CSCI-C311 Programming Languages

Racket: Input and Output

Dr. Hang Dinh

1

Reading Assignment for This Lecture

- The Racket Guide
  - Part 8. Input and Output
  - 8.1 Varieties of Ports
  - 8.2 Default Ports
  - 8.3 Reading and Writing Racket Data
  - 8.5 Bytes, Characters, and Encodings
  - 8.6 I/O Patterns

2

Racket Ports

- A Racket port
  - represents a source or sink of data such as a file, a terminal, a TCP connection, or an in-memory string
  - provides sequential access in which data can be read or written a piece at a time, without requiring the data to be consumed or produced all at once.
- Types of ports
  - *Input port*: represents a source from which a program can *read* data
  - *Output port*: represents a sink to which a program can *write* data
- A Racket port corresponds to the Unix notion of a stream
  - not to be confused with Racket streams.

3

Handling File Ports

- The open-output-file function opens a file for writing, returns an output port

```
> (define out (open-output-file "data"))
> (display "hello" out)
> (close-output-port out)
```

Function display writes a string to an output port.

- The open-input-file function opens a file for reading, returns an input port

```
> (define in (open-input-file "data"))
> (read-line in)
"hello"
> (close-input-port in)
```

Function read-line returns a string containing the line of bytes from an input port.

4

Handling File Ports

- If a file exists already, open-output-file raises an exception by default.
  - Use option #:exists 'truncate or #:exists 'update to re-write or update the file:

```
> (define out (open-output-file "data" #:exists 'truncate))
> (display "howdy" out)
> (close-output-port out)
```

  - The 'update writes to beginning of the file without deleting old data

```
> (define out (open-output-file "data" #:exists 'update))
> (display "Hi" out)
> (close-output-port out)
> (read-line (open-input-file "data"))
"Hiwdy"
```

5

Handling File Ports

- Instead of having to match the open calls with close calls, we can use the call-with-input-file and call-with-output-file functions
  - which take a function *f* to call to carry out the desired operation.
  - Function *f* gets as its only argument the port, which is automatically opened and closed for the operation.

```
> (call-with-output-file "data" #:exists 'update
                         (lambda (out) (display "hello1" out)))
> (call-with-input-file "data"
                         (lambda (in) (read-line in)))
"hello1"
```

6

Handling String Ports

- The open-output-string function creates a port that accumulates data into a string, and get-output-string extracts the accumulated string.
- The open-input-string function creates a port to read from a string.

```
> (define p (open-output-string))
> (display "hello" p)
> (get-output-string p)
"hello"
> (read-line (open-input-string "goodbye\nfarewell"))
"goodbye"
```

7

Default Ports

- If target port is omitted in an I/O function, the default port is
  - the *current input port*, returned by function current-input-port
  - the *current output port,* returned by function current-output-port
  - the *current error port,* returned by function current-error-port
- If you start the racket program in a terminal, then the current input, output, and error ports are all connected to the terminal.

```
> (display "Hi")
Hi
> (display "Hi" (current-output-port)) ; the same as above
Hi
> (display "Ouch!" (current-error-port))
Ouch!
```

8

Reading and Writing Racket Data

- Function read reads and returns a single datum from an input port
- Racket provides 3 ways to print an instance of a built-in value
  - write - prints a value in such a way that read on the output produces the value back
  - display – similar to write but character or byte datatypes are written as raw strings and characters without any adornments such as quotation or tick marks
  - print - prints a value in the same way that is it printed for a REPL result, similar to write but adds a bit more formatting to the output
- Overall,
  - print corresponds to the expression layer of Racket syntax,
  - write corresponds to the reader layer,
  - display roughly corresponds to the character layer.

9

Comparing Racket Data Writing Forms

```
> (print #\A)
#\A
> (print "Hello")
"Hello"
> (print #"Goodbye")
#"Goodbye"
> (print '("a" b c))
'("a" b c)
> (print #(a b c))
'#(a b c)
> (print 1/2)
1/2
```

```
> (write #\A)
#\A
> (write "Hello")
"Hello"
> (write #"Goodbye")
#"Goodbye"
> (write '("a" b c))
("a" b c)
> (write #(a b c))
#(a b c)
> (write 1/2)
1/2
```

```
> (display #\A)
A
> (display "Hello")
Hello
> (display #"Goodbye")
Goodbye
> (display '("a" b c))
(a b c)
> (display #(a b c))
#(a b c)
> (display 1/2)
1/2
```

10

Writing Data with New Line

- Functions println, writeln, displayln print a value like print, write, and display, respectively, but with a new line at the end.

```
> (print "give me ") (print "money")
"give me ""money"
> (println "give me ") (println "money")
"give me "
"money"
> (display "give me ") (display "money")
give me money
> (displayln "give me ") (displayln "money")
give me
money
```

11

Formatting Output

- The printf function supports simple formatting of data and text.
  - It takes a format string as its first argument and any number of other values as its other arguments
  - The format string uses ~a, ~s, or ~v as a placeholder
    - ~a displays the next argument,
    - ~s writes the next argument,
    - ~v prints the next argument.
  - There must be one placeholder for each of the arguments after format string

```
> (printf "Items ~a for ~s: ~v" '("list") '("John") '("milk"))
Items (list) for ("John"): '("milk")
```

12

Bytes, Characters, and Encoding

- Functions like read-line, read, display, and write all work in terms of characters (which correspond to Unicode scalar values).
  - Conceptually, they are implemented in terms of read-char and write-char.
- Ports read and write bytes, instead of characters.
  - The functions read-byte and write-byte read and write raw bytes.
- The read-char and write-char functions are conceptually implemented in terms of read-byte and write-byte.
  - A single byte's value less than 128 corresponds to an ASCII character.
  - Any other byte is treated as part of a UTF-8 sequence, where UTF-8 is a particular standard way of encoding Unicode scalar values in bytes.

13

Bytes, Characters, and Encoding

- A single read-char may call read-byte multiple times
- A single write-char may generate multiple output bytes.

```
> (define strport (open-output-string))
> (write-byte 206 strport)
> (write-byte 187 strport)
> (read-char (open-input-string (get-output-string strport)))
#\λ
```

The 2-byte sequence 206 187 is the UTF-8 encoding of λ

```
> (define strport1 (open-output-string))
> (write-char #\λ strport1)
> (read-byte (open-input-string (get-output-string strport1)))
206
```

14

Processing Individual Lines of a Port

- To process individual lines of a port, you can use for with in-lines:

```
> (define (upcase-all in)
    (for ([l (in-lines in)])
      (displayln (string-upcase l))))
> (upcase-all (open-input-string
               (string-append
                "Hello, World!\n" "Welcome!")))
HELLO, WORLD!
WELCOME!
```

15