CSE 341, Winter 2010, Assignment 8
Ruby Project
Due: Monday March 8, 10:00pm

25 points total

You can use up to 4 late days for this assignment.

This assignment is intended to give you experience with inheritance in Ruby and integrating with basic system classes.

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, differences, and products of other symbolic expressions, just as in the original Scheme program.

Put your differentiation classes in one file named `assign8.rb` and your unit tests in another file `assign8_tests.rb`. You can just download the unit tests from the 341 web page and use those. (You can add additional tests if you like but it's not necessary.) For the unit tests to work, you'll need to define a class `Variable` to hold symbolic variables, and to integrate it with Ruby's Numeric class so that it works properly in expressions. (See the unit tests 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 BinaryOperation < SymbolicExpression
  class Addition < BinaryOperation
  class Subtraction < 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. 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 check that to_s, basic_deriv and simplify are working. 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 (including trying to run any of the unit tests) 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 understand + and - and *, and also 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 Subtraction.new(self,other)
end
def * (other)
  return Multiplication.new(self,other)
end
def +@   # unary +
  return self
end
def -@   # unary -
  return Subtraction.new(0,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.

One way to make this work is to implement a class `ConstantHolder` that holds an integer or a float, and that defines + and `-` and * to also return an `Addition`, `Subtraction`, or `Multiplication`. Also implement a `coerce` method in `SymbolicExpression` that returns an array with a `ConstantHolder` containing the argument, and self. In this approach I didn't make `ConstantHolder` a subclass of `SymbolicExpression`, so it doesn't understand `deriv`, `simplify`, and so forth — its only purpose is as a temporary object to allow `Addition`, `Subtraction`, and `Multiplication` objects to be created. This isn't terribly elegant but it works — needing to do something like this is an artifact of how Ruby handles coercion for numbers. If you think of a nicer way to do this you can use that instead.

See the Ruby documentation, and also try `coerce` on integers and floats to see how coercion works in Ruby. The RomanNumeral example in the printed *Programming Ruby* book provides an additional example.

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 are already commented-out unit tests for exponentiation, `sin`, and `cos` in the unit test file.

Turnin. Turn in both the `assign8.rb` and `assign8_tests.rb` files. (You may well have not modified `assign8_tests.rb` — that's OK.) You don't need to turn in a script showing your program running — the TA's can just run the unit tests for that.