22 Files

We learned in Chapter 20 how to read from the keyboard and write to the screen. The same procedures (`read`, `read-line`, `display`, `show`, `show-line`, and `newline`) can also be used to read and write data files on the disk.

Ports

Imagine a complicated program that reads a little bit of data at a time from a lot of different files. For example, soon we will write a program to merge two files the way we merged two sentences in `mergesort` in Chapter 15. In order to make this work, each invocation of `read` must specify which file to read from this time. Similarly, we might want to direct output among several files.

Each of the input/output procedures can take an extra argument to specify a file:

```
(show '(across the universe) file1)
(show-line '(penny lane) file2)
(read file3)
```

What are `file1` and so on? You might think that the natural thing would be for them to be words, that is, the names of files.

It happens not to work that way. Instead, before you can use a file, you have to *open* it. If you want to read a file, the system has to check that the file exists. If you want to write a file, the system has to create a new, empty file. The Scheme procedures that open a file return a *port,* which is what Scheme uses to remember the file you opened. Ports are useful only as arguments to the input/output procedures. Here's an example:

```
> (let ((port (open-output-file "songs")))
    (show '(all my loving) port)
    (show '(ticket to ride) port)
    (show '(martha my dear) port)
    (close-output-port port))
```

(`Close-output-port`, like `define`, has an unspecified return value that we're not going to include in the examples.)

We've created a file named `songs` and put three expressions, each on its own line, in that file. Notice that nothing appeared on the screen when we called `show`. Because we used a port argument to `show`, the output went into the file. Here's what's in the file:

```
(ALL MY LOVING)
(TICKET TO RIDE)
(MARTHA MY DEAR)
```

The example illustrates two more details about using files that we haven't mentioned before: First, the name of a file must be given in double-quote marks. Second, when you're finished using a file, you have to *close* the port associated with it. (This is very important. On some systems, if you forget to close the port, the file disappears.)

The file is now permanent. If we were to exit from Scheme, we could read the file in a word processor or any other program. But let's read it using Scheme:

```
(define in (open-input-file "songs"))

> (read in)
(ALL MY LOVING)

> (read in)
(TICKET TO RIDE)

> (read in)
(MARTHA MY DEAR)

> (close-input-port in)
```

(In this illustration, we've used a global variable to hold the port because we wanted to show the results of reading the file step by step. In a real program, we'd generally use a `let` structure like the one we used to write the file. Now that we've closed the port, the variable `in` contains a port that can no longer be used.)

Writing Files for People to Read

A file full of sentences in parentheses is a natural representation for information that will be used by a Scheme program, but it may seem awkward if the file will be read by human beings. We could use `show-line` instead of `show` to create a file, still with one song title per line, but without the parentheses:

```
> (let ((port (open-output-file "songs2")))
    (show-line '(all my loving) port)
    (show-line '(ticket to ride) port)
    (show-line '(martha my dear) port)
    (close-output-port port))
```

The file `songs2` will contain

```
ALL MY LOVING
TICKET TO RIDE
MARTHA MY DEAR
```

What should we do if we want to read this file back into Scheme? We must read the file a line at a time, with `read-line`. In effect, `read-line` treats the breaks between lines as if they were parentheses:

```
(define in (open-input-file "songs2"))

> (read-line in)
(ALL MY LOVING)

> (close-input-port in)
```

(Notice that we don't have to read the entire file before closing the port. If we open the file again later, we start over again from the beginning.)

As far as Scheme is concerned, the result of writing the file with `show-line` and reading it with `read-line` was the same as that of writing it with `show` and reading it with `read`. The difference is that without parentheses the file itself is more "user-friendly" for someone who reads it outside of Scheme.*

* Another difference, not apparent in this example, is that `show` and `read` can handle structured lists. `Show-line` can print a structured list, leaving off only the outermost parentheses, but `read-line` will treat any parentheses in the file as ordinary characters; it always returns a sentence.

