CSSE 304 Assignment 5 Updated for Spring, 2016

Objectives: You should learn

- To be very confident and competent with functional-style recursive procedures dealing with lists and with lists of lists.

These instructions are identical to those for Assignment 4

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 5. You should test your procedures offline, using the test code file or other means, **before submitting to the server**.

Assume that arguments 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?` problem, 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)

Problem 1 uses the definitions and representations of intervals that are described in assignment 1.

#1 (20 points) `minimize-interval-list` Write a Scheme procedure `(minimize-interval-list ls)` that takes a nonempty list (not necessarily a set) of intervals and returns the set of intervals that has smallest cardinality among all sets of intervals whose unions are the same as the union of the list of intervals `ls`. In other words, combine as many intervals as possible. For example:

minimize-interval-list: *IntervalList* → *IntervalList*

```
(minimize-interval-list '((1 3) (2 3)))                → ((1 3))
(minimize-interval-list '((1 2) (3 4)))                → ((1 2) (3 4))
(minimize-interval-list '((1 3) (8 10) (2 4) (9 11)))  → ((1 4) (8 11))
(minimize-interval-list '((2 5) (1 7) (6 10) (10 11))) → ((1 11))
(minimize-interval-list '((1 2) (4 7) (1 2)))          → ((1 2) (4 7))
```

#2 (5 points) Write the procedure (`exists? pred ls`) that returns `#t` if `pred` applied to any element of `ls` returns `#t`, `#f` otherwise. It should short-circuit; i.e., if it finds an element of `ls` for which `pred` returns `#t`, it should immediately return without looking at the rest of the list

exists?: *predicate* × *relation* → *Boolean*

Examples:

```
(exists? number? '(a b 3 c d)) → #t
(exists? number? '(a b c d e)) → #f
```

#3 (10 points) Write the procedure (`list-index pred ls`) that returns the 0-based position of the first element of `ls` that satisfies the predicate `pred`. If no element of `ls` satisfies `pred`, then `list-index` should return `#f`.

list-index: *predicate* × *List* → *Integer* | *Boolean* (the | means that it can be either type)

Examples:

```
(list-index symbol? '(2 "e" 3 ab 4)) → 3
(list-index pair? '((a b) b c (a a)) → 0
(list-index string? '(a b 3 #\4 2)) → #f
```

#4 (20 points) `pascal-triangle`. If you are not familiar with Pascal's triangle, see this page: http://en.wikipedia.org/wiki/Pascal_triangle . The first recursive formula that appears on that page will be especially helpful for this problem.
Write a Scheme procedure `(pascal-triangle n)` that takes an integer `n`, and returns a "list of lists" representation of Pascal's triangle. The required format should be apparent form the examples below (note that line-breaks are insignificant; it's just the way Scheme's pretty-printer displays the output in a narrow window) **Don't forget: no mutation!**

pascal-triangle: *NonNegativeInteger* → *Listof*(*Listof*(*Integer*))

```
> (pascal-triangle 4)
((1 4 6 4 1) (1 3 3 1) (1 2 1) (1 1) (1))
> (pascal-triangle 12)
((1 12 66 220 495 792 924 792 495 220 66 12 1)
 (1 11 55 165 330 462 462 330 165 55 11 1)
 (1 10 45 120 210 252 210 120 45 10 1)
 (1 9 36 84 126 126 84 36 9 1)
 (1 8 28 56 70 56 28 8 1)
 (1 7 21 35 35 21 7 1)
 (1 6 15 20 15 6 1)
 (1 5 10 10 5 1)
 (1 4 6 4 1)
 (1 3 3 1)
 (1 2 1)
 (1 1)
 (1))
> (pascal-triangle 0)
((1))
> (pascal-triangle -3)
()
```

You should seek to do this simply and efficiently. You may need more than one helper procedure. If your collection of procedures for this problem starts creeping over 25 lines of code, perhaps you are making it too complicated. There is a straightforward solution that is considerably shorter than that.

