Com S 228
Fall 2010
Programming Assignment 4
Due at 11:59 pm Friday, November 19
300 points

This assignment is to be done on your own. If you need help, see me or one of the TAs. Please make sure you understand the "Academic dishonesty" section of the syllabus.

Please start the assignment as soon as possible and get your questions answered *early*.
Read through this specification completely before you start.

Introduction

The purpose of this assignment is to give you some experience creating and manipulating trees in a somewhat realistic setting.

WebCT

The WebCT discussion for Assignment 4 is a good place to post general questions. Please do not post or attach any source code for the assignment. Also check the "official clarification" thread for corrections, hints, and answers to common questions. *Any clarification posts prior to 24 hours before the deadline are considered to be part of this specification.*

Summary of tasks

Complete the class ExpressionTrees containing four public methods:

```
public static String getPostfixString(TreeNode root)
public static String getInfixString(TreeNode root)
public static TreeNode reduceConstants(TreeNode root)
public static int evaluate(TreeNode root, Integer... values)
   throws UnboundIdentifierException
public static TreeNode createTree(String expr) throws ParseException
```

Details can be found in the javadoc for the provided ExpressionTrees.java file. The sample code also includes classes Token and Tokenizer for scanning the input, and there is a hierarchy of node types, based on the abstract class TreeNode, which allow us to use polymorphism to

recursively evaluate an expression tree. (Note this is not the same as the generic TreeNode<E> class we have been using for examples in lecture.)

As usual, you should not modify any code except for the ExpressionTrees class.

Overview

In this assignment you will implement a parser for arithmetic expressions in infix form along with some utilities for traversing and transforming the resulting trees. In general the role of a parser is to take an expression in some language – that is, a string of text that is constructed according to some grammatical rules - and create a tree that represents the structure of the expression. Such a tree is called an *abstract syntax tree* or (in the special case we are dealing with here) an expression tree. Once the tree is obtained, it can be evaluated by an interpreter or used by a compiler to generate code, possibly after some transformations.

For this assignment we will use a very simple expression language consisting of arithmetic *operators* (+, *, -, /, ^), *operands* (integer literals and identifiers), and *parentheses*. (The "^" symbol represents exponentiation; note this operator is used for a different purpose in Java.) Examples:

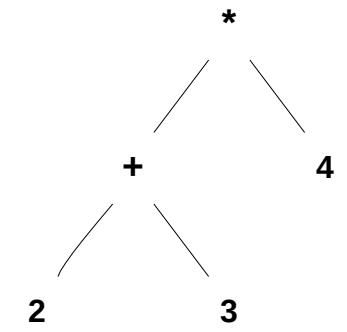

Figure 1: (2 + 3) * 4

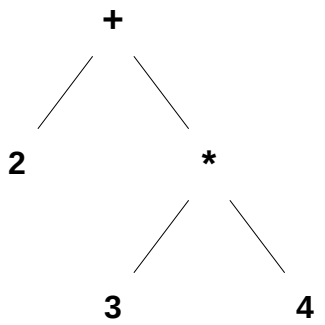

Figure 2: 2 + 3 * 4

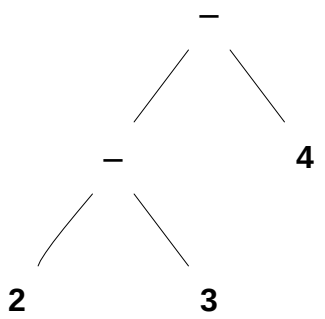

Figure 3: 2 – 3 – 4 (left-associative)

