Problem 1. Remove-letter

Consider a procedure `remove-letter` that takes two inputs, a letter and a sentence, and returns the sentence with all occurrences of the letter removed. For example:

`(remove-letter 'e '(here is a sentence with e in it)`	➔	`(hr is a sntnc with "" in it)`
`(remove-letter 'e '(not any within))`	➔	`(not any within)`
`(remove-letter 'e '())`	➔	`()`

Part A: Write `remove-letter` <u>without using any explicit recursion</u> (i.e., use higher order functions instead)

```
;; solution 1:
(define (remove-letter ltr sent)
   (every (lambda (wd)
                (remove-letter-from-word ltr wd))
          sent))
(define (remove-letter-from-word ltr wd)
   (keep (lambda (ltr-from-wd)
                (not (equal? ltr ltr-from-wd))
          wd))

;;solution 2:
(define (remove-letter ltr sent)
   (every (lambda (wd)
                (keep (lambda (ltr-from-wd)
                         (not (equal? ltr ltr-from-wd))
                      wd))
          sent))

;;solution 3:
(define (remove-letter-from-word ltr wd)
   (accumulate word
                 (every (lambda (ltr-from-wd)
                           (if (equal? ltr ltr-from-wd)
                               ""
                               ltr-from-wd))
                     wd))
```

Part B: Write `remove-letter` <u>without using higher-order functions</u> (i.e., use recursion instead).

```
;; solution 1:
 (define (remove-char char sent-or-word)
   (cond ((empty? sent-or-word) sent-or-word)
         ((sentence? sent-or-word)
          (se (remove-char (first sent-or-word))
              (remove-char (bf sent-or-word))))
         ((equal? (first sent-or-word) char)
          (remove-char (bf sent-or-word)))
         (else (word (first sent-or-word)
                     (remove-char (bf sent-or-word))))))

 ;; solution 2:
 (define (remove-char char sent)
   (if (empty? sent)
       sent
       (se (remove-char-from-word (first sent))
           (remove-char (bf sent)))))

 (define (remove-char-from-word char wd)
   (cond ((empty? wd) wd)
         ((equal? (first wd) char)
          (remove-char-from-wd (bf wd)))
         (else (word (first wd)
                     (remove-char-from-word (bf wd))))))
```

Problem 2. Not just a ticky-tack question

In tic-tac-toe, a pivot is an open square that identifies a winning move through the generation of a fork. In `ttt.scm`, the pivot procedure takes a sentence of triples and a player, and returns a sentence of pivots. The code in `ttt.scm` is reproduced in an appendix at the end of this exam.

For the board `b` equal to "`x o _ _ x _ _ _ o`", for example:

X	O	
	X	
		O

`(pivots (find-triples b) 'x)`	➔	`(4 7)`
`(pivots (find-triples b) 'o)`	➔	`()`

Rewrite `pivots` without using higher order procedures (i.e., using only recursion). You can use procedures defined in `ttt.scm` <u>as long as those procedures don't use higher order functions</u>. (You may use `appearances`).

Make sure to name your helper procedures and parameters well. You only need to comment when you think it necessary to help explain the intent of your procedure.

Here are some procedures you can use *without* writing them:

`keep-my-singles` takes a sentence of triples and a player and returns a sentence of triples that satisfy `my-single?` (that is, triples with two empty squares and one square filled by the player):

`(keep-my-singles (find-triples b) 'x)`	➔	`("4x6" x47 "3x7")`
`(keep-my-singles (find-triples b) 'o)`	➔	`("78o" "36o")`

`explode-all` takes a sentence of words and returns a sentence with each word "exploded" into single-letter words:

`(explode-all '(bob joe))`	➔	`(b o b j o e)`
`(explode-all '(25o 7o9))`	➔	`(2 5 o 7 o 9)`

Note: simply duplicating the algorithm in the book will get you into trouble. Remember, chapter 10 in Simply Scheme comes before recursion, and would have changed quite a bit had recursion been used.

There are a few different solutions, and most involved a helper procedure within which to do the recursion. The tail-recursive solution below does this, recursing down the list returned by `explode-all`.

```
(define (pivots triples me)
   (pivots-helper (explode-all (keep-my-singles triples me))
                  '()))

(define (pivots-helper squares current-pivots)
  (cond ((empty? squares) current-pivots)
        ((or (not (number? (first squares)))
             (not (member? (first squares) (bf squares)))
             (member? (first squares) current-pivots))
          (pivots-helper (bf squares) current-pivots))
        (else (pivots-helper (bf squares)
                             (se (current-pivots)
                                 (first squares))))))
```

Note that there are two recursive cases: when a pivot is found or when one isn't. To find a pivot involves checking to see that the current square is a number (to ignore the "x"s and "o"s that will be in the sentence), checking that the current square appears again later in the sentence, and checking that we haven't already found this square. The last check isn't too important – technically there is one rare case where `pivots-helper` would see the same square three times, but it wouldn't affect the rest of the program. With the embedded version of this code, it was very hard to check for this third case!

