CS 61A, Spring 2003, Midterm 3, Harvey

Question 0 (1 point): Fill out this front page correctly and put your name and login correctly at the top of each of the following pages.

Question 1 (6 points):
We want to use object-oriented programming to simulate the effects of traffic in Berkeley. So far, we've designed an `automobile` class. This class has two instantiation variables: `fuel-efficiency` (in miles per gallon) and `fuel-capacity` (in gallons). Automobiles also have two methods. The first, `(drive m)`, causes the automobile to drive for m miles, or until it runs out of gas, whichever comes first. The second `(fill-er-up)`, gives the automobile a full tank of gas. We'd like to add some more features to our simulation.

For each of the features below, identify the most appropriate way to add the feature to the simulation. **Also, give a one-sentence explanation of why you chose that answer.** Each feature may be one of the following:

- Child class (using the `parent` clause)
- Class variable
- Default method
- Initialize method/clause
- Instance variable
- Instantiation variable
- Method

`color`: Lets us know what color the automobile is.

`pollution`: Tells us how much pollution has been produced by all the automobiles of Berkeley, measured in smogallons. (One smogallon of pollution is produced whenever an automobile uses one gallon of gas).

`miles-remaining`: Tells us how far this automobile can go on its current fuel.

`old-automobile`: Like a regular automobile, but produces two smogallons of pollution per gallon of fuel rather than one.

`starting-fuel`: New automobiles contain a random amount of fuel (between 0 and their capacity).

Starting `miles-remaining`: New automobiles also announce their `miles-remaining`.

Question 2 (4 points):

The following expressions are typed, in sequence, at the Scheme prompt. <u>Circle `#t` or `#f`</u> 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
```

Question 3 (8 points):

This week is the ASUC election. We're going to simulate the election in the adventure game by adding two new classes, `voting-place` and `student`.

A `voting-place` is just like a regular place, but when a `student` enters one for the first time, s/he is asked to vote. The voting places must maintain a list of candidates (the same list for every polling place) to present to the student, and keep a vote count for each of the candidates. (Hint: Use a table to hold the vote counts).

At the beginning of the election, we send any voting place the message `start-election` with a list of candidates as argument:

```
> (define Sproul-Plaza (instantiate voting-place 'Sproul-Plaza))
> (ask sproul-plaza 'start-election '(Frankenstein Hoku Primm))
;; (Names are in alphabetical order - no endorsement implied!)
```

This message should also initialize the vote counts to zero, and remember that no students have voted yet.

A `student` is just like a person, but with a `vote` method that takes a list of candidates as argument, and returns one of the candidates. (You can choose the candidate randomly or however you like).

When a `student` who hasn't voted yet enters a voting place, the voting place should send the student a `vote` message with the list of candidates as argument. The return value will be one of the candidates, and the voting place should increase that candidate's vote count by one, and remember that this student has voted, so s/he won't be asked to vote again.

(Any object should accept a `student?` message, which should return true if the object is a student and false otherwise).

Note: To simplify the problem, we are *not* asking for some features that would be necessary for a full simulation of elections, such as a method to find out who won!

(a) Create the `student` class.

Continued on next page.

Question 3 continued:

(b) Now create the `voting-place` class.

Question 4 (8 points):

Fill in the blanks with the response to the indicated expressions. The answers are 11, 121, 1001, and 1111 but not necessarily in that order!

(Hint: You shouldn't have to draw environment diagrams to figure this out. In each case, ask yourself: Is `A` a class variable or an instance variable? Is `B` a class variable or an instance variable?)

```
(define make-foo1
  (let ((a 1))
    (lambda ()
      (let ((b 1))
        (lambda ()
          (set! a (* a 10))
          (set! b (+ b a))
           b)))))

(define foo1-1 (make-foo1))
(define foo1-2 (make-foo1))
(foo1-1)
(foo1-1)
(foo1-2)  ==>  ______________
```

```
(define make-foo2
  (let ((b 1))
    (lambda ()
      (let ((a 1))
        (lambda ()
          (set! a (* a 10))
          (set! b (+ b a))
           b)))))

(define foo2-1 (make-foo2))
(define foo2-2 (make-foo2))
(foo2-1)
(foo2-1)
(foo2-2)  ==>  ______________
```

```
(define make-foo3
  (let ((a 1)
        (b 1))
    (lambda ()
      (lambda ()
        (set! a (* a 10))
        (set! b (+ b a))
         b))))

(define foo3-1 (make-foo3))
(define foo3-2 (make-foo3))
(foo3-1)
(foo3-1)
(foo3-2)  ==>  ______________
```

```
(define make-foo4
  (lambda ()
    (let ((a 1)
          (b 1))
      (lambda ()
        (set! a (* a 10))
        (set! b (+ b a))
         b))))

(define foo4-1 (make-foo4))
(define foo4-2 (make-foo4))
(foo4-1)
(foo4-1)
(foo4-2)  ==>  ______________
```

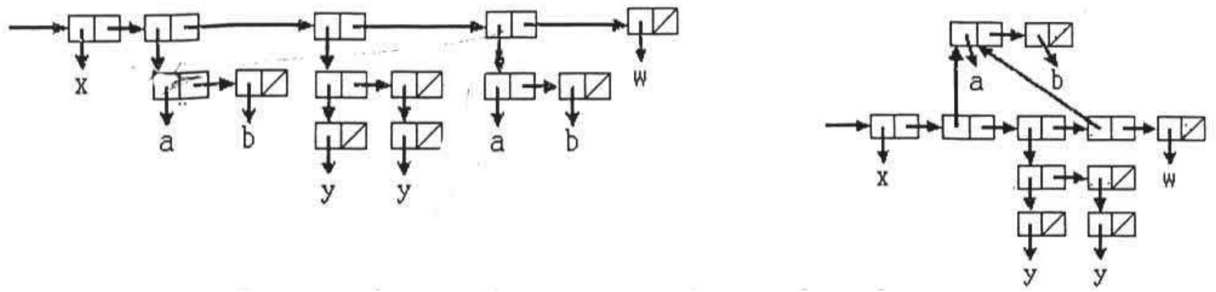
Question 5 (7 points):

We would like to save memory in our data structures by ensuring that there are no duplicated top-level elements (that is, elements that are equal but not identical) in a list. For example, if we have the list

```
(x (a b) ((y) (y)) (a b) w)
```

then we'd like to modify the list so that the two copies of `(a b)` are not only `equal?` But also `eq?`. But we don't care about the two copies of `(y)`; they're not top-level elements.

Before and after pictures:



Complete the following definition of `make-eq!` so that it takes a list as its argument, and turns duplicate sublists into identical ones by mutation. It should return the modified list. The argument list after the procedure call should be `equal?` to the list before the procedure call, but possibly not `eq?`. **Do not allocate any new pairs!** Don't eliminate duplicated elements of elements, just top-level ones.

Note: `(member 'c '(a b c d))` returns `(c d)`, not `#t`.

```
(define (make-eq! lst)
  (if (not (pair lst))
      lst
      (let ((dup (member (car lst) (make-eq! (cdr lst)))))
```