

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
 Department of Electrical Engineering and Computer Science  
 6.001—Structure and Interpretation of Computer Programs  
 Spring Semester, 1999

**Recitation – Wednesday, March 17**

## 0. Administrivia

- Don't miss lecture tomorrow! Really.
- PS #4 is due today (in section or, for Sandia, 36-115, for Hooman, 36-117).
- PS #5 is due on March 31st (not during spring break)

## 1. Functions Aren't Functions Anymore

Up until now, every time we called a procedure with the same arguments, we got the same value back. For example, if `(foo 7)` returned 12, then **every time** we called `(foo 7)` we would get back 12. Consider the example below. Draw the box and pointer diagram and evaluate the following expressions.

<pre>(define count (list 0)) (define (counter)   (set-car! count (+ (car count) 1))   (car count))</pre>	<pre>(counter) ⇒ 1 (counter) ⇒ 2 (counter) ⇒ 3</pre>
--	--

Notice that we're in danger if someone else is using the variable `count`. We'll step on each other's feet. How can we prevent this from happening?

```
(define counter
```

```
)
```

We'll see more tomorrow in lecture on how and why this technique of local state works.

## 2. Buffers

Write a function called `buffer` that takes one argument `x` and at each call, it returns the argument of the previous call to `buffer`. For example:

<pre>(buffer 1)      ⇒ #f (buffer 7)      ⇒ 1 (buffer 'x)     ⇒ 7 (buffer '(2 5)) ⇒ x (buffer +)      ⇒ (2 5)</pre>	<pre>(define buffer</pre>
---	---------------------------

```
)
```

## 3. Remembering Values

Write a function called `seen?` that takes one argument and returns `#t` only if that function has been called before with that argument. Assume you have the function `element?` that takes an object and a list, and returns `#t` if the object is in the list.

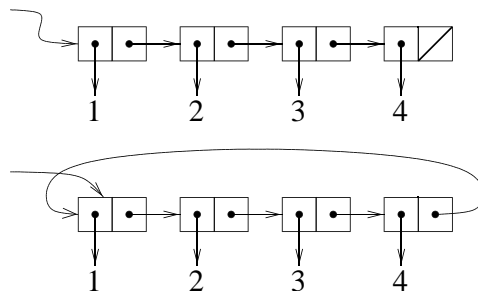
<pre>(seen? 1) ⇒ #f (seen? 'x) ⇒ #f (seen? 1) ⇒ #t (seen? 'y) ⇒ #f (seen? 'x) ⇒ #t (seen? 'y) ⇒ #t</pre>	<pre>(define seen?</pre>
--	--------------------------

```
)
```

## 4. List Mutation – Rings

Rings are circular structures similar to lists. If we define a ring `r`, `(define r (make-ring '(1 2 3 4)))`, the following is true:

```
(nth 0 r)
⇒ 1
(nth 1 r)
⇒ 2
...
(nth 4 r)
⇒ 1
```



Write the function `make-ring!` that takes a list and makes a ring out of it. *Hint: It might be helpful to write a helper function called `last-pair`.*

```
(define (make-ring! ring-list)
```

```
)
```

Write the procedure `rotate-left` that takes a ring and returns a ring that has been rotated one to the right.

```
(define (rotate-left ring)
```

```
)
```

```
(define r1 (rotate-left r))
```

```
(nth 0 r1)
```

```
⇒ 2
```

We can also rotate a ring to the right. Rotating to the right is harder than rotating to the left. Define a procedure `ring-length` that will tell us the length of the ring (which is the length of the original list).

*Hint: remember `eq?`*

```
(define (ring-length ring)
```

```
)
```

Now write the procedure `rotate-right`. If you're tired of writing helper procedures, you can use `repeated`.

```
(define (rotate-right ring)
```

```
)
```

```
(define r2 (rotate-right r))
```

```
(nth 0 r1)
```

```
⇒ 4
```

## 5. Append!

Recall the function `append` that takes two lists and returns a list of the two appended. Recall that to do this, we made a **copy** of the first list and **cons'd** it onto the second. Now that we had side effects, we can append two lists without creating a copy of one of them. Assuming you have the function `last-pair` that we wrote above, write the function `append!`.

```
(define (append! x y)
```

```
)
```