University of Toronto
CSC150—Accel. Intro. to Computer Science, St. George Campus, Fall 2007

Assignment 4: Expression Trees

Due: 10:10am (at the **beginning** *of class), Mon, Dec. 3*

Introduction

A binary tree provides a natural way to represent arithmetic expressions. Here we consider simple forms of expression trees formed from variable names, integers, the four binary arithmetic operators +, -, * and /, and the two unary sign operators + and -. Three sample expression trees are given below.

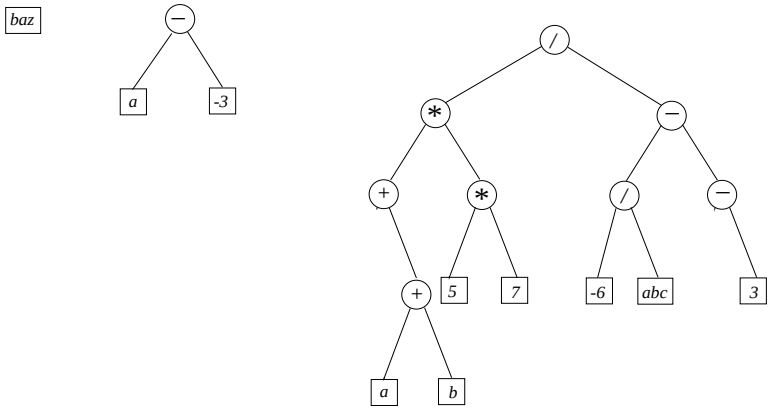

Figure 1: Examples of arithmetic expression trees.

The internal nodes of an expression tree contain any one of the binary or unary arithmetic operators, and these always have one or two children (if the tree has height larger than one, then the root node is also one of these arithmetic operators). The leaves of an expression tree are either variables or integers.

We will input arithmetic expressions as text strings. In this assignment we assume that the expression in the text string is fully parenthesized, and is provided on a single line. For example, suitable input for the above expression trees are:

```
baz
(a   - -3)
(  ( (  + (a + b)) *  (5 *  7  ) ) / (  ( -6 / abc) - ( - 3)))
```

A text expression is "fully parenthesized" when each arithmetic operator and its operands are surrounded by their own (unique) pair of parentheses (...). Spaces in these text expressions do not matter, except between a unary plus/minus and an integer. If there is no space, as in $+3$ or -6, then these strings represent positive and negative integers. Alternatively, if there is a space, as in $-\ 3$, then this denotes the unary minus operator applied to the positive integer 3.

Handout Code

The starter code provides a `Parser` class, in package `a4`, which reads text expressions and returns an ordered sequence of String tokens. The tokens denote integers, variables, and distinct strings for the six different operators. You can view the output from the parser by typing the following into the interaction pane:

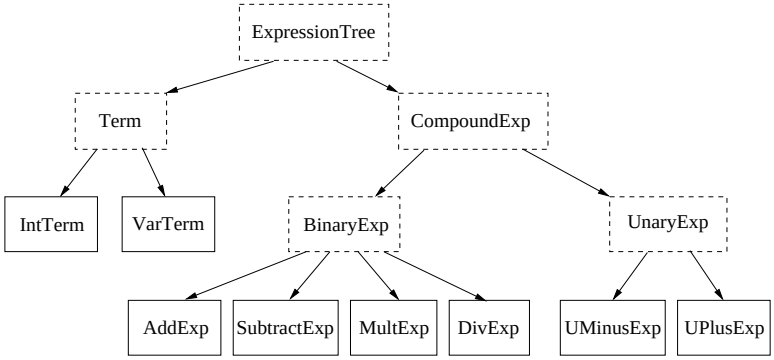

Figure 2: Class hierarchy of Java implementation. Dashed boxes indicate abstract classes.

```
import a4.Parser;
String line = "((( +(a+b)) *  (5 * 7)) / (( -6 / abc)-(- 3)))";
Parser p = new Parser(line);
while (p.hasNext())
  System.out.println(p.next());
```

For example, in the Parser output a binary add operation is denoted by the string "@b+", and a unary minus operation is denoted by "@u-".

In addition, the handout code includes the SymbolTable class definition. This class is used to store variables which are used in expressions along with their values. In this assignment all variables have Integer values. The following should work in the interaction pane:

```
import a4.SymbolTable;
SymbolTable sym = new SymbolTable();
sym.set("abc", -6);
sym.set("a", 3);
sym.set("b", 5);
sym.set("abc", 66);  // Resets abc to 66
sym.get("abc")   // returns Integer(66)
System.out.println(sym.toString());
// prints [a: 3, abc: 66, b: 5]
```

In order to evaluate an expression which contains variables, the eval(sym) method must be passed a symbol table including (at least) all the variables in the expression.

