

Class 23: Recursion with Files

Held: Wednesday, February 28, 2007

Summary: Today we expand our exploration of input and output to include files, particularly how one recurses over files.

Related Pages:

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

Notes:

- EC for tomorrow's Thursday extra (at noon) and convocation (at 11 a.m.).
- Are there any questions on HW8 or HW9?

Overview:

- Remaining I/O Topic: Repetitive Prompting.
- About Files.
- Key File Operations.
- File Recursion.
- Lab.

Repetitive Reading

- Some of you asked about how to write the solution for the “read a value and square it” problem in which you repeatedly prompt when given a non-number.
- Recursion is our natural mechanism for repetition.
- In the purest form, we don't even need a separate procedure; we just use a named let or letrec.

```
(letrec ((prompt-and-square (lambda ()
                              ...
                              ; if it's not a nubmer, try again
                              (prompt-and-square)
                              ...)))
  (prompt-and-square))
```

- We'll fill in the rest of the code in class.

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.
- 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?`

Processing Files Recursively

- Since files often contain an unpredictable amount of information, we typically process files recursively.
- You may recall that the pattern for recursion is

```
(define recursive-proc
  (lambda (val)
    (if (base-case-test)
        (base-case val)
        (combine (partof val)
                  (recursive-proc (simplify val))))))
```

- For files, the base-case-test is almost always

```
(eof-object? (peek-char input-port))
```

- The simplify is a bit more indirect. When we read a character or value from a file, we have gotten closer to the end of the file.
- Hence, the form is often

```
(define recursive-proc
  (lambda (input-port other-params)
    (if (eof-object? (peek-char input-port))
        (close-and-return input-port (base-case other-params))
        (combine (read input-port)
                  (recursive-proc input-port other-params))))
(define close-and-return
  (lambda (input-port return-value)
    (close input-port)
    return-value))
```

- For writing to a file, the recursion looks a bit more typical. That is, it depends on what we're recursing over.

Lab

- Continue the lab on files.

Copyright © 2007 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.