Sample midterm 3 #3

Problem 1 (box and pointer).

What will the Scheme interpreter print in response to **the last expression** in each of the following sequences of expressions? Also, draw a "box and pointer" diagram for the result of each printed expression. If any expression results in an error, **circle the expression that gives the error message.** Hint: It'll be a lot easier if you draw the box and pointer diagram *first*!

```
(let ((x (list 1 2 3)))
  (set-cdr! (cdr x) (cddr x))
  x)


(let ((x (list 1 2 3)))
  (set-car! x (cddr x))
  x)


(let ((x (list 1 2 3)))
  (set-car! (cdr x) (cdr x))
  x)


(let ((x (list 1 2 3 4)))
  (set-cdr! (cddr x) (cadr x))
  x)
```

Problem 2 (Assignment, State, and Environments).

The textbook provides the following definition of memoization:

```
(define (memoize f)
  (let ((table (make-table)))
    (lambda (x)
      (let ((previous-result (lookup x table)))
        (or previous-result
            (let ((result (f x)))
              (insert! x result table)
              result))))))
```

[Continued on next page.]

...but Louis Reasoner has lost his book! He tries to define `memoize` as the following:

```
(define memoize                    ;; this line changed (no parens or f)
  (let ((table (make-table)))
    (lambda (f)                    ;; this line added
      (lambda (x)
        (let ((previous-result (lookup x table)))
          (or previous-result
              (let ((result (f x)))
                (insert! x result table)
                result)))))))
```

Louis takes the usual `fib` procedure:

```
(define (fib x)
  (if (< x 2)
      x
      (+ (fib (- x 1))
         (fib (- x 2)))))
```

He then tests his version of `memoize` by memoizing `fib` exactly as in the book:

```
(define memo-fib
   (memoize (lambda (x)
              (if (< x 2)
                  x
                  (+ (memo-fib (- x 1))
                     (memo-fib (- x 2))))))))
```

Louis tries out his code, and traces through it. To his surprise, it seems to work! His `memo-fib` computes the answer in $O(n)$ time! Ben Bitdiddle looks at his code and comments: "Louis, your code has a major flaw in it..."

Does Louis' `memoize` give wrong answers?

_____Yes _____No

If yes, explain why, and give an example using Louis' `memoize` that will return an incorrect result.

If no, explain what flaw Ben means, and give an appropriate example.

For full credit, your explanation must be no more than 20 words. If you give two explanations, we will grade the less correct one.

Problem 3 (Drawing environment diagrams).

Draw the environment diagram that results from the following interactions, and fill in the blank with the value printed:

```
>(define a 8)

>(define b 9)

> (let ((a 3)
        (f (lambda (b) (+ a b))))
    (f 5))
```

Problem 4 (List mutation).

The following expressions are typed, in sequence, at the Scheme prompt. Circle `#t` or `#f` to indicate the return values from the calls to `eq?`.

```
(define a (list 'x))
(define b (list 'x))
(define c (cons a b))
(define d (cons a b))

(eq? a b)                    =>    #t   #f

(eq? (car a) (car b))        =>    #t   #f

(eq? (cdr a) (cdr b))        =>    #t   #f

(eq? c d)                    =>    #t   #f

(eq? (cdr c) (cdr d))        =>    #t   #f

(define p a)
(set-car! p 'squeegee)
(eq? p a)                    =>    #t   #f

(define q a)
(set-cdr! a q)
(eq? q a)                    =>    #t   #f

(define r a)
(set! r 'squeegee)
(eq? r a)                    =>    #t   #f
```

Problem 5 (Vectors).

Here is a *selection sort* program for lists of numbers:

```
(define (ssort lst)
  (if (null? lst)
      '()
      (let ((smallest (apply min lst)))
        (cons smallest (ssort (remove smallest lst))))))
```

It finds the smallest element, pulls it out to the front, and sorts the remaining elements.

Write a procedure `ssort!` that uses the same algorithm to sort a *vector*, in place — don't allocate a new vector. Find the smallest element, exchange it with the first element, then sort the remaining subvector. You may assume that you have a helper procedure

`(subvec-min-index vector start-index)`

that finds the smallest value in the subvector starting at element number `start-index`, and returns the *position in the vector* of that smallest value:

```
STk> (subvec-min-index #(6 1 781 105 741 770) 2)
3
```

because the smallest element starting at position 2 is 105, in position 3.

Don't convert the vector to a list, or create additional vectors.

Problem 6 (Concurrency).

In the book, `make-serializer` is implemented using a mutex. `Make-mutex` is implemented using the atomic `test-and-set!` operation, like this:

```
(define (make-mutex)   ; from SICP page 312
  (let ((cell (list false)))
    (define (the-mutex m)
      (cond ((eq? m 'acquire)
             (if (test-and-set! cell)
                 (the-mutex 'acquire)))  ; retry
            ((eq? m 'release) (clear! cell))))
    the-mutex))
```

Instead, suppose that you are given serializers as a primitive capability; write `make-mutex` using serializers (and *not* using `test-and-set!`) to provide concurrency control.

Problem 7 (Streams).

Define a stream named `all-integers` that includes all the integers: positive, negative, and zero.

You may use any stream or procedure defined in the text.