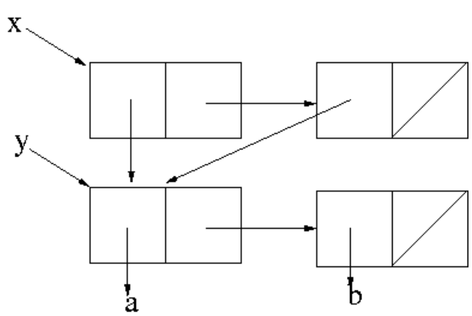
MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.037—Structure and Interpretation of Computer Programs
IAP 2018

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
```

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
```

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)
```

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)
```

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)
```

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
```