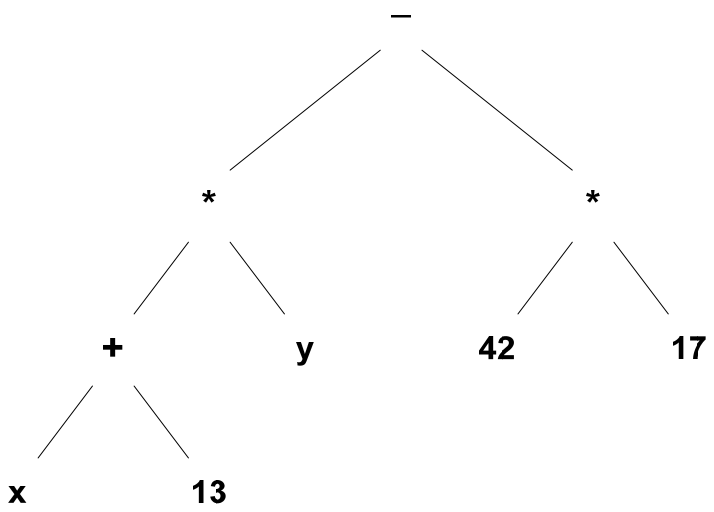

Figure 4: (x + 13) * y – 42*17

The interpretation of addition, subtraction, multiplication, and division will generally be performed according to the precedence rules you are familiar with[1], that is,

- addition and subtraction have equal precedence and are left-associative, that is, evaluated left-to-right. For example, 2 – 3 – 4 means (2 – 3) – 4, not 2 – (3 – 4) (see Figure 3)
- multiplication and division have equal precedence higher than addition and subtraction, and are left-associative
- operations in parentheses are evaluated first

Additional details associated with negation and exponentiation operators are discussed below.

In Real Life (tm) one constructs a parser by first describing a *grammar* for the language, and then using a tool called a *parser generator* (aka "compiler compiler") to create the code for the parser. In simple cases, however, a parser can be written by hand. For the expression language we are considering here, there is an elegant algorithm for parsing expressions using two stacks. That is what you will implement in the createTree() method.

The first step is to tokenize the input, that is, to break the character stream into distinct components recognized by the language's syntax. For example, the string (42+foo17)*137 consists of 7 distinct tokens:

a left parenthesis
an integer literal 42
the operator +
an identifier foo17
a right parenthesis
the operator *
an integer literal 137

You are provided with utility classes Token and Tokenizer for performing this step. Tokenizer implements the interface Iterable<Token>, and the iterator() method returns an Iterator over the Token stream. You can easily examine the token stream via the toString() of a Tokenizer, e.g.

```
System.out.println(new Tokenizer("(2 + 3) * 4");
```

yields the output

```
[LPAREN, INT:2, PLUS, INT:3, RPAREN, TIMES, INT:4]
```

[1] Please Excuse My Dear Aunt Sally?

(The strings displayed above show the "type" of each token; see the enumerated type TokenType.) The algorithm for evaluating the expression is described in pseudocode below. Assume that you have two stacks, one for operands and one for operators. It is convenient to regard the parentheses symbols as operators, where the precedence of the left parenthesis is strictly lower than the precedence of the right parenthesis, which is in turn strictly lower than addition and subtraction. (Also assume in the pseudocode below that we are ignoring the effects of the exponentiation operator and the unary minus.)

```
while there are more tokens in the stream
   current = next token
   if current is an operand, push it on the operand stack
   else if current is a left parenthesis, push it on the operator stack
   else
      min = precedence level of current (***)
      while the operator stack is nonempty and precedence of top item is >= min
         op = pop the operator stack
         pop the top two operands, perform op, and push the result on operand stack
      if current is a right parenthesis
         pop the matching left parenthesis off of the operator stack
      else
         push current on the operator stack

while the operator stack is nonempty
   op = pop the operator stack
   pop the top two operands, perform op, and push the result on operand stack

pop the operand stack and return the result
```

If the operand stack holds integer values, and "perform op" is interpreted to mean that we literally apply the operation to the two integers on top of the stack, then "result" is an integer and the algorithm above will return the value of the expression. However, if the operand stack is designed to hold nodes of a binary tree, and "perform op" is interpreted to mean "create a new node whose children are the two operands," then the algorithm will return the root of a syntax tree for the expression.

