

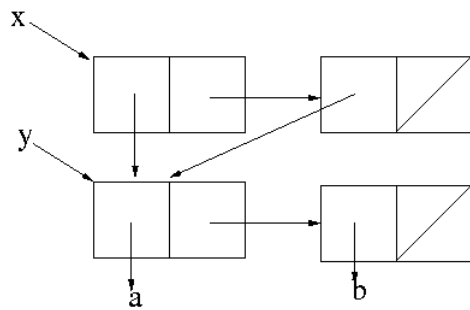
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
 Department of Electrical Engineering and Computer Science
 6.S184—Zombies Drink Caffeinated 6.001
 IAP 2011

Recitation 3

Mutation and the Environment Model ******SOLUTIONS******

Mutant pairs

Given this diagram:



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

(a)

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

No— cons cells are not shared (Two different (a b) lists) (show diagram)

(b)

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

No— missing a cons cell. Close, those. list instead of cons would work.

(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))
```

Yes— of course, because we showed it last. Draw it out.

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

((a b) (a b) b)

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)
```

Solution: Show without null? check first, then ask about input assumptions, then fix it.

```
(define append!
  (lambda (l1 l2)
    (if (null? l1)
        l2
        (begin
         (set-cdr! (last-cons l1) l2)
         l1))))

(define last-cons
  (lambda (lst)
    (if (null? (cdr lst))
        lst
        (last-cons (cdr lst)))))
```

What's different if we remove the "l1" from the end of `append!` ?

What are the advantages and disadvantages of this approach? **Mutates the original list, which might be in use elsewhere. Evaluate `foo` now.**

What happens when we evaluate these expressions?

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

Infinite list! (Dr.Scheme actually catches this when printing)

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
```

Solution: (Recursive) Discuss the base case. What if you only checked `lst` for null?

```
(define (reverse! lst)
  (if (or (null? lst) (null? (cdr lst)))
      lst
      (let ((the-rest (reverse! (cdr lst))))
        (set-cdr! (last-cons the-rest) lst)
        (set-cdr! lst '())
        the-rest)))
```

`bar => (4 3 2 1)`

`foo => (1)`

Solution: (Iterative) Why is this better? Step through with a picture before writing code.

```
(define (reverse! lst)
  (define (helper prev cur)
    (if (null? cur)
        prev
        (let ((next (cdr cur)))
          (set-cdr! cur prev)
          (helper cur next))))
  (helper '() lst))
```

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?`

Solution: This uses association lists, and a built-in procedure `assoc` which compares keys with `equal?`. There is also `assq` which compares with `eq?`

```
(define (make-table) (list 'table))
(define (put! table key value)
  (set-cdr! table (cons (list key value) (cdr table))))
(define (has-key? table key)
```

```
(not (not (assoc key (cdr table)))) ; double not to return #t/#f instead of leaking list
(define (get-value table key)
  (let ((search (assoc key (cdr table))))
    (if (not search)
        (error "Key " key " not found in table")
        (cadr search))))
```

Simple local state

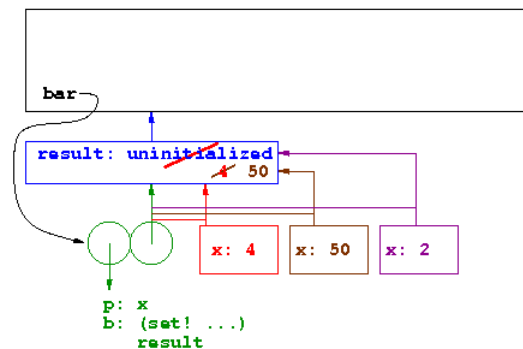
Draw an environment diagram to figure out how the following expressions are evaluated:

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

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

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

(bar 4)
;Value: 4
(bar 50)
;Value: 50
(bar 2)
;Value: 50
```



Solution:

