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.