

Files

Summary: Files permit you to save values between invocations of programs and to provide information to programs without typing the information interactively. In this reading, we explore key ideas in the use of files within Scheme.

Contents:

- Input Ports
- Input Ports
- Reading One Character at a Time
- Finding the End of a File
- The `read` Procedure
- Another Form of File Recursion
- Output
 - `write`
 - `display`
 - `newline`
- Creating New Files
- Writing One Character At A Time
- Miscellaneous Facilities
- Short Reference

When a Scheme program is designed to work with large volumes of data, it is often more convenient for the user to prepare its input in one or more separate files, using an appropriate tool (such as a text editor or a statistical package), than to type the data in as the program is running. The Scheme program itself finds the files containing the data and reads them, without user intervention.

Similarly, when a Scheme program generates a lot of output, it is often more convenient to have it store the output in one or more files, instead of displaying it in the window that the interactive interface is using. Other programs can recover the results from such files if further processing is needed.

Note that files therefore let us store values between invocation of Scheme programs (and other programs). This permanence is another benefit of using files.

Here we consider the techniques used in Scheme to read data from files and to write data to files.

Input Ports

Scheme provides two basic operations for reading from files, `read` and `read-char`. The `read` procedure reads the next Scheme value from the given file and returns it as a Scheme value. The `read-char` procedure reads the next character from the file and returns that character.

For example, if the file began with 23512 11 13, the first two values returned by the `read` procedure would be the integers 23512 and 11, while the first two values returned by the `read-char` procedure would be `#\2` and `#\3`.

These procedures take as their argument an *input port* through which the data will be read in. In theory, any kind of a device that supplies data on demand can be on the other side of the input port, and some implementations of Scheme provide several ways of creating them. However, we'll consider only the *default input port*, through which data typed at the keyboard are transmitted to a Scheme program interactively, and *file input ports*, through which Scheme programs read data stored in files.

When DrScheme starts up, it automatically creates the default input port and connects the keyboard to it. This is the input port on which the `read` procedure normally operates. When the user exits from Scheme, this port is closed as part of the cleanup process.

To read data from a file, however, the programmer must explicitly open an input port and connect that file to it. There is a built-in Scheme procedure to do this: `open-input-file` takes one argument, a string, and returns an input port to which the file named by the string is connected. For instance, the call

```
(open-input-file "/home/rebelsky/Web/Courses/CS151/2006F/Examples/sample.dat")
```

returns an input port to which the named file is connected.

Constructing the input port does you no good unless you give it a name, so `open-input-file` is almost always either named explicitly (e.g., with `define`) or used as the parameter to procedure call that expects a port.

```
(define source
  (open-input-file "..."))
```

or

```
(define helper
  (lambda (source)
    ...))
(helper (open-input-file "..."))
```

When you're done with a port, you should make sure to close it again with `close-input-port`. To finish the examples above,

```
; Prepare to read from a file.
(define source
  (open-input-file "..."))
; Read some parts of the file.
...
; We're done, so clean up.
(close-input-port source)
```

or

```
(define helper
  (lambda (source)
    ...
    (close-input-port source)))
```

Reading One Character at a Time

As the example above suggests, an input port is often used as an argument to `read-char`, which reads in (and returns) one character from the file on the other side of the input port. It can also be used as parameter to `peek-char`, which looks through the input port to see what the next character in the file is, and returns that character, but does not actually read it in from the file. The difference is that you can peek at the next character as often as you like, and it remains accessible through the input port, but once you read in a character there is no way to “un-read” it -- the port advances inexorably to the next character in the file.

