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.

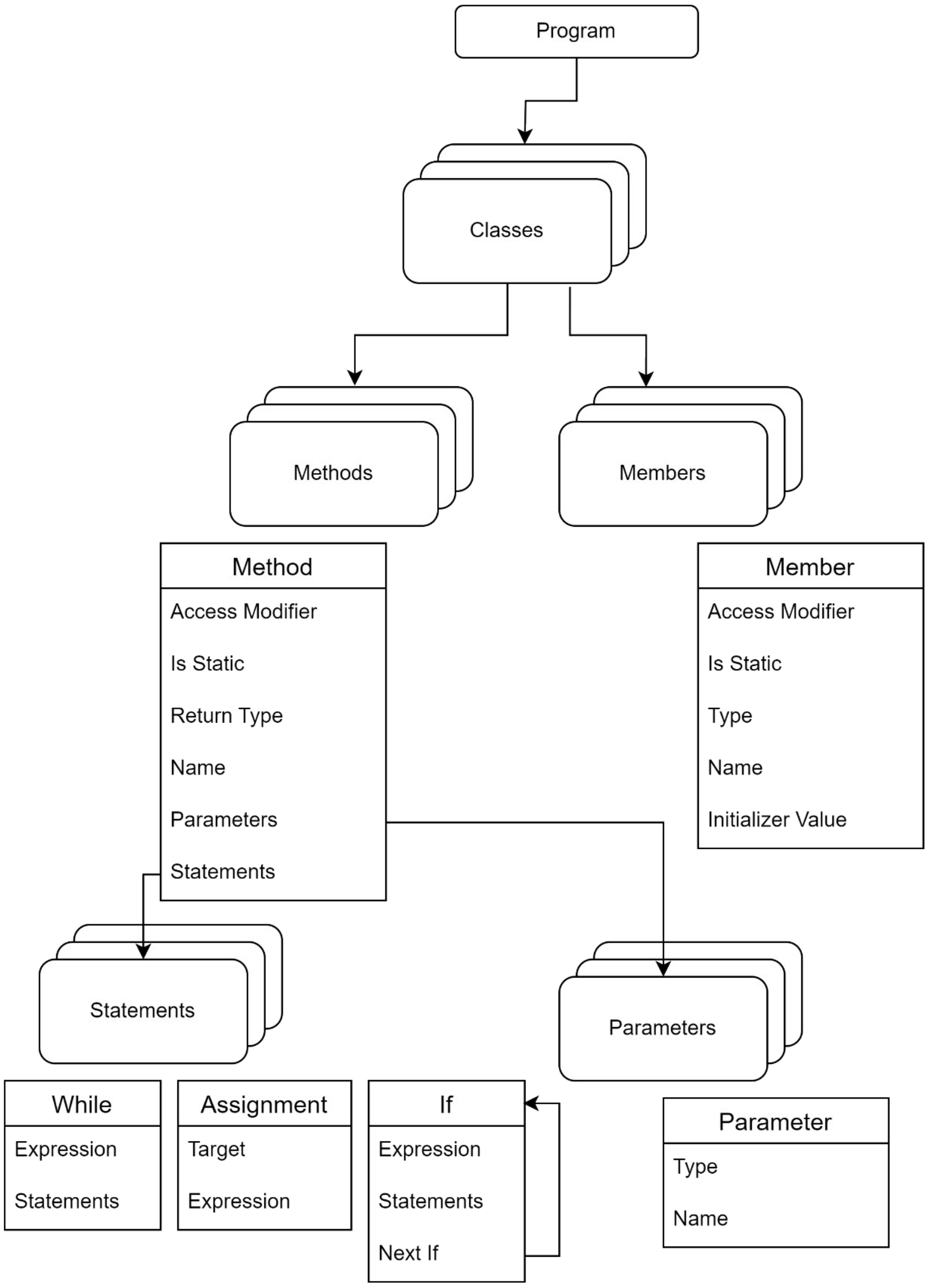
You must not zip or otherwise compress your assignment. Brightspace will allow you to submit multiple files.

You must submit every file for every assignment.

You must submit buildable .java files for credit.

Introduction

In the last two assignments, we transformed a text file into a linked list of tokens. Over the next several assignments, we will transform that list of tokens into a tree. This tree will be a little different, perhaps, than other trees you have seen. It is an n-ary tree with heterogeneous nodes. Consider Java for a moment – a Java program has a root node(Program). A program has one or more classes. A class has zero or more members and zero or more methods. A method has 1 or more statements. See the diagram below.



BASIC has a simpler structure – a program is just a collection of instructions.

This is a good point to mention – the symbol tree is literally the encoding of all the (important) information in a program. If you are not sure what to put in a node, look at what can go into the part of the program.

