CSSE 304 Assignment 1 Updated for Spring, 2016

Objectives You should learn

- the mechanics of editing and running Scheme programs.
- to write procedures that meet certain detailed specifications. (especially on problem #4)
- to test your code thoroughly.

Especially during the first two weeks, it is crucial that you don't get behind.

All three textbooks contain numerous exercises. You should read and give at least a little bit of thought to several of the exercises in the books, and actually work out as many as time allows. Some of the exercises in *EoPL* contain information that is crucial to understanding the later material in the text. I will assign most of these, but some will simply be thought problems rather than problems to turn in. This is because for many of these problems, the time required for writing them up would be greater than the time required to understand them.

Administrative preliminaries (most of these apply to later assignments also):

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.

The best way to learn Scheme is to jump in and do it. Hopefully the first problems are "shallow water" problems that you can wade into slowly; after a couple of assignments the water will get deeper quickly.

Assume that arguments' data types are correct: Unless a problem statement (on this and all CSSE 304 assignments) says otherwise, you may assume that each procedure you write will only be called with the correct number and correct type(s) of arguments. Your code does not have to check the argument count or argument types, unless different kinds of correct input are allowed. Of course, code that is very robust would check for erroneous input, but in most assignments for this course I want your focus to be on correctly processing correct arguments.

Indentation and style. Your programs should generally follow the rules in
http://www.cs.indiana.edu/proglang/scheme/indentation.html .
I will not be extremely strict about this, but I do want your code to be readable and to not have extremely long lines.

Square brackets: *Chez* Scheme allows you to use a [ ] pair anywhere that a ( ) pair may be used. Feel free to do this to make it easier to match parentheses and easier to read your code. Especially in

- The various clauses of `cond` or `cases`.
- The various variable definitions in a `let`, `let*` or `letrec`.

I will illustrate these uses of square brackets in numerous in-class examples.

Required comments and procedure order:

At the beginning of your solution file (I suggest naming the file *1.ss*), there should be a comment that includes your name and the assignment number.
Before the code for the required procedure for each problem, place a comment that includes the problem number.
If you write additional helper procedures, place their definitions near the required procedure's definition.
Please order the code in your file in order by problem number.

Automatic grading program overview. (submission details are below) Most of the programming assignments will be checked for correctness by the online grading program, https://plc.csse.rose-hulman.edu/. A certain number of points will be given for each test case. The grading program does not check for style issues, so those checks will be done by hand later for some of the assignments. In order to get all of the points for correctness, it is essential that each procedure that you write has the ***exact*** name that is specified in the problem, and that it expects the correct **number** of arguments of the correct **types**. You are allowed to test and debug as many times as you want, so usually no partial credit will be given on programming problems unless your code actually works for some of my test cases.
But the grading program is not the final authority. If you believe that it has made an error, treating your correct code as incorrect, talk to me. While I try to be very careful, it is certainly possible that the problem is with my test cases instead of your program. If you suspect that this is the case, send me an email that includes your code.

Also, if you feel that my test cases do not cover all of the specifications for a given procedure, feel free to make up your own test cases and send them to me or post them on Piazza. If I agree with you, I will use your test cases and give you extra credit for providing them.

FOR ALL ASSIGNMENTS, **I reserve the right to add test cases at any time**, including after the due date, and to re-run your program against those test cases. Your goal is to write a program that meets the problem specification, not just a program that passes my test cases.

How to submit this assignment. All of your Scheme code should be placed in one file; a good name is *1.ss*. Go to https://plc.csse.rose-hulman.edu/, and log in using your Rose-Human network username and password. Click **Student** and **Assignment 1**, then click the **Browse** button and browse to your `1.ss` file. Finally, click **upload**. If you do not get all of the points, you can change your `1.ss` file and re-submit as many times as you wish. But …

Be courteous: Test your code on your computer before each submission to the grading program. Using the server to test things that you can test off-line will slow down the server for everyone. For most assignments, I will provide a `test-cases` file that contains the same test-cases that the grading program uses, so you can test your code off-line instead of repeatedly submitting to the server. You can run the test cases for an individual problem, or run all of them at once by typing `(r)`.

Grading server disclaimer: The grading server (and usually the test cases) is made available before assignments are due as a service to you. It is intended to be a tool to help you discover the existence of some errors in your program. If the program gives you all of the possible points, it is likely that your code is correct, but not a guarantee; it merely says that your code passes all of the tests that I have placed on-line. You are still responsible for thinking of your own test cases to thoroughly test your code. I will rarely do this, but I reserve the right to use additional test cases when I actually grade your code. Also, if a "by hand" inspection of your code reveals significant issues in correctness, efficiency, or style, your score for a problem may be less than what the grading server says. The grading server is provided as a convenience to you and me. If it ever goes down, you have the ability to test your program without it. If it does go down, send email to plc-developers@rose-hulman.edu.

Important: Restriction on Mutation. One of the main goals of the first several assignments is to introduce you to the functional style of programming, in which we never modify the values of variables. (In Java we could accomplish this by declaring all variables `final`, but Java language limitations would make this impractical). Until further notice, you may not use **`set!`** or any other built-in procedure whose name ends in an exclamation point. Nor may you use any procedures that do input or output. It will be best to not use any exclamation points at all in your code. **You may receive zero credit for a problem if any procedure that is part of your solution for that problem changes the value of any variable that you define or reads a value from a file.** Note that **let** and **lambda** do not do mutation; you can use them freely.

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)

Reading Assignment and thought problems

Continue the reading assignments on the schedule page. In both TSPL and EoPL-1, you will encounter a few difficult concepts that we will clarify in class during the next few class days. From the reading, I want you to get the simple ideas yourself, and get exposure to the more difficult material, so it will make more sense when we discuss it in class.

Consider TSPL Exercise 2.2.3. For each part, figure out what the value should be, then try it out in Scheme to see if you are correct. If not, try to understand why (ask for help if needed).

Think about TSPL Exercises 2.2.4 and 2.2.5. Also 2.4.1 and 2.4.2 (if you think you have the right answers, you can check by trying it in Scheme).

Do Exercise TSPL 2.5.1. It would be silly for me to collect and grade it, because everyone can get the correct answers by simply entering the expressions in Scheme. So your goal, as always, should be to understand how these things work.

Much of the EOPL-1 reading will duplicate Assignment 0's reading in TSPL, but I believe it is good for you to get more than one perspective on these things. You should do EOPL-1 exercises 1.2.1, 1.2.2, and 1.2.3 mentally; then enter the code into Scheme to check your work. There is nothing to turn in for these exercises. **Page 22 is challenging**—see the Assignment 0 FAQ.

If you find some of the reading to be rough going, don't panic. The authors of both books have a habit of going along explaining simple stuff and suddenly throwing in an example that is very challenging. Just slow down, read it a couple more times, and write down questions that you can ask me, the assistants, or other students later.

Programming Problems to turn in (there are seven problems):

In all of these problems, you may assume that the procedures that you write will be called with legal arguments. You do not have to check for illegal input. For example, you may assume that the argument to the procedure from problem #1 is an integer or a rational number.

