A Little More Scheme

CS152

Chris Pollett

Nov. 5, 2008.

Outline

- More Message Passing
- Some built-in functions in Scheme

Closures

- Last day, we showed an example of a function returned by a function:

```
(define make-new-balance (lambda (balance)
   (lambda (amount)
      (if (< balance amount) "insufficient funds"
      (begin
          (set! balance (- balance amount))
          balance)))))
```

- So if I call

  (define my-acct (make-new-balance 100))

  my-acct a function whose value of balance is 100.
- That is, this function remembers the value of balance from the scope in which it was originally defined. This is called a **closure**.
- Calling (my-acct 20) uses set! to change this value of balance to a new value.
- So we have the effect of being able to remember state in functions defined by closures. Thus, we can use the message passing way to fake classes.

Message Passing in Scheme

- The basic way we can set up a class in Scheme is to define a constructor:

```
(define my-class (lambda (construct-arg1 …)
    (let ((my-field1 val1) ; it is also legal to nest define's in Scheme
          (my-field2 val2) …)
          (lambda (msg . args)
              (cond ((eqv? msg msg1)
                     ;do-some-action
                     )…
              ))))
```

- We can then create an instance of the class using:

```
(define my-instance (my-class construct-val1 …))
```

- We can then pass messages to the instance using lines like:

```
(my-instance msg1 msg1-args)
```

Some Built-in Functions

- Characters:
  - Character literals can be written like: #\a, #\b, #\space, #\newline, etc.
  - char=?, char<?, char-ci<?, char-alphabetic?, char-whitespace?, etc for comparing characters
  - Case conversion: char-upcase, char-downcase
  - Type conversion: char->integer, integer->char
- Strings:
  - String literals can be written like: "hi there"
  - string=?, string<?, string-ci<?, etc, can be used for comparing strings.
  - (string) - creates "", (string #\a #\b #\c) -creates "abc" etc.
  - string-length - returns the length of a string
  - (string-ref "hello" 3) returns 3rd char from "hello"
  - (string-set! str 3 #\m) changes 3rd char of str (provided mutable) to m.
  - string-copy - returns a copy of a string; string-append - concatenates the strings in its input; (substring string start end) returns a substring of given range.
  - string->list - converts the string to a list consisting of its characters
  - list->string - converts a list of chars to a string.

Vectors

- Lists in Scheme are implemented in memory as linked-lists. So you need to traverse the first i-1 elements of a list to get to the ith element.
- This can often be slow. Vectors in Scheme are more like arrays in other languages:
  - They have a fixed number of elements, but they support random access look-up.
- The notation for a vector is #(elt1 elt2 elt3).
- (vector) - creates the empty vector #(); (vector 'a 'b) - creates #(a b)
- (make-vector n) - creates a vector of length n; (make-vector n 'a) - creates a vector of length n all of whose elements are a.
- vector-length - returns the length of a vector
- (vector-ref vec n) - returns n elt of vec
- (vector-set! vec n obj) - sets the nth elt of vec to obj
- (list->vector list) - converts a list to a vector
- (vector->list vec) - converts a vector to a list

Symbols

- There are a couple functions which are useful to help one convert between strings and symbols:
  - (string->symbol "hi") outputs hi symbol
  - (symbol->string 'hi) outpus "hi" string

Input

- Input and output in Scheme is done using ports which are first-class objects. Such as #<port>
- These can be thought of as filehandles in other languages.
- (input-port? obj) - checks if obj is a port
- (current-input-port) - returns the current input port; (set-current-input-port! port) - sets this port
- (open-input-port filename) - opens the file and returns an input port to it.
- (close-input-port input-port) - closes the input port.
- (read port) - reads from input port. If at end of file returns an eof-object. If no port-supplied reads from current-input-port.
- (eof-object? obj) - checks if obj is a an end of file object.
- read-char, peek-char, char-ready? Similar to read but for characters.

Output

- Most of these funtions have names analogous to the names for input functions:
  - current-output-port, set-current-output-port!, output-port?, open-output-port, close-output-port.
- To write the functions are: (write obj port), (display obj port), (write-char char port), (newline port).
- If you do not have a port, then it defaults to the current one.