Input / Output

CSE 413, Autumn 2002

Programming Languages

http://www.cs.washington.edu/education/courses/413/02au/

Readings and References

- Reading

- Other References
  - » Section 6.6, *Revised^5 Report on the Algorithmic Language Scheme (R5RS)*
  - » Chapter 11, Input and Output, *PLT MzScheme: Language Manual*

Input / Output

- Typically, I/O is highly implementation and system dependent with lots of side-effects
  - » information about the underlying file system leaks out into the code, which makes it less portable and less elegant
- However, writing programs that produce useful results usually means some sort of connection to the outside world
  - » open, read, write, close - everybody does it ...

Ports

- Scheme input / output procedures let you read from an input port or write to an output port
  - » an input port is a Scheme object that can deliver characters upon command
  - » an output port is a Scheme object that can accept characters
- Ports are an abstraction
  - » not necessarily a physical device
- Ports can be associated with
  - » the console, files, strings (MzScheme)

Ports

- An input port is associated with a data source by opening the source
  - » **(open-input-file *filename*)**
  - » **(open-input-string *string-identifier*)**
- An output port is associated with a data sink by opening the sink
  - » **(open-output-file *filename*)**
  - » **(open-output-string)**
- All these procedures return a port identifier which can then be passed to the I/O procedures

Reading

- There are standard readers for
  - » characters
  - » s-expressions (complete Scheme expressions)
- Readers
  - » take an optional input port argument
  - » return the next character or a complete s-expression
  - » return an eof object on end-of-file read

```
(read)                          (read-char)
(read input-port)               (read-char input-port)
(eof-object? obj)
```


Writing

- There are standard writers for
  - » characters
  - » Scheme objects
- Writers
  - » take an optional output port argument
  - » `display` generates text intended for human readers
  - » `write` generates text intended for machine readers

```
(display obj)                   (write obj)
(display obj output-port)       (write obj output-port)

(newline)                       (write-char char)
(newline output-port)           (write-char char output-port)
```


Automatic open / close

- You can call procedures and have Scheme do the file open and close for you
- Scheme opens the file and assigns the result to the current input port or current output port

```
(with-input-from-file string proc)

(with-output-to-file string proc)
```

File utilities

- These procedures are not part of the Scheme standard, but are available in most implementations
- (file-exists? path)
  - » checks if its argument string names an existing file
- (delete-file path)
  - » deletes its argument file

Simple reader and writer procedures

```
; read items from a port

(define (reader port)
  (let ((obj (read port)))
    (if (not (eof-object? obj))
        (begin
          (display "This object was read in: ")
          (display " ")
          (write obj)
          (newline)
          (reader port)
          ))))

; display an item m on a port

(define (writer port m)
  (write m port)
  (newline port))
```

reader and writer do not know anything about the ports that they are using

io.scm and string-io.scm

file-evaluator procedure

```
; read, evaluate, print loop

(define (repl iport oport)
  (let ((obj (read iport)))
    (if (not (eof-object? obj))
        (begin
          (write obj oport)
          (display " => " oport)
          (display (evaluator obj) oport)
          (newline oport)
          (repl iport oport)
          ))))
```

also plot-tree.scm

Strings

- Strings are sequences of characters
- String literals are written with "double quotes"
  - » write a quote in a string as \" and backslash as \\
- Strings are not symbols
  - » a symbol is an object with a unique name
  - » a string is a sequence of characters
- There are numerous string procedures

```
(string-append s1 s2 …)
(substring string start end)
(string-length string)
…
```