Using a File as a Database

It's not very interesting merely to read the file line by line. Instead, let's use it as a very small database in which we can look up songs by number. (For only three songs, it would be more realistic and more convenient to keep them in a list and look them up with `list-ref`. Pretend that this file has 3000 songs, too many for you to want to keep them all at once in your computer's memory.)

```
(define (get-song n)
  (let ((port (open-input-file "songs2")))
    (skip-songs (- n 1) port)
    (let ((answer (read-line port)))
      (close-input-port port)
      answer)))

(define (skip-songs n port)
  (if (= n 0)
      'done
      (begin (read-line port)
             (skip-songs (- n 1) port))))

> (get-song 2)
(TICKET TO RIDE)
```

When we invoke `read-line` in `skip-songs`, we pay no attention to the value it returns. Remember that the values of all but the last expression in a sequence are discarded. `Read` and `read-line` are the first procedures we've seen that have both a useful return value and a useful side effect—moving forward in the file.

`Skip-songs` returns the word `done` when it's finished. We don't do anything with that return value, and there's no particular reason why we chose that word. But every Scheme procedure has to return *something*, and this was as good as anything.

What if we asked for a song number greater than three? In other words, what if we read beyond the end of the file? In that case, `read` will return a special value called an *end-of-file object*. The only useful thing to do with that value is to test for it. Our next sample program reads an entire file and prints it to the screen:

```
(define (print-file name)
  (let ((port (open-input-file name)))
    (print-file-helper port)
    (close-input-port port)
    'done))
```

```
(define (print-file-helper port)              ;; first version
  (let ((stuff (read-line port)))
    (if (eof-object? stuff)
        'done
        (begin (show-line stuff)
               (print-file-helper port)))))

> (print-file "songs")
ALL MY LOVING
TICKET TO RIDE
MARTHA MY DEAR
DONE
```

Did you notice that each recursive call in `print-file-helper` has exactly the same argument as the one before? How does the problem get smaller? (Up to now, recursive calls have involved something like the `butfirst` of an old argument, or one less than an old number.) When we're reading a file, the sense in which the problem gets smaller at each invocation is that we're getting closer to the end of the file. You don't `butfirst` the port; reading it makes the unread portion of the file smaller as a side effect.

Transforming the Lines of a File

Often we want to transform a file one line at a time. That is, we want to copy lines from an input file to an output file, but instead of copying the lines exactly, we want each output line to be a *function* of the corresponding input line. Here are some examples: We have a file full of text and we want to *justify* the output so that every line is exactly the same length, as in a book. We have a file of students' names and grades, and we want a summary with the students' total and average scores. We have a file with people's first and last names, and we want to rearrange them to be last-name-first.

We'll write a procedure `file-map`, analogous to `map` but for files. It will take three arguments: The first will be a procedure whose domain and range are sentences; the second will be the name of the input file; the third will be the name of the output file.

Of course, this isn't exactly like the way `map` works—if it were exactly analogous, it would take only two arguments, the procedure and the *contents* of a file. But one of the important features of files is that they let us handle amounts of information that are too big to fit all at once in the computer's memory. Another feature is that once we write a file, it's there permanently, until we erase it. So instead of having a `file-map` *function* that returns the contents of the new file, we have a procedure that writes its result to the disk.

```
(define (file-map fn inname outname)
  (let ((inport (open-input-file inname))
        (outport (open-output-file outname)))
    (file-map-helper fn inport outport)
    (close-input-port inport)
    (close-output-port outport)
    'done))

(define (file-map-helper fn inport outport)
  (let ((line (read-line inport)))
    (if (eof-object? line)
        'done
        (begin (show-line (fn line) outport)
               (file-map-helper fn inport outport)))))
```

Compare this program with the earlier `print-file` example. The two are almost identical. One difference is that now the output goes to a file instead of to the screen; the other is that we apply the function `fn` to each line before doing the output. But that small change vastly increases the generality of our program. We've performed our usual trick of generalizing a pattern by adding a procedure argument, and instead of a program that carries out one specific task (printing the contents of a file), we have a tool that can be used to create many programs.

We'll start with an easy example: putting the last name first in a file full of names. That is, if we start with an input file named `dddbmt` that contains

```
David Harmon
Trevor Davies
John Dymond
Michael Wilson
Ian Amey
```

we want the output file to contain

```
Harmon, David
Davies, Trevor
Dymond, John
Wilson, Michael
Amey, Ian
```

Since we are using `file-map` to handle our progress through the file, all we have to write is a procedure that takes a sentence (one name) as its argument and returns the same name but with the last word moved to the front and with a comma added:

```
(define (lastfirst name)
  (se (word (last name) ",") (bl name)))
```

We use `butlast` rather than `first` in case someone in the file has a middle name.

