Midterm 3 Review!

In addition to the midterm problems online (make sure you take a look at those too!) these are just more practice. Some questions are similar, some are not, but be able to be comfortable with the following topics:

1) OOP
2) Environment Diagrams
3) Assignment
4) Mutation (set-car! & set-cdr!)
5) Vectors
6) Above the line client/server
7) Above the line concurrency
8) Scheme 2

I may have forgotten something, but that's all I can think of at the moment =)

Problem 1: Mutation

What will Scheme print in response to each of the following expressions? Draw the box-and-pointer diagram for the result of each expression. If the expression causes an error, briefly explain why.

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

b)
```
(let ((x (list 1 2 3 4)))
    (set-car! (cddr x) (cddddr x))
     x)
```

c)
```
(let ((x (list 1 2 3 4)))
    (set-car! (cddr x) x)
     x)
```

d)
```
(let ((x (list 'surfin 'safari)))
    (set-cdr! (car x) 'jungle)
    x)
```

e)
```
(define monty (list 'the 'holy 'grail))

(define (python x)
    (set-car! x 'monty-python-and-the)
    (set! x (cons 'super-monkey (cdr x)))
    (let ((y x))
      (set-cdr! y '())
       y))

(python monty)
```
_____

```
monty
```

Problem 2: Concurrency

a) The reason for having more than one serializer is to avoid:
(CIRCLE ALL THAT APPLY)

(A) wrong answers (B) inefficiency
(C) deadlock (D) unfairness

b) For each of the following expressions, list the possible final values of moo and say whether deadlock is possible. Assume that `moo` is reset to 3 before each new parallel-execute is evaluated.

```
(define moo 3)
(define (a) (set! moo 4))
(define (b) (set! moo (* moo moo)))

(define s1 (make-serializer))
(define s2 (make-serializer))
(define s3 (make-serializer))
```

(i) `(parallel-execute a b)`
Possible values for moo: _____ Deadlock? Y/N
(ii) `(parallel-execute (s1 a) (s1 b))`
Possible values for moo: _____ Deadlock? Y/N
(iii) `(parallel-execute (s1 a) (s2 b))`
Possible values for moo: _____ Deadlock? Y/N
(iv) `(parallel-execute (s1 (s2 (s3 a))) (s1 (s3 (s2 b))))`
Possible values for moo: _____ Deadlock? Y/N

Problem 3: Environment Diagrams

a) Draw the Environment Diagram for the following Scheme session:

```
> (define (foo x)
      (let ((y 10))
         (lambda (z)
            (set! x (+ x 1))
            (set! y (+ y 10))
            (set! z (+ z 100))
            (list x y z))))
foo
> (define bar (foo 1))
bar
> (bar 100)
(2 20 200)
> (define baz (foo 5))
baz
> (baz 500)
(6 20 600)
> (baz 100)
(7 30 200)
```

b) Draw the Environment Diagrams for the following expressions

```
> (define (square x) (* x x))
> (let ((a (square 2))
        (b (+ 3 4)))
    (let ((c (+ a (let ((d 3))
                    (+ b d))))
          (e 14))
      (* (+ a b) (- e c))))
```

Problem 4: Recursion with Mutation (From Spring '96)

Write list-rotate! which takes two arguments, a nonnegative integer `n` and a list `seq`. It returns a mutated version of the argument list, in which the first n elements are moved to the end of the list, like this:

```
> (list-rotate! 3 (list 'a 'b 'c 'd 'e 'f 'g))
(d e f g a b c)
```

You may assume that `0` $\leq$ `n < (length seq)` without error checking.

Note: DO NOT allocate new pairs in your solution.

Problem 5: OOP TA's (From Spring '02 with minor TA tweeks)

We want to simulate the behavior of a TA responding to a request for a certain number of hours of wok helping students. All TAs are only supposed to help for 8 hours...after that, they go to sleep. Witness the following interactions with our simulation:

```
> (define my-ta (instantiate ta 'carolen))
> (ask my-ta 'time-left) ====> 8
> (ask my-ta 'help 24)   ====> (carolen cannot help that long) ;;
a sentence
> (ask my-ta 'help 5)    ====> (carolen helps for 5 hours)     ;;
a sentence
> (ask my-ta 'time-left) ====> 3
> (ask my-ta 'help 3)    ====> (carolen helps for 3 hours)     ;;
a sentence
> (ask my-ta 'help 1)    ====> (carolen is asleep!)            ;;
a sentence
```

a) Fill in missing blanks to define the TA class

```
(define-class (ta name)

  (method (help time)
     (cond ((= 0            )

                                                              )
           ((> time         )

                                                              )
           (else

                                                              ))))
```

b) Lous Reasoner wants to define a head-TA class which is exactly like a TA except that his name has head-ta-

prepended to it, and every time a head-TA works, he gets tired- prepended to the front of his name. Here is

his attempt:

```
(define-class (head-ta name)
  (parent (ta (word 'head-ta- name))
  (method (help time)
     (set! name (word 'tired- name))
     (usual 'help time)))
```

Predict the output from the following interaction:

```
> (define my-head-ta (instantiate head-ta 'alex))
> (ask my-head-ta 'name)   ===>
> (ask my-head-ta 'help 3) ===>
> (ask my-head-ta 'name)   ===>
> (ask my-head-ta 'help 3) ===>
```

c) Several TAs often gather to lead a `help-session`, which we'd also like to simulate. You can add TAs to a `help-session` and ask for the `total-hours` of TA help time there is left. When someone asks for help from the `help-session`, if there are enough `total-hours`, one-by-one the TAs are asked to help, in order that they were added, until the help is finished. The `help` method returns a list of the interaction with all the TAs asked:

```
> (define hs (instantiate help-session))
> (ask hs 'add (instantiate ta 'chung))
> (define t1 (instantiate ta 'tyler))
> (ask hs 'add t1)
> (ask t1 'help 8)  ;; asked outside of help session, falls
asleep
(tyler helps for 8 hours)
> (ask hs 'add (instantiate ta 'igor))
> (ask hs 'total-hours)      ;; tyler's asleep
16
> (ask hs 'help 10)
((chung helps for 8 hours) (tyler is asleep!) (igor helps 2
hours))
> (ask hs 'help 7)
(sorry tha is asking too much)
> (ask hs 'help 6)
((chung is asleep!) (tyler is asleep!) (igor helps 6 hours))
```

Complete the definition of the help-session class below.

```
(define-class (help-session)

   _____________________________________________________

   _____________________________________________________

   _____________________________________________________
  (method (help time)
    (if (> time ____________________________________ )
        '(sorry that is asking too much)

         _____________________________________________ ))
```

```
(method (help-helper time _________________________ )
  (if (= time 0)
      '()
      (let ((time-this-ta-takes (min time _______________ )))
        (let ((time-remaining (- time time-this-ta-takes)))

           ____________________________________________________

           ____________________________________________________ )))))
```