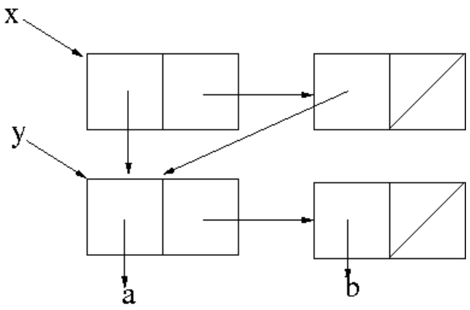
MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.037—Structure and Interpretation of Computer Programs
IAP 2019

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 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))
```

Stacking the deck

In lecture we showed a stack implementation that returned a new stack after each push and pop. Let's implement a version with mutable state. The abstraction shold include a constructor (`make-stack`), mutators (`push-stack!` and `pop-stack!`), acessors (`empty-stack?` and `stack-top`), and operators (`stack?`).

An example of use would look like:

```
(define my-stack (make-stack))
(stack? my-stack) => #t
(stack? 5) => #f
(empty-stack? my-stack) => #t
(push-stack! my-stack 'foo) => undefined
(push-stack! my-stack 'bar) => undefined
(empty-stack? my-stack) => #f
(stack-top my-stack) => bar
(pop-stack! my-stack) => bar
(pop-stack! my-stack) => foo
(empty-stack? my-stack) => #t
(pop-stack! my-stack) => ERROR
```

Solution:

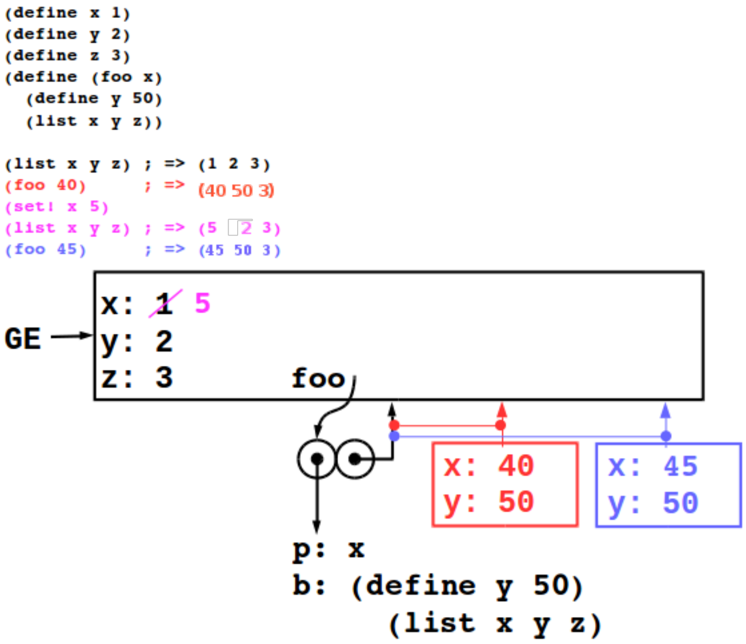
```
(define (make-stack)
  (cons 'stack '()))
(define (stack? stack)
  (and (pair? stack) (eq? 'stack (car stack))))
(define (empty-stack? stack)
  (if (stack? stack)
      (null? (cdr stack))
      (error "Object is not a stack:" stack)))
(define (push-stack! stack elt)
  (if (stack? stack)
      (set-cdr! stack (cons elt (cdr stack)))
      (error "Object is not a stack:" stack))
  stack)
(define (top-stack stack)
  (if (stack? stack)
      (cadr stack)
      (error "Object is not a stack:" stack)))
(define (pop-stack! stack)
  (if (not (empty-stack? stack))
      (let ((top (top-stack stack)))
        (set-cdr! stack (cddr stack))
        top)
      (error "Can't pop empty stack")))
```

Shadowing

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

```
(define x 1)
(define y 2)
(define z 3)
(define (foo x)
  (define y 50)
  (list x y z))

(list x y z)
(foo 40)
(set! x 5)
(list x y z)
(foo 45)
```



```
(define x 1)
(define y 2)
(define z 3)
(define (foo x)
  (define y 50)
  (list x y z))

