MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring 2006

Recitation 12 Solutions — (3/22/2006)
Mutation – Stacks and Queues

Procedures

1. `(set-car!` *cell value*`)`
   Evaluate *cell* to be a cons cell, and change its car pointer to pointer to be *value*.
2. `(set-cdr!` *cell value*`)`
   Same as `set-car!` except that it changes the cdr part.

   `set-car!` and `set-cdr!` have **no return value**.

Special Forms

1. `(set!` *name value*`)`
   Look for a prior definition for *name*, and change the binding to be the value from evaluating *value*. We'll cover this more on Thursday and Friday.

Problems

For this Scheme expression, (a) Draw a box-and-pointer representation of the expression's value. (b) Write what Scheme prints for the expression's value (if you can). (c) Show how the mutation (given on the right) affects the box-and-pointer diagram and the printed representation, assuming the value of the expression is named x.

```
(let ((w (list 3 4 5)))                              (set-car! (cdar x) (cddr x))
  (set-car! (cdr w) w)
  (list w w))
```

The inital value is unprintable (DrScheme prints it as `(#0=(3 #0# 5) #0#)`*, but this is nonstandard). After the mutation, it prints out as* `((3 () 5) (3 () 5))`*.*

For the printed list on the left, draw a box-and-pointer representation corresponding to it, using as few cons cells as possible. Write an expression that produces this box-and-pointer structure.

Show how the mutation (given on the right) affects the box-and-pointer structure and the printed representation, assuming the structure is named x.

1. `(c (b c) (a b c))`   `(set-cdr! (cdr x) (list-ref x 2))`

```
(define x (let ((z (list 'a 'b 'c)))
            (list (caddr z) (cdr z) z)))
```

After the mutation, the printed representation is `(c (b c) a b c)`

Queues and Stacks

In Lecture, we saw how to use mutation to build two data structures, queues and lists. Both structures have one operation that adds a new item to the list, and one that removes the next item. The difference is in which item is removed next.

1. A **Queue** adds new items via `enqueue!`. The accessor `dequeue!` removes the oldes item in the queue. Think lines in a store – the next customer to be served is the one that had been waiting in the queue the longest. FIFO (first-in-first-out).
2. A **Stack** adds new items via `push!`, and removes the next item with `pop!`. Think a pile of books – the next one you pick up will be the last one you added to the pile. LIFO (last-in-first-out).

Doubly linked lists

Lists in Scheme are normally singly linked – each cons cell contains a pointer to the next element in the list, but not the previous. We're going to build a doubly linked list (hereafter dubbed a `mutable-list` for no good reason), where each element contains both a link to the next and prior elements. This data structure will allow us to both add or remove elements from either the front or the rear in $\Theta(1)$ time, and to insert or remove elements in the middle of the list without affecting any other elements or needing to copy each one. We will then see how to construct both stacks and queues using the same underlying data structure.

To begin, here are some constructors and selectors for our new ADT. None of these procedures use mutation (yet). A mutable list is a tagged list, which contains a pointer to both the first and last element of the list.

```
(define (tagged-list? l tag)
  (and (pair? l)
       (eq? (car l) tag)))

(define (make-mutable-list)
  (list 'mutable-list '() '()))

(define (mutable-list? l)
  (tagged-list? l 'mutable-list))

(define (first-element m-l)
  (if (mutable-list? m-l)
      (cadr m-l)
      (error "not a mutable list")))

(define (last-element m-l)
  (if (mutable-list? m-l)
      (caddr m-l)
      (error "not a mutable list")))

(define (empty-mutable-list? m-l)
  (if (mutable-list? m-l)
      (and (eq? (first-element m-l) '())
           (eq? (last-element m-l) '()))
      (error "not a mutable list")))

(define (single-entry? m-l)
  (if (mutable-list? m-l)
      (eq? (first-element m-l)
           (last-element m-l))
      (error "not a mutable list")))
```

Instead of composing our new list out of cons cells, we're going to create a new data type called an `element` which has both an `element-value` (like the car part of a normal list element), a `element-next` and an `element-prev` that point to other elements.

```
(define (make-element e)
  (list 'mutable-list-element '() '() e))

(define (mutable-element? e)
  (tagged-list? e 'mutable-list-element))

(define (element-value e)
  (list-ref e 3))

(define (element-prev e)
  (list-ref e 1))

(define (element-next e)
  (list-ref e 2))
```

1. Define the procedure `set-last!` which modifies the first or last pointers of a mutable-list to point at the new elements. (`set-first!` is defined for you).