The file `/home/rebelsky/Courses/CS151/2006F/Examples/hi.dat` text file that contains one line, consisting of the cheerful greeting `Hi there!`. Let us see what happens when we read from this file using `read-char`.

```
> (define source
  (open-input-file "/home/rebelsky/Web/Courses/CS151/2001S/Examples/hi.dat"))
> (read-char source)
#\H
> (peek-char source)
#\i
> (peek-char source)
#\i
> (read-char source)
#\i
> (read-char source)
#\space
> (close-input-port source)
```

Notice that the `peek-char` procedure peeks through the port to see what the next available character of the file is, and returns the character it sees. The `read-char` procedure pulls that character in through the port and returns it, leaving the port open with the following character accessible through it.

Finding the End of a File

Scheme automatically provides a sentinel for every file input port it opens. The sentinel is a special value known as the *end-of-file object*. It is returned by any of the three input procedures when there is nothing left to be read from the file. DrScheme prints the end-of-file object as `#<eof>`. To continue the preceding example,

```
> (define source
  (open-input-file "/home/rebelsky/Web/Courses/CS151/2006F/Examples/hi.dat"))
> (read-char source)
#\H
> (read-char source)
#\i
> (read-char source)
```

```

#\space
> (read-char source)
#\T
> (read-char source)
#\h
> (read-char source)
#\e
> (read-char source)
#\r
> (read-char source)
#\e
> (read-char source)
#\!
> (read-char source)
#\newline
> (peek-char source)
#<eof>
> (read-char source)
#<eof>
> (read-char source)
#<eof>
> (close-input-port source)

```

The end-of-file object is not a character, and there is no standard Scheme name for the end-of-file object, but there is a primitive predicate `eof-object?` that detects it:

```

> (eof-object? (read-char source))
#t

```

As an example of the use of `read-char`, here's the definition of a procedure called `read-line`, which reads in characters through a given input port until it reaches the end of the file or encounters a `#\newline` character, then returns a string containing all of the characters that it has read in:

```

;;; Procedure:
;;;   read-line
;;; Parameters:
;;;   source, an input port
;;; Purpose:
;;;   Read one line of input from a source and return that line
;;;   as a string.
;;; Produces:
;;;   line, a string
;;; Preconditions:
;;;   The source is open for reading. [Unverified]
;;; Postconditions:
;;;   Has read characters from the source (thereby affecting
;;;   future calls to read-char and peek-char).
;;;   line represents the characters in the file from the
;;;   "current" point at the time read-line was called
;;;   until the first end-of-line or end-of-file character.
;;;   line does not contain a newline.
(define read-line
  (lambda (source)
    ; Read all the characters remaining on the line and
    ; then convert them to a string.
    (list->string (read-line-of-chars source))))

```

```

;;; Procedure:
;;;   read-line-of-chars
;;; Parameters:
;;;   source, an input port
;;; Purpose:
;;;   Read one line of input from a source and return that line
;;;   as a list of characters.
;;; Produces:
;;;   chars, a list of characters.
;;; Preconditions:
;;;   The source is open for reading. [Unverified]
;;; Postconditions:
;;;   Has read characters from the source (thereby affecting
;;;   future calls to read-char and peek-char).
;;;   chars represents the characters in the file from the
;;;   "current" point at the time read-line was called
;;;   until the first end-of-line or end-of-file character.
;;;   chars does not contain a newline.
(define read-line-of-chars
  (lambda (source)
    ; If we're at the end of the line or the end of the file,
    ; then there are no more characters, so return the empty list.
    (cond
      ; If we're at the end of the file, there are no more characters,
      ; so return the empty list.
      ((eof-object? (peek-char source)) null)
      ; If we're at the end of the line, we're done with the line
      ; skip over the end-of-line character and return the empty list.
      ((char=? (peek-char source) #\newline) (read-char source) null)
      ; Otherwise, read the current character, read the remaining
      ; characters, and join them together.
      (else (cons (read-char source) (read-line-of-chars source))))))

