CSSE 304 Assignment 1 44 points Updated for Fall, 2019-20

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.

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)

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. Many of these will be assigned, but some of them will simply be assigned as thought problems rather than problems to turn in. One reason is that often 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 from other students, the TAs and your instructor, but you should enter the code yourself 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 your focus should be on correctly processing correct arguments.

Indentation and style. Your programs should generally follow the style guidelines in
http://web.archive.org/web/20020809131500/www.cs.dartmouth.edu/~cs18/F2002/handouts/scheme-tips.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 or "orphan closing parentheses" like you'd do with curly braces in C or Java..

Optional 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`.

These uses of square brackets will be demonstrated in numerous in-class examples.

Required comments and procedure order:

- Begin your solution file (name it `1.ss`), with 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 place 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 for some of the assignments those checks will be done by hand later. 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 use the server 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 the test cases.
But the grading program is not the final authority. While we try to be very careful when creating test cases, it is certainly possible that sometimes the problem will with the test cases or answers on the PLC server instead of with your solution. If you suspect that this is the case, send your instructor an email that includes your code.
Also, if you think that the test cases do not cover all of the specifications for a given procedure, feel free to make up your own test cases and post them on Piazza. If the instructors agree with you, we will add your test cases to the server and give you extra credit for providing them.

FOR ALL ASSIGNMENTS, **we 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 the current server 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-Hulman network username and password. Click **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 submitting 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, a `test-cases` file will be provided. That file contains the same test-cases that the PLC grading server uses, so you can test your code off-line instead of repeatedly submitting to the server. Load that file into Scheme, along with your solution code.

Once you have loaded the file into Scheme, you can run the test cases for an individual problem by typing (test-problem-name) . An example from this assignment: (test-interval-contains). Or you can run all of the tests in the file at once by typing `(r)`.

Grading server disclaimer: The grading server and the test cases are made available before assignments are due (as opposed to just having you submit files and hope they work) 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 server tests. You are still responsible for thinking of your own test cases to thoroughly test your code. We reserve the right to add additional test cases when we 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.

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 if your code reads a value from a file.** Note that `let` and `lambda` do not mutate anything; you can use them freely.

Reading Assignment and Thought Problems (be sure to do these!)

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 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.

Some of the EOPL-1 reading duplicates Assignment 0's reading in TSPL, but we 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 all of the textbooks 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 your instructor, the assistants, or other students later.

Programming Problems to turn in There are eight problems. None of them should need loops or recursion.

Reminder: 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.

Background for problems 1-3 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, your code 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".

#1 (5 points) Write a Scheme procedure `(interval-contains? interval number)` where *interval* is an interval and *number* is an integer. The procedure returns a Boolean value that indicates whether *number* is in the closed *interval*.

interval-contains? : *Interval* × *Integer* → *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
```

#2 (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
```

#3 (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' procedures have produced `(1 6)` instead of `((1 6))`.

#4 (3 points) Write the procedures `first`, `second`, and `third` 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:

```
(first  '(a b c d e))  =>  a
(second '(a b c d e)) =>   b
(third  '(a b c d e))  =>  c
```

Most of the remaining problems (and some problems in Assignment 2) 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 represented by the same underlying Scheme type.

> Note that Scheme has a built-in `vector` type and associated procedures to manipulate vectors. Scheme's `vector` type is similar to the `Object[]` array type in Java. 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 Scheme's `vector` type to represent the vector in this problem, but we use lists instead, so that you will 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` ; This is 1*4 + 2*5 + 3*6

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

vector-magnitude: *Vector* → *Number*

Example:

```
(vector-magnitude '(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, the test cases 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
```

Piazza questions and answers from previous terms

For problem4 on 1.ss (interval-union)

When no intersection occurs, can the lists appear in the same order in which they were passed in, or do they have to be ordered by list elements?

Instructor answer: They can be in any order .

Should Questions 5,6,7,8 be treated as SCHEME vectors or lists?

I see that you want us to treat these problems as vectors in the descriptions for each problem, but when I run the tests on the PLC Grader server it seems that you want us to treat these "vectors" as lists. Should I go ahead and treat these vectors as lists so that I can get the questions right?

Austin Derrow-Pinion

Below the red text on the 2nd page of the assignment document he explains how to represent points and vectors. A vector for these problems is just a list of 3 numbers, not the Scheme **vector** datatype.

Vector's Lengths

A student writes

When we are dealing with the vectors can we assume that each list (vector) will have a matching number of components in it? i.e. is there a case where I could have to deal with one vector only having 2 components while the other has 3.

Answer: No, you can assume the input list-vectors will be of the same dimensions. 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.