hand is completely dealt out, you pick up the pile from the table and *turn it over* to form a new hand. What was the top card of the pile becomes the bottom card of the hand. The program achieves the same effect while dealing cards:

```
to deal
if emptyp :hand [make "hand reverse :pile setempty "pile]
if emptyp :hand [output []]
output pop "hand
end
```

The Berkeley Logo primitive operation `reverse` is used to reverse the order of the cards as they are moved from the pile to the hand.

## Program as Data

In order for the program to move a card, it must first make sure that the requested move is legal. The first step is to find the card's current position. (That's easy if the move

is requested by position, using the `P` command to play the card at the top of the pile, or a shifted stack number to move the entire shown stack; it's a little harder if the card is requested by its rank and suit. Even then, in order to be playable the card must be either on top of the pile or somewhere in a shown stack.) The next step is to look for another position into which the card can be moved; the only possibilities are a stack or a top. Only after both old and new positions have been verified can the program actually modify its data structures (and the screen display) to move the card.

When you type a card-moving command, `parsecmd` invokes one of three procedures: `playpile` for the `P` command, `playstack` for one of the shifted stack numbers (such as `#` for stack 3), or `play.by.name` for a rank and suit (such as `7D`). The first two of these must figure out which card is desired, and ensure that there is in fact a card in the requested position; `play.by.name` has the opposite job, since it already knows the card and must determine that it's in a playable position. But in either case, these procedures do not actually move the card. They ensure that the variable `card` has the desired card as its value, and that the variable `where` has as its value a representation of the card's current position. Then they call `playcard`, whose job is to ensure that there is a valid destination for the card, and if so, to move it:

```
to playcard
setempty "onto
if not coveredp [checktop]
if and not :upping
       or (emptyp :onto) (not upsafep rank :card)
   [checkonto]
if emptyp :onto [bell]
run :where
run first :onto
end
```

Subprocedures `checktop` and `checkonto` determine whether the requested card can be moved to the top or to a stack. (Each of these is called only if certain conditions are met. For `checktop`, the condition is that the desired card must not be in the middle of a shown stack; it must be either the bottommost card of a shown stack or visible on the pile. The condition for calling `checkonto` is more complicated. If the user's command was `.` or `W`, then cards are played only into the top, so there is no need to check the stacks. In other cases, to make the game move more quickly, the program will always move the card to the top if it is both possible and *safe* to do so. Such a move is considered safe if every card whose rank is less than that of the requested card by two or more is already in the top, because then any card of rank one less than the chosen card can be played to the top, and so the chosen card is not needed in the stacks.)

Just as `:where` identifies the card's current position, `:onto` will hold all of the possible destination positions. `Checktop` and `checkonto` add possible positions to this variable, which is a list. If `:onto` is empty after `checktop` and `checkonto` have been invoked, then there is no legal way to move this card.

I want to focus attention on the two `run` instructions. They are the ones that actually do the work of moving a card from one place to another; the first removes the card from its original position and the second inserts the card at its new position.

The value of the variable `where` is not merely a number or word indicating where the card is to be found, but a Logo instruction that invokes the procedure needed to remove the card. For example, suppose you type the letter `P` to play a card from the top of the pile. `Parsecmd` then invokes `playpile`:

```
to playpile
if emptyp :pile [bell]
if not emptyp :digit [bell]
make "card first :pile
make "where [rempile]
carddis :card
playcard
end
```

The first two instructions check for errors. The first checks for trying to play a card from the pile when there are no cards in the pile. The second checks for the *syntax* error of typing a rank and then typing `P` instead of a suit. Having cleared those hurdles, the next instruction finds the actual card (rank and suit) you want to play from the pile. The interesting part for the present discussion is that the variable `where` is given as its value the list

```
[rempile]
```

`Rempile` is the name of a procedure with no inputs, so this list contains a valid Logo instruction. The corresponding instruction in `playstack1` is

```
make "where sentence "remshown :num
```

which gives `where` a value like

```
[remshown 4]
```

if you've selected stack four. In either case, `playcard` can later remove the card from its original location just by `running :where`. At the same time, this Logo *instruction* can be examined as a *datum*. For example, `coveredp` contains the instruction

```
if equalp :where [rempile] [output "false]
```

Most programming languages don't have a facility like Logo's `run` command. In those languages, the variable `where` would have to contain, for example, a number indicating where the card is to be found. `Playcard` would then use this number to choose a course of action with a series of instructions like this:

```
if :where = 0 [rempile]
if :where = 1 [remshown 1]
if :where = 2 [remshown 2]
```

... and so on.

The situation concerning the variable `onto` is similar, except that there is a slight complication because there may be more than one legal destination for a card. (By contrast, every card starts out in exactly one place!) Therefore, `checktop` and `checkonto` set up `:onto` as a *list* of Logo instructions, one for every possible destination. If a card could be played onto stack 3, stack 6, or the top, `:onto` will be

```
[[playonto 3] [playonto 6] [playtop]]
```

`Playcard` runs the first member of this list. Why bother saving the other members? After a successful move, the user can type `M` to move the same card to a different destination. Here's how that is done:

```
to again
if not emptyp :digit [bell]
if emptyp :onto [bell]
make "where list "remshown last pop "onto
if emptyp :onto [bell]
carddis :card
run :where
run first :onto
end
```

This procedure uses the values that are still left over in `:card` and `:onto` from the last move. The first member of `:onto` is the instruction that moved the card onto a stack. (If the card was moved to the top, it's because there were no alternatives in `:onto`, because

playtop is always the last choice in the list.) That stack is now the card's position of origin! If :onto was

```
[[playonto 3] [playonto 6] [playtop]]
```

then the instruction

```
make "where list "remshown last pop "onto
```

will give where the value

```
[remshown 3]
```

and, because pop removes the first datum from the stack, leaves onto with the value

```
[[playonto 6] [playtop]]
```

The chosen card will be moved from stack three to stack six. If the user types M again, then the card will be moved from stack six to the top.

## Multiple Branching

Consider the procedure that interprets what you type at the keyboard while running the solitaire program:

```
to parsecmd                ;; abbreviated version
local "char
make "char uppercase readchar
if equalp :char "T [parsedigit 1 parsezero stop]
if memberp :char [1 2 3 4 5 6 7 8 9 A J Q K] [parsedigit :char stop]
if equalp :char "0 [parsezero stop]
if memberp :char :suits [play.by.name :char stop]
if equalp :char ". [allup stop]
if equalp :char "W [wingame stop]
if equalp :char "M [again stop]
; several more possibilities omitted...
bell
end
```

This sort of thing is common in Logo programming: a string of ifs in which each conditional instruction list ends with stop because the choices are mutually exclusive.

Some people find this use of `stop` offensive because it doesn't make it graphically apparent when reading the program that the choices are exclusive. The form of the program makes it seem that each decision (that is, each `if` instruction) is independent of the others.

It would be possible to meet this objection by using `ifelse`, putting each new test in the false part of the previous one:

```
to parsecmd
local "char
make "char uppercase readchar
ifelse equalp :char "T ~
       [parsedigit 1 parsezero]
       [ifelse memberp :char [1 2 3 4 5 6 7 8 9 A J Q K]
               [parsedigit :char]
               [ifelse equalp :char "0 [parsezero]
                       ; ...
                                        bell]]
end
```

It's not clear that this is an improvement, although the use of `ifelse` makes more sense as an alternative to `stop` when only a single decision is involved.

Some programming languages provide a special representation for such a *multiple branching* decision. A Logo equivalent might look like this:

```
to parsecmd
local "char
make "char uppercase readchar
branch [
  [[equalp :char "T] [parsedigit 1 parsezero]]
  [[memberp :char [1 2 3 4 5 6 7 8 9 A J Q K]] [parsedigit :char]]
  [[equalp :char "0] [parsezero]]
  [[memberp :char :suits] [play.by.name :char]]
  [[equalp :char "."] [allup]]
  [[equalp :char "W] [wingame]]
  [[equalp :char "M] [again]]
  ; several more possibilities omitted...
  [["true] [bell]] ]
end
```

`Branch` is a hypothetical command that takes a single input, a list of lists. Each member of the input is a list with two members. The first member must be a Logo predicate expression; the second must be a Logo instruction. `Branch` evaluates the first half of

each pair. If the value is `true`, `branch` then carries out the instruction in the second half of that pair, and then stops without evaluating the remaining pairs. If the value is `false`, `branch` goes on to the next pair. `Branch` is not a Logo primitive, but it can easily be written in Logo:

```
to branch :conditions
if emptyp :conditions [stop]
if (run first first :conditions) [run last first :conditions stop]
branch butfirst :conditions
end
```

Inventing control structures like this is the sort of thing Logo makes easy and other languages make impossible.

The trouble with this particular control structure in Logo is that the input to `branch` is typically a very long list, extending over several lines on the screen. Traditional Logo dialects have not done a good job of presenting such long lists with clear formatting. More recent versions can, however, handle instructions like that multi-line `branch` invocation.

## Further Explorations

I keep thinking of new features I'd like in this program. I think the most important is an Undo command, which would undo the effect of the previous user command. This should be pretty easy to implement; every time the program changes the value of a variable that represents card positions, it should make a list of the variable's name and its old value, and push that onto a changes list. For example, here's the procedure that removes a card from the pile:

```
to rempile
make "cards (list (pop "pile))
dispile
end
```

And here's how I'd change it for the undo command:

```
to rempile
push "undo.list (list "pile :pile)
make "cards (list (pop "pile))
dispile
end
```

The undo command itself would go through the list, restoring the values of all the variables it finds, and then call `redisplay` to make the display match the program's state. `Playcard` would `setempty` the undo list before moving any cards.

Another possibility is to improve the display. One person who tried this program commented that it's not enough to indicate whether the hidden part of a stack is empty or nonempty; he wanted to see exactly how many cards are present. Novice users might be helped by keeping an abbreviated command list in the empty space toward the right side of the screen.

A more ambitious direction you could pursue is to write a similar program for a different solitaire game. There are books of card games that include several variations on this kind of solitaire as well as versions of solitaire that are totally different in their rules and layouts.

Another direction would be to try to have the program offer strategic suggestions, or even play the game entirely by itself. As with any strategy game, you would have to choose between determining the strategy for the program in advance and letting it learn from its experience and modify its strategy. Which is better, playing cards to the top quickly or saving them in the stacks as long as possible? Which is better, playing a card from the pile or playing a card of the same rank and color from the stacks? You could research these questions by writing versions of the program with different strategies and collecting statistics on their performance.

Another possibility would be to abandon solitaire and program the computer to play one side of a two-player game with you. Blackjack is a simple example; poker is a harder one.

A different kind of exploration would be to try to speed up the running of this program. Earlier I suggested the possibility that the program might benefit from remembering explicitly the position of each card. You could find out whether or not that would really help. (It would speed up the searching process for a card, but it would also slow down the moving of cards because the program would have to remember the new location instead of the old one. My guess is that the speedup would be substantial and the slowdown minimal, but I'm not sure.) What other bottlenecks can you find in this program, and how can you improve them?

## Program Listing

If you trace the progress of a user command, let's say `P` to play from the pile, from `parsecmd` through `playpile` and `playcard` to `rempile` and then either `playtop` or

playonto, you'll understand most of the program. There are a few slightly complicated details in moving several cards from one stack to another (remshown and playonto) but nothing really hard to understand.

```
to solitaire
print [Welcome to solitaire]
instruct
localmake "allranks [A 2 3 4 5 6 7 8 9 10 J Q K]
localmake "numranks map "ranknum :allranks
localmake "suits [H S D C]
localmake "reds [H D]
localmake "deckarray (listtoarray (crossmap "word :allranks :suits) 0)
localmake "upping "false
catch "exit [forever [onegame cleartext]]
cleartext
end

to s
solitaire
end

to onegame
print [Shuffling, please wait...]
local [card cards digit pile where]
localmake "onto []
local map [word "top ?] :suits
local cascade 9 [(sentence (word "shown #) (word "hidden #) ?)] []
localmake "ranks :allranks
localmake "numstacks 7
local map [word "num ?] :numranks
foreach :numranks [make word "num ? 4]
localmake "hand shuffle 52 :deckarray
setempty "pile
initstacks
foreach :suits [settop ? "]
redisplay
catch "endgame [forever [catch "bell [parsecmd]]]
end
```

```
;; Initialization

to instruct
print [] print [Here are the commands you can type:]
type "|    | type (sentence standout "+ standout "=)
type "|  | print [Deal three cards onto pile]
instruct1 "P [Play top card from pile]
instruct1 "R [Redisplay the board]
instruct1 "? [Retype these instructions]
instruct1 "card [Play that card]
instruct1 "M [Move same card again]
instruct1 "W [Play up as much as possible (Win)]
instruct1 "G [Give up (start a new game)]
instruct1 "X [Exit to Logo]
print [A card consists of a rank:]
type "|   | print (sentence standout [A 2 3 4 5 6 7 8 9 10 J Q K]
                           "or standout "T [for 10])
print [followed by a suit:]
type "|   | print standout [H S D C]
print (sentence [or followed by] standout ".
               [to play all possible suits up])
print [] print [If you make a mistake, hit delete or backspace.]
print [] print [To move an entire stack,]
type "|   | print [hit the shifted stack number:]
type "|     | print (sentence standout [! @ # $ % ^ &] [for stacks])
type "|     | print [1 2 3 4 5 6 7]
print []
end

to instruct1 :key :meaning
type "|    |
type standout :key
repeat 5-count :key [type "| |]
print :meaning
end

to shuffle :len :array
if :len=0 [output arraytolist :array]
localmake "choice random :len
localmake "temp item :choice :array
setitem :choice :array (item :len-1 :array)
setitem :len-1 :array :temp
output shuffle :len-1 :array
end
```

```
to initstacks
for [num 1 7] [inithidden :num
               turnup :num]
end

to inithidden :num
localmake "name hidden :num
setempty :name
repeat :num [push :name deal]
end

;; Reading and interpreting user commands

to parsecmd
if emptyp :digit [setcursor [1 22] type "|      | setcursor [1 22]]
local "char
make "char uppercase readchar
if equalp :char "T [parsedigit 1 parsezero stop]
if memberp :char [1 2 3 4 5 6 7 8 9 A J Q K] [parsedigit :char stop]
if equalp :char "0 [parsezero stop]
if memberp :char :suits [play.by.name :char stop]
if equalp :char ". [allup stop]
if equalp :char "W [wingame stop]
if equalp :char "M [again stop]
if memberp :char [+ =] [hand3 stop]
if equalp :char "R [redisplay stop]
if equalp :char "? [helper stop]
if equalp :char "P [playpile stop]
if and equalp :char "|(| not emptyp :digit [cheat stop]
if and equalp :char "|)| not emptyp :digit [newstack stop]
if memberp :char [! @ # $ % ^ & * ( )] ~
   [playstack :char [! @ # $ % ^ & * ( )] stop]
if memberp :char (list "| | char 8 char 127) [rubout stop]
if equalp :char "G [throw "endgame]
if equalp :char "X [throw "exit]
bell
end

to parsedigit :char
if not emptyp :digit [bell]
make "digit :char
type :digit
end
```

```
to parsezero
if not equalp :digit 1 [bell]
make "digit 10
type 0
end

to rubout
setcursor [1 22]
type "|    |
setcursor [1 22]
setempty "digit
end

to bell
if not :upping [type char 7]
setempty "digit
throw "bell
end

;; Deal three cards from the hand

to hand3
if not emptyp :digit [bell]
if and emptyp :hand emptyp :pile [bell]
push "pile deal
repeat 2 [if not emptyp :hand [push "pile deal]]
dispile dishand
end

to deal
if emptyp :hand [make "hand reverse :pile setempty "pile]
if emptyp :hand [output []]
output pop "hand
end

;; Select card to play by position (pile or stack) or by name

to playpile
if emptyp :pile [bell]
if not emptyp :digit [bell]
make "card first :pile
make "where [rempile]
carddis :card
playcard
end
```

```
to playstack :which :list
if not emptyp :digit [bell]
foreach :list [if equalp :which ? [playstack1 # stop]]
end

to playstack1 :num
if greaterp :num :numstacks [bell]
if stackemptyp shown :num [bell]
make "card last thing shown :num
make "where sentence "remshown :num
carddis :card
playcard
end

to play.by.name :char
if emptyp :digit [bell]
if equalp :digit 1 [make "digit "a]
type :char
wait 0
make "card word :digit :char
setempty "digit
findcard
if not emptyp :where [playcard]
end

to findcard
if findpile [stop]
make "where findshown
if emptyp :where [bell]
end

to findpile
if emptyp :pile [output "false]
if equalp :card first :pile [make "where [rempile] output "true]
output "false
end

to findshown
for [num 1 :numstacks] ~
    [if memberp :card thing shown :num [output sentence "remshown :num]]
output []
end
```

```
;; Figure out all possible places to play card, then pick one

to playcard
setempty "onto
if not coveredp [checktop]
if and not :upping ~
       or (emptyp :onto) (not upsafep rank :card) ~
   [checkonto]
if emptyp :onto [bell]
run :where
run first :onto
end

to coveredp
if equalp :where [rempile] [output "false]
output not equalp :card first thing shown last :where
end

to upsafep :rank
if memberp :rank [A 2] [output "true]
output equalp 0 thing word "num ((ranknum :rank)-2)
end

to checktop
if (ranknum rank :card) = 1 + (ranknum top suit :card) ~
   [push "onto (list "playtop word "" suit :card)]
end

to checkonto
for [num :numstacks 1] ~
    [ifelse stackemptyp shown :num
            [checkempty :num]
            [checkfull :num thing shown :num]]
end

to checkempty :num
if equalp rank :card "k [push "onto (list "playonto :num)]
end

to checkfull :num :stack
if equalp (redp :card) (redp first :stack) [stop]
if ((ranknum rank first :stack) = 1 + (ranknum rank :card)) ~
   [push "onto (list "playonto :num)]
end
```

```
;; Play card, step 1: remove from old position

to rempile
make "cards (list (pop "pile))
dispile
end

to remshown :num
setempty "cards
remshown1 :num (count thing shown :num)
if stackemptyp shown :num [turnup :num disstack :num]
end

to remshown1 :num :length
do.until [push "cards (pop shown :num)] ~
         [equalp :card first :cards]
for [i 1 [count :cards]] ~
    [setcursor list (5*:num - 4) (5+:length-:i) type "|   |]
end

to turnup :num
setempty shown :num
if stackemptyp hidden :num [stop]
push (shown :num) (pop hidden :num)
end

;; Play card, step 2: put in new position

to playtop :suit
localmake "var word "num ranknum rank :card
settop :suit rank :card
distop :suit
make :var (thing :var)-1
if (thing :var)=0 [make "ranks butfirst :ranks]
end

to playonto :num
localmake "row 4+count thing shown :num
localmake "col 5*:num-4
for [i 1 [count :cards]] ~
    [localmake "card pop "cards
     push (shown :num) :card
     setcursor list :col :row+:i
     carddis :card]
end
```

```
;; Update screen display

to redisplay
cleartext
for [num 1 :numstacks] [disstack :num]
foreach :suits "distop
dispile
dishand
setcursor [1 22]
setempty "digit
end

to disstack :num
setcursor list (-3 + 5 * :num) 4
type ifelse stackemptyp hidden :num ["| |] ["-]
if stackemptyp shown :num [setcursor list (-4 + 5 * :num) 5
                           type "|   | stop]
localmake "stack (thing shown :num)
localmake "col 5*:num-4
for [i [count :stack] 1] ~
    [setcursor list :col :i+4
     carddis pop "stack]
end

to distop :suit
if emptyp top :suit [stop]
if equalp :suit "H [distop1 4 stop]
if equalp :suit "S [distop1 11 stop]
if equalp :suit "D [distop1 18 stop]
distop1 25
end

to distop1 :col
setcursor list :col 2
carddis word (top :suit) :suit
end

to dispile
setcursor [32 23]
ifelse emptyp :pile [type "|   |] [carddis first :pile]
end
```

```
to dishand
setcursor [27 23]
type count :hand
type "| |
end

to carddis :card
ifelse memberp suit :card :reds [redtype :card] [blacktype :card]
type "| |
end

to redtype :word
type :word
end

to blacktype :word
type standout :word
end


;; Miscellaneous user commands

to again
if not emptyp :digit [bell]
if emptyp :onto [bell]
make "where list "remshown last pop "onto
if emptyp :onto [bell]
carddis :card
run :where
run first :onto
end

to allup
if emptyp :digit [bell]
if equalp :digit 1 [make "digit "a]
localmake "upping "true
type ". wait 0
foreach map [word :digit ?] [H S D C] ~
        [catch "bell [make "card ?
                      findcard
                      if not emptyp :where [playcard]]]
setempty "digit
end
```

```
to helper
cleartext
instruct
print standout [type any key to continue]
ignore rc
redisplay
end

to wingame
type "W
localmake "cursor cursor
foreach :ranks [if not upsafep ? [stop]
                make "digit ? ~
                allup ~
                setempty "digit ~
                setcursor :cursor]
if equalp (map "top [H S D C]) [K K K K] ~
   [ct print [you win!] wait 120 throw "endgame]
end

to newstack
localmake "num :numstacks+1
setcursor [1 22] type "|   |
if not equalp :digit 9 [bell]
setempty hidden :num
setempty shown :num
make "numstacks :num
setempty "digit
end

to cheat
setcursor [1 22] type "|   |
if not equalp :digit 8 [bell]
if and emptyp :hand emptyp :pile [bell]
push "pile deal
dispile
dishand
setempty "digit
end

;; Data abstraction (ranks)

to rank :card
output butlast :card
end
```

```
to ranknum :rank
if emptyp :rank [output 0]
if numberp :rank [output :rank]
if :rank = "A [output 1]
if :rank = "J [output 11]
if :rank = "Q [output 12]
if :rank = "K [output 13]
end

;; Data abstraction (suits)

to suit :card
output last :card
end

to redp :card
output memberp (suit :card) :reds
end

;; Data abstraction (tops)

to top :suit
output thing word "top :suit
end

to settop :suit :value
make (word "top :suit) :value
end

;; Data abstraction (card stacks)

to shown :num
output word "shown :num
end

to hidden :num
output word "hidden :num
end

;; Data abstraction (pushdown stacks)

to stackemptyp :name
output emptyp thing :name
end

to setempty :stack
make :stack []
end
```

# 5 Program as Data

In most programming languages there is a sharp distinction between *program* and *data*. Data are the things you can manipulate in your program, things like numbers and letters. These things live in variables, which can be given new values by your program. But the program itself is not subject to manipulation; it's something you write ahead of time, and then it remains fixed.

In Logo the distinction is not so sharp. We've made extensive use of one mechanism by which a program can manipulate itself: the instruction lists that are used as inputs to `run`, `if`, and so on are data that can be computed by a program. For example, the solitaire program in Chapter 4 constructs a list of Logo instruction lists, each of which would move a card to some other legal position, and then says

```
run first :onto
```

to move the card to the first such position.

## `Text` and `Define`

In this chapter we'll use a pair of more advanced tools that allow a program to create more program. `Run` deals with a single *instruction;* now we'll be able to examine and create *procedures*.

`Text` is an operation that takes one input, a word. That word must be the name of a user-defined procedure. The output from `text` is a list. The first member of that list is a list containing the names of the inputs to the chosen procedure. (If the procedure

has no inputs, the list will be empty.)* The remaining members of the output list are instruction lists, one for each line in the definition of the procedure.

Here is an example. Suppose we've defined the procedure

```
to opinion :yes :no
print sentence [I like] :yes
print sentence [I hate] :no
end
```

Here's what the text of that procedure looks like:

```
? show text "opinion
[[yes no] [print sentence [I like] :yes] [print sentence [I hate] :no]]
```

In this example the output from `text` is a list with three members. The first member is a list containing the words `yes` and `no`, the names of `opinion`'s inputs. (Note that the colons that are used to indicate inputs in a title line are *not* used here.) The second and third members of the output list are instruction lists, one for each line in the definition. (Note that there is no `end` line in the definition; as I've remarked before, that line isn't an instruction in the procedure because `end` isn't a command.)

The opposite of `text` is the command `define`. This command takes two inputs. The first must be a word and the second a list. The effect of `define` is to define a procedure whose name is the first input and whose text is the second input. You can use `define` to define a new procedure or to change the definition of an old one. For example, I might redefine `opinion`:

```
? define "opinion [[yes no] [print sentence :yes [is yummy.]]
                            [print sentence :no [is yucky.]]]
? opinion [Ice cream] "Cheese
Ice cream is yummy.
Cheese is yucky.
? po "opinion
to opinion :yes :no
print sentence :yes [is yummy.]
print sentence :no [is yucky.]
end
```

* Berkeley Logo allows user-defined procedures with *optional* inputs. For such a procedure, this first sublist may contain lists, representing optional inputs, as well as words, representing required inputs.

Instead of replacing an old definition with an entirely new one, we can use `define` and `text` together to change a procedure's definition:

```
? define "opinion lput [print sentence :no "stinks!] ~
                       butlast text "opinion
? opinion "Logo "Basic
Logo is yummy.
Basic stinks!
```

(Of course, I didn't have to redefine the same procedure name. I could have said

```
? define "strong.opinion ~
         lput [print sentence :no "stinks!] butlast text "opinion
```

and then I would have had two procedures, the unchanged `opinion` and the new version named `strong.opinion`.)

It may be instructive to consider the analogy between *variables,* which hold data, and *procedures,* which hold instructions. Variables are given values with the `make` command and examined with the operation `thing`. Procedures are given definitions with the `define` command and examined with the operation `text`. (There is no abbreviation for `text`-quote, however, like the dots abbreviation for `thing`-quote.)

To illustrate `define` and `text`, I've used them in instructions typed in at top level. In practice, you wouldn't use them that way; it's easier to examine a procedure with `po` and to change its definition with `edit`. `Text` and `define` are meant to be used not at top level but inside a program.

## Automated Definition

Early in the first volume I defined the operation `second` this way:

```
to second :thing
output first butfirst :thing
end
```

Suppose I want more operations following this model, to be called `third`, `fourth`, and so on. I could define them all by hand or I could write a program to do it for me:

```
to ordinals
ord1 [second third fourth fifth sixth seventh] [output first butfirst]
end
```

```
to ord1 :names :instr
if emptyp :names [stop]
define first :names list [thing] (lput ":thing :instr)
ord1 (butfirst :names) (lput "butfirst :instr)
end

? ordinals
? po "fifth
to fifth :thing
output first butfirst butfirst butfirst butfirst :thing
end
```

(The name `ordinals` comes from the phrase *ordinal numbers,* which is what things like "third" are called. Regular numbers like "three" are called *cardinal numbers.*) This procedure automatically defined new procedures named `second` through `seventh`, each with one more `butfirst` in its instruction line.

## A Single-Keystroke Program Generator

A fairly common thing to do in Logo is to write a little program that lets you type a single character on the keyboard to carry out some instruction. For example, teachers of very young children sometimes use a program that accepts `F` to move the turtle forward some distance, `B` for back, and `L` and `R` for left and right. What I want to write is a *program-writing program* that will accept a name and a list of keystrokes and instructions as inputs and define a procedure with that name that understands those instructions.

```
to onekey :name :list
local "text
make "text [[] [local "char] [print [Type ? for help]]
            [make "char readchar]]
foreach :list [make "text lput (sentence [if equalp :char]
                                         (word "" first ?)
                                         butfirst ?)
                                  :text]
make "text lput (lput (list "foreach :list ""print)
                      [if equalp :char "?]) ~
                :text
make "text lput (list :name) :text
define :name :text
end
```

If we use this program with the instruction

```
onekey "instant [[F [forward 20]] [B [back 20]]
                 [L [left 15]] [R [right 15]]]
```

then it creates the following procedure:

```
to instant
local "char
print [type ? for help]
make "char readchar
if equalp :char "F [forward 20]
if equalp :char "B [back 20]
if equalp :char "L [left 15]
if equalp :char "R [right 15]
if equalp :char "? [foreach [[F [forward 20]] [B [back 20]]
                             [L [left 15]] [R [right 15]]]
                            "print]
instant
end
```

In addition to illustrating the use of `define`, this program demonstrates how `sentence`, `list`, and `lput` can all be useful in constructing lists, when you have to combine some constant members with some variable members.

Of course, if we only want to make one `instant` program, it's easier just to type it in. An automatic procedure like `onekey` is useful when you want to create several different procedures like `instant`, each with a different "menu" of characters. For example, consider these instructions:

```
onekey "instant [[F [forward 20]] [B [back 20]]
                 [L [left 15]] [R [right 15]] [P [pens]]]
onekey "pens [[U [penup stop]] [D [pendown stop]] [E [penerase stop]]]
```

With these definitions, typing `P` to `instant` prepares to accept a pen command from the second list. In effect, `instant` recognizes two-letter commands `PU` for `penup` and so on, except that the sequence `P?` will display the help information for just the pen commands. Here's another example:

```
onekey "tinyturns [[F [forward 20]] [B [back 20]]
                   [L [left 5]] [R [right 5]] [H [hugeturns]]]
onekey "hugeturns [[F [forward 20]] [B [back 20]]
                   [L [left 45]] [R [right 45]] [T [tinyturns]]]
```

## Procedure Cross-Reference Listings

When you're working on a very large project, it's easy to lose track of which procedure invokes which other one. We can use the computer to help solve this problem by creating a *cross-reference listing* for all the procedures in a project. For every procedure in the project, a cross-reference listing tells which other procedures invoke that one. If you write long procedures, it can also be helpful to list which instruction line in procedure `A` invokes procedure `B`.

The general strategy will be to look through the `text` of every procedure, looking for the name of the procedure we're interested in. Suppose we're finding all the references to procedure `X` and we're looking through procedures `A`, `B`, and `C`. For each line of each procedure, we want to know whether the word `X` appears in that line. (Of course you would not really name a procedure `A` or `X`. You'd use meaningful names. This is just an example.) We can't, however, just test

```
memberp "x :instr
```

(I'm imagining that the variable `instr` contains an instruction line.) The reason is that a procedure invocation can be part of a *sublist* of the instruction list if `X` is invoked by way of something like `if`. For example, the word `X` is not a member of the list

```
[if emptyp :list [x :foo stop]]
```

But it's a member of a member. (Earlier I made a big fuss about the fact that if that instruction were part of procedure `A`, it's actually `if` that invokes `X`, not `A`. That's the true story, for the Logo interpreter. But for purposes of a cross-reference listing, it does us no good to know that `if` invokes `X`; what we want to know is which procedure definition to look at if we want to find the instruction that uses `X`.)

So the first thing we need is a procedure `submemberp` that takes inputs like those of `memberp` but outputs `true` if the first input is a member of the second, or a member of a member, and so on.

```
to submemberp :thing :list
if emptyp :list [output "false]
if equalp :thing first :list [output "true]
if listp first :list ~
   [if submemberp :thing first :list [output "true]]
output submemberp :thing butfirst :list
end
```

Now we want a procedure that will take two words as input, both of which are the names of procedures, and will print a list of all the references to the first procedure in the text of the second.

```
to reference :target :examinee
ref1 :target :examinee butfirst text :examinee 1
end

to ref1 :target :examinee :instrs :linenum
if emptyp :instrs [stop]
if submemberp :target first :instrs ~
   [print sentence "|   | (word :examinee "\( :linenum "\) )]
ref1 :target :examinee butfirst :instrs :linenum+1
end
```

`Reference` uses `butfirst text :examinee` as the third input to `ref1` to avoid the list of inputs to the procedure we're examining. That's because one of those inputs might have the same name as the `target` procedure, and we'd get a false indication of success. (In the body of the definition of `:examinee`, any reference to a variable named `X` will not use the word `X` but rather the word `"X` or the word `:X`. You may find that statement confusing. When you type an *instruction* like

```
print "foo
```

the Logo evaluator interprets `"foo` as a request for the word `foo`, quoted (as opposed to evaluated). So `print` won't print a quotation mark. But if we look at the *list*

```
[print "foo]
```

then we are not, right now, evaluating it as a Logo instruction. The second member of that list is the word `"foo`, quote mark and all.)

We can still get "false hits," finding the word `X` (or whatever procedure name we're looking for) in an instruction list, but not being used as a procedure name:

```
print [w x y z]
```

But cases like that will be relatively rare compared to the cases of variables and procedures with the same name.

The reason I'm printing spaces before the information is that I'm working toward a listing that will look like this:

```
target1
   proca(3)
   procb(1)
   procc(4)
target2
   procb(3)
   procb(4)
```

This means that the procedure named `target1` is invoked in each of the procedures `proca`, `procb`, and `procc`; procedure `target2` is invoked by `procb` on two different instruction lines.

Okay, now we can find references to one specific procedure within the text of another specific procedure. Now we want to look for references to one procedure within *all* the procedures making up a project.

```
to xref :target :list
print :target
foreach :list [reference :target ?]
end
```

We're almost done. Now we want to apply `xref` to every procedure in the project. This involves another run through the list of projects:

```
to xrefall :list
foreach :list [xref ? :list]
end
```

To use this program to make a cross-reference listing of itself, you'd say

```
xrefall [xrefall xref reference ref1 submemberp]
```

To cross-reference all of the procedures in your workspace, you'd say

```
xrefall procedures
```

If you try this program on a project with a large number of procedures, you should expect it to take a *long* time. If there are five procedures, we have to examine each of them for references to each of them, so we invoke `reference` 25 times. If there are 10 procedures, we invoke `reference` 100 times! In general, the number of invocations is the square of the number of procedures. The fancy way to say this is that the program "takes quadratic time" or that it "behaves quadratically."

# 6 Example: BASIC Compiler

Program file for this chapter: `basic`

The BASIC programming language was designed by John Kemeny and Thomas Kurtz in the late 1960s. (The name is an acronym for Beginner's All-purpose Symbolic Instruction Code.) It was first implemented on a large, central computer facility at Dartmouth; the designers' goal was to have a language that all students could use for simple problems, in contrast to the arcane programming languages used by most experts at that time.

A decade later, when the microcomputer was invented, BASIC took on a new importance. Kemeny and Kurtz designed a simple language for the sake of the users, but that simplicity also made the language easy for the *computer!* Every programming language requires a computer program to translate it into instructions that the computer can carry out. For example, the Logo programs you write are translated by a Logo interpreter. But Logo is a relatively complex language, and a Logo interpreter is a pretty big program. The first microcomputers had only a few thousand bytes of memory. (Today's home computers, by contrast, have several million bytes.) Those early personal computers couldn't handle Logo, but it was possible to write a BASIC interpreter that would fit them. As a result, BASIC became the near-universal language for amateur computer enthusiasts in the late 1970s and early 1980s.

Today's personal computers come with translators for a wide variety of programming languages, and also with software packages that enable many people to accomplish their computing tasks without writing programs of their own at all. BASIC is much less widely used today, although it has served as the core for Microsoft's "Visual Basic" language.

In this chapter, I want to show how Logo's `define` command can be used in a program-writing program. My program will translate BASIC programs into Logo programs. I chose BASIC for the same reason the early microcomputers used it: It's a small language and the translator is relatively easy to write. (Kemeny and Kurtz, the designers of BASIC, have criticized the microcomputer implementations as *too* simple

and as unfaithful to their original goals. My implementation will share that defect, to make the project easier. Don't use this version as a basis on which to judge the language! For that you should investigate True Basic, the version that Kemeny and Kurtz wrote themselves for personal computers.)

Here's a typical short BASIC program:

```
10 print "Table of Squares"
20 print
30 print "How many values would you like?"
40 input num
50 for i=1 to num
60 print i, i*i
70 next i
80 end
```

And here's what happens when we run it:

```
Table of Squares

How many values would you like?
5
1       1
2       4
3       9
4       16
5       25
```

## A Short Course in BASIC

Each line in the sample BASIC program begins with a *line number*. These numbers are used for program editing. Instead of the modern screen editors with which you're familiar, the early versions of BASIC had a very primitive editing facility; you could replace a line by typing a new line with the same number. There was no way to replace less than an entire line. To delete a line completely, you'd enter a line containing just the number. The reason the line numbers in this program are multiples of ten is to leave room for inserting new lines. For example, I could say

```
75 print "Have a nice day."
```

to insert a new line between lines 70 and 80. (By the way, the earliest versions of Logo used a similar line numbering system, except that each Logo procedure was separately

numbered. The editing technique isn't really part of the language design; early systems used "line editors" because they had typewriter-like paper terminals instead of today's display screens. I'm using a line editor in this project because it's easy to implement!)

The BASIC language consists of one or two dozen commands, depending on the version used. My BASIC dialect understands only these ten commands:

```
LET variable = value
PRINT values
INPUT variables
FOR variable = value TO value
NEXT variable
IF value THEN command
GOTO linenumber
GOSUB linenumber
RETURN
END
```

Unlike Logo procedure calls, which consist of the procedure name followed by inputs in a uniform format, each BASIC command has its own format, sometimes including internal separators such as the equal sign and the word `to` in the `for` command format, or the word `then` in the `if` command format.

In some versions of BASIC, including this one, a single line can contain more than one command, if the commands are separated with colons. Thus the same program shown earlier could also be written this way:

```
10 print "Table of Squares":print
30 print "How many values would you like?":input num
50 for i=1 to num : print i, i*i : next i
80 end
```

The `let` command assigns a value to a variable, like Logo's `make` procedure. Unlike Logo, BASIC does not have the rule that all inputs are evaluated before applying the command. In particular, the word after `let` must be the name of the variable, not an expression whose value is the name. Therefore the name is not quoted. Also, a variable can't have the same name as a procedure, so there is no need for anything like Logo's use of the colon to indicate a variable value. (This restricted version of BASIC doesn't have named procedures at all, like some early microcomputer versions.)

```
make "x :y + 3        (Logo)
let x = y + 3         (BASIC)
```

In my subset of BASIC, the value of a variable must be a number. More complete BASIC dialects include string variables (like words in Logo) and arrays (like Logo's arrays).

The value to be assigned to a variable can be computed using an arithmetic expression made up of variables, numbers, the arithmetic operators `+`, `-`, `*`, and `/`, and parentheses for grouping.

The `print` command is similar to Logo's print procedure in that it prints a line on the screen. That line can include any number of values. Here is an example `print` command:

```
print "x = "; x, "y = "; y, "sum = "; x+y
```

In this example two kinds of values are printed: arithmetic values (as in the `let` command) and strings. A *string* is any sequence of characters surrounded by quotation marks.

Notice that the values in this example are separated by punctuation marks, either commas or semicolons. When a semicolon is used, the two values are printed right next to each other, with no space between them. (That's why each of the strings in this example ends with a space.) When a comma is used, BASIC prints a tab character between the two values, so that values on different lines will line up to form columns. (Look again at the table of squares example at the beginning of this chapter.)

The `input` command is the opposite of `print`; it reads values from the keyboard and assigns them to variables. There is nothing in Logo exactly like `input`. Instead, Logo has *operations* `readword` and `readlist` that output the contents of a line; those values can be assigned to variables using `make` or can be used in some other way. The Logo approach is more flexible, but the early versions of BASIC didn't have anything like Logo's operations. The `input` command will also accept a string in quotation marks before its list of variables; that string is printed as a prompt before BASIC reads from the keyboard. (BASIC does not start a new line after printing the prompt, so the effect is like Logo's `type` command rather than like `print`.) Here's an example:

```
input "Please enter x and y: " x,y
```

The user can type the values for x and y on the same line, separated by spaces, or on separate lines. BASIC keeps reading lines until it has collected enough numbers for the listed variables. Notice that the variable names in the `input` command must be separated by commas, not by semicolons.

The `for` and `next` commands work together to provide a numeric iteration capability like Berkeley Logo's `for` procedure. The `for` command format includes a

variable name, a starting value, and an ending value. (The step value is always 1.) The named variable is given the specified starting value. If that value is less than the ending value, then all of the commands between the `for` command and the matching `next` command (the one with the same named variable) are carried out. Then the variable is increased by 1, and the process continues until the ending value is reached. `For` and `next` pairs with different variables can be nested:

```
10 input "Input size: " num
20 for i = 1 to num
30 for j = i to num
40 print i;"  ";j
50 next j:next i
60 end

Input size: 4
1  1
1  2
1  3
1  4
2  2
2  3
2  4
3  3
3  4
4  4
```

Notice that the `next j` must come before the `next i` so that the `for/next` pairs are properly nested.

The `if` command allows conditional execution, much like Logo's `if` command, but with a different notation. Instead of taking an instruction list as an input, BASIC's `if` uses the keyword `then` to introduce a single conditional command. (If you want to make more than one command conditional, you must combine `if` with `goto`, described next.) The value that controls the `if` must be computed using one of the operators `=`, `<`, or `>` for numeric comparison.*

* Notice that the equal sign has two meanings in BASIC. In the `let` command, it's like Logo's `make`; in the `if` command, it's like Logo's `equalp`. In the early 1980s, Logo enthusiasts had fierce arguments with BASIC fans, and this sort of notational inconsistency was one of the things that drove us crazy! (More serious concerns were the lack of operations and of recursion in the microcomputer versions of BASIC.)

The `goto` command transfers control to the beginning of a command line specified by its line number. It can be used with `if` to make a sequence of commands conditional:

```
10 input x
20 if x > 0 then goto 100
30 print "x is negative."
40 print "x = "; x
50 goto 200
100 print "x is positive."
200 end
```

The `gosub` and `return` commands provide a rudimentary procedure calling mechanism. I call it "rudimentary" because the procedures have no inputs, and can only be commands, not operations. Also, the command lines that make up the procedure are also part of the main program, so you generally need a `goto` in the main program to skip over them:

```
10 let x=7
20 gosub 100
30 let x=9
40 gosub 100
50 goto 200
100 print x, x*x
110 return
200 end
```

Finally, the `end` command ends the program. There must be an `end` at the end of a BASIC program, and there should not be one anywhere else. (In this implementation of BASIC, an `end` stops the BASIC program even if there are more lines after it. It's roughly equivalent to a `throw` to `toplevel` in Logo.)

## Using the BASIC Translator

To start the translator, run the Logo procedure `basic` with no inputs. You will then see the BASIC prompt, which is the word `READY` on a line by itself.

At the prompt you can do either of two things. If you type a line starting with a line number, that line will be entered into your BASIC program. It is inserted in order by line number. Any previous line with the same number will be deleted. If the line you type contains *only* a line number, then the line in the program with that number will be deleted.

If your line does not start with a number, then it is taken as an *immediate* command, not as part of the program. This version of BASIC recognizes only three immediate commands: The word `run` means to run your program, starting from the smallest line number. The word `list` means to print out a listing of the program's lines, in numeric order. The word `exit` returns to the Logo prompt.

## Overview of the Implementation

There are two kinds of translators for programming languages: compilers and interpreters. The difference is that a compiler translates one language (the *source* language) into another (the *target* language), leaving the result around so that it can be run repeatedly without being translated again. An interpreter translates each little piece of source language into one action in the target language and runs the result, but does not preserve a complete translated program in the target language.

Ordinarily, the target language for both compilers and interpreters is the "native" language of the particular computer you're using, the language that is wired into the computer hardware. This *machine language* is the only form in which a program can actually be run. The BASIC compiler in this chapter is quite unrealistic in that it uses Logo as the target language, which means that the program must go through *another* translation, from Logo to machine language, before it can actually be run. For our purposes, there are three advantages to using Logo as the target language. First, every kind of computer has its own machine language, so I'd have to write several versions of the compiler to satisfy everyone if I compiled BASIC into machine language. Second, I know you know Logo, so you can understand the resulting program, whereas you might not be familiar with any machine language. Third, this approach allows me to cheat by leaving out a lot of the complexity of a real compiler. Logo is a "high level" language, which means that it takes care of many details for us, such as the allocation of specific locations in the computer's memory to hold each piece of information used by the program. In order to compile into machine language, I'd have to pay attention to those details.

Why would anyone want an interpreter, if the compiler translates the program once and for all, while the interpreter requires retranslation every time a command is carried out? One reason is that an interpreter is easier to write, because (just as in the case of a compiler with Logo as the target language) many of the details can be left out. Another reason is that traditional compilers work using a *batch* method, which means that you must first write the entire program with a text editor, then run the compiler to translate the program into machine language, and finally run the program. This is okay

for a working program that is used often, but not recompiled often. But when you're creating a program in the first place, there is a debugging process that requires frequent modifications to the source language program. If each modification requires a complete recompilation, the debugging is slow and frustrating. That's why interpreted languages are often used for teaching—when you're learning to program, you spend much more time debugging a program than running the final version.