```

There are many things we can now do with these procedures. For example, here's a simple procedure that takes a file name as an argument and prints the first line of a file.

```

;;; Procedure:
;;;   first-line
;;; Parameters:
;;;   file-name, a string that names a file.
;;; Purpose:
;;;   Reads and displays the first line of the file.
;;; Produces:
;;;   Absolutely nothing.
;;; Preconditions:
;;;   There is a file by the given name.
;;;   It is possible to write to the standard output port.
;;; Postconditions:
;;;   Does not affect the file.
;;;   The first line of the named file has been written to
;;;   the standard output.
(define first-line
  (lambda (file-name)
    (first-line-helper (open-input-file file-name))))

(define first-line-helper

```

```
(lambda (source)
  (display "The first line of ")
  (display file-name)
  (newline)
  (display (read-line source))
  (newline)
  (close-input-port source)))
```

The read Procedure

It is also possible to read from a file using the one-argument form of the `read` procedure, which pulls a complete Scheme datum (instead of just one character) through a given input port. It also leaves the port open, with the next character or Scheme datum accessible through it.

Consider, again, the file described above with the form

```
23512 11 13
```

If we were to work with this file using `read-char`, we would see a sequence of values like the following

```
> (define source
    (open-input-file "/home/rebelsky/Web/Courses/CS151/2006F/Examples/sample.dat"))
> (read-char source)
#\2
> (read-char source)
#\3
> (read-char source)
#\5
> (read-char source)
#\1
> (read-char source)
#\2
> (read-char source)
#\space
> (read-char source)
#\1
> (read-char source)
#\1
> (read-char source)
#\space
> (read-char source)
#\1
> (read-char source)
#\3
> (read-char source)
#\newline
> (read-char source)
#<eof>
> (close-input-stream source)
```

If, however, we were to use `read`, we would see the following sequence.

```

> (define source
  (open-input-file "/home/rebelsky/Web/Courses/CS151/2006F/Examples/sample.dat"))
> (read source)
23512
> (read source)
11
> (read source)
13
> (read source)
#<eof>
> (close-input-stream source)

```

Whether you use `read` or `read-char` depends on your particular application.

Another Form of File Recursion

Here's another example of how to use Scheme's facilities for input from a file. The `sum-of-file` procedure takes one argument, a string that names a file full of numbers; the procedure opens that file, reads in the numbers it contains one by one, adds each one in turn to a running total, closes the file, and returns the total.

```

;;; Procedure:
;;;   sum-of-file
;;; Parameters:
;;;   file-name, a string that names a file.
;;; Purpose:
;;;   Sums the values in the given file.
;;; Produces:
;;;   sum, a number.
;;; Preconditions:
;;;   file-name names a file. [Unverified]
;;;   That file contains only numbers. [Unverified]
;;; Postconditions:
;;;   Returns a number.
;;;   That number is the sum of all the numbers in the file.
;;;   Does not affect the file.
(define sum-of-file
  (lambda (file-name)
    (sum-of-file-helper (open-input-file file-name))))

;;; Helper:
;;;   sum-of-file-helper
;;; Notes:
;;;   A lot like sum-of-file, except that it reads the values from
;;;   an open input port rather than a file name. (The file name
;;;   is also passed in so that it can be used for error messages.)
;;;   Does not verify that the input port is open.
;;;   Skips over non-numbers in the input file.
(define sum-of-file-helper
  (lambda (source)
    (sum-of-file-helper-helper source (read source))))

(define sum-of-file-helper-helper
  (lambda (source nextval)
    (cond

```

```

; Are we at the end of the file? Then stop and return 0 for
; "no numbers read". Here, we're taking advantage of 0 being
; the arithmetic identity.
((eof-object? nextval) (close-input-port source) 0)
; Have we just read a number? If so, add it to the sum of the
; remaining numbers.
((number? nextval) (+ nextval
                      (sum-of-file-helper source)))
; Hmm ... not a number. Skip it.
(else (sum-of-file-helper source))))

```

In the base case of the recursion, there are no numbers left in the file, and the call to the `read` procedure immediately returns the end-of-file object. The helper closes the file and returns 0.

If the value of `(read source)` is a number, it is added to the value of a recursive call to the helper, which is the sum of all the subsequent numbers in the file.