#1 (5 points) Write a Scheme procedure `(Fahrenheit->Celsius temperature)` that takes an integer or rational number `temperature` as its argument. The argument represents a Fahrenheit temperature, and the procedure returns the corresponding Celsius temperature as an integer or rational number.

In the notation of EoPL, **Fahrenheit->Celsius :** *Number* → *Number*
I.e., *Fahrenheit→Celsius* is a function that takes a number as an argument and returns a number.

Examples:

```
(Fahrenheit->Celsius 32)     => 0
(Fahrenheit->Celsius 0)      => -160/9
(Fahrenheit->Celsius -40)    =>  -40
(Fahrenheit->Celsius 241/5)  => 9
```

Did you forget the conversion formula? A search on Google for "Fahrenheit to Celsius" may help.
Be sure to use integers or fractions, not floating-point numbers, in your conversion code.

Caution: Correct capitalization of the name `Fahrenheit->Celsius` is essential.

Background for problems 2-4 A (closed) **interval** of real numbers includes all numbers between the endpoints (including the endpoints). We can represent an interval in Scheme by a list of two numbers `'(first second)`. This represents the interval $\{x : first \leq x \leq second\}$. We will not allow empty intervals, so *first* must always be less than or equal to *second*. If *first* = *second*, the interval contains exactly one number. For simplicity, you may assume that the endpoints of all of our intervals are integers, so that you do not have to worry about floating-point "near equality".

#2 (5 points) Write a Scheme procedure `(interval-contains? interval number)` where *interval* is an interval and *number* is a number. It returns a Boolean value that indicates whether *number* is in the closed *interval*.

interval-contains? : *Interval* × *Number* → *Boolean*

Examples:

```
(interval-contains? '(5 8) 6)     => #t
(interval-contains? '(5 8) 5)     => #t
(interval-contains? '(5 8) 4)     => #f
(interval-contains? '(5 5) 14)    => #f
```

#3 (8 points) Write a Scheme procedure `(interval-intersects? i1 i2)` where *i1 and i2* are intervals. It returns a Boolean value that indicates whether the intervals have a nonempty intersection. **Edge case:** If the intersection contains a single number, this procedure should return #t.

interval-intersects? : *Interval* × *Interval* → *Boolean*

Examples:

```
(interval-intersects? '(1 4) '(2 5))       => #t
(interval-intersects? '(2 5) '(1 14))      => #t
(interval-intersects? '(2 5) '(1 2))       => #t
(interval-intersects? '(1 1) '(1 1))       => #t
(interval-intersects? '(1 3) '(12 17))     => #f
```

#4 (8 points) The *union* of two intervals is a **list** containing

- both intervals, if the intervals don't intersect, or
- a single, possibly larger, interval if the intervals do intersect.

Write a Scheme procedure `(interval-union i1 i2)` that returns the union of the intervals *i1* and *i2*.

interval-union: *Interval* × *Interval* →*Listof(Interval)*

Examples (make careful note of the form of the values returned by the first three):

```
(interval-union '(1 5) '(2 6))   => ((1 6))
(interval-union '(1 5) '(2 4))   => ((1 5))
(interval-union '(1 5) '(5 5))   => ((1 5))
(interval-union '(1 5) '(15 25)) => ((1 5) (15 25))
(interval-union '(5 5) '(25 25)) => ((5 5) (25 25))
```

Caution: Look carefully at the form of the return values.
In the past, sometimes students have produced `(1 6)` instead of `((1 6))`.

#5 (4 points) Write the procedure (`divisible-by-7? num`)
that returns `#t` if the non-negative integer *`num`* is divisible by 7, and `#f` otherwise.

divisible-by-7? : *NonNegativeInteger* → *Boolean*

Examples:

```
(divisible-by-7? 12) =>   #f
(divisible-by-7? 21) =>   #t
```

You may assume that `num` is a non-negative integer (and your code does not have to test for this).
The Scheme `modulo` procedure may be helpful here.

#6 (3 points) Write the procedure (`ends-with-7? num`) that returns `#t` if the decimal representation of *`num`* ends with 7, and `#f` otherwise.

Ends-with-7? : *NonNegativeInteger* → *Boolean*

Examples:

```
(ends-with-7? 96) =>   #f
(ends-with-7? 31489370283367) =>   #t
```

You may assume that *`num`* is a positive integer (i.e., your code does not have to test for this).
My intention is that your code will use arithmetic instead of string operations to determine the correct answer.

#7 (3 points) Write the procedures `1st`, `2nd`, and `3rd` that pick out those corresponding parts of a proper list. You do not have to handle the case where the list is too short to have the requested part.

Examples:

```
(1st  '(a b c d e)) =>   a
(2nd  '(a b c d e)) =>   b
(3rd  '(a b c d e)) =>   c
```

Optional "work-ahead" practice problems (not to be turned in; some of these may appear on later assignments)

These are problems that I assigned in previous years. If you feel like you need some extra practice, these would be good ones to do.

These practice problems will deal with points and vectors in 3 dimensions.

We will represent a point or a vector by a list of 3 numbers. For example, the list `(5 6 -7)` can represent either the vector 5**i** + 6**j** - 7**k** or the point (5, 6, -7).

Note that Scheme has a built-in `vector` type with associated procedures. This built-in type is used for representing arrays. In order to avoid having your code conflict with this built-in type, you should use `vec` instead of `vector` in the names of your functions and their arguments. We could use the built-in `vector` type for this problem, but I choose not to do so, so that you can get additional practice with picking out parts of lists.

Some of you have written similar functions in Python or Java. While the computations are essentially the same in Scheme, the Scheme code is generally simpler and shorter.

#P1 Write the procedure `(make-vec-from-points p1 p2)` that returns the vector that goes from the point `p1` to the point p2. For example,

```
(make-vec-from-points '(1 3 4) '(3 6 2)) => (2 3 -2)
```

Note that when I write "Write the procedure `(make-vec-from-points p1 p2)`", it is a shorthand for "Write the procedure `make-vec-from-points` with two arguments which I refer to here as `p1` and `p2`."
The definition of such a procedure would begin

```
(define make-vec-from-points
   (lambda (p1 p2)
        ... ))
```

#P2 Write the procedure `(dot-product v1 v2)` that returns the dot-product (scalar product) of the two vectors `v1` and `v2`. For example, `(dot-product '(1 2 3) '(4 5 6)) => 32`

#P3 Write the procedure `(vec-length v)` that returns the magnitude of the vector `v`.
For example, `(vec-length '(3 4 12)) => 13`

#P4 Write the procedure `(distance p1 p2)` that returns the distance from the point `p1` to the point `p2`.
For example, `(distance '(1 3 4) '(3 6 2)) => 4.1231056256176606`

#P5 Write the procedure `(cross-product v1 v2)` that returns the cross-product (vector product) of the two vectors `v1` and `v2`. For example, `(cross-product '(1 3 4) '(3 6 2)) => (-18 10 -3)`.

#P6 Write the procedure `(parallel? v1 v2)` that returns `#t` if `v1` and `v2` are parallel vectors, #f otherwise. (You only have to guarantee that it will work if the coefficients of both vectors are integers or rational numbers. Otherwise round-off error may make two parallel vectors appear to be non-parallel or vice-versa).
For example,

```
(parallel? '(1 3 4) '(3 6 2)) => #f.
(parallel? '(1 3 4) '(-3 -9 -12)) => #t.
```

Note that the zero vector is parallel to everything.

