Assignment #3 CS326 – Fall 2019

Parsing Simple Arithmetic Expressions in Java using ANTLR

In this programming assignment you will write a program in Java that parses simple arithmetic expressions and turns such expressions into abstract syntax trees. The objective of this assignment is to familiarize yourself with the process of creating a language using lexical rules and a context-free grammar (or **grammar** for short) – for the first part of this assignment, we will not use **BNF** (Backus-Naur Form) notation but more like a modified **EBNF** notation. While there exists several algorithms and tools for parsing context-free grammars (e.g., the Early parser, Java CUP, YACC/Bison, etc.), I will focus on the usage of ANTLR[1]. In particular, we will learn how to:

- Define a grammar.
- Create a parse tree.
- Use this parse tree structure in our program.

In this assignment you will write a very simple language based on your previous programming assignment and use ANTLR to parse arithmetic expressions. Please take the extra 1-2 hours it might take to make your assignment answer look nice – especially since I have to read and correct them.

Part 1 — The Grammar

Writing our language
Suppose we want to define a language for arithmetic expressions, here is what we need:

1. A set of terminal symbols that can appear in the string.
2. A set of non-terminal symbols that represent sets of strings being defined recursively.
3. A set of production rules (e.g., $A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$) where left-hand side non-terminal symbols are replaced with right-hand side symbols recursively, until a terminal symbol is reached.
4. A starting symbol representing a valid sentence.

Strictly speaking, we need a finite set of terminal symbols, non-terminal symbols, and productions to describe arithmetic expressions involving natural numbers. It should be clear that the set of non-terminal symbols will consist of digits and the operators $+$, $-$, $*$, $/$, and $**$ (short for exponential and used to disambiguate it from multiplication). Therefore, non-terminal symbols are:

[1]Refer to Appendix X for ANTLR installation and use.

1. a number, a symbol representing a single digit or more.
2. an expression, a symbol representing a whole expression using addition, subtraction, multiplication, division, or exponential.

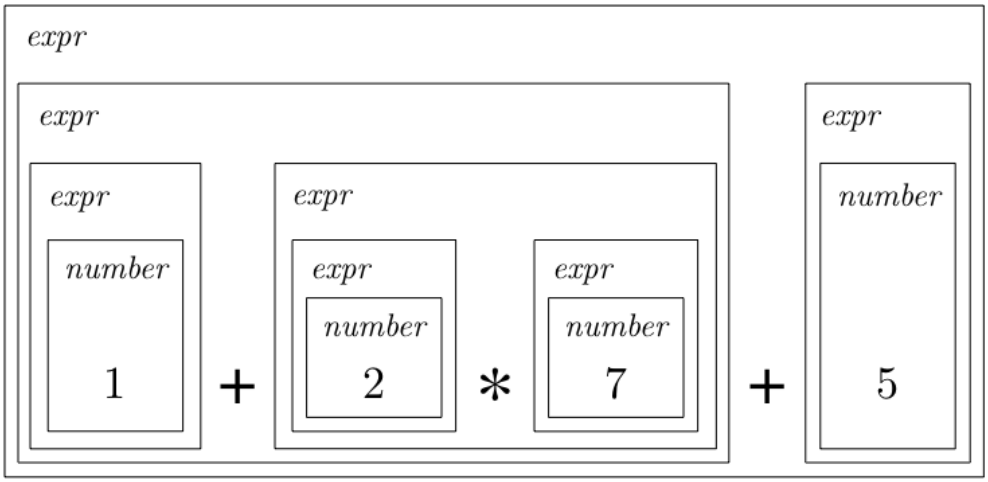
We will use productions for replacing:

- a number, which can be a non-empty string consisting of an arbitrary number of digits.
- an expression, which can be a single number or an expression followed by one of the arithmetic operators and then followed by another expression.

Now, let us talk about the mechanical aspects of grammars. The grammar below generates strings representing arithmetic expressions with the five operators $+$, $-$, $*$, $/$, $**$, and numbers as operands.

$$
\begin{array}{lcl}
\textit{expr} & \rightarrow & (\ \textit{expr}\ ) \\
 & | & \textit{expr}\ \mathbf{**}\ \textit{expr} \\
 & | & \textit{expr}\ \mathbf{*}\ \textit{expr} \\
 & | & \textit{expr}\ /\ \textit{expr} \\
 & | & \textit{expr} + \textit{expr} \\
 & | & \textit{expr} - \textit{expr} \\
 & | & \textit{number} \\
 & & \\
\textit{number} & \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

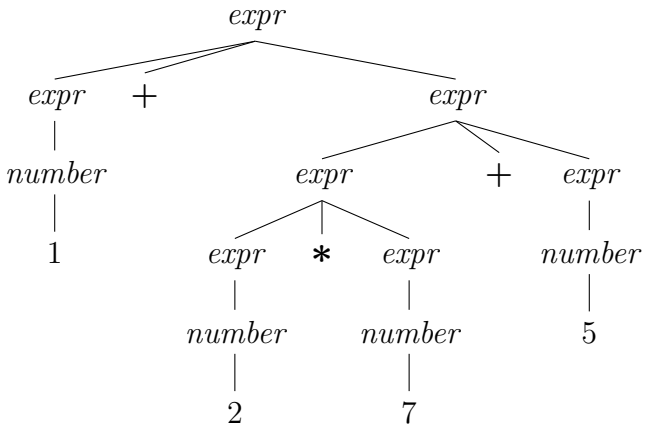
Using the above production rules, we can decompose and evaluate arithmetic expressions, such as $1 + 2 * 7 + 5$, as follows:



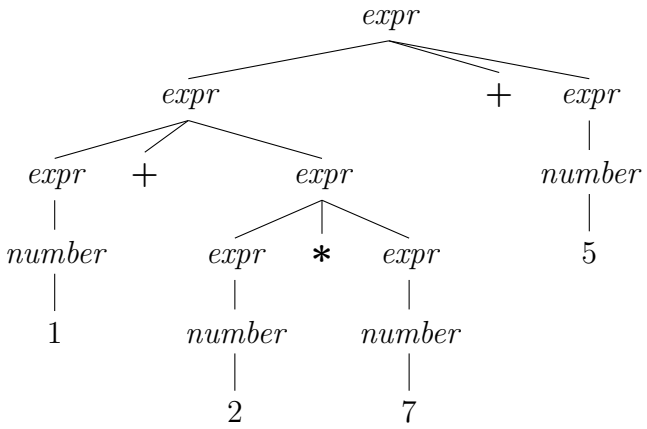
If we begin with the uppermost rectangular box *expression* and walk our way down to the lowermost rectangular box, we should be able to see that higher-precedence operators execute before operators with lower precedence; that is, the operator $*$ executes before the

operator +. However, looking at the decomposition of the same arithmetic expression, we can see that our grammar is ambiguous. This, of course, becomes a problem when we try to generate parse trees. Consider the same arithmetic expression $1 + 2 * 7 + 5$; which tree should we generate?

