CSE1729 - Introduction to Programming — April 18, 2017

Problem Set 9 Solutions

Practice with SCHEME objects

1. Define a SCHEME object—call it `basic-set`—that maintains an (initially empty) set of numbers. Your object should have methods that correspond to

   - empty?, which should return `#t` if the set is empty, and `#f` otherwise;
   - insert, which should insert a new element into the set; and
   - delete, which should delete a particular element from the set; and
   - element?, which should determine if a given element is in the set.

   Implement the set as a standard SCHEME list, and return a dispatcher that provides access to your methods, as we have done in class, via a token.

   Thus, your object should behave as follows:

```
> (define newset (basic-set))
> ((newset 'empty))
#t
> ((newset 'insert) 2)
> ((newset 'element?) 2)
#t
> ((newset 'delete) 2)
> ((newset 'empty))
#t
```

   Solution:

```
(define (basic-set)
  (let ((S '()))
    (define (empty) (null? S))
    (define (insert x) (set! S (cons x S)))
    (define (element? x)
      (define (element-subset T)
        (cond ((null? T) #f)
              ((eq? (car T) x) #t)
              (else (element-subset (cdr T)))))
      (element-subset S))
    (define (delete x)
      (define (delete-subset T)
        (cond ((null? T) T)
              ((eq? (car T) x) (delete-subset (cdr T)))
              (else (cons (car T) (delete-subset (cdr T))))))
      (set! S (delete-subset S)))
    (define (dispatch name)
      (cond ((eq? name 'empty) empty)
            ((eq? name 'insert) insert)
            ((eq? name 'element?) element?)
            ((eq? name 'delete) delete)))
    dispatch))
```

2. Define a SCHEME object—call it `stat-set`—that maintains an (initially empty) set of numbers along with several statistics pertaining to the set. Your object should have methods that correspond to

   - empty?, which should return `#t` if the set is empty, and `#f` otherwise;
   - insert, which should insert a new element into the set;
   - element?, which should determine if a given element is in the set;
   - largest, which should return the largest element in the set;
   - smallest, which should return the smallest element in the set;
   - size, which should return the size of the set; and
   - average, which should return the average of the elements in the set.

   Note that your set does *not* need to implement delete. Implement the set as a standard SCHEME list, and return a dispatcher that provides access to your methods, as we have done in class, via a token.

   Important amplification. Your `stat-set` methods that implement `largest`, `smallest`, `size`, and `average`, should use only a *constant* amount of computation to determine their return value: specifically, the amount of time they take to return should not depend on the size of the set. (Thus you *cannot* scan the list to determine its length, or the maximum element of the list.)

   How is this possible? One strategy is to set up some "private variables" in your object called, e.g. `current-max`, `current-min`, `current-length`, etc. These variables should maintain, at all times, the current values of these quantities. When a new number is inserted, you will have to update your list (that maintains the set) and, in addition, update each of these statistics appropriately.

   I think that the easiest way to "maintain" the current average is to actually maintain the current *sum* of all the elements in the set, and—when the user asks for the average—return the sum divided by the current size.

   Incidentally, why didn't we ask you to implement the delete operation?

```
(define (stat-set)
  (let ((S '())
        (max 0)
        (min 0)
        (sum 0)
        (size 0))
    (define (empty) (null? S))
    (define (average) (/ sum size))
    (define (insert x)
      (cond ((empty) (begin (set! S (list x))
                            (set! size 1)
                            (set! max x)
                            (set! min x)
                            (set! sum x)))
            ((not (element? x))
             (begin (if (> x max) (set! max x))
                    (if (< x min) (set! min x))
                    (set! sum (+ sum x))
                    (set! size (+ size 1))
                    (set! S (cons x S))))))
    (define (element? x)
      (define (element-subset T)
        (cond ((null? T) #f)
              ((eq? (car T) x) #t)
```

```
            (else (element-subset (cdr T)))))
      (element-subset S))
    (define (dispatch name)
      (cond ((eq? name 'empty) empty)
            ((eq? name 'largest) (lambda () max))
            ((eq? name 'smallest) (lambda () min))
            ((eq? name 'average) average)
            ((eq? name 'insert) insert)
            ((eq? name 'size) (lambda () size))
            ((eq? name 'element?) element?)))
    dispatch))
```

3. Using the conventional set notation $\{1,2,2\}$ and $\{1,2\}$ denote the same set (which contains the numbers 1 and 2 and no others): specifically, as far as a set is concerned, an element either appears or it does not—there is no notion of an object appearing more than once.

