stack, queue problems
We'll implement stacks and queues using the ADT, mutable-list, described in the accompanying handout. Here's an example.

(let* ((e (make-element 4))
       (x (make-mutable-list e e)))
  x)

(add-to-front! x 5)
Using the procedures for a new data type called mutable-list, provided in the accompanying handout, write the following procedures.

1. Define 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. (Recall that the car of a mutable-list is a tag, so the first list element is actually the cadr.)

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

```
(define (set-last! m-l e) ;; type: mutable-list, <element|null> → unspecified
  (if (mutable-list? m-l)
      (set-car! (caddr m-l) e)
      (error "not a mutable list")))
```
2. Define set-prev! and set-next! that change the prev or next field of a mutable-element.

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

(define (set-next! element next)
;; type: mutable-list, <element|null>→ unspecified
(if (mutable-elt? element)
  (set-car! (cdr element) next)
  (error "not mutable element")))
3a. 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.

```
(define (add-to-front! lst item)
    ;; type: mutable-list A -> unspecified
    (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))))
```
3b. Write add-to-back! which takes any value and adds a new element containing that value to the back of the list.

```
(define (add-to-back! lst item)
  ;; type: mutable-list A → unspecified
  (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))))
```
4a. Complete the definition for `remove-from-back!` which removes the last element and returns its value.

(define (remove-from-back! lst)
  ;; type: mutable-list → A
  (let ((e (make-element item)))
    (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))))))
4b. Write remove-from-front!  which removes the first element and returns its value

(define (remove-from-front! lst)
;; type: mutable-list ➔ A
(let ((e (make-element item)))
  (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))))))
stack and queue problems

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

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

6. Write `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.

```
e.g. (rpn-calc '(1 2 +)  \rightarrow  3
      (rpn-calc '(5 1 2 + - 10 + 6 / 3 *))  \rightarrow  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)))

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

(define (rpc-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)))
            (else (error "undefined operation"))))
    (define (helper)
      (if (empty-mutable-list? instruction-queue)
          (pop! stack)
          (begin (rpn-eval (dequeue! instruction-queue))
                 (helper)))))

(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)))
            (else (error "undefined operation"))))
    (define (helper)
      (if (empty-mutable-list? instruction-queue)
          (pop! stack)
          (begin (rpn-eval (dequeue! instruction-queue))
                 (helper)))))
```

8. Can you define rpn-calc without using any mutating procedure?

```
(define (rpc-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))
```