To use this procedure we call `file-map` like this:

```
> (file-map lastfirst "dddbmt" "dddbmt-reversed")
DONE
```

Although you don't see the results on the screen, you can

```
> (print-file "dddbmt-reversed")
```

to see that we got the results we wanted.

Our next example is averaging grades. Suppose the file `grades` contains this text:

```
John 88 92 100 75 95
Paul 90 91 85 80 91
George 85 87 90 72 96
Ringo 95 84 88 87 87
```

The output we want is:

```
John total: 450 average: 90
Paul total: 437 average: 87.4
George total: 430 average: 86
Ringo total: 441 average: 88.2
```

Here's the program:

```
(define (process-grades line)
  (se (first line)
      "total:"
      (accumulate + (bf line))
      "average:"
      (/ (accumulate + (bf line))
         (count (bf line)))))

> (file-map process-grades "grades" "results")
```

As before, you can

```
> (print-file "results")
```

to see that we got the results we wanted.

Justifying Text

Many word-processing programs *justify* text; that is, they insert extra space between words so that every line reaches exactly to the right margin. We can do that using `file-map`.

Let's suppose we have a file `r5rs`, written in some text editor, that looks like this:

```
Programming languages should be designed not by
piling feature on top of feature, but by
removing the weaknesses and restrictions that
make additional features appear necessary.
Scheme demonstrates that a very small number of
rules for forming expressions, with no
restrictions on how they are composed, suffice
to form a practical and efficient programming
language that is flexible enough to support most
of the major programming paradigms in use today.
```

(This is the first paragraph of the *Revised*^{5} *Report on the Algorithmic Language Scheme,* edited by William Clinger and Jonathan Rees.)

Here is what the result should be if we justify our `r5rs` text:

```
Programming languages should be  designed  not  by
piling  feature  on  top  of   feature,   but   by
removing  the  weaknesses  and  restrictions  that
make   additional   features   appear   necessary.
Scheme demonstrates that a very  small  number  of
rules   for   forming   expressions,    with    no
restrictions on how  they  are  composed,  suffice
to form  a  practical  and  efficient  programming
language that is flexible enough to  support  most
of the major programming paradigms in  use  today.
```

The tricky part is that ordinarily we don't control the spaces that appear when a sentence is printed. We just make sure the words are right, and we get one space between words automatically. The solution used in this program is that each line of the output file is constructed as a single long word, including space characters that we place explicitly within it. (Since `show-line` requires a sentence as its argument, our procedure will actually return a one-word sentence. In the following program, `pad` constructs the word, and `justify` makes a one-word sentence containing it.)

This program, although short, is much harder to understand than most of our short examples. There is no big new idea involved; instead, there are a number of unexciting

but necessary details. How many spaces between words? Do some words get more space than others? The program structure is messy because the problem itself is messy. Although it will be hard to read and understand, this program is a more realistic example of input/output programming than the cleanly structured examples we've shown until now.

`Justify` takes two arguments, the line of text (a sentence) and a number indicating the desired width (how many characters). Here's the algorithm: First the program computes the total number of characters the sentence would take up without adding extras. That's the job of `char-count`, which adds up the lengths of all the words, and adds to that the $n-1$ spaces between words. `Extra-spaces` subtracts that length from the desired line width to get the number of extra spaces we need.

