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.)

```scheme
(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
  ;; type: mutable-list, <element|null> -> unspecified
```
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

(define (set-next! element next)
  ;; type:  mutable-list, <element|null> ➔ unspecified
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.

```scheme
(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!)
          )
    )))
```

```scheme
(define (set-next! element next)
  ;; type: mutable-list, <element|null> → unspecified
```
4. Complete the definition for add-to-front! which takes any value and adds a new element containing that value to the front 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}
5. 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
6. Complete the definition for `remove-from-back!` which removes the last element and returns it.

```
(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)))
```
stack and queue problems

7. Write remove-from-front! which removes the first element and returns it.

(define (remove-from-front! lst)
  ;; type: mutable-list → A)
8. Write `push!` and `pop!` to use the mutable list as a stack.

9. Write `enqueue!` and `dequeue!` to use the mutable list as a queue.
10. 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 +) ➔ 3
     (rpn-calc '(5 1 2 + - 10 + 6 / 3 *)) ➔ 6
11. Can you define rpn-calc without using any mutating procedure?