CSC344 – Assignment 2

Microproject

Write a Clojure function which takes as input an unevaluated arbitrarily deep nesting of lists. The function will determine if any item in the list, not in function position, is non-numeric.

Main Project

Write a set of Clojure functions that perform symbolic simplification and evaluation of an expression computing a series of affine transforms of a point.

An affine transform is a transformation of points such that ratios of distances between points is preserved along with collinearity (points on a line before the transformation are still on a line after). Such a transformation in two dimensional space can be represented as a 2×2 matrix. The transformation of a point is calculated by multiplying this matrix with a 2×1 matrix containing the original x and y coordinates of the point, as follows:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

See the first 15 slides of this lecture (https://www.cs.utexas.edu/~fussell/courses/cs384g-fall2012/lectures/lecture07-Affine.pdf) from Texas A&M for further background and pictorial examples.

In Clojure a transform matrix will be represented as a vector of arity two, both of whose elements are arity 2 vectors.

Expressions representing application of one or more of these transformations are created as unevaluated lists. For example:

```
(def p1 '(transform [[a 3] [0 0]] [x y]))
(def p2 '(transform [[1 0] [0 (+ x 3)]] [(* x 2) y]))
(def p3 '(transform [[0 0] [1 1]]
           (transform [[2 0] [0 2]]
             (transform [[-1 0] [0 -1]] [x 2]))))
```

Any non-numeric value appearing in the transform matrix or the point should be considered a variable.

Your goal is to maximally simplify and evaluate these expressions. In some cases this will result

in a concrete numeric answer, as in p3 above, where the following simplification occurs:

```
(transform [[0 0] [1 1]]
  (transform [[2 0] [0 2]]
    (transform [[-1 0] [0 -1]] [x 2])))
=>
(transform [[0 0] [1 1]]
  (transform [[2 0] [0 2]] [(- x) -2]))
=>
(transform [[0 0] [1 1]] [(* (- x) 2) -4])
=>
[0 -4]
```

In other cases, your final result may not be fully evaluable and will contain variables.

In the process of writing your program you will need to simplify a limited set of mathematical expressions (those of arity 2 with * and +, and arity 1 with -). You will implement at least the following simplification rules for these expressions:

```
(* 1 x) => x
(* x 1) => x
(* 0 x) => 0
(* x 0) => 0
(+ 0 x) => x
(+ x 0) => x
(- (- x)) => x
```

You will also implement the evaluation of the affine transforms, using the above simplifications. Some examples are shown below.

```
(transform [[1 0] [0 1]] [x y]) => [x y]
(transform [[2 0] [0 1]] [x y]) => [(* 2 x) y]
(transform [[2 0] [0 1]] [2 y]) => [4 y]
(transform [[z 0] [0 1]] [2 y]) => [(* z 2) y]
(transform [[2 0] [0 1]] [(+ x 5) y]) => [(* 2 (+ x 5)) y]
```

Your program will also allow binding of some or all of the variables in the expression with constants (numbers) before simplification and (partial) evaluation. To do so, you should make use of the substitution functions discussed in class.

The main entry point of your program, `evalexp`, will call functions that simplify, bind, and evaluate the input expression.

The `evalexp` function should take a symbolic expression and a binding map and return the simplest form. One way to define this is

```
(defn evalexp [exp bindings] (simplify (bind-values bindings exp)))
```

Example:

```
(evalexp p1 '{a 5, y 2}) ;; p1 is '(transform [[a 3] [0 0]] [x y])
```

binds a and y (but not x) in p1, leading to `(transform [[5 3] [0 0]] [x 2]))` and then further simplifies to `[(+ (* 5 x) 6) 0]`.

Note: You may not use eval

Suggested Development Environment

Eclipse + Counterclockwise (http://doc.ccw-ide.org/documentation.html)
