CSE 335
Winter 2017 Homework 05

Suggested reading:
EOPL 2.4

This homework introduced you to "define-datatype", a tool available in the "eopl" library (meaning it comes with your textbook, and its not part of the standard scheme language).

Essentially, define-datatype is a more sophisticated way that allows you to easily define recursive datatypes, it automatically generates:

- a constructor for each variant
- one predicate for the top level datatype, e.g. Env?, it doesn't generate one predicate per variant.

```
> (#%require (lib "eopl.ss" "eopl"))


;define-datatype takes two top level parameters, its detailed
structure is explained below:
> (define-datatype name-of-datatype name-of-predicate
     ;each variant is described by a list of field-name predicate
     ;pairs
     ;here f1, f2 have to be numbers
     ;f3 is a list-of numbers, please refer to the "list-of" function
     ;from the "eopl" library for more information (you can look it
     ;up in the online racket index)
    (variant1 (f1 number?)
              (f2 number?)
              (f3 (list-of number?))
             )

  ;as you can notice, the fields may be named as you please, and you
  ;can use custom predicates to describe them
  (variant2 (i-can-give-the-fields-whatever-name-I-want string?)
            (n-or-s number-of-string?)
            ;it is possible to use the predicate of the data-type we
            ;are currently defining
            (field-of-this-datatype name-of-predicate)
            )

   (empty-variant)
  )

(define (number-of-string? x)
  (or (number? x) (string? x))
  )
```

The role of extractors and predicates is taken over by the "cases", syntactically it resembles a "cond" expression:

```
(define (fun adt)
  (cases name-of-datatype adt

    (variant1 (f1 f2 f3) #| expression that can use f1 f2 f3, each of
                            which are bound to the values of the three
                            fields of this variant |#
              (+ f1 f2))

    (variant2 (dont-have-to-use-same-name-as-in-the-definition
               but-it-is recommended) 42)

    (else (raise "If you leave out any variants from the cases
             expression then the else clause has to be present"))
    )
  )
```

Please read section 2.4. very carefully as define-datatype is specific to our textbook (EOPL).

NOTE: Please experiment with "define-datatype" and "cases" thoroughly because we will be building our interpreter using these two expressions extensively.

IMPORTANT:
The test cases will not compile until you implement your datatype using "define-datatype"; currently all the test cases are commented out.

===

1. [40p] Step revisited, again

Recall the <step> data-type:

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

1.a
Re-implement the interface using define-datatype.

1.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 1.a

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

For this exercise you *are required* to use the "cases" expression in the implementation of move. From here on out, when using define-datatype, we will no longer write entire interfaces unless necessary. The cases expression will do the job that was previously assigned to predicates and extractors.

==

2. [60p] Define-datatype implementation of environment

Recall the <environment> data type described in the lecture slides and in your textbook:

```
<environment> ::=
                   "empty-env"
                 | "extend-env" symbol value <environment>
```

(empty-env)
  initializes an environment

(extend-env)
  adds a new mapping from "symbol" to "value" in the old environment. If the same symbol is added twice, then the previous mapping is not replaced, it is only shadowed.

This data-type is accompanied by the function:
(apply-env env sym)
  this function will return the "value" mapped to "sym" in the given environment; it raises an exception if no such mapping exists.

2.a [20p]
  Implement the environment using define-datatype.

2.b [40p]

Implement a procedure (extend-env-wrapper sym val old-env final?)
Compared to the constructor "extend-env" it has a boolean parameter "final?" indicating whether or not the new mapping should be shadow-able.

Whenever an environment is extended using the "final" variant, the symbol from its corresponding mapping cannot be shadowed, any attempt to shadow the value of a "final" symbol will result in an error.