(list x y z) ; => (1 2 3)
(foo 40)     ; => (40 50 3)
(set! x 5)
(list x y z) ; => (5 2 3)
(foo 45)     ; => (45 50 3)
```

Solution:

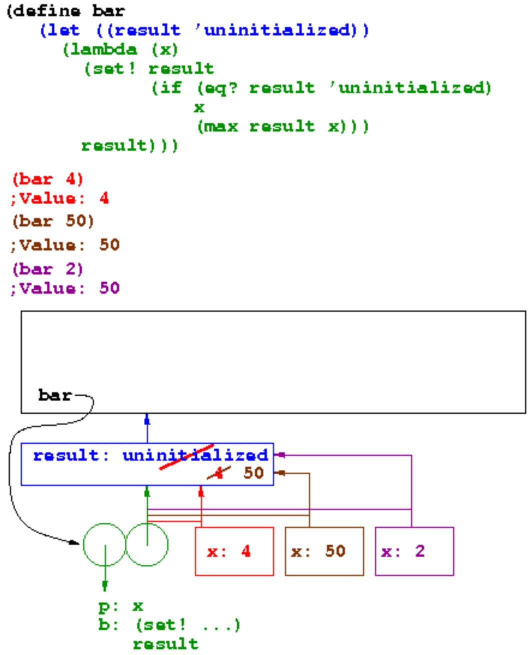
Simple local state

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)
```

Solution:



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

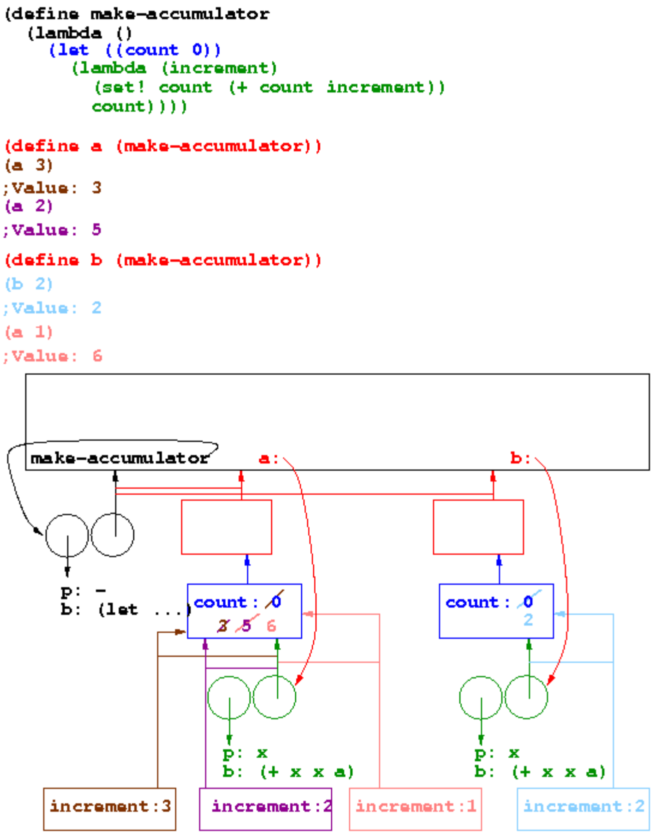
```
(bar 4)
;Value: 4
(bar 50)
;Value: 50
(bar 2)
;Value: 50
```

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
```

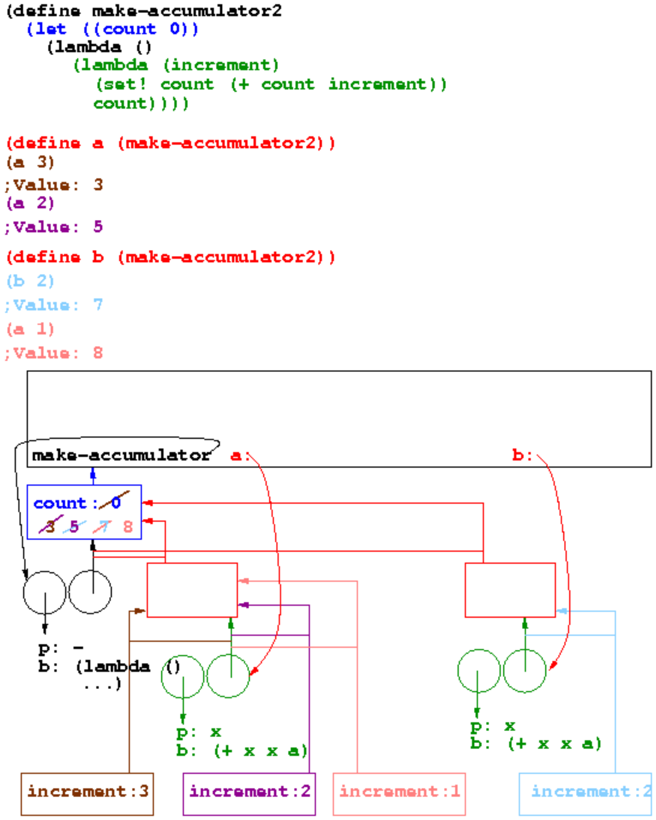
Solution:

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:

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! (cddr z) z)
(loops? x)
(loops? y)
(loops? z)
```