Question 1 (3 points):

What will the Scheme interpreter print in response to each of the following expressions? Also, draw a "box and pointer" diagram for the result of each expression. Hint: It'll be a lot easier if you draw the box and pointer diagram first!

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

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

(let ((x (list 1 (list 2 3) 4)))
  (set-car! (cadr x) (car x)))
  x)

Question 2 (4 points):

Define an object class called password-protect. The purpose of the class is to allow an object to be "hidden" so that a password is needed to send it messages. Here's how it works. Suppose we have this class definition:

(define-class (counter)
  (instance-vars (count 0))
  (method (next)
    (set! count (+ count 1))
    count))

In order to make a password-protected counter, we want to be able to do this:

> (define ppc (instantiate password-protect
  (instantiate counter) 'exotic))

PPC
> (ask ppc 'next)
ERROR: Password incorrect
> (ask (ask ppc 'exotic) 'next)
1
> (ask (ask ppc 'exotic) 'next)
2

In this example, exotic is the password for the protected counter. When sent this password as a message, the object ppc returns the underlying counter object, which can then be sent its own messages.
Question 3 (4 points):

Write \texttt{deep-map!}, a procedure that takes an arbitrary list structure, applies a given function to each leaf, and \textit{modifies the argument list} to replace each leaf with the value returned by the function. For example:

\begin{verbatim}
> (define x (list (list 3 4) 5 (list) (list (list 6))))
x
> x
((3 4) 5 () ((6)))
> (deep-map! square x)
((9 16) 25 () ((36)))
> x
((9 16) 25 () ((36)))
\end{verbatim}

For the purposes of this problem, a ``leaf'' is anything that isn't a pair or the empty list.

\textbf{Do not allocate any new pairs in your solution!} Modify the existing list structure. (You are not, of course, responsible for any pairs that might be allocated by the function you are given as argument, like \texttt{square} in the example above.)

Question 4 (4 points):

Define a stream named \texttt{oz} containing all possible lists whose elements are ones and zeros, like this:

\begin{verbatim}
> (show-stream oz 40)
(() (0) (1) (0 0) (1 0) (0 1) (1 1) (0 0 0) (1 0 0) (0 1 0) (1 1 0) (0 0 1) (1 0 1) (0 1 1) (1 1 1) (0 0 0 0) (1 0 0 0) (0 1 0 0) (1 1 0 0) (0 0 1 0) (1 0 1 0) (0 1 1 0) (1 1 1 0) (0 0 0 1) (1 0 0 1) (0 1 0 1) (1 1 0 1) (0 0 1 1) (1 0 1 1) (0 1 1 1) (1 1 1 1) (0 0 0 0 0) (1 0 0 0 0) (0 1 0 0 0) (1 1 0 0 0) (0 0 1 0 0) (1 0 1 0 0) (0 1 1 0 0) (1 1 1 0 0) (0 0 0 1 0) (0 0 0 1 1) (0 0 1 0 1) (0 0 1 1 0) (0 0 1 1 1) (0 0 1 0 0 0) (1 0 0 0 0 0) (0 1 0 0 0 0) (1 1 0 0 0 0) (0 0 1 0 0 0) (1 0 1 0 0 0) (0 1 1 0 0 0) (1 1 1 0 0 0) (0 0 0 1 0 0) (0 0 0 1 1 0) (0 0 1 0 1 0) (0 0 1 1 0 0) (0 0 1 1 1 0) (0 0 1 0 0 1) (0 0 1 1 0 1) (0 0 1 1 1 1) (0 0 1 1 1 1 0) (0 0 1 1 1 1 1) (0 0 1 1 1 1 1 0) (0 0 1 1 1 1 1 1) (0 0 1 1 1 1 1 1 0) (0 0 1 1 1 1 1 1 1) ...)
\end{verbatim}

Your solution need not have the elements in the same order as this example.

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

Group question (4 points):

Draw the environment diagram for the situation after the following definition and invocation have been evaluated:

\begin{verbatim}
(define maximizer
  (let ((value 0))
    (lambda (arg)
      (if (> arg value)
        (set! value arg))
      value)))

(define foo (maximizer 6))
\end{verbatim}

Take a peek at the solutions