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, 4th 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.
```