One which has the following parse tree:



Or, one which has this parse tree:

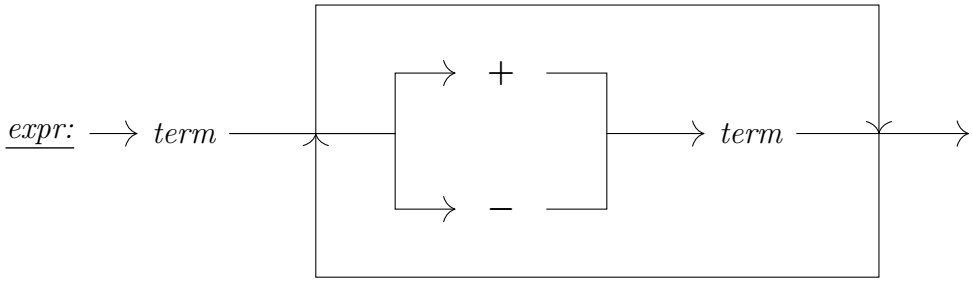


It should be pretty obvious that ambiguous grammars, such as the one above, are not desirable because the derivation tree provides conflicting information. No only can we derive an arithmetic expression two different ways, but we can also produce the same numeric result. For both trees, for example, the combination of numbers and operators yields 20! Clearly, our grammar needs to be rewritten to reflect the desirable parse tree. The latter is probably the one we want, as it interprets $1 + 2 * 7 + 5$ as $(1 + (2 * 7)) + 5$, which is what we are accustomed from algebra.

Rewriting our language

Before we begin rewriting our grammar, we need to make sure that any recursion happens to the left of + or − (this is to preserve left associativity). Further still, we must demonstrate that if a symbol matches a rule, such rule can generate all the characters in the symbol. To demonstrate that a symbol is legal according to some rule, we need to match all of its characters with all the items in the rule. If there exist an exact match, we classify the symbol as legal. Otherwise, we classify the symbol as illegal and say it does not match.

Graphically, we can represent a production rule with a syntax diagram. The syntax diagram (or syntax chart) illustrates how a sequence and repetition of two or more alternatives (a choice) interact for putting characters together. For example, we can use the diagram below to represent addition or subtraction operations and define the EBNF rule *expr*. We then simply follow the arrows to see whether we can get from the beginning to the end.



Put differently, when using the rule *expr*, and if we are trying to derive the expression 1 + 2 − 2, the way we would match a character with an item in the above diagram is depicted in the table below:

Status	*Action*
expr	Given
term { [+ | −] *term* }	Replace *expr* by right-hand side
1 { [+ | −] *term* }	Replace *term* with 1
1 [+ | −] *term* [+ | −] *term*	Use two repetitions
1 [+] *term* [+ | −] *term*	Choose + alternative
1 + 2 [+ | −] *term*	Replace *term* with 2
1 + 2 [−] *term*	Choose − alternative
1 + 2 − 2	Replace *term* with 2

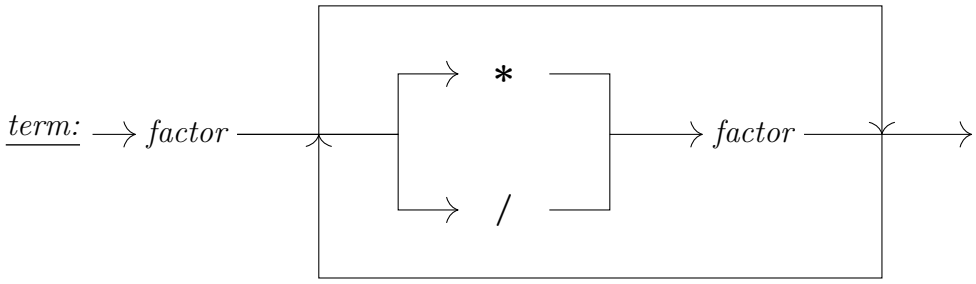
Therefore, the rule for describing the above syntax diagram is:

$$expr \rightarrow term\ (\ (\ +\ |\ -\ )\ term\ )*$$

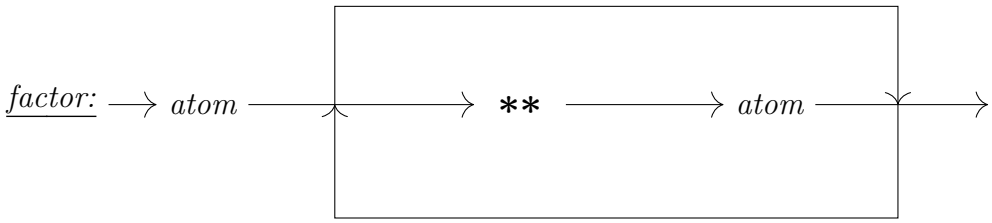
where ( ) is used to group various items and ∗ represents zero or more repetitions. Note that while a different choice can be made for each repetition, only one + or − character

can be repeated when a choice is made. To maintain the left associativity, we shall continue 'expanding' the non-terminal *expr*.

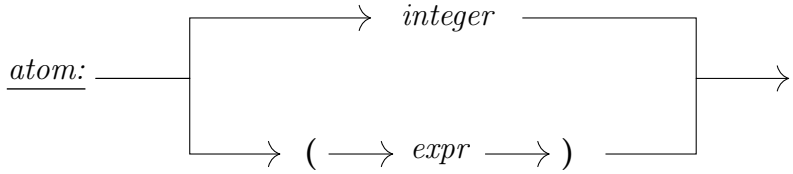
Keeping in mind the syntax diagram for addition or subtraction operations, we can create a similar diagram to represent multiplication or division operations and define the EBNF rule *term*.



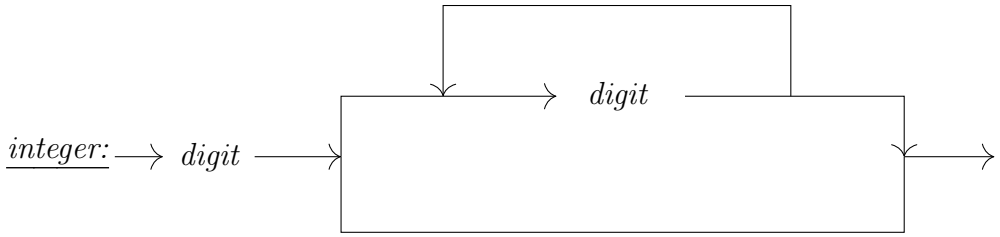
The production for the EBNF rule *factor*, which represents exponential operations, is defined in a similar way.