The best of both worlds is an *incremental compiler,* a compiler that can recompile only the changed part when a small change is made to a large program. For example, Object Logo is a commercial version of Logo for the Macintosh in which each procedure is compiled when it is defined. Modifying a procedure requires recompiling that procedure, but not recompiling the others. Object Logo behaves like an interpreter, because the user doesn't have to ask explicitly for a procedure to be compiled, but programs run faster in Object Logo than in most other versions because each procedure is translated only once, rather than on every invocation.

The BASIC translator in this chapter is an incremental compiler. Each numbered line is compiled into a Logo procedure as soon as it is typed in. If the line number is `40` then the resulting procedure will be named `basic%40`. The last step in each of these procedures is to invoke the procedure for the next line. The compiler maintains a list of all the currently existing line numbers, in order, so the `run` command is implemented by saying

```
run (list (word "basic% first :linenumbers))
```

Actually, what I just said about each procedure ending with an invocation of the next one is slightly simplified. Suppose the BASIC program starts

```
10 let x=3
20 let y=9
30 ...
```

and we translate that into

```
to basic%10
make "%x 3
basic%20
end
```

```
to basic%20
make "%y 9
basic%30
end
```

Then what happens if the user adds a new line numbered 15? We would have to recompile line 10 to invoke `basic%15` instead of `basic%20`. To avoid that, each line is compiled in a way that defers the choice of the next line until the program is actually run:

```
to basic%10
make "%x 3
nextline 10
end

to basic%20
make "%y 9
nextline 20
end
```

This solution depends on a procedure `nextline` that finds the next available line number after its argument:

```
to nextline :num
make "target member :num :linenumbers
if not emptyp :target [make "target butfirst :target]
if not emptyp :target [run (list (word "basic% first :target))]
end
```

`Nextline` uses the Berkeley Logo primitive `member`, which is like the predicate `memberp` except that if the first input is found as a member of the second, instead of giving `true` as its output, it gives the portion of the second input starting with the first input:

```
? show member "the [when in the course of human events]
[the course of human events]
```

If the first input is not a member of the second, `member` outputs an empty word or list, depending on the type of the second input.

The two separate `emptyp` tests are used instead of a single `if` because the desired line number might not be in the list at all, or it might be the last one in the list, in which case the `butfirst` invocation will output an empty list. (Neither of these cases should arise. The first means that we're running a line that doesn't exist, and the second means

that the BASIC program doesn't end with an `end` line. But the procedure tries to avoid disaster even in these cases.)

Look again at the definition of `basic%10`. You'll see that the variable named `x` in the BASIC program is named `%x` in the Logo translation. The compiler uses this renaming technique to ensure that the names of variables and procedures in the compiled program don't conflict with names used in the compiler itself. For example, the compiler uses a variable named `linenumbers` whose value is the list of line numbers. What if someone writes a BASIC program that says

```
10 let linenumbers = 100
```

This won't be a problem because in the Logo translation, that variable will be named `%linenumbers`.

The compiler can be divided conceptually into four parts:

- The *reader* divides the characters that the user types into meaningful units. For example, it recognizes that `let` is a single word, but `x+1` should be understood as three separate words.
- The *parser* recognizes the form of each of the ten BASIC commands that this dialect understands. For example, if a command starts with `if`, the parser expects an expression followed by the word `then` and another command.
- The *code generator* constructs the actual translation of each BASIC command into one or more Logo instructions.
- The *runtime library* contains procedures that are used while the translated program is running, rather than during the compilation process. The `nextline` procedure discussed earlier is an example.

Real compilers have the same structure, except of course that the code generator produces machine language instructions rather than Logo instructions. Also, a professional compiler will include an *optimizer* that looks for ways to make the compiled program as efficient as possible.

## The Reader

A *reader* is a program that reads a bunch of characters (typically one line, although not in every language) and divides those characters into meaningful units. For example, every

Logo implementation includes a reader that interprets square brackets as indications of list grouping. But some of the rules followed by the Logo reader differ among implementations. For example, can the hyphen character (`-`) be part of a larger word, or is it always a word by itself? In a context in which it means subtraction, we'd like it to be a word by itself. For example, when you say

```
print :x-3
```

as a Logo instruction, you mean to print three less than the value of the variable named `x`, not to print the value of a variable whose name is the three-letter word `x-3`! On the other hand, if you have a list of telephone numbers like this:

```
make "phones [555-2368 555-9827 555-8311]
```

you'd like the `first` of that list to be an entire phone number, the word `555-2368`, not just `555`. Some Logo implementations treat every hyphen as a word by itself; some treat every hyphen just like a letter, and require that you put spaces around a minus sign if you mean subtraction. Other implementations, including Berkeley Logo, use a more complicated rule in which the status of the hyphen depends on the context in which it appears, so that both of the examples in this paragraph work as desired.

In any case, Logo's reader follows rules that are not appropriate for BASIC. For example, the colon (`:`) is a delimiter in BASIC, so it should be treated as a word by itself; in Logo, the colon is paired with the variable name that follows it. In both languages, the quotation mark (`"`) is used to mark quoted text, but in Logo it comes only at the beginning of a word, and the quoted text ends at the next space character, whereas in BASIC the quoted text continues until a second, matching quotation mark. For these and other reasons, it's desirable to have a BASIC-specific reader for use in this project.

The rules of the BASIC reader are pretty simple. Each invocation of `basicread` reads one line from the keyboard, ending with the Return or Enter character. Within that line, space characters separate words but are not part of any word. A quotation mark begins a quoted word that includes everything up to and including the next matching quotation mark. Certain characters form words by themselves:

```
+ - * / = < > ( ) , ; :
```

All other characters are treated like letters; that is, they can be part of multi-character words.

```
? show basicread
30 print x;y;"foo,baz",z:print hello+4
[30 print x ; y ; "foo,baz" , z : print hello + 4]
```

Notice that the comma inside the quotation marks is not made into a separate word by `basicread`. The other punctuation characters, however, appear in the output sentence as one-character words.

`Basicread` uses the Logo primitive `readword` to read a line. `Readword` can be thought of as a reader with one trivial rule: The only special character is the one that ends a line. Everything else is considered as part of a single long word. `Basicread` examines that long word character by character, looking for delimiters, and accumulating a sentence of words separated according to the BASIC rules. The implementation of `basicread` is straightforward; you can read the procedures at the end of this chapter if you're interested. For now, I'll just take it for granted and go on to discuss the more interesting parts of the BASIC compiler.

## The Parser

The *parser* is the part of a compiler that figures out the structure of each piece of the source program. For example, if the BASIC compiler sees the command

```
let x = ( 3 * y ) + 7
```

it must recognize that this is a `let` command, which must follow the pattern

```
LET variable = value
```

and therefore `x` must be the name of a variable, while `( 3 * y ) + 7` must be an expression representing a value. The expression must be further parsed into its component pieces. Both the variable name and the expression must be translated into the form they will take in the compiled (Logo) program, but that's the job of the code generator.

In practice, the parser and the code generator are combined into one step; as each piece of the source program is recognized, it is translated into a corresponding piece of the object program. So we'll see that most of the procedures in the BASIC compiler include parsing instructions and code generation instructions. For example, here is the procedure that compiles a `let` command:

```
to compile.let :command
make "command butfirst :command
make "var pop "command
make "delimiter pop "command
if not equalp :delimiter "= [(throw "error [Need = in let.])]
make "exp expression
queue "definition (sentence "make (word ""% :var) :exp)
end
```

In this procedure, all but the last instruction (the line starting with `queue`) are parsing the source command. The last line, which we'll come back to later, is generating a Logo `make` instruction, the translation of the BASIC `let` in the object program.

BASIC was designed to be very easy to parse. The parser can read a command from left to right, one word at a time; at every moment, it knows exactly what to expect. The command must begin with one of the small number of command names that make up the BASIC language. What comes next depends on that command name; in the case of `let`, what comes next is one word (the variable name), then an equal sign, then an expression. Each instruction in the `compile.let` procedure handles one of these pieces. First we skip over the word `let` by removing it from the front of the command:

```
make "command butfirst :command
```

Then we read and remember one word, the variable name:

```
make "var pop "command
```

(Remember that the `pop` operation removes one member from the beginning of a list, returning that member. In this case we are removing the variable name from the entire `let` command.) Then we make sure there's an equal sign:

```
make "delimiter pop "command
if not equalp :delimiter "= [(throw "error [Need = in let.])]
```

And finally we call a subprocedure to read the expression; as we'll see later, that procedure also translates the expression to the form it will take in the object program:

```
make "exp expression
```

The parsers for other BASIC commands have essentially the same structure as this example. They repeatedly invoke `pop` to read one word from the command or `expression` to read and translate an expression. (The `if` command is a little more

complicated because it contains another command as a component, but that inner command is just compiled as if it occurred by itself. We'll look at that process in more detail when we get to the code generation part of the compiler.)

Each compilation procedure expects a single BASIC command as its input. Remember that a line in a BASIC program can include more than one command. The compiler uses a procedure named `split` to break up each line into a list of commands:

```
? show split [30 print x ; y ; "foo,baz" , z : print hello + 4]
[30 [print x ; y ; "foo,baz" , z] [print hello + 4]]
```

`Split` outputs a list whose first member is a line number; the remaining members are lists, each containing one BASIC command. `Split` works by looking for colons within the command line.

Here is the overall structure of the compiler, but with only the instructions related to parsing included:

```
to basic
forever [basicprompt]
end

to basicprompt
print "READY
make "line basicread
if emptyp :line [stop]
ifelse numberp first :line [compile split :line] [immediate :line]
end

to compile :commands
make "number first :commands
ifelse emptyp butfirst :commands ~
       [eraseline :number] ~
       [makedef (word "basic% :number) butfirst :commands]
end

to makedef :name :commands
...
foreach :commands [run list (word "compile. first ?) ?]
...
end
```

`Basic` does some initialization (not shown) and then invokes `basicprompt` repeatedly. `Basicprompt` calls the BASIC reader to read a line; if that line starts with a number, then `split` is used to transform the line into a list of commands, and `compile` is invoked with that list as input. `Compile` remembers the line number for later use, and then invokes `makedef` with the list of commands as an input. I've left out most of the instructions in `makedef` because they're concerned with code generation, but the important part right now is that for each command in the list, it invokes a procedure named `compile.`something based on the first word of the command, which must be one of the command names in the BASIC language.

## The Code Generator

Each line of the BASIC source program is going to be compiled into one Logo procedure. (We'll see shortly that the BASIC `if` and `for` commands are exceptions.) For example, the line

```
10 let x = 3 : let y = 4 : print x,y+6
```

will be compiled into the Logo procedure

```
to basic%10
make "%x 3
make "%y 4
type :%x
type char 9
type :%y + 6
print []
nextline 10
end
```

Each of the three BASIC commands within the source line contributes one or more instructions to the object procedure. Each `let` command is translated into a `make` instruction; the `print` command is translated into three `type` instructions and a `print` instruction. (The last instruction line in the procedure, the invocation of `nextline`, does not come from any of the BASIC commands, but is automatically part of the translation of every BASIC command line.)

To generate this object procedure, the BASIC compiler is going to have to invoke Logo's `define` primitive, this way:

```
define "basic%10 [[] [make "%x 3] [make "%y 4] ... [nextline 10]]
```

Of course, these actual inputs do not appear explicitly in the compiler! Rather, the inputs to `define` are variables that have the desired values:

```
define :name :definition
```

The variable `name` is an input to `makedef`, as we've seen earlier. The variable `definition` is created within `makedef`. It starts out as a list containing just the empty list, because the first sublist of the input to `define` is the list of the names of the desired inputs to `basic%10`, but it has no inputs. The procedures within the compiler that parse each of the commands on the source line will also generate object code (that is, Logo instructions) by appending those instructions to the value of `definition` using Logo's `queue` command. `Queue` takes two inputs: the name of a variable whose value is a list, and a new member to be added at the end of the list. Its effect is to change the value of the variable to be the extended list.

Look back at the definition of `compile.let` above. Earlier we considered the parsing instructions within that procedure, but deferred discussion of the last instruction:

```
queue "definition (sentence "make (word ""% :var) :exp)
```

Now we can understand what this does: It generates a Logo `make` instruction and appends that instruction to the object procedure definition in progress.

We can now also think about the output from the `expression` procedure. Its job is to parse a BASIC expression and to translate it into the corresponding Logo expression. This part of the compiler is one of the least realistic. A real compiler would have to think about such issues as the precedence of arithmetic operations; for example, an expression like `3+x*4` must be translated into two machine language instructions, first one that multiplies `x` by 4, and then one that adds the result of that multiplication to 3. But the Logo interpreter already handles that aspect of arithmetic for us, so all `expression` has to do is to translate variable references like `x` into the Logo form `:%x`.

```
? show expression [3 + x * 4]
[3 + :%x * 4]
```

(We'll take a closer look at translating arithmetic expressions in the Pascal compiler found in the third volume of this series, *Beyond Programming*.)

We are now ready to look at the complete version of `makedef`:

```
to makedef :name :commands
make "definition [[]]
foreach :commands [run list (word "compile. first ?) ?]
queue "definition (list "nextline :number)
define :name :definition
make "linenumbers insert :number :linenumbers
end
```

I hope you'll find this straightforward. First we create an empty definition. Then, for each BASIC command on the line, we append to that definition whatever instructions are generated by the code generating instructions for that command. After all the BASIC commands have been compiled, we add an invocation of `nextline` to the definition. Now we can actually define the Logo procedure whose text we've been accumulating. The last instruction updates the list of line numbers that `nextline` uses to find the next BASIC command line when the compiled program is running.

In a sense, this is the end of the story. My purpose in this chapter was to illustrate how `define` can be used in a significant project, and I've done that. But there are a few more points I should explain about the code generation for some specific BASIC commands, to complete your understanding of the compiler.

One such point is about the difference between `goto` and `gosub`. Logo doesn't have anything like a `goto` mechanism; both `goto` and `gosub` must be implemented by invoking the procedure corresponding to the given line number. The difference is that in the case of `goto`, we want to invoke that procedure and not come back! The solution is to compile the BASIC command

```
goto 40
```

into the Logo instructions

```
basic%40 stop
```

In effect, we are calling line 40 as a subprocedure, but when it returns, we're finished. Any additional Logo instructions generated for the same line after the `goto` (including the invocation of `nextline` that's generated automatically for every source line) will be ignored because of the `stop`.*

* In fact, the Berkeley Logo interpreter is clever enough to notice that there is a `stop` instruction after the invocation of `basic%40`, and it arranges things so that there is no "return" from that procedure. This makes things a little more efficient, but doesn't change the meaning of the program.

The next tricky part of the compiler has to do with the `for` and `next` commands. Think first about `next`. It must increment the value of the given variable, test that value against a remembered limit, and, if the limit has not been reached, go to... where? The `for` loop continues with the BASIC command just after the `for` command itself. That might be in the middle of a line, so `next` can't just remember a line number and invoke `basic%N` for line number N. To solve this problem, the line containing the `for` command is split into two Logo procedures, one containing everything up to and including the `for`, and one for the rest of the line. For example, the line

```
30 let x = 3 : for i = 1 to 5 : print i,x : next i
```

is translated into

```
to basic%30
make "%x 3
make "%i 1
make "let%i 5
make "next%i [%g1]
%g1
end

to %g1
type :%i
type char 9
type :%x
print []
make "%i :%i + 1
if not greaterp :%i :let%i [run :next%i stop]
nextline 30
end
```

The first `make` instruction in `basic%30` is the translation of the `let` command. The remaining four lines are the translation of the `for` command; it must give an initial value to the variable `i`, remember the limit value 5, and remember the Logo procedure to be used for looping. That latter procedure is named `%g1` in this example. The percent sign is used for the usual reason, to ensure that the names created by the compiler don't conflict with names in the compiler itself. The `g1` part is a *generated symbol*, created by invoking the Berkeley Logo primitive operation `gensym`. Each invocation of `gensym` outputs a new symbol, first `g1`, then `g2`, and so on.

The first four instructions in procedure `%g1` (three `type`s and a `print`) are the translation of the BASIC `print` command. The next two instructions are the translation

of the `next` command; the `make` instruction increments `i`, and the `if` instruction tests whether the limit has been passed, and if not, invokes the looping procedure `%g1` again. (Why does this say `run :next%i` instead of just `%g1`? Remember that the name `%g1` was created during the compilation of the `for` command. When we get around to compiling the `next` command, the code generator has no way to remember which generated symbol was used by the corresponding `for`. Instead it makes reference to a variable `next%i`, named after the variable given in the `next` command itself, whose value is the name of the procedure to run. Why not just call that procedure itself `next%i` instead of using a generated symbol? The trouble is that there might be more than one pair of `for` and `next` commands in the same BASIC program using the same variable, and each of them must have its own looping procedure name.)

There is a slight complication in the `print` and `input` commands to deal with quoted character strings. The trouble is that Logo's idea of a word ends with a space, so it's not easy to translate

```
20 print "hi there"
```

into a Logo instruction in which the string is explicitly present in the instruction. Instead, the BASIC compiler creates a Logo global variable with a generated name, and uses that variable in the compiled Logo instructions.

The trickiest compilation problem comes from the `if` command, because it includes another command as part of itself. That included command might be translated into several Logo instructions, all of which should be made to depend on the condition that the `if` is testing. The solution is to put the translation of the inner command into a separate procedure, so that the BASIC command line

```
50 if x<6 then print x, x*x
```

is translated into the two Logo procedures

```
to basic%50
if :%x < 6 [%g2]
nextline 50
end

to %g2
type :%x
type char 9
type :%x * :%x
print []
end
```

Unfortunately, this doesn't quite work if the inner command is a `goto`. If we were to translate

```
60 if :foo < 10 then goto 200
```

into

```
to basic%60
if :%foo < 10 [%g3]
nextline 60
end

to %g3
basic%200 stop
end
```

then the `stop` inside `%g3` would stop only `%g3` itself, not `basic%60` as desired. So the code generator for `if` checks to see whether the result of compiling the inner command is a single Logo instruction line; if so, that line is used directly in the compiled Logo `if` rather than diverted into a subprocedure:

```
to basic%60
if :%foo < 10 [basic%200 stop]
nextline 60
end
```

How does the code generator for `if` divert the result of compiling the inner command away from the definition of the overall BASIC command line? Here is the relevant part of the compiler:

```
to compile.if :command
make "command butfirst :command
make "exp expression
make "delimiter pop "command
if not equalp :delimiter "then [(throw "error [Need then after if.])]
queue "definition (sentence "if :exp (list c.if1))
end
```

```
to c.if1
local "definition
make "definition [[]]
run list (word "compile. first :command) :command
ifelse (count :definition) = 2 ~
       [output last :definition] ~
       [make "newname word "% gensym
        define :newname :definition
        output (list :newname)]
end
```

The first few lines of this are straightforwardly parsing the part of the BASIC `if` command up to the word `then`. What happens next is a little tricky; a subprocedure `c.if1` is invoked to parse and translate the inner command. It has to be a subprocedure because it creates a local variable named `definition`; when the inner command is compiled, this local variable "steals" the generated code. If there is only one line of generated code, then `c.if1` outputs that line; if more than one, then `c.if1` creates a subprocedure and outputs an instruction to invoke that subprocedure. This technique depends on Logo's dynamic scope, so that references to the variable named `definition` in other parts of the compiler (such as, for example, `compile.print` or `compile.goto`) will refer to this local version.

## The Runtime Library

We've already seen the most important part of the runtime library: the procedure `nextline` that gets the compiled program from one line to the next.

There is only one more procedure needed as runtime support; it's called `readvalue` and it's used by the BASIC `input` command. In `BASIC`, data input is independent of lines. If a single `input` command includes two variables, the user can type the two desired values on separate lines or on a single line. Furthermore, two *separate* `input` commands can read values from a single line, if there are still values left on the line after the first `input` has been satisfied. `Readvalue` uses a global variable `readline` whose value is whatever's still available from the last data input line, if any. If there is nothing available, it reads a new line of input.

A more realistic BASIC implementation would include runtime library procedures to compute built-in functions (the equivalent to Logo's primitive operations) such as absolute value or the trigonometric functions.

## Further Explorations

This BASIC compiler leaves out many features of a complete implementation. In a real BASIC, a string can be the value of a variable, and there are string operations such as concatenation and substring extraction analogous to the arithmetic operations for numbers. The BASIC programmer can create an array of numbers, or an array of strings. In some versions of BASIC, the programmer can define named subprocedures, just as in Logo. For the purposes of this chapter, I wanted to make the compiler as simple as possible and still have a usable language. If you want to extend the compiler, get a BASIC textbook and start implementing features.

It's also possible to expand the immediate command capabilities of the compiler. In most BASIC implementations, for example, you can say `list 100-200` to list only a specified range of lines within the source program.

A much harder project would be to replace the code generator in this compiler with one that generates machine language for your computer. Instead of using `define` to create Logo procedures, your compiler would then write machine language instructions into a data file. To do this, you must learn quite a lot about how machine language programs are run on your computer!

## Program Listing

I haven't discussed every detail of the program. For example, you may want to trace through what happens when you ask to delete a line from the BASIC source program. Here is the complete compiler.

```
to basic
make "linenumbers []
make "readline []
forever [basicprompt]
end

to basicprompt
print []
print "READY
print []
make "line basicread
if emptyp :line [stop]
ifelse numberp first :line [compile split :line] [immediate :line]
end
```

```
to compile :commands
make "number first :commands
make :number :line
ifelse emptyp butfirst :commands ~
       [eraseline :number] ~
       [makedef (word "basic% :number) butfirst :commands]
end

to makedef :name :commands
make "definition [[]]
foreach :commands [run list (word "compile. first ?) ?]
queue "definition (list "nextline :number)
define :name :definition
make "linenumbers insert :number :linenumbers
end

to insert :num :list
if emptyp :list [output (list :num)]
if :num = first :list [output :list]
if :num < first :list [output fput :num :list]
output fput first :list (insert :num butfirst :list)
end

to eraseline :num
make "linenumbers remove :num :linenumbers
end

to immediate :line
if equalp :line [list] [foreach :linenumbers [print thing ?] stop]
if equalp :line [run] [run (list (word "basic% first :linenumbers))
                       stop]
if equalp :line [exit] [throw "toplevel]
print sentence [Invalid command:] :line
end

;; Compiling each BASIC command

to compile.end :command
queue "definition [stop]
end

to compile.goto :command
queue "definition (list (word "basic% last :command) "stop)
end
```

```
to compile.gosub :command
queue "definition (list (word "basic% last :command))
end

to compile.return :command
queue "definition [stop]
end

to compile.print :command
make "command butfirst :command
while [not emptyp :command] [c.print1]
queue "definition [print []]
end

to c.print1
make "exp expression
ifelse equalp first first :exp "" ~
       [make "sym gensym
        make word "%% :sym butfirst butlast first :exp
        queue "definition list "type word ":%% :sym] ~
       [queue "definition fput "type :exp]
if emptyp :command [stop]
make "delimiter pop "command
if equalp :delimiter ", [queue "definition [type char 9] stop]
if equalp :delimiter "\; [stop]
(throw "error [Comma or semicolon needed in print.])
end

to compile.input :command
make "command butfirst :command
if equalp first first :command "" ~
   [make "sym gensym
    make "prompt pop "command
    make word "%% :sym butfirst butlast :prompt
    queue "definition list "type word ":%% :sym]
while [not emptyp :command] [c.input1]
end

to c.input1
make "var pop "command
queue "definition (list "make (word ""% :var) "readvalue)
if emptyp :command [stop]
make "delimiter pop "command
if not equalp :delimiter ", (throw "error [Comma needed in input.])
end
```

```
to compile.let :command
make "command butfirst :command
make "var pop "command
make "delimiter pop "command
if not equalp :delimiter "= [(throw "error [Need = in let.])]
make "exp expression
queue "definition (sentence "make (word ""% :var) :exp)
end

to compile.for :command
make "command butfirst :command
make "var pop "command
make "delimiter pop "command
if not equalp :delimiter "= [(throw "error [Need = after for.])]
make "start expression
make "delimiter pop "command
if not equalp :delimiter "to [(throw "error [Need to after for.])]
make "end expression
queue "definition (sentence "make (word ""% :var) :start)
queue "definition (sentence "make (word ""let% :var) :end)
make "newname word "% gensym
queue "definition (sentence "make (word ""next% :var)
                                  (list (list :newname)))
queue "definition (list :newname)
define :name :definition
make "name :newname
make "definition [[]]
end

to compile.next :command
make "command butfirst :command
make "var pop "command
queue "definition (sentence "make (word ""% :var) (word ":% :var) [+ 1])
queue "definition (sentence [if not greaterp]
                            (word ":% :var) (word ":let% :var)
                            (list (list "run (word ":next% :var)
                                        "stop)))
end
```

```
to compile.if :command
make "command butfirst :command
make "exp expression
make "delimiter pop "command
if not equalp :delimiter "then [(throw "error [Need then after if.])]
queue "definition (sentence "if :exp (list c.if1))
end

to c.if1
local "definition
make "definition [[]]
run list (word "compile. first :command) :command
ifelse (count :definition) = 2 ~
       [output last :definition] ~
       [make "newname word "% gensym
        define :newname :definition
        output (list :newname)]
end

;; Compile an expression for LET, IF, PRINT, or FOR

to expression
make "expr []
make "token expr1
while [not emptyp :token] [queue "expr :token
                           make "token expr1]
output :expr
end

to expr1
if emptyp :command [output []]
make "token pop "command
if memberp :token [+ - * / = < > ( )] [output :token]
if memberp :token [, \; : then to] [push "command :token  output []]
if numberp :token [output :token]
if equalp first :token "" [output :token]
output word ":% :token
end
```

```
;; reading input

to basicread
output basicread1 readword [] "
end

to basicread1 :input :output :token
if emptyp :input [if not emptyp :token [push "output :token]
                  output reverse :output]
if equalp first :input "| | [if not emptyp :token [push "output :token]
                             output basicread1 (butfirst :input)
                                               :output "]
if equalp first :input "" [if not emptyp :token [push "output :token]
                             output breadstring butfirst :input
                                                 :output "]
if memberp first :input [+ - * / = < > ( ) , \; :] ~
   [if not emptyp :token [push "output :token]
    output basicread1 (butfirst :input) (fput first :input :output) "]
output basicread1 (butfirst :input) :output (word :token first :input)
end

to breadstring :input :output :string
if emptyp :input [(throw "error [String needs ending quote.])]
if equalp first :input "" ~
   [output basicread1 (butfirst :input)
                      (fput (word "" :string "") :output)
                      "]
output breadstring (butfirst :input) :output (word :string first :input)
end

to split :line
output fput first :line split1 (butfirst :line) [] []
end

to split1 :input :output :command
if emptyp :input [if not emptyp :command [push "output reverse :command]
                  output reverse :output]
if equalp first :input ": [if not emptyp :command
                              [push "output reverse :command]
                           output split1 (butfirst :input) :output []]
output split1 (butfirst :input) :output (fput first :input :command)
end
```

```
;; Runtime library

to nextline :num
make "target member :num :linenumbers
if not emptyp :target [make "target butfirst :target]
if not emptyp :target [run (list (word "basic% first :target))]
end

to readvalue
while [emptyp :readline] [make "readline basicread]
output pop "readline
end
```

# 7 Pattern Matcher

Program file for this chapter: `match`

In a *conversational* program, one that carries on a conversation with the user, you may often have occasion to compare what the user types with some expected response. For example, a quiz program will compare the user's response with the correct answer; if you've just asked "how are you," you might look for words like "fine" or "lousy" in the reply. The main tools that Logo provides for such comparisons are `equalp`, which compares two values for exact equality, and `memberp`, which compares one datum with a list of alternatives. This project provides a more advanced comparison tool.

Most of the projects in this book are fairly complicated in their inner workings, but relatively simple in the external appearance of what they do. This project is the reverse; the actual program is not so complex, but it does quite a lot, and it will take a while to explain all of it. Pattern matching is a powerful programming tool, and I hope you won't be put off by the effort required to learn how to use it.

A *pattern* is a list in which some members are not made explicit. This definition is best understood by considering an example. Consider the pattern

```
[Every # is a #]
```

The words `every`, `is`, and `a` represent themselves explicitly. The two number signs, however, are symbols representing "zero or more arbitrary data." Here are some lists that would match the pattern:

```
[Every man is a mortal]
[Every computer programmer is a genius]
[Every is a word]
[Every datum is a word or a list]
```

Here are some lists that would *not* match the pattern:

```
[Socrates is a man]
[Every man is an animal]
[Everyone I know is a friend]
[I think every list is a match]
```

The first of these examples doesn't match the pattern because the word `every` is missing. The second has `an` instead of `a`, while the third has `everyone` instead of `every`. The fourth has the extra words `I think` before the word `every`. This last example *would* match the pattern

```
[# every # is a #]
```

because this new pattern allows for extra words at the beginning.

`Match` is a predicate that takes two inputs. The first input is a pattern and the second input is a sentence. The output is `true` if the sentence matches the pattern, otherwise `false`.

```
? print match [Every # is a #] [Every book is a joy to read]
true
? print match [Every # is a #] [Every adolescent is obnoxious]
false
```

Patterns can be more complicated than the ones I've shown so far. In the following paragraphs I'll introduce the many ways that you can modify patterns to control which sentences they'll match. As you read, you should make up sample patterns of your own and try them out with `match`.

Often, in a conversational program, it's not good enough just to know whether or not a sentence matches a pattern. You also want to know the pieces of the sentence that match the variable parts of the pattern. `Match` meets this requirement by allowing you to tell it the names of variables that you want to receive the matching words from the sentence. Here is an example:

```
? print match [#food is for #animal] [Hay is for horses]
true
? show :food
[Hay]
? show :animal
[horses]
```

```
? print match [#food is for #animal] [C++ is for the birds]
true
? show :food
[C++]
? show :animal
[the birds]
```

Here is a short conversational program using the parts of the pattern matcher we've discussed so far.

```
to converse
local [response name like]
print [Hi, my name is Toby and I like ice cream]
print [Tell me about yourself]
make "response readlist
if match [# my name is #name] :response [do.name strip.and :name]
if match [# i like #like] :response [do.like strip.and :like]
print [Nice meeting you!]
end

to do.name :name
print sentence "Hello, :name
end

to do.like :like
print sentence [I'm glad you like] :like
end

to strip.and :text
local "short
if match [#short and #] :text [output :short]
output :text
end

? converse
Hi, my name is Toby and I like ice cream
Tell me about yourself
My name is Brian and I like Chinese food
Hello, Brian
I'm glad you like Chinese food
Nice meeting you!
```

```
? converse
Hi, my name is Toby and I like ice cream
Tell me about yourself
I like spaghetti and meat balls
I'm glad you like spaghetti
Nice meeting you!
```

If `match` outputs `false`, there is no guarantee of what values will end up in the variables mentioned in the pattern. `Converse` uses the result of the match only if `match` outputs `true`.

`Converse` looks for each part of the sentence (the name and the thing the person likes) in two steps: first it finds the keywords `my name is` or `I like` and extracts everything following those phrases, then it looks within what it extracted for the word `and` and removes anything following it. For example, when I typed

```
My name is Brian and I like Chinese food
```

the result of matching the name pattern was to give the variable `name` the value

```
Brian and I like Chinese food
```

Then `strip.and` used a second pattern to eliminate everything after the `and`. You might be tempted to extract the name in one step by using a pattern like

```
[# my name is #name and #]
```

but I wanted to avoid that pattern because it won't match a sentence that only contains

```
my name is Mary
```

without expressing any likes or dislikes. The program as I've written it does accept these shorter sentences also. Later we'll see a more complicated pattern that accepts sentences with or without `and` using a single pattern.

The special symbol `#` in a pattern represents zero or more words. `Match` recognizes other symbols with different meanings:

| | |
|---|---|
| `#` | zero or more |
| `&` | one or more |
| `?` | zero or one |
| `!` | exactly one |

For example, if you'd like the `converse` program to recognize only the first name of the person using it, you could change the relevant pattern to

```
[# my name is !name #]
```

Then a conversation with the program might look like this:

```
? converse
Hi, my name is Toby and I like ice cream
Tell me about yourself
My name is Brian Harvey
Hello, Brian
Nice meeting you!
?
```

The word `!name` in the pattern matched just the single word `Brian`, not the multiple words `Brian Harvey` that the original pattern would have selected. (If you modify `converse` in this way, it should be possible to remove the invocation of `strip.and` in computing the input to `do.name`. The single word stored in the variable `name` won't contain any other clauses.)

So far, the patterns we've seen allow two extremes: the pattern can include a single word that must be matched exactly, or it can allow *any* word at all to be matched. It is also possible to write a pattern that calls for words in some specified category—that is, words that satisfy some predicate. Here is an example:

```
to ask.age
local "age
print [How old are you?]
if match [# !age:numberp #] readlist ~
   [print (sentence [You are] :age [years old.]]
end

? ask.age
How old are you?
I will be 36 next month
You are 36 years old.
```

This is a slightly silly example, but it does illustrate the use of a predicate to restrict which words can match a variable part of a pattern. The pattern used in `ask.age` looks for a single word for which `numberp` is `true`, that is, for a number. Any number of words surrounding the number are allowed.

Of course, a predicate used in a pattern need not be a primitive one like `numberp`. You may find it useful to write your own predicates that select categories of words. Such a predicate might have a list built in:

```
to colorp :word
output memberp :word [red orange yellow blue green violet white black]
end
```

Or you could check some inherent property of a word:

```
to ends.y :word
output equalp last :word "y
end
```

In either case, what is essential is that your predicate must take a word as its single input, and must output `true` if you want `match` to accept the word to fill a slot in the pattern.

It is most common to want a predicate like `colorp` above—one that tests its input word for membership in a certain list. A special notation makes it possible to include such a list in the pattern itself, instead of writing a predicate procedure. For example, suppose you are writing a quiz program, and you want to ask the question, "What is the quickest route from Boston to Framingham?" You'd like to accept answers like these:

```
Mass Pike
the Massachusetts Turnpike
the Pike
```

but not `the Ohio Turnpike`! Here is a pattern you could use.

```
[?:in [the] ?:in [Mass Massachusetts] !:in [Pike Turnpike]]
```

The special predicate `in` is a version of `memberp` that knows to look in the pattern, right after the element that invokes `in`, for the list of acceptable words. This pattern accepts zero or one `the`, zero or one of `Mass` or `Massachusetts`, and one of `Pike` or `Turnpike`. That is, the first two words are optional and the third is required.

Earlier I rejected the use of a pattern

```
[# my name is #name and #]
```

because I wanted also to be able to accept sentences without `and` following the name. I promised to exhibit a pattern that would accept both sentence forms. Here it is:

```
[# my name is #name:notand #]
```

This pattern uses a predicate `notand` that allows any word except `and`. It's easy to write this predicate:

```
to notand :word
output not equalp :word "and
end
```

(By the way, the symbols indicating the number of words to match are meant to be mnemonic. The question mark indicates that it's questionable whether or not the word will be in the sentence. The exclamation point looks a little like a digit 1, and also shouts emphatically that the word is present. The number sign means that any number of words (including zero) is okay, and the ampersand indicates that more words are required, namely at least one instead of at least zero.)

We've seen various combinations of quantifiers (that's what I'll call the characters like # that control how many words are matched), variable names, and predicates:

| | |
|---|---|
| `#` | no variable, no predicate (accept any word) |
| `#name` | set variable, no predicate |
| `?:in` | no variable, test predicate |
| `!age:numberp` | set variable, test predicate |

We are now about to discuss some of the more esoteric features of the `match` program. So far, we have always compared a pattern against a *sentence,* a list of words. It is also possible to match a pattern against a structured list, with smaller lists among its members. `Match` treats a sublist just like a word, if you don't want to examine the inner structure of the sublist. Here are some examples.

```
? print match [hello #middle goodbye] [hello is [very much] like goodbye]
true
? show :middle
[is [very much] like]
? print match [hi #middle:wordp bye] [hi and then bye]
true
? show :middle
[and then]
? print match [hi #middle:wordp bye] [hi and [then] bye]
false
```

```
? print match [hi #mid:wordp #dle:listp bye] [hi and [then] bye]
true
? show :mid show :dle
[and]
[[then]]
```

A more interesting possibility is to ask `match` to apply a sub-pattern to a sublist. This is done by using the pattern (that is, a list) in place of the name of a predicate. Here is an example:

```
? print match [a #:[x # y] b] [a [x 111 y] [x 222 y] b]
true
? print match [a #:[x # y] b] [a [x 333 zzz] b]
false
```

It is possible to include variable names in the subpattern, but this makes sense only if the quantifier outside the pattern is `!` or `?` because otherwise you may be trying to assign more than one value to the same variable. Here's what I mean:

```
? print match [a #all:[x #some y] b] [a [x 111 y] [x 222 y] b]
true
? show :all show :some
[[x 111 y] [x 222 y]]
[222]
```

The variable `all` is properly set to contain both of the lists that matched the subpattern, but the variable `some` only contains the result of the second match.

If a list appears in a pattern without a quantifier before it, `match` treats it as if it were preceded by "`!:`"; in other words, it tries to match the subpattern exactly once.

A pattern element like `#:predicate` can match several members of the target sentence; the predicate is applied to each candidate member separately. For example:

```
? print match [#nums:numberp #rest] [3 2 1 blastoff!]
true
? show :nums show :rest
[3 2 1]
[blastoff!]
```

Sometimes you may want to match several members of a sentence, but apply the predicate to all of the candidates *together* in one list. To do this, use the quantifier `@`:

```
to threep :list
output equalp count :list 3
end

? print match [@begin:threep #rest] [a b c d e]
true
? show :begin show :rest
[a b c]
[d e]
```

In this example, I haven't used the predicate to examine the *nature* of the matching words, but rather to control the *number* of words that are matched. Here is another example that looks "inside" the matching words.

```
to headtailp :list
if (count :list) < 2 [output "false]
output equalp first :list last :list
end

? print match [#front @good:headtailp #back] [a b c x d e f g x h i]
true
? show :front show :good show :back
[a b c]
[x d e f g x]
[h i]
```

Think about all the different tests that `match` has to make to find this match! Also, do you see why the first instruction of `headtailp` is needed?

Some patterns are *ambiguous;* that is, there might be more than one way to associate words from the matched sentence with quantifiers in the pattern. For example, what should the following do?

```
match [#front xx #back] [a b c d xx e f g xx h i]
```

The word `xx` appears twice in the matched sentence. The program could choose to use everything up to the first `xx` as `front`, leaving six words for `back`, or it could choose to use everything up to the second `xx` as `front`, leaving only two words for `back`. In fact, each quantifier, starting from the left, matches as many words as it can:

```
? print match [#front xx #back] [a b c d xx e f g xx h i]
true
? show :front show :back
[a b c d xx e f g]
[h i]
```

If that's not what you want, the quantifier ^ behaves like # except that it matches as *few* words as possible.

```
? print match [^front xx #back] [a b c d xx e f g xx h i]
true
? show :front show :back
[a b c d]
[e f g xx h i]
```

We can use the ^ quantifier to fix a bug in the `converse` program on page 111:

```
? converse
Hi, my name is Toby and I like ice cream
Tell me about yourself
My name is Brian and I like bacon and eggs
Hello, Brian and I like bacon
I'm glad you like bacon
Nice meeting you!
```

The problem here is that the pattern used by `strip.and` divided the sentence at the second `and`, just as the earlier example chose the second `xx` when I used # as the quantifier. We can fix it this way:

```
to strip.and :text
local "short
if match [^short and #] :text [output :short]
output :text
end

? converse
Hi, my name is Toby and I like ice cream
Tell me about yourself
My name is Brian and I like bacon and eggs
Hello, Brian
I'm glad you like bacon
Nice meeting you!
```

There is just one more special feature of `match` left to describe. It is another special predicate, like `in`, but this one is called `anyof`. When you use `anyof`, the next member of the pattern should be a *list of patterns* to test. `Match` tries each pattern in turn, applied to list members as determined by the quantifier used. In practice, though, `anyof` only makes sense when applied to several members as a group, so the quantifier `@` should always be used. An example may make this clear. I'm going to rewrite the `converse` program to check for names and likes all at once.

```
to converse
local [response name like rest]
print [Hi, my name is Toby and I like ice cream]
print [Tell me about yourself]
make "response readlist
while match [@:anyof [[My name is #name:notand]
                      [I like #like:notand]
                      [&:notand]]
                     ?:in [and] #rest] ~
            :line ~
      [make "response :rest]
if not emptyp :name [print sentence "Hello, :name]
if not emptyp :like [print sentence [I'm glad you like] :like]
print [Nice meeting you!]
end
```

This program uses the `notand` predicate I wrote earlier. It checks for clauses separated by the word `and`. Each clause can match any of three patterns, one for the name, one for the liking, and a general pattern that matches any other clause. The clauses can appear in any order.

```
? converse
Hi, my name is Toby and I like ice cream
Tell me about yourself
My name is Brian and I hate cheese
Hello, Brian
Nice meeting you!

? converse
Hi, my name is Toby and I like ice cream
Tell me about yourself
I like wings and my name is Jonathan
Hello, Jonathan
I'm glad you like wings
Nice meeting you!
```

## Reinventing `Equalp` for Lists

`Match` is a kind of fancy `equalp` with a complicated understanding of what equality means. One way to approach an understanding of `match` is to begin with this question: Suppose Logo's primitive `equalp` only worked for comparing two *words* for equality. (For the remainder of this section, I won't use the word `equalp` at all; I'll call this imaginary primitive `wordequalp` instead.) How would you write a `listequalp` to compare two lists? This is basically a `butfirst`-style recursive operation, but you have to be a little careful about the fact that either input might be smaller than the other.

```
to listequalp :a :b
if emptyp :a [output emptyp :b]
if emptyp :b [output "false]
if wordequalp first :a first :b ~
   [output listequalp butfirst :a butfirst :b]
output "false
end
```

(This procedure contains the instruction `output "false` twice, but it never says `output "true`. How can it ever say that two lists are equal?)

There is one deficiency in the procedure as I've defined it. The problem is that it only works for *sentences*—lists whose members are words. If either list contains a sublist, `listequalp` will try to apply `wordequalp` to that sublist. If you enjoy the exercise of reinventing Logo primitives, you may want to fix that. But for my purposes, the version here is good enough as a basis for further development of the pattern matcher.

## A Simple Pattern Matcher

We can extend the idea of `listequalp` slightly to make a pattern matcher that only recognizes the special word `#` to mean "match zero or more words." We won't do any of the fancy things like storing the matching words in a variable.

```
to match :pat :sen
if emptyp :pat [output emptyp :sen]
if emptyp :sen [if equalp first :pat "#
                   [output match butfirst :pat :sen]
                   [output "false]]
if equalp first :pat "# [output or match butfirst :pat :sen
                                   match :pat butfirst :sen]
if equalp first :pat first :sen ~
   [output match butfirst :pat butfirst :sen]
output "false
end
```

The end test is more complicated in this program than in `listequalp` because the combination of an empty sentence and a nonempty pattern can still be a match, if the pattern is something like `[#]` that matches zero or more words.

The really interesting part of this procedure is what happens if a `#` is found in the pattern. The match succeeds (outputs `true`) if one of two smaller matches succeeds. The two smaller matches correspond to two possible conditions: the `#` can match zero words, or more than zero. The first case is detected by the expression

```
match butfirst :pat :sen
```

For example, suppose you want to evaluate

```
match [# cream] [cream]
```

This expression should yield the value `true`, with the `#` matching no words in the sentence. In this example the expression

```
match butfirst :pat :sen
```

is equivalent to

```
match [cream] [cream]
```

which straightforwardly outputs `true`.

On the other hand, the expression

```
match :pat butfirst :sen
```

comes into play when the `#` has to match at least one word. For example, consider the expression

```
match [# cream] [ice cream]
```

Here the `#` should match the word `ice`. The expression

```
match :pat butfirst :sen
```

is here equivalent to

```
match [# cream] [cream]
```

But this is the example that was `true` just above.