   A *multiset* is like a set in which elements can appear more than once. People often use the same "set style" notation for multisets: e.g., $\{1,2,2\}$ denotes the multiset in which 1 appears once and 2 appears twice. Of course, as multisets $\{1,2,2\}$ and $\{1,2\}$ are different.

   Notice that to maintain a multiset (of numbers, say), one needs to keep track of *how many times* each number appears in the set (which could be zero).

   Consider the representation of a multiset as a sorted list of pairs, where the first part of each pair represents an integer, and the second represents the number of times it appears (a non-negative integer). Thus the multiset $\{1,2,2,3,8,8\}$ would be represented as the list `((1 . 1) (2 . 2) (3 . 1) (8 . 2))`. (Note that numbers which do not appear at all in the multiset do not have a pair in the list.)

   Define a multiset object, which provides the methods:

   - empty?, which should return `#t` if the multiset is empty, and `#f` otherwise;
   - insert, which should insert a new element into the set; of course, if the element already exists in the multiset, the *number of times* it appears in the set should increase;
   - multiplicity?, which should return the number of times a given element appears in the multiset (it should return 0 if the element does not appear at all);
   - delete, which should delete one appearance of a particular element from a multiset; and
   - delete-all, which remove all appearances of an element in a multiset.

```
(define (multiset)
  (let ((M '()))
    (define (empty) (null? M))
    (define (element? x)
      (define (element-sub N)
        (cond ((null? N) #f)
              ((eq? N (caar N)) #t)
              (else (element-sub (cdr N)))))
      (element-sub M))
    (define (insert x)
      (define (insert-sub N)
        (cond ((null? N) (list (cons x 1)))
              ((eq? x (caar N)) (cons (cons x (+ 1 (cdar N)))
                                      (cdr N)))
              (else (cons (car N) (insert-sub (cdr N))))))
```

```
      (set! M (insert-sub M)))
    (define (multiplicity? x)
      (define (mult-sub N)
        (cond ((null? N) 0)
              ((eq? x (caar N)) (cdar N))
              (else (mult-sub (cdr N)))))
      (mult-sub M))
    (define (delete x)
      (define (delete-sub N)
        (cond ((null? N) N)
              ((eq? x (caar N)) (if (eq? (cdar N) 1)
                                    (cdr N)
                                    (cons (cons x (- (cdar N) 1))
                                          (cdr N))))
              (else (cons (car N) (delete-sub (cdr N))))))
      (set! M (delete-sub M)))
    (define (delete-all x)
      (define (delete-all-sub N)
        (cond ((null? N) N)
              ((eq? x (caar N)) (cdr N))
              (else (cons (car N) (delete-all-sub (cdr N))))))
      (set! M (delete-all-sub M)))
    (define (dispatch method)
      (cond ((eq? method 'empty) empty)
            ((eq? method 'insert) insert)
            ((eq? method 'multiplicity?) multiplicity?)
            ((eq? method 'delete) delete)
            ((eq? method 'delete-all) delete-all)))
    dispatch))
```

An alternate proposal. If you wish, you may give a different implementation of the multiset ADT described above. Rather than the list-based implementation described above, implement multi-set in the following way: Use your already polished set implementation from problem 1 to maintain the set of all elements that appear a non-zero number of times in the multiset. To keep track of the multiplicities (that is, the number of times each element appears), keep a local function in the object (that maps elements in the set to natural numbers) that determines how many times each element appears.

(*Watch out!* This is not as straightforward as it seems; you will have to handle your private function carefully.)

```
(define (multiset)
  (let ((M-set (basic-set))
        (mults (lambda (x) 0)))
    (define empty (M-set 'empty))
    (define (element? x)
      (> (mults x) 0))
    (define (insert x)
      (let ((current-m mults))
        (begin
          (set! mults
                (lambda (z)
```

```
                (if (eq? x z) (+ 1 (current-m x))
                    (current-m z))))
        ((M-set 'insert) x))))
  (define (delete x)
    (if (not (= 0 (mults x)))
        (begin
          (if (= 1 (mults x)) ((M-set 'delete) x))
          (let ((current-m mults))
            (set! mults
                  (lambda (z)
                    (if (= z x)
                        (- (current-m x) 1)
                        (current-m z))))))))
  (define (delete-all x)
    (begin ((M-set 'remove) x)
           (let ((current-m mults))
             (set! mults
                   (lambda (z)
                     (if (= x z) 0
                         (current-m z)))))))
  (define (dispatch method)
    (cond ((eq? method 'empty) empty)
          ((eq? method 'insert) insert)
          ((eq? method 'multiplicity) mults)
          ((eq? method 'delete) delete)
          ((eq? method 'delete-all) delete-all)))
  dispatch))
```