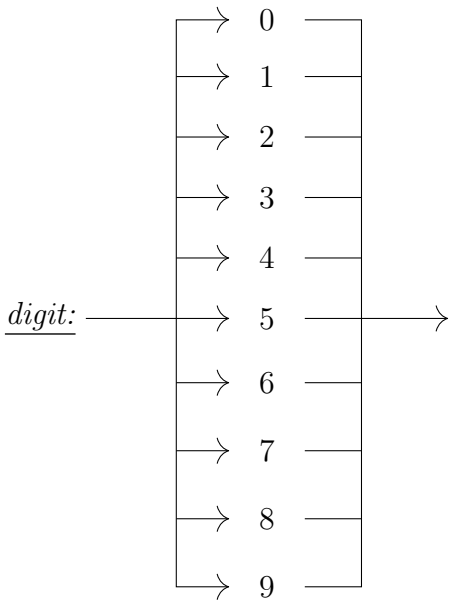


Since an expression can be a single value, such as 7, or something more complicated in parentheses, such as $(2 * 7)$, we define the EBNF rule *atomic* expression (or just *atom* because it is made out of smaller expressions) with the following diagram:



Next, we define the EBNF rule *integer* as a sequence of two items: a single digit followed by a repetition of zero or more digits, where each digit is chosen from the list of alternatives in the *digit* rule.

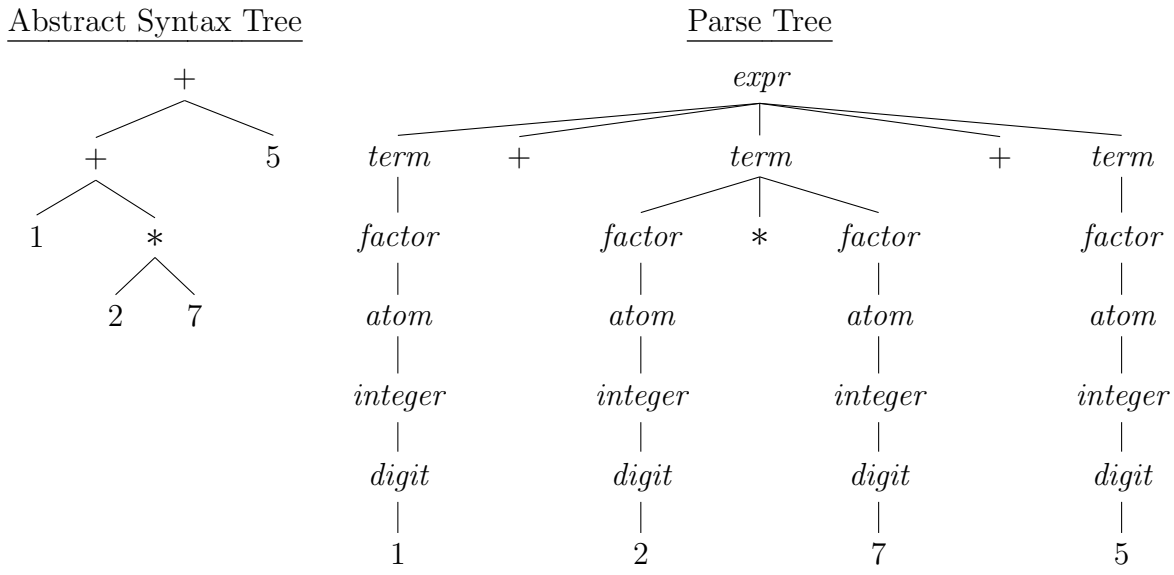




As we can see, each of the above diagrams correspond to a production rule in our initial grammar. Naturally, using these syntax diagrams, we can rewrite our initial grammar to eliminate ambiguity and prevent infinite recursion while maintaining order of operations. After doing that, we wind up with the grammar bellow (a bit like what's in your book):

$$
\begin{array}{lcl}
\textit{expr} & \rightarrow & \textit{term}\ (\ (\ +\ |\ -\ )\ \textit{term}\ )* \\
\textit{term} & \rightarrow & \textit{factor}\ (\ (\ \textbf{*}\ |\ /\ )\ \textit{factor}\ )* \\
\textit{factor} & \rightarrow & \textit{atom}\ (\ \textbf{**}\ \textit{atom}\ )* \\
\textit{atom} & \rightarrow & \textit{integer}\ |\ (\ \textit{expr}\ ) \\
 & & \\
\textit{integer} & \rightarrow & \textit{digit}\ (\ \textit{digit}\ )* \\
\textit{digit} & \rightarrow & 0\ |\ 1\ |\ 2\ |\ 3\ |\ 4\ |\ 5\ |\ 6\ |\ 7\ |\ 8\ |\ 9
\end{array}
$$

All we need to do is write the above rules and let ANTLR do its magic! ANTLR takes a grammar that specifies a language and uses it to generate a lexer and a parser. A lexer turns a string of characters into tokens. These tokens are then used by the parser to create an Abstract Syntax Tree (AST) and to interpret the code or translate it into some other form according to the rules of the grammar. Visually, a tree for the expression $1 + 2 * 7 + 5$ looks like this:



While the AST is much smaller than the parse tree, both trees capture the essence of the input. In the picture above, we can see that both trees help us understand how the input was interpreted by the parser. After viewing the execution trace of each tree (specially the one belonging to the parse tree), we can see the sequence of rules the parser applied to recognize the input. In other words, it shows how the start symbol of our grammar derives a certain string in the language. For example, the expression $1 + 2 * 7 + 5$ is derived as follows:

$$
\begin{aligned}
\textit{expr} &\Rightarrow \overbrace{\textit{term}}^{\textit{factor}} + \textit{term} + \textit{term} \\
&\Rightarrow \overbrace{\textit{factor}}^{\textit{atom}} + \textit{term} + \textit{term} \\
&\Rightarrow \overbrace{\textit{atom}}^{\textit{integer}} + \textit{term} + \textit{term} \\
&\Rightarrow \overbrace{\textit{integer}}^{\textit{digit}} + \overbrace{\textit{factor} * \textit{factor}} + \textit{term} \\
&\Rightarrow \overbrace{1}^{\textit{digit}} + \overbrace{\textit{atom}}^{\textit{integer}} * \textit{factor} + \textit{term} \\
&\Rightarrow 1 + \overbrace{\textit{integer}}^{\textit{digit}} * \overbrace{\textit{factor}}^{\textit{atom}} + \textit{term} \\
&\Rightarrow 1 + \overbrace{\textit{digit}}^{2} * \overbrace{\textit{atom}}^{\textit{integer}} + \overbrace{\textit{term}}^{\textit{factor}} \\
&\Rightarrow 1 + 2 * \overbrace{\textit{integer}}^{\textit{digit}} + \overbrace{\textit{factor}}^{\textit{atom}} \\
&\Rightarrow 1 + 2 * \overbrace{\textit{digit}}^{7} + \overbrace{\textit{atom}}^{\textit{integer}} \\
&\Rightarrow 1 + 2 * 7 + \overbrace{\textit{integer}}^{\textit{digit}} \\
&\Rightarrow 1 + 2 * 7 + \overbrace{5}
\end{aligned}
$$