#P7 Write the procedure `(collinear? p1 p2 p3)` that returns `#t` if the points `p1`, `p2`, and `p3` are all on the same straight line, `#f` otherwise. Same disclaimer about round-off error as in #6.
For example, `(collinear? '(1 3 4) '(3 6 2) '(7 12 -2) => #t.`

#P8 Write the procedure `(same-point? p1 p2)` that returns `#t` if the points `p1`, and `p2` have the same coordinates (and thus represent the same point), and returns `#f` otherwise.
For example, `(same-point? '(1 3 4) '(1 3 4)) => #t`

```
(same-point? '(1 3 4) '(3 2 4)) => #f
```

#P9 Write the procedure `(member-point? p list-of-points)` that returns `#t` if the point `p` is one of the points in the list `list-of-points` and returns `#f` otherwise.

For example,
```
(member-point? '(1 3 4) '()) => #f
(member-point? '(1 3 4) '( (2 3 5) (1 3 4) (1 1 4))) => #t
(member-point? '(1 3 4) '( (2 3 5) (1 3 5) (1 5 4))) => #f
```

#P10 Write the procedure `(nearest-point p list-of-points)` that returns the point in the non-empty list `list-of-points` that is closest to `p`. If two points "tie" for nearest, return either one.

For example, `(nearest-point '(1 2 1) '((7 5 0) (2 1 0) (-6 -7 -8)) => (2 1 0)`

Problems P1 - P10 are optional practice problems (not to be turned in); the seven A1 turn-in problems are on earlier pages.

CSSE 304 Assignment 2 Updated for Spring, 2016

Objectives You should learn

- to write procedures that precisely meet given specifications.
- to gain experience with picking out parts of lists.
- to write procedures that make simple decisions.
- to test your code thoroughly.

Administrative preliminaries (most of these apply to later assignments also). If you did not read the administrative preliminaries for Assignment 1 in detail, you should do so.

This is an individual assignment. You can talk to anyone 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. You should never give or receive code for the individual assignments.

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

Turning in this assignment. Write all of the required procedures in one file, **2.ss**, and upload it for assignment A2 to the PLC grading server. As with A1, do testing on your own computer first, so the server does not get bogged down.

Restriction on Mutation continues. One of the main goals of the first few assignments is to introduce you to the functional style of programming, in which the values of variables are never modified. Until further notice, you may not use `set!` or any other built-in procedure whose name ends in an exclamation point. It will be best to not use any exclamation points at all in your code. **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)

Reading Assignment: see the schedule page

Some of the EOPL-1 reading covers topics similar to the reading in TSPL, but I believe it is good for you to get more than one perspective on this (in particular, a perspective that is similar to that of EoPL).

Assume valid inputs. As in assignment 1, you do not have to check for illegal arguments to your procedures.

Problems to turn in:

#1 (5 points) (a) (0)Write the procedure `(fact n)` which takes a non-negative integer *n* and returns *n* factorial. You can just copy this procedure form Assignment 0, and call it from your `choose` procedure from part (b).

fact: *NonNegativeInteger* → *Integer*

Examples:

```
(fact 0) => 1
(fact 1) => 1
(fact 5) => 120
```

(b) (5) Write the procedure `(choose n k)` which returns the number of different subsets of *k* items chosen from a set of *n* items. This is also known as the binomial coefficient. If you've forgotten the formula for this, a Google search for "Binomial Coefficient" should be helpful.

choose: *NonNegativeInteger* × *NonNegativeInteger* → *NonNegativeInteger* **(examples on next page)**

Examples:

```
(choose 0 0)  => 1
(choose 5 1)  => 5
(choose 10 5) => 252
```

#2 (8 points) Write the procedure `(range m n)` that returns the ordered list of integers starting at the integer *m* and increasing by one until just before the integer *n* is reached (do not include n in the resulting list). This is similar to Python's *range* function. If *n* is less than or equal to *m*, `make-range` returns the empty list.

range: *Integer* × *Integer* → *Listof(Integer)*

Examples:

```
(range 5 10) → (5 6 7 8 9)
(range 5 6)  → (5)
(range 5 5)  → ( )
```

#3 (10 points) In mathematics, we informally define a *set* to be a collection of items with no duplicates. In Scheme, we could represent a set by a (single-level) list. We say that a list is a set if and only if it contains no duplicates. We say that two objects `o1` and `o2` are duplicates if `(equal? o1 o2)`. Write the predicate `(set? list)`, that takes any list as an argument and determines whether it is a set.

set? : *list* → *Boolean*

Examples:

```
(set? '()) → #t                          ; empty set
(set? '(1 (2 3) (3 2) 5)) → #t           ; (2 3) and (3 2) are not equal?
(set? '(r o s e - h u l m a n)) → #t
(set? '(c o m p u t e r s c i e n c e)) → #f
```

#4 (5 points) Write a procedure `(sum-of-squares lon)` that takes a (single-level) list of numbers, `lon`, and returns the sum of the squares of the numbers in `lon`.

sum-of-squares: *Listof(Number)* → *Number*

Examples:

```
(sum-of-squares '(1 3 5 7)) → 84
(sum-of-squares '()) → 0
```

The remaining problems (and some problems in Assignment 3) will deal with points and vectors in three dimensions.

We will represent a point or a vector by a list of three numbers. For example, the list `(5 6 -7)` can represent either the vector 5**i** + 6**j** - 7**k** or the point (5, 6, -7). In the procedure type specifications below, I'll use *Point* and *Vector* as the names of the types, even though both will be implemented by the same underlying Scheme type.

Note that Scheme has a built-in `vector` type with associated procedures. This `vector` type is used for representing arrays. In order to avoid having your code conflict with this built-in type, you should use `vec` instead of `vector` in the names of your functions and their arguments. We could use the built-in `vector` type for this problem, but I choose not to do so, so that you can get additional practice with picking out parts of lists.

#5 (5 points) Write the procedure `(make-vec-from-points p1 p2)` that returns the vector that goes from the point `p1` to the point p2.

make-vec-from-points: *Point* × *Point* → *Vector*

Example:

```
(make-vec-from-points '(1 3 4) '(3 6 2)) → (2 3 -2)
```

#6 (5 points) Write the procedure `(dot-product v1 v2)` that returns the dot-product (scalar product) of the two vectors `v1` and `v2`.

dot-product: *Vector* × *Vector* → *Number*

Example:
```
(dot-product '(1 2 3) '(4 5 6)) ➔ 32
```

#7 (5 points) Write the procedure `(vec-length v)` that returns the magnitude of the vector `v`. So that we do not have to worry about round-off error, my tests will only use examples where the result is an integer.

vec-length: *Vector* → *Number*

Example:
```
(vec-length '(3 4 12)) ➔ 13
```

#8 (5 points) Write the procedure `(distance p1 p2)` that returns the distance from the point `p1` to the point `p2`. So that we do not have to worry about round-off error, my tests will only use examples where the returned value is an integer. [Hint: You may want to call some previously-defined procedures in your definition.