The hard part of the job is done by `pad`. It's invoked with three arguments: the part of the line not yet processed, the number of opportunities there are to insert extra spaces in that part of the line (that is, the number of words minus one), and the number of extra spaces that the program still needs to insert. The number of extra spaces to insert *this time* is the integer quotient of the number `pad` wants to insert and the number of chances it'll have. That is, if there are five words on the line, there are four places where `pad` can insert extra space. If it needs to insert nine spaces altogether, then it should insert 9/4 or two spaces at the first opportunity. (Are you worried about the remainder? It will turn out that `pad` doesn't lose any spaces because it takes the quotient over again for each word break. The base case is that the number of remaining word breaks (the divisor) is one, so there will be no remainder, and all the leftover extra spaces will be inserted at the last word break.)

```
(define (justify line width)
  (if (< (count line) 2)
      line
      (se (pad line
               (- (count line) 1)
               (extra-spaces width (char-count line))))))

(define (char-count line)
  (+ (accumulate + (every count line))      ; letters within words
     (- (count line) 1)))                   ; plus spaces between words

(define (extra-spaces width chars)
  (if (> chars width)
      0                                     ; none if already too wide
      (- width chars)))
```

```
(define (pad line chances needed)
  (if (= chances 0)                          ; only one word in line
      (first line)
      (let ((extra (quotient needed chances)))
        (word (first line)
              (spaces (+ extra 1))
              (pad (bf line) (- chances 1) (- needed extra))))))

(define (spaces n)
  (if (= n 0)
      ""
      (word " " (spaces (- n 1)))))
```

Because `justify` takes two arguments, we have to decide what line width we want to give it. Here's how to make each line take 50 characters:

```
> (file-map (lambda (sent) (justify sent 50)) "r5rs" "r5rs-just")
```

Preserving Spacing of Text from Files

If we try to print the file `r5rs-just` from the previous section using `print-file`, it'll look exactly like `r5rs`. That's because `read-line` doesn't preserve consecutive spaces in the lines that it reads. `Read-line` cares only where each word (consisting of non-space characters) begins and ends; it pays no attention to how many spaces come between any two words. The lines

```
All     My          Loving
```

and

```
All My Loving
```

are the same, as far as `read-line` tells you.

For situations in which we do care about spacing, we have another way to read a line from a file. The procedure `read-string` reads all of the characters on a line, returning a single word that contains all of them, spaces included:*

* Like all the input and output primitives, `read-string` can be invoked with or without a port argument.

```
> (define inport (open-input-file "r5rs-just"))

> (read-string inport)
"Programming languages should be  designed  not  by"

> (read-string inport)
"piling  feature  on  top  of   feature,   but   by"

> (close-input-port inport)
```

We can use `read-string` to rewrite `print-file` so that it makes an exact copy of the input file:

```
(define (print-file-helper port)
  (let ((stuff (read-string port)))
    (if (eof-object? stuff)
        'done
        (begin (show stuff)
               (print-file-helper port)))))
```

(We only had to change the helper procedure.)

Merging Two Files

Suppose you have two files of people's names. Each file has been sorted in alphabetical order. You want to combine them to form a single file, still in order. (If this sounds unrealistic, it isn't. Programs that sort very large amounts of information can't always fit it all in memory at once, so they read in as much as fits, sort it, and write a file. Then they read and sort another chunk. At the end of this process, the program is left with several sorted partial files, and it has to merge those files to get the overall result.)

The algorithm for merging files is exactly the same as the one we used for merging sentences in the `mergesort` program of Chapter 15. The only difference is that the items to be sorted come from reading ports instead of from `first`ing a sentence.

```
(define (filemerge file1 file2 outfile)
  (let ((p1 (open-input-file file1))
        (p2 (open-input-file file2))
        (outp (open-output-file outfile)))
    (filemerge-helper p1 p2 outp (read-string p1) (read-string p2))
    (close-output-port outp)
    (close-input-port p1)
    (close-input-port p2)
    'done))
```

```
(define (filemerge-helper p1 p2 outp line1 line2)
  (cond ((eof-object? line1) (merge-copy line2 p2 outp))
        ((eof-object? line2) (merge-copy line1 p1 outp))
        ((before? line1 line2)
         (show line1 outp)
         (filemerge-helper p1 p2 outp (read-string p1) line2))
        (else (show line2 outp)
              (filemerge-helper p1 p2 outp line1 (read-string p2)))))

(define (merge-copy line inp outp)
  (if (eof-object? line)
      #f
      (begin (show line outp)
             (merge-copy (read-string inp) inp outp))))
```

