JavaCC21 Tutorial

Compilers - L.EIC 026

Pedro Pinto, Dr. Tiago Carvalho, Lázaro Costa and Dr. João Bispo

University of Porto/FEUP
Department of Informatics Engineering

version 1.1, March 2022

Contents

Dependencies and Materials to Get Started

Before you get started with this tutorial, make sure you have all the software dependencies installed and download all the needed files we are providing.

The software dependencies are Java 11+, Gradle 5+, and Git. Please note that there is a compatibility matrix for Java and Gradle versions that you should take into account. Other than that, you will need the base code, which you can find here. There are three important subfolders inside the main folder. First, inside the subfolder named `javacc` you will find the initial grammar definition. Then, inside the subfolder named `src` you will find the entry point of the application. Finally, the subfolder named `tutorial` contains code solutions for each step of the tutorial.

Once you've downloaded the materials, and should the dependencies be met, the project should work out-of-the-box through the command line interface. However, we suggest the use of an IDE as it will improve your productivity and ease the development of your applications. IDEs such as Eclipse, Idea or Visual Studio Code support the Gradle build system, albeit with the possible need of a plugin.

As far as we are aware there are no restrictions on which OS you can use, but we recommend using an up-to-date Linux distribution or Windows OS.

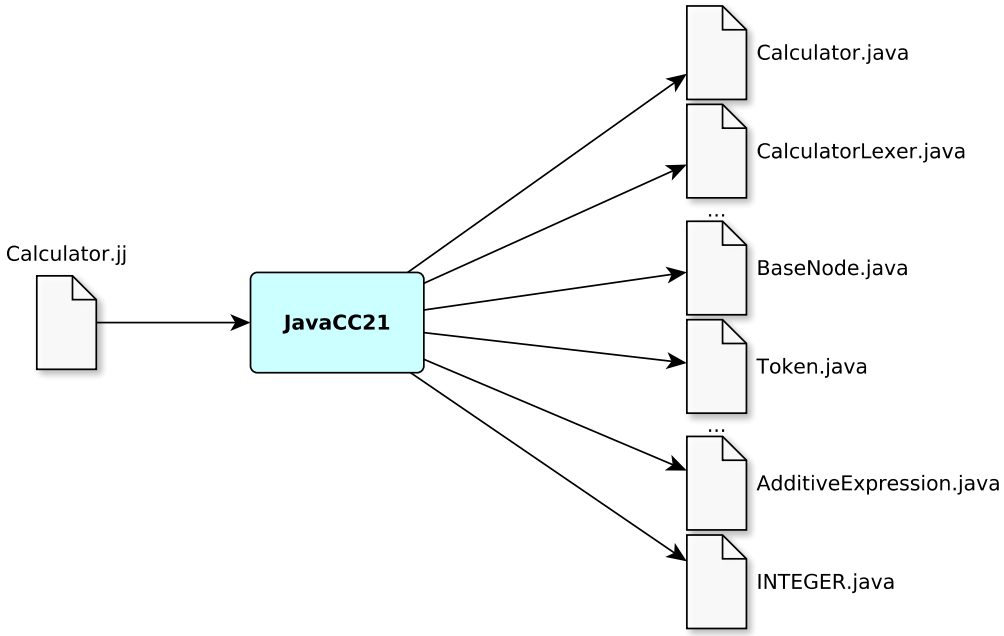

Figure 1: Tool flow of the JavaCC21 parser generator.

1 Parser Generation using JavaCC21

The goal of this tutorial is to introduce you to parser (syntactic analyzer) technology in Java, namely JavaCC21. To this end, you will build a tool to parse and evaluate simple arithmetic expressions. Previous knowledge about predictive top-down syntax analysis is helpful, but strictly not required.

JavaCC21[1] is a parser generator that, given a grammar specification, will generate a Java program[2] to parse a so called input string, matching it to the specified language, and, if successful, builds the corresponding parse tree respecting to the specific grammar.

JavaCC21 requires the description of the tokens and grammar rules in a specific text file (with a `jj` extension). The tool generates a set of Java classes, including one class with the same name as the parser. Figure 1 shows a generic view of the JavaCC21 flow, in which a file describing the grammar is given as input and the tool generates all the classes for the tree nodes and for parsing.

