CSSE 304 Assignment 4 Updated for Spring, 2016

Objectives: You should learn

- to write more complex recursive procedures in a functional style.

.

This is an individual assignment. You can talk to anyone you want and get as much help as you need, but you should type in the code and do the debugging process, as well as the submission process.

At the beginning of your file, there should be a comment that includes your name and the assignment number. Before the code for each problem, place a comment that includes the problem number. Place the code for the problems in order by problem number.

Turning in this assignment. Write all of the required procedures in one file, and upload it for assignment 4. You should test your procedures offline, using the test code file or other means, **before submitting to the server**.

Unless it is specified otherwise for a particular problem, assume that arguments passed to your procedures have the correct format. If a problem description says that an argument will have a certain type, you may assume that this is true; your code does not have to check for it. **Note:** In the `matrix?` and `multi-set?` problems, there are no assumptions about the argument's format.

Restriction on Mutation continues. As in the previous assignments, you will receive zero credit for a problem if any procedure that you write for that problem uses mutation or calls a procedure that mutates something.

Abbreviations for the textbooks:

EoPL - Essentials of Programming Languages, 3rd Edition
TSPL - The Scheme Programming Language, 4rd Edition (available free scheme.com)
EoPL-1 - Essentials of Programming Languages, 1st Edition
(small 4-up excerpt handed out in class, also on Moodle)

A *set* is a list of items that has no duplicates. In a *multi-set*, duplicates are allowed, and we keep track of how many of each element are present in the multi-set. For problems in this course, we will assume that each element of a multi-set is a symbol. We represent a multi-set as a list of 2-lists. Each 2-list contains a symbol as its first element and a positive integer as its second element. So the multi-set that contains one **a**, three **b**s and two **c**s might be represented by `((b 3) (a 1) (c 2))` or by `((a 1) (c 2) (b 3))` .

#1 (10 points) Write a Scheme procedure `(multi-set? obj)` that returns `#t` if `obj` is a representation of a multi-set, and `#f` otherwise.

multi-set? : *scheme-object* → *Boolean*

Examples:

```
(multi-set? '())              ➔ #t
(multi-set? '(a b))           ➔ #f
(multi-set? '((a 2)))         ➔ #t
(multi-set? '((a 0)))         ➔ #f
(multi-set? '(a b))           ➔ #f
(multi-set? '((a 2) (b 3)))   ➔ #t
(multi-set? '((a 2) (a 3)))   ➔ #f
(multi-set? '((a 3) b))       ➔ #f
```

#2 (6 points) Write a Scheme procedure `(ms-size ms)` that returns the total number of elements in the multi-set `ms`. Can you do this with very short code that uses `map` and `apply`?

ms-size : *multi-set* → *integer*

Examples:

```
(ms-size '())             ➔ 0
(ms-size '((a 2)))        ➔ 2
(ms-size '((a 2)(b 3)))   ➔ 5
```

#3 (3 points) A *matrix* is a rectangular grid of data items. We can represent a matrix in Scheme by a list of lists (the inner lists must all have the same length. For example, we represent the matrix

1	2	3	4	5
4	3	2	1	5
5	4	3	2	1

by the list of lists `((1 2 3 4 5) (4 3 2 1 5) (5 4 3 2 1))`. We say that this matrix has 3 rows and 5 columns or (more concisely) that it is a 3×5 matrix. **A matrix must have at least one row and one column.**

Write a Scheme procedure `(matrix-ref m row col)`, where `m` is a matrix, and `row` and `col` are integers. Like every similar structure in modern programming languages, the index numbers begin with 0 for the first row or column. This procedure returns the value that is in row `row` and column `col` of the matrix `m`. Your code does not have to check for illegal inputs or out-of-bounds issues. You can use `list-ref` in your implementation.

matrix-ref : *Listof*(*Listof*(*Integer*)) × *Integer* × *Integer* → *Integer* (assume that the first argument actually is a matrix)

Examples:
If *m* is the above matrix,

```
(matrix-ref m 0 0) ➔ 1
(matrix-ref m 1 2) ➔ 2
```

#4 (10 points) The predicate `(matrix? obj)` should return `#t` if the Scheme object `obj` is a matrix (a nonempty list of nonempty lists of numbers, with all sublists having the same length), and return `#f` otherwise.

matrix?: *SchemeObject* → *Boolean*

Examples:

```
(matrix? 5)                          ➔ #f
(matrix? "matrix")                   ➔ #f
(matrix? '(1 2 3))                   ➔ #f
(matrix? '((1 2 3)(4 5 6)))          ➔ #t
(matrix? '#((1 2 3)(4 5 6)))         ➔ #f
(matrix? '((1 2 3)(4 5 6)(7 8)))     ➔ #f
(matrix? '((1)))                     ➔ #t
(matrix? '(()()()))                  ➔ #f
```

#5 (10 points) Each row of `(matrix-transpose m)` is a column of *m* and vice-versa.

matrix-transpose: *Listof*(*Listof*(*Integer*)) → *Listof*(*Listof*(*Integer*)) (assume that the argument actually is a matrix)

Examples:

```
(matrix-transpose '((1 2 3) (4 5 6))) ➔ ((1 4) (2 5) (3 6))
(matrix-transpose '((1 2 3)))         ➔ ((1) (2) (3))
(matrix-transpose '((1) (2) (3)))     ➔ ((1 2 3))
```

#6 (3 points) Write a recursive Scheme procedure `(last ls)` which takes a list of elements and returns the last element of that list. This procedure is in some sense the opposite of `car`. You may assume that your procedure will always be applied to a non-empty proper list. You are not allowed to reverse the list or to use `list-tail`. [**Something to think about** (not directly related to doing this problem): Note that `car` is a constant-time operation. What about `last`?]

last: *Listof*(*SchemeObject*) → *SchemeObject*

Examples:

```
(last '(1 5 2 4))  ➔ 4
(last '(c)) ➔ c
```

#7 (5 points) Write a recursive Scheme procedure `(all-but-last lst)` which returns a list containing all of `lst`'s elements but the last one, in their original order. In a sense, this procedure is the opposite of `cdr`. You may assume that the procedure is always applied to a valid argument. You may not reverse the list. You may assume `lst` is a nonempty proper list. [**Something to think about** (not directly related to doing this problem): `cdr` is a constant-time operation. What about `all-but-last`?]

all-but-last: *Listof*(*SchemeObject*) → *Listof*(*SchemeObject*)

Examples:

```
(all-but-last '(1 5 2 4))  ➔ (1 5 2)
(all-but-last '(c)) ➔  ()
```