CSE1729 - Introduction to Programming April 15, 2017

Laboratory Assignment 11 Solutions

Objectives

- Work with objects

Activities

1. Extend the bank account example in the slides with the following upgrade and new methods:
   - `withdraw` – modify this method so a negative balance is shown instead of a printed message.
   - `accrue` – add 1 year of simple interest to the balance. At first assume an interest rate of 1 %
   - `setrate` – change the interest rate to the given argument.

   You may start with the following code which is copied from the lecture slides:

```
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (define (deposit f)
      (begin
        (set! balance
              (+ balance f))
        balance))
    (define (withdraw f)
      (if (> f balance)
          "Insufficient␣funds"
          (begin
            (set! balance
                  (- balance f))
            balance)))
    (define (bal-inq) balance)
    (lambda (method)
      (cond ((eq? method 'deposit) deposit)
            ((eq? method 'withdraw) withdraw)
            ((eq? method 'balance-inquire) bal-inq)))))
```

Solution:

```
(define (new-account initial-balance)
  (let ((balance initial-balance)
        (interestrate 0.01))
    (define (deposit f)
      (begin
        (set! balance
              (+ balance f))
        balance))
    (define (withdraw f)
            (begin
              (set! balance
                    (- balance f))
              balance))
      (define (bal-inq) balance)
      (define (accrue) (begin (set! balance
                                    (+ balance
                                       (* balance
                                          interestrate)))
                              balance))
      (define (setrate r) (set! interestrate r))
      (lambda (method)
        (cond ((eq? method 'deposit) deposit)
              ((eq? method 'withdraw) withdraw)
              ((eq? method 'balance-inquire) bal-inq)
              ((eq? method 'accrue) accrue)
              ((eq? method 'setrate) setrate)))))
```

2. Create two bank account objects and demonstrate that the balance and rate can be set and modified independently.

3. For this question, we will use the heap-supporting functions as seen in the lecture slides as building blocks for this assignment to build a new heap implementation, i.e., using objects. Specifically, we will create a *priority queue* object. A priority queue is a modified queue, such that when one requests the next element off the front of the queue, the highest-priority element is retrieved first.

   The objective is to construct a priority queue object which:

   1. maintains the priority queue as a heap,
   2. uses a generic (first order) order relation,
   3. exposes methods `empty?`, `insert`, `extract-min`, and `size`.

   Thus, the object constructor for the priority queue should take one argument (the order relation), and return a dispatcher yielding 4 methods.

   Hint: Of course it is possible to implement `size` by actually traversing the heap every time it is called, but you can do something smarter by maintaining a private variable `q-size`, which is destructively updated every time something is inserted or extracted from the heap.

Solution:

```
(define (tree-lt ta tb)
  (define (weight T)
    (if (null? T)
        0
        (cdar T)))
  (< (weight ta) (weight tb)))

(define (make-heap-pqueue lt?)
  (let ((theHeap '())
    (q-size 0))
;; Tree tools
(define (make-tree value left right)
  (list value left right))
(define (value T) (car T))
(define (left T) (cadr T))
(define (right T) (caddr T))
;; Heap tools
(define (combine Ha Hb)
  (cond ((null? Ha) Hb)
        ((null? Hb) Ha)
        ((lt? (value Hb) (value Ha)) (combine Hb Ha))
        (else (make-tree (value Ha)
                         (combine (left Ha) (right Ha))
                         Hb))))
(define (heap-insert x workHeap)
  (cond ((null? workHeap) (make-tree x '() '()))
        ((lt? x (value workHeap))
         (make-tree x
                    (heap-insert (value workHeap)
                    (right workHeap))
                    (left workHeap)))
        (else
         (make-tree (value workHeap)
                    (heap-insert x (right workHeap))
                    (left workHeap)))))
;; Method definitions
(define (empty?) (null? theHeap))
(define (insert x)
  (begin
    (set! q-size (+ q-size 1))
    (set! theHeap (heap-insert x theHeap))))
(define (extract-min)
  (let ((smallest (value theHeap)))
    (begin
      (set! q-size (- q-size 1))
      (set! theHeap (combine (left theHeap)
                             (right theHeap)))
      smallest)))
(define (size) q-size)
(define (show) theHeap)
;; This is the method dispatcher
(lambda (method)
  (cond ((eq? method 'empty) empty?)
        ((eq? method 'insert) insert)
        ((eq? method 'size) size)
        ((eq? method 'show) show)
        ((eq? method 'extract-min) extract-min))))
```