

Files

- Input Ports
- A Simple Sample File
- Reading one value at a time
- Reading one character at a time
- Finding the end of a file
- File Recursion
- Creating New Files
- Writing One Character At A Time
- Miscellaneous Facilities

Input Ports

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.

To provide for this possibility, each of Scheme's input procedures can be provided with an extra argument that specifies the *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 or MzScheme 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/CS151/2001S/Examples/hi.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 invoked within some binding construction, such as a definition or a `let`-expression:

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

or

```
(let ((source (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

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

A Simple Sample File

The `hi.dat` file is a text file that contains one line, consisting of the cheerful greeting `Hi there!`. One can now access the contents of the file by calling Scheme's built-in input procedures, but giving them the input port `source` as an argument.

Reading one value at a time

The easiest way to read from an input port is to use the `read` procedure that you've already encountered for reading from the keyboard. When you want `read` to read from the keyboard, you give it no parameters. When you want `read` to read from an input port, you give it that port as a parameter.

For example, let's read the first value in the sample file.

```
(let* (; Prepare to read from a file
      (source (open-input-file "..."))
      ; Read one value
      (value (read source)))
  ; We're done, so clean up.
  (close-input-port source)
  ; Return the value read
  value)
```

Reading one character at a time

An input port can also be used as an argument to two primitive input procedures: `read-char`, which reads in (and returns) one character from the file on the other side of the input port, and `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.

For example, using the `source` input port that we defined above:

```
> (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. MzScheme prints the end-of-file object as `#<eof>`. To continue the preceding example,

```
> (define source
  (open-input-file "/home/rebelsky/Web/Courses/CS151/2001S/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>

```

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)
    ; Get the next character.
    (let ((next (read-char 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.
      (if (or (eof-object? next)
              (char=? next #\newline))
          null
          ; Otherwise, read the remaining characters and shove this
          ; one on the front of the list.
          (cons next (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:
;;;   firstline
;;; 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 firstline
  (lambda (file-name)
    (let ((source (open-input-file file-name)))
      (display "The first line of ")
      (display file-name)
      (newline)
      (display (read-line source))
      (newline)
      (close-input-port source))))

```

File Recursion

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

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. [Verified]
;;; 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)
    (let* ((source (open-input-file file-name))
           (result (sum-of-file-kernel file-name source)))
      (close-input-port source)
      result)))

;;; Helper:
;;;   sum-of-file-kernel
;;; 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.
;;;   Crashes (with an error message) if the file contains
;;;   non-numbers. In that case, it still closes the input port.
(define sum-of-file-kernel
  ; A helper to a helper. Used only when we need to crash and burn.
  (let ((failure (lambda (file-name source)
                   (close-input-port source)
                   (error "sum-of-file"
                          (string-append "The file "
                                          file-name
                                          " contains a non-number."))))))
    (lambda (file-name source)
      ; Read a value from the port.
      (let ((nextval (read source)))
        (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) 0)
; Have we just read a number? If so, add it to the sum of the
; remaining numbers.
((number? nextval) (+ nextval
                      (sum-of-file-kernel file-name source)))
; Hmmm ... something has gone horribly wrong.
(else (failure file-name source))))))

```

In the base case of the recursion, there are no left numbers in the file, and the call to the `read` procedure immediately returns the end-of-file object. The helper returns 0. The main function closes the file and returns the 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 one of its preconditions has been violated, and it closes the file and reports the error.

Creating New Files

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.

To provide for this possibility, each of Scheme's output procedures can be provided with an extra argument 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, there are 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`.

As an example, here's a procedure that takes two arguments -- the first a string that names the output file to be created, the second a positive integer -- and writes the exact divisors of the positive integer into the specified output file:

```

;;; 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)
    ; Get ready to write to the file.
    (let ((target (open-output-file file-name)))
      (store-divisors-kernel target 1 dividend)
      (close-output-port target))))

;;; Helper:
;;; store-divisors-kernel
;;; 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-kernel
  ; A simple helper to write a number to a file
  (let ((write-number (lambda (target value)
                        (write value target)
                        (newline target))))
    (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))))))

```

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, and the analogous predicate `output-port?`.

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.