The classes we will use to implement expression trees are arranged in a hierarchy, as depicted in Fig. 2. Here dashed boxes indicate abstract classes, and solid boxes indicate normal classes. These classes are only partially implemented in the hand-out code, and they won't compile. If you try to compile them you will see many errors caused by the fact that various abstract methods have not yet been implemented in the non-abstract classes.

Completing the ExpressionTree

Vital Rule 1. You must make appropriate use of the class hierarchy depicted in Fig. 2, with each class implementing the appropriate type of item. For example, any internal node in the expression tree must be an instance of the `CompoundExp` class, and any leaf must be an instance of `Term`. You can change the contents of these class definitions by adding import statements, adding new variables and methods, subject to the following restrictions:

- Your class hierarchy is still as depicted in Fig. 2.
- You cannot add any new classes at all (except for your testing code, which will not be handed in).

- You cannot change any of the implemented methods already present in the hand-out code. (Except you can make any abstract methods non-abstract, if you choose.)
- You cannot change anything in package `a4`.

Vital Rule 2. You must use recursion instead of loops in all of the code you write (except in your testing code).

Style Rule. Think carefully about where to implement the abstract methods in ExpressionTree, perhaps by using helper methods. Two rules of thumb are:

- Try to keep the use of the "instanceof" keyword to a minimum.
- Try to avoid duplicating code across several classes. Instead, try to achieve the same functionality using one method placed higher up in the class hierarchy.

Question 1. Write a static method:

```
public static ExpressionTree createTree(Parser parser)
```

in ExpressionTree.java which builds the expression tree corresponding to the tokens provided by the parser. In particular, when the parser is obtained from `new Parser(line)` with a suitable input line (such as one of the examples provided above), then your createTree(Parser) method should construct the corresponding ExpressionTree. Here, to make your task easier, assume that the input is fully parenthesized. If there is an error on the input, createTree should throw a MalformedExpressionException (defined in package a4).

Hint. This, unfortunately, is the most difficult method to write and the rest of the assignment all depends on this method. To get yourself started, first try writing a version which only allows the binary plus operator and only integer constants to be used in the expression. In order to get this initial classes to compile, you can comment out most of the abstract methods in ExpressionTree, and remove all of the classes you don't need from DrJava (eg. VarTerm, UnaryExp, and any of the specific arithmetic operator classes other than AddExp).

Once this trimmed down version appears to be working, you can extend it to include unary minus operations. Given such a partial implementation, you can try implementing some of the methods described below before returning to this one. (Obviously, be careful to submit your most complete version of ExpressionTree.java. We do not want to see these initial attempts.)

Question 2. Complete the implementation of toString() in ExpressionTree (it could still be abstract in ExpressionTree.java, but it must be implemented somewhere in the class hierarchy). Your toString() method must operate recursively by generating String representations for sub-expressions within your expression tree, and not use some other algorithm. The detailed form of the output is illustrated by the following example (perhaps run in the interaction pane):

```
import a4.Parser;
String line = "(((+(a+b))*(5*7))/((-6/abc)-(-   3)))";
Parser parser = new Parser(line);
ExpressionTree t = ExpressionTree.createTree(parser);
t.toString();
```

then the first line below should be output:

```
``(((+ (a + b)) * (5 * 7)) / ((-6 / abc) - (- 3)))''
``baz''
``(a - -3)''
```

Similarly, the second and third lines above show the desired output for other expressions shown in Fig. 1. Note that in this output each binary arithmetic operator has one blank before it and another after it, while each unary operator has one blank after it. Otherwise there are no blanks in these Strings. Your output should match this exactly in order to pass the automarking (but, obviously, we won't check these particular expressions)!

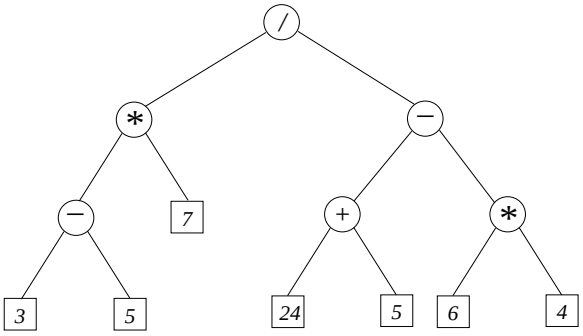

Figure 3: An expression tree which evaluates to -2. Note integer division is used to convert -14/5 to -2.

