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 caddr '((2 3 5) (7 11 13) (17 19)))

(list (cons 2 (cons 3 5)))

(append (list '(2) '(3)) (cons '(4) '(5)))
```

Question 2 (7 points)

Suppose you are in a team working on a social networking program. You are given the constructor of a "person" data structure.

```
(define (make-person first-name last-name favorite-sport
favorite-movie)
  (list (list first-name last-name)
        favorite-sport
        (cons 'movie-favorite-movie)))
```

a) Write the selectors:

```
  (define (first-name person))

  (define (last-name person))

  (define (favorite-movie person))

  (define (favorite-sport person))
```

b) Your partner wrote a procedure find-partners that takes a person p and a list of persons lst as arguments, and returns the persons in the list that have the same favorite sport as p. Fix all the data abstraction violations in his code.

```
  (define (find-partners p lst)
          (cond ((null? lst) '())
                ((equal? (cadr p) (cadr (first lst)))
                 (cons (first lst) (find-partners p (butfirst
lst))))
                (else (find-partners p (butfirst lst)))))
```

Question 3 (5 points):
For reference, here are the central procedures of scheme-l, with the lines numbered:

```
1 (define (eval-1 exp)
2 (cond ((constant? exp) exp)
3 ((symbol? exp) (eval exp))
4 ((quote-exp? exp) (cadr exp))
5 ((if-exp? exp)
6 (if (eval-1 (cadr exp))
7 (eval-1 (caddr exp))
8 (eval-1 (cadddr exp))))
9 ((lambda-exp? exp) exp)
10 ((pair? exp) (apply-1 (eval-1 (car exp))
11 (map eval-1 (cdr exp))))
12 (else (error ''bad expr: '' exp))))
13 (define (apply-) proc args)
14 (cond ((procedure? proc)
15 (apply proc arga))
16 ((lambda-exp? proc)
17 (eval-1 (substitute (caddr proc)
18 (cadr proc)
19 args
20 '()))))
21 (else (error ''bad proc: '' proc))))
```

A student tries to type this into his computer, but makes one mistake. Here is a transcript of some of his test cases:

```
Scheme-1 : (+ 2 3)
5

Scheme-1: (+ (* 2 2) 3)
ERROR

Scheme-1 : ((lambda (x) (* x x)) 2)
4

Scheme-l: (lambda (x) (+ x x)) (+ 1 1))
ERROR

Scheme-1 : (if #t 2 3)
2

Scheme-) : (if (> 3 0) (+ 2 1) (+ 3 1))
3
```

Based on the test cases, what is wrong with his version of scheme-1? Indicate the line number with the problem, and what the student typed on that line.

Question 4 (8 points):

Write deep-depths. It takes a deep list as its argument, and returns a list of the same shape, but with every atomic element replaced with its depth in sublists, as in these examples:

```
> (deep-depths '(a b c))
(0 0 0)
> (deep-depths '(a (b c) d))
(0 (1 1) 0)
> (deep-depths '((((a)))))
((((3))))
```

6

Question 5 (5 points):
You and your friend Timmy want to be able to carry on conversations that your parents can't understand. Your aunt Evelyn teaches you Pig Latin, but after some disastrous failures to keep conversations secret you realize that your parents know how to speak Pig Latin, too. So you decide to invent Super Pig Latin, with even more rules for more categories of letters.

You realize that it's going to take some experimentation to invent rules that are complicated enough to confuse your parents, but simple enough for you and Timmy to speak and understand. So you decide to use data directed programming.

For every letter of the alphabet you create two table entries:

1. An operation called next that provides the argument to the next call to superpigl, if this letter is the first letter of the argument word.
2. A true/false value called done that indicates whether the value returned by next is the final translation into Super Pig Latin, so no more calls to superpigl are needed.

For example, here's how you'd set up the rules for Pig Latin;

```
(put 'a 'next (lambda (wd) (word wd 'ay)))
(put 'a 'done #t) ; same for other vowels
(put 'b 'next (lambda (wd) (word (butfirst wd) (first wd))))
(put 'b 'done #f) ; same for other consonants
```

If your Super Pig Latin rules are "just like Pig Latin, but leave out every D before the first vowel, and if the first vowel is A, change it to E,'' then you'd have the following two exceptional entries:

```
(put 'd 'next (lambda (wd) (butfirst wd)))
(put 'a 'next (lambda (wd) (word 'e (butfirst wd) 'ay)))
```

Fill in the blanks in superpigl:

```
(define (superpigl wd)
(if ______________________________
    ______________________________
    ______________________________))
```

Question 6 (8 points):
Write depth-of-datum. It takes in a tree and an datum, and returns the depth of the datum in the tree. You can assume that the datum is in the tree at most once. If the datum is not in the tree, return #f.

For example, if mytree is the tree

```
                   A
                  / \
                 /   \
                B     C
               / \   / \
              D   E F   G
             /         / \
            H         I   J
                     / \   \
                    K   L   M
```

then

```
> (depth-of-datum mytree 'a)
0
> (depth-of-datum mytree 'j)
3
> (depth-of-datum mytree 'z)
#f
```