If the `#` has to match more than one word, several recursive invocations of `match` are required, each one taking the `butfirst` of the sentence once. For example, suppose we start with

```
match [# cream] [vanilla ice cream]
```

Here is the sequence of recursive invocations leading to a `true` match:

```
match :pat butfirst :sen      match [# cream] [ice cream]
  match :pat butfirst :sen      match [# cream] [cream]
    match butfirst :pat :sen      match [cream] [cream]
```

I have been talking as if Logo only evaluated whichever of the two expressions

```
match butfirst :pat :sen
```

and

```
match :pat butfirst :sen
```

is appropriate for the particular inputs used. Actually, *both* expressions are evaluated each time, so there are many recursive invocations of `match` that come out `false`. However, the purpose of the primitive operation `or` is to output `true` if *either* of its inputs is `true`. To understand fully how `match` works, you'll almost certainly have to trace a few examples carefully by hand.

## Efficiency and Elegance

Pattern matching is a complicated task, and even the best-written programs are not blindingly fast. But what is the "best-written" program? In the simple pattern matcher of the last section, the instruction

```
if equalp first :pat "# [output or match butfirst :pat :sen
                                   match :pat butfirst :sen]
```

is extremely compact and elegant. It packs a lot of power into a single instruction, by combining the results of two recursive invocations with `or`. The similarity of the inputs to the two invocations is also appealing.

The trouble with this instruction is that it is much slower than necessary, because it always tries both recursive invocations even if the first one succeeds. A more efficient way to program the same general idea would be this:

```
if equalp first :pat "# ~
   [if match butfirst :pat :sen
       [output "true]
       [output match :pat butfirst :sen]]
```

This new version is much less pleasing to the eye, but it's much faster. The reason is that if the expression

```
match butfirst :pat :sen
```

outputs `true`, then the other recursive invocation is avoided.

It's a mistake to make efficiency your only criterion for program style. Sometimes it's worth a small slowdown of your program to achieve a large gain in clarity. But this is a case in which the saving is quite substantial. Here is a partial trace of the evaluation of

```
match [cat # bat] [cat rat bat]
```

using the original version of the procedure:

```
match [cat # bat] [cat rat bat]              [cat # bat]   [cat rat bat]
  match butfirst :pat butfirst :sen          [# bat]       [rat bat]
    match butfirst :pat :sen                 [bat]         [rat bat]
    match :pat butfirst :sen                 [# bat]       [bat]
      match butfirst :pat :sen               [bat]         [bat]
        match butfirst :pat butfirst :sen    []            []
*     match :pat butfirst :sen               [# bat]       []
*       match butfirst :pat :sen             [bat]         []
```

The two invocations marked with asterisks are avoided by using the revised version. These represent 25% of the invocations of `match`, a significant saving. (Don't think that the program necessarily runs 25% faster. Not all invocations take the same amount of time. This is just a rough measure.) If there were more words after the # in the pattern, the saving would be even greater.

In this situation we achieve a large saving of time by reorganizing the flow of control in the program. This is quite different from a more common sort of concern for efficiency, the kind that leads people to use shorter variable names so that the program

will be a little smaller, or to worry about whether to use `fput` or `sentence` in a case where either would do. These small "bumming" kinds of optimization are rarely worth the trouble they cause. Figuring out how many times `match` is invoked using each version is a simple example of the branch of computer science called *analysis of algorithms;* a more profound analysis might use mathematical techniques to compare the two versions in general, rather than for a single example.

In the full version of the pattern matcher, listed at the end of this project description, I've taken some care to avoid unnecessary matching. On the other hand, the full version has less flexibility than the simple version because of its ability to assign matching words to variables. Consider a case like

```
match [# #] [any old list of words]
```

Which `#` matches how many words? It doesn't matter if you don't store the result of the match in variables. But if the pattern is `[#a #b]` instead, there has to be a uniform rule about which part of the pattern matches what. (In my pattern matcher, all of the words would be assigned to `a`, and `:b` would be empty. In general, pattern elements toward the left match as many words as possible when there is any ambiguity.) The simple pattern matcher doesn't have this problem, and can be written to match the ambiguous pattern whichever way gives a `true` result most quickly.

By the way, what if the two expressions that invoke `match` recursively were reversed in the revised instruction? That is, what if the instruction were changed again, to read

```
if equalp first :pat "# ~
   [if match :pat butfirst :sen
       [output "true]
       [output match butfirst :pat :sen]]
```

Would this be more or less efficient than the previous version?

## Logo's Evaluation of Inputs

The discussion about efficiency started because Logo *evaluates* the inputs to the primitive operation `or` before invoking the procedure. That is, in the example in question, Logo invokes `match` twice before using `or` to check whether either invocation output `true`. This is consistent with the way Logo does things in general: To evaluate an expression that uses some procedure, Logo first evaluates all the inputs for that procedure, and then invokes the procedure with the evaluated inputs. Logo's rule is extremely consistent

(except for the `to` command), but it isn't the only possible way. In Lisp, a language that's like Logo in many ways, each procedure can choose whether or not its inputs should be evaluated in advance.

An example may make it clearer what I mean by this. Lisp has a procedure called `set` that's equivalent to the Logo `make`. You say

```
(set 'var 27)
```

as the equivalent of

```
make "var 27
```

But Lisp also has a version called `setq` whose first input is *not* evaluated before `setq` is invoked. It's as if there were an automatic quote mark before the first input, so you just say

```
(setq var 27)
```

with the same effect as the other examples.

Except for the special format of the `to` command that forms the title line of a procedure, Berkeley Logo and many other Logo dialects do not have any form of automatically-quoted inputs. The design principle was that consistency of evaluation would make the rules easier to understand. Some other versions of Logo do use auto-quoting for certain procedures. For example, in Berkeley Logo, to edit the definition of a procedure named `doit` you type the instruction

```
edit "doit
```

But in some other versions of Logo you instead say

```
edit doit
```

because in those versions, the `edit` command auto-quotes its input. One possible reason for this design decision is that teachers of young children like to present Logo without an explicit discussion of the evaluation rules. They teach the `edit` command as a special case, rather than as just the invocation of a procedure like everything else. Using this approach, auto-quoting the input avoids having to explain what that quotation mark means.

The advantage of the non-auto-quoting version of `edit` isn't just in some abstract idea of consistency. It allows us to take advantage of composition of functions. Suppose you are working on a very large project, a video game, with hundreds of procedures. You

want to edit all the procedures having to do with the speed of the spaceships, or whatever moves around the screen in this game. Luckily, all the procedures you want have the word `speed` as part of their names; they are called `shipspeed` or `asteroidspeed` or `speedcontrol`. You can say

```
edit filter [substringp "speed ?] procedures
```

(`Procedures` is a Berkeley Logo primitive operation that outputs a list of all procedures defined in the workspace; `substringp` is a predicate that checks whether one word appears as part of a longer word.) An auto-quoting `edit` command wouldn't have this flexibility.

The reason all this discussion is relevant to the pattern matcher is that the Lisp versions of `or` and `and` have auto-quoted inputs, which get evaluated one by one. As soon as one of the inputs to `or` turns out to be `true` (or one of the inputs to `and` is `false`), the evaluation stops. This is very useful not only for efficiency reasons, as in the discussion earlier, but to prevent certain kinds of errors. For example, consider this Logo instruction:

```
if not emptyp :list [if equalp first :list 1 [print "one]]
```

It would be pleasant to be able to rewrite that instruction this way:

```
if and (not emptyp :list) (equalp first :list 1) [print "one]
```

The use of `and`, I think, makes the program structure clearer than the nested `if`s. That is, it's apparent in the second version that something (the `print`) is to be done if two conditions are met, and that that's all that happens in the instruction. In the first version, there might have been another instruction inside the range of the first (outer) `if`; you have to read carefully to see that that isn't so.

Unfortunately, the second version won't work in Logo. If `:list` is in fact empty, the expression

```
(equalp first :list 1)
```

is evaluated before `and` is invoked; this expression causes an error message because `first` doesn't accept an empty input. In Lisp, the corresponding instruction *would* work, because the two predicate expressions would be evaluated serially and the second wouldn't be evaluated if the first turned out to be false.

The serial evaluation of inputs to `and` and `or` is so often useful that some people have proposed it for Logo, even at the cost of destroying the uniform evaluate-first rule.

But if you want a serial `and` or `or`, it's easy enough to write them, if you explicitly quote the predicate expressions that are its inputs:

```
to serial.and :pred1 :pred2
if not run :pred1 [output "false]
output run :pred2
end

to serial.or :pred1 :pred2
if run :pred1 [output "true]
output run :pred2
end
```

Here's how you would use `serial.and` to solve the problem with the nested `if`s:

```
if (serial.and [not emptyp :list] [equalp first :list 1]) [print "one]
```

Similarly, you could use `serial.or` instead of `or` to solve the efficiency problem in the first version of the pattern matcher:

```
output serial.or [match butfirst :pat :sen] [match :pat butfirst :sen]
```

These procedures depend on the fact that the predicate expressions that are used as their inputs are presented inside square brackets; that's why they are not evaluated before `serial.and` or `serial.or` is invoked.

## Indirect Assignment

From now on, I'll be talking about the big pattern matcher, not the simple one I introduced to illustrate the structure of the problem. Here is the top-level procedure `match`:

```
to match :pat :sen
local [special.var special.pred special.buffer in.list]
if or wordp :pat wordp :sen [output "false]
if emptyp :pat [output emptyp :sen]
if listp first :pat [output special fput "!: :pat :sen]
if memberp first first :pat [? # ! & @ ^] [output special :pat :sen]
if emptyp :sen [output "false]
if equalp first :pat first :sen ~
   [output match butfirst :pat butfirst :sen]
output "false
end
```

As you'd expect, there are more cases to consider in this more featureful version, but the basic structure is similar to the simple matcher. The instructions starting `if emptyp`, `if memberp`, `if emptyp`, and `if equalp` play the same roles as similar instructions in the other version. (The `memberp` test replaces the comparison against the word `#` with a wider range of choices.)

The first `if` instruction tests for errors in the format of the pattern or the sentence to be matched, in which a word is found where a list was expected. It's not important if you use well-formed inputs to `match`. The `listp` test essentially converts a pattern like

```
[foo [some # pattern] baz]
```

to the equivalent form

```
[foo !:[some # pattern] baz]
```

The interesting new case comes when `match` sees a word in the pattern that starts with one of the six special quantifier characters. In this case, `match` invokes `special` to check for a match.

One of the interesting properties of `special` is that it has to be able to assign a value to a variable whose name is not built into the program, but instead is part of the *data* used as input to the program. That is, if the word

```
?howmany:numberp
```

appears in the pattern, `special` (or one of its subprocedures) must assign a value to the variable named `howmany`, but there is no instruction of the form

```
make "howmany ...
```

anywhere in the program. Instead, `match` has *another* variable, whose name is `special.var`, whose *value* is the *name* `howmany`. The assignment of the matching words to the pattern-specified variable is done with an instruction like

```
make :special.var ...
```

Here the first input to `make` is not a quoted word, as usual, but an expression that must be evaluated to figure out which variable to use.

`Special`, then, has two tasks. First it must divide a word like

```
?howmany:numberp
```

into its component parts; then it must carry out the matching tasks that are the *meaning* of those parts. These two tasks are like a smaller version of what a programming language interpreter like Logo does. Finding the meaningful parts of an instruction is called the *syntax* of a language, and understanding what the parts mean is called the *semantics* of the language. `Special` has two instructions, one for the syntax and one for the semantics:

```
to special :pat :sen
set.special parse.special butfirst first :pat "
output run word "match first first :pat
end
```

To *parse* something is to divide it into its pieces. `Parse.special` outputs a list of the form

```
[howmany numberp]
```

for the example we're considering. Then `set.special` assigns the two members of this list as the values of two variables. The variable named `special.var` is given the value `howmany`, and the variable named `special.pred` is given the value `numberp`. This preliminary work is what makes possible the indirect assignment described earlier.

## Defaults

What happens if the pattern has a word like

```
?:numberp
```

without a variable name? What happens when the program tries to assign a value to the variable named in the pattern? `Set.special` contains the instruction

```
if emptyp :special.var [make "special.var "special.buffer]
```

The effect of this instruction is that if you do not mention a variable in the pattern, the variable named `special.buffer` will be used to hold the results of the match. This variable is the *default* variable, the one used if no other is specified.

It's important, by the way, that the variable `special.buffer` is declared to be local in procedure `match`. What makes it important is that `match` is recursive; if you use a pattern like

```
[a # b # c]
```

then the matching of the second # is a subproblem of the matching of the first one. Match invokes special, which invokes match#, which invokes #test, which invokes match on the butfirst of the pattern. That butfirst contains another #. Each of these uses the variable special.buffer to remember the words it is trying as a match; since the variable is declared local, the two don't get confused. (This means, by the way, that you can really confuse match by using the same variable name twice in a pattern. It requires a fairly complicated pattern to confuse match, but here is an example. The first result is correct, the second incorrect.

```
? print match [a #x b &y ! c] [a i b j c b k c]
true
? show :x show :y
[i]
[j c b]
? print match [a #x b &x ! c] [a i b j c b k c]
false
```

The only change is that the variable name x is used twice in the second pattern, and as a result, match doesn't find the correct match. You'll know that you really understand how match works if you can explain why it *won't* fail if the ! is removed from the pattern.)

When writing a tool to be used in other projects, especially if the tool will be used by other people, it's important to think about defaults. What should the program do if some piece of information is missing? If you don't provide for a default explicitly, the most likely result is a Logo error message; your program will end up trying to take first of an empty list, or something like that.

Another default in the pattern matcher is for the predicate used to test matches. For example, what happens when the word

```
?howmany
```

appears in the pattern, without a predicate? This case is recognized by parse.special, in the instruction

```
if emptyp :word [output list :var "always]
```

The special predicate always is used if no other is given in the pattern. Always has a very simple definition:

```
to always :x
output "true
end
```

## Program as Data

The instruction in `special` that carries out the semantics of a special pattern-matching instruction word is

```
output run word "match first first :pat
```

If the pattern contains the word

```
?howmany:numberp
```

then this instruction extracts the quantifier character `?` (the first character of the first word of the pattern) and makes from it a procedure name `match?`. That name is then `run` as a Logo expression; that is, `special` invokes a procedure whose name is `match?`.

Most programming languages do not allow the invocation of a procedure based on finding the name of the procedure in the program's data. Generally there is a very strict separation between program and data. Being able to manipulate data to create a Logo instruction, and then run it, is a very powerful part of Logo. It is also used to deal with the names of predicates included in the pattern; to see if a word in the sentence input to `match` is a match for a piece of the pattern, the predicate `try.pred` contains the instruction

```
output run list :special.pred quoted first :sen
```

This instruction generates a list whose first member is the name of the predicate found in the pattern and whose second and last member is a word from the sentence. Then this list is run as a Logo expression, which should yield either `true` or `false` as output, indicating whether or not the word is acceptable.

## Parsing Rules

When you are reading the program, remember that the kind of pattern that I've written as

```
[begin !:[smaller # pattern] end]
```

is read by Logo as if I'd written

```
[begin !: [smaller # pattern] end]
```

That is to say, this pattern is a list of four members. I think of the middle two as a unit, representing a single thing to match. The sublist takes the place of a predicate name after the `!` quantifier. But for Logo, there is no predicate name in the word starting with the exclamation point; the pattern is a separate member of the large list. That's why `set.special` uses the expression

```
emptyp :special.pred
```

to test for this situation, rather than `listp`. After `parse.special` does its work, all it has found is a colon with nothing following it. `Set.special` has to look at the next member of the pattern list in order to find the subpattern.

## Further Explorations

Chapter 9 is a large program that uses `match`. It may give you ideas for the ways in which this tool can be used in your own programs. Here, instead of talking about applications of `match`, I'll discuss some possible extensions or revisions of the pattern matcher itself.

There are many obvious small extensions. For example, to complement the special `in` primitive, you could write `notin`, which would accept all *but* the members of the following list. You could allow the use of a number as the predicate, meaning that exactly that many matching words are required. That is, in the example for which I invented the predicate `threep`, I would instead be able to use

```
[@begin:3 #rest]
```

as the pattern.

There is no convenient way to say in a pattern that some subpattern can be repeated several times, if the subpattern is more than a single word. That is, in the second version of `converse`, instead of having to use `while` to chop off pieces of the matched sentence into a variable `rest`, I'd like to be able to say in the pattern something like

```
[@@: [@:anyof [[my name is #name:notand]
               [i like #like:notand]
               [&:notand]]
      ?:in [and]]]
```

Here the doubled atsign (`@@`) means that the entire pattern that follows should be matched repeatedly instead of only once.

For other approaches to pattern matching, you might want to read about the programming languages Snobol and Icon, each of which includes pattern matching as one of its main features.

## Program Listing

```
to match :pat :sen
local [special.var special.pred special.buffer in.list]
if or wordp :pat wordp :sen [output "false]
if emptyp :pat [output emptyp :sen]
if listp first :pat [output special fput "!: :pat :sen]
if memberp first first :pat [? # ! & @ ^] [output special :pat :sen]
if emptyp :sen [output "false]
if equalp first :pat first :sen
   [output match butfirst :pat butfirst :sen]
output "false
end

;; Parsing quantifiers

to special :pat :sen
set.special parse.special butfirst first :pat "
output run word "match first first :pat
end

to parse.special :word :var
if emptyp :word [output list :var "always]
if equalp first :word ": [output list :var butfirst :word]
output parse.special butfirst :word word :var first :word
end

to set.special :list
make "special.var first :list
make "special.pred last :list
if emptyp :special.var [make "special.var "special.buffer]
if memberp :special.pred [in anyof] [set.in]
if not emptyp :special.pred [stop]
make "special.pred first butfirst :pat
make "pat fput first :pat butfirst butfirst :pat
end
```

```
to set.in
make "in.list first butfirst :pat
make "pat fput first :pat butfirst butfirst :pat
end

;; Exactly one match

to match!
if emptyp :sen [output "false]
if not try.pred [output "false]
make :special.var first :sen
output match butfirst :pat butfirst :sen
end

;; Zero or one match

to match?
make :special.var []
if emptyp :sen [output match butfirst :pat :sen]
if not try.pred [output match butfirst :pat :sen]
make :special.var first :sen
if match butfirst :pat butfirst :sen [output "true]
make :special.var []
output match butfirst :pat :sen
end

;; Zero or more matches

to match#
make :special.var []
output #test #gather :sen
end

to #gather :sen
if emptyp :sen [output :sen]
if not try.pred [output :sen]
make :special.var lput first :sen thing :special.var
output #gather butfirst :sen
end

to #test :sen
if match butfirst :pat :sen [output "true]
if emptyp thing :special.var [output "false]
output #test2 fput last thing :special.var :sen
end
```

```
to #test2 :sen
make :special.var butlast thing :special.var
output #test :sen
end

;; One or more matches

to match&
output &test match#
end

to &test :tf
if emptyp thing :special.var [output "false]
output :tf
end

;; Zero or more matches (as few as possible)

to match^
make :special.var []
output ^test :sen
end

to ^test :sen
if match butfirst :pat :sen [output "true]
if emptyp :sen [output "false]
if not try.pred [output "false]
make :special.var lput first :sen thing :special.var
output ^test butfirst :sen
end

;; Match words in a group

to match@
make :special.var :sen
output @test []
end

to @test :sen
if @try.pred [if match butfirst :pat :sen [output "true]]
if emptyp thing :special.var [output "false]
output @test2 fput last thing :special.var :sen
end
```

```
to @test2 :sen
make :special.var butlast thing :special.var
output @test :sen
end

;; Applying the predicates

to try.pred
if listp :special.pred [output match :special.pred first :sen]
output run list :special.pred quoted first :sen
end

to quoted :thing
if listp :thing [output :thing]
output word "" :thing
end

to @try.pred
if listp :special.pred [output match :special.pred thing :special.var]
output run list :special.pred thing :special.var
end

;; Special predicates

to always :x
output "true
end

to in :word
output memberp :word :in.list
end

to anyof :sen
output anyof1 :sen :in.list
end

to anyof1 :sen :pats
if emptyp :pats [output "false]
if match first :pats :sen [output "true]
output anyof1 :sen butfirst :pats
end
```

# 8 Property Lists

In the first volume of this series, I wrote a procedure named `french` that translates words from English to French using a dictionary list like this:

```
[[book livre] [computer ordinateur] [window fenetre]]
```

This technique works fine with a short word list. But suppose we wanted to undertake a serious translation project, and suppose we wanted to be able to translate English words into several foreign languages. (You can buy hand-held machines these days with little keyboards and display panels that do exactly that.) `Butfirst`ing through a list of tens of thousands of words would be pretty slow, and setting up the lists in the first place would be very difficult and error-prone.

If we were just dealing with English and French, one solution would be to set up many variables, with each an English word as its *name* and the corresponding French word as its *value:*

```
make "paper "papier
make "chair "chaise
make "computer "ordinateur
make "book "livre
make "window "fenetre
```

Once we've done this, the procedure to translate from English to French is just `thing`:

```
? print thing "book
livre
```

The advantage of this technique is that it's easy to correct a mistake in the translation; you just have to assign a new value to the variable for the one word that is in error, instead of trying to edit a huge list.

But we can't quite use this technique for more than one language. We could create variables whose names contained both the English word and the target language:

```
make "book.french "livre
make "book.spanish "libro

to spanish :word
output thing word :word ".spanish
end
```

This is a perfectly workable technique but a little messy. Many variables will be needed. A compromise might be to collect all the translations for a single English word into one list:

```
make "book [livre libro buch libro liber]

to spanish :word
output item 2 thing :word
end
```

## Naming Properties

The only thing wrong with this technique is that we have to remember the correct order of the foreign languages. This can be particularly tricky because some of the words are the same, or almost the same, from one language to another. And if we happen not to know the translation of a particular word in a particular language, we have to take some pains to leave a gap in the list. Instead we could use a list that tells us the languages as well as the translated words:

```
[French livre Spanish libro German buch Italian libro Latin liber]
```

A list in this form is called a *property list.* The odd-numbered members of the list are property *names,* and the even-numbered members are the corresponding property *values.*

You can see that this solution is a very flexible one. We can add a new language to the list later, without confusing old procedures that expect a particular length of list. If we don't know the translation for a particular word in a particular language, we can just leave it out. The order of the properties in the list doesn't matter, so we don't have to

remember it. The properties need not all be uniform in their significance; we could, for example, give `book` a property whose name is `part.of.speech` and whose value is `noun`.

To make this work, Berkeley Logo (along with several other dialects) has procedures to create, remove, and examine properties. The command `pprop` (Put PROPerty) takes three inputs; the first two must be words, and the third can be any datum. The first input is the name of a property list; the second is the name of a property; the third is the value of that property. The effect of `pprop` is to add the new property to the named list. (If there was already a property with the given name, its old value is replaced by the new value.) The command `remprop` (REMove PROPerty) takes two inputs, which must be words: the name of a property list and the name of a property in the list. The effect of `remprop` is to remove the property (name and value) from the list. The operation `gprop` (Get PROPerty) also takes two words as inputs, the name of a property list and the name of a property in the list. The output from `gprop` is the value of the named property. (If there is no such property in the list, `gprop` outputs the empty list.)

```
? print gprop "book "German
buch
```

## Writing Property List Procedures in Logo

It would be possible to write Logo procedures that would use ordinary variables to hold property lists, which would work just like the ones I've described. Since Berkeley Logo provides property lists as a primitive capability, you won't need to load these into your computer, but later parts of the discussion will make more sense if you understand how they work. Here they are:

```
to pprop :list :name :value
if not namep :list [make :list []]
make :list pprop1 :name :value thing :list
end

to pprop1 :name :value :oldlist
if emptyp :oldlist [output list :name :value]
if equalp :name first :oldlist ~
   [output fput :name (fput :value (butfirst butfirst :oldlist))]
output fput (first :oldlist) ~
            (fput (first butfirst :oldlist)
                  (pprop1 :name :value (butfirst butfirst :oldlist)))
end
```

```
to remprop :list :name
if not namep :list [make :list []]
make :list remprop1 :name thing :list
end

to remprop1 :name :oldlist
if emptyp :oldlist [output []]
if equalp :name first :oldlist [output butfirst butfirst :oldlist]
output fput (first :oldlist) ~
            (fput (first butfirst :oldlist)
                  (remprop1 :name (butfirst butfirst :oldlist)))
end

to gprop :list :name
if not namep :list [output []]
output gprop1 :name thing :list
end

to gprop1 :name :props
if emptyp :props [output []]
if equalp :name first :props [output first butfirst :props]
output gprop1 :name (butfirst butfirst :props)
end
```

Note that the input called `list` in each of these procedures is not a property list itself but the *name* of a property list. That's why each of the superprocedures evaluates

```
thing :list
```

to pass down as an input to its subprocedure.

## Property Lists Aren't Variables

The primitive procedures that support property lists in Berkeley Logo, however, do *not* use `thing` to find the property list. Just as the same word can independently name a procedure and a variable, a property list is a *third* kind of named entity, which is separate from the `thing` with the same name. For example, if we gave `book` the property list shown with a series of instructions like

```
pprop "book "French "livre
pprop "book "Spanish "libro
```

and so on, we would not be creating a *variable* named `book`.

```
? print :book
book has no value
```

(Of course, we could give `book` a value with a `make` instruction, but that value would have nothing to do with the property list.) Instead there is a fourth primitive procedure called `plist` that can be used to examine a property list. `Plist` takes one input, a word. It outputs the property list associated with that word. If there is no such property list, `plist` outputs the empty list.

## How Language Designers Earn Their Pay

If you're like me, you may have some questions about why this Logo feature works the way it does. The form of a property list, for example, may seem arbitrary to you. Why should it be a flat list, with names as the odd-numbered members and values as the even-numbered ones? Wouldn't it be more sensible to structure the list this way:

```
[[French livre] [Spanish libro] [German buch]
 [Italian libro] [Latin liber]]
```

In this scheme each member of a property list is *a property*. A property has two parts, a name and a value. A list structured in this way would be easier to use with iterative tools like `map`. (Try to figure out a way to redefine `map` so that it could map a function over *pairs* of members of its input list. Your goal is to find a way that isn't a kludge.) You wouldn't have to think "What if the list has an odd number of members" when writing procedures to manipulate property lists.

So why does Logo use the notation it does? I'm afraid the real answer is "It's traditional." Logo property lists are the way they are because that's what property lists look like in Lisp, the language from which Logo is descended. Now, why was that decision made in the design of Lisp? I'm not sure of the answer, but one possible reason is that the flat property lists take up less room in the computer's memory than the list-of-lists that I'd find more logical. (Logo measures its available memory in *nodes*. It takes two overhead nodes per property, not including the ones that actually contain the name and the value, for the flat property list; it would take three overhead nodes per property for the list-of-lists.)

Another minor advantage is that if you want to live dangerously, you can use `memberp` to see if a particular property name exists in a property list. It's living dangerously because

the property name might, by coincidence, be the *value* of some other property. (In the dictionary example, this would be the situation if the German word for "book" were "Greek"!) The advantage is that `memberp` is a primitive procedure, so it's faster than one you could write yourself that would check just the odd-numbered members of the property list.

## Fast Replacement

Another question you might ask is this one: Why have property list primitives at all? The list is a very general data structure, which can be organized in many ways. Why single out this particular way of using lists as the one to support with special primitive procedures? After all, it's easy enough to implement property lists in Logo, as I've done in this chapter.

One answer is that the primitives can be much faster than the versions I've written in Logo because they can replace a value inside a property list without recopying the rest of the list. My procedure `pprop1`, for example, has to do two `fput`s for each property in the list every time you want to change a single property. The primitive version of `pprop` doesn't reconstruct the entire list; it just rips out the old value from inside the list and sticks in a new value.

Aside from the question of speed, the difference between changing something inside a list and making a modified copy of the list may not seem like a big deal. But it does raise a subtle question. If you say

```
make "myprops plist "myself
```

and then, later, use `pprop` or `remprop` to change some of the properties of `myself`, does the value of the variable `myprops` change? The answer is no; `plist` really outputs a *copy* of the property list as it exists at the moment you invoke `plist`. That copy becomes the value of `myprops`, and it doesn't change if the property list itself is changed later. (Berkeley Logo, like Lisp, does have primitives that allow you to change things inside lists in general, and this possibility of a variable magically changing in value because you change something else really does arise!)

## Defaults

Another language design question you might be wondering about is why `gprop` outputs the empty list if you ask for a property that doesn't exist. How do you say "book" in Urdu?

```
? show gprop "book "urdu
[]
```

If you ask for a *variable* that doesn't exist, you get an error message. Why doesn't Logo print something like

```
book has no urdu property
```

in this situation?

The name for "what you get when you haven't provided an answer" is a *default.* There aren't very many situations in which Logo provides defaults. One obscure example in Berkeley Logo is the *origin* of an array—the number used to select its first member. By default the first member is number one, but it's possible to set up an array that begins with some other number (most commonly zero).

The question of what should be considered an error is always a hot one among language designers. The issue is one of programming convenience versus ease of debugging. Suppose `thing` output the empty list if asked for a nonexistent variable. It would have been easier for me to write the property list procedures in this chapter; I could have left out the `if not namep` instructions. This is a situation in which I might ask for a variable that hasn't been given a value "on purpose," with a perfectly clear idea of what result I want. On the other hand, if `thing` were permissive in this way, what would happen if I gave it an input that wasn't a variable name because I made a spelling error? Instead of getting an error message right away, my program would muddle on with an empty list instead of whatever value was really needed. Eventually I'd get a different error message or an incorrect result, and it would be much harder to find the point in the program that caused the problem.

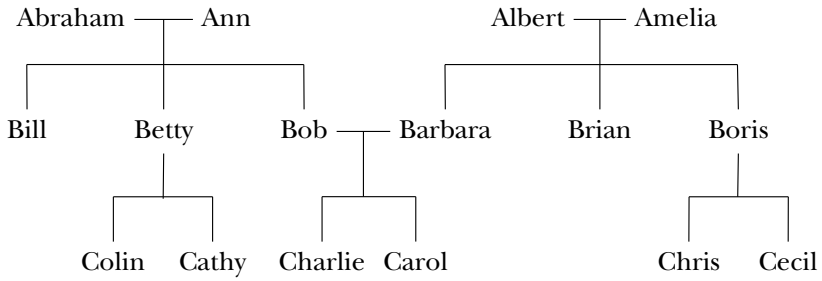
The same issue arises, by the way, about operations like `first`. What should `first` do if you give it an empty list as input? Logo considers this an error, as do most versions of Lisp. Some versions of Lisp, though, output an empty list in this situation.

It's most common to need "permissive" primitives when working on extensions to Logo itself, such as property lists, as opposed to specific application programs. An application programmer has complete control over the inputs that procedures will be given; an implementor of a programming language (or an extension to it) has to handle anything that comes up. I think that's why, traditionally, it's always been the *teachers* of Logo who vote in favor of error messages and the *implementors* who prefer permissive primitives.

So why is `gprop` permissive when all other Logo primitives are not? Well, the others were designed early in the history of the language when teachers were in charge at the design meetings. Property lists were added to Logo more recently; the implementors showed up one day and said, "Guess what? We've put in property lists." So they did it their way!

## An Example: Family Trees

Here is an example program using property lists. My goal is to represent this family tree:




Each person will be represented as a property list, containing the properties `mother`, `father`, `kids`, and `sex`. The first two will have words (names) as their values, `kids` will be a list of names, and `sex` will be `male` or `female`. Note that this is only a partial family tree; we don't know the name of Betty's husband or Boris's wife. Here's how I'll enter all this information:

```
to family :mom :dad :girls :boys
catch "error [pprop :mom "sex "female]
catch "error [pprop :dad "sex "male]
foreach :girls [pprop ? "sex "female]
foreach :boys [pprop ? "sex "male]
localmake "kids sentence :girls :boys
catch "error [pprop :mom "kids :kids]
catch "error [pprop :dad "kids :kids]
foreach :kids [pprop ? "mother :mom   pprop ? "father :dad]
end

family "Ann "Abraham [Betty] [Bill Bob]
family "Amelia "Albert [Barbara] [Brian Boris]
family "Betty [] [Cathy] [Colin]
family "Barbara "Bob [Carol] [Charlie]
family [] "Boris [] [Chris Cecil]
```

The instructions that catch errors do so in case a family has an unknown mother or father, which is the case for two of the ones in our family tree.

Now the idea is to be able to get information out of the tree. The easy part is to get out the information that is there explicitly:

```
to mother :person
output gprop :person "mother
end

to kids :person
output gprop :person "kids
end

to sons :person
output filter [equalp (gprop ? "sex) "male] kids :person
end
```

Of course several more such procedures can be written along similar lines.

The more interesting challenge is to deduce information that is not explicitly in the property lists. The following procedures make use of the ones just defined and other obvious ones like `father`.

```
to grandfathers :person
output sentence (father father :person) (father mother :person)
end

to grandchildren :person
output map.se [gprop ? "kids] (kids :person)
end

to granddaughters :person
output justgirls grandchildren :person
end

to justgirls :people
output filter [equalp (gprop ? "sex) "female] :people
end

to aunts :person
output justgirls sentence (siblings mother :person) ~
                          (siblings father :person)
end
```

```
to cousins :person
output map.se [gprop ? "kids] sentence (siblings mother :person) ~
                                       (siblings father :person)
end

to siblings :person
local "parent
if emptyp :person [output []]
make "parent mother :person
if emptyp :parent [make "parent father :person]
output remove :person kids :parent
end
```

In writing `siblings`, I've been careful to have it output an empty list if its input is empty. That's because `aunts` and `cousins` may invoke `siblings` with an empty input if we're looking for the cousins of someone whose father or mother is unknown.

You'll find, if you try out these procedures, that similar care needs to be exercised in some of the "easy" procedures previously written. For example, `grandfathers` will give an error message if applied to a person whose mother *or* father is unknown, even if the other parent is known. One solution would be a more careful version of `father`:

```
to father :person
if emptyp :person [output []]
output gprop :person "father
end
```

The reason for choosing an empty list as output for a nonexistent person rather than an empty word is that the former just disappears when combined with other things using `sentence`, but an empty word stays in the resulting list. So `grandfathers`, for example, will output a list of length 1 if only one grandfather is known rather than a list with an empty word in addition to the known grandfather. Procedures like `cousins` also rely heavily on the flattening effect of `sentence`.

This is rather an artificial family tree because I've paid no attention to family names, and all the given names are unique. In practice, you wouldn't be able to assume that. Instead, each property list representing a person would have a name like `person26` and would include properties `familyname` and `givenname` or perhaps just a `name` property whose value would be a list. All the procedures like `father` and `cousins` would output lists of these funny `person26`-type names, and you'd need another procedure `realnames` that would extract the real names from the property lists of people in a list. But I thought it would be clearer to avoid that extra level of naming confusion here.

# 9 Example: Doctor

Program file for this chapter: `doctor`

```
? doctor
Hello, I am the doctor.  What can i do for you?
Please end your remarks with an empty line.

My spirits are low.

Your spirits are low?

I'm really depressed.

I'm sorry to hear you are depressed.

It's because nobody likes me.

Really, nobody?

That's right.

I am not sure I understand you fully.

I'm lonely.

Is it because you are lonely that you came to me?

Of course.

Earlier you said your spirits are low.
```

This is the beginning of a dialogue with a program called `doctor`. The program simulates the behavior of a psychotherapist. People who interact with this program are

often fooled into thinking that the program understands what they are saying, although in fact the program does not understand at all. Instead, the program recognizes certain key words in a sentence, words like "problem" or "nobody." For each of these words, there is a list of *rules;* a rule consists of a pattern (of the sort used by the pattern matcher of Chapter 7) and a set of possible responses. For example, associated with the word "nobody" is the pattern

```
[# !a:in [everyone everybody nobody noone] #]
```

Corresponding to that pattern is a list of responses:

```
[[Really, :a?]
 [Surely not :a.]
 [Can you think of anyone in particular?]
 [Who, for example?]
 [You are thinking of a very special person.]
 [Who, may I ask?]
 [Someone special perhaps.]
 [You have a particular person in mind, don't you?]
 [Who do you think you're talking about?]
 [I suspect you're exaggerating a little.]]
```

The program has so many possible responses so that each time you use the word "nobody" (or "everybody," etc.) you get a different answer. Even though many of the answers have the same meaning, the variety in the wording helps to convince you that it's a real person at the other end of the conversation.

Many versions of this program have been written, but the first was written in 1966 by Joseph Weizenbaum, a professor of computer science at MIT. He called his program Eliza after Eliza Doolittle, a character in the play *Pygmalion* by George Bernard Shaw. Eliza started life poor and uneducated, with rough speech, but in the play she is taught to speak better. The program, too, can be taught rules to help it speak better.

> Because conversations must be about something, that is, because they must take place within some context, the program was constructed in a two-tier arrangement, the first tier consisting of the language analyzer and the second of a script. The script is a set of rules rather like those that might be given to an actor who is to use them to improvise around a certain theme. Thus Eliza could be given a script to enable it to maintain a conversation about cooking eggs or about managing a bank checking account, and so on. Each specific script thus enabled Eliza to play a specific conversational role.

> For my first experiment, I gave Eliza a script designed to permit it to play (I should really say parody) the role of a Rogerian psychotherapist engaged in an initial interview with a patient. The Rogerian psychotherapist is relatively easy to imitate because much of his technique consists of drawing his patient out by reflecting the patient's statements back to him...
>
> [Joseph Weizenbaum, *Computer Power and Human Reason* (Freeman, 1976), page 3.]

It has long been a popular Logo programming project to implement a small subset of `doctor`'s conversational ability through a program that embodies directly a few of the rules in Weizenbaum's Doctor script for Eliza. (See, for example, Harold Abelson, *Apple Logo* (BYTE Publications, 1982), page 158.) Home computers now have enough memory to permit the implementation in Logo of Weizenbaum's original two-tier approach. The program presented here is not a line-for-line translation of Eliza into Logo, but does embody the same fundamental strategy (namely pattern matching) as the original. The script is a close adaptation of Weizenbaum's version, via a Lisp version by Jon L. White.

You should try conversing with `doctor` yourself a few times. Whenever you get a response that seems linguistically bizarre, make a note of it. Later, as I'm talking about the way the program works, you can ask yourself how you would modify the script to eliminate the bad responses you've noted.

## Eliza and Artificial Intelligence

When Eliza was first unveiled, many people considered it a major advance in the pursuit of *artificial intelligence:* the search for ways to make computers as intelligent as people. Especially to a person unfamiliar with computer programming, a conversation with the Doctor can seem very real indeed. But as Weizenbaum himself points out, the underlying techniques used in the program do not involve any real understanding of the conversation. The program "cheats."

Today there are language-understanding programs that use techniques much more sophisticated than the simple pattern matching of Eliza. The authors of some such programs maintain that they really do understand what they are saying and hearing, in a sense in which Eliza does not. Other people, including Weizenbaum, suggest that even these state-of-the-art artificial intelligence programs are merely cheating in more complex ways. What would it mean for a program "really" to understand a conversation? This is a deep question that would take too long to explore here. (I'll come back to it in the discussion of artificial intelligence in the third volume of this series.) If you're

interested, you should begin by reading Weizenbaum's book, *Computer Power and Human Reason*, from which I quoted a passage earlier. He argues not only that computers *cannot* do certain things, but also that people *should not* use computers for certain purposes even if it were possible. In particular, he is horrified at the suggestion, made even by some psychiatrists, that programs like Eliza should be used to provide therapy to actual patients.

## Eliza's Linguistic Strategy

The program allows the user to enter one or more sentences, typing several lines if necessary. The first step in processing the user's remarks is to string the several lines into one long list. This list is a "sentence" in the Logo technical sense; that is, it's a list of words with no sublist structure. It may, however, contain one or more English sentences. The generation of a response involves several steps:

1. Find the punctuation in the input and use it to extract a single sentence to answer.
2. Find keywords in the sentence for which the script includes rules.
3. Apply word-for-word translations as specified by the script; these are primarily to convert first person ("I") to second person ("you") and vice versa.
4. Pick the highest priority keyword.
5. Find a rule for that keyword whose pattern matches the input sentence.
6. Choose a response according to that rule.

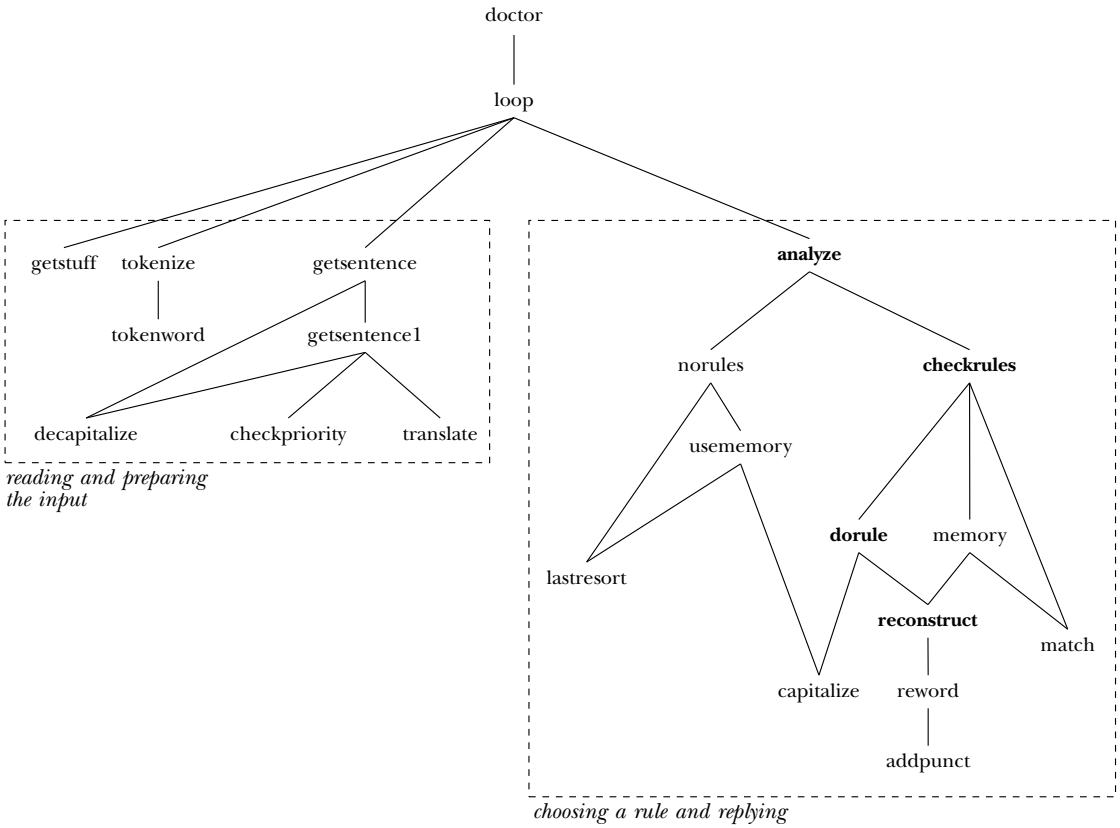
These steps are not really quite sequential; for example, the first two are done in parallel because, if there is more than one sentence, the program chooses a sentence that contains one or more keywords.

I have omitted from this list several possible complications. One important one is that there is a second kind of rule (besides the response rules) called a *memory* rule. These rules do not generate immediate responses to what the user types. Instead, they generate sentences that are appropriate for later use, like "Earlier you said your spirits are low" in the sample dialogue earlier. That response was not generated from the user's comment just before it, but was instead added to memory at the time of the user's first remark. Because the user's comment "Of course" contained no keywords, the program chose to use a response from memory at that time.

Other complications have to do with the nature of the rules associated with a keyword. Instead of patterns and responses, a keyword can just have *another* keyword as its rules, meaning that the rules for that keyword should be used for this one also. Alternatively, another keyword can be used in place of a response for a particular pattern,

so the alternate keyword is used only if that pattern is matched. Another alternative to a response is the word "newkey," which means that the program should abandon this keyword and try another one in the same sentence. Yet another alternative is an instruction to rearrange the sentence and try matching patterns again with the new version. As you read the procedures shown below, try not to get caught up in these complications the first time through. Just forget about the `if` instructions that check for these special cases and concentrate on the usual situation.