Unlike other parser generators (such as JavaCC), JavaCC21 builds a tree with nodes for the individual tokens. These nodes have attributes, namely `image`, which can be used to retrieve the token's text (*i.e.*, its character sequence), the name of an identifier or even the text representation of an integer (a string).

As for the grammar specification, it is captured in a text file, with four main sections:

- Options settings (optional): where the global settings are defined, including the name of the parser class and the package for the generated code;
- Token definitions: where we define each token, either literally or through a regular expression;
- Grammar rules: where we define the production rules of the grammar (accepts the symbols `+`, `*`, and `?` with the same meaning as in regular expressions);
- Code injection blocks: where we define arbitrary Java methods and attributes that will be injected into the generated classes.

[1] JavaCC21 is an evolution of another tool, JavaCC, with numerous improvements, such as new features, bug fixes, a saner default configuration, easier tool flow, and streamlined syntax.

[2] JavaCC21 can also generate a parser code in either Python or C#.

For more information you can check JavaCC21's website[3], GitHub[4], and wiki[5].

2 Simple Calculator Example

In this example we will develop a calculator for simple arithmetic expressions with integer numbers, which uses the syntax tree generated by the parser to perform the calculations.

Consider the following tokens and grammar rules:

LF	=	'\n'
INTEGER	=	[0-9]+
Start	→	AddExpression LF
AddExpression	→	MulExpression [('+' | '-') MulExpression]
MulExpression	→	Factor [('*' | '/') Factor]
Factor	→	INTEGER
	|	'-' Factor
	|	'(' AddExpression ')'

We specify this grammar in a file with the `jj` extension and in addition specify the name of the parser class and the package where the new classes will be generated. The following code contains the previous grammatical rules and is sufficient to generate ll the code for the parser and tree nodes. If desired, we can associate arbitrary Java code sections (between brackets) to each production which is then executed during execution of the function associated with that specific production.

```
PARSER_PACKAGE=pt.up.fe.comp;
PARSER_CLASS=SimpleCalculator;

SKIP : " " | "\t" | "\r" ;

TOKEN :
  < PLUS : "+" > |
  < MINUS : "-" > |
  < TIMES : "*" > |
  < DIVIDE : "/" > |
  < OPEN_PAREN : "(" > |
  < CLOSE_PAREN : ")" > |
  < INTEGER : (["0" - "9"])+ > |
  < LF : "\n" >
;

Start : AdditiveExpression <LF> ;

AdditiveExpression :
  MultiplicativeExpression
  (
    (< PLUS > | < MINUS >)
    MultiplicativeExpression
  )?
;

MultiplicativeExpression :
  Factor
  (
    (< TIMES > | < DIVIDE >)
    Factor
  )?
;

Factor :
  < INTEGER > |
  < MINUS> Factor |
  < OPEN_PAREN > AdditiveExpression < CLOSE_PAREN >
;
```

[3] JavaCC21's website: https://javacc.com/
[4] JavaCC21's GitHub: https://github.com/javacc21/javacc21
[5] JavaCC21's wiki: https://doku.javacc.com/doku.php

To test the calculator parser proceed as follows:

1. Go to the root directory of the provided code;
2. Compile and install the program, executing `gradle installDist`. This will compile your classes and create a launcher script in the folder `./build/install/comp2022-00/bin`. For convenience, there are two script files (in the root directory), one for Windows (`comp2022-00.bat`) and another for Linux (`comp2022-00`), that call the launcher script;
3. After compilation, a series of tests will be automatically executed. The build will stop if any test fails. Whenever you want to ignore the tests and build the program anyway, you can call Gradle with the flag `-x test`.