   ```
   ;type: mutable-list, <element|null> -> unspecified
   (define (set-first! m-l e)
     (if (mutable-list? m-l)
         (set-car! (cdr m-l) e)
         (error "not a mutable list")))
   ```

```
;type: mutable-list, <element|null>) -> unspecified
(define (set-last! m-l e)
  (if (mutable-list? m-l)
      (set-car! (cddr m-l) e)
      (error "not a mutable list")))
```

2. Define procedures `set-prev!` and `set-next!` that change the prev or next field of a mutable-element:

```
;type: element, <element|null> -> unspecified
(define (set-prev! element prev)
  (if (mutable-element? element)
      (set-car! (cdr element) prev)
      (error "set-prev: not a mutable element")))

;type: element, <element|null> -> unspecified
(define (set-next! element next)
  (if (mutable-element? element)
      (set-car! (cdr (cdr element)) next)
      (error "not a mutable element")))
```

3. Complete the definition for `add-to-front!` which takes any value and adds a new element to the front of the list containing that value. Then define `add-to-back!` which does the same for the back of the list.

```
;type: mutable-list, A -> unspecified
(define (add-to-front! lst item)
  (let ((e (make-element item)))
    (cond ((not (mutable-list? lst))
           (error "not a mutable list"))
          ((empty-mutable-list? lst)
           (set-first! lst e)
           (set-last! lst e))
          (else
           (set-next! e (first-element lst))
           (set-prev! (first-element lst) e)
           (set-first! lst e)))))
```

```
;type: mutable-list, A -> unspecified
(define (add-to-back! lst item)
  (let ((e (make-element item)))
    (cond ((not (mutable-list? lst))
           (error "not a mutable list"))
          ((empty-mutable-list? lst)
           (set-first! lst e)
           (set-last! lst e))
          (else
           (set-prev! e (last-element lst))
           (set-next! (last-element lst) e)
           (set-last! lst e)))))
```

4. Define procedures `remove-from-back!` and `remove-from-front!` which remove the first or last element and returns it.

```
;type mutable-list -> A
(define (remove-from-back! lst)
  (cond ((not (mutable-list? lst))
         (error "not a mutable list"))
        ((empty-mutable-list? lst)
         (error "list is empty"))
        ((single-entry? lst)
         (let ((e (last-element lst)))
           (set-first! lst '())
           (set-last! lst '())
           (element-value e)))
        (else
         (let ((e (last-element lst)))
           (set-last! lst
                      (element-prev e))
           (set-next! (last-element lst)
                      '())
           (element-value e)))))
```

```
;type mutable-list -> A
(define (remove-from-front! lst)
  (cond ((not (mutable-list? lst))
         (error "not a mutable list"))
        ((empty-mutable-list? lst)
         (error "list is empty"))
        ((single-entry? lst)
         (let ((e (first-element lst)))
           (set-first! lst '())
           (set-last! lst '())
           (element-value e)))
        (else
         (let ((e (first-element lst)))
           (set-first! lst
                       (element-next e))
           (set-prev! (first-element lst)
                       '())
           (element-value e)))))
```

5. Define procedures `push!` and `pop!` to use the mutable list as a stack.

```
(define push! add-to-back!)
(define pop! remove-from-back!)
```

6. Define procedures `enqueue!` and `dequeue!` to use the mutable list as a queue.

```
(define enqueue! add-to-back!)
(define dequeue! remove-from-front!)
```

7. Using either a stack or a queue (or both!) define a procedure `rpn-calc` that takes a simple arithmetic expression in postfix notation and evaluates it. You may assume a procedure `list->mutable-list` which takes a scheme list and returns the corresponding doubly-linked list. For example:

```
(rpn-calc '(1 2 +)) --> 3

(rpn-calc '(5 1 2 + - 10 + 6 / 3 *)) --> 6
```

```
(define (list->mutable-list lst)
  (define (helper l m-l)
    (if (null? l) m-l
        (begin
          (enqueue! m-l (car l))
          (helper (cdr l) m-l))))
  (helper lst (make-mutable-list)))
```

This version also works with a command called `show` which displays the value at the top of the stack.

```
(define *binary-operations*
  (list
   (list '+ +)
   (list '- -)
   (list '/ /)
   (list '* *)))

(define (rpn-calc exp)
  (let ((stack (make-mutable-list))
        (instruction-queue (list->mutable-list exp)))
    (define (rpn-eval atom)
      (cond ((number? atom)
              (push! stack atom))
             ((eq? atom 'show)
              (let ((v (pop! stack)))
                (display v)
                (newline)
                (push! stack v)))
             ((assq atom *binary-operations*)
              (let ((opl (assq atom *binary-operations*))
                    (a1 (pop! stack)))
                (let ((a2 (pop! stack)))
                  (push! stack ((cadr opl) a2 a1)))))
             (else (error "undefined operation"))))
    (define (helper)
      (if (empty-mutable-list? instruction-queue)
          (pop! stack)
          (begin (rpn-eval (dequeue! instruction-queue))
                 (helper))))
    (helper)))
```

8. Can you define rpn-calc without using any mutating procedures? Why or why not, and if so, which is better?

```
(define (rpn-calc exp)
  (define (rpn-eval stack exp)
    (cond ((null? exp) (car stack))
          ((number? (car exp))
           (rpn-eval (cons (car exp) stack)
                     (cdr exp)))
          ((eq? (car exp) 'show)
           (display (car stack))
           (newline)
           (rpn-eval stack (cdr exp)))
          ((assq (car exp) *binary-operations*)
           (let ((op (cadr (assq (car exp) *binary-operations*))))
             (rpn-eval (cons (op (cadr stack) (car stack))
                             stack)
                       (cdr exp))))
          (else (error "undefined operation"))))
  (rpn-eval '() exp))
```