I'll explain how each part of the linguistic strategy is carried out by the procedures in the project. Here is a diagram of the subprocedure/superprocedure relationships. The top-level `doctor` does some initialization and then invokes `loop`, which does the real work.



```
to doctor
local [text sentence stuff a b c rules keywords memory]
make "memory []
print [Hello, I am the doctor. What can I do for you?]
print [Please end your remarks with an empty line.]
print []
loop
end

to loop
make "text tokenize getstuff []
make "sentence getsentence :text
analyze :sentence :keywords
print []
loop
end
```

`Loop` uses `getstuff` to read several lines of the user's typing into a long list. This list is processed by several procedures in turn.

```
to getstuff :stuff
localmake "line readlist
if emptyp :line [output :stuff]
output getstuff sentence :stuff :line
end
```

The first step in my numbered list is to find the punctuation and use it to extract a single sentence from the list. The first part, finding the punctuation, is the job of `tokenize` and its subprocedure `tokenword`. (These procedures get their name from the computer scientist's term *token*, which means the smallest possible meaningful string of characters. In the BASIC compiler of Chapter 6, the procedure called `reader` separates a line of text into tokens.)

```
to tokenize :text
output map.se [tokenword ? "] :text
end

to tokenword :word :out
if emptyp :word [output :out]
if memberp first :word [, " ] [output tokenword butfirst :word :out]
if memberp first :word [. ? ! |;|] [output sentence :out ".]
output tokenword butfirst :word word :out first :word
end
```

The program's understanding of punctuation is very simple. Some punctuation, like a quotation mark, is just ignored completely. Periods, question marks, and semicolons are all treated as having the same meaning, namely, they mark the end of a sentence. `tokenword` turns a word like `why?` into the two-word list `[why .]` so that the period as a separate word serves as the separator between sentences in the long list.

Extracting a single sentence is done at the same time as steps 2 and 3, finding keywords and applying translations. All of these are done by `getsentence`, which uses `checkpriority` and `translate` as subprocedures for tasks 2 and 3 respectively. (The version of Doctor in the first edition worked with all capital letters. In this new version, I've added a procedure `decapitalize` that turns the first letter of each sentence to lower case, and when the program prints a response it uses the inverse procedure `capitalize` to capitalize the first word. This is necessary because the first word of the user's sentence might end up in the middle of the program's response, and vice versa.)

```
to getsentence :text
make "keywords []
output getsentence1 decapitalize :text []
end

to getsentence1 :text :out
if emptyp :text [output :out]
if equalp first :text ". ~
   [ifelse emptyp :keywords
           [output getsentence1 decapitalize butfirst :text []]
           [output :out]]
checkpriority first :text
output getsentence1 butfirst :text sentence :out translate first :text
end

to decapitalize :text
if emptyp :text [output []]
output fput lowercase first :text butfirst :text
end

to checkpriority :word
localmake "priority gprop :word "priority
if emptyp :priority [stop]
if emptyp :keywords [make "keywords ( list :word ) stop]
ifelse :priority > ( gprop first :keywords "priority ) ~
       [make "keywords fput :word :keywords] ~
       [make "keywords lput :word :keywords]
end
```

```
to translate :word
localmake "new gprop :word "translation
output ifelse emptyp :new [:word] [:new]
end
```

At each sentence separator (a word containing only a period), `getsentence1` checks whether any keywords have been found yet. If so, the sentence before the separator is the one the program uses. If not, `getsentence1` goes on to examine the next sentence in the list. If the last sentence ends without any keywords found, that last sentence is chosen anyway.

Both `checkpriority` and `translate` work through the use of property lists that are associated with words. The script part of `doctor` (Weizenbaum's second tier) consists of these property lists. For example, here's part of the script setup:

```
pprop "my "priority 2
pprop "my "translation "your
```

The first of these means that `my` is a keyword, with priority 2. In the Doctor script, most keywords have priority 0. `My` is a little more important than most words, but not as important as `dreamed` (priority 4) or `computer` (priority 50)! `Checkpriority` arranges the list of keywords so that the word with highest priority is first in the list. The `translation` property means that the word `my` is changed to `your` in generating responses. For example, at the beginning of the sample dialogue above, the sentence "My spirits are low" is echoed as "Your spirits are low?" (The keyword list created by `checkpriority` has *untranslated* keywords. That's why the patterns associated with the keyword `you`, for example, all contain the word `I` instead; the patterns deal with the translated version.)

Step 4, finding the highest priority keyword, is simply a matter of choosing the `first` keyword in the list because of the way `checkpriority` has done its job. This selection is made by `analyze`, which then invokes `checkrules` as a subprocedure. (`Analyze` also recognizes the special situation in which one keyword refers to the rules of another.)

```
to analyze :sentence :keywords
local [rules keyword]
if emptyp :keywords [norules stop]
make "keyword first :keywords
make "rules gprop :keyword "rules
if wordp first :rules ~
   [make "keyword first :rules make "rules gprop :keyword "rules]
checkrules :keyword :rules
end
```

`Checkrules` handles step 5, finding an applicable rule for the chosen keyword. That is, `checkrules` invokes `match` to match the selected sentence against each pattern associated with the given keyword. (The program assumes that the script is written so that there will always be at least one matching pattern. Most keywords have `[#]` as the pattern in the last rule.) When `checkrules` finds a matching pattern, it invokes `dorule` to examine the corresponding list of responses. (One complication in understanding these procedures is that the input to `dorule` is *the name of a property* whose value is the list of responses. I'll get back to discussing why it's done this way later; for now, all that's really important is that `dorule` chooses one of the responses and uses it as the input to `reconstruct`.)

```
to checkrules :keyword :rules
if not match first :rules :sentence ~
   [checkrules :keyword butfirst butfirst :rules stop]
dorule first butfirst :rules
end

to dorule :rule
localmake "print first gprop :keyword :rule
pprop :keyword :rule lput :print butfirst gprop :keyword :rule
if equalp :print "newkey [analyze :sentence butfirst :keywords stop]
if wordp :print [checkrules :print gprop :print "rules stop]
if equalp first :print "pre ~
   [analyze reconstruct first butfirst :print
            butfirst butfirst :print
    stop]
print capitalize reconstruct :print
memory :keyword :sentence
end
```

The usual task of `dorule` is to carry out step 6, the generation of a response. For example, when the user typed

```
My spirits are low.
```

the highest priority keyword found was `my`. There are three patterns associated with this keyword:

```
[# your # !a:familyp #b]
[# your &stuff]
[#]
```

(Again, the pattern is matched against the sentence *after* translation, so the patterns contain the word `your` even though the actual keyword is `my`.) The first of these patterns does not match the sentence. (`Familyp` is a predicate that's true for words like `mother` and `brother`.) The second pattern, however, does match the sentence. `Match` gives the variable `stuff` the list

```
[spirits are low]
```

as its value. (The period that ended what the user typed was removed by `tokenize`.)

Associated with that second pattern is this list of responses:

```
[[Your :stuff?]
 [Why do you say your :stuff?]
 [Does that suggest anything else which belongs to you?]
 [Is it important to you that your :stuff?]]
```

`Dorule` chooses the first of these, and invokes `reconstruct` to substitute the actual value of the variable into the response. By the way, although I've used Logo's notation of colon to mean "the value of the variable," `reconstruct` isn't exactly like the Logo interpreter. For one thing, it recognizes punctuation marks; it knows that this response refers to a variable named `stuff`, not `stuff?`.

```
to reconstruct :sentence
if emptyp :sentence [output []]
if not equalp ": first first :sentence ~
   [output fput first :sentence reconstruct butfirst :sentence]
output sentence reword first :sentence reconstruct butfirst :sentence
end

to reword :word
if memberp last :word [. ? ,] ~
   [output addpunct reword butlast :word last :word]
output thing butfirst :word
end

to addpunct :stuff :char
if wordp :stuff [output word :stuff :char]
if emptyp :stuff [output :char]
output sentence butlast :stuff word last :stuff :char
end
```

## Stimulus-Response Psychology

Historically, there have been two ways of looking at the purpose of artificial intelligence research. One way is to see it as research into what computers can do, and into the meaning of intelligence in general, without any special reference to how *people* think. Researchers who take this approach are willing to use any technique that will solve a problem, even if it's perfectly obvious that people don't think that way. The second approach is to see artificial intelligence as a way to shed light on human intelligence. In this approach, the idea is to use the computer as a *model* for the human mind. Researchers who follow this path try to write their programs to mimic human behavior and, they hope, even the inner mechanisms of human brains. Recently there has been a tendency for researchers to declare themselves as wholly in one camp or the other. People who want to solve problems even if by non-human methods are part of the "knowledge engineering" field; the programs they develop are called "expert systems." People who want to use the computer to help build theories of human intelligence are in the field of "cognitive science."

Weizenbaum's work on Eliza is an early example of the former approach. He was emphatically *not* claiming that his program worked the way people work. Indeed, one of the purposes of the program was to demonstrate how realistic the *behavior* of a computer program can be, even when we are quite sure that the underlying *mechanism* is completely unrealistic.

Nevertheless, Eliza could be taken as a computer model of a certain theory of human psychology. It may not be obvious what it means for a computer program to model a theory about people, so it may be worthwhile to examine Eliza from this point of view. One theory about how people think is called *behaviorism,* or the *stimulus-response* theory. According to this theory, a person's mind is a "black box," and we can't know what's inside. What we *can* know, however, is how a person reacts to different situations and events. Whatever situation presents itself to you is called a *stimulus.* A stimulus can be something very abrupt like an electric shock, or it can be something more subtle like a particular sentence spoken by a particular person. When you are presented with a given stimulus, you produce a certain *response*: you say something back, or you jump, or you fall asleep. People learn to associate certain responses with certain stimuli. People can be trained to change the response associated with a stimulus by using *conditioning* techniques. If you are rewarded for producing a certain response, you'll produce it more often.

The behaviorist theory was very influential several years ago, although hardly anyone believes it any more. What would it mean to write a computer model for this theory? Well, the model would have two main parts: one that recognizes stimuli, and one that produces

a response for a given stimulus. In Eliza, the first part is the pattern matcher. I haven't spoken much about that part of the program in this description because I discussed it at length in the last project. But in fact `match` and its subprocedures are a substantial part of the complete `doctor` program. The second part, the one that produces responses, is `dorule` and `reconstruct`.

Eliza does not represent a very sophisticated form of stimulus-response theory because it leaves out the idea of *learning*. In Eliza, the responses are all provided in advance, as part of the script. People can develop new responses over time, and behaviorist theory has a lot to say about exactly what the rules are that govern such learning. (That's what education is, to a behaviorist: learning new responses to stimuli.) Since the script for Eliza is stored in lists, and those lists can be manipulated by the program, it would be possible to modify the rules of the program so that it can learn new rules while it's running. For example, if the user types something like "What are you talking about?" then the program could decide that its previous response was inappropriate. It would learn to avoid that response next time. You might like to think about how to design such an extension to the project as presented here.

Researchers who, unlike Weizenbaum, are deliberately trying to model theories of human psychology generally use a more complicated program structure. For example, experiments measuring the reaction time of human beings in different situations seem to indicate that people have a short-term memory and a long-term memory. The former may hold the telephone number you're dialing right now, for instance, while the latter holds all the telephone numbers of all your friends. Short-term memory is faster than long-term, but much smaller; you can only remember a few things at a time in it. Computers do not inherently have these two kinds of memory; they're really good at remembering many things *and* finding them quickly. But cognitive scientists write programs that deliberately limit the computer's ability to remember things quickly, trying to model the inner structure of the brain in this way.

## Property Lists

The response rules, memory rules, translations, and priorities that make up the script are all stored in the form of property lists. Each keyword has a property list. For example, the property list for the word `my` looks like this:

```
[priority 2
 translation your
 rules [[# your # !a:familyp #b] g1 [# your &stuff] g2 [#] g3]
```

```
 g1 [[Tell me more about your family.]
     [Who else in your family :b?]
     [Your :a?]
     [What else comes to mind when you think of your :a?]]
 g2 [[Your :stuff?]
     [Why do you say your :stuff?]
     [Does that suggest anything else which belongs to you?]
     [Is it important to you that your :stuff?]]
 g3 [newkey]
 memr [[# your &stuff] g4]
 g4 [[Earlier you said your :stuff.]
     [But your :stuff.]
     [Does that have anything to do with your statement about :stuff?]]]
```

Remember that a property list contains pairs of members; the odd numbered members are names of properties, and the even numbered members are the values of those properties. The word `my` has properties named `priority`, `translation`, `rules`, `g1`, `g2`, `g3`, `memr`, and `g4`. The `priority` and `translation` properties are straightforward. The `rules` and `memr` properties have as their values lists in which the odd numbered members are patterns and the even numbered members are names of other properties. These other properties contain the lists of responses. The names of these secondary properties are arbitrary and are generated by the program.

To create these property lists, I used `pprop` directly for some of the properties, but wrote setup procedures to help with the more complicated parts. Here are the instructions that contribute to this property list.

```
pprop "my "priority 2
pprop "my "translation "your
addrule "my [# your # !a:familyp #b]
  [[Tell me more about your family.]
   [Who else in your family :b?]
   [Your :a?]
   [What else comes to mind when you think of your :a?]]
addrule "my [# your &stuff]
  [[Your :stuff?]
   [Why do you say your :stuff?]
   [Does that suggest anything else which belongs to you?]
   [Is it important to you that your :stuff?]]
addrule "my [#] [newkey]
addmemr "my [# your &stuff]
  [[Earlier you said your :stuff.]
   [But your :stuff.]
   [Does that have anything to do with your statement about :stuff?]]
```

In general, the order in which properties are added to the list doesn't matter. However, the order of the `addrule` instructions *does* matter, because the rule that's added first is the one that `checkrules` tries first. It's important, therefore, that the rules go from most specific pattern to least specific pattern. In this case, the first pattern checks for a remark about a member of the user's family; the second checks for a remark about some other object or characteristic belonging to the user; and the third is a catch-all pattern just in case the other two fail.

## Generated Symbols

The procedures `addrule` and `addmemr` are very similar, since the `rules` and `memr` properties are similar in format.

```
to addrule :word :pattern :results
localmake "propname gensym
pprop :word "rules (sentence gprop :word "rules list :pattern :propname)
pprop :word :propname :results
end

to addmemr :word :pattern :results
localmake "propname gensym
pprop :word "memr (sentence gprop :word "memr list :pattern :propname)
pprop :word :propname :results
end
```

Each of these procedures uses a local variable `propname` to contain the name of the response property, a "generated symbol" or *gensym*. These are the words like `g3` in the example above. Each procedure carries out two `pprop` instructions. The first appends a new pattern and a new gensym to the previous value of the `rules` or `memr` property; the second creates a new property with the gensym as its name and the response (or memory) list as its value. `Gensym` is a Berkeley Logo library procedure.

## Modification of List Structure

Why are generated symbols needed in this program at all? In the Lisp version of Doctor, property lists are still used, but the entire collection of rules is one big list, the value of the property `rules`. It's as if the Logo property list looked like this:

```
[priority 2
 translation your
 rules
   [[# your # !a:familyp #b]
      [[Tell me more about your family.]
       [Who else in your family :b?]
       [Your :a?]
       [What else comes to mind when you think of your :a?]]
    [# your &stuff]
      [[Your :stuff?]
       [Why do you say your :stuff?]
       [Does that suggest anything else which belongs to you?]
       [Is it important to you that your :stuff?]]
    [#]
      [newkey]]
 memr
   [[# your &stuff]
      [[Earlier you said your :stuff.]
       [But your :stuff.]
       [Does that have anything to do with your statement
        about :stuff?]]]
]
```

I chose not to use one big list of rules in the Logo version. In Lisp (and in Berkeley Logo, but not in the versions of Logo I had available when writing the first edition), it's possible to change one of the members of a list without recopying the rest of the list. Without that capability, it's better to divide the rules into separate, smaller lists, so that only a little recopying is needed to change one.

Each pattern has several responses associated with it. When the program matches a particular pattern, it does not choose a response at random. Instead, it rotates through the list of responses in order. That is, it uses the first response first, then the second, and so on until the end of the list; if another response is needed, it starts over at the beginning of the list. This strict rotation is sometimes important because some of the responses say things like "I already told you that..."

The way the program keeps track of the rotation of the responses for a given rule is that it actually changes the response list so that what used to be the first response is moved to the end. Thus, `dorule` contains the instructions

```
localmake "print first gprop :keyword :rule
pprop :keyword :rule lput :print butfirst gprop :keyword :rule
```

The first of these instructions extracts the first response from the list of responses. The second one replaces the list of responses with a new list, in which the old first response is `lput` behind the remaining ones.

What if the rules were one big list? To see what would be required, let's look at a smaller list, in which it will be easier to follow what needs to be changed. Suppose some word's `rules` property had as its value this list:

```
[1 [A B C] 2 [D E F] 3 [G H I]]
```

In this example, the numbers represent patterns, while the letters represent responses. Now suppose that the program finds a match for pattern number `2`. It should then issue the response `D`. Then it should rotate the three responses associated with pattern `2` so that the new `rules` property is

```
[1 [A B C] 2 [E F D] 3 [G H I]]
```

The only way to do this in most versions of Logo is to construct a new copy of the entire list. Here is a way you could write such a program:

```
to rotate :keyword :pattern
pprop :keyword "rules (rotate1 :pattern gprop :keyword "rules)
end

to rotate1 :pattern :rules
if emptyp :rules [output []]
if equalp :pattern first :rules
   [output sentence (list :pattern rotate2 first butfirst :rules)
                    (butfirst butfirst :rules)]
output sentence (list first :rules first butfirst :rules) ~
                (rotate1 :pattern butfirst butfirst :rules)
end

to rotate2 :list
output lput first :list butfirst :list
end
```

You'd use this program with an instruction like

```
rotate "word 2
```

where `2` represents the pattern in the example above.

The trouble with this approach is that it's slow. It does a lot of `list` and `sentence` operations to reconstruct the modified list. More importantly, the *entire* list must be copied, even though only one rule is to be modified.

In Lisp, and in Berkeley Logo, there are primitive commands that can be used to change the contents of a list without recopying the unchanged parts. In Berkeley Logo they are called `.setfirst` and `.setbutfirst`; using them, we could write `rotate` this way:

```
to rotate :keyword :pattern
rotate1 :pattern (gprop :keyword "rules)
end

to rotate1 :pattern :rules
if emptyp :rules [stop]
ifelse equalp :pattern first :rules ~
       [.setfirst (butfirst :rules) (rotate2 first butfirst :rules)]
       [rotate1 :pattern butfirst butfirst :rules]
end
```

(I'll leave `rotate2` the same as in the earlier version, for now.) This is a tricky sort of procedure. Here's a trace of how it might be used:

```
rotate "word 2
  rotate1 2 [1 [A B C] 2 [D E F] 3 [G H I]]
    rotate1 2 [2 [D E F] 3 [G H I]]
```

In the lower-level invocation of `rotate1`, the `equalp` test outputs `true`, so the `ifelse` instruction evaluates its second input. This is equivalent to the instruction

```
.setfirst [[D E F] 3 [G H I]] (rotate2 [D E F])
```

or

```
.setfirst [[D E F] 3 [G H I]] [E F D]
```

To understand what this means, you must realize that the primitive operation `butfirst` does not make a *copy* of the butfirst of its input. Instead, the list output by `butfirst` is actually part of the list that is its input—they share the same cells in the computer's memory. Therefore, to change something in the `butfirst` also changes the larger list. The `setfirst` instruction ends up changing the `rules` property of the word `word` even though there is no explicit `pprop` to change the property.

If you're not a Lisp programmer, this probably seems like magic. It certainly violates some of the rules you've learned about the evaluation of Logo instructions. For example, if you actually typed the instruction

```
.setfirst [[D E F] 3 [G H I]] [E F D]
```

explicitly at top level, it would *not* change the property list of `word`, because the list that `setfirst` modifies would *not* be part of that property list, even though it has the same members. It's only because `setfirst`'s input is derived from that property list by a series of `butfirst` operations that they share the same memory.

Do you find this confusing? The original designers of Logo chose not to include `.setfirst` in the language because it *is* hard to understand, and because it can produce some very strange results if you're not careful with it. For example, consider these instructions:

```
make "c [x y]
.setfirst (butfirst :c) (:c)
```

This `.setfirst` instruction will produce a *circular list*, one that contains itself as a member. If you try to print `:c`, you'll see something like

```
[x [x [x [x [x [x [x [x [x [x [x [x [x [x ...
```

going on forever.

Once we have these list modification tools, even the implicit recopying done by `lput` can be avoided. Here's a more efficient version of `rotate1`, but it's really tricky to understand and it isn't a technique that I recommend:

```
to rotate1 :pattern :rules
if emptyp :rules [stop]
ifelse equalp :pattern first :rules ~
       [rotate2 butfirst :rules]
       [rotate1 :pattern butfirst butfirst :rules]
end

to rotate2 :rulelist
localmake "firstresponse first :rulelist
localmake "restresponses butfirst :firstresponse
.setfirst :rulelist :restresponses
.setbutfirst :firstresponse []
while [not emptyp butfirst :restresponses] ~
      [make "restresponses butfirst :restresponses]
.setbutfirst :restresponses :firstresponse
end
```

In `rotate2`, the `.setfirst` instruction removes the first response from the head of the list of responses; the two `.setbutfirst` instructions "splice" that first response back into the list at the end, following what used to be the last response.

Leaving `.setfirst` out of Logo was a controversial decision. Some people take the position that, as a "language for learners," Logo should not include mechanisms for which we can't provide an easy-to-follow metaphor; it's counterproductive for the language to encourage you to think in terms of what's where in memory. Other people refer to this idea scornfully as "protecting the user from himself," arguing that if a mechanism is useful it should be provided even though it's error-prone.

In any case, since Logo didn't have `.setfirst` and I didn't want the `doctor` program to be slowed down by having to recopy the `rules` property all the time, I decided to make each response list a separate property, so that each response list can be modified independently of the others. That's the reason for the gensym property names: so that `dorule` can rotate the responses for a particular pattern without disturbing the responses for other patterns. I could have changed this in the Berkeley Logo version, but it didn't seem worthwhile; using names for the rules is a little inelegant but doesn't hurt the program's efficiency.

## Linguistic Structure

Because it treats a sentence as simply a string of words, Eliza is limited in its linguistic sophistication. For example, the Doctor script has this pattern associated with the keyword `I`:

```
[# you are # !stuff:in [sad unhappy depressed sick] #]
```

(Remember that the pattern is matched against the input sentence after translation, so the words `you are` in the pattern really match a sentence containing the words `I am`.) The purpose of the pattern is to match a sentence like

```
I think I am really depressed because Susan doesn't like me.
```

The response of the program might be

```
I'm sorry to hear you are depressed.
```

The `#` between `are` and `!stuff` in the pattern is meant to catch adverbs like the word `really` in the example I just gave. But it could also "absorb" some more structurally important parts of a sentence:

```
I am sure that I would be depressed if she left me.
```

This sentence matches the pattern, but it doesn't really fit the intent of the pattern. The person who types this sentence is not saying "I am depressed" at all.

The trouble is that the string of words "sure that I would be" is not equivalent to an adverb. In fact, these words do not form a phrase at all. The program is making a grammatical error by connecting the word `depressed` with the word `am` as a predicate adjective. To avoid such errors, it's not good enough to have more and more detailed patterns to match. You can't anticipate every possible string of words by that technique. Instead, the program would have to impose a tree structure on the sentence, sort of like what you did in diagraming sentences in elementary school. The true structure of this sentence is something like

```
[[subject I] [predicate [[verb am] [nominative [[adjective sure]
   [adverb [clause [[connective that] [subject I] [verb [would be] ...
```

and so on. (Actually, I've just made up this structure to illustrate the idea, and it's not very realistic. I've tried too hard to preserve the order of the words in the original sentence. A more practical structure would probably center on the verb in each clause, and have subordinate slots for the subject, object, and so on. A connective like "that" might just be thrown away completely; the purpose served by such words in spoken text would instead be filled by the very sublist organization itself.) People have in fact written several computer programs that transform English sentences into a structured representation. It's very hard to do a perfect job, though, because of problems like homonyms: the word "like" can be a verb (I like ice cream) or a preposition (I want to be like my big brother).

## Further Explorations

There are three main directions in which you can explore the territory that this project begins. First, you can try to refine the existing Doctor script, so that it does a better job within the same general framework. Whenever you get an inappropriate response from `doctor`, see if you can think of a new rule that would solve that case without messing up other sentences.

A second possibility would be to write an entirely new Eliza script, so that instead of being a doctor the program can carry on some different sort of conversation. How about taking orders in a fast food restaurant? Answering questions from some data base about presidents or baseball players?

The third direction would be to abandon Eliza and look into some of the other approaches to understanding and generating English sentences that have been developed.

## Program Listing

The procedures from the pattern matcher of Chapter 7 are included in this program, but they are not listed again here.

```
to doctor
local [text sentence stuff a b c rules keywords memory]
make "memory []
print [Hello, I am the doctor. What can I do for you?]
print [Please end your remarks with an empty line.]
print []
loop
end

;; Controlling the conversation

to loop
make "text tokenize getstuff []
make "sentence getsentence :text
analyze :sentence :keywords
print []
loop
end

;; Reading and preparing the input

to getstuff :stuff
localmake "line readlist
if emptyp :line [output :stuff]
output getstuff sentence :stuff :line
end

to tokenize :text
output map.se [tokenword ? "] :text
end

to tokenword :word :out
if emptyp :word [output :out]
if memberp first :word [, " ] [output tokenword butfirst :word :out]
if memberp first :word [. ? ! |;|] [output sentence :out ".]
output tokenword butfirst :word word :out first :word
end

to getsentence :text
make "keywords []
output getsentence1 decapitalize :text []
end
```

```
to getsentence1 :text :out
if emptyp :text [output :out]
if equalp first :text ". ~
   [ifelse emptyp :keywords ~
           [output getsentence1 decapitalize butfirst :text []] [output :out]]
checkpriority first :text
output getsentence1 butfirst :text sentence :out translate first :text
end

to decapitalize :text
if emptyp :text [output []]
output fput lowercase first :text butfirst :text
end

to checkpriority :word
localmake "priority gprop :word "priority
if emptyp :priority [stop]
if emptyp :keywords [make "keywords ( list :word ) stop]
ifelse :priority > ( gprop first :keywords "priority ) ~
       [make "keywords fput :word :keywords] ~
       [make "keywords lput :word :keywords]
end

to translate :word
localmake "new gprop :word "translation
output ifelse emptyp :new [:word] [:new]
end

;; Choosing the rule and replying

to analyze :sentence :keywords
local [rules keyword]
if emptyp :keywords [norules stop]
make "keyword first :keywords
make "rules gprop :keyword "rules
if wordp first :rules ~
   [make "keyword first :rules make "rules gprop :keyword "rules]
checkrules :keyword :rules
end

to checkrules :keyword :rules
if not match first :rules :sentence ~
   [checkrules :keyword butfirst butfirst :rules stop]
dorule first butfirst :rules
end
```

```
to dorule :rule
localmake "print first gprop :keyword :rule
pprop :keyword :rule lput :print butfirst gprop :keyword :rule
if equalp :print "newkey [analyze :sentence butfirst :keywords stop]
if wordp :print [checkrules :print gprop :print "rules stop]
if equalp first :print "pre ~
   [analyze reconstruct first butfirst :print butfirst butfirst :print stop]
print capitalize reconstruct :print
memory :keyword :sentence
end

to reconstruct :sentence
if emptyp :sentence [output []]
if not equalp ": first first :sentence ~
   [output fput first :sentence reconstruct butfirst :sentence]
output sentence reword first :sentence reconstruct butfirst :sentence
end

to reword :word
if memberp last :word [. ? ,] [output addpunct reword butlast :word last :word]
output thing butfirst :word
end

to addpunct :stuff :char
if wordp :stuff [output word :stuff :char]
if emptyp :stuff [output :char]
output sentence butlast :stuff word last :stuff :char
end

to capitalize :text
if emptyp :text [output []]
output fput (word uppercase first first :text butfirst first :text) butfirst :text
end

to memory :keyword :sentence
local [rules rule name]
make "rules gprop :keyword "memr
if emptyp :rules [stop]
if not match first :rules :sentence [stop]
make "name last :rules
make "rules gprop :keyword :name
make "rule first :rules
pprop :keyword :name lput :rule butfirst :rules
make "memory fput reconstruct :sentence :memory
end

to norules
ifelse :memflag [usememory] [lastresort]
make "memflag not :memflag
end
```

```
to lastresort
print first :lastresort
make "lastresort lput first :lastresort butfirst :lastresort
end

to usememory
if emptyp :memory [lastresort stop]
print capitalize first :memory
make "memory butfirst :memory
end

;; Predicates for patterns

to beliefp :word
output not emptyp gprop :word "belief
end

to familyp :word
output not emptyp gprop :word "family
end

;; Procedures for adding to the script

to addrule :word :pattern :results
localmake "propname gensym
pprop :word "rules (sentence gprop :word "rules list :pattern :propname)
pprop :word :propname :results
end

to addmemr :word :pattern :results
localmake "propname gensym
pprop :word "memr (sentence gprop :word "memr list :pattern :propname)
pprop :word :propname :results
end

;; data

make "gensym.number 80

make "lastresort [[I am not sure I understand you fully.] [Please go on.]
                  [What does that suggest to you?]
                  [Do you feel strongly about discussing such things?]]

make "memflag "false

pprop "alike "priority 10
pprop "alike "rules [dit]
```

```
pprop "always "priority 1
pprop "always "rules [[#] g69]
pprop "always "g69 [[Can you think of a specific example?] [When?]
                    [What incident are you thinking of?]
                    [Really, always?] [What if this never happened?]]

pprop "am "priority 0
pprop "am "translation "are
pprop "am "rules [[# are you #stuff] g18 [#] g19]
pprop "am "g18 [[Do you believe you are :stuff?] [Would you want to be :stuff?]
                [You wish I would tell you you are :stuff.]
                [What would it mean if you were :stuff?] how]
pprop "am "g19 [[Why do you say "am"?] [I don't understand that.]]

pprop "are "priority 0
pprop "are "rules [[#a there are #b you #c] g20 [# there are &stuff] g21
                   [# are I #stuff] g22 [are #] g23 [# are #stuff] g24]
pprop "are "g20 [[pre [:a there are :b] are]]
pprop "are "g21 [[What makes you think there are :stuff?]
                 [Do you usually consider :stuff?]
                 [Do you wish there were :stuff?]]
pprop "are "g22 [[Why are you interested in whether I am :stuff or not?]
                 [Would you prefer if I weren't :stuff?]
                 [Perhaps I am :stuff in your fantasies.]
                 [Do you sometimes think I am :stuff?] how]
pprop "are "g23 [how]
pprop "are "g24 [[Did you think they might not be :stuff?]
                 [Would you like it if they were not :stuff?]
                 [What if they were not :stuff?] [Possibly they are :stuff.]]

pprop "ask "priority 0
pprop "ask "rules [[# you ask #] g77 [# you ! asking #] g78 [# I #] g79 [#] g80]
pprop "ask "g77 [how]
pprop "ask "g78 [how]
pprop "ask "g79 [you]
pprop "ask "g80 [newkey]

pprop "because "priority 0
pprop "because "rules [[#] g64]
pprop "because "g64 [[Is that the real reason?]
                     [Don't any other reasons come to mind?]
                     [Does that reason seem to explain anything else?]
                     [What other reasons might there be?]
                     [You're not concealing anything from me, are you?]]

pprop "believe "belief "true

pprop "bet "belief "true

pprop "brother "family "true
```

```
pprop "can "priority 0
pprop "can "rules [[# can I #stuff] g58 [# can you #stuff] g59 [#] g60]
pprop "can "g58 [[You believe I can :stuff, don't you?] how
                [You want me to be able to :stuff.]
                [Perhaps you would like to be able to :stuff yourself.]]
pprop "can "g59 [[Whether or not you can :stuff depends more on you than on me.]
                [Do you want to be able to :stuff?]
                [Perhaps you don't want to :stuff.] how]
pprop "can "g60 [how newkey]

pprop "cant "translation "can't

pprop "certainly "priority 0
pprop "certainly "rules [yes]

pprop "children "family "true

pprop "computer "priority 50
pprop "computer "rules [[#] g17]
pprop "computer "g17 [[Do computers worry you?]
                      [Why do you mention computers?]
                      [What do you think machines have to do with your problem?]
                      [Don't you think computers can help people?]
                      [What about machines worries you?]
                      [What do you think about machines?]]

pprop "computers "priority 50
pprop "computers "rules [computer]

pprop "dad "translation "father
pprop "dad "family "true

pprop "daddy "translation "father
pprop "daddy "family "true

pprop "deutsch "priority 0
pprop "deutsch "rules [[#] g15]
pprop "deutsch "g15 [[I'm sorry, I speak only English.]]

pprop "dit "rules [[#] g72]
pprop "dit "g72 [[In what way?] [What resemblance do you see?]
                [What does that similarity suggest to you?]
                [What other connections do you see?]
                [What do you suppose that resemblance means?]
                [What is the connection, do you suppose?]
                [Could there really be some connection?] how]

pprop "dont "translation "don't
```

```
pprop "dream "priority 3
pprop "dream "rules [[#] g9]
pprop "dream "g9 [[What does that dream suggest to you?] [Do you dream often?]
                 [What persons appear in your dreams?]
                 [Don't you believe that dream has something to do
                  with your problem?]
                 [Do you ever wish you could flee from reality?] newkey]

pprop "dreamed "priority 4
pprop "dreamed "rules [[# you dreamed #stuff] g7 [#] g8]
pprop "dreamed "g7 [[Really :stuff?]
                    [Have you ever fantasied :stuff while you were awake?]
                    [Have you dreamed :stuff before?] dream newkey]
pprop "dreamed "g8 [dream newkey]

pprop "dreams "translation "dream
pprop "dreams "priority 3
pprop "dreams "rules [dream]

pprop "dreamt "translation "dreamed
pprop "dreamt "priority 4
pprop "dreamt "rules [dreamed]

pprop "espanol "priority 0
pprop "espanol "rules [deutsch]

pprop "everybody "priority 2
pprop "everybody "rules [everyone]

pprop "everyone "priority 2
pprop "everyone "rules [[# !a:in [everyone everybody nobody noone] #] g68]
pprop "everyone "g68 [[Really, :a?] [Surely not :a.]
                      [Can you think of anyone in particular?]
                      [Who, for example?]
                      [You are thinking of a very special person.]
                      [Who, may I ask?] [Someone special perhaps.]
                      [You have a particular person in mind, don't you?]
                      [Who do you think you're talking about?]
                      [I suspect you're exaggerating a little.]]

pprop "father "family "true

pprop "feel "belief "true

pprop "francais "priority 0
pprop "francais "rules [deutsch]

pprop "hello "priority 0
pprop "hello "rules [[#] g16]
pprop "hello "g16 [[How do you do. Please state your problem.]]
```

```
pprop "how "priority 0
pprop "how "rules [[#] g63]
pprop "how "g63 [[Why do you ask?] [Does that question interest you?]
                 [What is it you really want to know?]
                 [Are such questions much on your mind?]
                 [What answer would please you most?] [What do you think?]
                 [What comes to your mind when you ask that?]
                 [Have you asked such questions before?]
                 [Have you asked anyone else?]]

pprop "husband "family "true

pprop "i "priority 0
pprop "i "translation "you
pprop "i "rules [[# you !:in [want need] #stuff] g32
                 [# you are # !stuff:in [sad unhappy depressed sick] #] g33
                 [# you are # !stuff:in [happy elated glad better] #] g34
                 [# you was #] g35 [# you !:beliefp you #stuff] g36
                 [# you # !:beliefp # i #] g37 [# you are #stuff] g38
                 [# you !:in [can't cannot] #stuff] g39
                 [# you don't #stuff] g40 [# you feel #stuff] g41
                 [# you #stuff i #] g42 [#stuff] g43]
pprop "i "g32 [[What would it mean to you if you got :stuff?]
               [Why Do you want :stuff?] [Suppose you got :stuff soon.]
               [What if you never got :stuff?]
               [What would getting :stuff mean to you?] [You really want :stuff.]
               [I suspect you really don't want :stuff.]]
pprop "i "g33 [[I'm sorry to hear you are :stuff.]
               [Do you think coming here will help you not to be :stuff?]
               [I'm sure it's not pleasant to be :stuff.]
               [Can you explain what made you :stuff?] [Please go on.]]
pprop "i "g34 [[How have I helped you to be :stuff?]
               [Has your treatment made you :stuff?]
               [What makes you :stuff just now?]
               [Can you explain why you are suddenly :stuff?]
               [Are you sure?] [What do you mean by :stuff?]]
pprop "i "g35 [was]
pprop "i "g36 [[Do you really think so?] [But you are not sure you :stuff.]
               [Do you really doubt you :stuff?]]
pprop "i "g37 [you]
pprop "i "g38 [[Is it because you are :stuff that you came to me?]
               [How long have you been :stuff?]
               [Do you believe it normal to be :stuff?]
               [Do you enjoy being :stuff?]]
pprop "i "g39 [[How do you know you can't :stuff?] [Have you tried?]
               [Perhaps you could :stuff now.]
               [Do you really want to be able to :stuff?]]
pprop "i "g40 [[Don't you really :stuff?] [Why don't you :stuff?]
               [Do you wish to be able to :stuff?] [Does that trouble you?]]
```

```
pprop "i "g41 [[Tell me more about such feelings.] [Do you often feel :stuff?]
              [Do you enjoy feeling :stuff?]
              [Of what does feeling :stuff remind you?]]
pprop "i "g42 [[Perhaps in your fantasy we :stuff each other.]
              [Do you wish to :stuff me?] [You seem to need to :stuff me.]
              [Do you :stuff anyone else?]]
pprop "i "g43 [[You say :stuff.] [Can you elaborate on that?]
              [Do you say :stuff for some special reason?]
              [That's quite interesting.]]

pprop "i'm "priority 0
pprop "i'm "translation "you're
pprop "i'm "rules [[# you're #stuff] g31]
pprop "i'm "g31 [[pre [you are :stuff] I]]

pprop "if "priority 3
pprop "if "rules [[#a if #b had #c] g5 [# if #stuff] g6]
pprop "if "g5 [[pre [:a if :b might have :c] if]]
pprop "if "g6 [[Do you think it's likely that :stuff?] [Do you wish that :stuff?]
              [What do you think about :stuff?]]

pprop "is "priority 0
pprop "is "rules [[&a is &b] g61 [#] g62]
pprop "is "g61 [[Suppose :a were not :b.] [Perhaps :a really is :b.]
               [Tell me more about :a.]]
pprop "is "g62 [newkey]

pprop "italiano "priority 0
pprop "italiano "rules [deutsch]

pprop "like "priority 10
pprop "like "rules [[# !:in [am is are was] # like #] g70 [#] g71]
pprop "like "g70 [dit]
pprop "like "g71 [newkey]

pprop "machine "priority 50
pprop "machine "rules [computer]

pprop "machines "priority 50
pprop "machines "rules [computer]

pprop "maybe "priority 0
pprop "maybe "rules [perhaps]

pprop "me "translation "you

pprop "mom "translation "mother
pprop "mom "family "true
```

```
pprop "mommy "translation "mother
pprop "mommy "family "true

pprop "mother "family "true

pprop "my "priority 2
pprop "my "translation "your
pprop "my "rules [[# your # !a:familyp #b] g55 [# your &stuff] g56 [#] g57]
pprop "my "g55 [[Tell me more about your family.] [Who else in your family :b?]
                [Your :a?] [What else comes to mind when you think of your :a?]]
pprop "my "g56 [[Your :stuff?] [Why do you say your :stuff?]
                [Does that suggest anything else which belongs to you?]
                [Is it important to you that your :stuff?]]
pprop "my "g57 [newkey]
pprop "my "memr [[# your &stuff] g12]
pprop "my "g12 [[Earlier you said your :stuff.] [But your :stuff.]
                [Does that have anything to do with your statement about :stuff?]]

pprop "myself "translation "yourself

pprop "name "priority 15
pprop "name "rules [[#] g14]
pprop "name "g14 [[I am not interested in names.]
                  [I've told you before I don't care about names\;
                   please continue.]]

pprop "no "priority 0
pprop "no "rules [[no] g53 [#] g54]
pprop "no "g53 [xxyyzz [pre [x no] no]]
pprop "no "g54 [[Are you saying "no" just to be negative?]
                [You are being a bit negative.] [Why not?] [Why "no"?] newkey]

pprop "nobody "priority 2
pprop "nobody "rules [everyone]

pprop "noone "priority 2
pprop "noone "rules [everyone]

pprop "perhaps "priority 0
pprop "perhaps "rules [[#] g13]
pprop "perhaps "g13 [[You don't seem quite certain.] [Why the uncertain tone?]
                     [Can't you be more positive?] [You aren't sure.]
                     [Don't you know?]]
```

```
pprop "problem "priority 5
pprop "problem "rules [[#a !b:in [is are] your !c:in [problem problems] #] g73
                       [# your !a:in [problem problems] !b:in [is are] #c] g74
                       [#] g75]
pprop "problem "g73 [[:a :b your :c.] [Are you sure :a :b your :c?]
                     [Perhaps :a :b not your real :c.]
                     [You think you have problems?]
                     [Do you often think about :a?]]
pprop "problem "g74 [[Your :a :b :c?] [Are you sure your :a :b :c?]
                     [Perhaps your real :a :b not :c.]
                     [You think you have problems?]]
pprop "problem "g75 [[Please continue, this may be interesting.]
                     [Have you any other problems you wish to discuss?]
                     [Perhaps you'd rather change the subject.]
                     [You seem a bit uneasy.] newkey]
pprop "problem "memr [[#stuff is your problem #] g76]
pprop "problem "g76 [[Earlier you mentioned :stuff.]
                     [Let's talk further about :stuff.]
                     [Tell me more about :stuff.]
                     [You haven't mentioned :stuff for a while.]]

pprop "problems "priority 5
pprop "problems "rules [problem]

pprop "remember "priority 5
pprop "remember "rules [[# you remember #stuff] g2
                        [# do I remember #stuff] g3 [#] g4]
pprop "remember "g2 [[Do you often think of :stuff?]
                     [Does thinking of :stuff bring anything else to mind?]
                     [What else do you remember?]
                     [Why do you remember :stuff just now?]
                     [What in the present situation reminds you of :stuff?]]
pprop "remember "g3 [[Did you think I would forget :stuff?]
                     [Why do you think I should recall :stuff now?]
                     [What about :stuff?] what [You mentioned :stuff.]]
pprop "remember "g4 [newkey]

pprop "same "priority 10
pprop "same "rules [dit]

pprop "sister "family "true

pprop "sorry "priority 0
pprop "sorry "rules [[#] g1]
pprop "sorry "g1 [[Please don't apologize.] [Apologies are not necessary.]
                  [What feelings do you have when you apologize?]
                  [I've told you that apologies are not required.]]

pprop "svenska "priority 0
pprop "svenska "rules [deutsch]
```

```
pprop "think "belief "true

pprop "was "priority 2
pprop "was "rules [[# was you #stuff] g26 [# you was #stuff] g27
                   [# was I #stuff] g28 [#] g29]
pprop "was "g26 [[What if you were :stuff?] [Do you think you were :stuff?]
                 [Were you :stuff?] [What would it mean if you were :stuff?]
                 [What does " :stuff " suggest to you?] how]
pprop "was "g27 [[Were you really?] [Why do you tell me you were :stuff now?]
                 [Perhaps I already knew you were :stuff.]]
pprop "was "g28 [[Would you like to believe I was :stuff?]
                 [What suggests that I was :stuff?] [What do you think?]
                 [Perhaps I was :stuff.] [What if I had been :stuff?]]
pprop "was "g29 [newkey]

pprop "we "translation "you
pprop "we "priority 0
pprop "we "rules [I]

pprop "were "priority 0
pprop "were "translation "was
pprop "were "rules [was]

pprop "what "priority 0
pprop "what "rules [[!:in [what where] #] g10 [# !a:in [what where] #b] g11]
pprop "what "g10 [how]
pprop "what "g11 [[Tell me about :a :b.] [:a :b?]
                  [Do you want me to tell you :a :b?]
                  [Really.] [I see.] newkey]

pprop "where "priority 0
pprop "where "rules [how]

pprop "why "priority 0
pprop "why "rules [[# why don't I #stuff] g65
                   [# why can't you #stuff] g66 [#] g67]
pprop "why "g65 [[Do you believe I don't :stuff?]
                 [Perhaps I will :stuff in good time.]
                 [Should you :stuff yourself?] [You want me to :stuff?] how]
pprop "why "g66 [[Do you think you should be able to :stuff?]
                 [Do you want to be able to :stuff?]
                 [Do you believe this will help you to :stuff?]
                 [Have you any idea why you can't :stuff?] how]
pprop "why "g67 [[Why indeed?] [Why "why"?] [Why not?] how newkey]

pprop "wife "family "true

pprop "wish "belief "true

pprop "wont "translation "won't
```

```
pprop "xxyyzz "priority 0
pprop "xxyyzz "rules [[#] g50]
pprop "xxyyzz "g50 [[You're being somewhat short with me.]
                    [You don't seem very talkative today.]
                    [Perhaps you'd rather talk about something else.]
                    [Are you using monosyllables for some reason?] newkey]

pprop "yes "priority 0
pprop "yes "rules [[yes] g51 [#] g52]
pprop "yes "g51 [xxyyzz [pre [x yes] yes]]
pprop "yes "g52 [[You seem quite positive.] [You are sure.] [I see.]
                 [I understand.] newkey]

pprop "you "priority 0
pprop "you "translation "I
pprop "you "rules [[# I remind you of #] g44 [# I are # you #] g45
                   [# I # are #stuff] g46 [# I #stuff you] g47
                   [# I &stuff] g48 [#] g49]
pprop "you "g44 [dit]
pprop "you "g45 [newkey]
pprop "you "g46 [[What makes you think I am :stuff?]
                 [Does it please you to believe I am :stuff?]
                 [Perhaps you would like to be :stuff.]
                 [Do you sometimes wish you were :stuff?]]
pprop "you "g47 [[Why do you think I :stuff you?]
                 [You like to think I :stuff you, don't you?]
                 [What makes you think I :stuff you?] [Really, I :stuff you?]
                 [Do you wish to believe I :stuff you?]
                 [Suppose I did :stuff you. what would that mean?]
                 [Does someone else believe I :stuff you?]]
pprop "you "g48 [[We were discussing you, not me.] [Oh, I :stuff?]
                 [Is this really relevant to your problem?] [Perhaps I do :stuff.]
                 [Are you glad to know I :stuff?] [Do you :stuff?]
                 [What are your feelings now?]]
pprop "you "g49 [newkey]

pprop "you're "priority 0
pprop "you're "translation "I'm
pprop "you're "rules [[# I'm #stuff] g30]
pprop "you're "g30 [[pre [I are :stuff] you]]

pprop "your "priority 0
pprop "your "translation "my
pprop "your "rules [[# my #stuff] g25]
pprop "your "g25 [[Why are you concerned over my :stuff?]
                  [What about your own :stuff?]
                  [Are you worried about someone else's :stuff?]
                  [Really, my :stuff?]]

pprop "yourself "translation "myself
```

# 10 Iteration, Control Structures, Extensibility

In this chapter we're taking a tour "behind the scenes" of Berkeley Logo. Many of the built-in Logo procedures that we've been using all along are not, strictly speaking, primitive; they're written in Logo itself. When you invoke a procedure, if the Logo interpreter does not already know a procedure by that name, it automatically looks in a *library* of predefined procedures. For example, in Chapter 6 I used an operation called `gensym` that outputs a new, unique word each time it's invoked. If you start up a fresh copy of Logo you can try these experiments:

```
? po "gensym
I don't know how  to gensym
? show gensym
g1
? po "gensym
to gensym
if not namep "gensym.number [make "gensym.number 0]
make "gensym.number :gensym.number + 1
output word "g :gensym.number
end
```

The first interaction shows that `gensym` is not really a Logo primitive; the error message indicates that there is no such procedure. Then I invoked `gensym`, which made Berkeley Logo read its definition automatically from the library. Finally, once Logo has read the definition, I can print it out.