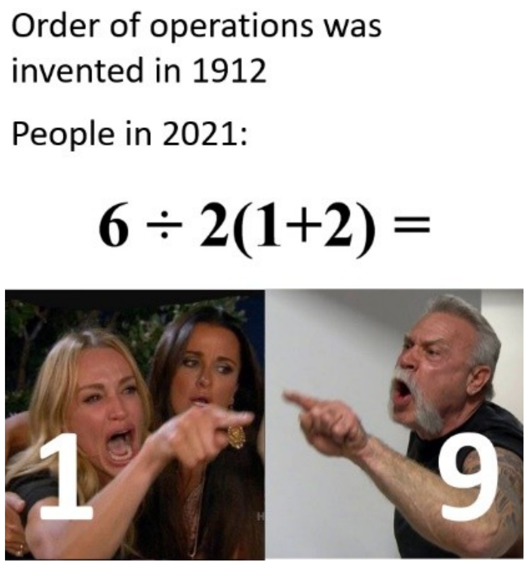
We want an abstract base class (Node). All our nodes will derive from it. Every node **must** have a ToString() method. I made mine formatted to try to look close to what the program itself will look like. That means that when I call ToString() on my ProgramNode, I get output that looks kind of like my input program. This is a very powerful way to visually check your tree.

Finally, we will be using Optional<> a great deal in the parser. This class "wraps" another data type to indicate that there may or may not be one. You can read more about it here:

https://docs.oracle.com/en/java/javase/19/docs/api/java.base/java/util/Optional.html

Order of operations is a critical part of building your parser. Let's start by reviewing.

You've all seen these memes →



These all revolve around understanding order of operations. Many of us learned in grade school:

PEMDAS (Please excuse my dear Aunt Sally):

Parenthesis

Exponents

Multiplication & Division

Addition & Subtraction

The traditional way to deal with these in Parsers is the Expression-Term-Factor pattern:

Expression: TERM {+|- TERM}

Term: FACTOR {*|/ FACTOR}

Factor: number | (EXPRESSION)

A couple of notation comments:

| means "or" – either + or -, either * or /

{} means an optional repeat. That's what allows this to parse 1+2+3+4

Think of these as functions. Every CAPTIALIZED word is a function call.

Consider our meme expression above: 6/2*(1+2)

We will start by looking at expression. Expression starts with a TERM. We can't do anything until we resolve TERM, so let's go there.

A term starts with FACTOR. Again, we can't do anything until we deal with that.

A factor is a number or a (EXPRESSION). Now we can look at our token (hint: MatchAndRemove). We see a number. OK – our factor is a number. We "return" that. Remember that we got to factor from term. Let's substitute our number in:
TERM: FACTOR(6) {*|/ FACTOR}

Now we deal with our optional pattern. Is the next character * or /? Yes! Now is the next thing a factor?

It turns out that it is. Let's substitute that in:
TERM: FACTOR(6) / FACTOR(2)

But remember that our pattern is a REPEATING pattern (hint: loop):

TERM: FACTOR(6) / FACTOR(2) {*|/ FACTOR}

We see the * and call factor. But this time, the factor is not a number but a parenthetical expression.

Factor: number | (EXPRESSION)

Factor calls expression.

Expression calls term, term calls factor, factor returns the number 1. Term doesn't see a * or / so it passes the 1 up. Expression sees the + and calls term. Term calls factor which returns the 2. Expression doesn't see a +|- value so ends the loop and returns 1+2.

So, remember that we were here:

TERM: FACTOR(6) / FACTOR(2) * FACTOR

Our factor is (1+2). That can't be broken down. That math HAS to be done before can multiply or divide. That's what enforces order of operations.

Details

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.

Much like we had a StringManager in the Lexer, we want a "TokenManager". This class manages the token "stream" in the same way that the StringManager managed the character "stream". Create this class with a single private member – a linked list of tokens. Make a constructor that accepts that linked list and sets the private member. Then make the following methods:

Optional<Token> Peek(int j) – peek "j" tokens ahead and return the token if we aren't past the end of the token list.

boolean MoreTokens – returns true if the token list is not empty

Optional<Token> MatchAndRemove(TokenType t) – looks at the head of the list. If the token type of the head is the same as what was passed in, remove that token from the list and return it. In all other cases, returns Optional.Empty(). You will use this **extensively.**

Note that there is no accessor for the token list – you must use MatchAndRemove.

Create a Parser class. Much like the Lexer, it has a constructor that accepts a LinkedList of Token and creates a TokenManager that is a private member.

The next thing that we will build is a helper method – boolean AcceptSeperators(). One thing that is always tricky in parsing languages is that people can put empty lines anywhere they want in their code. Since the parser expects specific tokens in specific places, it happens frequently that we want to say, a new line, but there can be more than one". That's what this function does – it accepts any number of separators (EndOfLine) and returns true if it finds at least one.

Create a Parse method that returns a ProgramNode. While there are more tokens in the TokenManager, it should loop, looking for more Expressions (see below) with separators in between.

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. Expression, Term and Factor give us order of operations!

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).

Implement Expression, Term and Factor. 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. Also think carefully about how to process "number", since we have two different possible nodes (FloatNode or IntegerNode).

Make sure that you extensively unit test your parser's methods. 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.

Change your main to call parse on the parser. Print your AST by using the "ToString" that you created.

Rubric	Poor	OK	Good	Great
Code Style	Few comments, bad names (0)	Some good naming, some necessary comments (3)	Mostly good naming, most necessary comments (6)	Good naming, non-trivial methods well commented, static only when necessary, private members (10)
Unit Tests	Don’t exist (0)	At least one (6)	Missing tests (12)	All functionality tested (20)
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)
Factor Method	None (0)		Significantly Attempted (5)	Correct (10)
Expression Method	None (0)		Significantly Attempted (5)	Correct (10)
Term Method	None (0)		Significantly Attempted (5)	Correct (10)
Changes to main	None (0)			Calls parser, prints parse tree (10)