distance: *Point* × *Point* → *Number*

Example:
```
(distance '(3 1 2) '(15 -15 23)) => 29
```

#9 (5 points) Write the procedure `(cross-product v1 v2)` that returns the cross-product (vector product) of the two vectors `v1` and `v2`.

cross-product: *Vector* × *Vector* → *Vector*

Examples:
```
(cross-product '(1 3 4) '(3 6 2))  ➔ (-18 10 -3)
(cross-product '(1 3 4) '(3 9 12)) ➔ (0 0 0)
```

#10 (5 points) Write the procedure `(parallel? v1 v2)` that returns `#t` if `v1` and `v2` are parallel vectors, `#f` otherwise. Note that the zero vector is parallel to everything. You only have to guarantee that your procedure will work if the coefficients of both vectors contain only integers or rational numbers. Otherwise round-off error may make two parallel vectors appear to be non-parallel or vice-versa.

parallel?: *Vector* × *Vector* → *Boolean*

Examples:
```
(parallel? '(1 3 4) '(3 6 2)) ➔ #f.
(parallel? '(1 3 4) '(-3 -9 -12)) ➔ #t.
```

#11 (3 points) Write the procedure `(collinear? p1 p2 p3)` that returns `#t` if the points `p1`, `p2`, and `p3` are all on the same straight line, `#f` otherwise. Same disclaimer about round-off error as in the previous problems.

```
(collinear? '(1 3 4) '(3 6 2) '(7 12 -2) ➔ #t
(collinear? '(1 3 4) '(3 6 2) '(7 12 1) ➔ #f.
```

CSSE 304 Assignment 3 Updated for Spring, 2016

Objectives You should learn

- to write procedures that meet certain specifications.
- to practice using previously-written procedures as helpers for new procedures
- to write more complex recursive procedures in a functional style.
- to test your code thoroughly.

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 3. 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.

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)

Problems to turn in:

The first problem refers to the point and vector framework from Assignment 2, which is again described here. You may copy any of the procedures that you wrote for that assignment into this assignment and use them here.

> We will represent a point or a vector by a list of 3 numbers. For example, the list `(5 6 -7)` can represent either the vector 5**i** + 6**j** - 7**k** or the point (5, 6, -7). In the procedure type specifications below, I'll use *Point* and *Vector* as the names of the types, even though both will be implemented by the same underlying Scheme type.
>
> Note that Scheme has a built-in `vector` type with associated procedures. This `vector` type is used for representing arrays. In order to avoid having your code conflict with this built-in type, you should use `vec` instead of `vector` in the names of your functions and their arguments. We could use the built-in `vector` type for this problem, but I choose not to do so, so that you can get additional practice with picking out parts of lists.

#1 (10 points) Write the procedure (`nearest-point p list-of-points`) that returns the point in the non-empty list `list-of-points` that is closest to `p`. If two points "tie" for nearest, return the one that appears first in `list-of-points`.

nearest-point: *Point* × *Listof*(*Point*) → *Point*

Examples:

```
(nearest-point '(1  2  1) '((7 5 0) (2 1 0) (-6 -7 -8)))   → (2 1 0)
```

The next problems refer to the definition of *sets in Scheme* from Assignment 2, which are repeated here.

> We represent a set by a (single-level) list of objects, which may themselves be lists. We say that such a list is a set if and only if it contains no duplicates. By "no duplicates", I mean that no two items in the list are `equal?` .

#2 (5 points) The `union` of two sets is the set of all items that occur in either or both sets (the order does not matter).

union: *Set* × *Set* → *Set*

Examples:

```
(union '(a f e h t b) '(g c e a b)) → (a f e h t b g c)   ; (or some permutation of it)
(union '(2 3 4) '(1 a b 2)) → (2 3 4 1 a b)              ; (or some permutation of it)
```

#3 (10 points) Write the procedure `(intersection s1 s2)` The intersection of two sets is a set containing all items that occur in both sets (order does not matter). You may assume that both arguments are sets.

intersection: *set* × *set* → *set*

Examples:

```
(intersection '(a f e h t b p) '(g c e a b)) ➔ (a e b)   ; (or some permutation of it)
(intersection '(2 3 4) '(1 a b)) ➔ ()
```

You may assume that the arguments are sets; you do not have to test for that. Again, use `equal?` as your test for duplicate items.

#4 (10 points) A set X is a *subset* of the set Y if every member of X is also a member of Y. The procedure `(subset? s1 s2)` takes two sets as arguments and tests whether `s1` is a subset of `s2`. You may want to write a helper procedure. You may assume that both arguments are sets.

subset?: *set* × *set* → *Boolean*

Examples

```
(subset? '(c b) '(a c d b e)) ➔ #t
(subset? '(c b) '(a d b e)) ➔ #f
(subset? '() '(a d b e)) ➔ #t
```

#5 (15 points) A *relation* is defined in mathematics to be a set of ordered pairs. The set of all items that appear as the first member of one of the ordered pairs is called the *domain* of the relation. The set of all items that appear as the second member of one of the ordered pairs is called the *range* of the relation. In Scheme, we can represent a relation as a list of 2-lists (a 2-list is a list of length 2). For example `((2 3) (3 4) (-1 3))` represents a relation with domain `(2 3 -1)` and range `(3 4)`. Write the procedure `(relation? obj)` that takes any Scheme object as an argument and determines whether or not it represents a relation. You will probably want to use `set?` from a previous exercise in your definition of `relation?`. [Note that because you were just getting started on Scheme, my tests for `set?` did not include any values that were not lists. Now you may want to go back and "beef up" your `set?` procedure so it returns `#f` if its argument is not a list. Note that you may use `list?` In your code if you wish.

relation?: *scheme-object* → *Boolean*

Examples

```
(relation? 5) ➔ #f
(relation? '()) ➔ #t
(relation? '((a b) (b c))) ➔ #t
(relation? '((a b) (b a) (b b) (a a))) ➔ #t
(relation? '((a b) (b c d))) ➔ #f
(relation? '((a b) (c d) (a b))) ➔ #f
(relation? '((a b) (c d) "5")) ➔ #f
(relation? '((a b) . (b c))) ➔ #f
```

#6 (10 points) Write a procedure `(domain r)` that returns the set that is the domain of the given relation. Recall that the *domain* of a relation is the set of all elements that occur as the first element of an ordered pair in the relation.

domain: *relation* → *set*

Examples

```
(domain '((1 2) (3 4) (1 3) (1 6))) ➔ (1 3)   ; or some permutation of it
(domain '()) ➔ ()
(domain '((a b) (b d) (a e) (c e))) ➔ (a b c) ; or some permutation of it
```

#7 (15 points) A relation is *reflexive* if every element of the domain and range is related to itself. I.e., if `(a b)` is in the relation, so are `(a a)` and `(b b)`. The procedure `(reflexive? r)` returns `#t` if relation r is reflexive and `#f` otherwise. You may assume that `r` is a relation.

reflexive?: *relation* → *Boolean*

Examples:

```
(reflexive? '((a b) (b a) (b b) (a a))) ➔ #t

(reflexive? '((a b) (b c) (a c))) ➔ #f
```

#8 (10 points) Consider hailstone sequences, related to the Collatz conjecture. If the positive integer *n* is a number in such a sequence, the next number in the sequence is (image below is from the linked Wikipedia page).

$$f(n) = \begin{cases} n/2 & \text{if } n \equiv 0 \pmod 2 \\ 3n+1 & \text{if } n \equiv 1 \pmod 2. \end{cases}$$

The first case is for *n* even, the second is for *n* odd. The conjecture is that all such sequences eventually reach the number 1. Define the procedure `(hailstone-step-count n)` to be the number of applications of the above function *f* required to reach 1 if we start with *n*. If the conjecture happens to be false, then there exists some n>0 such that hailstone-step-count(n) is infinite. You will not encounter any such numbers in my tests for this problem, so your code does not need to attempt to check for this!

