Counter

Before this week, every time we evaluated a procedure with a given argument, we got the same value back. For example, if a procedure `(foo 7)` returned 12, `(foo 7)` would always return 12. No longer! Consider the following example:

```scheme
(define count (list 0)) (counter) ==> 1
(define (counter) (counter) ==> 2
(set-car! count (+ (car count) 1)) (counter) ==> 3
(car count))
```

There’s one problem with this approach though – what if count is defined somewhere else? Redefine `counter` to fix this problem:

```scheme
(define counter
(let ((count (list 0)))
  (lambda ()
    (set-car! count (+ (car count) 1))
    (car count))))
```

Remember

Write a function called `remember` that takes one argument `x` and returns the value of the last call to `remember`. For example:

```scheme
(remember 1) ==> #f
(remember 2) ==> 1
(remember 'x) ==> 2
(remember '(y)) ==> x
(remember -) ==> (y)
```

```scheme
(define remember
  (let ((saved (list #f)))
    (lambda (x)
      (let ((result (car saved)))
        (set-car! saved x) result)))
```

Rings

Rings are a circular structure, similar to a list. Unlike a list however, the cdr of the last pair of a ring points back to the first element:

1. Write a function called make-ring! that takes a list and makes a ring out of it. You may want to start off writing a helper procedure called last-pair.

   (define (make-ring! ring-list)
     (define (last-pair lst)
       (if (null? (cdr lst))
           lst
           (last-pair (cdr lst))))
     (or (pair? ring-list) (error "cannot ringify ()")
         (set-cdr! (last-pair ring-list) ring-list)
         ring-list))

2. Write a procedure rotate-left that takes a ring and returns a rotated version of the same ring. This procedure should take $\Theta(1)$ time, and not create any new cons cells.

   A left-rotated version of the ring above:
   (define (rotate-left ring)
     (cdr ring))

3. Write a procedure ring-length which returns the length (number of elements) in a ring

   (define (ring-length ring)
     (define (helper n here)
       (if (eq? here ring) n
           (helper (+ 1 n) (cdr here))))
     (helper 1 (cdr ring)))

4. Write a procedure rotate-right that rotates a ring to the right. Unlike rotate-left, rotate-right takes $\Theta(n)$ operations, though it still should not create any new cons cells.

   A right-rotated version of the ring above:
   (define (rotate-right ring)
     ((repeated rotate-left
       (- (ring-length ring) 1)) ring))
Ring Buffer

Using the ring procedures defined previously, design an ADT for a queue of fixed maximum capacity. It should have a constructor (make-ring-buffer n), which creates a ring of n elements. (ring-enqueue! x) should add x to the queue, and (ring-dequeue!) should return the next element from the queue. Each enqueue or dequeue operation should take constant time, and not create any new cons cells. The queue may contain at most n elements at any one time. Adding more than n elements is an error.

For example:

(define rb (make-ring-buffer 2))  --> unspecified
(ring-enqueue! rb 1)            --> unspecified
(ring-enqueue! rb 2)            --> unspecified
(ring-dequeue! rb)             --> 1
(ring-enqueue! rb 3)            --> unspecified
(ring-enqueue! rb 4)            --> error -- too many elements

(define (make-ring-buffer n)
  (define (helper n)
    (if (= n 0)
      '()
      (cons 'initial-value (helper (- n 1))))
  (let ((rl (helper n)))
    (make-ring! rl)
    (list 'ring-buffer n 0 rl rl)))

(define (ring-buffer-size-pair rb)
  (cdr rb))

(define (ring-buffer-filled-pair rb)
  (cddr rb))

(define (ring-buffer-read-pair rb)
  (cdddr rb))

(define (ring-buffer-fill-pair rb)
  (cddddr rb))

(define (empty-ring-buffer? rb)
  (if (not (ring-buffer? rb))
    (error "not a ring buffer")
    (eq? (car (ring-buffer-filled-pair rb)) 0)))

(define (full-ring-buffer? rb)
  (if (not (ring-buffer? rb))
    (error "not a ring buffer")
    (eq? (car (ring-buffer-filled-pair rb)) n)))
(error "not a ring buffer")
(eq? (car (ring-buffer-filled-pair rb))
    (car (ring-buffer-size-pair rb))))

(define (ring-enqueue! rb e)
  (cond ((not (ring-buffer? rb))
     (error "not a ring buffer"))
       ((full-ring-buffer? rb)
        (error "too many elements"))
       (else (set-car! (car (ring-buffer-fill-pair rb)) e)
        (set-car! (ring-buffer-fill-pair rb)
          (rotate-left
           (car (ring-buffer-fill-pair rb))))
        (set-car! (ring-buffer-filled-pair rb)
          (+ 1 (car (ring-buffer-filled-pair rb)))))))

(define (ring-dequeue! rb)
  (cond ((not (ring-buffer? rb))
     (error "not a ring buffer"))
       ((empty-ring-buffer? rb)
        (error "buffer empty"))
       (else
        (let ((val (car (ring-buffer-read-pair rb))))
          (set-car! (ring-buffer-read-pair rb)
            (rotate-left
             (car (ring-buffer-read-pair rb))))
          (set-car! (ring-buffer-filled-pair rb)
            (- (car (ring-buffer-filled-pair rb)) 1))
          (car val)))))