Flip-flops

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

```

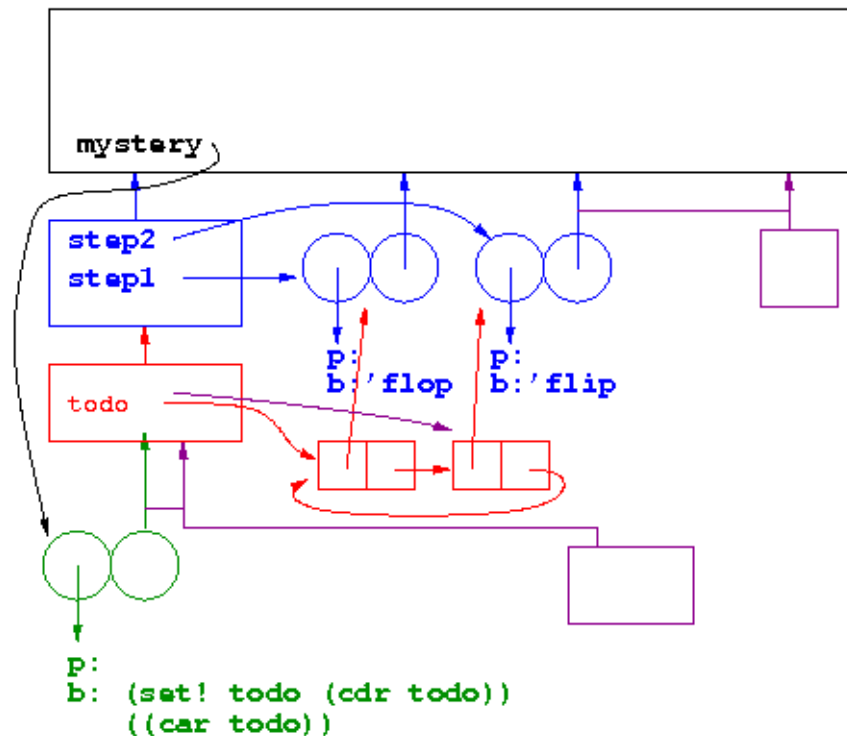
(define mystery
  (let ((step1 (lambda () 'flop))
        (step2 (lambda () 'flip)))
    (let ((todo (list step1 step2)))
      (set-cdr! (cdr todo) todo)
      (lambda ()
        (set! todo (cdr todo))
        ((car todo))))))
(mystery)
(mystery)
(mystery)

```

```

(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)

```



Let's not get carried away here

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

```

(define (foo a)
  (let ((a 5)

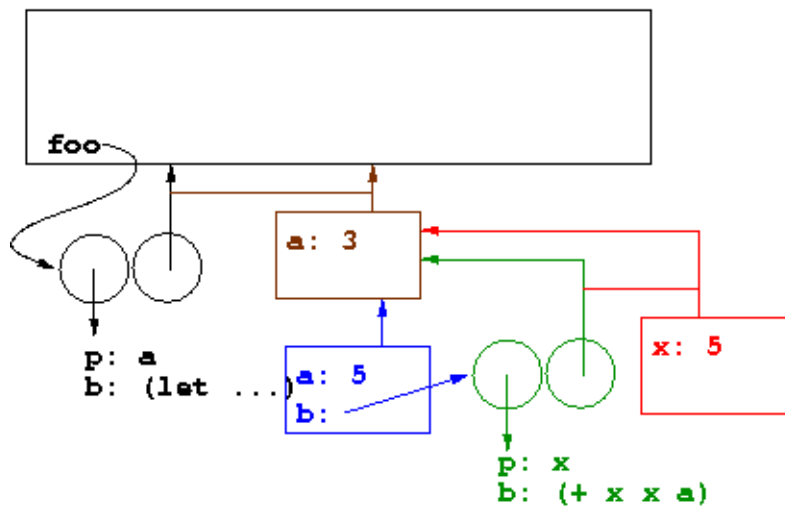
```

```
(b (lambda (x) (+ x x a)))
(b a))
```

```
(foo 3)
```