In particular, most of the tools we've used to carry out a computation repeatedly are not true Logo primitives: `for` for numeric iteration, `foreach` for list iteration, and `while` for predicate-based iteration are all library procedures. (The word *iteration* just means "repetition.") The only iteration mechanisms that are truly primitive in Logo are `repeat` and recursion.

Computers are good at doing things over and over again. They don't get bored or tired. That's why, in the real world, people use computers for things like sending out payroll checks and telephone bills. The first Logo instruction I showed you, in the first volume, was

```
repeat 50 [setcursor list random 75 random 20 type "Hi]
```

When you were first introduced to turtle graphics, you probably used an instruction like

```
repeat 360 [forward 1 right 1]
```

to draw a circle.

## Recursion as Iteration

The trouble with `repeat` is that it always does *exactly* the same thing repeatedly. In a real application, like those payroll checks, you want the computer to do *almost* the same thing each time but with a different person's name on each check. The usual way to program an almost-`repeat` in Logo is to use a recursive procedure, like this:

```
to polyspi :side :angle :number
if :number=0 [stop]
forward :side
right :angle
polyspi :side+1 :angle :number-1
end
```

This is a well-known procedure to draw a spiral. What makes it different from

```
repeat :number [forward :side right :angle]
```

is that the first input in the recursive invocation is `:side+1` instead of just `:side`. We've used a similar technique for almost-repetition in procedures like this one:

```
to multiply :letters :number
if equalp :number 0 [stop]
print :letters
multiply (word :letters first :letters) :number-1
end
```

Since recursion can express any repetitive computation, why bother inventing other iteration tools? The answer is that they can make programs easier to read. Recursion is such a versatile mechanism that the intention of any particular use of recursion may be hard to see. Which of the following is easier to read?

```
to fivesay.with.repeat :text
repeat 5 [print :text]
end
```

or

```
to fivesay.with.recursion :text
fivesay1 5 :text
end

to fivesay1 :times :text
if :times=0 [stop]
print :text
fivesay1 :times-1 :text
end
```

The version using `repeat` makes it obvious at a glance what the program wants to do; the version using recursion takes some thought. It can be useful to invent mechanisms that are intermediate in flexibility between `repeat` and recursion.

As a simple example, suppose that Logo did not include `repeat` as a primitive command. Here's how we could implement it using recursion:

```
to rep :count :instr
if :count=0 [stop]
run :instr
rep :count-1 :instr
end
```

(I've used the name `rep` instead of `repeat` to avoid conflict with the primitive version.) The use of `run` to carry out the given instructions is at the core of the techniques we'll use throughout this chapter.

## Numeric Iteration

`Polyspi` is an example of an iteration in which the value of a numeric variable changes in a uniform way. The instruction

```
polyspi 50 60 4
```

is equivalent to the series of instructions

```
forward 50 right 60
forward 51 right 60
forward 52 right 60
forward 53 right 60
```

As you know, we can represent the same instructions this way:

```
for [side 50 53] [forward :side right 60]
```

The `for` command takes two inputs, very much like `repeat`. The second input, like `repeat`'s second input, is a list of Logo instructions. The first input to `for` is different, though. It is a list whose first member is the name of a variable; the second member of the list must be a number (or a Logo expression whose value is a number), which will be the *initial* value of that variable; and the third member must be another number (or numeric expression), which will be the *final* value of the variable. In the example above, the variable name is `side`, the initial value is `50`, and the final value is `53`. If there is a fourth member in the list, it's the amount to add to the named variable on each iteration; if there is no fourth member, as in the example above, then the *step* amount is either 1 or −1, depending on whether the final value is greater than or less than the initial value.

As an example in which expressions are used instead of constant numeric values, here's the `polyspi` procedure using `for`:

```
to polyspi :start :angle :number
for [side :start [:start+:number-1]] [forward :side right :angle]
end
```

Most of the work in writing `for` is in evaluating the expressions that make up the first input list. Here is the program:

```
to for :values :instr
localmake "var first :values
local :var
localmake "initial run first butfirst :values
localmake "final run item 3 :values
localmake "step forstep
localmake "tester ~
          ifelse :step < 0 [[:value < :final]] [[:value > :final]]
forloop :initial
end
```

```
to forstep
if (count :values)=4 [output run last :values]
if :initial > :final [output -1]
output 1
end

to forloop :value
make :var :value
if run :tester [stop]
run :instr
forloop :value+:step
end
```

One slightly tricky part of this program is the instruction

```
local :var
```

near the beginning of `for`. The effect of this instruction is to make whatever variable is named by the first member of the first input local to `for`. As it turns out, this variable isn't given a value in `for` itself but only in its subprocedure `forloop`. (A *loop,* by the way, is a part of a program that is invoked repeatedly.) But I'm thinking of these three procedures as a unit and making the variable local to that whole unit. The virtue of this `local` instruction is that a program that uses `for` can invent variable names for `for` freely, without having to declare them local and without cluttering up the workspace with global variables. Also, it means that a procedure can invoke another procedure in the instruction list of a `for` without worrying about whether *that* procedure uses `for` itself. Here's the case I'm thinking of:

```
to a
for [i 1 5] [b]
end

to b
for [i 1 3] [print "foo]
end
```

Invoking `A` should print the word `foo` fifteen times: three times for each of the five invocations of `B`. If `for` didn't make `I` a local variable, the invocation of `for` within `B` would mess up the value of `I` in the outer `for` invoked by `A`. Got that?

Notice that the `make` instruction in `forloop` has `:var` as its first input, not `"var`. This instruction does not assign a new value to the variable `var`! Instead, it assigns a new value to the variable whose name is the value of `var`.

The version of `for` actually used in the Berkeley Logo library is a little more complicated than this one. The one shown here works fine as long as the instruction list input doesn't include `stop` or `output`, but it won't work for an example like the following. To check whether or not a number is prime, we must see if it is divisible by anything greater than 1 and smaller than the number itself:

```
to primep :num
for [trial 2 [:num-1]] [if divisiblep :num :trial [output "false]]
output "true
end

to divisiblep :big :small
output equalp remainder :big :small 0
end
```

This example will work in the Berkeley Logo `for`, but not in the version I've written in this chapter. The trouble is that the instruction

```
run :instr
```

in `forloop` will make `forloop` output `false` if a divisor is found, whereas we really want `primep` to output `false`! We'll see in Chapter 12 how to solve this problem.

## Logo: an Extensible Language

There are two ways to look at a program like `for`. You can take it apart, as I've been doing in these last few paragraphs, to see how it works inside. Or you can just think of it as an extension to Logo, an iteration command that you can use as you'd use `repeat`, without thinking about how it works. I think both of these perspectives will be valuable to you. As a programming project, `for` demonstrates some rather advanced Logo techniques. But you don't have to think about those techniques each time you use `for`. Instead you can think of it as a primitive, as we've been doing prior to this chapter.

The fact that you can extend Logo's vocabulary this way, adding a new way to control iteration that looks just like the primitive `repeat`, is an important way in which Logo is more powerful than toy programming languages like C++ or Pascal. C++ has several iteration commands built in, including one like `for`, but if you think of a new one, there's no way you can add it to the language. In Logo this kind of language extension is easy. For example, here is a preview of a programming project I'm going to develop later

in this chapter. Suppose you're playing with spirals, and you'd like to see what happens if you change the line length *and* the turning angle at the same time. That is, you'd like to be able to say

```
multifor [[size 50 100 5] [angle 50 100 10]] [forward :size right :angle]
```

and have that be equivalent to the series of instructions

```
forward 50 right 50
forward 55 right 60
forward 60 right 70
forward 65 right 80
forward 70 right 90
forward 75 right 100
```

`Multifor` should step each of its variables each time around, stopping whenever any of them hits the final value. This tool strikes me as too specialized and complicated to provide in the Logo library, but it seems very appropriate for certain kinds of project. It's nice to have a language in which I can write it if I need it.

## No Perfect Control Structures

Among enthusiasts of the Fortran family of programming languages (that is, all the languages in which you have to say ahead of time whether or not the value of some numeric variable will be an exact integer), there are fierce debates about the "best" control structure. (A *control structure* is a way of grouping instructions together, just as a *data structure* is a way of grouping data together. A list is a data structure. A procedure is a control structure. Things like `if`, `repeat`, and `for` are special control structures that group instructions in particular ways, so that a group of instructions can be evaluated conditionally or repeatedly.)

For example, all of the Fortran-derived languages have a control structure for numeric iteration, like my `for` procedure. But they differ in details. In some languages the iteration variable must be stepped by 1. In others the step value can be either 1 or –1. Still others allow any step value, as `for` does. Each of these choices has its defenders as the "best."

Sometimes the arguments are even sillier. When Fortran was first invented, its designers failed to make explicit what should happen if the initial value of an iteration variable is greater than the final value. That is, they left open the interpretation of a

Fortran `do` statement (that's what its numeric iteration structure is called) equivalent to this `for` instruction:

```
for [var 10 5 1] [print :var]
```

In this instruction I've specified a positive step (the only kind allowed in the Fortran `do` statement), but the initial value is greater than the final value. (What will `for` do in this situation?) Well, the first Fortran compiler, the program that translates a Fortran program into the "native" language of a particular computer, implemented `do` so that the statements controlled by the `do` were carried out once before the computer noticed that the variable's value was already too large. Years later a bunch of computer scientists decided that that behavior is "wrong"; if the initial value is greater than the final value, the statements shouldn't be carried out at all. This proposal for a "zero trip `do` loop" was fiercely resisted by old-timers who had by then written hundreds of programs that relied on the original behavior of `do`. Dozens of journal articles and letters to the editor carried on the battle.

The real moral of this story is that there is no right answer. The right control structure for *you* to use is the one that best solves *your* immediate problem. But only an extensible language like Logo allows you the luxury of accepting this moral. The Fortran people had to fight out their battle because they're stuck with whatever the standardization committee decides.

In the remainder of this chapter I'll present various kinds of control structures, each reflecting a different way of looking at the general idea of iteration.

## Iteration Over a List

Numeric iteration is useful if the problem you want to solve is about numbers, as in the `primep` example, or if some arbitrary number is part of the rules of a game, like the seven stacks of cards in solitaire. But in most Logo projects, it's more common to want to carry out a computation for each member of a list, and for that purpose we have the `foreach` control structure:

```
? foreach [chocolate [rum raisin] pumpkin] [print sentence [I like] ?]
I like chocolate
I like rum raisin
I like pumpkin
```

In comparing `foreach` with `for`, one thing you might notice is the use of the question mark to represent the varying datum in `foreach`, while `for` requires a user-specified variable name for that purpose. There's no vital reason why I used these different mechanisms. In fact, we can easily implement a version of `foreach` that takes a variable name as an additional input. Its structure will then look similar to that of `for`:

```
to named.foreach :var :data :instr
local :var
if emptyp :data [stop]
make :var first :data
run :instr
named.foreach :var (butfirst :data) :instr
end
```

```
? named.foreach "flavor [lychee [root beer swirl]] ~
                        [print sentence [I like] :flavor]
I like lychee
I like root beer swirl
```

Just as in the implementation of `for`, there is a recursive invocation for each member of the data input. We assign that member as the value of the variable named in the first input, and then we `run` the instructions in the third input.

In order to implement the version of `foreach` that uses question marks instead of named variables, we need a more advanced version of `run` that says "run these instructions, but using this value wherever you see a question mark." Berkeley Logo has this capability as a primitive procedure called `apply`. It takes two inputs, a *template* (an instruction list with question marks) and a list of values. The reason that the second input is a list of values, rather than a single value, is that `apply` can handle templates with more than one slot for values.

```
? apply [print ?+3] [5]
8
? apply [print word first ?1 first ?2] [Peter Dickinson]
PD
```

It's possible to write `apply` in terms of `run`, and I'll do that shortly. But first, let's just take advantage of Berkeley Logo's built-in `apply` to write a simple version of `foreach`:

```
to foreach :list :template
if emptyp :list [stop]
apply :template (list first :list)
foreach (butfirst :list) :template
end
```

`Apply`, like `run`, can be either a command or an operation depending on whether its template contains complete Logo instructions or a Logo expression. In this case, we are using `apply` as a command.

The version of `foreach` in the Berkeley Logo library can take more than one data input along with a multi-input template, like this:

```
? (foreach [John Paul George Ringo] [rhythm bass lead drums]
           [print (sentence ?1 "played ?2)]
John played rhythm
Paul played bass
George played lead
Ringo played drums
```

We can implement this feature, using a special notation in the title line of `foreach` to notify Logo that it accepts a variable number of inputs:

```
to foreach [:inputs] 2
foreach.loop (butlast :inputs) (last :inputs)
end

to foreach.loop :lists :template
if emptyp first :lists [stop]
apply :template firsts :lists
foreach.loop (butfirsts :lists) :template
end
```

First look at the title line of `foreach`. It tells Logo that the word `inputs` is a formal parameter—the name of an input. Because `:inputs` is inside square brackets, however, it represents not just one input, but any number of inputs in the invocation of `foreach`. The values of all those inputs are collected as a list, and that list is the value of `inputs`. Here's a trivial example:

```
to demo [:stuff]
print sentence [The first input is] first :stuff
print sentence [The others are] butfirst :stuff
end

? (demo "alpha "beta "gamma)
The first input is alpha
The others are beta gamma
```

As you know, the Logo procedures that accept a variable number of inputs have a *default* number that they accept without using parentheses; if you want to use more or fewer than that number, you must enclose the procedure name and its inputs in parentheses, as I've done here with the `demo` procedure. Most Logo primitives that accept a variable number of inputs have two inputs as their default number (for example, `sentence`, `sum`, `word`) but there are exceptions, such as `local`, which takes one input if parentheses are not used. When you write your own procedure with a single input name in brackets, its default number of inputs is zero unless you specify another number. `Demo`, for example, has zero as its default number. If you look again at the title line of `foreach`, you'll see that it ends with the number `2`; that tells Logo that `foreach` expects two inputs by default.

`Foreach` uses all but its last input as data lists; the last input is the template to be applied to the members of the data lists. That's why `foreach` invokes `foreach.loop` as it does, separating the two kinds of inputs into two variables.

Be careful when reading the definition of `foreach.loop`; it invokes procedures named `firsts` and `butfirsts`. These are not the same as `first` and `butfirst`! Each of them takes a *list of lists* as its input, and outputs a list containing the first members of each sublist, or all but the first members, respectively:

```
? show firsts [[a b c] [1 2 3] [y w d]]
[a 1 y]
? show butfirsts [[a b c] [1 2 3] [y w d]]
[[b c] [2 3] [w d]]
```

It would be easy to write `firsts` and `butfirsts` in Logo:

```
to firsts :list.of.lists
output map "first :list.of.lists
end

to butfirsts :list.of.lists
output map "butfirst :list.of.lists
end
```

but in fact Berkeley Logo provides these operations as primitives, because implementing them as primitives makes the iteration tools such as `foreach` and `map` (which, as we'll see, also uses them) much faster.

Except for the use of `firsts` and `butfirsts` to handle the multiple data inputs, the structure of `foreach.loop` is exactly like that of the previous version of `foreach` that only accepts one data list.

Like `for`, the version of `foreach` presented here can't handle instruction lists that include `stop` or `output` correctly.

## Implementing `Apply`

Berkeley Logo includes `apply` as a primitive, for efficiency, but we could implement it in Logo if necessary. In this section, so as not to conflict with the primitive version, I'll use the name `app` for my non-primitive version of `apply`, and I'll use the percent sign (`%`) as the placeholder in templates instead of question mark.

Here is a simple version of `app` that allows only one input to the template:

```
to app :template :input.value
run :template
end

to %
output :input.value
end
```

This is so simple that it probably seems like magic. `App` seems to do nothing but `run` its template as though it were an ordinary instruction list. The trick is that a template *is* an instruction list. The only unusual thing about a template is that it includes special symbols (`?` in the real `apply`, `%` in `app`) that represent the given value. We see now that those special symbols are really just ordinary names of procedures. The question mark (`?`) procedure is a Berkeley Logo primitive; I've defined the analogous `%` procedure here for use by `app`.

The `%` procedure outputs the value of a variable, `input.value`, that is local to `app`. If you invoke `%` in some context other than an `app` template, you'll get an error message because that variable won't exist. Logo's dynamic scope makes it possible for `%` to use `app`'s variable.

The real `apply` accepts a procedure name as argument instead of a template:

```
? show apply "first [Logo]
L
```

We can extend `app` to accept named procedures, but the definition is somewhat messier:

```
to app :template.or.name :input.value
ifelse wordp :template.or.name ~
       [run list :template.or.name "%] ~
       [run :template.or.name]
end
```

If the first input is a word, we construct a template by combining that procedure name with a percent sign for its input. However, in the rest of this section I'll simplify the discussion by assuming that `app` accepts only templates, not procedure names.

So far, `app` takes only one value as input; the real `apply` takes a list of values. I'll extend `app` to match:

```
to app :template :input.values
run :template
end

to % [:index 1]
output item :index :input.values
end
```

No change is needed to `app`, but `%` has been changed to use another new notation in its title line. `Index` is the name of an *optional input*. Although this notation also uses square brackets, it's different from the notation used in `foreach` because the brackets include a *default value* as well as the name for the input. This version of `%` accepts either no inputs or one input. If `%` is invoked with one input, then the value of that input will be associated with the name `index`, just as for ordinary inputs. If `%` is invoked with no inputs, then `index` will be given the value 1 (its default value).

```
? app [print word first (% 1) first (% 2)] [Paul Goodman]
PG
```

A percent sign with a number as input selects an input value by its position within the list of values. A percent sign by itself is equivalent to `(% 1)`.

The notation `(% 1)` isn't as elegant as the `?1` used in the real `apply`. You can solve that problem by defining several extra procedures:

```
to %1                to %2                to %3
output (% 1)         output (% 2)         output (% 3)
end                  end                  end
```

Berkeley Logo recognizes the notation `?2` and automatically translates it to `(? 2)`, as you can see by this experiment:

```
? show runparse [print word first ?1 first ?2]
[print word first ( ? 1 ) first ( ? 2 )]
```

(The primitive operation `runparse` takes a list as input and outputs the list as it would be modified by Logo when it is about to be run. That's a handwavy description, but the internal workings of the Logo interpreter are too arcane to explore here.)

Unlike the primitive `apply`, this version of `app` works only as a command, not as an operation. It's easy to write a separate version for use as an operation:

```
to app.oper :template :input.values
output run :template
end
```

It's not so easy in non-Berkeley versions of Logo to write a single procedure that can serve both as a command and as an operation. Here's one solution that works in versions with `catch`:

```
to app :template :input.values
catch "error [output run :template]
ignore error
end
```

This isn't an ideal solution, though, because it doesn't report errors other than "`run` didn't output to `output`." It could be improved by testing the error message more carefully instead of just ignoring it.

Berkeley Logo includes a mechanism that solves the problem more directly, but it's not very pretty:

```
to app :template :input.values
.maybeoutput run :template
end
```

The primitive command `.maybeoutput` is followed by a Logo expression that may or may not produce a value. If so, that value is output, just as it would be by the ordinary `output` command; the difference is that it's not considered an error if no value is produced.

From now on I'll use the primitive `apply`. I showed you `app` for two reasons. First, I think you'll understand `apply` better by seeing how it can be implemented. Second, this implementation may be useful if you ever work in a non-Berkeley Logo.

## Mapping

So far the iteration tools we've created apply only to commands. As you know, we also have the operation `map`, which is similar to `foreach` except that its template is an expression (producing a value) rather than an instruction, and it accumulates the values produced for each member of the input.

```
? show map [?*?] [1 2 3 4]
[1 4 9 16]
? show map [first ?] [every good boy does fine]
[e g b d f]
?
```

When implementing an iteration tool, one way to figure out how to write the program is to start with a specific example and generalize it. For example, here's how I'd write the example about squaring the numbers in a list without using `map`:

```
to squares :numbers
if emptyp :numbers [output []]
output fput ((first :numbers) * (first :numbers)) ~
            (squares butfirst :numbers)
end
```

`Map` is very similar, except that it applies a template to each datum instead of squaring it:

```
to map :template :values
if emptyp :values [output []]
output fput (apply :template (list first :values)) ~
            (map :template butfirst :values)
end
```

You may be wondering why I used `fput` rather than `sentence` in these procedures. Either would be just as good in the example about squares of numbers, because each datum is a single word (a number) and each result value is also a single word. But it's important to use `fput` in an example such as this one:

```
to swap :pair
output list last :pair first :pair
end

? show map [swap ?] [[Sherlock Holmes] [James Pibble] [Nero Wolfe]]
[[Holmes Sherlock] [Pibble James] [Wolfe Nero]]

? show map.se [swap ?] [[Sherlock Holmes] [James Pibble] [Nero Wolfe]]
[Holmes Sherlock Pibble James Wolfe Nero]
```

Berkeley Logo does provide an operation `map.se` in which `sentence` is used as the combiner; sometimes that's what you want, but not, as you can see, in this example. (A third possibility that might occur to you is to use `list` as the combiner, but that never turns out to be the right thing; try writing a `map.list` and see what results it gives!)

As in the case of `foreach`, the program gets a little more complicated when we extend it to handle multiple data inputs. Another complication that wasn't relevant to `foreach` is that when we use a word, rather than a list, as the data input to `map`, we must use `word` as the combiner instead of `fput`. Here's the complete version:

```
to map :map.template [:template.lists] 2
op map1 :template.lists 1
end

to map1 :template.lists :template.number
if emptyp first :template.lists [output first :template.lists]
output combine (apply :map.template firsts :template.lists)
               (map1 bfs :template.lists :template.number+1)
end

to combine :this :those
if wordp :those [output word :this :those]
output fput :this :those
end
```

This is the actual program in the Berkeley Logo library. One feature I haven't discussed until now is the variable `template.number` used as an input to `map1`. Its purpose is to allow the use of the number sign character `#` in a template to represent the position of each datum within its list:

```
? show map [list ? #] [a b c]
[[a 1] [b 2] [c 3]]
```

The implementation is similar to that of `?` in templates:

```
to #
output :template.number
end
```

It's also worth noting the base case in `map1`. When the data input is empty, we must output either the empty word or the empty list, and the easiest way to choose correctly is to return the empty input itself.

## Mapping as a Metaphor

In this chapter, we got to the idea of mapping by this route: iteration, numeric iteration, other kinds of iteration, iteration on a list, iterative commands, iterative operations, mapping. In other words, we started thinking about the mapping tool as a particular kind of repetition in a computer program.

But when I first introduced `map` as a primitive operation, I thought about it in a different way. Never mind the fact that it's *implemented* through repetition. Instead think of it as extending the power of the idea of a list. When we started thinking about lists, we thought of the list as one complete entity. For example, consider this simple interaction with Logo:

```
? print count [how now brown cow]
4
```

`Count` is a primitive operation. It takes a list as input, and it outputs a number that is a property of the entire list, namely the number of members in the list. There is no need to think of `count` as embodying any sort of repetitive control structure. Instead it's one kind of handle on the *data* structure called a list.

There are other operations that manipulate lists, like `equalp` and `memberp`. You're probably in the habit of thinking of these operations as "happening all at once," not as examples of iteration. And that's a good way to think of them, even though it's also possible to think of them as iterative. For example, how does Logo know the `count` of a list? How would *you* find out the number of members of a list? One way would be to count them on your fingers. That's an iteration. Logo actually does the same thing, counting off the list members one at a time, as it would if we implemented `count` recursively:

```
to cnt :list
if emptyp :list [output 0]
output 1+cnt butfirst :list
end
```

I'm showing you that the "all at once" Logo primitives can be considered as iterative because, in the case of `map`, I want to shift your point of view in the opposite direction. We started thinking of `map` as iterative; now I'd like you to think of it as happening all at once.

Wouldn't it be nice if we could say

```
? show 1+[5 10 15]
[6 11 16]
```

That is, I'd like to be able to "add 1 to a list." I want to think about it that way, not as "add 1 to each member of a list." The metaphor is that we're doing something to the entire list at once. Well, we can't quite do it that way, but we can say

```
? show map [1+?] [5 10 15]
[6 11 16]
```

Instead of thinking "Well, first we add 1 to 5, which gives us 6; then we add..." you should think "we started with a list of three numbers, and we've transformed it into another list of three numbers using the operation add-one."

## Other Higher Order Functions

Along with `map`, you learned about the higher order functions `reduce`, which combines all of the members of a list into a single result, and `filter`, which selects some of the members of a list. They, too, are implemented by combining recursion with `apply`. Here's the Berkeley Logo library version of `reduce`:

```
to reduce :reduce.function :reduce.list
if emptyp butfirst :reduce.list [output first :reduce.list]
output apply :reduce.function (list (first :reduce.list)
                                    (reduce :reduce.function
                                            butfirst  :reduce.list))
end
```

If there is only one member, output it. Otherwise, recursively reduce the butfirst of the data, and apply the template to two values, the first datum and the result from the recursive call.

The Berkeley Logo implementation of `filter` is a little more complicated, for some of the same reasons as that of `map`: the ability to accept either a word or a list, and the `#` feature in templates. So I'll start with a simpler one:

```
to filter :template :data
if emptyp :data [output []]
if apply :template (list first :data) ~
   [output fput (first :data)
                (filter :template butfirst :data)]
output filter :template butfirst :data
end
```

If you understand that, you should be able to see the fundamentally similar structure of the library version despite its extra details:

```
to filter :filter.template :template.list [:template.number 1]
localmake "template.lists (list :template.list)
if emptyp :template.list [output :template.list]
if apply :filter.template (list first :template.list) ~
   [output combine (first :template.list)
                   (filter :filter.template (butfirst :template.list)
                           :template.number+1)]
output (filter :filter.template (butfirst :template.list)
               :template.number+1)
end
```

Where `map` used a helper procedure `map1` to handle the extra input `template.number`, `filter` uses an alternate technique, in which `template.number` is declared as an optional input to `filter` itself. When you invoke `filter` you always give it the default two inputs, but it invokes itself recursively with three.

Why does `filter` need a local variable named `template.lists`? There was a variable with that name in `map` because it accepts more than one data input, but `filter` doesn't, and in fact there is no reference to the value of `template.lists` within `filter`. It's there because of another feature of templates that I haven't mentioned: you can use the word `?rest` in a template to represent the portion of the data input to the right of the member represented by `?` in this iteration:

```
to remove.duplicates :list
output filter [not memberp ? ?rest] :list
end

?  show remove.duplicates [ob la di ob la da]
[di ob la da]
```

Since `?rest` is allowed in `map` templates as well as in `filter` templates, its implementation must be the same for both:

```
to ?rest [:which 1]
output butfirst item :which :template.lists
end
```

## Mapping Over Trees

It's time to move beyond the iteration tools in the Logo library and invent our own new ones.

So far, in writing operations on lists, we've ignored any sublist structure within the list. We do something for each top-level member of the input list. It's also possible to take advantage of the complex structures that lists make possible. For example, a list can be used to represent a *tree,* a data structure in which each branch can lead to further branches. Consider this list:

```
[[the [quick brown] fox] [[jumped] [over [the [lazy] dog]]]]
```

My goal here is to represent a sentence in terms of the phrases within it, somewhat like the sentence diagrams you may have been taught in elementary school. This is a list with two members; the first member represents the subject of the sentence and the second represents the predicate. The predicate is further divided into a verb and a prepositional phrase. And so on. (A representation something like this, but more detailed, is used in any computer program that tries to understand "natural language" interaction.)

Suppose we want to convert each word of this sentence to capital letters, using Berkeley Logo's `uppercase` primitive that takes a word as input. We can't just say

```
map [uppercase ?] ~
    [[the [quick brown] fox] [[jumped] [over [the [lazy] dog]]]]
```

because the members of the sentence-list aren't words. What I want is a procedure `map.tree` that applies a template to each *word* within the input list but maintains the shape of the list:

```
? show map.tree [uppercase ?]~
     [[the [quick brown] fox] [[jumped] [over [the [lazy] dog]]]]
[[THE [QUICK BROWN] FOX] [[JUMPED] [OVER [THE [LAZY] DOG]]]]
```

After our previous adventures in mapping, this one is relatively easy:

```
to map.tree :template :tree
if wordp :tree [output apply :template (list :tree)]
if emptyp :tree [output []]
output fput (map.tree :template first :tree) ~
            (map.tree :template butfirst :tree)
end
```

This is rather a special-purpose procedure; it's only good for trees whose "leaves" are words. That's sometimes the case but not always. But if you're dealing with sentence trees like the one in my example, you might well find several uses for a tool like this. For now, I've introduced it mainly to make the point that the general idea of iteration can take many different forms, depending on the particular project you're working on. (Technically, this is *not* an iteration, because it doesn't have a two-part structure in which the first part is to perform one step of a computation and the second part is to perform all the rest of the steps. `Map.tree` does have a two-part structure, but *both* parts are recursive calls that might carry out several steps. But `map.tree` does generalize the broad idea of dividing a large computation into similar individual pieces. We'll go into the nature of iteration more carefully in a moment.)

## Iteration and Tail Recursion

If you look back at the introduction to recursion in the first volume, you'll find that some recursive commands seem to be carrying out an iteration, like `down`, `countdown`, or `one.per.line`. (In this chapter we've seen how to implement `countdown` using `for`, and you should easily be able to implement `one.per.line` using `foreach`. `Down` isn't exactly covered by either of those tools; can you see why I call it an iterative problem anyway?) Other recursive commands don't seem to be repeating or almost-repeating something, like `downup` or `hanoi`. The difference is that these commands don't do something completely, then forget about it and go on to the next repetition. Instead,

the first invocation of `downup`, for example, still has work of its own to do after all the lower-level invocations are finished.

It turns out that a command that is *tail* recursive is one that can be thought of as carrying out an iteration. A command that invokes itself somewhere before the last instruction is not iterative. But the phrase "tail recursive" doesn't *mean* "equivalent to an iteration." It just happens to work out, for commands, that the two concepts are equivalent. What "tail recursive" means, really, is "invokes itself just before stopping."

I've said before that this isn't a very important thing to worry about. The reason I'm coming back to it now is to try to clear up a confusion that has been part of the Logo literature. Logo implementors talk about tail recursion because there is a tricky way to implement tail recursion that takes less memory than the more general kind of recursion. Logo *teachers,* on the other hand, tend to say "tail recursive" when they really mean "iterative." For example, teachers will ask, "Should we teach tail recursion first and then the general case?" What's behind this question is the idea that iteration is easier to understand than recursion. (By the way, this is a hot issue. Most Logo teachers would say yes; they begin by showing their students an iterative command like `poly` or `polyspi`. I generally say no; you may recall that the first recursive procedure I showed you is `downup`. One reason is that I expect some of my readers have programmed in Pascal or C, and I want to make it as hard as possible for such readers to convince themselves that recursion is just a peculiar way to express the idea of iteration.)

There are two reasons people should stop making a fuss about tail recursion. One is that they're confusing an idea about control structures (iteration) with a Logo implementation strategy (tail recursion). The second is that this way of thinking directs your attention to commands rather than operations. (When people think of iterative procedures as "easier," it's always commands that they have in mind. Tail recursive operations are, if anything, less straightforward than versions that are non-tail recursive.) Operations are more important; they're what gives Logo much of its flexibility. And the best way to think about recursive operations isn't in implementation terms but in terms of data transformation abstractions like mapping, reduction, and filters.

## Multiple Inputs to `For`

Earlier I promised you `multifor`, a version of `for` that controls more than one numeric variable at a time. Its structure is very similar to that of the original `for`, except that we use `map` or `foreach` (or `firsts` or `butfirsts`, which are implicit uses of `map`) in almost every instruction to carry out `for`'s algorithm for each of `multifor`'s numeric variables.

```
to multifor :values.list :instr
localmake "vars firsts :values.list
local :vars
localmake "initials map "run firsts butfirsts :values.list
localmake "finals map [run item 3 ?] :values.list
localmake "steps (map "multiforstep :values.list :initials :finals)
localmake "testers map [ifelse ? < 0 [[?1 < ?2]] [[?1 > ?2]]] :steps
multiforloop :initials
end

to multiforstep :values :initial :final
if (count :values)=4 [output run last :values]
if :initial > :final [output -1]
output 1
end

to multiforloop :values
(foreach :vars :values [make ?1 ?2])
(foreach :values :finals :testers [if run ?3 [stop]])
run :instr
multiforloop (map [?1+?2] :values :steps)
end
```

This is a very dense program; I wouldn't expect anyone to read and understand it from a cold start. But if you compare it to the implementation of `for` on page 184, you should be able to make sense of how each line is transformed in this version.

Here is an example you can try:

```
? multifor [[a 10 100 5] [b 100 10 -10]] ~
           [print (sentence :a "+ :b "= (:a + :b))]
10 + 100 = 110
15 + 90 = 105
20 + 80 = 100
25 + 70 = 95
30 + 60 = 90
35 + 50 = 85
40 + 40 = 80
45 + 30 = 75
50 + 20 = 70
55 + 10 = 65
?
```

## The Evaluation Environment Bug

There's a problem with all of these control structure tools that I haven't talked about. The problem is that each of these tools uses `run` or `apply` to evaluate an expression that's provided by the calling procedure, but the expression is evaluated with the tool's local variables active, in addition to those of the calling procedure. This can lead to unexpected results if the name of a variable used in the expression is the same as the name of one of the local variables in the tool. For example, `forloop` has an input named `final`. What happens if you try

```
to grade :final
for [midterm 10 100 10] [print (sum :midterm :final) / 2]
end

? grade 50
```

Try this example with the implementation of `for` in this chapter, not with the Logo library version. You might expect each iteration to add 10 and 50, then 20 and 50, then 30 and 50, and so on. That is, you wanted to add the iteration variable `midterm` to the input to `grade`. In fact, though, the variable that contributes to the sum is `forloop`'s `final`, not `grade`'s `final`.

The way to avoid this problem is to make sure you don't use variables in superprocedures of these tools with the same names as the ones inside the tools. One way to ensure that is to rewrite all the tool procedures so that their local variables have bizarre names:

```
to map :template :inputs
```

becomes

```
to map :map.qqzzqxx.template :map.qqzzqxx.inputs
```

Of course, you also have to change the names wherever they appear inside the definition, not just on the title line. You can see why I preferred not to present the procedures to you in that form!

It would be a better solution to have a smarter version of `run`, which would allow explicit control of the *evaluation environment*—the variable names and values that should be in effect while evaluating `run`'s input. Some versions of Lisp do have such a capability.

# 11 Example: Cryptographer's Helper

Program file for this chapter: `crypto`

A *cryptogram* is a kind of word puzzle, like a crossword puzzle. Instead of definitions, though, a cryptogram gives you the actual words of a quotation, but with each letter replaced with a different letter. For example, each letter A in the original text might be replaced with an F. Here is a sample cryptogram:

> LB RA, BT YBL LB RA: LJGL CQ LJA FUAQLCBY: KJALJAT 'LCQ YBRXAT CY LJA DCYP LB QUSSAT LJA QXCYWQ GYP GTTBKQ BS BULTGWABUQ SBTLUYA, BT LB LGHA GTDQ GWGCYQL G QAG BS LTBURXAQ, GYP RM BIIBQCYW AYP LJAD?

The punctuation marks and the spaces between words are the same in this cryptogram as they are in the original ("clear") text.

A cryptogram is a kind of secret code. The formal name for this particular kind of code is a *simple substitution cipher*. Strictly speaking, a *code* is a method of disguising a message that uses a dictionary of arbitrarily chosen replacements for each possible word. A foreign language is like a code. A *cipher* is a method in which a uniform algorithm or formula is used to translate each word. A *substitution* cipher is one in which every letter (or sometimes every pair of letters, or some such grouping) is replaced by a disguised equivalent. A *simple* substitution cipher is one in which each letter has a single equivalent replacement, which is used throughout the message. (A more complicated substitution cipher might be something like this: the first letter A in the message is replaced with F, the second A is replaced with G, the third with H, and so on.)

Years ago, Arthur Conan Doyle and Edgar Allen Poe were able to write mystery stories in which simple substitution ciphers were used by characters who really wanted to keep a message secret. Today, partly because of those stories, too many people know how to "break" such ciphers for them to be of practical use. Instead, these ciphers are used as word puzzles.

The technique used for decoding a cryptogram depends on the fact that some letters are more common than others. The letter A is much more common in English words than the letter Z. If, in a cryptogram, the letter F occurs many times, it's more likely to represent a letter like A in the original text than a letter like Z.

The most commonly used letter in English is E, by a wide margin. T is in second place, with A and O nearly tied for third. I, N, and R are also very commonly used. These rankings apply to *large* texts. In the usual short cryptogram, the most frequent letter doesn't necessarily represent E. But the letter that represents E will probably be among the two or three most frequent.

Before reading further, you might want to try to solve the cryptogram shown above. Make a chart of the number of times each letter appears, then use that information to make guesses about which letter is which. As you're working on it, make a note of what other kinds of information are helpful to you.

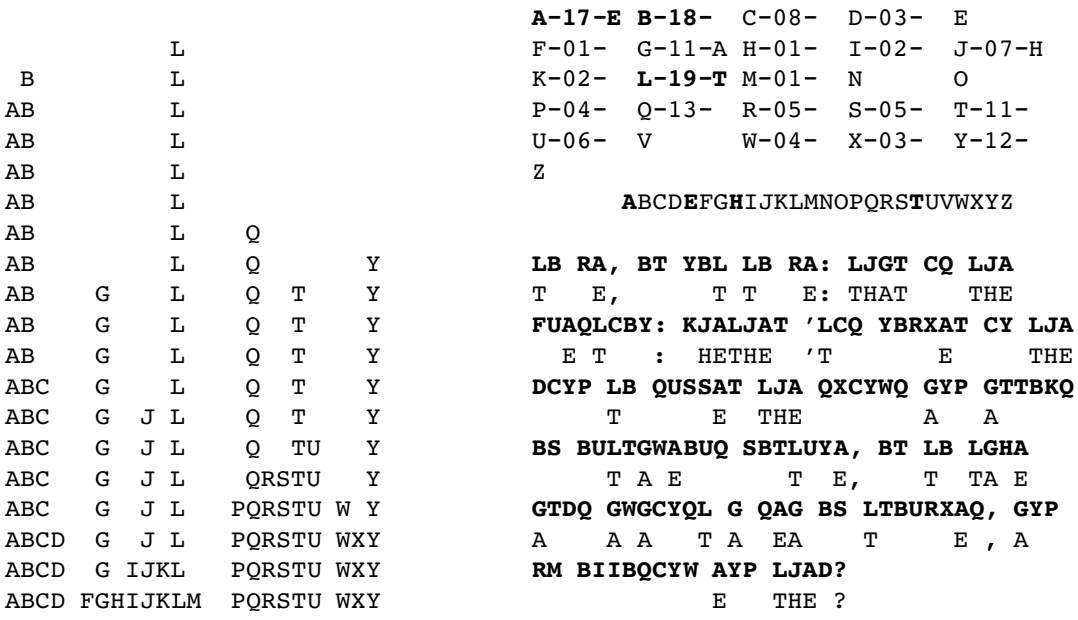
This project is a program to help you solve cryptograms. The program doesn't solve the puzzle all by itself; it doesn't know enough about English vocabulary. But it does some of the more boring parts of the job automatically, and can make good guesses about some of the letters.

The top-level procedure is `crypto`. It takes one input, a list whose members are the words of the cryptogram. Since these lists are long and easy to make mistakes in, you'll probably find it easier to type the cryptogram into the Logo editor rather than directly at a question mark prompt. You might make the list be the value of a variable, then use that variable as the input to `crypto`. (The program file for this project includes four such variables, named `cgram1` through `cgram4`, with sample cryptograms.)

`Crypto` begins by going through the coded text, letter by letter. It keeps count of how often each letter is used. You can keep track of this counting process because the program draws a *histogram* on the screen as it goes. A histogram is a chart like the one at the top of the next page.

