ICSI 311 Assignment 3 – Start the Parser

This assignment is extremely important – (nearly) every assignment after this one uses this one!

If you have bugs or missing features in this, you will need to fix them before you can continue on to new assignments. This is very typical in software development outside of school.

You must submit .java files. Any other file type will be ignored. Especially ".class" files.

You must not zip or otherwise compress your assignment. Blackboard will allow you to submit multiple files.

You must submit buildable .java files for credit.

This assignment **must** have five new source files (Parser.java, Node.java, IntegerNode.java, FloatNode.java, MathOpNode.java) **as well as** your first three source files (Basic.java, Token.java, Lexer.java).

The parser will take the collection of tokens from the lexer and build them into a tree of AST nodes. To start this process, make an abstract Node class. Add an abstract ToString override. Now create an IntegerNode class that derives from Node. It **must** hold an integer number in a private member and have a read-only accessor. Create a similar class for floating point numbers called FloatNode.java. Both of these classes should have appropriate constructors and ToString() overrides.

Create a new class called MathOpNode that also derives from Node. MathOpNode **must** have an enum indicating which math operation (add, subtract, multiply, divide) the class represents. The enum **must** be read-only. The class **must** have two references (left and right) to the Nodes that will represent the operands. These references must also be read-only and an appropriate constructor and ToString() must be created.

Reading all of this, you might think that we can just transform the tokens into these nodes. This would work, to some degree, but the order of operations would be incorrect. Consider 3 * 5 + 2. The tokens would be INTEGER TIMES INTEGER PLUS INTEGER. That would give us MathNode(*,3,MathNode(+,5,2)) which would yield 21, not 17.

Create a Parser class (does not derive from anything). It **must** have a constructor that accepts your collection of Tokens. Create a public parse method (no parameters, returns "Node"). Parse **must** call expression (it will do more later). You **must** creating some helper methods as matchAndRemove() as described in lecture.

The classic grammar for mathematical expressions (to handle order of operations) looks like this:

EXPRESSION = TERM { (plus or minus) TERM}

TERM = FACTOR { (times or divide) FACTOR}

FACTOR = {-} number or lparen EXPRESSION rparen

Turn each of these (expression, term, factor) into a method of Parser. Use matchAndRemove to test for the presence of a token. Each of these methods should return a class derived from Node. Factor will return a FloatNode or an IntegerNode OR the return value from the EXPRESSION. Note the unary minus in factor – that is important to bind the negative sign more tightly than the minus operation. Also note that the curly braces are "0 or more times". Think about how 3*4*5 should be processed with these rules. Hint – use recursion and a helper method. Also think carefully about how to process "number", since we have two different possible nodes (FloatNode or IntegerNode). Depending on how you implemented your lexer, factor may or may not need to deal with negating the number.

Make sure that you test your code. Change your main to call parse on the parser. Right now, it will only process a single line. Print your AST by using the "ToString" that you created. Use several different mathematical expressions and be sure that order of operations is respected. Your lexer can create tokens that your parser cannot handle yet. That is OK.

Rubric	Poor	OK	Good	Great
Comments	None/Excessive (0)	"What" not "Why", few (5)	Some "what" comments or missing some (7)	Anything not obvious has reasoning (10)
Variable/Function naming	Single letters everywhere (0)	Lots of abbreviations (5)	Full words most of the time (8)	Full words, descriptive (10)
Create the AST classes	None (0)	Classes missing (5)	All classes present, some methods missing (10)	All classes and methods (20)
Parser class	None (0)		Constructor or private member (5)	Constructor and private member (10)
Helper Method(s)	None(0)			At least 1 (5)
Factor Method	None (0)		Significantly Attempted (10)	Correct (15)
Expression Method	None (0)		Significantly Attempted (10)	Correct (15)
Term Method	None (0)		Significantly Attempted (10)	Correct (15)