.

Examples:

```
(hailstone-step-count 1) ➔ 0      ; already 1, so no applications of f are needed
(hailstone-step-count 2) ➔ 1      ; 2 > 1
(hailstone-step-count 3) ➔ 7      ; 3 > 10 > 5 > 16 > 8 > 4 > 2 > 1
(hailstone-step-count 4) ➔ 2      ; 4 > 2  > 1
(hailstone-step-count 7) ➔ 16     ; 7 > 22 > 11 > 34 > 17 > 52 > 26 > 13 > 40 > 20 > 10 > 5 > 16 > 8 > 4 > 2 > 1
(hailstone-step-count 871) ➔ 178
```

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)) ➔  ()
```

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)
```

CSSE 304 Assignment 6 Updated for Spring, 2016

Objectives: You should learn

- To begin to use first-class procedures effectively.
- To think a bit about efficiency of programs.

Details for these instructions are in the previous assignment

Individual assignment**.** Comments at beginning, before each problem, when you do anything non-obvious. Submit to server (test offline first). Mutation not allowed.

Abbreviations for the textbooks:

EoPL - Essentials of Programming Languages, 3rd Edition
TSPL - The Scheme Programming Language, 4rd Edition
EoPL-1 - Essentials of Programming Languages, 1st Edition (1st day handout)

Reading Assignment: See the schedule page. Have you been keeping up with the reading?

Problems to turn in: For many of these, you will want to write one or more helper procedures.

Some of the problems deal with ***currying***. http://en.wikipedia.org/wiki/Currying describes this as:

> In mathematics and computer science, currying (schönfinkeling) is the technique of transforming a function that takes multiple arguments (or a tuple of arguments) in such a way that it can be called as a chain of functions, each with a single argument (partial application). It was originated by Moses Schönfinkel and later worked out by Haskell Curry.

Optional, not required knowledge for this course: An interesting discussion of the advantages of currying (the language of discourse is Haskell, but I think you can still follow much of the discussion).
http://www.reddit.com/r/programming/comments/181y2a/what_is_the_advantage_of_currying/
Some simple examples of currying appear on pages 26 (last sentence) through 28 of EoPL-1. The first two turnin-problems are from that section, and I recommend that you also think about problem 1.3.6.

#1 (10 points) `curry2`. This is EoPL-1 Exercise 1.3.4, page 28. Examples are on that page.

#2 (10 points) EoPL-1 Exercise 1.3.5, page 28. Call your procedure `curried-compose` .
For example, `(((curried-compose car) cdr) '(a b c))` ➔ `b`

#3 (10 points) `compose`. EoPL-1 Exercise 1.3.7, page 29. This one will most likely begin

```
(define compose
  (lambda list-of-functions      ; notice the lack of parentheses around the argument name.
((compose list list) 'abc)             ➔ ((abc))
((compose car cdr cdr) '(a b c d))     ➔ c
```

#4 (10 points) Write the procedure `make-list-c` that is a curried version of `make-list`.
(Note that the original `make-list` is described in TSPL Exercise 2.8.3, and done in class Day 3.
We also wrote `make-list` in class during Week 1.

make-list-c : *Integer* → (*SchemeObject* → *Listof(SchemeObject*))

Examples:

```
((make-list-c 3) 'xyz)          ➔   (xyz xyz xyz)
(let ([triple (make-list-c 3)])
  (triple "cat"))               ➔   ("cat" "cat" "cat")
```

#5 (10 points) Write `let->application` which takes a `let` expression (represented as a list) and returns the equivalent expression, also represented as a list, representing an application of a procedure created by a `lambda` expression. Your solution should not change the body of the `let` expression. This procedure's output list replaces only the top-level `let` by an equivalent application of a `lambda` expression. You do not have to find and replace replace any non-top-level `let`s. You may assume that the `let` expression has the proper form; your procedure does not have to check for this. Furthermore, you may assume that the `let` expression is *not* a named `let`. **(continued next page)**

let->application : *SchemeCode* → *SchemeCode*

Example:

```
(let->application '(let ((x 4) (y 3))
                     (let ((z 5))
                        (+ x (+ y z)))))
   ➔
((lambda (x y)
   (let ((z 5))
     (+ x (+ y z))))
 4 3)
```

#6 (10 points) Write `let*->let` which takes a let* expression (represented as a list) and returns the equivalent nested `let` expression. This procedure replaces only the **top-level** `let*` by an equivalent nested `let` expression. You may assume that the `let*` expression has the proper form.

let*->let: *SchemeCode* → *SchemeCode*

Example:

```
(let*->let '(let* ([a 3] [b (+ a 4)]) b ))
➔
(let ([a 3])
   (let ([b (+ a 4)])
      b))
```

#7 (10 points) Write (`filter-in pred? lst`) where the type of each element of `lst` is appropriate for an application of the predicate `pred?`. It returns a list (in their original order) of all elements of `lst` for which `pred?` returns `#t`.

filter-in: *Procedure* × *List* → *List*

Examples:

```
(filter-in positive? '(-1 2 0 3 -6 5)) ➔ (2 3 5)
(filter-in null? '(() (1 2) (3 4) () ())) ➔ (() () ())
(filter-in list? '(() (1 2) (3 . 4) #2(4 5))) ➔ (() (1 2))
(filter-in pair? '(() (1 2) (3 . 4) #2(4 5))) ➔ ((1 2) (3 . 4))
(filter-in null? '()) ➔ ()
```

#8 (10 points) Write (`filter-out pred? lst`) where each element of the list `lst` has a type that is appropriate for an application of the predicate `pred?`. It returns a list (in their original order) of all elements of `lst` for which `pred?` returns `#f`.

filter-out: *Procedure* × *List* → *List*

Examples (These test cases and their answers may also help you to better understand the `list?` and `pair?` procedures**):**

```
(filter-out positive? '(-1 2 0 3 -6 5 0)) ➔ (-1 0 -6 0)
(filter-out null? '(() (1 2) (3 4) () ())) ➔ ((1 2) (3 4))
(filter-out list? '(() (1 2) (3 . 4) #2(4 5))) ➔ ((3 . 4) #(4 5))
(filter-out pair? '(() (1 2) (3 . 4) #2(4 5))) ➔ (() #(4 5))
(filter-out null? '()) ➔ ()
```

#9 (10 points) Write a Scheme procedure `(sort-list-of-symbols los)` which takes a list of symbols and returns a list of the same symbols sorted as if they were strings. You will probably find the following procedures to be useful:

`symbol->string`, `map`, `string<?`, `sort` (look it up in the Chez Scheme Users' Guide). Note that we have not covered specifics related to this problem, It is time for you to read some documentation and figure out how to use things.

sort-list-of-symbols: *ListOf*(*Symbol*) → *ListOf*(*Symbol*)

Example `(sort-list-of-symbols '(b c d g ab f b r m))` ➔ `(ab b b c d f g m r)`

#10 (10 points) `invert` EoPL 1.16, page 26

#11 (10 points) `vector-index` Like `list-index`, but its second argument is a vector, not a list.