You can use any reasonable choice for the stack implementation, but the obvious thing to do is use the java.util.Deque interface (which is implemented by LinkedList).

The tree viewer

The project code includes a rudimentary Swing-based tree viewer that is useful for checking your trees. To start a viewer simply invoke TreeViewer.start(root); where root is the root node of the tree. For example, the expression `-2 + 3 * -(4 * (5 - 6))` would look like this:

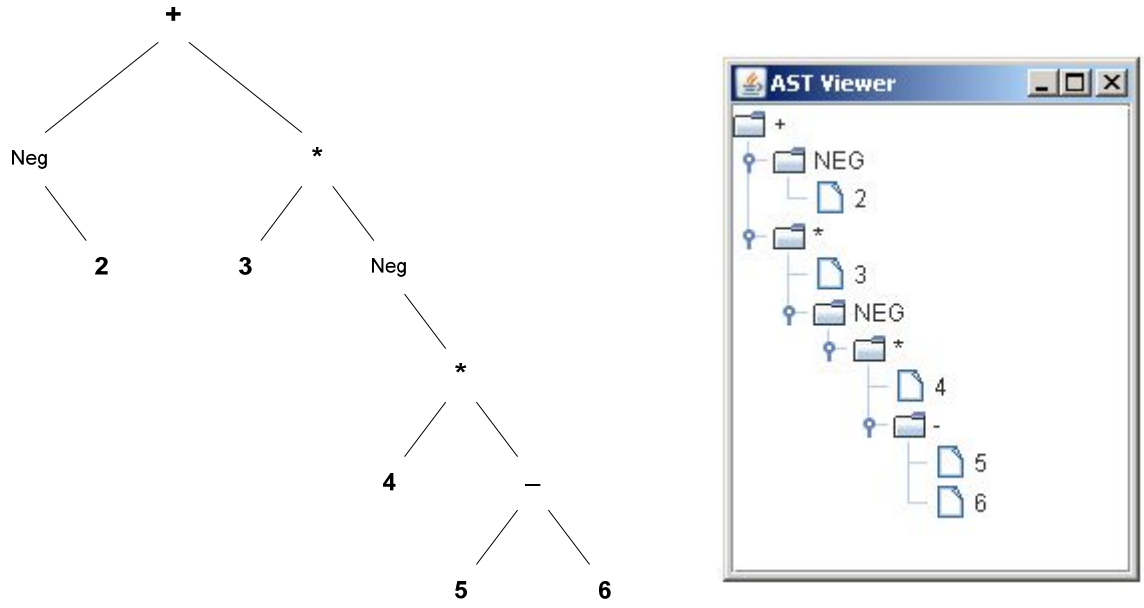

Figure 5

Note that the left child is drawn *above* the right child in the viewer. "Neg" refers to the negation nodes discussed below under "Handling negation". (The folder and file icons, of course, are meaningless in this context.)

Error handling

The tokenizer is fairly stupid, i.e., it can recognize operators and parentheses and whitespace, but it knows nothing about the intended structure of a valid expression. So you can expect to have to handle invalid expressions. One of the features of infix notation is that in general, after each token you can tell whether an operator or an operand is expected next. If you get an operand when an operator is expected, or vice versa, you should throw a ParseException with a suitable message. Doing so requires maintaining a boolean state. e.g. "operandExpected". The basic

rules are something like this (see below for further explanation regarding “unary minus”): Initially an operand is expected. If you see a left paren, that’s ok, but you continue to expect an operand. If you see an operand (int or id token), then on the next iteration an operator is expected. If you see a minus sign when an operand is expected, then it should be interpreted as a negation and you continue to expect an operand. But if you get any other operator, it’s an error. Conversely, when an operator is expected, if you see a right paren, that’s ok but you continue to expect an operator. If you see an operator (other than a paren), then on the next iteration an operand is expected. If you see a left paren or operand, it’s an error.

