Question 1 (6 points):

What will Scheme print in response to the following expressions? If an expression produces an error message, you may just write "error"; you don't have to provide the exact text of the message. Also, draw a box and pointer diagram for the value produced by each expression.

```
(map list '(1 2 3))
```

```
(let     ((x '(1 2))
          (y '(8 9)))
  (cons x (append y x)))
```

```
(cons (cons 1 2) (append '(18 3) '()))
```

Question 2 (8 points):

Suppose we want to represent books using OOP. We have a `book` class and a `book-store` class. For each of the following, state whether it should be an instance, child class, instance variable, instantiation variable, or class variable; and state the associated class (`book` or `bookstore`). Each may be used any number of times.

SICP ______________________________ ____________

novel ______________________________ ____________

title ______________________________ ____________

ASUC Bookstore ______________________________ ____________

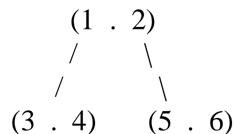
inventory of books ______________________________ ____________

Question 3 (5 points):

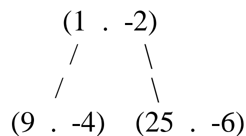
In this problem we are interested in Trees (datum/children) in which each datum is a pair. We'll call this a "pairTree."

We want to write a procedure `pairtree-map` that takes *three* inputs: a function to apply to the `car` of each datum, a function to apply to the `cdr` of each datum, and a pairTree. It should return a pairTree with the same shape as the argument pairTree, but in which each datum is replaced with a pair containing the results of calling the two functions on the two halves of each datum.

So if `mytree` is the pairTree



The the result of `(pairtree-map square – mytree)` is



Find and correct all data abstraction violations.

```
(define (pairtree-map car-fn cdr-fn tree)

        (let ((this (car tree)))

            (make-tree (make-tree (car-fn (datum this)) (cdr-fn (cdr this)))

                           (pair-forest-map car-fn cdr-fn (children tree)))))

(define (pair-forest-map car-fn cdr-fn forest)

        (if (null? Forest)

           ’()

        (make-tree (pairtree-map car-fn cdr-fn (datum foest))

                      (pair-forest-map car-fn cdr-fn (cdr forest)))))
```

Question 4 (4 points):

Suppose we type this into Scheme-1:

```
((lambda (x y) (lambda (z) (z x y))) 5 7)
```

(a) What is the result?

(b) Throughout the process of getting the above result, how many calls to `eval-1` are made in which the argument expression is

a number? ________________

a special form? ________________

an application of a primitive procedure? ________________

an application of a non-primitive procedure? ________________

Question 5 (8 points):

This question deals with the Mobile and Branch ADT from exercise 2.29.

Recall:

* a Mobile has two Branches.

* a Branch consists of a length and a structure, which is either a number (the weight) or another Mobile.

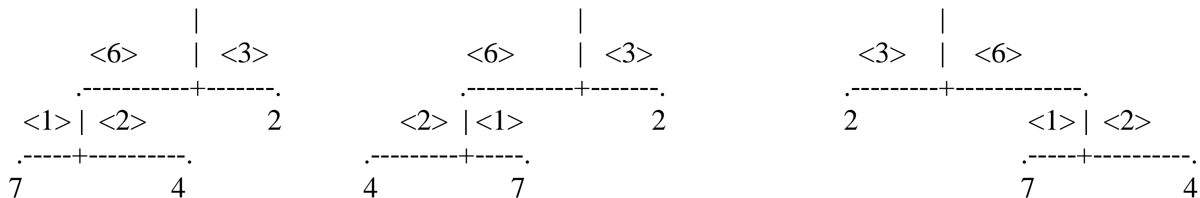
Constructors and Selectors:

```
(make-mobile left right)

(left-branch M)          ; you may abbreviate this as LB

(right-branch M)     ; you may abbreviate this as RB

(make-branch length structure)

(branch-length B)    ; you may abbreviate this as BRL

(branch-structure B) ; you may abbreviate this as BRS
```

If you hang a mobile, the weights can rotate freely, so that the same mobile might have “left” and “right” reversed at a different time. For example these are the same mobile:

(where numbers such as <6> are lengths, and plain numbers such as 2 are weights).

Similarly all these are the same:



Define a procedure called `same-structure?` That takes two mobiles as arguments, and returns `#t` if and only if the two are the same, possibly including rotations anywhere in the structure.

Question 6 (8 points):

Write a procedure `three-branching?` That takes a list as arugment. It should return `#t` if and only if the list and every list that appears as an element, or an element of an element, etc., has three elements. For example:

```
(three-branching? '(1 2 3))                  => #t

(three-branching? '((1 2 3) 2 3))            => #t

(three-branching? '((1 2 3) (4 5 6)))        => #f

(three-branching? '(1 2) (3 4) 5)            => #f

(three-branching? '())                       => #f
```