CSSE 304 Assignment 7 162 points Updated for Spring, 2016

Objectives: You should learn

- More about list processing.
- More about the use of let, letrec, named let, map, and apply.
- How to base recursive programs on recursive datatype definitions.

This is divided into two parts, but each part is substantial. Start early. Especially problems 5 and 7

.

Details for these instructions are in the previous assignment, so not repeated here: Individual assignment**.** Comments at beginning, before each problem, and when you do anything non-obvious. Submit to server (test offline first). Your code must not mutate (unless a particular problem calls for it), read, or write anything. Assume arguments have correct form unless problem says otherwise.

Abbreviations for the textbooks:

EoPL - Essentials of Programming Languages, 3rd Edition
TSPL - The Scheme Programming Language, 4rd Edition
EoPL-1 - Essentials of Programming Languages, 1st Edition (handout)

Reading Assignment: See the schedule page. Have you been keeping up with the reading?

Problems to turn in: For many of these, you will want to write one or more helper procedures.

Some of the problems come from Chapter 1 of EoPL (3rd edition), which you should have been reading already.

#1 (10 points) `vector-append-list` `(vector-append-list v lst)` returns a new vector with the elements of `lst` attached to the end of `v`. Do this without using `vector->list`, `list->vector`, or `append`.
For this problem only, you can and should use mutation: namely the vector-set! procedure. Note that this procedure dos not return a value.

#2 (20 points) Write `qsort`. `(qsort pred ls)`, a Scheme procedure whose arguments are
a predicate (total ordering) which takes two arguments `x` and `y`, and returns `#t` if `x` is "less than" `y`, `#f` otherwise.
a list whose items can be compared using this predicate.
`qsort` should produce the sorted list using a QuickSort algorithm (write your own; do not use Scheme's `sort` procedure).

For example:

```
(qsort < '(4 2 4 3 2 4 1 8 2 1 3 4)) ➔ (1 1 2 2 2 3 3 4 4 4 4 8)

(qsort (lambda (x y) (< (abs (- x 10)) (abs (- y 10))))
       '(5 1 10 8 16 17 23 -1))
     ➔ (10 8 5 16 17 1 -1 23)
```

If you do not remember how QuickSort works, see http://en.wikipedia.org/wiki/Quicksort or Chapter 7 of the Weiss book used for CSSE230. There are quicksort algorithms that do fancy things when choosing the pivot in order to attempt to avoid the worst case. You do not need to do any of those things here; you can simply use the `car` of the list as the pivot. Since mutation is not allowed, your algorithm cannot do the sort in-place. Furthermore, you are not allowed to copy the list elements to a vector, then sort the vector and copy back to a list. All of your work should be done with lists.

#3 (15 points) Write a Scheme predicate `(connected? g)` that takes an undirected graph (represented in the same way as in previous assignments) and determines whether it is connected. A graph is connected if every vertex can be reached from every other vertex *via* a sequence of edges. Starting point: the null graph and the graph with one vertex are connected. For example:

```
(connected? '((a (c)) (b (c)) (c (a b))))          ➔ #t
(connected? '((a ()) (b (c)) (c (b))))             ➔ #f
(connected? '((a (b)) (b (a)) (c (d)) (d (c))))    ➔ #f
```

#4 (10 points) Write `(reverse-it lst)` that takes a single-level list `lst` and returns a new list that has the elements of `lst` in reverse order. The original list should not be changed. Can you do this in O(n) time? You probably cannot if you use `append`. Obviously, you should not use Scheme's `reverse` procedure in your implementation.

#5 (40 points) **Examples are in the test cases**. A Binary Search Tree (BST) datatype is defined on page 10 of EoPL. Define the following procedures:

1. `(empty-BST)` takes no arguments and creates an empty tree, which is represented by an empty list.
2. `(empty-BST? obj)` takes a Scheme object `obj`. It returns `#t` if `obj` is an empty BST and `#f` otherwise.
3. `(BST-insert num bst)` returns a BST *result*. If `num` is already in `bst`, *result* is structurally equivalent to `bst`. If `num` is not already in `bst`, *result* adds `num` in its proper place relative to the other nodes in a tree whose shape is the same as the original. Like any BST insertion, this should have a worst-case running time that is O(height(`bst`)).
4. `(BST-inorder bst)` should (in O(N) time) produce an ordered list of the values in `bst`.
5. `(BST? obj)` returns `#t` if Scheme object `obj` is a BST and `#f` otherwise.
6. `BST-element`, `BST-left`, `BST-right`. Accessor procedures for the parts of a node.
7. `(BST-insert-nodes bst nums)` starts with tree `bst` and inserts each integer from the list `nums`, in the given order, returning the tree that includes all of the inserted nodes. The original tree is not changed, and this procedure does no mutation.
8. `(BST-contains? bst num)` determines, in time that is O(height(`bst`)), whether `num` is in `bst`.

#6 (10 points) Write (`map-by-position fn-list arg-list`) where

- `fn-list` and `arg-list` have the same length,
- and each element of the list `fn-list` is a procedure that can take one argument,
- and each element of the list `arg-list` has a type that is appropriate to be an argument of the corresponding element of `fn-list`.

`map-by-position` returns a list (in their original order) of the results of applying each function from `fn-list` to the corresponding value from `arg-list`. **You must use `map` to solve this problem; no explicit recursion is allowed.**

```
(map-by-position (list cadr - length (lambda(x)(- x 3)))
                 '((1 2) -2 (3 4) 5))                    ➔ (2 2 2 2)
```

#7 (57 points) Consider the following syntax definition from page 9 of EoPL:

```
<bintree> ::= <integer>  | (<symbol> <bintree> <bintree>)
```

Write the following procedures:

- `(bt-leaf-sum T)` finds the sum of all of the numbers in the leaves of the `bintree` `T`.
- `(bt-inorder-list T)` creates a list of the symbols from the *interior* nodes of T, in the order that they would be visited in an inorder traversal of the binary tree.
- `(bt-max T)` returns the largest integer in the tree.
- `(bt-max-interior T)` takes a binary tree with at least one interior node, and returns (in O(N) time, where N is the number of nodes) the symbol associated with an interior node whose subtree has a maximal leaf sum (at least as large as the sum from any other interior node in the tree). If multiple nodes in the tree have the same maximal leaf-sum, return the symbol associated with the leftmost maximal node.

This `bt-max-interior` procedure is trickier than it looks at first! Leave yourself a day or more between when you start it and when it is due.