You might think, comparing `filemerge-helper` with such earlier examples as `print-file-helper` and `file-map-helper`, that it would make more sense for `filemerge-helper` to take just the three ports as arguments and work like this:

```
(define (filemerge-helper p1 p2 outp)        ;; wrong
  (let ((line1 (read-string p1))
        (line2 (read-string p2)))
    (cond ((eof-object? line1) (merge-copy p2 outp))
          ((eof-object? line2) (merge-copy p1 outp))
          ((before? line1 line2)
           (show line1 outp)
           (filemerge-helper p1 p2 outp))
          (else (show line2 outp)
                (filemerge-helper p1 p2 outp)))))
```

Unfortunately, this won't work. Suppose that the first line of `file2` comes before the first line of `file1`. This program correctly writes the first line of `file2` to the output file, as we expect. But what about the first line of `file1`? Since we called `read-string` on `file1`, we've "gobbled"* that line, but we're not yet ready to write it to the output.

In each invocation of `filemerge-helper`, only one line is written to the output file, so unless we want to lose information, we'd better read only one line. This means that we can't call `read-string` twice on each recursive call. One of the lines has to be handed down from one invocation to the next. (That is, it has to be an argument to the

* Computer programmers really talk this way.

recursive call.) Since we don't know in advance *which* line to keep, the easiest solution is to hand down both lines.

Therefore, `filemerge-helper` also takes as arguments the first line of each file that hasn't yet been written to the output. When we first call `filemerge-helper` from `filemerge`, we read the first line of each file to provide the initial values of these arguments. Then, on each recursive call, `filemerge-helper` calls `read-string` only once.

Writing Files for Scheme to Read

You may be thinking that the three file-reading procedures we've shown, `read`, `read-line`, and `read-string`, have been getting better and better. `Read` ignores case and forces you to have parentheses in your file. `Read-line` fixes those problems, but it loses spacing information. `Read-string` can read anything and always gets it right.

But there's a cost to the generality of `read-string`; it can read any file, but it loses *structure* information. For example, when we processed a file of people's names with `file-map`, we used this function:

```
(define (lastfirst name)
  (se (word (last name) ",") (bl name)))
```

It's easy to break a name into its components if you have the name in the form of a sentence, with the words separated already. But if we had read each line with `read-string`, `last` of a line would have been the last letter, not the last name.

The `lastfirst` example illustrates why you might want to use `read-line` rather than `read-string`: `Read-line` "understands" spaces. Here's an example in which the even more structured `read` is appropriate. We have a file of Beatles songs and the albums on which they appear:

```
((love me do) (please please me))
((do you want to know a secret?) (please please me))
((think for yourself) (rubber soul))
((your mother should know) (magical mystery tour))
```

Each line of this file contains two pieces of information: a song title and an album title. If each line contained only the words of the two titles, as in

```
love me do please please me
```

how would we know where the song title stops and the album title starts? The natural way to represent this grouping information is to use the mechanism Scheme provides for grouping, namely, list structure.

If we use `read-line` to read the file, we'll lose the list structure; it will return a sentence containing words like `"((love"`. `Read`, however, will do what we want.

How did we create this file in the first place? We just used one `show` per line of the file, like this:

```
> (show '((love me do) (please please me)) port)
```

But what about the movie soundtracks? We're going to have to come to terms with the apostrophe in "A Hard Day's Night."

The straightforward solution is to put `day's` in a string:

```
(show '((and i love her) (a hard "day's" night)) port)
```

The corresponding line in the file will look like this:

```
((AND I LOVE HER) (A HARD day's NIGHT))
```

This result is actually even worse than it looks, because when we try to `read` the line back, the `'s` will be expanded into `(quote s)` in most versions of Scheme. Using a string made it possible for us to get an apostrophe into Scheme. If the word `day's` were inside quotation marks in the file, then `read` would understand our intentions.

Why aren't there double quotes in the file? All of the printing procedures we've seen so far assume that whatever you're printing is intended to be read by people. Therefore, they try to minimize distracting notation such as double-quote marks. But, as we've discovered, if you're writing a file to be read by Scheme, then you do want enough notation so that Scheme can tell what the original object was.

