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
```

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
```
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)

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

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

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

   (define (ring-length ring)

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

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

Using the ring procedures defined previously, design an ADT for a queue of fixed maximum capacity. It should have a constructor \((\text{make-ring-buffer } n)\), which creates a ring of \(n\) elements. \((\text{ring-enqueue! } x)\) should add \(x\) to the queue, and \((\text{ring-dequeue!})\) 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:

\[
\begin{align*}
\text{(define rb (make-ring-buffer 2))} & \quad \rightarrow \text{unspecified} \\
\text{(ring-enqueue! rb 1)} & \quad \rightarrow \text{unspecified} \\
\text{(ring-enqueue! rb 2)} & \quad \rightarrow \text{unspecified} \\
\text{(ring-dequeue! rb)} & \quad \rightarrow 1 \\
\text{(ring-enqueue! rb 3)} & \quad \rightarrow \text{unspecified} \\
\text{(ring-enqueue! rb 4)} & \quad \rightarrow \text{error -- too many elements}
\end{align*}
\]