Assignment 2

Microproject

Examine the below main project. Write one or more Clojure functions which perform symbolic simplification on (un-nested) expressions using the `and` logical connective. Be sure to generalize to all possible variables and valid numbers of arguments to and. For example:

```
mp=> (and-simplify '(and true))
true
mp=> (and-simplify '(and x true))
x
mp=> (and-simplify '(and true false x))
false
mp=> (and-simplify '(and x y z true))
(and x y z)
```

Main Project

Write a set of Clojure functions that perform symbolic simplification and evaluation of boolean expressions using `and`, `or`, and `not`. `not` will be assumed to take one argument, while `and` and `or` will take one or more. You should use `false` for False and `true` for True.

Expressions are created as unevaluated lists. Three sample expressions could be defined as follows:

```
(def p1 '(and x (or x (and y (not z)))))
(def p2 '(and (and z false) (or x true)))
(def p3 '(or true a))
```

You could also define functions to build unevaluated lists for for you, such as:

```
(defn andexp [e1 e2] (list 'and e1 e2))
(defn orexp  [e1 e2] (list 'or e1 e2))
(defn notexp [e1] (list 'not e1))
```

Using these, `p3` could have been created using

```
(def p3 (orexp true 'a))
```

If you wish to use these in your project, you will need to modify `andexp` and `orexp` to allow for one or more arguments.

The main function to write, `evalexp`, entails calling functions that simplify, bind, and evaluate these expressions.

Simplification consists of replacing particular forms with equivalent functions, including the following:

```
(or true) => true
(or false) => false
(and true) => true
(and false) => false
(or x false) => x
(or false x) => x
(or true x) => true
(or x true) => true
(or x y true) => true
(and x false) => false
(and false x) => false
(and false x y) => false
(and x true) => x
(and true x) => x
(not false) => true
(not true) => false
(not (and x y)) => (or (not x) (not y))
(not (or x y)) => (and (not x) (not y))
(not (not x)) => x
```

You should generalize for any length expression based on these patterns. Your program must work for any arbitrary variables used. As in the microproject, you may wish to write functions to handle certain kinds of cases, and handle the non-recursive case before you handle the recursive one.

Binding consists of replacing some or all of the variables in expressions with provided constants (`true` or `false`), and then returning the partially evaluated form.

The evalexp function should take a symbolic expression and a binding map and return the simplest form (that might just be a constant). One way to define this is

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

Example:

```
1 (evalexp p1 '{x false, z true})
```

binds x and z (but not y) in p1, leading to `(and false (or false (and y (not true))))` and then further simplifies to just false.
