Spring 2008 final review (in lecture)

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 '())`	➔	`()`

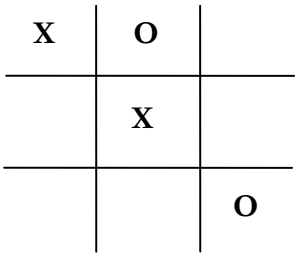
Part A: Write `remove-letter` without using any explicit recursion (i.e., use higher order functions instead)

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

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:



`(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` as long as those procedures don't use higher order functions. (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)`

```
(define (ttt position me)
  (ttt-choose (find-triples position) me))

(define (find-triples position)
  (every (lambda (comb) (substitute-triple
comb position))
         '(123 456 789 147 258 369 159
357)))

(define (substitute-triple combination
position)
  (accumulate word
              (every (lambda (square)
                       (substitute-letter
square position))
                     combination) ))

(define (substitute-letter square position)
  (if (equal? '_ (item square position))
      square
      (item square position) ))

(define (ttt-choose triples me)
  (cond ((i-can-win? triples me))
        ((opponent-can-win? triples me))
        ((i-can-fork? triples me))
        ((i-can-advance? triples me))
        (else (best-free-square triples))
))

(define (i-can-win? triples me)
  (choose-win
   (keep (lambda (triple) (my-pair? triple
me))
         triples)))

(define (my-pair? triple me)
  (and (= (appearances me triple) 2)
       (= (appearances (opponent me)
triple) 0)))

(define (opponent letter)
  (if (equal? letter 'x) 'o 'x))

(define (choose-win winning-triples)
  (if (empty? winning-triples)
      #f
      (keep number? (first winning-
triples)) ))

(define (opponent-can-win? triples me)
  (i-can-win? triples (opponent me)) )

(define (i-can-fork? triples me)
  (first-if-any (pivots triples me)) )

(define (first-if-any sent)
  (if (empty? sent)
      #f
      (first sent) ))

(define (pivots triples me)
  (repeated-numbers (keep (lambda (triple)
(my-single? triple me))
                          triples)))

(define (my-single? triple me)
  (and (= (appearances me triple) 1)
       (= (appearances (opponent me)
triple) 0)))

(define (repeated-numbers sent)
  (every first
         (keep (lambda (wd) (>= (count wd)
2))
               (sort-digits (accumulate
word sent)) )))

(define (sort-digits number-word)
  (every (lambda (digit) (extract-digit
digit number-word))
         '(1 2 3 4 5 6 7 8 9) ))

(define (extract-digit desired-digit wd)
  (keep (lambda (wd-digit) (equal? wd-digit
desired-digit)) wd))

(define (i-can-advance? triples me)
  (best-move (keep (lambda (triple) (my-
single? triple me)) triples)
             triples
             me))

(define (best-move my-triples all-triples
me)
  (if (empty? my-triples)
      #f
      (best-square (first my-triples) all-
triples me) ))

(define (best-square my-triple triples me)
  (best-square-helper (pivots triples
(opponent me))
                      (keep number? my-
triple)))

(define (best-square-helper opponent-pivots
pair)
  (if (member? (first pair) opponent-
pivots)
      (first pair)
      (last pair)))

(define (best-free-square triples)
  (first-choice (accumulate word triples)
                '(5 1 3 7 9 2 4 6 8)))

(define (first-choice possibilities
preferences)
  (first (keep (lambda (square) (member?
square possibilities))
               preferences)))
```

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.

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))`

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.

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`

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?

        _____________________________________________

        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 two-letter abbreviated name 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)
```