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

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.

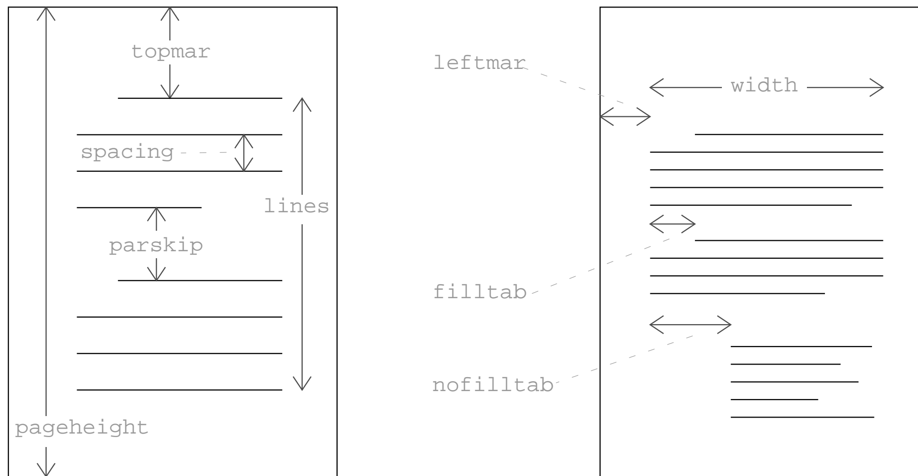
Computer
Science
Apprenticeship

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!