A histogram is a kind of graph, but it's different from the *continuous* graphs you use in algebra. Histograms are used to show quantities of *discrete* things, like letters of the alphabet.

The main reason the program draws the histogram is that it needs to know the frequencies of occurrence of the letters for later use. When I first wrote the program, it counted the letters without printing anything on the screen. Since this counting is a fairly slow process, it got boring waiting for the program to finish. The histogram display is a sort of video thumb-twiddling to keep you occupied while the program is creating an invisible histogram inside itself.




Histogram Screen display

By the way, since there are only 24 lines on the screen, the top part of the histogram may be invisible if the cryptogram is long enough to use some letters more than 24 times.

The shape of this histogram is pretty typical. A few letters are used many times, while most letters are clumped down near the bottom. In this case, A, B, and L stand out. You might guess that they represent the most commonly used letters: E, T, and either A or O. But you need more information to be able to guess which is which.

After it finishes counting letters, the program presents a screen display like the one shown above. The information provided in this display comes in three parts. At the top is an alphabetical list of the letters in the cryptogram. For each letter, the program displays the number of times that letter occurs in the enciphered text. For example, the letter P occurs four times. The letter that occurs most frequently is highlighted by showing it in reverse video characters, represented in the book with boldface characters. In this example, the most frequently used letter is L, with 19 occurrences. Letters with occurrence counts within two of the maximum are also highlighted. In the example, A with 17 and B with 18 are highlighted. If a letter does not occur in the cryptogram at all, no count is given. In the example, there is no E in the enciphered text.

The top part of the display shows one more piece of information: if either the program or the person using it has made a guess as to the letter that a letter represents, that guess is shown after the frequency count. For example, here the program has guessed that the letter L in the cryptogram represents the letter T in the clear text. (You can't tell from the display that this guess was made by the program rather than by the person using it. I just happen to know that that's what happened in this example!)

The next section of the display is a single line showing all the letters of the alphabet. In this line, a letter is highlighted if a guess has been made linking some letter in the cryptogram with that letter in the clear text. In other words, this line shows the linkages in the reverse direction from what is shown in the top section of the display. For example, I just mentioned that L in the cryptogram corresponds to T in the clear text. In the top part of the display, we can find L in alphabetical order, and see that it has a T linked to it. But in the middle part of the display, we find *T*, not L, in alphabetical order, and discover that *something* is linked to it. (It turns out that we don't usually have to know which letter corresponds to T.)

Here is the purpose of that middle section of the display: Suppose I am looking at the second word of the cryptogram, RA. We've already guessed that A represents E, so this word represents something-E. Suppose I guess that this word is actually HE. This just happens to be the first two-letter word I think of that ends in E. So I'd like to try letting R represent H. Now I look in the middle section of the display, and I see that H is already highlighted. So some other letter, not R, already represents H. I have to try a different guess.

The most important part of the display is the bottom section. Here, lines of cryptogram alternate with their translation into clear text, based on the guesses we've made so far. The cryptogram lines are highlighted, just to make it easy to tell which lines are which. The program ensures that each word entirely fits on a single line; there is no wrapping to a new line within a single word.

There is room on the screen for eight pairs of lines. If the cryptogram is too big to fit in this space, only a portion of it will be visible at any time. In a few paragraphs I'll talk about moving to another section of the text.

The program itself is very limited in its ability to guess letters. For the most part, you have to do the guessing yourself when you use it. There are three guessing rules in the program:

1. The most frequently occurring single-letter word is taken to represent A.
2. Another single-letter word, if there is one, is taken to represent I.

3. The most frequently occurring three-letter word is taken to represent THE, but only if its last letter is one of the ones highlighted in the top part of the display.

In the example, the only single-letter word in the cryptogram is G, in the next-to-last line. The program, following rule 1, has guessed that G represents A. Rule 2 did not apply, because there is no second single-letter word. The most frequently used three-letter word is LJA, which occurs three times. The last letter of that word, A, is highlighted in the top section because it occurs 17 times. Therefore, the program guesses that L represents T, J represents H, and A represents E.

Of course you understand that these rules are not infallible; they're just guesses. (A fancy name for a rule that works most of the time is a *heuristic*. A rule that works all the time is called an *algorithm*.) For example, the three-letter word GYP appears twice in the cryptogram, only once less often than LJA. Maybe GYP is really THE. However, the appearance of the word THAT in the translation of the first line is a pretty persuasive confirmation that the program's rules have worked out correctly in this case.

If you didn't solve the cryptogram on your own, at my first invitation, you might want to take another look at it, based on the partial solution you now have available. Are these four letters (A, E, I, and T) enough to let you guess the rest? It's a quotation you'll probably recognize.

Once this display is on the screen, you can make further guesses by typing to the program. For example, suppose you decide that the last word of the cryptogram, LJAD, represents THEM. Then you want to guess that D represents M. To do that, type the letters D and M in that order. Don't use the RETURN key. Your typing will not be echoed on the screen. Instead, three things will happen. First, the entry in the top section of the display that originally said

```
D-03-
```

will be changed to say

```
D-03-M
```

Second, the letter M will be highlighted in the alphabet in the second section of the display. Finally, the program will type an M underneath every D in the cryptogram text.

If you change your mind about a guess, you can just enter a new guess about the same cryptogram letter. For example, if you decide that LJAD is really THEY instead of THEM, you could just type D and Y. Alternatively, if you decide a guess was wrong but

you don't have a new guess, type the cryptogram letter (D in this example) and then the space bar.

If you guess that D represents M, and then later you guess that R also represents M, the program will complain at you by beeping or by flashing the screen, depending on what your computer can do. If you meant that R should represent M *instead of* D representing M, you must first undo the latter guess by typing D, space bar, R, and M.

The process of redisplaying the clear text translation of the cryptogram after each guess takes a fairly long time, since the program has to look up each letter individually. Therefore, the program is written so that you don't have to wait for this redisplay to finish before guessing another letter representation. As soon as you type any key on the keyboard, the program stops retyping the clear text. Whatever key you typed is taken as the first letter of a two-letter guess command.

If the cryptogram is too long to fit on the screen, there are three other things you can type to change which part of the text is visible. Typing a plus sign (+) eliminates the first four lines of the displayed text (that is, four lines of cryptogram and four corresponding lines of cleartext) and brings in four new lines at the end. Typing a minus sign (–) moves backwards, eliminating the four lines nearest the bottom of the screen and bringing back four earlier lines at the top. These *windowing* commands have no effect if you are already seeing the end of the text (for +) or the beginning of the text (for –).

The third command provided for long cryptograms is the atsign (@) character. This is most useful after you've figured out all of the letter correspondences. It clears the screen and displays only the clear text, without the letter frequencies, table of correspondences, or the enciphered text. This display allows 23 lines of clear text to fit on the screen instead of only eight. If you don't have the solution exactly right, you can type any character to return to the three-part display and continue guessing.

The program never stops; even after you have made guesses for all the letters, you might find an error and change your mind about a guess. When you're done, you stop the program with control-C or command-period or whatever your computer requires.

In the complete listing at the end of this chapter, there are a few cryptograms for you to practice with. They are excerpted from one of my favorite books, *Compulsory Miseducation* by Paul Goodman.

## Program Structure

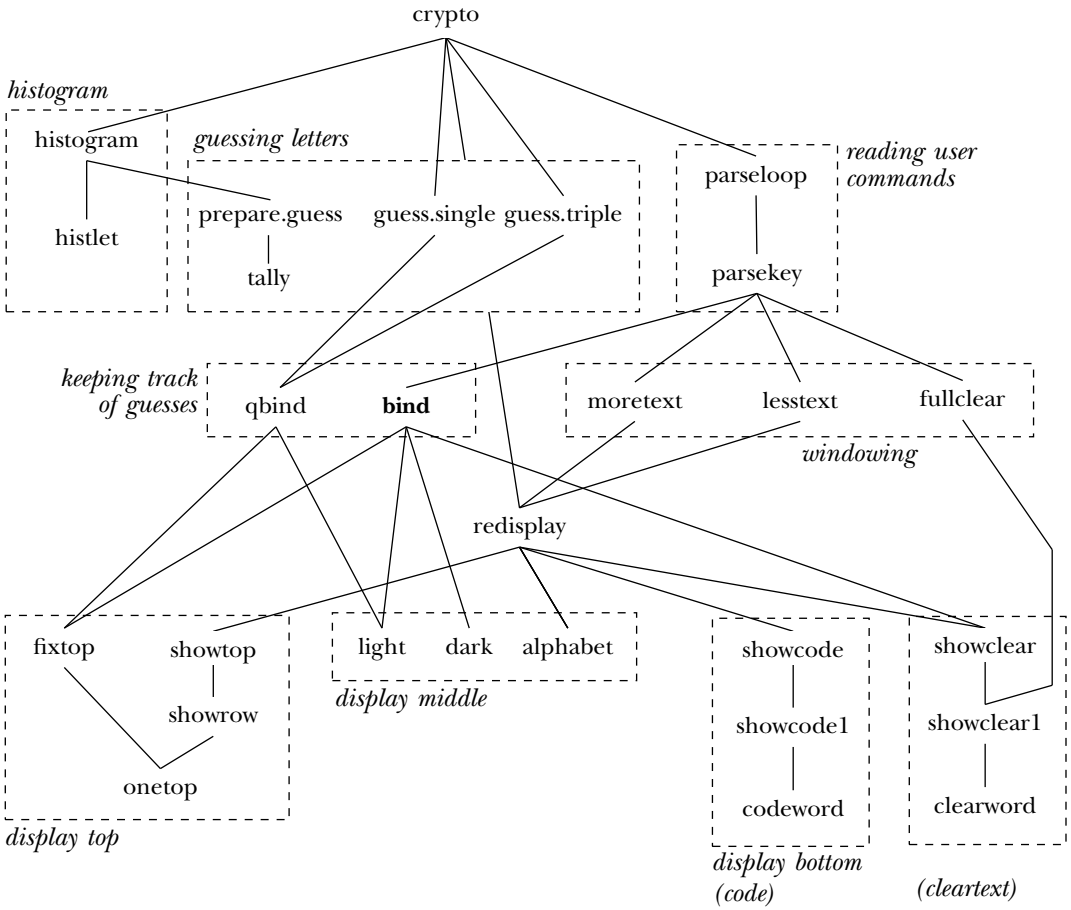
There are about 50 procedures in this program. These procedures can be roughly divided into several purposes:

- initialization
- frequency counting and displaying the histogram
- guessing letters automatically
- reading user commands
- keeping track of guesses
- top section of display (frequencies)
- middle section of display (alphabet)
- bottom section of display (cryptogram text and cleartext)
- windowing and full-text displays
- data abstraction and other helper procedures

The diagram on the next page shows superprocedure/subprocedure relationships within the main categories. (Helper procedures aren't shown, to make the diagram more readable.) The bottom half of the diagram has the procedures that are concerned primarily with presenting information on the screen. `Redisplay`, near the center of the diagram, is called whenever the entire screen display must be redrawn: when the initialization part of the program is finished, and whenever the user chooses a new portion of the text to display. When the display changes slightly, because a new guess is made, procedures such as `fixtop`, `light`, and `dark` are used instead of redrawing everything.

`Bind` is the most important procedure, because it records and displays each new guess. As the diagram shows, it invokes several subprocedures to update the display; more importantly, it changes the values of several variables to keep track of the new guess. There is also a similar procedure `qbind` that's used when a guess is made by the program rather than by the user. (The "Q" stands for either "quick" or "quiet," since this version never has to undo an old guess, omits some error checking, and can't beep because there are no errors in automatic guesses.) If you ignore initialization and displaying information, the entire structure of the program is that `crypto` calls `parseloop`, which repeatedly calls `parsekey`, which calls `bind` to record a guess.

Unfortunately, it's not so easy in practice to divide up the procedures into groups, with a single purpose for each group. Several procedures carry out two tasks at once. For example, `light` and `dark` have those names because they switch individual letters between normal and inverse video in the alphabet display in the middle part of the screen. But those procedures also set variables to remember that a particular cleartext letter has or hasn't been guessed, so they are also carrying out part of `bind`'s job, keeping track of guesses.




## Guided Tour of Global Variables

`Crypto` uses many global variables to hold the information it needs. This includes information about individual letters, about words, and about the text as a whole.

There are several sets of 26 variables, one for each letter of the alphabet. For these variables, the last letter of the variable name is the letter about which the variable holds information. In the table that follows, the italic *x* in each name represents any letter.

| | |
|---|---|
| *x* | Cleartext letter that is guessed to match *x* in the cryptogram. |
| `bound`*x* | `True` if *x* appears in the *cleartext* as guessed so far; `false` otherwise. |
| `cnt`*x* | Count of how many times *x* appears in the cryptogram. |
| `posn`*x* | Screen cursor position where the frequency count and guess for *x* is shown in the top part of the display. |

These variables are set up initially by `initvars`, except for the `posn` variables, which are set by `showrow`. The variables with single-letter names start out with a space character as their value. This choice allows `showclear` to use `thing :letter` as the thing to type for every letter in the cryptogram. If no guess has been made for a letter, it will be displayed as a blank space in the partially-decoded version of the text.

Here are the variables that have to do with *words* in the cryptogram text. These variables are needed for the part of the program that automatically makes guesses, by looking for words that might represent A, I, and THE in the cleartext. In the following variable names, *y* represents either a one-letter word or a three-letter word in the cryptogram text.

| | |
|---|---|
| `count.single` | The number of occurrences of the most frequent one-letter word. |
| `count.triple` | The number of occurrences of the most frequent three-letter word. |
| `list.single` | List of one-letter words in the cryptogram text. |
| `list.triple` | List of three-letter words in the cryptogram text. |
| `max.single` | The most frequent one-letter word in the cryptogram text. |
| `max.triple` | The most frequent three-letter word in the cryptogram text. |
| `single`*y* | The number of occurrences of the one-letter word *y*. |
| `triple`*y* | The number of occurrences of the three-letter word *y*. |

These variables are used only during the initial histogram counting, to keep track of which one-letter word and which three-letter word are the most frequent in each category. Once the most frequently occurring words have been determined, the actual count is no longer important.

Finally, there are some variables that contain information about the text as a whole:

| | |
|---|---|
| `fulltext` | The complete cryptogram text. |
| `text` | The part of the cryptogram that is displayed on the screen right now. |
| `moretext` | The part of the text that should be displayed after a `+` command. |
| `textstack` | A list of old values of `text`, to be restored if the `-` command is used. |
| `maxcount` | The number of occurrences of the most frequently used letter. |

`:Maxcount` is used to know which letters should be highlighted in the top section of the display. `:Text` is used by `showcode` and `showclear` to maintain the bottom section of the display. `Fulltext`, `moretext`, and `textstack` are part of the windowing feature. At first, `text` is equal to `fulltext`, and `textstack` is empty. `Moretext` contains the portion of the text starting on the fifth line that is displayed, providing there is some text at the end of the cryptogram that didn't fit on the screen. If the end of the text is visible, then `moretext` is empty. Here is what happens if you type the plus sign:

```
to moretext
if emptyp :moretext [beep stop]
push "textstack :text
make "text :moretext
redisplay "true
end
```

If `:moretext` is empty, there is no more text to display, and the procedure stops with a complaint. Otherwise, we want to remember what is now in `:text` in case of a later `-` command, and we want to change the value of `text` to the version starting four lines later that is already in `:moretext`.

In the solitaire project, I used a lot of `local` instructions in the top-level procedures to avoid creating global variables. In this project, I didn't bother. There's no good reason why I was lazier here than there; you can decide for yourself whether you think it's worth the effort.

## What's In a Name?

In revising this program for the second edition, I was struck by the ways in which bad choices of procedure or variable names had made it needlessly hard to read. Changing names was one of the three main ways in which I changed the program. (The other two were an increased use of data abstraction and the introduction of iteration tools to eliminate some helper procedures.)

I'll start with a simple example. As I've mentioned, when I first wrote the program it didn't draw the histogram on the screen during the initial counting of letter frequencies. Since the top part of the screen display is primarily a presentation of those frequencies, I thought of that top part as the program's "histogram" even though it doesn't have the form of a real histogram. That's why, in the first edition, the procedures that maintain the top part of the display were called `showhist`, `fixhist`, and so on; when I added the `histogram` and `histlet` procedures that draw the real histogram, it was hard to keep track of which "`hist`" names were part of the initial histogram and which were part of the letter frequency chart at the top of the program's normal screen display. I've now changed `showhist` to `showtop`, `fixhist` to `fixtop`, and so on. The procedures with `hist` in their names are about the real histogram, and the ones with `top` in their names are about the frequency chart.

Here's another example. In several parts of the program, I had to determine whether a character found in the cryptogram text is a letter or a punctuation mark. The

most straightforward way to do this would be an explicit check against all the letters in the alphabet:

```
to letterp :char
output memberp :char "ABCDEFGHIJKLMNOPQRSTUVWXYZ
end
```

But comparing the character against each of the 26 letters would be quite slow. Instead, I took advantage of the fact that there happen to be variables in the program named after each letter. That is, there's a variable `A`, a variable `B`, and so on, but there aren't variables named after punctuation characters. Therefore, I could use the Logo primitive `namep` to see whether or not the character I'm considering is a variable name, and if so, it must be a letter. The first edition version of `crypto` is full of instructions of the form

```
if namep :char ...
```

This is clever and efficient, but not at all self-documenting. Someone reading the program would have no way to tell that I'm using `namep` to find out whether a character is a letter. The solution was to add an instruction to the initialization in `crypto`:

```
copydef "letterp "namep
```

The `copydef` primitive is used to give a new name to an existing procedure. (The old name continues to work.) The existing procedure can be either primitive or user-defined. The new name is not saved by the `save` command; that's why `crypto` performs the `copydef` instruction each time.

Probably the worst example of bad naming was in the `tally` procedure. This procedure has a complicated job; it must keep track of the most common one-letter and three-letter words, in preparation for the program's attempts to make automatic guesses for A, I, and THE. Here is the version in the first edition:

```
to tally :type :word
local "this
make "this word :type :word
if not memberp :word list. :type ~
   [setlist. :type fput :word list. :type   make :this 0]
make :this sum 1 thing :this
make "this thing :this
if :this > (count. :type) ~
   [setcount. :type :this  make (word "max. :type) :word]
end
```

The input named `type` is either the word `single` or the word `triple`. One thing that makes this procedure hard to read is the local variable named `this`. What a vague name! This what? Is it this word, or this letter, or this word length, or this guess? To make things worse, partway through the procedure I recycled the same name to hold a different value. At first, `:this` is a word that will be used as the name of a variable, counting the number of times a given word appears. For example, if the word YBL appears in the cryptogram, then `tally` will create a variable named `tripleybl` whose value will be the number of times that YBL occurs in the text. The value of `this` will be the word `tripleybl`, so the expression `thing :this` represents the actual number. Then, near the end of the procedure, I used the instruction

```
make "this thing :this
```

From then on, `:this` is the number itself, not the variable name! It's really hard to read a procedure in which the same name is used to mean different things in different instructions.

Here's the new version:

```
to tally :type :word
localmake "countvar word :type :word
if not memberp :word list. :type ~
   [setlist. :type fput :word list. :type   make :countvar 0]
localmake "count (thing :countvar)+1
make :countvar :count
if :count > (count. :type) ~
   [setcount. :type :count   setmax. :type :word]
end
```

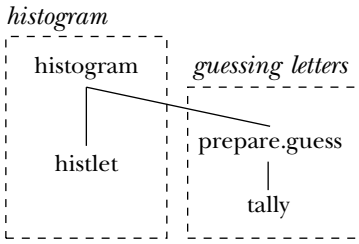
The name `this` is gone. Instead, I've first created a local variable named `countvar` whose value is the name of the count variable. Then I create another local variable named `count` that contains the actual count. These names are much more descriptive of the purposes of the two variables.

Another change in the new version is a more consistent use of data abstraction. The original version used the constructor `setlist.` and the selector `list.` to refer to the list of all known cryptogram words of the appropriate length (the variable `list.single` or `list.triple`), but used the instruction

```
make (word "max. :type) :word
```

to construct the variable containing the most frequently appearing word of that length. The new version uses a constructor named `setmax.` that's analogous to the `setlist.` constructor.

Rethinking the names of procedures can reorganize your ideas about how to group the procedures into categories. For example, in the first edition I was upset about the fact that `historgram`, whose job is to count letter frequencies and draw the histogram of those counts, also invokes prepare.guess, whose job is to count *word* frequencies in preparation for automatic guessing.




The reason for this mixture of tasks is efficiency. To prepare the histogram, the program must extract the letters (omitting punctuation) from each word of the text, and count them. To prepare for guessing words, the program must extract the letters from each word, and count the occurrences of the letters-only words. I could have done these things separately:

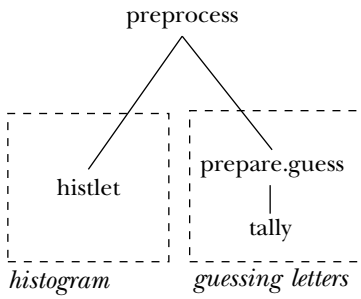
```
to histogram :text
foreach :text [foreach (filter "letterp ?) "histlet]
end

to count.words :text
foreach :text [prepare.guess (filter "letterp ?)]
end
```

But it seemed better to scan the words of the text just once, and extract the letters from each word just once:

```
to histogram :text
foreach :text [localmake "word filter "letterp ?
               foreach :word "histlet
               prepare.guess :word]
end
```

But the punch line of this story is that I could avoid the confusing jump between boxes—the feeling of mixing two tasks—merely by changing the name of the `histogram` procedure to something neutral like `preprocess`. Then the structure would be




Now we have one initialization procedure that includes invocations for two separate kinds of preprocessing. It's not really the program structure that is inappropriate, but only using the name `histogram` for a procedure whose job includes more than the creation of the histogram.

## Flag Variables

Procedure `redisplay` has the job of redrawing the entire screen when there is a major change to what should be shown, like moving to a different window in the cryptogram text.

```
to redisplay :flag
cleartext
showtop
alphabet
showcode :text
if :flag [showclear :text]
end
```

The input to `redisplay` is a *flag variable*. It must have the value `true` or `false`. (The name comes from the flags on mailboxes, which are either up or down to indicate whether or not there is mail in the box.) It's there because `redisplay` has two slightly different jobs to do at two different points in the program. First, `redisplay` is invoked by `crypto`, the top-level procedure, to draw the screen initially. At this time, no letters have been guessed yet. Therefore, it is not necessary to invoke `showclear` (which indicates the guessed letters in the bottom part of the display). `Crypto` executes the instruction

```
redisplay "false
```

to avoid that unnecessary work. `Redisplay` is also invoked by `moretext`, `lesstext`, and `showclear`. Each of these procedures uses the instruction

```
redisplay "true
```

to include `showcode`. If the flag variable weren't used, there would have to be two different versions of `redisplay`.

I used the latter technique in the procedures `bind` and `qbind`. These could also have been one procedure with a flag variable input. The advantage of the technique used in `redisplay` is that it makes the program easier to read by reducing the number of procedures, and keeping similar purposes together. The advantage of using two procedures is that it's a little faster, because you don't have to test the flag variable with `if`.

A flag variable is somewhat analogous to a *predicate,* a procedure that always outputs `true` or `false`. The advantage of using these particular values for flag variables is that they're easy to test; you can say

```
if :flag [do.something]
```

whereas, if you used some other pair of values like `yes` and `no`, you'd have to say

```
if equalp :flag "yes [do.something]
```

Some people like to give flag variables names ending with `p`, as in the convention for predicates. (The special variable `redefp` that controls redefinition of primitives in some versions of Logo, including Berkeley Logo, is an example.) I'm somewhat uncomfortable with that practice because to me it raises a confusion about whether a particular word is the name of a variable or the name of a procedure. I'd rather put `flag` in the names of flag variables.

The 26 `bound`*x* variables in this program are also flag variables; each is `true` if the corresponding letter has been guessed as the cleartext half of a binding. They don't have "flag" in their names, but their names aren't used directly in most of the program anyway. Instead they are hidden behind data abstraction procedures. `Setbound` and `setunbound` are used to set any such variable `true` or `false`, respectively; the selector `boundp` alerts you by the P in its name that it's a predicate.

## Iteration Over Letters

One of the ways in which I simplified the program for this edition was to replace some recursive helper procedures with invocations of `foreach`. At several points in the program, some action must be taken for each letter in a word, or for each word in the text.

Another kind of iteration problem that was not so easily solved by the standard higher order procedures in Berkeley Logo was one in which some action must be taken, not for each letter in a word, but for each letter in the alphabet, or for some subset of the alphabet, as in the case of `showrow`, which displays one row of the top part of the screen, with information about five consecutive letters. Of course these problems could be solved with instructions like

```
foreach "ABCDEFGHIJKLMNOPQRSTUVWXYZ [...]
```

but that seemed unaesthetic to me. I wanted to be able to specify the starting and ending letters, as in this example:

```
to alphabet
setcursor [6 6]
forletters "A "Z [ifelse boundp ? [invtype ?] [type ?]]
end
```

(The job of `alphabet` is to generate the middle part of the screen display, which is all of the letters of the alphabet, in order, with each letter in inverse video if that letter has been guessed as part of the cleartext.)

The difficulty in implementing `forletters` is to get from one letter to the next. How does a program know that the letter after A is B? Here is my solution:

```
to forletters :from :to :action
for [lettercode [ascii :from] [ascii :to]]
    [apply :action (list char :lettercode)]
end
```

The operation `ascii` takes a letter (or other character) as input. Its output is the number that represents that letter in the computer's memory. Most computers use the same numbers to represent characters; this standard representation is called ASCII, for American Standard Code for Information Interchange. (It's pronounced "ask E.") By using `ascii` to translate the starting and ending letters into numeric codes, I've

transformed the problem into one that can be solved using the standard `for` tool that allows an action to be carried out for each number in a given range.

But in the template input to `forletters`, I want the question mark to represent a letter, not its numeric code. `Char` is the inverse operation to `ascii`. Given a number that is part of the ASCII sequence, `char` outputs the character that that number represents. For example:

```
?print ascii "A
65
?print char 65
A
```

`Forletters` applies the template input to the character corresponding to the number in the `lettercode` variable controlled by the `for`.

Adding 1 to an ASCII code to get the code for the next letter depends on the fact that the numbers representing the letters are in sequence. Fortunately, this is true of ASCII. A is 65, B is 66, C is 67, and so on. Not all computer representations for characters have this property. The code that was used in the days of punched cards had the slash (/) character in between R and S!

By the way, the lower case letters have different ASCII codes from the capitals. In this program I've used the primitive operation `uppercase` to translate every character that the program reads into upper case, just to be sure that each letter has only one representation.

## Computed Variable Names

Another programming technique that is heavily used in this project is the use of `word` to compute variable names dynamically. Ordinarily, you assign a value to a variable named `var` with an instruction like

```
make "var 87
```

and you look at the value of the variable with the expression

```
:var
```

But in this project, there are variables for each letter, with names like `posna`, `posnb`, `posnc`, and so on. To assign a value to these variables, the program doesn't use 26 separate instructions like

```
make "posna [0 0]
```

(Each of these variables contains a list of screen coordinates for use with `setcursor` to find the corresponding letter in the top part of the display.) Instead, the procedure `showrow`, which draws that section of the display, contains the instruction

```
forletters :from :to [setposn ? cursor   onetop ?]
```

`Setposn` is a data abstraction procedure:

```
to setposn :letter :thing
make (word "posn :letter) :thing
end
```

When the variable `letter` contains the letter `a`, the `make` instruction has the same effect as if it were

```
make "posna :thing
```

Similarly, the dots notation (`:posna`) isn't used to examine the values of these variables. Instead, `thing` is invoked explicitly:

```
to posn :letter
output thing (word "posn :letter)
end
```

Another point to consider is that I could have used a different approach altogether, instead of using `word` to piece together a variable name. For instance, I could have used property lists:

```
to setposn :letter :thing
pprop "posn :letter :thing
end

to posn :letter
output gprop "posn :letter
end
```

As it happens, I first wrote this project in Atari 800 Logo, which didn't have property list primitives. So the question didn't arise for me. In a version of Logo that does support property lists, I see no *stylistic* reason to prefer one approach over the other. It's entirely a question of which is more efficient. Which is faster, searching through a list of 26 times 2 members (times 2 because each property has a name and a value) or concatenating strings with `word` to generate the name of a variable that can then be examined quickly?

I'd have to experiment to find out. Alternatively, instead of using `posn` as the name of a property list and the letters as names of properties, I could reverse the two roles. That would give me more lists, but shorter lists.

What *is* a stylistic issue is that using procedures like `posn` and `setposn` to isolate the storage mechanism from the rest of the program makes the latter easier to read.

## Further Explorations

I have three suggestions about how to extend this project. The first is to put in more rules by which the program can make guesses automatically. For example, a three-letter word that isn't THE might be AND. Sequences of letters within a word can also be tallied; TH is a common two-letter sequence, for example. A double letter in the cryptogram is more likely to represent OO than HH.

If you have many rules in the program, there will be situations in which two rules lead to contradictory guesses. One solution is just to try the most reliable rule first, and ignore a new guess if it conflicts with an old one. (`Qbind` applies this strategy by means of the instruction

```
if letterp thing :from [stop]
```

which avoids adding a guess to the data base if the cryptogram letter is already bound to a cleartext letter.)

Another solution would be to let the rules "vote" about guesses. If the program had many rules, it might happen that three rules suggest that F represents E, while two rules suggest that W represents E. In this case, three rules outvote two rules, and the program would guess that F represents E.

The second direction for exploration in this program is to try to make it more efficient. For example, every time you make a guess, `showclear` is invoked to redisplay the partially decoded text. Much of this redisplay is unnecessary, since most of the guesses haven't changed. How can you avoid the necessity to examine every letter of the cryptogram text? One possibility would be to keep a list, for every letter in the text, of the screen positions in which that letter appears. Then when a new guess is made, the program could just type the corresponding cleartext letter at exactly those positions. The cost of this technique would be a lot of storage space for the lists of positions, plus a slower version of `showcode`, which would have to create these position lists.

The third direction for further exploration is to find out about more complicated ciphers. For example, suppose you started with a simple substitution cipher, but every time the letter A appeared in the cleartext you shifted the corresponding cryptogram letters by one. That is, if E is initially represented by R, the first time an A appears you'd start using S to represent E. The second time A appears you'd switch to T representing E. And so on. The effect of this technique would be that a particular cleartext letter is no longer represented by a single cryptogram letter all the way through. Therefore, you can't just count the frequencies of the cryptogram letters and assume that frequently-used letters represent E and T. How could you possibly decipher such a message?

## Program Listing

```
to crypto :text
make "text map "uppercase :text
make "fulltext :text
make "moretext []
make "textstack []
copydef "letterp "namep
forletters "A "Z "initvars
make "maxcount 0
initcount "single
initcount "triple
cleartext
histogram :text
redisplay "false
if or guess.single guess.triple [showclear :text]
parseloop
end

;; Initialization

to initcount :type
setlist. :type []
setcount. :type 0
end

to initvars :letter
setcnt :letter 0
make :letter "| |
setunbound :letter
end
```

```
;; Histogram

to histogram :text
foreach :text [localmake "word filter "letterp ?
               foreach :word "histlet
               prepare.guess :word]
end

to histlet :letter
localmake "cnt 1+cnt :letter
setcursor list (index :letter) (nonneg 24-:cnt)
type :letter
setcnt :letter :cnt
if :maxcount < :cnt [make "maxcount :cnt]
end

;; Guessing letters

to prepare.guess :word
if equalp count :word 1 [tally "single :word]
if equalp count :word 3 [tally "triple :word]
end

to tally :type :word
localmake "countvar word :type :word
if not memberp :word list. :type ~
   [setlist. :type fput :word list. :type   make :countvar 0]
localmake "count (thing :countvar)+1
make :countvar :count
if :count > (count. :type) ~
   [setcount. :type :count   setmax. :type :word]
end

to guess.single
if emptyp (list. "single) [output "false]
if emptyp butfirst (list. "single) ~
   [qbind first (list. "single) "A  output "true]
qbind (max. "single) "A
qbind (ifelse equalp first (list. "single) (max. "single)
              [last (list. "single)]
              [first (list. "single)]) ~
      "I
output "true
end
```

```
to guess.triple
if emptyp (list. "triple) [output "false]
if :maxcount < (3+cnt last (max. "triple))         ~
   [qbind first (max. "triple) "T
    qbind first butfirst (max. "triple) "H
    qbind last (max. "triple) "E
    output "true]
output "false
end

;; Keyboard commands

to parseloop
forever [parsekey uppercase readchar]
end

to parsekey :char
if :char = "@ [fullclear stop]
if :char = "+ [moretext stop]
if :char = "- [lesstext stop]
if not letterp :char [beep stop]
bind :char uppercase readchar
end

;; Keeping track of guesses

to bind :from :to
if not equalp :to "| | [if not letterp :to [beep stop]
                        if boundp :to [beep stop]]
if letterp thing :from [dark thing :from]
make :from :to
fixtop :from
if letterp :to [light :to]
showclear :text
end

to qbind :from :to
if letterp thing :from [stop]
make :from :to
fixtop :from
light :to
end
```

```
;; Maintaining the display

to redisplay :flag
cleartext
showtop
alphabet
showcode :text
if :flag [showclear :text]
end

;; Top section of display (letter counts and guesses)

to showtop
setcursor [0 0]
showrow "A "E
showrow "F "J
showrow "K "O
showrow "P "T
showrow "U "Y
showrow "Z "Z
end

to showrow :from :to
forletters :from :to [setposn ? cursor   onetop ?]
print []
end

to onetop :letter
localmake "count cnt :letter
if :count = 0 [type word :letter "|      | stop]
localmake "text (word :letter "- twocol :count "- thing :letter)
ifelse :maxcount < :count+3 [invtype :text] [type :text]
type "| |
end

to twocol :number
if :number > 9 [output :number]
output word 0 :number
end

to fixtop :letter
setcursor posn :letter
onetop :letter
end
```

```
;; Middle section of display (guessed cleartext letters)

to alphabet
setcursor [6 6]
forletters "A "Z [ifelse boundp ? [invtype ?] [type ?]]
end

to light :letter
setcursor list 6+(index :letter) 6
invtype :letter
setbound :letter
end

to dark :letter
setcursor list 6+(index :letter) 6
type :letter
setunbound :letter
end

;; Bottom section of display (coded text)

to showcode :text
make "moretext []
showcode1 8 0 :text
end

to showcode1 :row :col :text
if emptyp :text [make "moretext [] stop]
if :row > 22 [stop]
if and equalp :row 16 equalp :col 0 [make "moretext :text]
if (:col+count first :text) > 37 [showcode1 :row+2 0 :text stop]
codeword :row :col first :text
showcode1 :row (:col+1+count first :text) butfirst :text
end

to codeword :row :col :word
setcursor list :col :row
invtype :word
end

;; Bottom section of display (cleartext)

to showclear :text
showclear1 8 0 :text 2
end
```

```
to showclear1 :row :col :text :delta
if emptyp :text [stop]
if :row > 23 [stop]
if keyp [stop]
if (:col+count first :text) > 37 ~
   [showclear1 :row+:delta 0 :text :delta stop]
clearword :row :col first :text
showclear1 :row (:col+1+count first :text) butfirst :text :delta
end

to clearword :row :col :word
setcursor list :col :row+1
foreach :word [ifelse letterp ? [type thing ?] [type ?]]
end

;; Windowing commands

to fullclear
cleartext
showclear1 0 0 :fulltext 1
print []
invtype [type any char to redisplay]
ignore readchar
redisplay "true
end

to moretext
if emptyp :moretext [beep stop]
push "textstack :text
make "text :moretext
redisplay "true
end

to lesstext
if emptyp :textstack [beep stop]
make "text pop "textstack
redisplay "true
end

;; Iteration tool for letters

to forletters :from :to :action
for [lettercode [ascii :from] [ascii :to]] ~
    [apply :action (list char :lettercode)]
end
```

```
;; Data abstraction (constructors and selectors)

to setbound :letter
make word "bound :letter "true
end

to setunbound :letter
make word "bound :letter "false
end

to boundp :letter
output thing word "bound :letter
end

to setcnt :letter :thing
make (word "cnt :letter) :thing
end

to cnt :letter
output thing (word "cnt :letter)
end

to setposn :letter :thing
make (word "posn :letter) :thing
end

to posn :letter
output thing (word "posn :letter)
end

to setcount.  :word :thing
make (word "count. :word) :thing
end

to count.  :word
output thing (word "count. :word)
end

to setlist.  :word :thing
make (word "list. :word) :thing
end

to list.  :word
output thing (word "list. :word)
end
```

```
to setmax.  :word :thing
make (word "max. :word) :thing
end

to max.  :word
output thing (word "max. :word)
end

;; Miscellaneous helpers

to index :letter
output (ascii :letter)-(ascii "A)
end

to beep
tone 440 15
end

to invtype :text
type standout :text
end

to nonneg :number
output ifelse :number < 0 [0] [:number]
end

;; Sample cryptograms

make "cgram1 [Dzynufqyjulli, jpqhq ok yr hoxpj qnzeujory qceqwj xhrtoyx
   zw oyjr u trhjptpolq trhln. oynqqn, rzh qceqkkogq eryeqhy tojp
   whrvlqfk rd qnzeujory uj whqkqyj kofwli fquyk jpuj jpq |xhrty-zwk| nr
   yrj pugq kzep u trhln. u nqeqyj qnzeujory uofk uj, whqwuhqk drh, u
   frhq trhjptpolq dzjzhq, tojp u noddqhqyj erffzyoji kwohoj, noddqhqyj
   reezwujoryk, uyn frhq hqul zjoloji jpuy ujjuoyoyx kjujzk uyn kuluhi.]

make "cgram2 [Lvo vfkp lfzj md opaxflimn iz lm gitokflo fnp zlkonblvon f
   hmalv'z inilifliuo, fnp fl lvo zfyo liyo lm zoo lm il lvfl vo jnmwz
   wvfl iz noxozzfkh lm xmco wilv lvo mnbminb fxliuilioz fnp xaglako md
   zmxiolh, zm lvfl viz inilifliuo xfn to kogoufnl. il iz ftzakp lm
   lvinj lvfl lviz lfzj xfn to fxxmycgizvop th zm yaxv zillinb in f tms
   dfxinb dkmnl, yfnicagflinb zhytmgz fl lvo pikoxlimn md pizlfnl
   fpyinizlkflmkz. lviz iz kflvok f wfh lm kobiyonl fnp tkfinwfzv.]
```

```
make "cgram3 [Pcodl hbdcx qxdrdlh yihcodr, hbd rzbiier gxd lih ziyqdhdlh
   hi hdgzb gwhbdlhcz echdxgzf, xdgnclp gr g ydglr ia ecudxghcil gln
   zwehcoghcil. gln c niwuh hbgh yirh ia wr jbi rdxciwref xdgn gln jxchd
   hbd dlpecrb eglpwgpd dodx edgxldn ch uf hbd xiwhd ia "xwl, rqih, xwl"
   hi rcegr ygxldx.]

make "cgram4 [Jw btn xnsgsyp ejke gfebbcg, dtyjbn fbccsksg, ryu fbccsksg
   nswcsfpsu pes usgjns, wnssuba, ryu wtptns bw pes qbtyk, pesns zbtcu
   ls yb knrujyk, yb psgpjyk svfsxp rg r psrfejyk aspebu, ryu yb
   lcrfilbrnu dtykcsg. jy wrfp, zs rns ksppjyk cbfigpsx gfesutcjyk ryu
   knrujyk pb pes xbjyp bw pbnptns.]
```

# 12 Macros

I mentioned that the versions of `for` and `foreach` shown in Chapter 10 don't work if their instruction templates include `stop` or `output` commands. The problem is that we don't want, say, `foreach` to stop; we want the procedure that *invoked* `foreach` to stop.

What we need to fix this problem is a way for a subprocedure to *make its caller* carry out some action. That is, we want something like `run`, but the given expression should be run in a different context. Berkeley Logo includes a mechanism, called *macros,* to allow this solution. As I write this in 1996, no other version of Logo has macros, although this capability is commonly provided in most versions of Logo's cousin, the programming language Lisp.*

## Localmake

Before we fix `for` and `foreach`, and even before I explain in detail what a macro is, I think it's best to start with a simple but practical example. Throughout this book I've been using a command called `localmake` that creates a local variable and assigns it a value. The instruction

```
localmake "fred 87
```

is an abbreviation for the two instructions

```
local "fred
make "fred 87
```

* Readers who are familiar with Lisp macros should take note that Logo macros do not prevent argument evaluation.

Any version of Logo will allow those two separate instructions. It's tempting to write a procedure combining them:

```
to localmake :name :value                    ;; wrong!
local :name
make :name :value
end
```

What's wrong with this solution? If you're not sure, define `localmake` as above and try an example, like this:

```
to trial
localmake "fred 87
print :fred
end

? trial
fred has no value  in trial
[print :fred]
```

When `trial` invokes `localmake`, a local variable named `fred` is created *inside the invocation of* `localmake`! That variable is then assigned the value 87. Then `localmake` returns to `trial`, and `localmake`'s local variables disappear. Back in `trial`, there is no variable named `fred`.

Here's the solution. If `localmake` is an ordinary procedure, there's no way it can create a local variable in its caller. So we have to define `localmake` as a special kind of procedure:

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

The command `.macro` is like `to`, except that the procedure it defines is a macro instead of an ordinary procedure. (It's a Logo convention that advanced primitives that could be confusing to beginners have names beginning with a period.)

It's a little hard to read exactly what this procedure does, so for exploratory purposes I'll define an ordinary procedure with the same body:

```
to lmake :name :value
output (list "local (word "" :name) "make (word "" :name) :value)
end
```

```
? show lmake "fred 87
[local "fred make "fred 87]
```

As you see from the example, `lmake` outputs a list containing the instructions that we would like its caller to carry out.

The macro `localmake` outputs the same list of instructions. But, because `localmake` is a macro, that output is then *evaluated* by the procedure that called `localmake`. If `trial` is run using the macro version of `localmake` instead of the ordinary procedure version that didn't work, the effect is as if `trial` contained a `local` instruction and a `make` instruction in place of the one `localmake` invocation. (If you defined the incorrect version of `localmake`, you can say

```
erase "localmake
```

and then the official version will be reloaded from the library the next time you use it.)

You may find the expression `(word "" :name)` that appears twice in the definition of `localmake` confusing. At first glance, it seems that there is already a quotation mark in the first input to `localmake`, namely, `"fred`. But don't forget that that quotation mark is not part of the word! For example, when you say

```
print "fred
```

Logo doesn't print a quotation mark. What the quotation mark means to Logo is "use the word that follows as the value for this input, rather than taking that word as the name of a procedure and invoking that procedure to find the input value." In this example, the first input to `localmake` is the word `fred` itself, rather than the result of invoking a procedure named `fred`. If we want to construct an instruction such as

```
local "fred
```

based on this input, we must put a quotation mark in front of the word explicitly.

In fact, so far I've neglected to deal with the fact that a similar issue about quotation may arise for the value being assigned to the variable. In the `trial` example I used the value 87, a number, which is *self-evaluating;* when a number is typed into Logo as an expression, the number itself is the value of the expression. But if the value is a non-numeric word, then a quotation mark must be used for it, too. The version of `localmake` shown so far would fail in a case like

```
localmake "greeting "hello
```

because the macro would return the list

```
[local "greeting make "greeting hello]
```

which, when evaluated, would try to invoke a procedure named `hello` instead of using the word itself as the desired value.

