CS 61A, Spring 2007, Midterm 2, Harvey

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 car '((cons 1 2) (list 3 4)))
```

```
(let (( x (list 1 2))
      (y  (list 7 8)))
   (cons x (list y x)))
```

```
(list (append '(a b) '(c))  (cons '(a b) '(c)))
```

Question 2 (5 points):

We're going to represent a date as simply a list of three numbers

```
(define (make-date year month day)
  (list year month day))
```

(a) Write selectors for year, month, and date. Given a date, they should return the appropriate value.

```
(define (year date)

   ______________________________)

(define (month date)

   ______________________________)

(define (day date)

   ______________________________)
```

(b) Now, we're going to use those dates to represent events. An event has a start date, an end date, and a description in the form of a sentence.

Given the following selectors for an event, write a constructor for an event.

```
(define (start-date event)
  (caar event))

(define (end-date event)
  (cdar event))

(define (description event)
  (cdr event))



(define (make-event start-date end-date description)
```

Question 3 (6 points):

The procedure below takes as argument a **Tree** (with datum and children selectors) in which the data are **sentences**. It returns a Tree of the same shape, in which the datum at each node is the second word of the sentence at the corresponding node of the original Tree.

The code has some data abstraction violations in it. Your task is to find and fix them.

```
(define (seconds tree)


 (make-tree (cadr (first tree))


              (f-seconds (bf tree))))




(define (f-seconds forest)


 (if (null? forest)

    '()


    (make-tree (seconds (car forest))


                   (f-seconds (cdr forest)))))
```

Question 4 (8 points):

This question concerns the Tree abstract data type (with datum and children) discussed in lecture.

Write a procedure levelsum that takes a Tree of numbers and a depth as arguments, and returns the sum of the data at that level of the tree. The root is at depth 0.

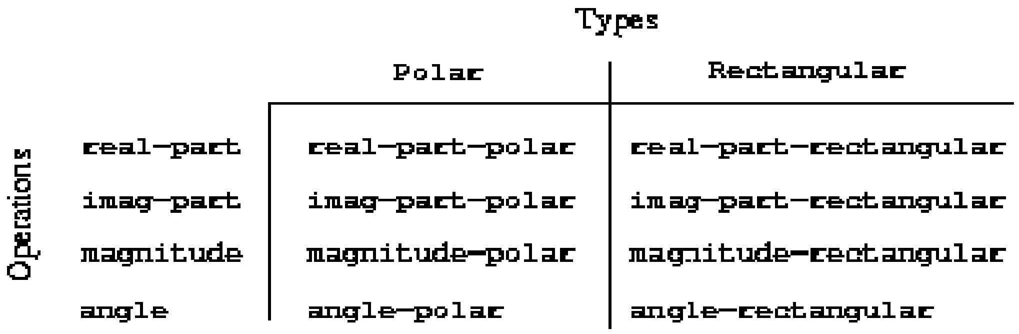
For example, suppose mytree is this tree:

```
          3
         / \
       2     4
      / \   / \
     1   0 5   7
    /
   9
```

```
> (levelsum 0 mytree)
3
> (levelsum 1 mytree)
6
> (levelsum 2 mytree)
13
> (levelsum 3 mytree)
9
> (levelsum 4 mytree)
0
```

Question 5 (6 points):

Ben Bitdiddle has an idea to simplify the implementation of data-directed programming. He has been looking at the table of operators for complex arithmetic, reproduced below from page 181 of the text:



Operations	Polar	Rectangular
real-part	real-part-polar	real-part-rectangular
imag-part	imag-part-polar	imag-part-rectangular
magnitude	magnitude-polar	magnitude-rectangular
angle	angle-polar	angle-rectangular

Ben notices a pattern in the eight procedure names that make up the table entries. Each name is formed by combining the name of an operator with the name of a type. Ben says, "Why bother having a table at all? The general operate procedure already has those two names as arguments. Instead of looking them up in a table with get, we can just use the word procedure to combine them, thereby forming the name of the procedure we need. To add a new type or a new operator, we don't have to use the put procedure; we need only write the correctly-named procedures that implement the new algorithms. For example, if we want to be able to find the conjugate of a complex number [never mind if you don't know what that is!] all we do is write procedures named conjugate-polar and conjugate-rectangular."

Ben's proposed implementation is to make one change to the operate procedure. Here is the version in the lecture notes (reader p. 285):

```
(define (operate op obj)
  (let (( proc (get (type obj) op)))
    (if (not (null? proc))
          (proc (contents obj))
          (error "Operator undefined for this type -- OPERATE"
               (list op obj)))))
```

Ben wants to change the let so that it begins

```
(let (( proc (word op '- (type obj))))
```

(This question continues on the next page.)

(Question 5 continued):

(a) Will this work? If so, give an example. If not, why not?

(b) Suppose it will work, or can be made to work. Is it an improvement? Are there circumstances in which the book’s technique is easier to use? Explain your answer.

Question 6 (8 points):

Write a procedure diffs that takes as arguments two **deep lists** with the same shape, but possibly different atomic deep elements, and returns a list of pairs containing the different atomic elements (the order of the list doesn't matter). For example,

```
STk> (diffs '(a (b ((c)) d e) f) '(1 (b ((2)) 3 4) f))
((a . 1) (c . 2) (d . 3) (e . 4))

STk> (diffs '() '())
()

STk> (diffs '((a b) c) '((a b) c))
()
```