Grammar file

As previously stated, we will be using a very simple grammar to do integer calculations. To get started we should first describe the grammar:

```
grammar Infix;

goal        :   expression EOF;

expr        :   term ((PLUS | MINUS) term)*;
term        :   factor ((MUL | DIV) factor)*;
factor      :   atom (POWER atom)*;
atom        :   INTEGER | LBRACE expression RBRACE;

PLUS        :   '+';
MINUS       :   '-';
MUL         :   '*';
DIV         :   '/';
POWER       :   '**';
LBRACE      :   '(';
RBRACE      :   ')';
INTEGER     :   [0-9]+;
WS          :   [ \t\r\n]+ -> skip;
```

The above grammar contains everything ANTLR needs to generate a correct lexer and parser. The first line is the name of our grammar (and the name of our ANTLR file, which is `Infix.g4`). Next, we define the start symbol `goal` as an expression followed by the end-of-line marker. Recall, an expression can be an expression between braces, an exponential, multiplication, division, summation, subtraction or a number. The way in which the rules are arranged demonstrates the precedence and associativity of terminal symbols in our grammar. In other words, an expression between braces has a higher precedence than anything else, exponential has a higher precedence than multiplication and division, multiplication has a higher precedence that summation and subtraction, etc. Here is the precedence table for reference.

Higher Precedence ↓

Precedence Level	Associativity	Operators
1	left	+, −
2	left	*, /
3	left	**
4	left	( )

You may have noticed that our grammar contains a combination of lower and uppercase words. Lower case words are parser expressions, while upper case words are lexer expressions.

The latter are rules that we use to match characters on the input stream and to form the foundation for our parser rules. For example, we define 'tokens' for characters we want to mark as special **words**, such as in the following example

```
INTEGER: [0-9]+
```

which is a token consisting of one or more digits. Another example is the following lexer expression:

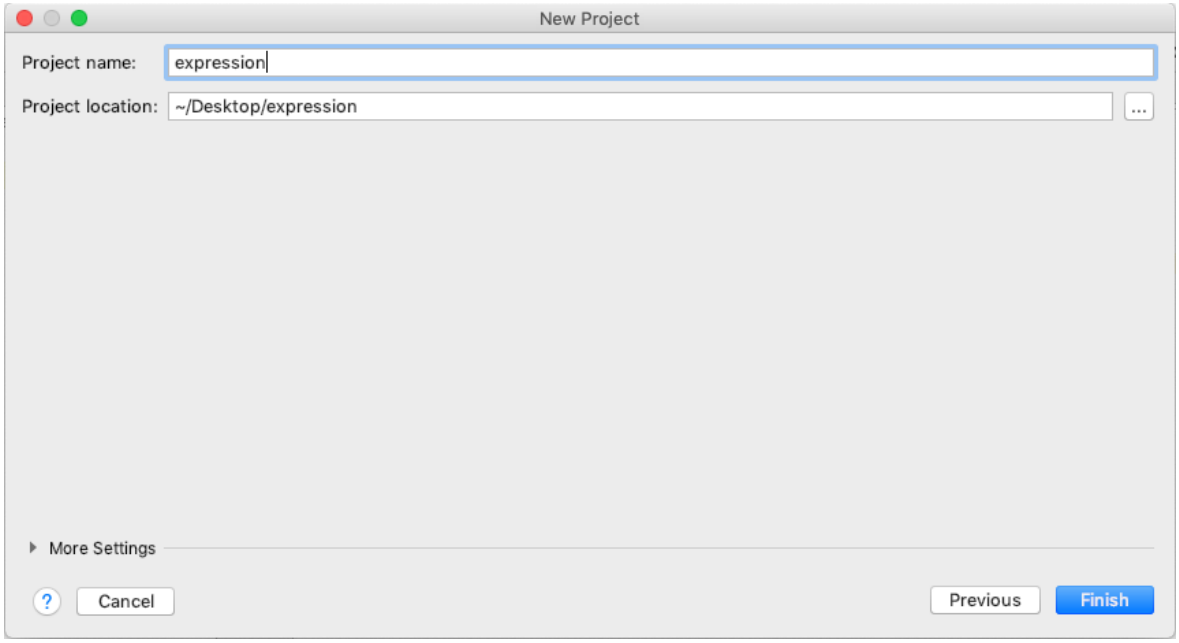
```
WS: [ \t\r\n]+ -> skip
```

Here we define a whitespace token consisting of one or more spaces, tabs, and line breaks, including the ANTLR `skip` token (which cause the lexer to discard white spaces, tabs, line breaks, etc.). Therefore, we use lexer expressions to identify different words in a sentence and parser expressions to determine whether the sentence is semantically correct after identifying every word in it.