The most straightforward solution is to write a procedure that will include a quotation mark only when it's needed:

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

to quoted :thing
if numberp :thing [output :thing]
if listp :thing [output :thing]
output word "" :thing
end
```

A somewhat less obvious solution, but one I find more appealing, is to avoid the entire issue of quotation by putting the inputs to `make` in a list, which we can do by using `apply`:

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

On the other hand, it may take some thinking to convince yourself that the `""make` in that version is correct!

## Backquote

Even a simple macro like `localmake` is very hard to read, and hard to write correctly, because of all these invocations of `list` and `word` to build up a structure that's partly constant and partly variable. It would be nice if we could use a notation like

```
[local "NAME apply "make [NAME VALUE]]
```

for an "almost constant" list in which only the words in capital letters would be replaced by values of variables.

That particular notation can't work, because in Logo the case of letters doesn't matter when a word is used as the name of something. But we do have something almost as good. We can say

```
`[local ,[word "" :name] apply "make [,[:name] ,[:value]]]
```

The first character in that line, before the opening bracket, is a *backquote*, which is probably near the top left corner of your keyboard. To Logo, it's just an ordinary character, and happens to be the name of a procedure in the Berkeley Logo library. The list that follows the backquote above is the input to the procedure.

What the ` procedure does with its input list is to make a copy, but wherever a word containing only a comma (`,`) appears, what comes next must be a list, which is `run` to provide the value for that position in the copy. I've put the commas right next to the lists that follow them, but this doesn't matter; whenever Logo sees a bracket, it delimits the words on both sides of the bracket, just as if there were spaces around the bracket.

So if `:name` has the value `fred` and `:value` has the value 87, then this sample invocation of ` has the value

```
[local "fred apply "make [fred 87]]
```

Like macros, backquote is a feature that Berkeley Logo borrows from Lisp. It's not hard to implement:

```
to ` :backq.list
if emptyp :backq.list [output []]
if equalp first :backq.list ", ~
   [output fput run first butfirst :backq.list
               ` butfirst butfirst :backq.list]
if equalp first :backq.list ",@ ~
   [output sentence run first butfirst :backq.list
                    ` butfirst butfirst :backq.list]
if wordp first :backq.list ~
   [output fput first :backq.list ` butfirst :backq.list]
output fput ` first :backq.list ` butfirst :backq.list
end
```

This procedure implements one feature I haven't yet described. If the input to ` contains the word `,@` (comma atsign), then the next member of the list must be a list, which is `run` as for comma, but the *members* of the result are inserted in the output, instead of the result as a whole. Here's an example:

```
? show `[start ,[list "a "b] middle ,@[list "a "b] end]
[start [a b] middle a b end]
```

Using backquote, we could rewrite `localmake` a little more readably:

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

In practice, though, I have to admit that the Berkeley Logo library doesn't use backquote in its macro definitions because it's noticeably slower than constructing the macro with explicit calls to `list` and `word`.

By the way, this implementation of backquote isn't as complex as some Lisp versions. Most importantly, there is no provision for *nested* backquotes, that is, for an invocation of backquote within the input to backquote. (Why would you want to do that? Think about a macro whose job is to generate a definition for another macro.)

## Implementing Iterative Commands

It's time to see how macros can be used to implement iterative control structures like `for` and `foreach` correctly. I'll concentrate on `foreach` because it's simpler to implement, but the same ideas apply equally well to `for`.

Perhaps the most obvious approach is to have the `foreach` macro output a long instruction list in which the template is applied to each member of the list. That is, if we say

```
foreach [a b c] [print ?]
```

then the `foreach` macro should output the list

```
[print "a print "b print "c]
```

To achieve precisely this result we'd have to look through the template for question marks, replacing each one with a possibly quoted datum. Instead it'll be easier to generate the uglier but equivalent instruction list

```
[apply [print ?] [a] apply [print ?] [b] apply [print ?] [c]]
```

this way:

```
.macro foreach :data :template
output map.se [list "apply :template (list ?)] :data
end
```

(To simplify the discussion, I'm writing a version of `foreach` that only takes two inputs. You'll see in a moment that the implementation will be complicated by other considerations, so I want to avoid unnecessary complexity now. At the end I'll show you the official, complete implementation.)

This version works correctly, and it's elegantly written. We could stop here. Unfortunately, this version is inefficient, for two reasons. First, it uses another higher order procedure, `map.se`, to construct the list of instructions to evaluate. Second, for a large data input, we construct a very large instruction list, using lots of computer memory, just so that we can evaluate the instructions once and throw the list away.

Another approach is to let the `foreach` macro invoke itself recursively. This is a little tricky; you'll see that `foreach` does not actually invoke itself within itself. Instead, it constructs an instruction list that contains another use of `foreach`. For example, the instruction

```
foreach [a b c] [print ?]
```

will generate the instruction list

```
[apply [print ?] [a] foreach [b c] [print ?]]
```

Here's how to write that:

```
.macro foreach :data :template
output `[apply ,[:template] [,[first :data]]
         foreach ,[butfirst :data] ,[:template]]
end
```

In this case the desired instruction list is long enough so that I've found it convenient to use the backquote notation to express my intentions. If you prefer, you could say

```
.macro foreach :data :template
output (list "apply :template (list (first :data))
             "foreach (butfirst :data) :template)
end
```

This implementation (in either the backquote version or the explicit list constructor version) avoids the possibility of constructing huge instruction lists; the constructed list has only the computation for the first datum and a recursive `foreach` that handles the entire rest of the problem.

But this version is still slower than the non-macro implementation of `foreach` given in Chapter 10. Constructing an instruction list and then evaluating it is just a slower

process than simply doing the necessary computation within `foreach` itself. And that earlier approach works fine unless the template involves `stop`, `output`, or `local`. We could have our cake and eat it too if we could find a way to use the non-macro approach, but notice when the template tries to stop its computation. This version is quite a bit trickier than the ones we've seen until now:

```
.macro foreach :data :template
catch "foreach.catchtag
      [output foreach.done runresult [foreach1 :data :template]]
output []
end

to foreach1 :data :template
if emptyp :data [throw "foreach.catchtag]
apply :template (list first :data)
.maybeoutput foreach1 butfirst :data :template
end

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

To help you understand how this works, let's first consider what happens if the template does not include `stop` or `output`. In that case, the program structure is essentially this:

```
.macro simpler.foreach :data :template
catch "foreach.catchtag
      [this.stuff.never.invoked run [simpler.foreach1 :data :template]]
output []
end

to simpler.foreach1 :data :template
if emptyp :data [throw "foreach.catchtag]
apply :template (list first :data)
simpler.foreach1 butfirst :data :template
end
```

The instruction list that's evaluated by the `catch` runs a smaller instruction list that invokes `simpler.foreach1`. That procedure is expected to output a value, which is then used as the input to some other computation (namely, `foreach.done` in the actual version). But when `simpler.foreach1` reaches its base case, it doesn't output anything; it `throw`s back to the instruction after the `catch`, which outputs an empty list.

So all of the work of `foreach` is done within these procedures; the macro outputs an empty instruction list, which is evaluated by the caller of `foreach`, but that evaluation has no effect.

Now forget about the `simpler` version and return to the actual `foreach`. What if the template carries out a `stop` or `output`? If that happens, `foreach1` will never reach its base case, and will therefore not `throw`. It will either stop or output a value. The use of `.maybeoutput` in `foreach1` is what makes it possible for `foreach1` to function either as a command (if it stops) or as an operation (if it outputs) without causing an error when it invokes itself recursively. If the recursive invocation stops, so does the outer invocation. If the recursive invocation outputs a value, the outer invocation outputs that value.

`Foreach` invoked `foreach1` using Berkeley Logo's `runresult` primitive operation. `Runresult` is just like `run`, except that it always outputs a value, whether or not the computation that it runs produces a value. If so, then `runresult` outputs a one-member list containing the value. If not, then `runresult` outputs an empty list.

The output from `runresult` is used as input to `foreach.done`, whose job is to construct an instruction list as the overall output from the `foreach` macro. If the input to `foreach.done` is empty, that means that the template included a `stop`, and so `foreach` should generate a `stop` instruction to be evaluated by its caller. If the input isn't empty, then the template included an `output` instruction, and `foreach` should generate an `output` instruction as its return value.

This version is quite fast, and handles `stop` and `output` correctly. It does not, however, handle `local` correctly; the variable will be local to `foreach1`, not to the caller. It was hard to decide which version to use in the Berkeley Logo library, but slowing down every use of `foreach` seemed too high a price to pay for `local`. That's why, for example, procedure `onegame` in the solitaire program of Chapter 4 includes the instructions

```
local map [word "num ?] :numranks
foreach :numranks [make word "num ? 4]
```

instead of the more natural

```
foreach :numranks [localmake word "num ? 4]
```

That single instruction would work with the first implementation of `foreach` in this chapter, but doesn't work with the actual Berkeley Logo implementation!

## Debugging Macros

It's easy to make mistakes when writing a macro, because it's hard to keep straight what has to be quoted and what doesn't, for example. And it's hard to debug a macro, because you can't easily see the instruction list that it outputs. You can't say

```
show foreach ...
```

because the output from `foreach` is *evaluated,* not passed on to `show`.

One solution is to trace the macro.

```
? trace "foreach
? foreach [a b c] [print ?]
( foreach [a b c] [print ?] )
a
b
c
foreach outputs []
? foreach [a b 7 c] [if numberp ? [stop] print ?]
( foreach [a b 7 c] [if numberp ? [stop] print ?] )
a
b
foreach outputs [stop]
Can only use stop inside a procedure
```

In this case, I got an error message because, just as the message says, it doesn't make sense to use `stop` in a template unless this invocation of `foreach` is an instruction inside a procedure definition. Here I invoked `foreach` directly at the Logo prompt.

The Berkeley Logo library provides another solution, a `macroexpand` operation that takes as its input a Logo expression beginning with the name of a macro. It outputs the expression that the macro would output, without causing that expression to be evaluated:

```
? show macroexpand [foreach [a b 7 c] [if numberp ? [stop] print ?]]
a
b
[stop]
```

This time I didn't get an error message, because the instruction list that `foreach` outputs wasn't actually evaluated; it became the input to `show`, which is why it appears at the end of the example.

`Macroexpand` works by using `define` and `text` to define, temporarily, a new procedure that's just like the macro it wants to expand, but an ordinary procedure instead of a macro:

```
to macroexpand :expression
define "temporary.macroexpand.procedure text first :expression
...
end
```

You might enjoy filling in the rest of this procedure, as an exercise in advanced Logo programming, before you read the version in the library.

(What if you want to do the opposite, defining a macro with the same text as an ordinary procedure? Berkeley Logo includes a `.defmacro` command, which is just like `define` except that the resulting procedure is a macro. We don't need two versions of `text`, because the text of a macro looks just like the text of an ordinary procedure. To tell the difference, there is a primitive predicate `macrop` that takes a word as input, and outputs `true` if that word is the name of a macro.)

## The Real Thing

Here is the complete version of `foreach`, combining the macro structure developed in this chapter with the full template flexibility from Chapter 10.

```
.macro foreach [:foreach.inputs] 2
catch "foreach.catchtag ~
      [output foreach.done runresult [foreach1 butlast :foreach.inputs
                                                last :foreach.inputs 1]]
output []
end

to foreach1 :template.lists :foreach.template :template.number
if emptyp first :template.lists [throw "foreach.catchtag]
apply :foreach.template firsts :template.lists
.maybeoutput foreach1 butfirsts :template.lists ~
                      :foreach.template :template.number+1
end

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

And here, without any discussion, is the actual library version of `for`. This, too, combines the ideas of this chapter with those of Chapter 10.

```
.macro for :for.values :for.instr
localmake "for.var first :for.values
localmake "for.initial run first butfirst :for.values
localmake "for.final run item 3 :for.values
localmake "for.step forstep
localmake "for.tester (ifelse :for.step < 0
                              [[(thing :for.var) < :for.final]]
                              [[(thing :for.var) > :for.final]])
local :for.var
catch "for.catchtag [output for.done runresult [forloop :for.initial]]
output []
end

to forloop :for.initial
make :for.var :for.initial
if run :for.tester [throw "for.catchtag]
run :for.instr
.maybeoutput forloop ((thing :for.var) + :for.step)
end

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

to forstep
if equalp count :for.values 4 [output run last :for.values]
output ifelse :for.initial > :for.final [-1] [1]
end
```

# 13 Example: Fourier Series Plotter

Program file for this chapter: `plot`

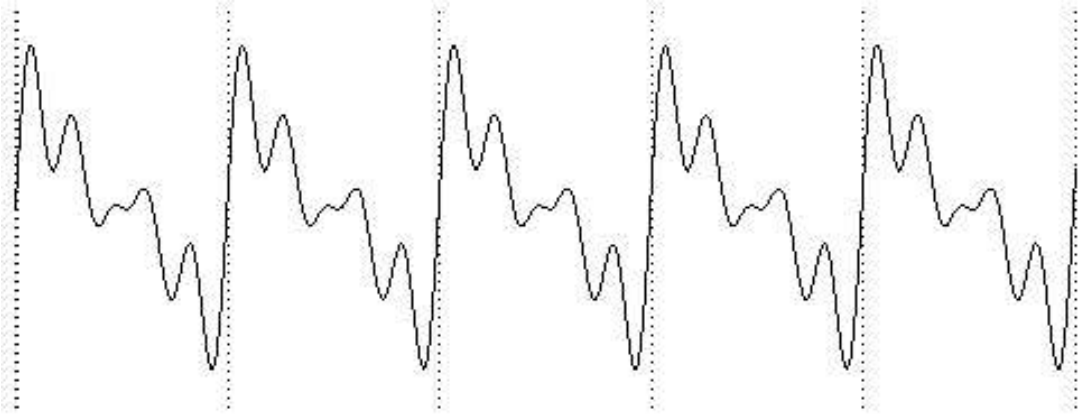
A particular musical note (middle C, say) played on a piano and played on a violin sound similar in some ways and different in other ways. Two different notes played on the violin also have similarities and differences. How do you hear which note is being played, and how do you know what instrument you're listening to?

To do justice to these questions would fill up an entire book. For example, a piano produces sound when a felt-covered wooden hammer hits metal wires, or strings. Each piano key controls one hammer, but each hammer may hit from one to three strings. It turns out that the strings for a particular note are not tuned to exactly the same pitch. Part of the richness of the piano's sound comes from the interplay of slightly different pitches making up the same note.

Another contributing factor to the recognition of different instruments is their differences in attack and decay. Does the sound of a note start abruptly, or gradually? The differences are not only a matter of loudness, though. A few instruments start out each note with a very pure, simple tone like a tuning fork. Gradually, the tone becomes more complex until it finally reaches the timbre you associate with the instrument. But a bowed violin, a more typical example, starts out each note almost as a burst of pure noise, as the bow hits the strings, and gradually mellows into the sound of a particular note. If you are experimentally inclined, try tape recording the same note as played by several instruments. Then cut out the beginnings and ends of the notes, and retain only the middle section. Play these to people and see how well they can identify the instruments, compared to their ability to identify the complete recorded notes.

For this chapter, though, I'm going to ignore these complications, and concentrate on the differences in the *steady-state* central part of a note as played by a particular instrument. What all such steady musical sounds have in common is that they are largely *periodic*. This means that if you graph the air pressure produced by the instrument over

time (or the voltage when the sound is represented electrically in a hifi system), the same pattern of high and low pressures repeats again and again. Here is an example. In this picture, the motion of your eye from left to right represents the passing of time.

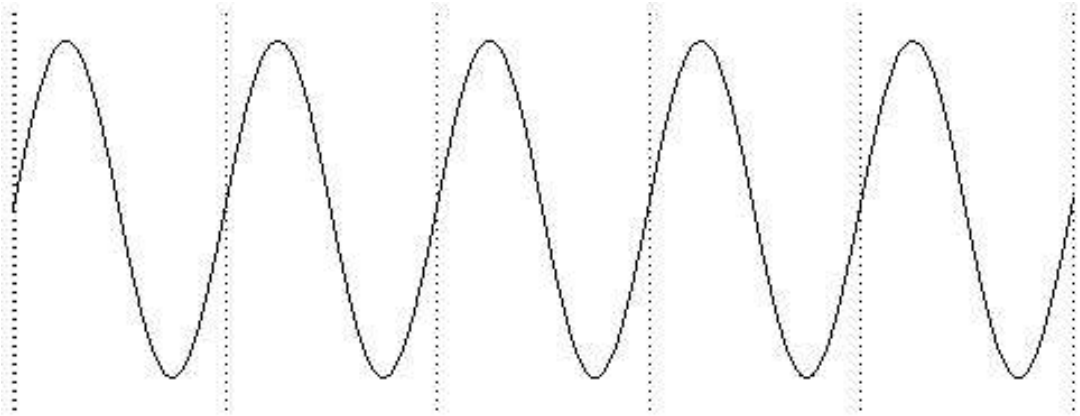


The height of the squiggle on the page, at any particular moment, represents the sound pressure at that moment. So what this picture shows is that there are many small up-and-down oscillations superimposed on one large, regular up-and-down motion. (This one large oscillation is called the *fundamental* frequency.) You can see that the entire picture consists of five repetitions of a smaller squiggle with just one of the large oscillations.

From what I've said about oscillations, you might get the impression that this is a picture of something like a foghorn or siren, in which you can hear an alternation of loud and soft moments. But this is actually the picture of what sounds like a perfectly steady tone. The entire width of the page represents about one one-hundredth of a second. There are a few hundred repetitions of the single large up-and-down cycle in each second of a musical note. The exact number of repetitions is the *frequency* of the note, and is the same for every instrument. For example, the note A above middle C has a pitch of 440 cycles per second, or 440 Hertz.

All instruments playing A above middle C will have a picture with the same fundamental frequency of 440 Hertz. What is different from one instrument to another is the exact shape of the squiggle. (By the way, the technical name for a squiggle is a *waveform*. You can see the waveform for a note by connecting a microphone to an oscilloscope, a device that shows the waveform on a TV-like screen.)

Here is a picture of the simplest, purest possible tone:

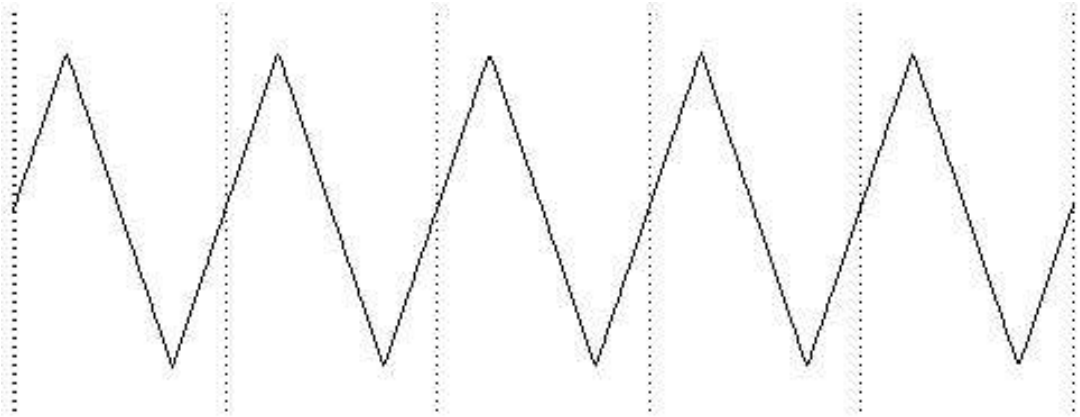


This is the waveform you'd get from an ideal tuning fork, with no impurities or bumps. It is called a *sine wave*. This particular kind of oscillation turns up in many situations, not just musical sounds. For example, suppose you write a program that starts a turtle moving in a circle forever.

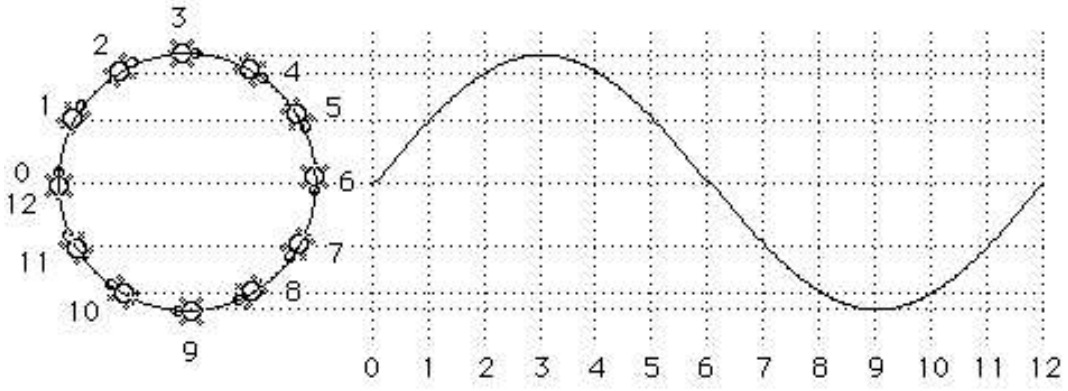
```
to circle
fd 1
rt 1
circle
end
```

Think about the motion of the turtle, and concentrate only on its vertical position on the screen. Never mind its motion from left to right. The up-and-down part of the turtle's motion over time looks just like this sine wave.

This says more than simply that the turtle alternates moving up and down. For example, the turtle's vertical motion might have looked like this over time:



If this were the picture of the turtle's motion, it would mean that the turtle's vertical position climbed at a steady rate until it reached the top of the circle, then abruptly turned around and started down again. But in fact what happens is that the height of the turtle changes most quickly when the turtle is near the "Equator" of its circle. The turtle's vertical speed gets less and less as the turtle gets near the "poles." This speed change corresponds to the gradual flattening of the sine wave near the top and bottom. (You may find it confusing when I say that the turtle's vertical motion slows down, because the turtle's speed doesn't seem to change as it draws. But what happens is that near the Equator, the turtle's speed is mostly vertical; near the poles, its speed is mostly horizontal. We aren't thinking about the horizontal aspect of its motion right now.)



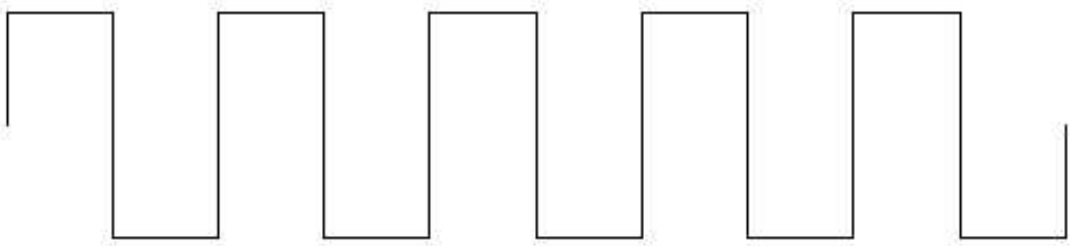

What makes sine waves most important, though, is that *any* periodic waveform can be analyzed as the sum of a bunch of sine waves of different frequencies. (Sometimes an infinite number of since waves must be added together.) The frequencies of the sine waves will always be multiples of the fundamental frequency. This important mathematical result was discovered by the French mathematician Jean-Baptiste-Joseph Fourier (1768–1830). The representation of a mathematical function as a sum of sine waves is called a *Fourier series*.

For example, when a violin plays A above middle C, the waveform that results will include a sine wave with frequency 440 Hertz, one with frequency 880 Hertz, one at 1320 Hertz, and so on. Not all of these contribute equally to the complete waveform. The *amplitude* of each sine wave (the amount of swing, or the vertical distance in the picture) will be different for each. Typically, the fundamental frequency has the largest amplitude, and the others (which are called *harmonics* or *overtones*) have smaller amplitudes. The precise amplitudes of each harmonic are what determine the steady-state timbre of a particular instrument.

## Square Waves

Two traditional musical instruments, the clarinet and the pipe organ, share a curious characteristic: their Fourier series contain only odd harmonics. In other words, if a clarinet is playing A above middle C, the waveform includes frequencies of 440 Hertz, 1320 Hertz (3 times 440), 2200 Hertz (5 times 440), and so on. But the waveform does not include frequencies of 880 Hertz (2 times 440), 1760 Hertz (4 times 440), and so on. (I'm oversimplifying a bit in the case of the pipe organ. What I've said about only odd harmonics is true about each pipe, but the organ can be set up to combine several pipes in order to include even harmonics of a note.)

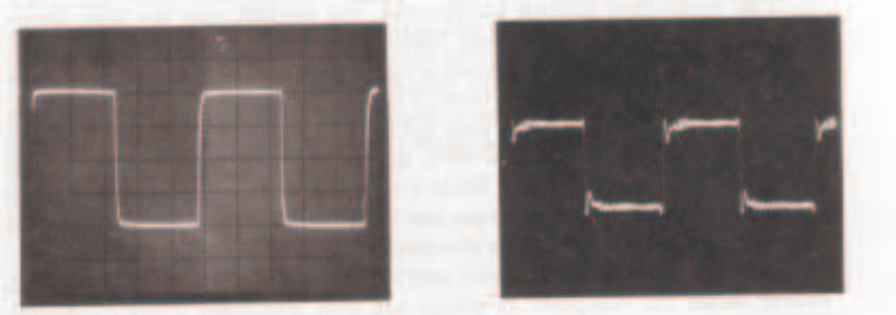
In recent times, a third musical instrument has come to share this peculiar Fourier series: the computer. (Perhaps you were wondering where computers come into this.) Today there are computer-controlled musical instruments that can generate any possible sound. Musicians have even used computers to create new instrument timbres that are not possible with ordinary instruments. But the particular timbre that most people associate with computer music is the one produced by the simplest possible computer sound generator. Instead of a steady oscillation in sound pressure, this simple device can only be on or off at a given moment. The computer produces sound by flipping the device from on to off and back at a particular rate. Such a device produces a *square wave*, like this:



No sound that occurs in nature has a waveform that turns corners so abruptly. But what is "natural" in nature isn't necessarily what's "natural" for a computer. For many years, computer-generated music invariably meant square waves except in very fancy music research centers.

More recently, new integrated circuit technology has made it relatively inexpensive to equip computers with "music chips" that generate sine waves. The stereotyped sound of computer music is becoming uncommon. But I still find square waves fascinating for several reasons.

One place where square waves are still used is in the hifi magazines, in their tests of amplifiers. The testing laboratories feed a square wave into an amplifier, and show oscilloscope pictures of the waveform going into the amp and the waveform coming out. Here is an example:



The oscillation that is visible in the output near the corners of the input is called *ringing*. A lot of ringing indicates that the amplifier doesn't have good high-frequency response.

Here is why a square wave is a good test of high frequencies: The Fourier series corresponding to the square wave includes an infinite number of odd-harmonic sine wave components. In other words, a perfect square wave includes infinitely high frequencies. (In practice, the input picture isn't a perfect square wave. You can see that the vertical segments aren't *quite* truly vertical, for example.) No amplifier can reproduce infinitely high frequencies faithfully. The result is that the output from the amplifier includes only some of the harmonics that make up the input. It turns out that such a *partial series*, with relatively few of the harmonics included, produces a waveform in which the ringing phenomenon at the corners is clearly visible.

If you think about it, that's a bit unexpected. Normally, the more harmonics, the more complicated the waveform. For example, the simplest waveform is the one with only the fundamental, and no added harmonics. Yet, *removing* harmonics from the square wave produces a *more* complicated picture. I like paradoxes like that. I wanted to write a computer program to help me understand this one.

Before you can look into the square wave in detail, you have to know not only the fact that it uses odd harmonics, but also the amplitude of each harmonic. A square wave with fundamental frequency $f$ has this formula:

$$\frac{\sin(fx)}{1} + \frac{\sin(3fx)}{3} + \frac{\sin(5fx)}{5} + \cdots$$

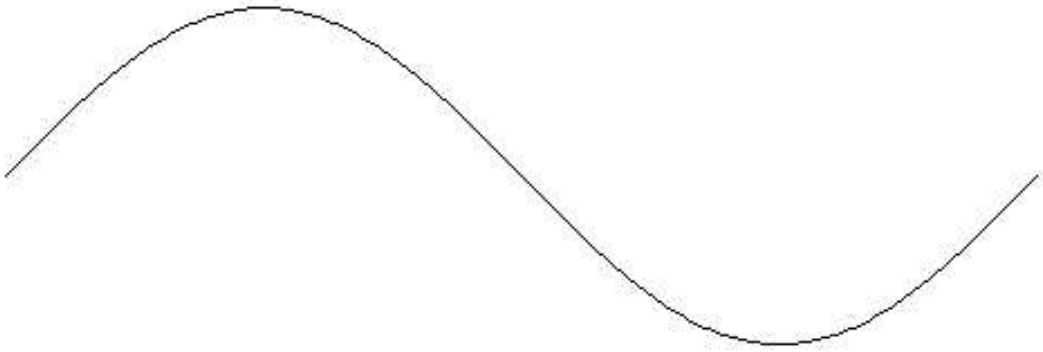
The dots at the end indicate that this series goes on forever. The amplitude of each sine wave is the reciprocal of the harmonic number (one divided by the number).

This project draws pictures of waveforms containing some number of terms of this series. (Each sine wave is called a term.) The program allows many different ways of controlling exactly what is drawn.

To start with something very simple, try this instruction:

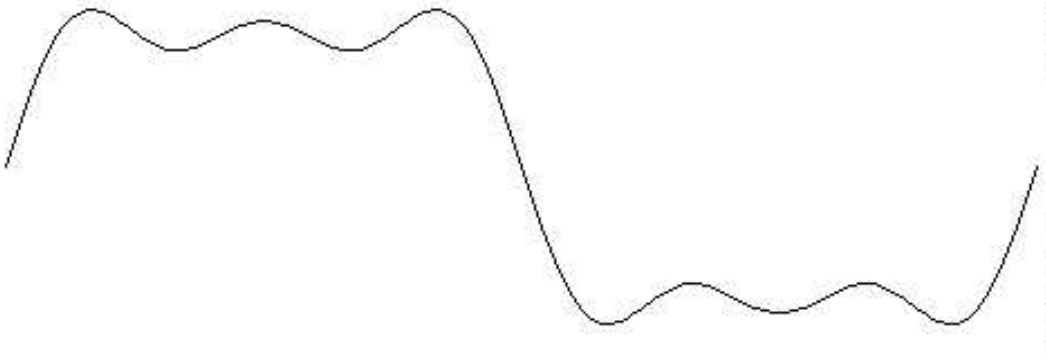
```
plot 1
```

The effect of this command is to draw one cycle of a pure sine wave:



This is the first term of the series for the square wave. Now try this:

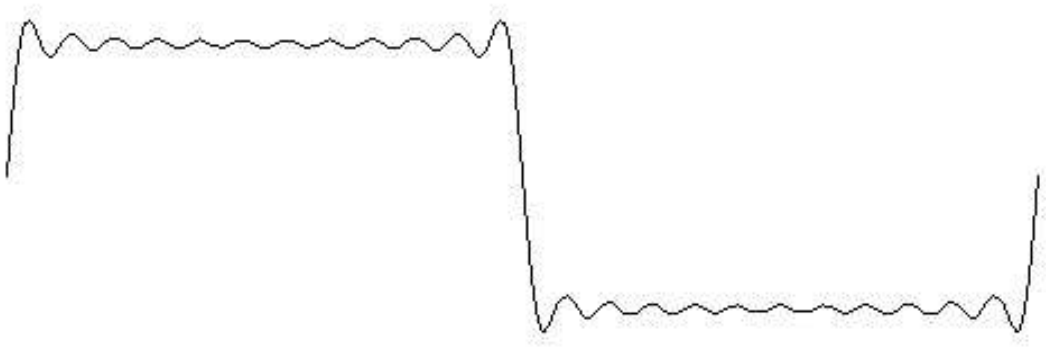
```
plot 5
```



The input to `plot` is the harmonic number of the highest harmonic. In this example, we've drawn three sine waves added together: the fundamental, third harmonic, and fifth harmonic.

To see a plot looking somewhat more like the pictures in the amplifier tests, try

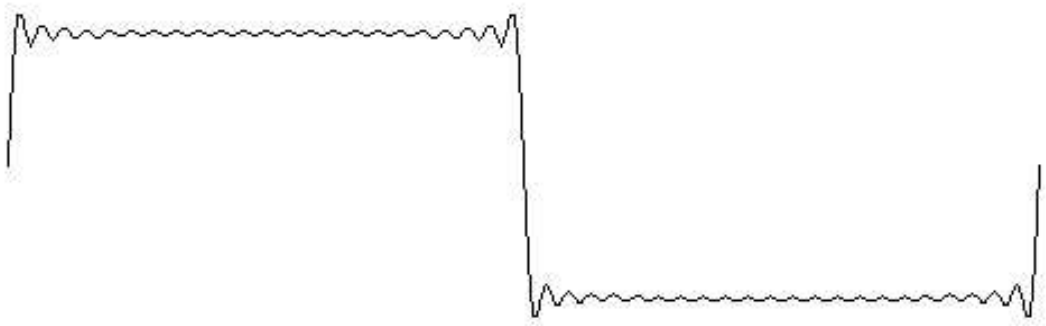
```
plot 23
```



This contains the first 12 odd harmonics. (Remember to use an odd number as input, if you want to see something that looks like a square wave.) You can see that the result still includes some oscillation in the horizontal sections, but does have an overall square shape.

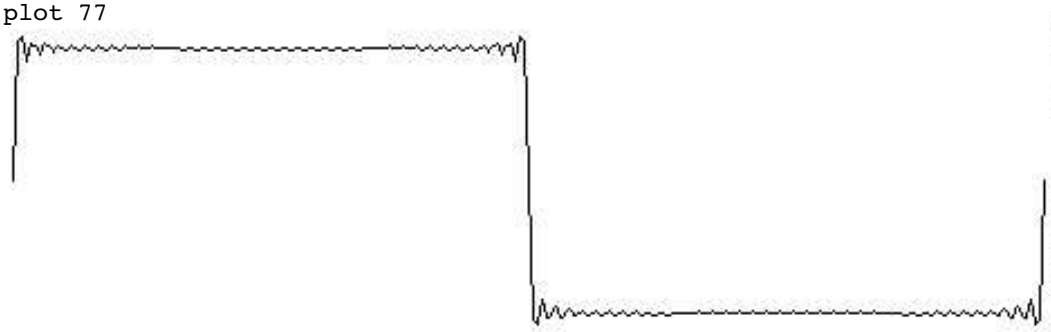
A mediocre hifi amp has a frequency response that is good to about 20,000 Hertz. This is about the 45th harmonic of 440 Hertz. To see how A above middle C would come out on such an amplifier, try

```
plot 45
```



There is still some ringing near the corners, but the middle of the horizontal segment is starting to look really flat. A better amplifier might be good to 30,000 Hertz. To see how that would look, try

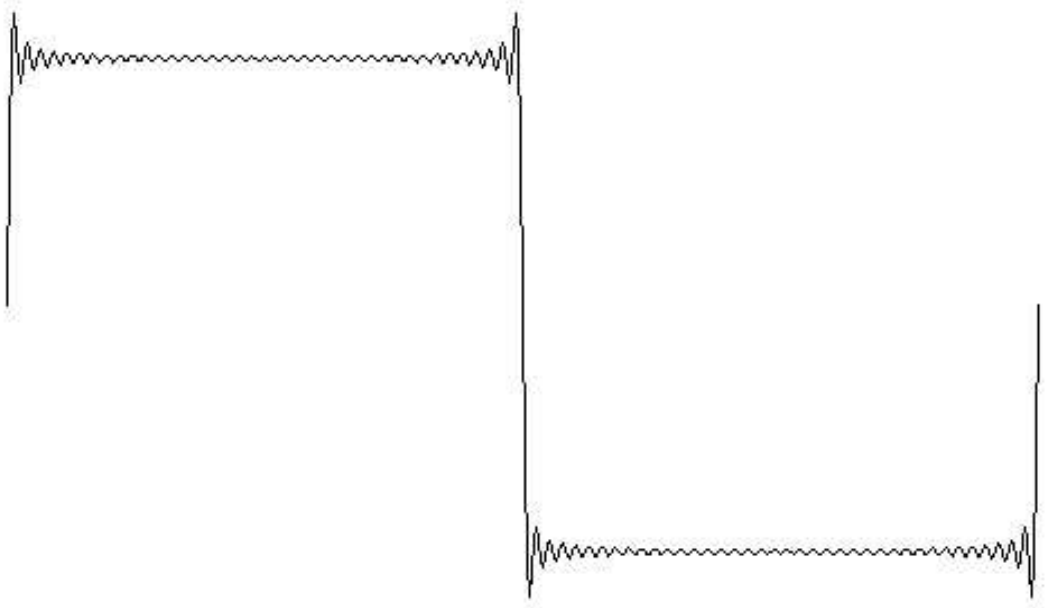
```
plot 77
```



(The drawing of the picture takes longer when you use a larger input to `plot`, because the program has to calculate more terms of the series.)

So far, we have only changed one of the possible parameters controlling the waveform, namely the highest harmonic. The program allows you to control several other elements of the picture. For example, try this:

```
plot [maxharm 77 yscale 140 deltax 1]
```

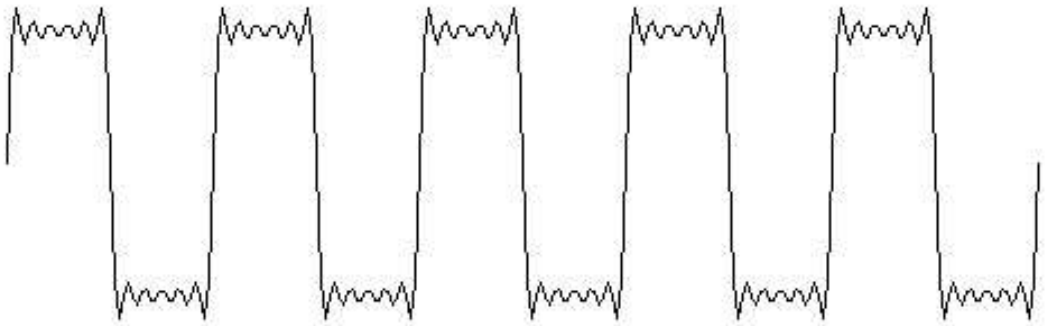


`Plot` takes one input, but this time the input is a list instead of a single number. The members of the list are used as sort of "sub-inputs." The odd-numbered members are the *names* of parameters, for which the even-numbered members provide *values*.

`Maxharm` stands for "maximum harmonic"; it is the parameter you were setting when you used a single number as the input. `Yscale` is an adjustment for the height of the plot. (To "scale" a bunch of numbers means to multiply all of them by some constant value, the "scale factor.") You may have noticed that as the number of harmonics has increased, the pictures have been getting smaller in the vertical direction; by increasing the value of `yscale` we can expand the height of the plot to show more detail. Similarly, `deltax` allows us to show more horizontal detail, not by widening the picture but by computing the value for every dot. Ordinarily, the program saves time by calculating every second dot. This approximation is usually good enough, but sometimes not. (`Deltax` means "change in X." Delta is the name of the Greek letter D (Δ), which mathematicians use to represent a small change in something.)

Here's another example:

```
plot [11 cycles 5]
```



`Cycles` indicates the number of complete cycles you want to see. By saying `cycles 5` in this example, I drew a picture like the ones near the beginning of this chapter, with five repetitions of the fundamental oscillation.

Notice also that we didn't have to say `maxharm`. If a number appears in the input list where a name should be, it's automatically assigned to `maxharm`.

`Plot` allows you to specify any of six parameters. Each parameter has a *default* value, the value that is used if you don't say anything about it. For example, the default value for `deltax` is 2. Here are all the parameters:

| name | default | purpose |
|---|---|---|
| `maxharm` | 5 | highest harmonic number included in series |
| `deltax` | 2 | number of turtle steps skipped between calculations |
| `yscale` | 75 | vertical motion is multiplied by this number |
| `cycles` | 1 | number of cycles of fundamental shown |
| `xrange` | 230 | highest X coordinate allowed |
| `skip` | 2 | number of harmonics skipped between terms |

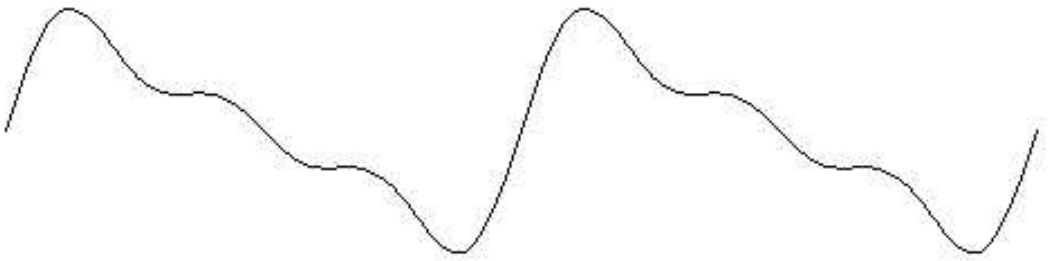
You've already seen what `maxharm`, `yscale`, `deltax`, and `cycles` are for. Now I'll explain the others.

`Xrange` is mainly changed when moving the program from one computer to another. Each computer allows a particular number of turtle steps to fit on the screen in each dimension, horizontal and vertical. `Xrange` is the largest horizontal position `plot` is allowed to use. This is set a little below the largest possible X coordinate, just to make sure that there is no problem with wrapping around the screen.

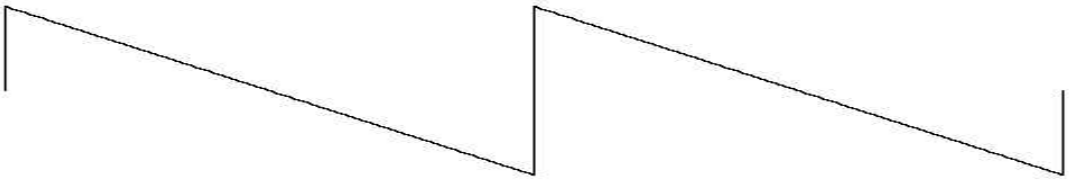
`Skip` is the number of harmonics skipped between terms. To get odd harmonics, which we need for the square wave, we have to skip by 2 each time, from 1 to 3, from 3 to 5, and so on. Different values for `skip` will give very different shapes.

For example, if you are at all adventurous, you must have tried an even value of `maxharm` a while ago, getting a result like this:

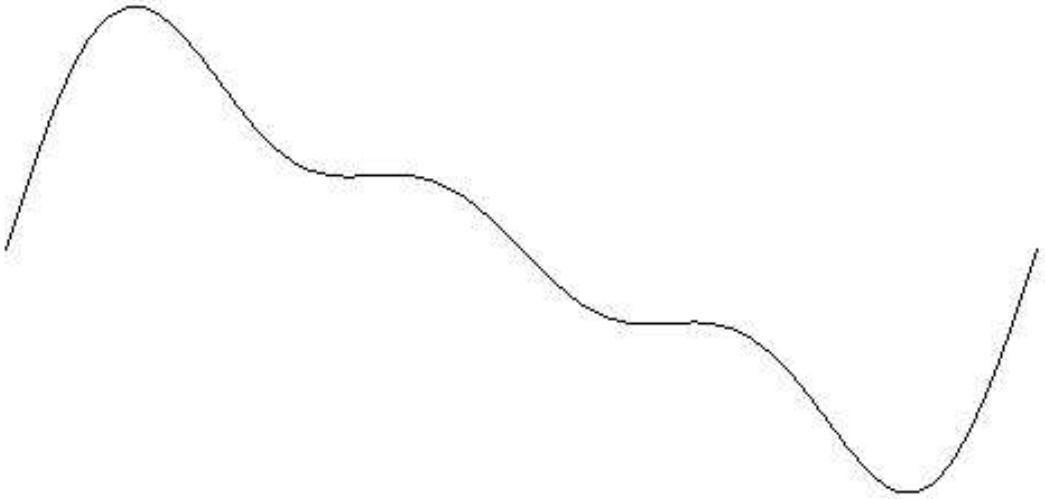
```
plot 6
```



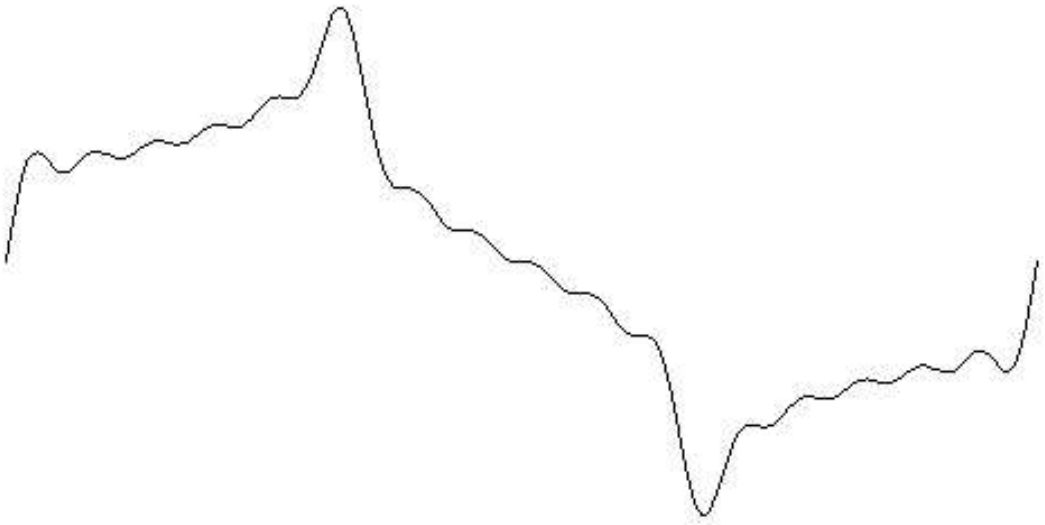
What you see is two cycles of an approximation to another shape, the *sawtooth*:



Why two cycles? Well, `plot 6` uses the second, fourth, and sixth harmonics. Supposing that the fundamental frequency is 440 again, this means that `plot` added frequencies of 880, 1760, and 2640 Hertz. But these are also the fundamental, second harmonic, and third harmonic of 880 Hertz. By choosing only even harmonics, you've essentially chosen *all* the harmonics of *twice the fundamental frequency* you had in mind. It is this doubling of the fundamental frequency that produces two cycles on the screen. You could get one cycle of the same waveform by saying `plot [3 skip 1]`:



You can see much more bizarre waveforms by using other values of `skip`. The best one I've found is `plot [16 skip 3]`:



I chose a `maxharm` of 16 because it includes the fundamental plus five additional harmonics (4, 7, 10, 13, 16). If I'd made `maxharm` 15 or 17, I wouldn't have included the fundamental.

## Keyword Inputs

There are two different points of interest about this project. One is the whole business of waveforms and Fourier series. The second is the use of *keyword* inputs, which is the name for this system of giving information to `plot`. The more usual style of Logo programming would have been to make `plot` a procedure with six inputs. To draw a default graph, you would then have had to say

```
plot 5 2 75 1 230 2
```

Since most of the time you want to use the default values for most of the inputs, all this typing would be an annoyance. It would also be easy to make a mistake about the correct order of the inputs. (This more usual Logo technique is called *positional* inputs.) The combination of many necessary inputs with standard values for most of them makes the keyword technique appropriate here. It isn't always appropriate. You wouldn't want to have to say

```
print item [index 2 list [vanilla chocolate strawberry]]
```

because you have no trouble remembering which input to `item` is which, and you always want to provide both of them.

The procedure that interprets the keyword inputs is called `keyword`. `Keyword` was written to be a general tool, not limited to this particular program. It takes two inputs. The first is the input that you, the user, provide. The second is a list of defaults. When `plot` invokes `keyword`, the second input is this:

```
[maxharm 5 deltax 2 yscale 75 cycles 1 xrange 230 skip 2]
```

This input tells `keyword` the names of all the keyword inputs as well as their default values. It's in the same form as the actual input you give (a list of alternating names and values), and in fact `keyword` uses a single subprocedure, first to process the default list and then to process your input.

`Keyword` is actually not *perfectly* general because it uses the assumption that all the values it gets are numeric. The virtue of this assumption is that it allows `keyword` to

recognize a number without a name as implicitly referring to the `maxharm` keyword. (The name `maxharm` is not built into the procedure. Instead, the first name in the list of default values is used.) To use `keyword` in a context in which non-numeric words could be values as well as names, this assumption would have to be removed.

I didn't have keyword inputs in mind from the beginning. When I started working on this project, the only input to `plot` was what I now call `maxharm`, the highest harmonic number to include. All the other numbers were "wired in"; if I wanted to change something like what is now called `:xrange`, I'd edit all the procedures and change the numbers in the editor.

Editing all the procedures wasn't too difficult, since without the keyword-processing procedures everything fits in a single screenful. Changing the resolution (what is now `:deltax`) was a bit annoying, since I had to edit three different parts of the program. (You can see that `:deltax` appears three times in the final version.) When I finally got tired of that editing process, I decided to use keyword inputs.

## Making the Variables Local

The job of `keyword` is to create variables, one for each keyword, and assign a value to each variable. If the user provides a value for a particular keyword, that's the value to use; if not, the default value is used.

When I first did this project, I wrote a version of `keyword` that creates global variables for the keywords:

```
to keyword :inputs :defaults
if or (wordp :inputs) (numberp first :inputs) ~
   [make "inputs sentence (first :defaults) :inputs]
setup.values :defaults
setup.values :inputs
end

to setup.values :list
if emptyp :list [stop]
make first :list first butfirst :list
setup.values butfirst butfirst :list
end
```

`Keyword` checks for the special cases of a single number (as in `plot 5`) or a list beginning with a number; in either case, a new list is made with the first keyword (`maxharm`) inserted before the number. Then the default values are assigned to all the keyword variables, and

finally the user's values are assigned to whatever keywords the user provided, replacing the defaults.

Since these keyword variables are only used within the `plot` program, it would be cleaner to make them local to `plot`, just as ordinary positional inputs are automatically local to a procedure. I could have had `plot` take care of this before calling `keyword`:

```
to plot :inputs
local [maxharm deltax yscale cycles xrange skip]
keyword :inputs [maxharm 5 deltax 2 yscale 75 cycles 1 xrange 230 skip 2]
...
```

but I thought it would be unaesthetic to have to type the names twice! What I really want is for `keyword` to be able to make the variables local. But I can't just say

```
to keyword :inputs :defaults
local filter [not numberp ?] :defaults
if or (wordp :inputs) (numberp first :inputs) ~
   [make "inputs sentence (first :defaults) :inputs]
setup.values :defaults
setup.values :inputs
end
```

because that would make the variables local to `keyword` itself, not to its caller, `plot`. This is the same problem I had in writing `localmake` in Chapter 12, and the solution is the same: Make `keyword` a macro!

