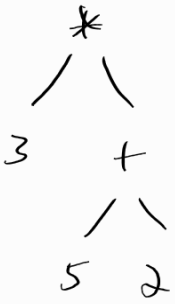
Consider 3 * 5 + 2. The tokens would be *NUMBER TIMES NUMBER PLUS NUMBER*. That would give us *MathOpNode(*, 3, MathOpNode(+, 5, 2))* which would yield 21, not 17.

Why does this yield 21? Let's look at the expression tree for this MathOpNode and see how it's evaluated.

Expression trees are evaluated from the bottom up (think bottom up recursion).
For MathOpNode(*, 3, MathOpNode(+, 5, 2)), we have this tree:

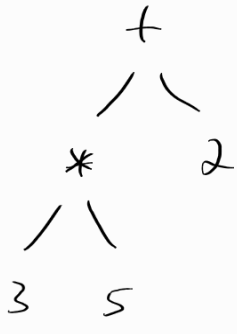


Notice how the operator is always the parent of two Integer (or Float) nodes.
Let's evaluate this expression tree from the bottom up.

We first evaluate the MathOpNode(+, 5, 2). That is would just be 5 + 2 = 7. Now our expression tree looks like this:

Evaluating this expression tree would just be 3 * 7 = 21. This is why *MathOpNode(*, 3, MathOpNode(+, 5, 2))* is evaluated as 21. However, this is MathOpNode represents 3 * (5 + 2), not the expression 3 * 5 + 2 as discussed earlier.

The correct expression tree for 3 * 5 + 2 is:



Evaluating this expression tree is the same process as above. You start from the bottom and move upward. You first evaluate 3 * 5 = 15 and replace that part of the tree with 15. Finally, you evaluate 15 + 2, yielding 17.

Converting this expression tree back to a MathOpNode (as specified in the assignment document, you get:
MathOpNode(+, MathOpNode(, 3, 5))*

So, how do we form MathOpNodes that respect order of operations? Recall the assignment document has:

EXPRESSION = TERM { {plus or minus} TERM}
TERM = FACTOR { {times or divide} FACTOR}
FACTOR = {-} number or lparen EXPRESSION rparen

To summarize the above three statements,

- EXPRESSION can consist of an infinite number of TERM's delimited by a + (addition)

or - (subtraction).
- TERM can consist of an infinite number of FACTOR's delimited by * (multiplication) or / (division).
- FACTOR is a number (possibly negative) or a "(", lparen followed by an EXPRESSION and ended by a ")", rparen.

Notice how in FACTOR, there is another EXPRESSION. How do we handle an expression inside an expression? -- I'll leave this for you to think about and figure out.

These three lines are three methods that handle different parts of a mathematical expression. *Hint: you will need to use your MathOpNode class here*

A match and remove method is just a method that checks if the Token at the front of the list of Tokens (the list of Token returned from your Lexer class) is the correct one (i.e., equal to the one you're looking for). If the Token matches, remove and return it. Otherwise, return something to signify that the Token was not found.

These four, mixed together is how you respect order of operations in mathematical expressions.

Hint: Expression calls Term which then calls Factor

How do we parse mathematical expressions?

First determine what the left operand is by calling the above mentioned methods one after another. An operand can be an instance of an IntegerNode, a FloatNode, or another MathOpNode.

Recall what a FACTOR can be. How do you look for a specific Token?
You use MatchAndRemove to check for number and return it to Factor.
MatchAndRemove returns to Factor which returns the Token, now parsed as either an Integer or Float node to Term. Term then checks if the next Token is a multiplication or division. How do we check for either of these Tokens?

Term repeats this process until there are no more immediate multiplication or division signs. Once Term has completed this process, it returns to Expression and repeats the process, but instead checks for addition or subtraction signs.

Once Expression completes this process, we now have the left operand. Repeat this exact process for determining the right operand.

Other things you will need to implement:
- What if there are multiple numbers delimited by different operators?
    - How do you keep track of an existing left and right operand and its operator while also continuing to parse a new right operand?
    Hint: Recall what an Node type an operand of a MathOpNode can be.

- What do you do if you encounter a left parenthesis?

Example:

Input: 3 + 5
After Lexer: NUMBER(3) PLUS NUMBER(5)

Expression -> Term -> Factor
Expression calls Term, which initially calls Factor. Factor looks for a number (possibly negative) or an lparen.

Number token found! (Recall how we check for Tokens). Return IntegerNode(3) and return to Term
Term then looks for a multiplication or division sign. None found. Return to Expression

Expression now has IntegerNode(3) as a left operand.
Expression then continues to look for an addition or subtraction sign and stores it as the operator.

Expression then call Term, which calls Factor to determine what the right operand is. Recall again what a Factor looks for.

Number token found! Return IntegerNode(5). Term continues and again checks for any immediate multiplication or division signs. None found. Return to Expression

Expression continues again to look for another addition or subtraction sign. None found, return to Expression.

Expression now has the left operand, the right operand, and the operator. What do we do with this information? -- Create a MathOpNode and return it.

For the input of "3 + 5", you should get back *MathOpNode(+, 3, 5)*.