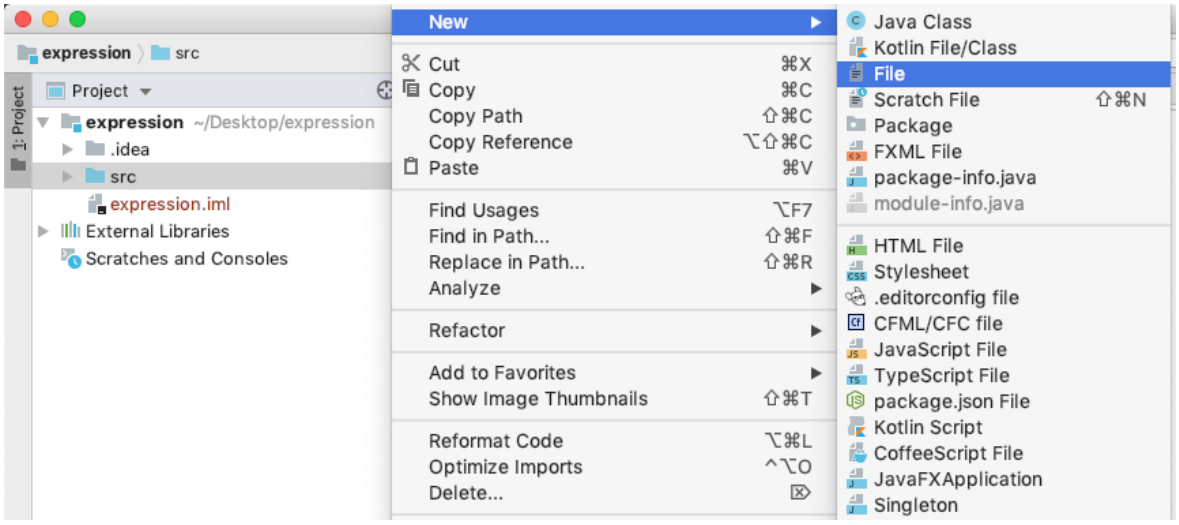
ANTLR and Intellij IDEA

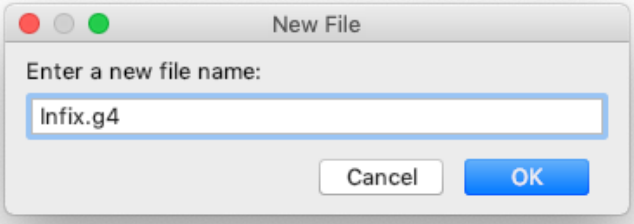
In this part of the assignment we will use ANTLR with Intellij IDEA to parse a simple arithmetic expression. If you do not have the ANTLR plugin installed, please refer to Appendix X.

Open Intellij IDEA, create a new project and name it `expression`.



Create a grammar file in the `expression` folder under `src` and name it `Infix.g4`, which is the file extension for version 4 of ANTLR.





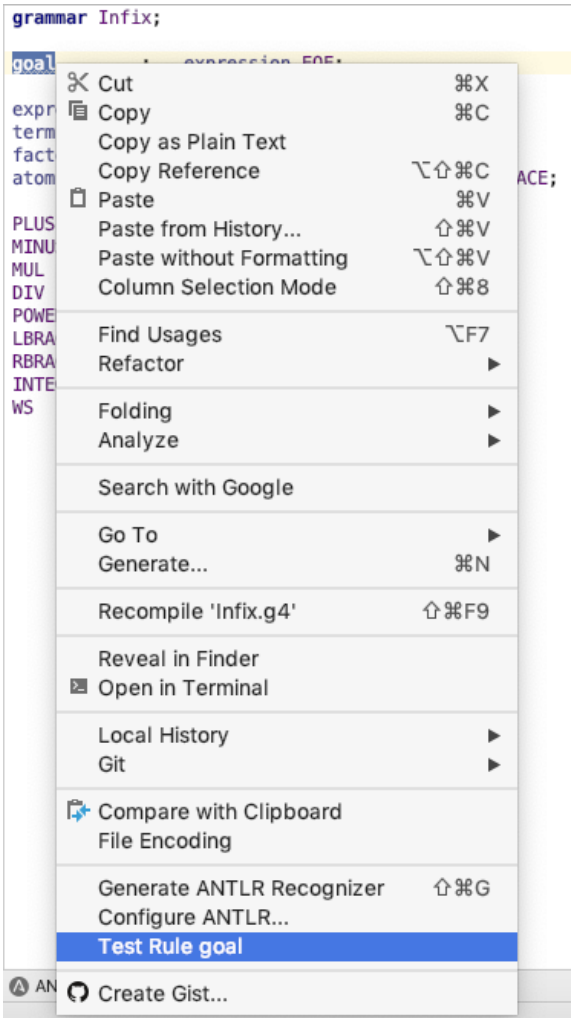
Now that you have created an empty grammar file, write down the parser rules with which our program will interact directly. Note that the name of the grammar file must match the name specified in the first line.

At the bottom of the window, you should see a button named ANTLR Preview.

When you click on ANTLR Preview, a new panel should appear which should display the following message:

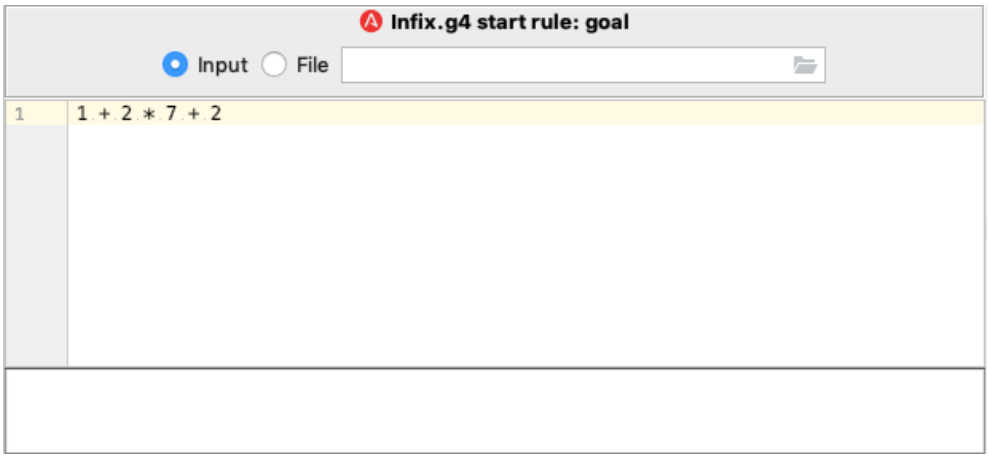
```
Infix.g4 start rule: <select from navigator or grammar>
```

To fix this, right click on the `goal` rule in the `Infix.g4` file and then select 'Test Rule goal'.



On the right-hand side of the ANTLR Preview panel, a default parse tree (which is how I refer to it) should appear. Note, ANTLR defaults to whatever the first alternative in the rule is; in our grammar, the start symbol `goal` defaults to an expression followed by the end-of-file marker.

Type in the expression $1 + 2 * 7 + 5$ in the text area on the left-hand side of the ANTLR Preview panel.



You should be able to see the parse tree created by ANTLR on the right-hand side of the same panel. Did it produce the correct parse tree? Why/why not (Hint: add the *integer* and *digit* rules to the grammar – see page #6 for reference. Make sure to remove the `INTEGER` lexical rule).

