CSCI 3353 Object Oriented Design

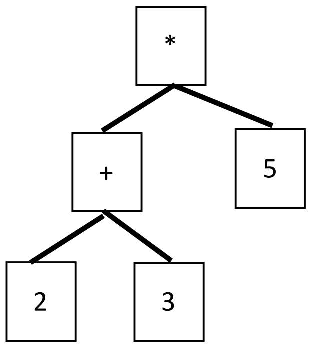
Homework Assignment 7

Due Monday, November 7

An arithmetic expression is naturally represented as a tree. The leaves of the tree are numbers, and the internal nodes are operators. For example, the arithmetic expression

```
(2 + 3) * 5
```

corresponds to the tree



A polynomial is a generalization of an arithmetic expression, in which a leaf node can also be a variable. For example the polynomial

```
(x + 3) * y
```

has the same representation as the above tree, except that the nodes for 2 and 5 are nodes for x and y.

In this assignment I want you to write code to construct and manipulate polynomials. Here are the basic requirements:

- You only need to support integer values.
- You need to support three operators: PLUS, MINUS, and PRODUCT. These are binary operators – that is, their nodes in an expression tree will always have exactly two children.
- An operator's children can be any polynomial. In particular, a child could be an integer, a variable, or another operator.
- Values and variables are implemented in their own class, as are each kind of operator. These classes should be named *Number*, *Variable*, *PlusOp*, *ProductOp*, and *MinusOp*. Each class should implement the interface *Polynomial*.

The interface *Polynomial* should look like this:

```
public interface Polynomial extends Iterable<Polynomial> {
  public String toString();
  public boolean equals(Polynomial p);
  public int evaluate(Map<String,Integer> m);
  public Polynomial reduce();
}
```

Your job will be to implement these methods for the five classes that implement *Polynomial*, so here is a brief explanation of what they do.

toString

The method *toString* returns an infix representation of the polynomial. Recall that an infix expression may require parentheses. I'm not fussy about how you do the parenthesization, as long as the string is meaningful and is equivalent to the tree. You don't have to worry about operator precedence if you don't want to, and I don't care if the expression has extra, redundant parentheses.

equals

The method *equals* tests to see if the two polynomials have the same structure, taking into consideration the fact that addition and multiplication are commutative. For example, the following polynomials are all equal:

```
(x + 3) * y
(3 + x) * y
y * (x + 3)
y * (3 + x)
```

You should not use arithmetic to determine equality. For example, the following polynomials are not equal to the above ones:

```
(x + 3) * (y + 0)
(x + (2 + 1)) * y
```

evaluate

The method *evaluate* evaluates the polynomial and returns an integer. The specified map provides a value for each variable; the variable name is the key, and the variable's value is the value. For example, consider the polynomial (x + 3) * y. If I evaluate it using the mapping [x=6, y=7], the answer is 63; if I evaluate it using the mapping [y=8, z=1, x=2] the answer is 40. It is ok for the map to have values for unmentioned variables. But if the map doesn't have a value for a mentioned variable, then the evaluation method has no choice but to throw an exception. Note that if the polynomial has no variables, then the *evaluate* method will return the same value regardless of what map is provided.

reduce

The method *reduce* does "partial evaluation" of the polynomial. That is, it evaluates the polynomial as much as it can without using a map, and returns the resulting polynomial. For example, the following polynomials all reduce to (x + 3) * y:

```
(x + ((2 * 2) - 1)) * y
(x + 3) * (y + 0)
((x * 1) + 3) * ((y + 0) + (z * 0))
((x + 3) * y) + ((z + x) – (x + z))
```

The first example demonstrates that reduction evaluates all arithmetic expressions within the polynomial. The second and third examples demonstrate that reduction can make use of the special properties of 0 and 1. The fourth example demonstrates that if p and q are polynomials such that p.equals(q) is true, then reduction should transform the polynomial (p – q) to 0.

If a polynomial *p* has no variables, then *p.reduce()* returns a polynomial having a single node, whose value is what gets returned by calling *p.evaluate*.

Note that the method *reduce* should not change the existing polynomial. Instead, the returned polynomial should consist entirely of newly created nodes.

iterator

The interface *Polynomial* extends *Iterable*, which means that you also need to write a method *iterator().* If the polynomial is an operator node, then its iterator returns its children. If the polynomial is a leaf node (i.e. an integer or a variable), then the method should return an empty iterator, and NOT the value *null*.

WHAT YOU SHOULD DO

1. Draw a class diagram corresponding to your design for these classes. Use the composite pattern.

2. Write the Java code to implement the interface *Polynomial* and its five classes. Your code is allowed use the **instanceof** operator to implement the methods *equals* and *reduce*, but nowhere else. The constructor of each operator class should have two arguments, denoting the children of the operator node. That is, you don't need an *add* method.

3. Write a paragraph briefly describing why the **instanceof** is useful for implementing the *equals* and *reduce* methods. Suppose that I had said that you couldn't use **instanceof** to implement *reduce*. Describe an alternative implementation of *reduce* that would require modifying the *Polynomial* interface.

4. Write a default method for the *Polynomial* interface called *traverse*, having the following signature:

```
public default void traverse(Consumer<Polynomial> c)
```

Your code should be very similar to the *traverse* method in the *MenuComponent* interface of the composite3 demo.

5. Write a program, called *HW7Test.java*, that uses your code. In this program you should perform the following tasks:

- Write a static method *getVarsExternal*, having the following signature:

  ```
  public static Set<Variable> getVarsExternal(Polynomial p)
  ```

  This method should use the polynomial's *iterator* method (explicitly or implicitly, your choice) to traverse the tree and return a set containing the variables it finds. (The reason for using a set is that it removes duplicates automatically.)

- Write a static method *getVarsInternal*, having the same signature and returning the same output. The difference is that it should perform internal iteration, using the *traverse* method that you wrote for question 4.

- Construct objects corresponding to the two polynomials:

  ```
  ((x * 1) + 3) * ((y + 0) + (z * 0))
  ```

```
((x + 3) * y) + ((z + x) – (x + z))
```

For an example of how I want you to construct polynomials, download the file *SimpleHW7Test.java* from Canvas. That code creates an object corresponding to the polynomial (x+3)*y and prints it.

- For each polynomial:
  - Call *toString* to print it.
  - Call *evaluate* to evaluate it using the map [x=3, y=4, z=5], and print the result.
  - Call *reduce* to reduce it, and print the reduced polynomial.
  - Call *getVarsExternal* to get its set of variables, and then print those variables.
  - Call *getVarsInternal* to get its set of variables, and then print those variables.

WHAT TO SUBMIT: You will need to create the following files:

- A document containing your answers to questions 1 and 3.
- The six files *Polynomial.java*, *Number.java*, *Variable.java*, *PlusOp.java*, *ProductOp.java*, and *MinusOp.java*, as described in questions 2 and 4.
- The file *HW7Test.java*, as described in question 5.

If your Java files use packages, you should strip off the package designations. Then create a zip file containing these files and submit it.