`Write` is a printing procedure just like `display`, except that it includes quote marks around strings:*

* There are other kinds of data that `write` prints differently from `display`, but we don't use them in this book. The general rule is that `display` formats the output for human readers, while `write` ensures that Scheme can reread the information unambiguously. `Show` and `show-line` are extensions that we wrote using `display`. We could have written `show-in-write-format`, for example, but happened not to need it.

```
> (write '(a hard "day's" night))
(A HARD "day's" NIGHT)
```

Once we're using strings, and since we're not extracting individual words from the titles, we might as well represent each title as one string:

```
> (write '("And I Love Her" "A Hard Day's Night") port)
```

Pitfalls

⇒ One pitfall crucial to avoid when using files is that if there is an error in your program, it might blow up and return you to the Scheme prompt without closing the open files. If you fix the program and try to run it again, you may get a message like "file busy" because the operating system of your computer may not allow you to open the same file on two ports at once. Even worse, if you exit from Scheme without closing all your ports, on some computers you may find that you have unreadable files thereafter.

To help cope with this problem, we've provided a procedure `close-all-ports` that can be invoked to close every port that you've opened since starting Scheme. This procedure works only in our modified Scheme, but it can help you out of trouble while you're learning.

⇒ Be sure you don't open or close a file within a recursive procedure, if you intend to do it only once. That's why most of the programs in this chapter have the structure of a procedure that opens files, calls a recursive helper, and then closes the files.

⇒ As we explained in the `filemerge` example, you can't read the same line twice. Be sure your program remembers each line in a variable as long as it's needed.

Exercises

22.1 Write a `concatenate` procedure that takes two arguments: a list of names of input files, and one name for an output file. The procedure should copy all of the input files, in order, into the output file.

22.2 Write a procedure to count the number of lines in a file. It should take the filename as argument and return the number.

22.3 Write a procedure to count the number of words in a file. It should take the filename as argument and return the number.

22.4 Write a procedure to count the number of characters in a file, including space characters. It should take the filename as argument and return the number.

22.5 Write a procedure that copies an input file to an output file but eliminates multiple consecutive copies of the same line. That is, if the input file contains the lines

```
John Lennon
Paul McCartney
Paul McCartney
George Harrison


Paul McCartney
Ringo Starr
```

then the output file should contain

```
John Lennon
Paul McCartney
George Harrison

Paul McCartney
Ringo Starr
```

22.6 Write a `lookup` procedure that takes as arguments a filename and a word. The procedure should print (on the screen, not into another file) only those lines from the input file that include the chosen word.

22.7 Write a `page` procedure that takes a filename as argument and prints the file a screenful at a time. Assume that a screen can fit 24 lines; your procedure should print 23 lines of the file and then a prompt message, and then wait for the user to enter a (probably empty) line. It should then print the most recent line from the file again (so that the user will see some overlap between screenfuls) and 22 more lines, and so on until the file ends.

22.8 A common operation in a database program is to *join* two databases, that is, to create a new database combining the information from the two given ones. There has to be some piece of information in common between the two databases. For example,

suppose we have a class roster database in which each record includes a student's name, student ID number, and computer account name, like this:

```
((john alec entwistle) 04397 john)
((keith moon) 09382 kmoon)
((peter townshend) 10428 pete)
((roger daltrey) 01025 roger)
```

We also have a grade database in which each student's grades are stored according to computer account name:

```
(john 87 90 76 68 95)
(kmoon 80 88 95 77 89)
(pete 100 92 80 65 72)
(roger 85 96 83 62 74)
```

We want to create a combined database like this:

```
((john alec entwistle) 04397 john 87 90 76 68 95)
((keith moon) 09382 kmoon 80 88 95 77 89)
((peter townshend) 10428 pete 100 92 80 65 72)
((roger daltrey) 01025 roger 85 96 83 62 74)
```

in which the information from the roster and grade databases has been combined for each account name.

Write a program `join` that takes five arguments: two input filenames, two numbers indicating the position of the item within each record that should overlap between the files, and an output filename. For our example, we'd say

```
> (join "class-roster" "grades" 3 1 "combined-file")
```

In our example, both files are in alphabetical order of computer account name, the account name is a word, and the same account name never appears more than once in each file. In general, you may assume that these conditions hold for the item that the two files have in common. Your program should *not* assume that every item in one file also appears in the other. A line should be written in the output file only for the items that do appear in both files.