You may not use mutation.
You may not traverse any subtree twice (such as by calling `bt-leaf-sum` on every interior node).
You may not create any additional size-O(N) data structures that you then traverse to get the answer.
Think about how to return enough info from each recursive call to solve this without doing another traversal.

Note: We will revisit this linear-time max-interior problem several times later. If you do not get this version, the later versions will be harder for you, so you should do what it takes to get this one.

CSSE 304 Assignment 8 Updated for Spring, 2016

Problem 4 is a challenging problem. It is also very important, because later assignments will ask you to write variations of it.

Objectives To practice **Following the Grammar** when writing recursive code.

Same rules as the previous assignments. In particular, mutation is not allowed.
No argument error-checking is required. You may assume that all arguments have the correct form.

You may find *Chez* Scheme's `trace-let` to be helpful for debugging your code; if you use it (or any of the other `trace` syntactic forms), be sure to remove it before submitting to the grading server.

#1 (50 points) These s-list procedures have a lot in common with the s-list procedures that we wrote during our Session 8 class. Recall the extended BNF grammar for s-lists:

```
<s-list>       ::= ( {<s-expression>}* )
<s-expression> ::=  <symbol> | <s-list>
```

(a) `(slist-map proc slist)` applies `proc` to each element of `slist`.

```
(slist-map
   (lambda (x)
     (let ([s (symbol→string x)])
       (string→symbol (string-append s s))))
   '((b (c) d) e ((a)) () e)) ➔  ((bb (cc) dd) ee ((aa)) () ee)
```

(b) `(slist-reverse slist)` reverses `slist` and all of its sublists.

```
(slist-reverse '(a (b c) ( ) (d (e f)))) ➔ (((f e) d) ( ) (c b) a)
```

(c) `(slist-paren-count slist)` counts the number of parentheses required to produce the printed representation of `slist`. You must do this by traversing the structure, not by having Scheme give you a string representation of the list and counting parenthesis characters. You can get this count by looking at `car`s and `cdr`s of `slist`).

```
(slist-paren-count '()) ➔ 2
(slist-paren-count '(a (b c) d)) ➔ 4
(slist-paren-count '(a (b) (c () ((d))))) ➔ 12
```

Note: s-lists are always *proper* lists.

(d) `(slist-depth slist)` finds the maximum nesting-level of parentheses in the printed representation of `slist`. You must do this by traversing the structure, and ***not*** by having Scheme give you a string representation of the list and counting the maximum nesting of parenthesis characters.

```
(slist-depth '()) ➔ 1
(slist-depth '(a b c)) ➔ 1
(slist-depth '(a (b c) d))) ➔ 2
(slist-depth '(a (b (c)) (a b))) ➔ 3
(slist-depth '(((a) (( )) b) (c d) e)) ➔ 4
```

(e) `(slist-symbols-at-depth slist d)` returns a list of the symbols from whose depth is the positive integer `d`. They should appear in the same order in the return value as in the original s-list. This one has the basic pattern of the other s-list procedures, but when writing the solution, I found it easier to use a slight variation on that pattern.

```
(slist-symbols-at-depth '(a (b c) d) 2) ➔ (b c)
(slist-symbols-at-depth '(a (b c) d) 1) ➔ (a d)
(slist-symbols-at-depth '(a (b c) d) 3) ➔ ()
```

#2 (15 points) `(group-by-two ls)` takes a list `ls`. It returns a list of lists: the elements of `ls` in groups of two. If `ls` has an odd number of elements, the last sublist of the return value will have one element.

```
> (group-by-two '())
()
> (group-by-two '(a))
((a))
> (group-by-two '(a b))
((a b))
> (group-by-two '(a b c))
((a b) (c))
> (group-by-two '(a b c d e f g))
((a b) (c d) (e f) (g))
> (group-by-two '(a b c d e f g h))
((a b) (c d) (e f) (g h))
```

#3 (20 points) `(group-by-n ls n)` takes a list `ls` and an integer `n` (you may assume that $n \geq 2$). Returns a list of lists: the elements of `ls` in groups of `n`. If `ls` has a number of elements that is not a multiple of `n`, the length of the last sublist of the return value will be less than n.

```
> (group-by-n '() 3)
()
> (group-by-n '(a b c d e f g) 3)
((a b c) (d e f) (g))
> (group-by-n '(a b c d e f g) 4)
((a b c d) (e f g))
> (group-by-n '(a b c d e f g h) 4)
((a b c d) (e f g h))
> (group-by-n '(a b c d e f g h i j k l m n o) 7)
((a b c d e f g) (h i j k l m n) (o))
> (group-by-n '(a b c d e f g h) 17)
((a b c d e f g h))
> (group-by-n '(a b c d e f g h i j k l m n o p q r s t) 17)
((a b c d e f g h i j k l m n o p q) (r s t))
```

#4 (40 points) On pp 20-22 of EoPL, you should have read about `(subst new old slist)`, which substitutes *new* for each occurrence of symbol *old* in the s-list *slist*. *We also wrote this procedure during Session 9 (in Fall, 2015 term; may happen in a different session in a later term).*

Now write `subst-leftmost`, which takes the same arguments (plus a comparison predicate, described below), but only substitutes *new* for the **leftmost** occurrence of *old*. By "leftmost", I mean the occurrence that would show up first if Scheme printed the list. Another way of saying it is "the one that is encountered first in a preorder traversal". Your procedure must "short-circuit", i.e. avoid traversing the `cdr` of any sublist if the substitution has already been done anywhere in the `car` of that sublist. You should only traverse the parts of the tree that are necessary to find the leftmost occurrence and do the substitution, then copy the references to all of the remaining `cdr`s without traversing the sublists of those `cdr`s. Also, ***you must not traverse the same sublist twice*.**
Hint: if your code calls `equal?` or `contains?` or any other procedure that traverses an entire s-list, you are probably violating the "don't-traverse-twice" rule.

In order to make the procedure slightly more general (and easier for me to test the above constraint), *subst-leftmost* will have an additional argument that *subst* does not have. It is an equality procedure, used to determine whether an individual symbol or number in the list matches *old*.

```
(subst-leftmost 'k 'b '((c d a (e () f b (c b)) (a b)) (b)) eq?) ➔
                      ((c d a (e () f k (c b)) (a b)) (b))
(subst-leftmost 'b 'a '(c (A e) a d)
     (lambda (x y) (string-ci<=? (symbol->string x) (symbol->string y)))) ➔ (c (b e) a d)
```

```
(define subst-leftmost   ; substitute new for leftmost occurrence of old in slist (match defined by equality-pred?)
  (Lambda (new old slist equality-pred?)   ; you fill in the rest.
        ... ))
```

Note: Mutation could possibly be used to do in this problem, but I want you to get a bit more practice on purely functional programming, and this problem will certainly give you that practice! It has a lot in common with `bt-max-interior` from a previous assignment.

CSSE 304 Assignment 9 Updated for Spring, 2016

Objectives To experiment with procedural abstraction.

Same rules as the previous assignments, including the prohibition of mutation.

No argument error-checking is required. You may assume that all arguments have the correct form.