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

```
(foo 3)
;Value: 13
```



Solution:

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)
```

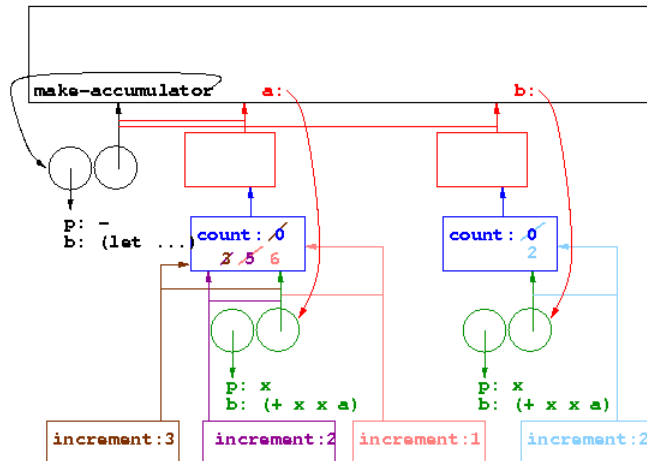
```

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

(define a (make-accumulator))
(a 3)
;Value: 3
(a 2)
;Value: 5

(define b (make-accumulator))
(b 2)
;Value: 2
(a 1)
;Value: 6

```



Solution:

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)

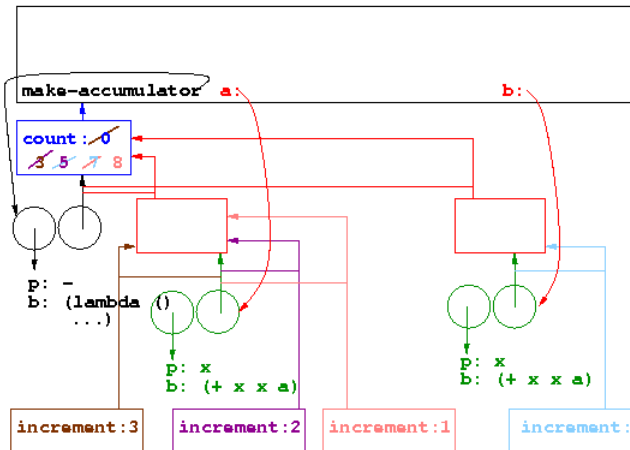
```

```

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

(define a (make-accumulator2))
(a 3)
;Value: 3
(a 2)
;Value: 5
(define b (make-accumulator2))
(b 2)
;Value: 7
(a 1)
;Value: 8

```



Solution:

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" 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)
```

Solution:

```
(define (memoize proc)
  (let ((table (make-table)))
    (lambda (x)
      (if (has-key? table x)
          (get-value table x)
          (let ((result (proc x)))
            (put! table x result)
            result))))))
```

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?

The original procedure calls fib, not fast-fib, recursively. (define fib fast-fib)

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
```

Solution: You could build a table (if it uses eq? for testing for key equality, not equal? (Else might loop!)) that notes “already visited cons cells.” Iterate down the list, checking for the end of the list or a repeated cons cell. Or, you can try this cute thing instead:

```
(define (loops? lst)
  (define (helper near far)
    (cond ((eq? near far) #t)
          ((or (null? far) (null? (cdr far))) #f)
          (else (helper (cdr near) (cddr far)))))
  (if (or (null? lst) (null? (cdr lst)))
      #f
      (helper lst (cddr lst))))
```

;Tests, not-looping:

```
(loops? '())
(loops? '(1))
(loops? '(1 2))
(loops? '(1 2 3))
```

```
;Tests, looping:
(define x (cons 1 2))
(set-cdr! x x)
(define y (list 1 2))
(set-cdr! (cdr y) y)
(define z (list 1 2 3))
(set-cdr! (caddr z) z)
(loops? x)
(loops? y)
(loops? z)
```