Problem 3. The card game "clubs"

This question concerns a game called clubs. In this game, card are worth a number of points: 1 point for every club, 13 points for the queen of spades, and 0 points otherwise. A *card* is a list of the rank and suit of the card. Ranks are the number or the letter `a, j, q, or k`. Suits are one of the letter `c, s, d, h` (for clubs, spades, diamonds, and hearts).

A player has a *hand* of up to 5 cards, and the number of points in the hand is the sum of the points for each card. A hand is represented by a list with the name of the player followed by each of the cards in the hand. `(amy (a d) (3 d) (6 h) (q s))`, `(jack (2 c))`, and `(fred)` are all proper hands. These hands are worth 13, 1, and 0 points respectively.

A *game-state* is defined as a list of hands. It represents the state of the game at one particular time.

Part A: Write the proper selectors to get the rank and suit of a card, and the name and cards of a hand.

```
(define (rank card) (car card))
(define (suit card) (cadr card))

(define (name hand) (car hand))
(define (cards hand) (cdr hand))
```

Part B: Write the procedure `totals` which takes a game-state and returns a table of player names paired with the total points of their hand. For instance,

`(totals '((sam (a c) (2 c) (3 c) (4 c))` `(bob (a h) (2 h) (3 h) (4 h))))`	➔	`((sam 4)` `(bob 0))`

`(totals '((amy (a d) (3 d) (6 h) (q s))`
`(jack (2 c))`
`(fred)))`
	➔	`((amy 13)`
		`(jack 1)`
		`(fred 0))`

```
(define (worth card)
   (cond ((and (equal? (suit card) 's)
               (equal? (rank card) 'q))
          13)
         ((equal? (suit card) 'c)
          1)
         (else 0)))

(define (score hand)
   (apply + (map worth (cards hand))))

(define (totals game-state)
   (map (lambda (hand)
           (list (name hand)
                 (score hand)))
        game-state))
```

Part C: Write a procedure current-score which takes a game state and a player name and returns the current score for that players hand. Don't write any additional procedures; assume the procedures for parts A and B are functioning correctly.

```
(define (current-score gs name)
   (cadr (assoc name (totals gs))))
```

Problem 4. Predicates and generalized lists: `Deep-any?`

Part A: Write a procedure called `deep-any?`, which takes a one-argument predicate and a generalized list, and returns `#t` if any word in within that list or sublist satisfies the predicate.

`(deep-any? even? '( 5 ((3) ((2))) 11 ))`	➔	`#t`
`(deep-any? even? '( 5 ((3) ((7))) 11 ))`	➔	`#f`

```
(define (deep-any? pred L)
   (cond ((null? L)
          #f)
         ((list? (car L))
          (or (deep-any? pred (car L))
              (deep-any? pred (cdr L))))
         (else
          (or (pred (car L))
              (deep-any? pred (cdr L))))
         ))
```

(there are plenty of other ways, however...)

Part B: Fill in the blank in the following procedure so that given a generalized list, the procedure will return #t if there are any numbers greater than 20 in the list. Note that the list may contain anything (not necessarily numbers). Don't define any other procedures.

```
(define (deep-any-numbers-greater-than-20? g-list)
   (deep-any?

         _______(lambda (e) (and (number? e) (> e 20)))________

         g-list))
```

Problem 5. Election processing with lists

Write a higher-order procedure named `electoral-votes` which takes a predicate as its single argument. The procedure will sum up the 2008 electoral votes for states that satisfy the predicate.

`(electoral-votes california?)` ➔ `55`

`(electoral-votes blue-state?)` ➔ `212`

The database of states and their electoral votes is in a global variable `*states*`:

```
((ca 55) (me 4) (nj 15) …)
```

The predicate takes the state's <u>two-letter abbreviated name</u> as its argument. You do not have to write these predicates; rather, you only need to write `electoral-votes` such that it works properly with any proper predicate. Do not use explicit recursion.

```
(define (electoral-votes pred)
   (apply +

      (map state-ev

         (filter (lambda (state)
                       (pred (state-name state)))
                 *states*) ) ) )
```

or, without using `filter` first

```
(define (electoral-votes pred)
   (apply +

      (map (lambda (state)

              (if (pred (state-name state))

                  (state-ev state)

                  0))

           *states*)))
```