#5 (10 points) Write the procedure (`product set1 set2`) that returns a list of 2-lists (lists of length 2) that represents the Cartesian product of the two sets. The 2-lists may appear in any order.

product: *set* × *set* → *set of 2-lists*

Examples:

`(product '(a b c) '(x y)))` ➔ `((a x) (a y) (b x) (b y) (c x) (c y))`

Background for problems 6–8 A **graph** can be represented in Scheme as a list of vertices. A vertex is represented as a list containing a symbol and a list of symbols. These are the name of the vertex and a list of the names of the vertices directly adjacent to it. No two vertices may be labeled with the same symbol. Also, no vertex can be adjacent to itself. For example, the complete graph on three symbols could be represented as `'((a (b c)) (b (a c)) (c (a b)))`. Undirected graphs assume that if an edge exists between a and b then b will appear in a's edge list and a will appear in b's. Assume that graphs are undirected unless told otherwise.

#6 (5 points) `max-edges`. Write a Scheme procedure `(max-edges n)` that takes a nonnegative integer and returns the maximum number of edges that an undirected graph of *n* vertices could have. The formula is well-known. Look online if you don't know it already. Do not use the factorial function. This should be able to run on very large inputs.

max-edges : *Integer* → *Integer*

For example

```
(max-edges 0)    ➔ 0
(max-edges 1)    ➔ 0
(max-edges 2)    ➔ 1
(max-edges 14)   ➔ 91
```

#7 (15 points) `complete?` Write a Scheme predicate `(complete? G)` that takes a graph, *G* (you may assume that it is a valid graph), and determines whether it is complete (i.e. every vertex is directly connected by an edge to every other vertex once but not to itself). Note that the null graph and the graph containing only one vertex are both complete. You may assume *G* is a valid graph, as defined in assignment 2.

Complete? : *GraphOf*(*Symbol*) → *Boolean*

For example:

```
(complete? '((a (b c d)) (b (a c d)) (c (a b d)) (d (a b c)))) ➔ #t
(complete? '((alpha (beta)) (beta (alpha)) (gamma ())))        ➔ #f
(complete? '())      ➔ #t
```

#8 (10 points) complete. Write a Scheme procedure `(complete ls)` that takes a list of symbols and returns the complete graph on the set of vertices labeled by those symbols.

complete: *ListOf*(*Symbol*) → ***matrix-ref :*** *Graphof(Symbol)*

For example:

```
(complete '(a))                  ➔ ((a ()))
(complete '(a b c))              ➔ ((a (b c)) (b (a c)) (c (a b)))
(complete '())                   ➔ ()
(complete? (complete '(q w e r t y u i o p)))  ➔ #t
```

#9 (10 points) Write a recursive Scheme procedure `(replace old new ls)` which takes two numbers, one to be replaced and one new value, as well as a simple list of numbers. It replaces all occurrences of `old` with `new` and returns the new list..

Examples:

```
(replace 5 7 '(1 5 2 5 7))          => (1 7 2 7 7)
(replace 5 7 '())                   => ()
```

#10 (10 points) Write a recursive Scheme procedure `(remove-first element ls)` which takes a symbol and a simple list of symbols. It returns a list that contains everything but the first occurrence of `element` in the list `ls`. If `element` is not in the list, the new list contains all of the elements of the original list.

Examples:

```
(remove-first 'b '(a b c b d))      => (a c b d)
(remove-first 'b '(a c d))          => (a c d)
(remove-first 'b '(a c b d))                => (a c d)
```

#11 (15 points) Write a recursive Scheme procedure `(remove-last element ls)` which takes a symbol and a simple list of symbols. It returns a list that contains everything but the last occurrence of `element` in the list `ls`. If `element` is not in the list, the new list contains all of the elements of the original list.

Examples:

```
(remove-last 'b '(a b c b d))       => (a b c d)
(remove-last 'b '(a c d))           => (a c d)
(remove-last 'b '(a c b d))         => (a c d)
```