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.