Stacks

(define (make-stack)
  (list 'stack '()))

(define (stack? x)
  (and (pair? x) (eq? (car x) 'stack)))

1. Write stack-add:

   (define (stack-add elem stack)

2. Write stack-next:

   (define (stack-next stack)

3. Write stack-empty?:

   (define (stack-empty? stack)

Queues

(define (make-queue)
  (list 'queue '()))

(define (queue? x)
  (and (pair? x) (eq? (car x) 'queue)))
4. Write queue-add:

   (define (queue-add elem queue)

5. Write queue-next:

   (define (queue-next queue)

6. Write queue-empty?:

   (define (queue-empty? queue)

Iterators

Abstraction on top of stacks and queues. Three methods: add, next, empty?. Should be able to drop in a stack or a queue and use them interchangably.

   (define (iadd elem it)
      (cond ((stack? it) (stack-add elem it))
            ((queue? it) (queue-add elem it))
            (else (error "huh?"))))

   (define (inext it)
      (cond ((stack? it) (stack-next it))
            ((queue? it) (queue-next it))
            (else (error "huh?"))))

Problems?
(define (make-iterator constructor add next empty?)
  (let ((val (constructor)))
    (list 'iterator
      (lambda (elem)
        (add elem val))
      (lambda ()
        (next val))
      (lambda ()
        (empty? val))))

7. Write iadd:

  (define (iadd elem it)
    ...

8. Write inext:

  (define (inext it)
    ...

9. Write iempty?:

  (define (iempty? it)
    ...

10. Environment model?
11. Add a counter that counts the number of operations done on the iterator. Also add an extra procedure that returns the value of the counter.