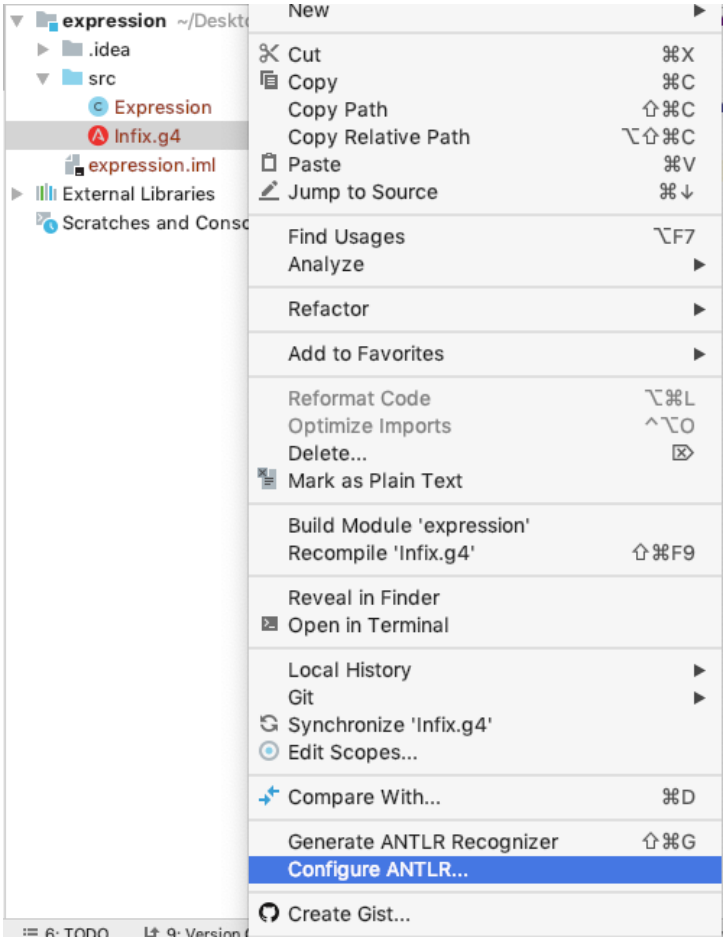
To verify that our grammar is correct, you should also try other arithmetic expressions, including invalid ones.

Writing the program

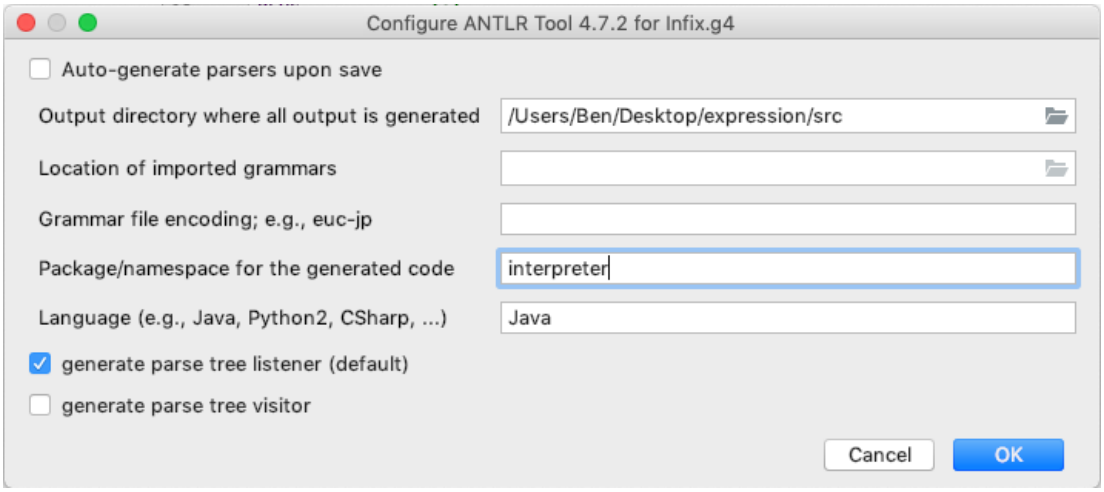
Now that we have our grammar file ready, we can begin writing our Java program to parse

arithmetic expressions. The goal would be to have our program accept strings and parse their content – we will not use any input stream file for this part of the assignment. Before we do this, we need to tell ANTLR where to generate code; that is, based on our grammar, ANTLR will create tokens, a parser, and lexer classes which we will integrate to our program.

Right click on the grammar file and choose 'Configure ANTLR'



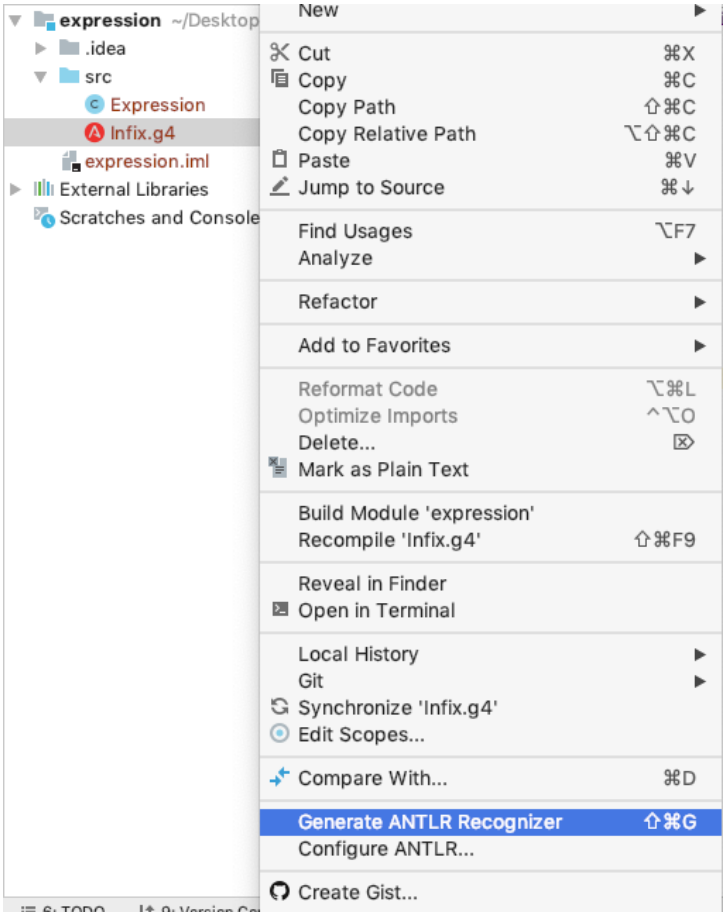
Enter the root path where your project's sources are saved, mine is `Users/Ben/Desktop/expression/src`. Then, enter the name of the package the parser should use, I named it `interpreter`, and the specified target language which should be Java. Click on 'OK' to continue.



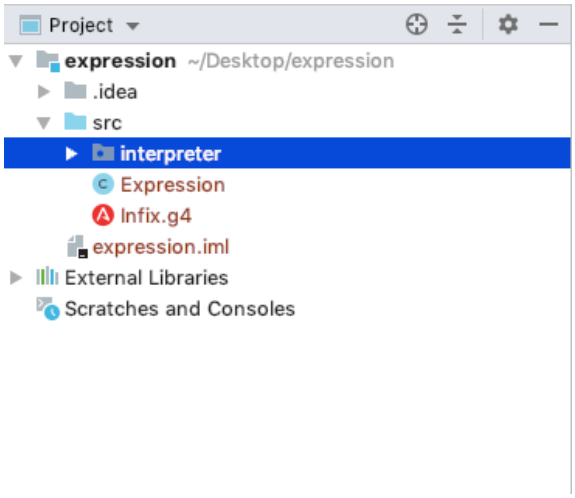
Note that Java must start with the uppercase letter J; otherwise, when generating the recognizer, you will see the following error message:

```
ANTLR cannot generate java code as of version 4.7.2
```

Click on the grammar file and choose 'Generate ANTLR Recognizer'.