If the helper discovers a non-number in the file whose contents it is adding up, then we skip it. (Ideally, we would report an error, but this seems safest given our knowledge to this point.)

Output

The Scheme model we have worked with so far is both simple and straightforward: The user types a Scheme expression, the computer thinks for awhile, and then prints the value of the expression. However, some programs may benefit from additional output printed while the program is computing. For example, one helpful technique for understanding recursive procedures is to print out the current call at each step. (You've done so in some experiments.) More importantly, output procedures (along with corresponding input procedures) permit programmers to write save results for later analysis or to transmit those results to other programs.

Scheme provides four basic output operations: `write`, `display`, `newline`, and `write-char`.

`write`

The `write` procedure takes one argument and prints out a representation of that argument. The nature of the value that it *returns* is unspecified (under DrScheme, for instance, it's the special "void" value) -- the printing is a side effect of the evaluation of the call to `write`, not its result.

If you use `write` interactively, DrScheme also encloses the material that `write` prints out inside an interaction box. You can distinguish user input from program output in an interaction box by its color: User input is displayed in green, program output in purple. Both are distinguished from DrScheme's usual way of exhibiting the value of an expression, which is to print it in dark blue without drawing an interaction box. (Note that we may use different colors in the readings and labs.)

display

The `display` procedure also takes one argument and prints out a representation of it, but it differs from the `write` procedure in that it does not enclose the representations of strings in double quotation marks and does not print the mesh-backslash combination when displaying a character:

```
> (display "sample string")
sample string
> (write "sample string")
"sample string"
> (display #\A)
A
> (write #\A)
#\A
```

newline

The `newline` procedure takes no arguments and returns an unspecified value; as a side effect, it terminates the current output line. Successive calls to `write` and `display` normally produce output that is all strung together on one line. Calls to `newline` are used to break up such output into separate lines.

```
> (begin
  (display "all-")
  (display "on-")
  (display "one-")
  (display "line")
  (newline)
  (display "This is on a ")
  (display "separate line.")
  (newline))
all-on-one-line
This is on a separate line.
```

The call `(newline)` has exactly the same effect as `(display #\newline)`, for which you can consider it a convenient shorthand.

Creating New Files

To provide for the possibility of having Scheme create files and write data to those files, each of Scheme's output procedures can be provided with a parameter that specifies the output port through which the data will be written. As before, we'll consider only the default output port -- the interaction box, under DrScheme -- and file output ports, through which Scheme programs write data to files.

If you followed the discussion of input ports, you should encounter few surprises about output ports. The default output port is created when the Scheme interactive interface starts up and closed when it shuts down; in between, Scheme uses this port for most calls to `write`, `display`, and `newline`. To write data to a file instead, the programmer must explicitly invoke `open-output-file`, which returns a file output port; once this output port is given a name, it can be used as an extra argument to any of the output procedures, with the effect that the values will be written to the file rather than to the interaction window. When no more output is to be written to the file, the programmer must explicitly close the port by