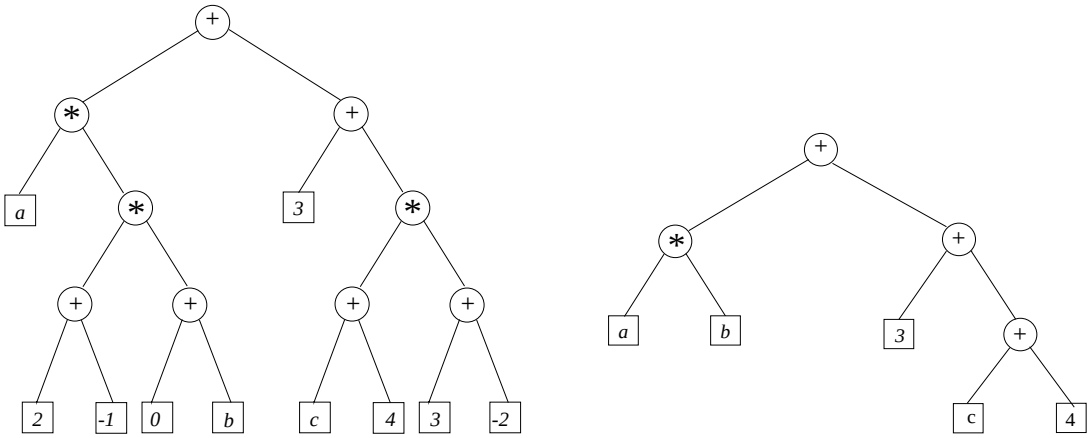

Figure 4: The original tree (left) and a simplification (right).

Question 3. Complete the method eval(SymbolTable) which returns an int for the value of the expression. Here each variable in the expression is replaced by it's current value, stored in the symbol table. Your eval(SymbolTable) algorithm must recursively evaluate sub-expressions within the tree, and not use some other algorithm. Note that the division operator "/" denotes integer division (see Fig. 3), so expressions always evaluate to integers (unless a divide by zero occurs). If a division by zero is encountered during the evaluation, throw a java.lang.ArithmeticException.

Question 4. Complete the method copy(), which returns a copy of the current ExpressionTree. The objects representing the leaves of the tree **should not be duplicated** and re-represented in terms of new objects (since VarTerms and IntTerms are immutable).

Question 5. Complete the method simplify(). This returns a new expression tree which is (possibly) simpler than the original one. The original tree is not changed. Use the following two simplification rules:

1. Whenever x and y are IntTerms, replace subexpressions of the form $(x + y)$, $(x - y)$, $(x * y)$, (x / y), $(-x)$ and $(+x)$ by their integer values.

2. Whenever x is a subexpression, replace $(x + 0)$, $(0 + x)$, $(x - 0)$, $(x * 1)$, $(1 * x)$, and $(x / 1)$ by x. Similarly, replace $(x * 0)$, $(0 * x)$, and $(0 / x)$ by 0. Finally, also replace $(0 - x)$ by $(-x)$, and replace both $(-(-x))$ and $(+x)$ by simply x.

The returned tree must be simplified as much as possible. That is, neither of these two rules should be applicable to simplifying the result further. For example, if the original expression tree looks like the one shown in Fig. 4-left, then the simplified tree is the one shown in Fig. 4-right. Notice that we are **not** asking you to do any fancier simplification, such as replacing the subexpression $(3 + (c + 4))$ in Fig. 4-right by $(c + 7)$. Also, notice that in doing this simplification you should not use the current values of the variables. Rather, for most choices of the values of the variables, the simplification should evaluate to the same result as the original tree. (This may not be true for **all choices** of the values of the variables due to things like

$((0 * y)/x)$ being simplified to 0. The orginal tree will throw an ArithmeticException when evaluated for $x = 0$, but the simplified tree may not.)

Bonus (20%). Make your static method createTree(Parser) capable of correctly parsing expressions with or without parentheses. Specifically, your createTree(Parser) should work on any valid arithmetic expression that can be written in Java, using only the operators in Fig. 2, and using only integers and variables for the terms. Moreover it should interpret the order of the expressions according to the same precedence rules that Java uses. For example,

```
``- (a + b) * 5 * 7 / (-6 / abc - + 3)''
```

is a valid (but not fully parenthesized) Java expression. It is evaluated by Java in the same order as

```
``((((- (a + b)) * 5) * 7) / ((-6 / abc) - (+ 3)))''
```

Both of the above strings should lead to the identical ExpressionTrees from your createTree(Parser).

What to Hand In

You need to electronically submit each of the java files for the classes shown in Fig. 2. This can be done over the web, just like you did for previous assignments. You should also print these java files out, stapling the pages for each individual class together. Then put these printed copies in a manila envelope. Write your name, student number and "CSC148 Assignment#4" on the outside of the envelope, and hand it in at the beginning of class on the due date.

If you choose to do the bonus question, include a short (one or two page) write up describing in detail the algorithm you used for your enhanced createTree(Parser) method.