# 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
```