invoking `close-output-port`.

```

;;; Procedure:
;;; store-divisors
;;; Parameters:
;;;   file-name, a string that names a file
;;;   dividend, a natural number
;;; Purpose:
;;;   Compute all the divisors of dividend and store them
;;;   to the named file.
;;; Produces:
;;;   Nothing. That is, it returns no values. It does
;;;   create a file.
;;; Preconditions:
;;;   It must be possible to open the desired output file.
;;;   dividend must be a non-negative, exact, integer.
;;; Postconditions:
;;;   The file with name file-name now contains many integers.
;;;   All the values in that file evenly divide dividend.
(define store-divisors
  (lambda (file-name dividend)
    (store-divisors-helper (open-output-file file-name) 1 dividend)))

;;; Helper:
;;; store-divisors-helper
;;; Parameters:
;;;   target, an output port
;;;   trial-divisor, the smallest divisor we should try
;;;   dividend, the number we're working with
;;; Purpose:
;;;   Stores all divisors of dividend that are at least as
;;;   large as trial-divisor to target.
;;; Produces:
;;;   Nothing.
;;; Preconditions:
;;;   It is possible to write to the target port.
;;;   Both trial-divisor and dividend are natural numbers.
;;; Postconditions:
;;;   All divisors of dividend that are at least as large as
;;;   trial-divisor have been added to target.
;;;   target is still open for writing
(define store-divisors-helper
  (lambda (target trial-divisor dividend)
    ; We only continue to work when the trial-divisor is not
    ; larger than the dividend. Note that I'm using cond because
    ; cond permits multiple operations when the test succeeds.
    (cond ((<= trial-divisor dividend)
           ; Okay, does the current trial-divisor evenly divide
           ; dividend?
           (if (zero? (remainder dividend trial-divisor))
               ; It does! Write it to the file
               (write-number target trial-divisor))))

```

```

; Continue with any other potential divisors
(store-divisors-kernel target (+ 1 trial-divisor) dividend))
; If the trial divisor is bigger than the dividend, then we're
; done, so close the port and stop.
(else (close-output-port target))))))

(define write-number
  (lambda (target value)
    (write value target)
    (newline target)))

```

Not-so-surprisingly, Scheme doesn't let you call `open-output-file` using a file that already exists. To enable the programmer to test the precondition for `open-output-file`, DrScheme supplies a `file-exists?` predicate, which takes a string as argument and returns `#t` if it is the name of an existing file and `#f` if it is not. It also supplies a `delete-file` procedure that takes a string as argument and tries to annihilate the file that it names (if there is such a file). Neither of these procedures is standard, however, so other Scheme implementations do not always provide them.

Writing One Character At A Time

Besides `write`, `display`, and `newline`, Scheme provides a primitive procedure `write-char` that is used to create an output file one character at a time. It takes two arguments, the character to be written and the output port through which it is to be sent.

Miscellaneous Facilities

Scheme provides the type predicate `input-port?`, which can be applied to any object to determine whether it is an input port. It also provides the analogous `output-port?` predicate.

The `current-input-port` procedure, which takes no arguments, returns the default input port, in case you want to give it a name, pass it as an argument to a procedure that expects a port, and so on. Similarly, the `current-output-port` procedure takes no arguments and returns the default output port. It is a bad idea to attempt to close the default ports. The best thing that can happen is that whatever implementation of Scheme you're using will ignore the attempt or report it as an error.

Short Reference

```

(close-input-port input-port)
  Close an open input port.
(close-output-port output-port)
  Close an open output port.
(current-input-port)
  Get the current input port, typically associated with the keyboard.
(current-output-port)
  Get the current output port, typically associated with the screen.
(delete-file filename)
  Delete the specified file, if it exists.

```

`(write value)`
Print a more readable representation of the value on the screen.

`(write value output-port)`
Print a more readable representation of the value to the specified port.

`(eof-object? value)`
Determine if the given value is something returned by `read` to indicate the end of the file.

`(file-exists? filename)`
Determine whether the specified file exists.

`(input-port? value)`
Determine whether the given value is an input port.

`(newline)`
Write a newline to the screen.

`(newline port)`
Write a newline to the specified port.

`(open-input-file filename)`
Open the specified file for reading. Returns an input port.

`(open-output-file filename)`
Open the specified file for writing. Returns an output port.

`(output-port? value)`
Determine whether the given value is an output port.

`(peek-char input-port)`
Determine the next character available on the specified port.

`(read input-port)`
Read the next value available on the specified port.

`(read-char input-port)`
Read the next character available on the specified port.

`(write value)`
Print the verbose representation of the value on the screen.

`(write value output-port)`
Print the verbose representation of the value to the specified port.

Copyright © 2006 Samuel A. Rebelsky. This work is licensed under a Creative Commons Attribution-NonCommercial 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.