In addition to the errors noted above, you should generate a ParseException for other parse errors, for example:

An INT token contains text that cannot be parsed as an integer
Missing operands (i.e., operand stack is empty when it should not be)
Extra operands (operand stack not empty after last pop)
Unmatched parentheses (i.e. missing or extra left paren on stack)
Extra operators left on operator stack

Each ParseException should include an appropriate message indicating, as well as possible, where the error occurred. In particular, the message should include the list of tokens already processed up to the point at which the error occurred, as well as the token being read when the error occurred. You will find this incredibly helpful when debugging the algorithm.

Handling negation (unary minus)

The tokenizer does not distinguish between the subtraction operation and the negation of an expression (the so called “unary” minus), both of which use the same symbol. For example, 2 + – (3 – 4) yields the token stream

```
[INT:2, PLUS, MINUS, LPAREN, INT:3, MINUS, INT:4, RPAREN]
```

In order to correctly handle negation, your parser will have to detect when a MINUS token should be interpreted as unary minus, and when it should be interpreted as subtraction. This is not nearly as difficult as it might sound; as noted in the discussion of error handling, if you are expecting an operand and a minus sign appears in the stream, you know it has to be a unary minus.

What does it mean to “interpret a minus sign as a unary minus”? When you encounter a unary minus, you can push a NEGATION token on the operator stack. (Just remember that when you evaluate a NEGATION operator, it only applies to the top operand, not the top two operands.)

When you create the tree, you use the special node type NegationNode: it has only a right child, and its value is the negation of the value of its child. See Figure 6.

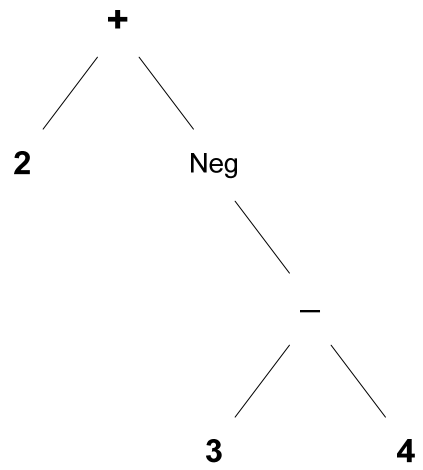

Figure 6: 2 + – (3 – 4)

(It is also possible to preprocess the token stream, replacing each unary MINUS with the NEGATION token, and after that you don't have to worry about whether an operator or operand is expected.)

Handling exponentiation

Java does not have a special symbol for exponentiation (you have to use a library function) but in this expression language we'll use the symbol "^". That is, 2^3 is 2 to the power 3. Normally exponentiation is considered to have higher precedence than unary minus, so that, e.g. –2^4 is –16, not 16. This is all well and good until you ask about the value of something like 2^3^2, which is supposed to evaluate as 2^(3^2) = 2^9, not (2^3)^3 = 8^2. That is, exponentiation is *right-associative*. Where are the association rules built in to the algorithm? The answer is in the line labeled (***) in the pseudocode:

min = precedence level of current (***)
while the operator stack is nonempty and precedence of top item is >= min
op = pop the operator stack
pop the top two operands, perform op, and push the result on operand stack

That is, if you have something like 2 – 3 – 4, when you encounter the second minus, you'll be sitting there with a minus on the operator stack and the 2 and 3 on the operand stack. The choice of "min" above says that we will first perform any pending operations of the *greater or equal* precedence, so the 2 – 3 gets completed before we push the second minus. That's what gives us left-associativity, and it results in the tree shown in Figure 3.

Now if you have 2^3^4, when you encounter the second "^" you'll have the first "^" on the operator stack and the 2 and 3 on the operand stack. But now you want to *leave* the "^" on the operator stack until a lower-precedence operator (or the end of the expression) is encountered. Effectively what you want to do is change the ">=" to a strict inequality for the case of the exponentiation operator. See Figure 7.

