Tenth Quiz

You have 15 minutes from the start of class to complete this quiz. Read the questions with care; work with deliberate speed. Don't give us more than we ask for. The usual instructions apply. Good luck!

Problem 1 (10 points)

Along with this quiz is a copy of the Restaurants program (a different version from last week).

(a) (1 point) What data structure does this program use to implement the collection of restaurants?

(b) (2 points) One or more of the collection functions are recursive; list their name(s). [Don't consider the view/controller functions or the restaurant/menu/dish functions.]

(c) (2 points) One or more of the collection functions are *tail*-recursive; list their name(s).

(d) (2 points) Pick one of the collection functions that isn't tail-recursive. Copy below the line(s) of code from that function that make it not tail-recursive.

(e) (2 points) One or more of the collection functions uses the accumulator style; list their name(s).

(f) (1 point) If you ignore the collection functions whose names end in -alternative or -original or -aux, you see that the search, remove, and change functions all depend on collection-process. We have seen examples before of "refactoring," or collecting similar functionality together. Why did the author of this code generalize all these operations into one place; what was he or she trying to avoid? [Two words are enough.]

Problem 2 (10 points)

Suppose we run a multiplex cinema with five screens (i.e., five theaters), numbered 0 through 4. On each screen (in each theater) we have six shows daily, numbered 0 through 5. We can store one day's ticket sales for each show in each theater with a two-dimensional table (a vector of vectors with a row for each screen and a column for each show—we'll call this a "sales table") by calling

```
(define Nov29 (create-table 5 6))
```

using this definition:

```
;; create-table: number(screens) number(shows) -> sales-table
;; Return a two-dimensional table with the specified number of rows and columns, all 0
(define create-table
  (lambda (num-screens num-shows)
    (build-vector num-screens (lambda (n) (build-vector num-shows (lambda (i) 0))))))
```

This function creates a table full of zeroes for one day; we called it above to create the table named `Nov29`. We can retrieve a given sales figure from the table (which will be more interesting once we insert some values—see below), in this case sales for the last show (number 5) in the first theater (on screen number 0), by calling

```
(sales-figure Nov29 0 5)
```

using this definition:

```
;; sales-figure: sales-table number(screen) number(show) -> number
;; Return the value stored for the specified screen and show
(define sales-figure
  (lambda (table screen show)
    (vector-ref (vector-ref table screen) show)))
```

We can record a sale of two tickets for screen number 1's show number 4 by calling

```
(add-sale Nov29 2 1 4)
```

using this definition:

```
;; add-sale: sales-table number(tickets) number(screen) number(show) -> sales-table
;; Return the table with the specified element increased by the number of tickets
(define add-sale
  (lambda (table tickets screen show)
    (vector-set!
      (vector-ref table screen) show (+ (sales-figure table screen show) tickets))))
```

(a) (2 points) Write a Scheme expression to return the number of tickets sold for the last show in theater number 3 (in our `Nov29` table).

(b) (2 points) Write a Scheme expression to record (in the table `Nov29`) a group sale of 15 tickets for the first show in the last theater.

(c) (6 points) Complete the definition of the function below, which handles returned tickets. This is very similar to a definition above.

```
;; return-tickets: sales-table number(tickets) number(screen) number(show) -> sales-table
;; Return the table with the specified element decreased by the amount
(define return-sale
  (lambda (table tickets screen show)
```