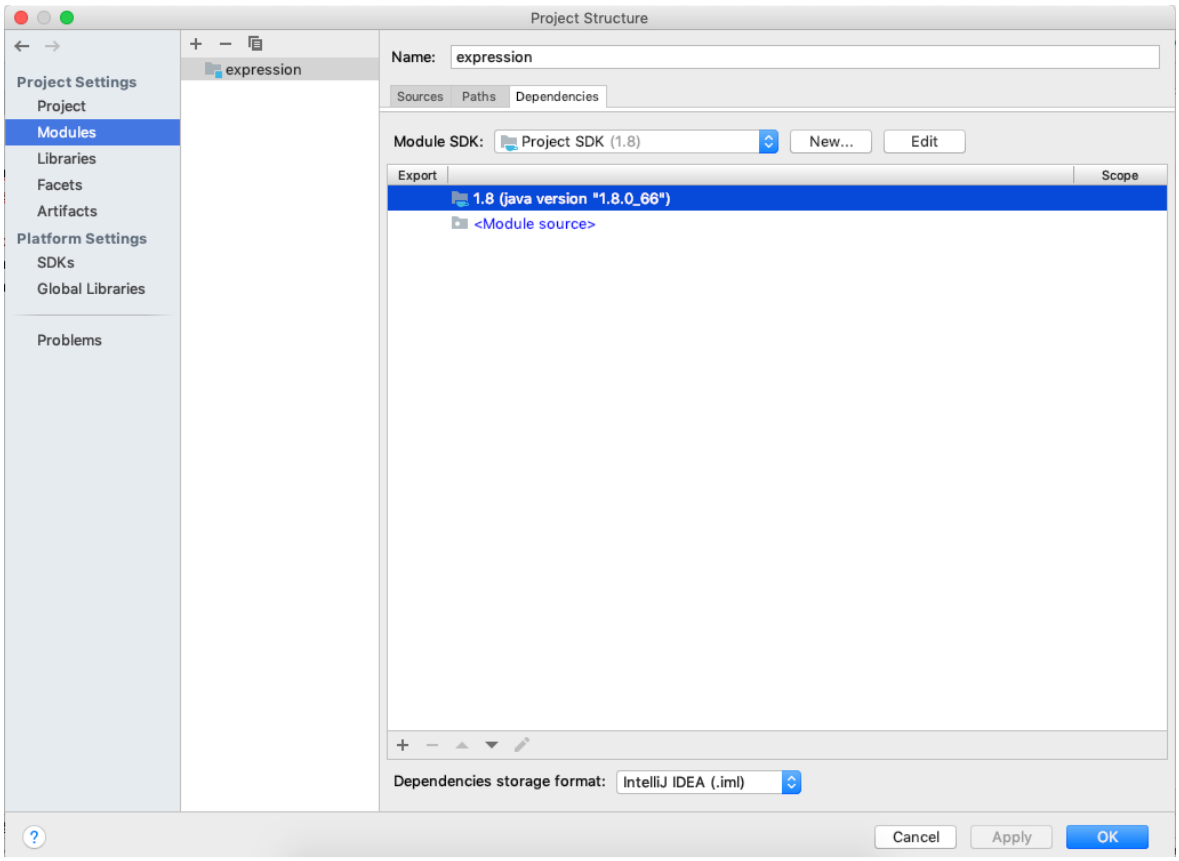
This should create a new folder, named `interpreter`, containing the ANTLR generated files. If everything went well, Intellij IDEA should recognize the folder.



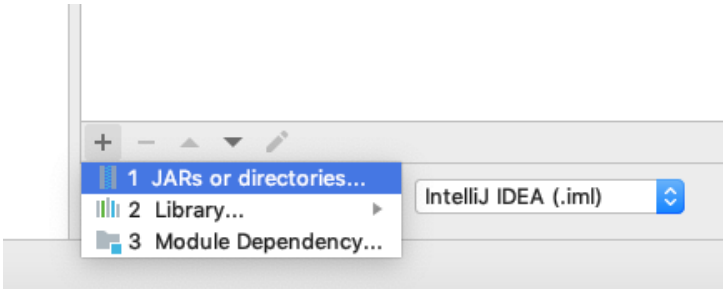
Before incorporating the ANTLR generated files into our program, we need to add the ANTLR runtime to our project. Download the ANTLR jar file from this website:

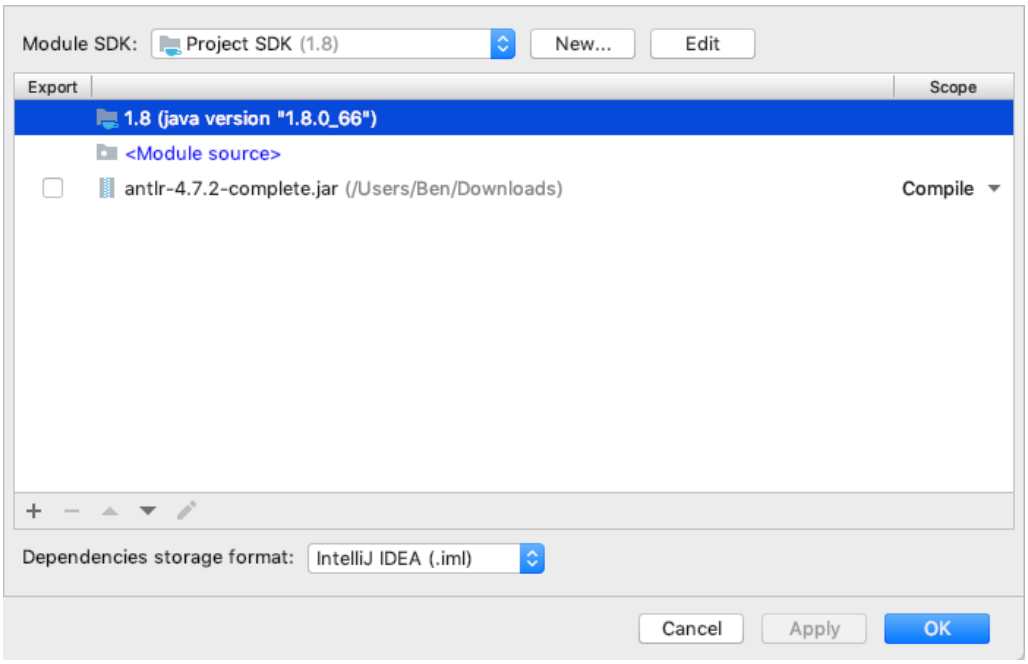
https://www.antlr.org/download/antlr-4.7.2-complete.jar

To add the jar to our project, go to `File`, then click on `Project Structure`. A new window should appear.