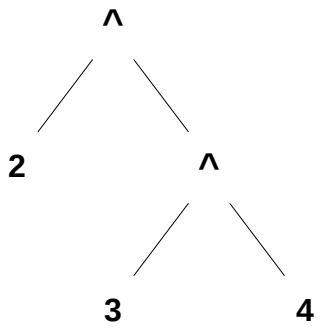

Figure 7: 2^3^4 (right-associative)

Note that when you have double or triple exponentiation, it does not take long before you overflow the capacity of an int. You are not expected to do anything about this, just be aware of it when testing.

Note also that if you expect to handle multiple negations such as – – –42, you'll need to make them right-associative.

Evaluation and the "environment"

The TreeNode abstract class defines an operation eval() that initiates a postorder traversal of the tree, returning the value of the expression represented by the tree. In this application, we are relying on polymorphism to obtain the value of each subtree; each node type "knows" how to evaluate itself given the values of its children. See the node hierarchy in the edu.iastate.cs228.hw4.nodes package.

If the expression contains variables, it can't be evaluated without knowing the values of those variables, i.e., we have to know the *environment* in which the expression is being used. That is the purpose of the Map argument to eval(). The Map interface is defined in the java.util package. An object of type Map<String, Integer> simply associates an Integer value with a given String. To create the association, use the put() method. To get the Integer associated with a given String, use the get() method. (The get() method returns null if there is no value associated with the string.)

To evaluate an expression with variables, you'll need to create a concrete instance of Map<String, Integer>. The standard implementation is called HashMap, e.g.

```
Map<String, Integer> env = new HashMap<String, Integer>();
env.put("x", 2);
env.put("y", 3);
```

defines an environment in which "x" has the value 2 and "y" has the value 3. See the IDNode class for further usage. Note that eval() will throw an UnboundIdentifierException if you have a variable in the expression that does not have a value in the environment.

You'll need to create an environment for testing expressions with variables. In addition, you'll need to create one within your implementation of the evaluate() method of ExpressionTrees, which has the signature

```
public static int evaluate(TreeNode root, Integer... values)
```

The "**`Integer...`**" represents a variable-length argument list. This is a language feature introduced in Java 5. When you call the method, you can supply zero or more int or Integer values, separated by commas, e.g.

```
evaluate(myTree, 2, 3, 4, 5)
```

The compiler converts the list of values into an array. In your implementation of the method, just treat **`values`** as an array of Integer. See the javadoc for ExpressionTrees for more detail about what to do with it.

You do not have to do anything special for other runtime errors such as division by zero or arithmetic overflow.

Some ideas for getting started

1. Start with the simplest case: assume all expressions are valid and that there are no variables, no unary minus, and no exponentiation.

2. Before you do anything else, work through some examples using the given parser algorithm. Draw the stacks at each step.

3. Implement the parser algorithm for infix expressions using integers, that is, instead of creating a tree, just produce the value of the expression.

4. Review the sample code in TreeExample.java (see tree_examples.zip on WebCT). This file contains a simple example that constructs an expression tree from a postfix expression (using a single stack). Then modify the algorithm from step 3 to create a tree instead of computing the value.

5. Once you create a tree, you can call eval(null) on the root node. As long as there are no variables, this should work fine. Note that the node hierarchy, including the eval() method for each type, is fully implemented.

Documentation and style

Up to 15% of the points will be for documentation and style. See the style guidelines posted on WebCT.

What to turn in

All of your code should be in the class edu.iastate.cs228.hw4. Submit a zip file, containing your source code and all of the distributed code, in the correct package structure. The top-level directory should be edu.

See the submission HOWTO on the WebCT main page if you not certain how to do this. You **will** lose points for submission errors (and in the worst case, if you do not correct your submission errors before the late deadline, you may receive no credit at all for the assignment). It is your responsibility to check the file that you have uploaded to WebCT and make sure that it is complete and correct.

Late Penalties

Assignments may be submitted up to 24 hours after the deadline with a 25% penalty (not counting weekends/holidays). No credit will be given for assignments submitted after that time.