

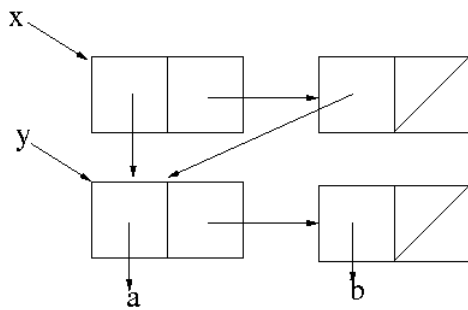
MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
 Department of Electrical Engineering and Computer Science  
 6.S184—Zombies Drink Caffeinated 6.001  
 IAP 2011

**Recitation 3**

Mutation and the Environment Model

**Mutant pairs**

Given this diagram:



1. What does `y` print as when evaluated?
2. What does `x` print as when evaluated?
3. Which of the following expressions produce the same structure?

(a) `(define x (list (list 'a 'b) (list 'a 'b)))`  
`(define y (car x))`

(b) `(define y '(a b))`  
`(define x (cons y y))`

(c) `(define x (cons 'x (cons 'x '())))`  
`(define y '())`  
`(let ((z (list 'a 'b)))`  
`(set-car! x z)`  
`(set-car! (cdr x) z)`  
`(set! y z))`

4. After evaluating `(set-cdr! (cdr x) (cdr (car x)))` what does `x` print as?

## Get it together

Previously, you've seen a procedure `append` which appends two lists by copying one of them. Write a procedure `append!` that accomplishes list concatenation without creating any new `cons` cells. Your procedure should return a pointer to the start of the list (the first `cons` cell), like so:

```
(define foo (list 1 2 3))
(define bar (list 4 5 6))
(define baz (append! foo bar))
baz => (1 2 3 4 5 6)
```

What are the advantages and disadvantages of this approach?

What happens when we evaluate these expressions?

```
(define foo (list 1 2 3))
(define bar (append! foo foo))
bar
```

## Coming or going?

Previously you wrote a procedure `reverse` which reversed a list by creating a new list with the same elements stored in the opposite order. Now, write a variant, `reverse!`, which does not create any new `cons` cells but relinks the list in-place. Then evaluate these expressions:

```
(define foo (list 1 2 3 4))
(define bar (reverse! foo))
bar
foo
```

## Can we table this for later?

Let's define an abstraction for a simple key-value table. Include a constructor, `make-table`, a mutator `put!`, and operators `has-key?` and `get-value`. Use of the table would look like this:

```
(define my-table (make-table))
(put! my-table 'ben-bitdiddle 'chocolate) => undefined
(put! my-table 'alyssa-p-hacker 'cake) => undefined
(has-key? my-table 'ben-bitdiddle) => #t
(has-key? my-table 'louis-reasoner) => #f
(get my-table 'ben-bitdiddle) => chocolate
(get my-table 'louis-reasoner) => ERROR
```

Assume that there won't be a large number of key-value pairs. What will you use to test for key-equality in `get` and `has-key`?

## Simple local state

Draw an environment diagram to figure out how the following expressions are evaluated:

```
(define bar
  (let ((result 'uninitialized))
    (lambda (x)
      (set! result
        (if (eq? result 'uninitialized)
            x
            (max result x)))
      result)))
```

```
(bar 4)
(bar 50)
(bar 2)
```

## Flip-flops

What does evaluating these expressions produce? Draw an environment diagram.

```
(define mystery
  (let ((step1 (lambda () 'flop))
        (step2 (lambda () 'flip)))
    (let ((todo (list step1 step2)))
      (set-cdr! (cdr todo) todo)
      (lambda ()
        (set! todo (cdr todo))
        ((car todo))))))
(mystery)
(mystery)
(mystery)
```

## Let's not get carried away here

What does evaluating these expressions produce? Draw an environment diagram.

```
(define (foo a)
  (let ((a 5)
        (b (lambda (x) (+ x x a))))
    (b a)))
```

```
(foo 3)
```

## Accumulation anticipated

What does evaluating these expressions produce? Draw an environment diagram.

```
(define make-accumulator
  (lambda ()
    (let ((count 0))
      (lambda (increment)
        (set! count (+ count increment))
        count))))

(define a (make-accumulator))
(a 3)
(a 2)
(define b (make-accumulator))
(b 2)
(a 1)
```

## Next verse, same as the first?

What does evaluating these expressions produce? Draw an environment diagram.

```
(define make-accumulator2
  (let ((count 0))
    (lambda ()
      (lambda (increment)
        (set! count (+ count increment))
        count))))

(define c (make-accumulator2))
(c 3)
(c 2)
(define d (make-accumulator2))
(d 2)
(c 1)
```

## Memoization

Recall the recursive definition for `fib` from the first recitation. Here's a version with an extra piece of code to print out a message each time it is called.

```
(define (fib n)
  (printf "fib called with n=~a" n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))
```

We saw that this procedure shows *exponential* growth, and worse, it computes the same thing over and over again. (How many times is `(fib 3)` called when computing `(fib 5)`?)

Ben Bitdiddle comes up with idea of keeping a list of prior results in a table, so that if called upon to compute something that's already been computed, the prior result can simply be returned again without any further work. Ben starts to modify his `fib` procedure to include a key-value table, where the keys are the values of `n` passed in, and the values are the result of computing `(fib n)`.

Alyssa P. Hacker sees this, and scolds Ben for not thinking bigger. “Surely this concept could apply to other procedures, not just `fib`, Ben!” She starts writing a procedure `memoize` which takes a procedure of one argument and returns a procedure of one argument. She shows Ben to use it like this:

```
(define memoize ... )
(define fast-fib (memoize fib))
(fast-fib 5)
```

She is dismayed to discover that her `fast-fib` still calls `fib` with the same argument multiple times. What detail did she overlook in its use? How can she fix it?

## Bonus

Write a procedure `loops?` that returns `#t` if given a list that loops back upon itself, `#f` otherwise.

```
(define safe (list 1 2 3))
(define uhoh (list 1 2 3))
(begin (append! uhoh uhoh) 'trap-set)
(loops? safe) => #f
(loops? uhoh) => #t
```