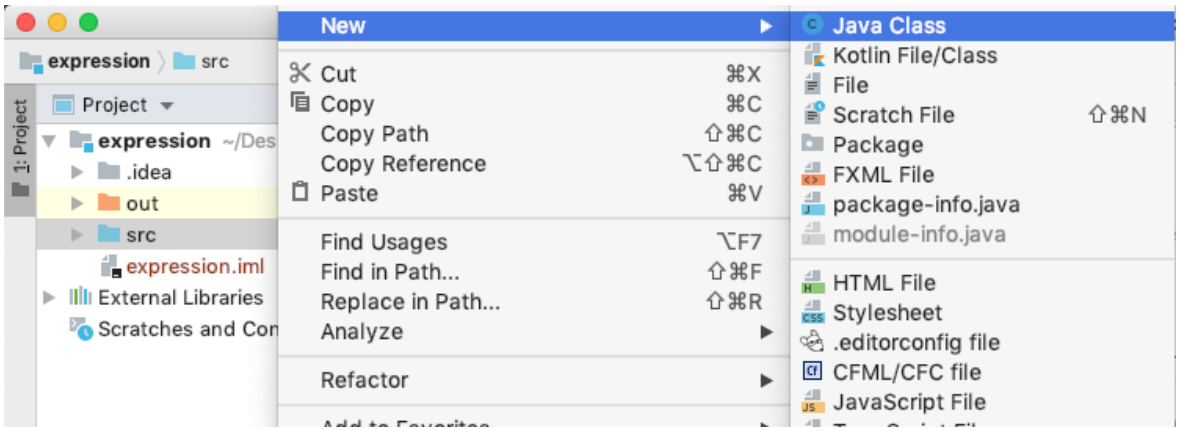


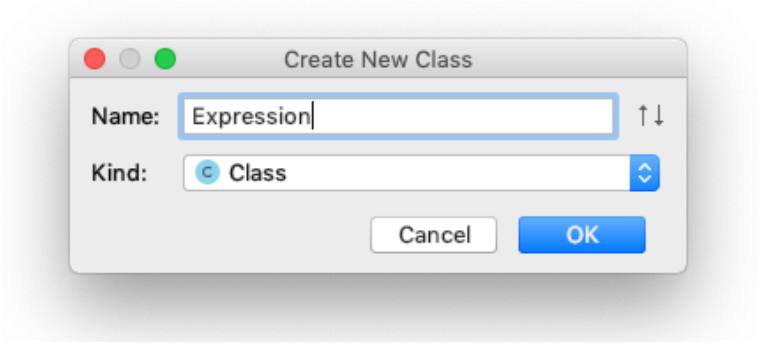
At the center bottom of the window, you should see a plus icon. Click on the plus icon and select 'JAR or directories...' to add the jar. You will need to browse to where the jar file was downloaded from the ANTLR website; in Mac systems, the jar should be in the Downloads folder.





Click on 'OK' to continue. Now that we have the ANTLR runtime in our project, create a Java class in the `expression` folder under `src` and name it `Expression`.





Next, write down the following code snippet.

```
import interpreter.InfixParser;
import org.antlr.v4.gui.Trees;
import org.antlr.v4.runtime.*;
import interpreter.InfixLexer;

public class Expression {

    public static void main(String[] args) {
        InfixLexer infixLexer =
                new InfixLexer(CharStreams.fromString("1 + 2 * 7 + 5"));
        InfixParser parser =
                new InfixParser(new CommonTokenStream(infixLexer));

        ParserRuleContext ruleContext = parser.goal();
        Trees.inspect(ruleContext, parser);
    }
}
```

In the code above, we start off by passing the string `"1 + 2 * 7 + 5"` to the lexer, named `infixLexer`, that was created when we let ANTLR generate code for us.

```
InfixLexer infixLexer =
            new InfixLexer(CharStreams.fromString("1 + 2 * 7 + 5"));
```

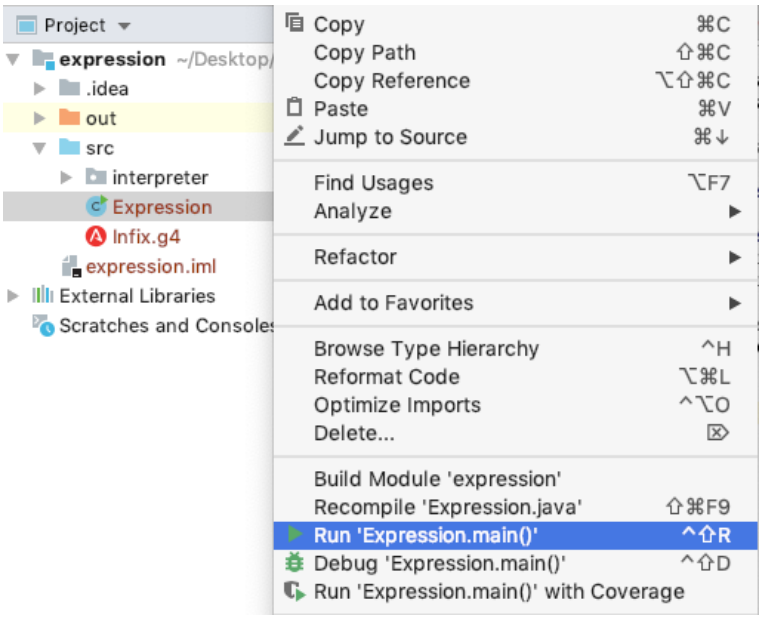
The lexer's results will be a stream of tokens which can be passed to the parser.

```
InfixParser parser =
            new InfixParser(new CommonTokenStream(infixLexer));
```

As an intermediate step, we can let ANTLR draw us a parser tree diagram – this is excellent for debugging and making sure the grammar is correct. To do this, we use the `ParserRuleContext` to track all of the tokens and rule invocations starting from the start symbol `goal`.

```
ParserRuleContext ruleContext = parser.goal();
Trees.inspect(ruleContext, parser);
```

Now, run the code to see the parse tree. Right click on the Expression file and choose 'Run Expression.main()'.



What do you see? Is this the correct parse tree?

What to Submit
Create a directory HW03 for this programming assignment. Once your code is tested and working, submit it along with your Phase #3 project for marks.

Remarks:

- **Your program must compile successfully; otherwise, you will receive a grade of 0.**
- **Assignments submitted after the due date/time will not be accepted.**

Challenge Problems
None for this assignment.