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) \[
   \begin{align*}
   &\text{define } x \text{ (list (list 'a 'b) (list 'a 'b))} \\
   &\text{(define } y \text{ (car } x)\text{)}
   \end{align*}
   \]
   
   (b) \[
   \begin{align*}
   &\text{(define } y \text{ '(a b))} \\
   &\text{(define } x \text{ (cons } y \text{ } y)\text{)}
   \end{align*}
   \]
   
   (c) \[
   \begin{align*}
   &\text{(define } x \text{ (cons 'x (cons 'x '())))} \\
   &\text{(define } y \text{ '())} \\
   &\text{(let ((z (list 'a 'b))} \\
   &\text{ (set-car! } x \text{ z)} \\
   &\text{ (set-car! (cdr } x \text{) } z) \\
   &\text{ (set! } y \text{ z)})
   \end{align*}
   \]
   
4. After evaluating \((\text{set-cdr! (cdr } x) \text{ (cdr (car } x)))\) 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:

```scheme
(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?

```scheme
(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:

```scheme
(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 should include a constructor (`make-stack`), mutators (`push-stack!` and `pop-stack!`), accessor `empty-stack?` and `stack-top`, and operators `stack?`.

An example of use would look like:

```scheme
(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
```
Simple local state

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

```scheme
(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.

```scheme
(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.

```scheme
(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.

```scheme
(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.

```scheme
(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.

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