CSE 341: Homework 1

due 10/4/99

All functions you are asked to implement and turn in should be typed and tested with the scheme interpreter. With each function turn in the source code listing as well as sample output for a demonstrative set of tests. Do not forget to test *end cases*. Use the `turnin` program to turn in an electronic copy of your source code.

Reading:

- Sethi: Chapter 1 (handout).
- *SICP: Chapter 1.*
- R^4RS (handout).

Part I: (Not to be turned in.)

Try these questions out before section. First try to figure out the answers in your head. Then try typing them into the scheme interpreter.

1. Scheme variables and list constructs.

   Given the following definitions:

   ```
   (define x 10)
   (define y 44)
   (define foo '(1 2 3))
   (define bar '(a b c))
   ```

   what are the results of the following pieces of scheme code:

   (a) `x`
   (b) `(cons x foo)`
   (c) `(cons y bar)`
   (d) `(list x y)`
   (e) `(list x foo)`
   (f) `'(x y)`
   (g) `(car bar)`
   (h) `(cdr bar)`
   (i) `(+ x y)`

2. Scheme flow control.

   What are the results of the following pieces of code?

   (a) `(if #f 5 10)`
   (b) `(if (or #t #f)  5 10)`
   (c)
   ```
   (if #t
       (if #f 2 3)
       10
   )
   ```

(d)
```
(define x 3)
(cond ((= x 1)     44)
      ((= x 2)     (+ 5 50))
      ((= x 3)     (* 6 11))
      (else        (- 80 3))
)
```

(e)
```
(let ((x 5)
      (y 10)
     )
     (+ x y)
)
```

Part II: (Not to be turned in.)

These are some sample problems that we will be discussing in section.

1. Define `pi` and a function `circle-area` that computes the area of a circle when given the radius of the circle. For example:

```
pi                              => 3.14159
(circle-area 2)                 => 12.56636
(circle-area 10)                => 314.159
```

2. Define the factorial function `fact`. For example:

```
(fact 0)                        => 1
(fact 4)                        => 24
```

3. Define the constructor and selector functions for an implementation of abstract point data type. In particular, define `make-point`, `x-point`, and `y-point`. For example:

```
(define p1 (make-point 5 10))   => p1
(x-point p1)                    => 5
(y-point p1)                    => 10
```

4. Recursively define a function `positive-only` that takes a list and returns a list with the positive elements in it. For example:

```
(positive-only '())             => ()
(positive-only '(1 2 -3 4))     => (1 2 4)
(positive-only '(-1 -2 -3))     => ()
```

Part III: (Due Monday, 10/4/99. Turn this in.)

Complete the following. You do not have to handle errors in input gracefully; however, if you want to you may use the `error` primitive function; e.g. `(error "error in input")`. Do not forget to turn in a print out of your source code as well as example executions of your code on test cases.

In the example output given below, we abbreviate `make-point` as `mp`. As well, when we show function output, we assume a two element list implementation of points. That is `(make-point 1 2) => (1 2)`.

1. (5 points) Write a function `closer-than?` that takes in two points as defined in Part II, and computes which point is closer to the origin, $(0,0)$. It returns true if the first point is strictly closer to the origin than the second point. False otherwise. For example:

   ```
   (closer-than? (mp 2 0) (mp 8 2))              => #t
   (closer-than? (mp -5 1) (mp 5 -1))            => #f
   ```

2. (5 points) Write a *recursive* function `partition-points` takes as parameters a point, `x`, and a list of points, `lst`. It returns a two item list where the first item is a list of all the points in `lst` that are closer than `x` and the second item is a list with the rest of the points.

   For example:

   ```
   (partition-points (mp 3 0) (list (mp 1 0) (mp 3 0) (mp 4 0)))
             => (((1 0)) ((3 0) (4 0)))
   ```

3. (5 points) Write a recursive function `quicksort-points` that takes as its only parameter, a list of points, and uses `partition-points` and the scheme primitive function `append` to sort the points in increasing distance from the origin.

   For example:

   ```
   (quicksort-points (list (mp 3 0) (mp 2 0) (mp -4 0) (mp 1 0)))
             => ((1 0) (2 0) (3 0) (-4 0))
   ```