#1 (60 points) `snlist-recur`. I have slightly adapted the book's definition of `s-lists` (EoPL, p 8) to allow numbers as well as symbols in the lists.

```
<sn-list>       ::= ( {<sn-expression>}* )
<sn-expression> ::=  <number> | <symbol> | <sn-list>
```

Define a procedure `snlist-recur` that is similar to `list-recur` (written in class), but it returns procedures that work on `sn-list`s. When we call `snlist-recur` with arguments of the correct types, it returns a procedure that

- takes an `sn-list` as its only argument, and
- traverses the entire `sn-list` and its sub-lists, and does the intended computation.

After you write `snlist-recur`, test it by using it to define the functions in (a)-(f). Each of them will have an `sn-list` as one of its arguments.

When you use `snlist-recur` to produce a procedure `f`: in some cases (namely parts b and e), you may need to first write a curried version of `f` (as we did in class when we wrote curried versions of `member?` and `map`) in order to get a procedure that recurs on only one argument (the `sn-list`). Procedures that you pass as arguments to `snlist-recur` **must not be explicitly recursive,** nor may they call `map`, which is a substitute for recursion. All of the recursion on the sn-lists in your solutions should be produced by `snlist-recur` itself. To reiterate, none of your functions in parts (a) - (f) should have any explicit recursive calls.

There are ways to write all of the required procedures without properly using `snlist-recur`. But that would miss the point of this problem. Thus, if you do not use `snlist-recur` as prescribed above, you will not earn any points, even if the grading program says that your code works for all of the test cases.

Note: When your `snlist-recur` procedure is applied to argument(s) of the proper type(s), it must return a procedure that expects exactly one argument (an `sn-list`).

Hint for this problem: `snlist-recur` will probably need to take three arguments (`snlist` refers to an argument passed to the procedure returned by a call to `snlist-recur`):

A base-value to be returned when `snlist` is the empty list.
A procedure to be applied when the `car` of `snlist` is a list (i.e., it is a pair or the empty list).
A procedure to be applied when the `car` of `snlist` is not a list (i.e., it is a symbol or number).

Note: However you design it, when your `snlist-recur` procedure is applied to arguments of the proper types, it must return a procedure that expects exactly one argument (an `sn-list`).

Notes:

1. Your code for these and for the other procedures you write may assume that all of their arguments are the correct type(s). You do not have to do any checking of arguments for validity.
2. Your code for each of the following parts can be fairly short. My longest one is 8 lines long.

(a) `(sn-list-sum snlst)` finds the sum of all of the numbers within `snlst` (which contains no symbols).

```
(sn-list-sum '((2 (3) 4) 5 ((1)) ())) ➔ 15
 sn-list-sum '()) ➔ 0
```

(b) `(sn-list-map proc snlst)` applies `proc` to each element of `snlst`.

```
(sn-list-map (lambda (x) (+ 1 x)) '((2 (3) 4) 5 ((1)) () 5))
  ➔   ((3 (4) 5) 6 ((2)) () 6)
```

(c) `(sn-list-paren-count snlst)` counts the number of parentheses required to produce the printed representation of `snlst`. (You can get this count by looking at `car`s and `cdr`s of `snlst`).

```
(sn-list-paren-count '()) ➔ 2
(sn-list-paren-count '(2 (3 4) 5)) ➔ 4
(sn-list-paren-count '(2 (3) (4 () ((5))))) ➔ 12
```

Note: sn-lists are always *proper* lists.

(d) `(sn-list-reverse snlst)` reverses `snlst` and all of its `sublists`.

```
(sn-list-reverse '(a (b c) ( ) (d (e f)))) ➔ (((f e) d) ( ) (c b) a)
```

(e) (`sn-list-occur s snlst)` counts how many times the symbol `s` occurs in the sn-list `snlst`)

```
(sn-list-occur 'a  '(() a ((a)) a (a a b a) (a a))) ➔ 8
```

(f) `(sn-list-depth snlst)` finds the maximum nesting-level of parentheses in the printed representation of `snlst`.

```
(sn-list-depth '()) ➔ 1
(sn-list-depth '(1 2 3)) ➔ 1
(sn-list-depth '(1 (2 3) 4)) ➔ 2
(sn-list-depth '(1 (2 (3)) (2 3))) ➔ 3
(sn-list-depth '(((3) (( )) 2) (2 3) 1)) ➔ 4
```

#2 (20 points) Recall the following syntax definition from page 9 of EOPL:

```
<bintree> ::= <number>  | (  <symbol> <bintree> <bintree>  )
```

Write a `bt-recur` procedure, similar to the `list-recur` and `snlist-recur` procedures from class and this homework. Calling `bt-recur` produces a procedure that recurs over all of the elements of a `bintree`.

Then use `bt-recur` to create the following two procedures:

- **`(bt-sum T)`** finds the sum of all of the numbers in the leaves of the `bintree` `T`.
- **`(bt-inorder T)`** creates a list of the symbols from the *interior* nodes of T, in the order that they would be visited in an inorder traversal of the binary tree.

The following transcript should help your understand what `bt-sum` and `bt-inorder` do. I do not show the code that was used to construct `t1`.

```
> t1
(a (b 1 4) (c (d 2 5) 3))
> (bt-sum t1)
15
> (bt-inorder t1)
(b a d c)
> (define t2 (list 'e 6 t1))
> t2
(e 6 (a (b 1 4) (c (d 2 5) 3)))
> (bt-sum t2)
21
> (bt-inorder t2)
(e b a d c)
```

Note: As in the `snlist-recur` problems from the previous problem, the definitions of `bt-sum` and `bt-inorder` should not contain any explicit recursive calls. All recursion must be produced by `bt-recur`.

(continued on next page)

3. (25 points) Predefined Scheme procedures like `cadr` and `cdadr` are compositions of up to four `car`s and `cdr`s. You are to write a generalization called `make-c...r`, which does the composition of any number of `car`s and `cdr`s. It takes one argument, a string of `a`'s and `d`'s, which are used like the `a`'s and `d`'s in the names of the predefined functions. For example, `(make-c..r "addddr")` is equivalent to `(compose car cdr cdr cdr cdr)`.

```
> (define caddddr (makec...r "adddd"))
> (caddddr '(a (b) (c) (d) (e) (f)))
(e)
> ((make-c...r "") '(a b c))
(a b c)
> ((make-c...r "a") '(a b c))
a
> ((make-c...r "ddaddd") '(a b c ((d e f g) h i j)))
(i j)
> ((make-c...r "adddddddddddd") '(a b c d e f g h i j k l m))
l
```

I have provided the code for `compose`. For full credit, you should write `make-c...r` in a functional style that only uses built-in procedures, anonymous procedures, and `compose` in the definition of `make-c...r`.

Hint: My solution uses the built-in procedures `map`, `apply`, `string->list`, `list->string`, `string->symbol`, and `eval`; also the character constants `#\c` and `#\r`. You are not required to use these, but you may find them helpful. I do not assume that you are already familiar with all of them; *The Scheme Programming Language* contains info on them; my intention is that you demonstrate an ability to look up and use new procedures.

```
(define compose
  (case-lambda
   [() (lambda (x) x)]
   [(first . rest)
    (let ([composed-rest (apply compose rest)])
      (lambda (x) (first (composed-rest x))))]))
```