Sample midterm 3 #2

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! (car x) 4)
  x)
```

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

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

```
(define a ((lambda (z) (cons z z)) (list 'a)))
(set-cdr! (car a) '(b))
a
```

Problem 2 (Assignment, State, and Environments).

In this problem you're going to write a piece of a simplified Adventure game, not using our OOP notation, just in regular Scheme. We are concentrating on the behavior of people, so a place will just be represented as a room number. You are given a function `next-room` that takes as arguments a room number and a direction; its result is the room where you end up if you move in the given direction from the given room:

```
==> (next-room 14 'South)
9
```

means that room 9 is south of room 14. You are to write a procedure `make-player` that creates a player object. This object (i.e., a procedure) should accept messages like `South` and should move from its current position in the indicated direction. It should remember the new location as local state, and should also return the new location as its result. The argument to `make-player` is the initial room:

```
==> (define Frodo (make-player 14))
FRODO
==> (Frodo 'South)
9
==> (Frodo 'South)
26
```

You *must* make each move relative to the result of the previous move, not starting from the initial room each time!

Problem 3 (Drawing environment diagrams).

Draw the environment diagram resulting from evaluating the following expressions, and show the result printed by the last expression where indicated.

```
> (define foo
    (lambda (x f)
      (if f
          (f 7)
          (foo 5 (lambda (y) (+ x y))))))

> (foo 3 #f)
```

____

Problem 4 (List mutation).

Write `merge!`, a procedure that takes two arguments, each of which is a list of numbers in increasing order. It returns a combined, ordered list of all the numbers:

```
> (merge! (list 3 5 22 26) (list 2 7 10 30))
(2 3 5 7 10 22 26 30)
```

Your procedure must do its work by mutation, changing the pointers between pairs to create the new combined list. The original lists will no longer exist after your procedure is finished.

Note: **Do not allocate any new pairs** in your solution. Rearrange the existing pairs.

Problem 5 (Vectors).

Write a program `rotate!` that rotates the elements of a vector by one position. The function should alter the existing vector, not create a new one. It should return the vector. For example:

```
> (define v (make-vector 4))
> (vector-set! v 0 'a)
okay
> (vector-set! v 1 'b)
okay
> (vector-set! v 2 'c)
okay
> (vector-set! v 3 'd)
okay
> v
#(a b c d)
> (rotate! v)
#(d a b c)
```

Problem 6 (Concurrency).

(a) Suppose we say

```
> (define baz 10)
> (define s (make-serializer))

> (parallel-execute (s (lambda () (set! baz (/ baz 2))))
                    (s (lambda () (set! baz (+ baz baz)))))
```

What are the possible values of `baz` after this finishes?

(b) Now suppose that we change the example to leave out the serializer, as follows:

```
> (define baz 10)

> (parallel-execute (lambda () (set! baz (/ baz 2)))
                    (lambda () (set! baz (+ baz baz))))
```

What are *all* of the possible values of `baz` this time?

Problem 7 (Streams).

We want to change the stream abstraction so that an ordinary list can be used as a stream. That is, we want the selectors `stream-car` and `stream-cdr` to accept either an ordinary list or a stream as argument. Write the new versions of `stream-car` and `stream-cdr`.