```
.macro keyword :inputs :defaults
if or (wordp :inputs) (numberp first :inputs) ~
   [make "inputs sentence (first :defaults) :inputs]
output '[local ,[filter [not numberp ?] :defaults]
         setup.values ,[:defaults]
         setup.values ,[:inputs]]
end
```

Now it will be `plot`, instead of `keyword`, that creates the local variables and calls `setup.values`.

## Indirect Assignment

The actual assignment of values to the keywords is a good illustration of indirect assignment in Logo. The instruction that does the assignment is this:

```
make first :list first butfirst :list
```

Usually the first input to `make` is an explicit quoted word, but in this program the variable names are computed, not explicit. This technique would be impossible in most programming languages.

## Numeric Precision

It's important that the program computes the Fourier series starting with the higher harmonic numbers, adding in the fundamental term last. Recall the formula for the series:

$$\frac{\sin(fx)}{1} + \frac{\sin(3fx)}{3} + \frac{\sin(5fx)}{5} + \cdots$$

The value of the sine function for each term is divided by the harmonic number of the term. In general, this means that the terms for higher numbered harmonics contribute smaller values to the sum.

Theoretically, it shouldn't matter in what order you add up a bunch of numbers. But computers carry out numeric computations with only a limited precision. Usually there is a particular number of *significant digits* that the computer can handle. It doesn't matter how big or small the number is. The numbers 1234, 1.234, and 0.00000001234 all have four significant digits.

To take a slightly oversimplified case, suppose your computer can handle six significant digits. Suppose that the value of the fundamental term is exactly 1. Then the computer could add 0.00001 to that 1 and get 1.00001 as the result. But if you tried to add 0.000001 to 1, the result (1.000001) would require seven significant digits. The computer would round this off to exactly 1.

Now suppose that the 23rd term in some series is 0.000004, the 24th term is 0.000003, and the 25th is 0.000002. (I just made up these values, but the general idea that they'd be quite small is true.) Suppose we are adding the terms from left to right in the formula, and the sum of the first 22 terms is 2.73. Adding the 23rd term would make it 2.730004, which is too many significant digits. This sum would be rounded off to 2.73 again. Similarly, the 24th and 25th terms would make absolutely no difference to the result.

But now suppose we add up the terms from right to left. The sum of the 25th and 24th terms is 0.000005, and adding in the 23rd term give 0.000009. If we were to add this to 2.73 the result would be 2.730009. Although this is still too many significant digits, the computer would round it off to 2.73001. The three terms at the end *would* make a small difference in the result.

In the square wave series, the successive terms get smaller quite slowly. You'd have to add very many terms before the problem I'm describing would really be important. But other series have terms that get smaller quickly, so that even for a small number of terms it's important to add in the smaller terms before the larger ones.

By the way, the procedure `series` that computes the value of the series for some particular `x` value is written recursively, but its task is iterative. I could have said

```
to series
localmake "result 0
for [harmonic :maxharm 1 [-:skip]] ~
    [make "result :result + (term :harmonic)]
output :result
end
```

but the use of `make` to change the value of a variable repeatedly isn't very good Logo style. What I really want is an *operation* corresponding to `for`, analogous to `map` as the operation corresponding to `foreach`. Then I could say

```
to series
output accumulate "sum [harmonic :maxharm 1 [-:skip]] [term :harmonic]
end
```

You might enjoy using the techniques of Chapter 10 to implement `accumulate`.

## Dynamic Scope

One final point about the programming style of this project has to do with the use of Logo's dynamic scope. Every procedure has access to the variables of its superprocedures, and this project takes advantage of the fact. Many people think it's better style if every procedure is given all the information it needs as inputs. I didn't follow that rule in this project because, as I've said, many of the variables were invented late in the development process, and I did as little rewriting as possible.

For example, here is the procedure that computes one term of the Fourier series:

```
to term :harmonic
output (sin :xscale * :harmonic * :x) / :harmonic
end
```

Of the three numbers that are used in this computation, `:xscale` is constant throughout the program, so it's not unreasonable for it to be used globally. But `:x` changes for every point. `X` is, in fact, a local variable, but it's local to `plot` rather than to `term`.

Why does it matter? One reason is that people find it easier to understand a program if it's made very clear where each variable gets its value. In this example, it's easy to miss the fact that `x` is the index variable in a `for` loop. The second reason is that as written, `term` can be used only within `plot`. If we wanted to compute the value of a Fourier series for some reason other than plotting it, we couldn't easily use the same procedures.

I decided to leave the program as it is, instead of cleaning it up for publication, so that you could see an example of a style that, after all, Logo makes possible by choosing to use dynamic scope. In some ways, this style is simpler than one in which each procedure would have several more inputs. You can decide for yourself if you find it easier or harder to read the program this way, compared to the officially approved style:

```
to term :x :harmonic :xscale
output (sin :xscale * :harmonic * :x) / :harmonic
end
```

In this version of the procedure, I've made `x` the first input to `term`. This seems most natural to me, considering that `term`'s job is to compute an algebraic function of `x`. The argument to a function is very naturally represented in a computer program as the input to a procedure.

All I've changed in `term` itself is the title line. But of course the invocation of `term`, which is in the procedure `series`, must be changed also. And to be consistent, `series` should get *its* needed values as explicit inputs, too:

```
to series :x :harmonic :skip :xscale
if :harmonic < 1 [output 0]
output (term :harmonic)+(series :harmonic-:skip)
end
```

We're up to four inputs, and it'd be easy to get them out of order when `series` is invoked by `plot`. You can see why "inheriting" variables from a procedure's caller can simplify its use.

## Further Explorations

The fact that each term in the series is divided by `:harmonic` limits this program to a particular family of waveforms, the family that includes square waves and sawtooth waves. In general, real musical instruments don't have such regularity in the extent to which each term contributes to the sum. For example, I started by saying that clarinets and

pipe organs are made of odd harmonics, just as square waves are. But clarinets don't sound like organs, and neither sound like square waves. There is a family resemblance, but there are definite differences too. The differences are due to the different "weights" that each instrument gives to each harmonic.

Instead of the `maxharm` and `skip` variables in the program as I've written it, you could have an input called `timbre` (a French word for the characteristic sound of an instrument, pronounced sort of like "tamper" with a B instead of the P) that would be a list of weighting factors. The equivalent of `plot 5` would be this timbre list:

```
[1 0 0.3333 0 0.2]
```

This list says that the fundamental has a weight of 1, the second harmonic has a weight of 0 (so it's not used at all), the third harmonic has a weight of 1/3, and so on.

The `timbre` version of the program would be perfectly general. You could create any instrument, if you could find the right weighting factors. But so much generality makes it hard to know where to begin exploring all the possibilities. Another thing you could do would be to try different kinds of formulas for weighting factors. For example, you could write this new version of `term`:

```
to term :harmonic
op (sin :xscale * :harmonic * :x)/(:harmonic * :harmonic)
end
```

What waveforms would result from this change?

If you're really interested in computer-generated music, you'll want to hear what these waveforms sound like. Unfortunately, it's hard to do that with the standard sound generators in personal computers, which allow little or no control of timbre. But if you have one of the computer-controllable musical instruments that have become available recently, you may be able to program them to reproduce the timbre of your choice.

On the other hand, you can hear the effect of different waveforms without a computer if you visit the Exploratorium in San Francisco, the world's best museum. Among their exhibits are several that let you experiment with different ways of generating sounds. One of these exhibits is a machine that does audibly the same thing we've been doing graphically, adding up selected harmonics of a fundamental pitch. If you don't live near San Francisco, the Exploratorium is well worth the trip, no matter how far away you are!

## Program Listing

As mentioned in the text, the appropriate value of `xrange` may be different depending on which computer you're using.

```
to plot :inputs
keyword :inputs ~
        [maxharm 5 deltax 2 yscale 75 cycles 1 xrange 230 skip 2]
localmake "xscale :cycles*180/:xrange
splitscreen clearscreen hideturtle penup
setpos list (-:xrange) 0
pendown
for [x :deltax [2*:xrange] :deltax] ~
    [setpos list (xcor+:deltax) (:yscale * series :maxharm)]
end

;; Compute the Fourier series values

to series :harmonic
if :harmonic < 1 [output 0]
output (term :harmonic)+(series :harmonic-:skip)
end

to term :harmonic
output (sin :xscale * :harmonic * :x) / :harmonic
end

;; Handle keyword inputs

.macro keyword :inputs :defaults
if or (wordp :inputs) (numberp first :inputs) ~
   [make "inputs sentence (first :defaults) :inputs]
output `[local ,[filter [not numberp ?] :defaults]
         setup.values ,[:defaults]
         setup.values ,[:inputs]]
end

to setup.values :list
if emptyp :list [stop]
make first :list first butfirst :list
setup.values butfirst butfirst :list
end
```

# Appendices

# Berkeley Logo Reference Manual



## Entering and Leaving Logo

The process to start Logo depends on your operating system:

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

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

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

To leave Logo, enter the command `bye`.

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

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

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

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

## Tokenization

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

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

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

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

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

```
( ? 37 )
```

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

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

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

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

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

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

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

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

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

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

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

will print `5`, but

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

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

## Data Structure Primitives

***Constructors***

**word** *word1 word2*
**(word** *word1 word2 word3* **...)** outputs a word formed by concatenating its inputs.

**list** *thing1 thing2*
**(list** *thing1 thing2 thing3* **...)** outputs a list whose members are its inputs, which can be any Logo datum (word, list, or array).

**sentence** *thing1 thing2*
**se** *thing1 thing2*
**(sentence** *thing1 thing2 thing3* **...)**
**(se** *thing1 thing2 thing3* **...)** outputs a list whose members are its inputs, if those inputs are not lists, or the members of its inputs, if those inputs are lists.

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

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

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

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

**listtoarray** *list* **(**library procedure**)**
**(listtoarray** *list origin***)** outputs an array of the same size as the input list, whose members are the members of the input list.

**arraytolist** *array* **(**library procedure**)** outputs a list whose members are the members of the input array. The first member of the output is the first member of the array, regardless of the array's origin.

**combine** *thing1 thing2* **(**library procedure**)** If *thing2* is a word, outputs the result of `word` *thing1 thing2*. If *thing2* is a list, outputs the result of `fput` *thing1 thing2*.

**reverse** *list* **(**library procedure**)** outputs a list whose members are the members of the input list, in reverse order.

**gensym (**library procedure**)** outputs a unique word each time it's invoked. The words are of the form `G1`, `G2`, etc.

***Selectors***

**first** *thing* If the input is a word, outputs the first character of the word. If the input is a list, outputs the first member of the list. If the input is an array, outputs the origin of the array (that is, the *index of* the first member of the array).

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

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

but is provided as a primitive in order to speed up the iteration tools `map`, `map.se`, and `foreach`.

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

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

**butfirst** *wordorlist*
**bf** *wordorlist* If the input is a word, outputs a word containing all but the first character of the input. If the input is a list, outputs a list containing all but the first member of the input.

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

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

but is provided as a primitive in order to speed up the iteration tools `map`, `map.se`, and `foreach`.

**butlast** *wordorlist*
**bl** *wordorlist* If the input is a word, outputs a word containing all but the last character of the input. If the input is a list, outputs a list containing all but the last member of the input.

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

**mditem** *indexlist array* **(**library procedure**)** outputs the member of the multidimensional *array* selected by the list of numbers *indexlist*.

**pick** *list* **(**library procedure**)** outputs a randomly chosen member of the input list.

**remove** *thing list* **(**library procedure**)** outputs a copy of *list* with every member equal to *thing* removed.

**remdup** *list* **(**library procedure**)** outputs a copy of *list* with duplicate members removed. If two or more members of the input are equal, the rightmost of those members is the one that remains in the output.

**quoted** *thing* **(**library procedure**)** outputs its input, if a list; outputs its input with a quotation mark prepended, if a word.

***Mutators***

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

**mdsetitem** *indexlist array value* **(**library procedure**)** command. Replaces the member of *array* chosen by *indexlist* with the new *value*.

**.setfirst** *list value* command. Changes the first member of *list* to be *value*. *Warning:* Primitives whose names start with a period are dangerous. Their use by non-experts is not recommended. The use of **.setfirst** can lead to circular list structures, which will get some Logo primitives into infinite loops; unexpected changes to other data structures that share storage with the list being modified; and the loss of memory if a circular structure is released.

**.setbf** *list value* command. Changes the butfirst of *list* to be *value*. *Warning:* Primitives whose names start with a period are dangerous. Their use by non-experts is not recommended. The use of **.setbf** can lead to circular list structures, which will get some Logo primitives into infinite loops; unexpected changes to other data structures that share storage with the list being modified; Logo crashes and coredumps if the butfirst of a list is not itself a list; and the loss of memory if a circular structure is released.

**.setitem** *index array value* command. Changes the *index*th member of *array* to be *value*, like **setitem**, but without checking for circularity. *Warning:* Primitives whose names

start with a period are dangerous. Their use by non-experts is not recommended. The use of `.setitem` can lead to circular arrays, which will get some Logo primitives into infinite loops; and the loss of memory if a circular structure is released.

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

**pop** *stackname* **(**library procedure**)** outputs the most recently `pushed` member of the stack that is the value of the variable whose name is *stackname* and removes that member from the stack.

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

**dequeue** *queuename* **(**library procedure**)** outputs the least recently `queued` member of the queue that is the value of the variable whose name is *queuename* and removes that member from the queue.

***Predicates***

**wordp** *thing*
**word?** *thing* outputs `true` if the input is a word, `false` otherwise.

**listp** *thing*
**list?** *thing* outputs `true` if the input is a list, `false` otherwise.

**arrayp** *thing*
**array?** *thing* outputs `true` if the input is an array, `false` otherwise.

**emptyp** *thing*
**empty?** *thing* outputs `true` if the input is the empty word or the empty list, `false` otherwise.

**equalp** *thing1 thing2*
**equal?** *thing1 thing2*
*thing1* = *thing2* outputs `true` if the inputs are equal, `false` otherwise. Two numbers are equal if they have the same numeric value. Two non-numeric words are equal if they contain the same characters in the same order. If there is a variable named `caseignoredp` whose value is `true`, then an upper case letter is considered the same as the corresponding lower case letter. (This is the case by default.) Two lists are equal if their members are equal. An array is only equal to itself; two separately created arrays are never equal even if their members are equal. (It is important to be able to know if two expressions have the same array as their value because arrays are mutable; if, for example, two variables have the same array as their values then performing `setitem` on one of them will also change the other.)

**beforep** *word1 word2*
**before?** *word1 word2* outputs `true` if *word1* comes before *word2* in ASCII collating

sequence (for words of letters, in alphabetical order). Case-sensitivity is determined by the value of `caseignoredp`. Note that if the inputs are numbers, the result may not be the same as with `lessp`; for example, `beforep 3 12` is false because `3` collates after `1`.

**`.eq`** *`thing1 thing2`* outputs `true` if its two inputs are the same datum, so that applying a mutator to one will change the other as well. Outputs `false` otherwise, even if the inputs are equal in value. *Warning:* Primitives whose names start with a period are dangerous. Their use by non-experts is not recommended. The use of mutators can lead to circular data structures, infinite loops, or Logo crashes.

**`memberp`** *`thing1 thing2`*
**`member?`** *`thing1 thing2`* If *`thing2`* is a list or an array, outputs `true` if *`thing1`* is `equalp` to a member of *`thing2`*, `false` otherwise. If *`thing2`* is a word, outputs `true` if *`thing1`* is a one-character word `equalp` to a character of *`thing2`*, `false` otherwise.

**`substringp`** *`thing1 thing2`*
**`substring?`** *`thing1 thing2`* If *`thing1`* or *`thing2`* is a list or an array, outputs `false`. If *`thing2`* is a word, outputs `true` if *`thing1`* is `equalp` to a substring of *`thing2`*, `false` otherwise.

**`numberp`** *`thing`*
**`number?`** *`thing`* outputs `true` if the input is a number, `false` otherwise.

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

### *Queries*

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

**`ascii`** *`char`* outputs the integer (between 0 and 255) that represents the input character in the ASCII code. Interprets control characters as representing backslashed punctuation, and returns the character code for the corresponding punctuation character without backslash. (Compare `rawascii`.)

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

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

**member** *thing1 thing2* If *thing2* is a word or list and if `memberp` with these inputs would output `true`, outputs the portion of *thing2* from the first instance of *thing1* to the end. If `memberp` would output `false`, outputs the empty word or list according to the type of *thing2*. It is an error for *thing2* to be an array.

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

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

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

**parse** *word* outputs the list that would result if the input word were entered in response to a `readlist` operation. That is, `parse readword` has the same value as `readlist` for the same characters read.

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

## Communication

### *Transmitters*

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

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

**print** *thing*
**pr** *thing*
**(print** *thing1 thing2* **...)**
**(pr** *thing1 thing2* **...)** command. Prints the input or inputs to the current write stream (initially the terminal). All the inputs are printed on a single line, separated by spaces, ending with a newline. If an input is a list, square brackets are not printed around it, but brackets are printed around sublists. Braces are always printed around arrays.

**type** *thing*
**(type** *thing1 thing2* **...)** command. Prints the input or inputs like `print`, except that no newline character is printed at the end and multiple inputs are not separated by spaces. Note: printing to the terminal is ordinarily *line buffered;* that is, the characters you print using `type` will not actually appear on the screen until either a newline character is printed (for example, by `print` or `show`) or Logo tries to read from the keyboard (either at the request of your program or after an instruction prompt). This buffering makes the program much faster than it would be if each character appeared immediately, and in most cases the effect is not disconcerting. To accommodate programs that do a lot of positioned text display using `type`, Logo will force printing whenever `setcursor` is invoked. This solves most buffering problems. Still, on occasion you may find it necessary to force the buffered characters to be printed explicitly; this can be done using the `wait` command. `Wait 0` will force printing without actually waiting.

**show** *thing*
**(show** *thing1 thing2* **...)** command. Prints the input or inputs like `print`, except that if an input is a list it is printed inside square brackets.

***Receivers***

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

**readword**
**rw** reads a line from the read stream and outputs that line as a word. The output is a single word even if the line contains spaces, brackets, etc. If the read stream is a file, and the end of file is reached, `readword` outputs the empty list (not the empty word). `Readword` processes backslash, vertical bar, and tilde characters in the read stream. In the case of a tilde used for line continuation, the output word *does* include the tilde and the newline characters, so that the user program can tell exactly what the user entered. Vertical bars in the line are also preserved in the output. Backslash characters are not preserved in the output, but the character following the backslash is marked internally; programs can use `backslashedp` to check for this marking. (Backslashedness is preserved only for certain characters. See `backslashedp`.)

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

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

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

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

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

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

The Macintosh, of course, is not programmable.

### *File Access*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

***Terminal Access***

**keyp**
**key?** predicate, outputs `true` if there are characters waiting to be read from the read stream. If the read stream is a file, this is equivalent to `not eofp`. If the read stream is the terminal, then echoing is turned off and the terminal is set to `cbreak` (character at a time instead of line at a time) mode. It remains in this mode until some line-mode reading is requested (e.g., `readlist`). The Unix operating system forgets about any pending characters when it switches modes, so the first `keyp` invocation will always output `false`.

**cleartext**
**ct** command. Clears the text screen of the terminal.

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

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

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

## Arithmetic

***Numeric Operations***

**sum** *num1 num2*
**(sum** *num1 num2 num3* **...)**
*num1* **+** *num2* outputs the sum of its inputs.

**difference** *num1 num2*
*num1* **-** *num2* outputs the difference of its inputs. Minus sign means infix difference in ambiguous contexts (when preceded by a complete expression), unless it is preceded by a space and followed by a nonspace.

**minus** *num*
**-** *num* outputs the negative of its input. Minus sign means unary minus if it is immediately preceded by something requiring an input, or preceded by a space and followed by a nonspace. There is a difference in binding strength between the two forms:

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

**product** *num1 num2*
**(product** *num1 num2 num3* **...)**
*num1* ***** *num2* outputs the product of its inputs.

**quotient** *num1 num2*
**(quotient** *num***)**
*num1* **/** *num2* outputs the quotient of its inputs. The quotient of two integers is an integer if and only if the dividend is a multiple of the divisor. (In other words, `quotient 5 2` is 2.5, not 2, but `quotient 4 2` is 2, not 2.0—it does the right thing.) With a single input, `quotient` outputs the reciprocal of the input.

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

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

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

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

**round** *num* outputs the nearest integer to the input.

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

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

**exp** *num* outputs e (2.718281828+) to the input power.

**log10** *num* outputs the common logarithm of the input.

**ln** *num* outputs the natural logarithm of the input.

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

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

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

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

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

**radarctan** *num*
**(radarctan** *x y***)** outputs the arctangent, in radians, of its input. With two inputs, outputs the arctangent of $y/x$, if $x$ is nonzero, or $\pi/2$ or $-\pi/2$ depending on the sign of $y$, if $x$ is zero.

The expression `2*(radarctan 0 1)` can be used to get the value of $\pi$.

### *Predicates*

**lessp** *num1 num2*
**less?** *num1 num2*
*num1* `<` *num2* outputs `true` if its first input is strictly less than its second.

**greaterp** *num1 num2*
**greater?** *num1 num2*
*num1* `>` *num2* outputs `true` if its first input is strictly greater than its second.

### *Random Numbers*

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

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

### *Print Formatting*

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

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

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

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

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

***Bitwise Operations***

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

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

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

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

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

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

## Logical Operations

**and** *tf1 tf2*
**(and** *tf1 tf2 tf3* **...)** outputs `true` if all inputs are `true`, otherwise `false`. All inputs must be `true` or `false`. (Comparison is case-insensitive regardless of the value of `caseignoredp`. That is, `true` or `True` or `TRUE` are all the same.)

**or** *tf1 tf2*
**(or** *tf1 tf2 tf3* **...)** outputs `true` if any input is `true`, otherwise `false`. All inputs must be `true` or `false`. (Comparison is case-insensitive regardless of the value of `caseignoredp`. That is, `true` or `True` or `TRUE` are all the same.)

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

## Graphics

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

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

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

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

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

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

Logo begins with a black background and white pen.

### *Turtle Motion*

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

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

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

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

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

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

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

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

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

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

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

### *Turtle Motion Queries*

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

**xcor** (library procedure) outputs a number, the turtle's X coordinate.

**ycor** (library procedure) outputs a number, the turtle's Y coordinate.

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

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

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

### *Turtle and Window Control*

**showturtle**
**st** makes the turtle visible.

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

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

**clearscreen**
**cs** erases the graphics window and sends the turtle to its initial position and heading. Like `home` and `clean` together.

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

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

**fence** tells the turtle to enter fence mode: From now on, if the turtle is asked to move past the boundary of the graphics window, it will move as far as it can and then stop at the edge with an "out of bounds" error message. Compare `wrap` and `window`.

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

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

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

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

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

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

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

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

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

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

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

### *Turtle and Window Queries*

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

### *Pen and Background Control*

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

**`pendown`**
**`pd`** sets the pen's position to `down`, without changing its mode.

**`penup`**
**`pu`** sets the pen's position to `up`, without changing its mode.

**`penpaint`**
**`ppt`** sets the pen's position to `down` and mode to `paint`.

**`penerase`**
**`pe`** sets the pen's position to `down` and mode to `erase`.

**`penreverse`**
**`px`** sets the pen's position to `down` and mode to `reverse`. (This may interact in hardware-dependent ways with use of color.)

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

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

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

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

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

***Pen Queries***

**pendownp**
**pendown?** outputs `true` if the pen is down, `false` if it's up.

**penmode** outputs one of the words `paint`, `erase`, or `reverse` according to the current pen mode.

**pencolor**
**pc** outputs a color number, a nonnegative integer that is associated with a particular color by the hardware and operating system.

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

**pensize**
**penpattern** output hardware-specific pen information.

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

**background**
**bg** outputs the graphics background color.

***Procedure Definition***

**to** *procname :input1 :input2 ...* **(**special form**)** command. Prepares Logo to accept a procedure definition. The procedure will be named *procname* and there must not already be a procedure by that name. The inputs will be called *input1* etc. Any number of inputs are allowed, including none. Names of procedures and inputs are case-insensitive.

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

This version of Logo allows variable numbers of inputs to a procedure. Every procedure has a *minimum, default,* and *maximum* number of inputs. (The latter can be infinite.)

The *minimum* number of inputs is the number of required inputs, which must come first. A required input is indicated by the *:inputname* notation.

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

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

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

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

If the procedure is invoked by saying

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

then the variable `inlist` will have the value `[a b c]` and the variable `startvalue` will have the value `a`. If the procedure is invoked by saying

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

then `inlist` will have the value `[a b c]` and `startvalue` will have the value `x`.

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

```
[:inputname]
```

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

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

If this procedure is invoked by saying

```
proc "x
```

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

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

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

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

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

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

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

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

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

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

**`fulltext`** *`procname`* outputs a representation of the procedure *`procname`* in which formatting information is preserved. If the procedure was defined with `to`, `edit`, or `load`, then the output is a list of words. Each word represents one entire line of the definition in the form output by `readword`, including extra spaces and continuation lines. The last member of the output represents the `end` line. If the procedure was defined with `define`, then the output is a list of lists. If these lists are printed, one per line, the result will look like a definition using `to`. Note: the output from `fulltext` is not suitable for use as input to `define`!

**copydef** *newname oldname* command. Makes *newname* a procedure identical to *oldname*. The latter may be a primitive. If *newname* was already defined, its previous definition is lost. If *newname* was already a primitive, the redefinition is not permitted unless the variable `redefp` has the value `true`. Definitions created by `copydef` are not saved by `save`; primitives are never saved, and user-defined procedures created by `copydef` are buried. (You are likely to be confused if you `po` or `pot` a procedure defined with `copydef` because its title line will contain the old name. This is why it's buried.)

Note: dialects of Logo differ as to the order of inputs to `copydef`. This dialect uses "`make` order," not "`name` order."

### *Variable Definition*

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

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

**local** *varname*
**local** *varnamelist*
**(local** *varname1 varname2* **...)** command. Accepts as inputs one or more words, or a list of words. A variable is created for each of these words, with that word as its name. The variables are local to the currently running procedure. Logo variables follow dynamic scope rules; a variable that is local to a procedure is available to any subprocedure invoked by that procedure. The variables created by `local` have no initial value; they must be assigned a value (e.g., with `make`) before the procedure attempts to read their value.

**localmake** *varname value* **(**library procedure**)** command. Makes the named variable local, like `local`, and assigns it the given value, like `make`.

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

```
thing "
```

so that `:foo` means `thing "foo`.

### *Property Lists*

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

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

**gprop** *plistname propname* outputs the value of the *propname* property in the *plistname* property list, or the empty list if there is no such property.

**remprop** *plistname propname* command. Removes the property named *propname* from the property list named *plistname*.

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

### *Predicates*

**procedurep** *name*
**procedure?** *name* outputs `true` if the input is the name of a procedure.

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

**definedp** *name*
**defined?** *name* outputs `true` if the input is the name of a user-defined procedure, including a library procedure. (However, Logo does not know about a library procedure until that procedure has been invoked.)

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

### *Queries*

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

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

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

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

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

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

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

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

***Inspection***

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## Control Structures

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

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

**runresult** *instructionlist* runs the instructions in the input; outputs an empty list if those instructions produce no output, or a list whose only member is the output from running the input instructionlist. Useful for inventing command-or-operation control structures:

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

**repeat** *num instructionlist* command. Runs the *instructionlist* repeatedly, *num* times.

**if** *tf instructionlist*
**(if** *tf instructionlist1 instructionlist2***)** command. If the first input has the value `true`, then `if` runs the second input. If the first input has the value `false`, then `if` does nothing. (If given a third input, `if` acts like `ifelse`, as described below.) It is an error if the first input is not either `true` or `false`.

For compatibility with earlier versions of Logo, if an `if` instruction is not enclosed in parentheses, but the first thing on the instruction line after the second input expression is a literal list (i.e., a list

in square brackets), the `if` is treated as if it were `ifelse`, but a warning message is given. If this aberrant `if` appears in a procedure body, the warning is given only the first time the procedure is invoked in each Logo session.

**`ifelse`** *`tf instructionlist1 instructionlist2`* command or operation. If the first input has the value `true`, then `ifelse` runs the second input. If the first input has the value `false`, then `ifelse` runs the third input. `Ifelse` outputs a value if the instructionlist contains an expression that outputs a value.

**`test`** *`tf`* command. Remembers its input, which must be `true` or `false`, for use by later `iftrue` or `iffalse` instructions. The effect of `test` is local to the procedure in which it is used; any corresponding `iftrue` or `iffalse` must be in the same procedure or a subprocedure.

**`iftrue`** *`instructionlist`*
**`ift`** *`instructionlist`* command. Runs its input if the most recent `test` instruction had a `true` input. The `test` must have been in the same procedure or a superprocedure.

**`iffalse`** *`instructionlist`*
**`iff`** *`instructionlist`* command. Runs its input if the most recent `test` instruction had a `false` input. The `test` must have been in the same procedure or a superprocedure.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

**`bye`** command. Exits from Logo; returns to the operating system.

**`.maybeoutput`** *`value`* **`(`**special form**`)`** works like `output` except that the expression that provides the input value might not, in fact, output a value, in which case the effect is like `stop`.

This is intended for use in control structure definitions, for cases in which you don't know whether or not some expression produces a value. Example:

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

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

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

**ignore** *value* **(**library procedure**)** command. Does nothing. Used when an expression is evaluated for a side effect and its actual value is unimportant.

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

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

will print

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

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

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

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

```
3.5
5
6.5
```

**do.while** *instructionlist tfexpression* **(**library procedure**)**
command. Repeatedly evaluates the *instructionlist* as long as the evaluated *tfexpression* remains `true`. Evaluates the first input first, so the *instructionlist* is always run at least once. The *tfexpression* must be an expressionlist whose value when evaluated is `true` or `false`.

**while** *tfexpression instructionlist* **(**library procedure**)** command. Repeatedly evaluates the *instructionlist* as long as the evaluated *tfexpression* remains `true`. Evaluates the first input first, so the *instructionlist* may never be run at all. The *tfexpression* must be an expressionlist whose value when evaluated is `true` or `false`.

**do.until** *instructionlist tfexpression* **(**library procedure**)**
command. Repeatedly evaluates the *instructionlist* as long as the evaluated *tfexpression* remains `false`. Evaluates the first input first, so the *instructionlist* is always run at least once. The *tfexpression* must be an expressionlist whose value when evaluated is `true` or `false`.

**until** *tfexpression instructionlist* **(**library procedure**)** command. Repeatedly evaluates the *instructionlist* as long as the evaluated *tfexpression* remains `false`. Evaluates the first input first, so the *instructionlist* may never be run at all. The *tfexpression* must be an expressionlist whose value when evaluated is `true` or `false`.

### *Template-Based Iteration*

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

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

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

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

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

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

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

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

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

The third form of template is *named-slot* or *lambda* form. This form is indicated by a template list containing more than one member, whose first member is itself a list. The first member is taken as a list of names; local variables are created with those names and given the available data in order as their values. The number of names must equal the number of available data. This form is needed primarily when one iteration tool must be used within the template list of another, and the `?` notation would be ambiguous in the inner template. Example:

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

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

**invoke** *template input* **(**library procedure**)**
**(invoke** *template input1 input2* **...)** command or operation. Exactly like `apply` except that the inputs are provided as separate expressions rather than in a list.

**foreach** *data template* **(**library procedure**)**
**(foreach** *data1 data2 ... template***)** command. Evaluates the template list repeatedly, once for each member of the data list. If more than one data list are given, each of them must be the same length. (The data inputs can be words, in which case the template is evaluated once for each character.

In a template, the symbol `?rest` represents the portion of the data input to the right of the member currently being used as the `?` slot-filler. That is, if the data input is `[a b c d e]` and the template is being evaluated with `?` replaced by `b`, then `?rest` would be replaced by `[c d e]`. If multiple parallel slots are used, then `(?rest 1)` goes with `?1`, etc.

In a template, the symbol `#` represents the position in the data input of the member currently being used as the `?` slot-filler. That is, if the data input is `[a b c d e]` and the template is being evaluated with `?` replaced by `b`, then `#` would be replaced by `2`.

**map** *template data* **(**library procedure**)**
**(map** *template data1 data2* **...)** outputs a word or list, depending on the type of the

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

In a template, the symbol `?rest` represents the portion of the data input to the right of the member currently being used as the `?` slot-filler. That is, if the data input is `[a b c d e]` and the template is being evaluated with `?` replaced by `b`, then `?rest` would be replaced by `[c d e]`. If multiple parallel slots are used, then `(?rest 1)` goes with `?1`, etc.

In a template, the symbol `#` represents the position in the data input of the member currently being used as the `?` slot-filler. That is, if the data input is `[a b c d e]` and the template is being evaluated with `?` replaced by `b`, then `#` would be replaced by `2`.

**`map.se`** *`template data`* **(**library procedure**)**
**`(map.se`** *`template data1 data2 ...`***`)`** outputs a list formed by evaluating the template list repeatedly and concatenating the results using `sentence`. That is, the members of the output are the members of the results of the evaluations. The output list might, therefore, be of a different length from that of the data input(s). (If the result of an evaluation is the empty list, it contributes nothing to the final output.) The data inputs may be words or lists.

In a template, the symbol `?rest` represents the portion of the data input to the right of the member currently being used as the `?` slot-filler. That is, if the data input is `[a b c d e]` and the template is being evaluated with `?` replaced by `b`, then `?rest` would be replaced by `[c d e]`. If multiple parallel slots are used, then `(?rest 1)` goes with `?1`, etc.

In a template, the symbol `#` represents the position in the data input of the member currently being used as the `?` slot-filler. That is, if the data input is `[a b c d e]` and the template is being evaluated with `?` replaced by `b`, then `#` would be replaced by `2`.

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

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

In a template, the symbol `?rest` represents the portion of the data input to the right of the member currently being used as the `?` slot-filler. That is, if the data input is `[a b c d e]` and the template is being evaluated with `?` replaced by `b`, then `?rest` would be replaced by `[c d e]`. If multiple parallel slots are used, then `(?rest 1)` goes with `?1`, etc.

In a template, the symbol `#` represents the position in the data input of the member currently being used as the `?` slot-filler. That is, if the data input is `[a b c d e]` and the template is being evaluated with `?` replaced by `b`, then `#` would be replaced by `2`.

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

In a template, the symbol `?rest` represents the portion of the data input to the right of the member currently being used as the `?` slot-filler. That is, if the data input is `[a b c d e]` and the template is being evaluated with `?` replaced by `b`, then `?rest` would be replaced by `[c d e]`. If multiple parallel slots are used, then `(?rest 1)` goes with `?1`, etc.

In a template, the symbol `#` represents the position in the data input of the member currently being used as the `?` slot-filler. That is, if the data input is `[a b c d e]` and the template is being evaluated with `?` replaced by `b`, then `#` would be replaced by `2`.

**reduce** *template data* **(**library procedure**)** outputs the result of applying the template to accumulate the members of the data input. The template must be a two-slot function. Typically it is an associative function name like `"sum`. If the data input has only one constituent (member in a list or character in a word), the output is that consituent. Otherwise, the template is first applied with `?1` filled with the next-to-last consitient and `?2` with the last constituent. Then, if there are more constituents, the template is applied with `?1` filled with the next constituent to the left and `?2` with the result from the previous evaluation. This process continues until all constituents have been used. The data input may not be empty.

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

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

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

Alternatively, `reduce` can be used to write `max` as a procedure that accepts any number of inputs, as `sum` does:

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

**crossmap** *template listlist* **(**library procedure**)**
**(crossmap** *template data1 data2* **...)** outputs a list containing the results of template evaluations. Each data list contributes to a slot in the template; the number of slots is equal to the number of data list inputs. As a special case, if only one data list input is given, that list is taken as a list of data lists, and each of its members contributes values to a slot. `Crossmap` differs from `map` in that instead of taking members from the data inputs in parallel, it takes all possible combinations of members of data inputs, which need not be the same length.

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

For compatibility with the version in the first edition of *Computer Science Logo Style,* `crossmap` templates may use the notation `:1` instead of `?1` to indicate slots.

**`cascade`** *`endtest template startvalue`* **(**library procedure**)**
**(`cascade` *`endtest tmp1 sv1 tmp2 sv2 ...`*`)`**
**(`cascade` *`endtest tmp1 sv1 tmp2 sv2 ... finaltemplate`*`)`**
outputs the result of applying a template (or several templates, as explained below) repeatedly, with a given value filling the slot the first time, and the result of each application filling the slot for the following application.

In the simplest case, `cascade` has three inputs. The second input is a one-slot expression template. That template is evaluated some number of times (perhaps zero). On the first evaluation, the slot is filled with the third input; on subsequent evaluations, the slot is filled with the result of the previous evaluation. The number of evaluations is determined by the first input. This can be either a nonnegative integer, in which case the template is evaluated that many times, or a predicate expression template, in which case it is evaluated (with the same slot filler that will be used for the evaluation of the second input) repeatedly, and the `cascade` evaluation continues as long as the predicate value is `false`. (In other words, the predicate template indicates the condition for stopping.)

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

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

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

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

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

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

**cascade.2** *endtest temp1 startval1 temp2 startval2*
**(**library procedure**)** outputs the result of invoking `cascade` with the same inputs. The only difference is that the default number of inputs is five instead of three.

**transfer** *endtest template inbasket* **(**library procedure**)** outputs the result of repeated evaluation of the template. The template is evaluated once for each member of the list *inbasket*. `Transfer` maintains an *outbasket* that is initially the empty list. After each evaluation of the template, the resulting value becomes the new outbasket.

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

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

## Macros

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Here's another example:

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

It's used this way:

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

`Localmake` outputs the list

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

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

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

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

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

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

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

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

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

## Error Processing

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

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

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

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

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

### *Error Codes*

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

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

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

## Special Variables

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

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

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

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

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

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

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

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

# Index of Defined Procedures

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

## D



## E



## F



## G



## H



## I

## J



## K



## L



## M



## N



## O



## P

## Q



## R



## S



## T

## U



## W



## X



## Y



## Z

# General Index

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

## C



## D

## I



## J



## K



## L



## M

## N



## O



## P

## Q



## R



## S

## T



## U

## V



## W



## X



## Y