CSE 335
Winter 2017 Homework 4

Suggested reading:
   Feb 13 through Feb 15 course lecture slides in particular.
   EOPL 2.1 - 2.3

IMPORTANT:
Always check the tests first to see the full usage of the functions.
Not all use cases are listed in this file.

==
1. [10p]

Let us borrow the "for" expression from Scala Programming language and implement something very similar in Scheme. To this end we will be using the "define-syntax-rule", which was discussed and exercised in class.

An example of what we would like to see is:

```
> (for {a-var <- '(0 1 2 3 4)}
          yield (+ a-var 35)
  )
‘(35 36 37 38 39)
```

--
Syntax:

```
(for {<looping-variable> <- <a-list>}
      yield <expression>
)
```

Everything that is not in between the angular brackets, “<>”, shown above, has to appear as is and in that order.
--
Semantics:
The for loop will bind an element of <a-list> to the <looping-variable>, then it will compute <expression> and it will use the results of every iteration to create a new list of the same length as <a-list>.

Friendly Note:
For some curious mind, the solution can be written as a one liner.

==
2. [20p]

The purpose of this exercise is two-fold: to give you more practice with syntax definitions and to test your understanding of higher procedures. To this purpose, we will be imposing restrictions on what you are allowed to use to create these syntax definitions. You are

allowed to use helper functions, but they too are subject to the same
constraints.

====
2.a [10p]

Using "define-syntax-rule" and only the lambda expression,define a new
syntax abstraction "seq" for sequence that runs twoexpressions one
after
the other.

```
> (seq (print '1) (print '2))
12
> (define x 0)
> (seq (print x) (set! x (+ x 1)))
0
> x
1
> (seq (seq (display x) (set! x (+ x 1))) (display x))
12
```

=====
2.b [10p]

Using "define-syntax-rule" and only the lambda expression, if
expression, and let/letrec expressions, define a new syntax
abstraction "while" for while loops that takes two expressions:
one for ”condition" and one for "body". It runs the body while
the condition remains true.

A transcript that documents some example usage of this expression is
given below.

```
> (define x 0)
> (while (< x 100) (set! x (+ x 1)))
0
> x
100

> (while (< x 105) (begin (display " * ") (set! x (+ x 1))))
 *  *  *  *  * 0
```

Note:
"body" and "condition" can be *any* arbitrarily complex compound
expression, with an arbitrarily high level of nesting. As you can see,
the while loop outputs the integer value 0 (zero) once terminates.

==
BNF grammar overview.

Recall that we explained in class the definition of list data-type
using a BNF grammar:

```
<list> ::=  null
         | <data> <list>

<data> ::=  number
         | string
         | procedure
         | symbol
         | <list>
```

In plain english: here a list is defined to be either null or a pair
of data and another list. Data can be either a number, a string, a
procedure, a symbol, or even a list.

The alternatives shown in the grammar rule to define a data type are
called "variants". Here, null is a variant of list. (How many
variants does the list data type have? What is the other variant in
addition to null? Look at the parts of the grammar above that defines
<List> ?)

The vertical bar ”|” means "or", while the terms enclosed in angular
brackets, <>, are called non-terminals and the ones that are not are
called terminals.

A Non-terminal can be substituted for any valid value of that
non-terminal. For instance, take the second definition of the <list>:

```
   <data> <list>
```

Here <data> could be substituted for the value number, and <list> for
null, which then effectively describing a list that contains only one
number.

==
=
3. [30p] Step

Consider the following data type that describes the kinds of movements
that can be taken on a two dimensional surface:

```
<step> ::=  <step>  <step>       "seq-step"
         | "up" number          "up-step"
         | "down" number        "down-step"
         | "left" number        "left-step"
         | "right" number       "right-step"
```

"up", "down", "left" and "right" are only markers in this BNF grammar spec that help differentiate between two steps. These are not required to show up as strings in an implementation, although they could.

"seq-step", "up-step", "down-step", "left-step" and "right-step" at the end of each grammar production for <step> are symbolic names we use to identify each variant.

3.a [15p]

For each variant of the <step> datatype you should implement the following types of functions:

- a constructor bearing the name of the variant. Constructors take any relevant data as parameters and return a new value of that type variant.
- a predicate with the name of the variant followed by a "?" mark
- extractors for each piece of data of the variants. e.g. we write one extractor for up, down, left, right steps and two for seq-step.

Predicates and extractors are called observers. And we will be encountering this pattern quite often in the future.

The signatures of the above functions are already written down in the answer sheet. You should take the time to see which procedure falls into which category and infer their behavior from the test cases.

SO - PLEASE read the test code carefully and thoroughly.

A few general rules to keep in mind:

- constructors throw exceptions when they receive invalid input
- predicates never throw exceptions, they either return #t or #f
- extractors may throw exceptions, for example, when they receive invalid input

---
3.b [15p]

Implement the function:

```
              (move starting-point step)
```

Input:

- starting-point: a point in the x,y coordinate system, represented as a 2 element list
- step: a step, as defined in 3.a

Output:

- the end-point after moving the specified number of steps

```
> (move '(0 0) (up-step 35))
'(0 35)
;we've moved up on the y axis 35 steps

> (move '(0 0) (left-step 35))
'(-35 0)
;we've moved to the left on the x axis by 35 coordinates
```

==
==
==

NOTE:
In the following part of this homework you will be using ***procedural representation*** to implement the data-types. Please see Feb 15 lecture slides and EOPL 2.1 ~ 2.3 for more details.

The following is the TOP Guideline for implementing data-types as procedures/functions:

What procedural representation isn’t :

- a list containing functions that simply return each individual element of the list
- if you use lists for anything but temporary computation, such as using (range n) for a map/fold/andmap operation, you are probably doing it wrong.

===
===
===

4. [20p] Functional Sets

You will be using procedural representation to implement sets of numbers. Each set is a one argument procedure/function that takes a number as argument and tells whether or not the number is in the set (that is, returns #t or #f).

Hints:

- the procedures/functions up until "exists?" can be written as one liners and do not require recursion or other functional operations to solve.
- just because a procedure/function has the word "map" in it, it doesn't mean you have to use map higher order function to implement it.
- think about what it means for an element to be in a set, rather than how to implement it. (Again, “what” rather than “how” !)

===

5. [20p] Step revisited

Remember the "step" data-type in problem 3 of this homework:

```
<step> ::=  <step>  <step>       "seq-step"
          | "up" number          "up-step"
          | "down" number        "down-step"
          | "left" number        "left-step"
          | "right" number       "right-step"
```

5.a
Re-implement it using procedural-based representation(see Feb 15 course slides). As you can notice from the test file, they are the same, by definition, the look and behavior of "interfaces" should not change depending on the underlying implementation.

5.b
Implement the function:

```
              (move starting-point step)
```

Input:

```
  - starting-point: a point in the x,y coordinate system, represented
                    as a 2 element list
  - step:           a step, as defined in 4.a
```

Output:

```
  - the end-point after moving the specified number of steps
```