# CSE 341, Autumn 2008, Assignment 8
# Ruby Project
# Due: Monday December 1, 10:00pm

(updated Nov 21)

25 points total (5 points for Question 1, 20 points for Question 2)

You can use all the rest of your late days for this assignment.

This assignment is intended to give you experience with inheritance in Ruby, mixins, closures, integrating with basic system classes such as integers, and other good object-oriented stuff. It consists of two questions: one that is about defining better versions of binary and n-ary trees in Ruby, and the other that is about an object-oriented version of the symbolic differentiation program you worked on for Scheme. Put your tree and differentiation classes in one file (say `assign8.rb`) and your unit tests in another (say `assign8_tests.rb`).

1. The binary and n-ary trees in the Ruby warmup assignment can be written much more cleanly using mixins. Rewrite `Leaf`, `BinaryNode`, and `NaryNode` to use the `Enumerable` mixin. The only methods you need in your new classes is `initialize` and `each`. With the mixin and the appropriate `each` method, the tree classes will have not just `min` and `max`, but also `sort`, `collect`, `find`, `reject`, and other useful methods.

   In a separate file of unit tests, include unit tests that test all of the methods you implement for `Leaf`, `BinaryNode`, and `NaryNode`. In addition, include unit tests of `min` and `max` from the `Enumerable` mixin.

   Hint: the `each` method for `Leaf` is straightforward. However, for `BinaryNode` and `NaryNode`, you'll need to convert from blocks to Procs and back. There is an example of doing this linked from the 341 Ruby web page.

2. Implement a version of the Scheme symbolic differentiation program in Ruby. Your program should be able to differentiate symbolic expressions that involve variables, constants, and sums and products of other symbolic expressions, just as in the original Scheme program. There is a sample file of unit tests linked from the 341 assignments page that you can use. These should be enough tests, unless you do the extra credit assignment. Note that you need to define a class `Variable` to hold symbolic variables, and to integrate it with Ruby's integer and float classes so that it works properly in expressions. (See the unit test examples for the syntax.)

   Recommended solution path: don't try to write the entire program at once and then start testing it — instead, take small steps, testing as you go. It is probably easiest to get the symbolic differentiation program working without integrating it with the existing numeric Ruby classes first, and then hook in with those. (That's the path suggested here.)

   - First, define a class hierarchy for symbolic differentiation. Here are some suggested classes to implement. It's OK to do this differently, as long as it's a clean object-oriented design.

     ```
     class SymbolicExpression
     class Variable < SymbolicExpression
     class ConstantHolder < SymbolicExpression
     class BinaryOperation < SymbolicExpression
     class Addition < BinaryOperation
     class Multiplication < BinaryOperation
     ```

     `SymbolicExpression` is an abstract class that serves as the superclass for other symbolic expression classes. A `Variable` should have an instance variable `name`, which is a string. A `ConstantHolder` should hold an integer or float. (You can also just use integers and floats

directly in symbolic expressions. The reason for `ConstantHolder` is that it will come in handy later when you write code to support expressions like `3*x`, in which you coerce integers or floats into something that can interoperate with other symbolic expressions. This is not terribly elegant, but it works OK in my sample solution. If you come up with a better way to handle this feel free to use it instead.) The other classes should be self-explanatory.

All expressions should understand the following messages: `basic_deriv`, `simplify`, `to_s`, and `==`. `basic_deriv` takes a `Variable` as an argument, and returns the derivative of the expression with respect to that variable (not simplified). `simplify` takes no arguments, and returns a simplified version of the expression, using the same rules as in the Scheme program (0+x simplifies to x, and so forth). The reason for implementing `simplify` separately, rather than combining it with the `initialize` method, is that simplifying an expression might return an instance of a different class.

Also define the following methods in `SymbolicExpression`, so that they are inherited by the other classes:

```
def inspect
  return to_s
end
def deriv(v)
  return self.basic_deriv(v).simplify
end
```

Implementing `inspect` will be useful for debugging, since your expressions will print out in a more readable way. `deriv` takes the derivative and then simplifies the result — you only need to implement this method once in `SymbolicExpression`.

You should now be able to test that `to_s`, `basic_deriv` and `simplify` are working on a few simple test cases. However, it's tedious to write test cases at this point, since you have to construct expressions by hand, e.g.

```
x = Variable.new("x")
s = Addition.new(x,3)
t = Multiplication.new(10,s)
# now try:
# s
# t
# s.basic_deriv(x)
# s.deriv(x)
# t.basic_deriv(x)
# t.deriv(x)
```

So save the bulk of the testing for after you get the next part working.

- Add implementations of `deriv`, `basic_deriv`, and `simplify` to the built-in class `Numeric`. (Then both integers and floats will inherit them.)
- Now get symbolic expressions to interoperate correctly with integers and floats. To do this, add the following methods to `SymbolicExpression`:

```
def + (other)
  return Addition.new(self,other)
end
def * (other)
  return Multiplication.new(self,other)
end
def +@   # unary +
```

```
  return self
end
def -@   # unary -
  return Multiplication.new(-1,self)
end
```

Presto! Expressions like `x+2` and `-x` now work! But what about `2+x`? That ought to work as well, but here 2 is getting the message + with a `SymbolicVariable` as an argument ... and it's not sure what to do. I'm not going to give the answer away for this — instead, have a look at the RomanNumeral example in the *Programming Ruby* book and see what they did there.

**Extra Credit** (max 10% extra):

- The `to_s` method as tested in the unit tests puts parentheses around expressions to handle operator precedence in a straightforward way. But the result is more complicated than need be. Change `to_s` to avoid all unneeded parentheses.
- Also support differentiation of expressions involving `**`, `sin`, and `cos`, including suitable simplification rules. Since the way Ruby handles `sin` and `cos` is not very object-oriented, fix this while you are at it, so that these are messages to numbers and to symbolic expressions. There is already a disabled unit test for `sin` and `cos` (change the name to enable it).

**Turnin:** Turn in your two files, one with the tree and symbolic differentiation program, the other with your unit tests. Clearly separate out your unit tests for trees from those for symbolic differentiation by making a separate test class — see the starter code for directions. You can add some more tests for differentiation if you want, but unless you do the extra credit assignment you don't need to.

You don't need to turn in a script showing your program running — the TA's can just run the unit tests for that.