

Class 33: File Basics

Held: Tuesday, 1 April 2008

Summary: We consider *files*, a technique for structuring information that permits the information to persist across invocations of Scheme. Files also let our Scheme programs share information with other programs.

Related Pages:

- EBoard.
- Lab: Files.
- Reading: Files.

Notes:

- I will continue to reserve time at the start of class for discussion of campus issues.
- Happy April Fool's Day!
- I distributed notes on exam 2 via email. Please let me know if you did not get those notes.
- Are there questions on Assignment 7?
- Reading for tomorrow: Pixel Maps.

Overview:

- File basics: Data persistence and beyond.
- Ports.
- Basic file operations.

Why Use Files?

- As I hope you've figured out by now, it is possible (although not necessarily easy) to use Scheme to do "anything" you can do on the computer.
- Two similar things that you often want to do are to save data to files and to recover data from files.
- Why?
 - So that data can last a long time.
 - So that you can deal with more data than you can easily enter by hand.
 - So that you can write a word processor.
 - ...
- As you might guess, you can do both activities with Scheme.

Ports

- Rather than dealing directly with files, Scheme adds a layer of abstraction called a *port*.
- Each port is associated with something that can be used for input or output.
 - That thing can be a file.
 - That thing can also be the keyboard (for input), the screen (for output), or a network connection.
- Why do we have ports?
 - So that the process of writing anywhere (or reading anywhere) is the same; our code doesn't need to change.
 - So that we can read from the same file more than once simultaneously and not get lost about where we are in the file.
- In effect, an input port has two parts:
 - a link to the file that it is associated
 - a *cursor* that indicates where in the file the next read should take place.
- To create a port that corresponds to a file that you want to read from, use `(open-input-file file-name)`.
- To create a port that corresponds to a file that you want to write to, use `(open-output-file file-name)`.
- You can read from input ports with `(read port)`
- You can write to ports with
 - `(newline port)`
 - `(write value port)`
 - `(display value port)`
- When you're done with an input port, use `(close-input-port port)`
- When you're done with an output port, use `(close-output-port port)`
- What does `read` do when there's nothing left in the file? It returns a special value (which DrScheme displays as `#<eof>`).
- You can tell that that value indicates the end of the file with `eof-object?`

Characters

- You can read a character at a time (rather than a value at a time) using `(read-char port)`.
- Why is this useful? It lets you be more general than the basic `read` procedure.
- You can also look at the next character that you're about to read using `(peek-char port)`
- Why is this useful? Sometimes you want to read differently based on what you see next.
 - If you see a semicolon, read the whole line as a string and ignore it.
 - If you see anything else, read it as a Scheme value
- When `read-char` encounters the end of the file, it returns the same special value as `read`

Lab

- Do the lab on files.
-

Copyright © 2007-8 Janet Davis, Matthew Kluber, and Samuel A. Rebelsky. (Selected materials copyright by John David Stone and Henry Walker and used by permission.) This material is based upon work partially supported by the National Science Foundation under Grant No. CCLI-0633090. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. 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.