If everything works accordingly, your compilation output will be similar to this:

```
~\comp2022-00> gradle installDist

> Task :javacc21

JavaCC 21 Parser Generator
Go to https://javacc.com for more information.
(type "java -jar javacc.jar" with no arguments for help)


Outputting: ~\comp2022-00\generated\pt\up\fe\comp\SimpleCalculatorLexer.java
Outputting: ~\comp2022-00\generated\pt\up\fe\comp\SimpleCalculatorNfaData.java
Outputting: ~\comp2022-00\generated\pt\up\fe\comp\SimpleCalculatorConstants.java
Outputting: ~\comp2022-00\generated\pt\up\fe\comp\SimpleCalculator.java
Outputting: ~\comp2022-00\generated\pt\up\fe\comp\Node.java
Outputting: ..\generated\pt\up\fe\comp\INTEGER.java
Outputting: ..\generated\pt\up\fe\comp\Start.java
Outputting: ..\generated\pt\up\fe\comp\AdditiveExpression.java
Outputting: ..\generated\pt\up\fe\comp\MultiplicativeExpression.java
Outputting: ..\generated\pt\up\fe\comp\Factor.java
Parser generated successfully.

BUILD SUCCESSFUL in 4s
5 actionable tasks: 5 executed
```

Note that if you are having issues related to Java and/or Gradle, refer to the compatibility matrix for Java and Gradle to see if your versions are compatible. To see your Java version just run: `java -version`; and to see your Gradle version: `gradle --version`.

You can now test the compiled application. Just execute the `comp2022-00` script (or `comp2022-00.bat` if you are in Windows) and write an arithmetic expression, such as this:

```
1  ~\comp2022-00> .\comp2022-00.bat
2  Executing with args: []
3  2+1
4  Input code: 2+1
5
6  Start
7    AdditiveExpression
8      2
9      +
10     1
11
12
13 JmmNode interface not yet implemented, returning null root node
```

When you run the script without any additional arguments, you will notice that the program will halt. This is simply the program waiting for your arithmetic input. On the other hand you can simply provide the expression as an argument of the script, e.g., `comp2022-00.bat 2+1`.

Another observation is that there will be an exception saying "interface not implemented" (line 12 in the previous example). Do not worry as this is expected. It is something you will have to implement in the future.

The output you see in the console is called a *dump* of the syntax tree. This is a simple print of the tree created by your parser, in an hierarchical way. Despite being simple, it is a good way to understand the relationship between nodes (refer to `generated/pt/up/fe/comp/Node.java` file, function `dump`). If a `Node` is in fact a `Token`, then it will only contain the value of the Token, otherwise

it will be a `BasicNode` that can contain more complex artifacts, including custom properties and multiple child nodes.

Until now, the code generated by JavaCC21 has performed the following stages of the compiler:

1. Lexical: convert text into Tokens recognizable by the compiler. If a lexical error happens, then the input text is not recognized by the compiler;
2. Syntactical: verify if Tokens have a correct (and logical) structure based on the rules of the grammar. If so, those rules are used to create a syntax tree, with the Tokens as leaves.

After these testing steps, you can understand how to traverse the generated parse trees to compute numeric values associated with each arithmetic expression (*i.e.*, to evaluate), and thus validate the implementation.

Observation: after the lexical and syntactical stages, there are a lot of other important stages in a compiler cycle, such as semantic analysis, optimization, and code generation. This tutorial is fairly simple in its scope, so you will only touch on the very first stages. Later on, you will have contact with the other stages in the full project of the course.

3 Expression Evaluation

By relying on the information stored in the tree during the parsing step, you can develop a method to compute the value of the input arithmetic expression. We will define an evaluation function, `eval`, for each relevant node using the code injection mechanism of JavaCC21. Code injection takes user-defined code and inserts it into newly generated classes. The new grammar is as follows.

```
PARSER_PACKAGE=pt.up.fe.comp;
PARSER_CLASS=SimpleCalculator;

SKIP : " " | "\t" | "\r" ;

TOKEN :
  < PLUS : "+" > |
  < MINUS : "-" > |
  < TIMES : "*" > |
  < DIVIDE : "/" > |
  < OPEN_PAREN : "(" > |
  < CLOSE_PAREN : ")" > |
  < INTEGER : (["0" - "9"])+ > |
  < LF : "\n" >
;

Start : AdditiveExpression <LF> ;

AdditiveExpression :
  MultiplicativeExpression
  (
    (< PLUS > | < MINUS >)
    MultiplicativeExpression
  )?
;

MultiplicativeExpression :
  Factor
  (
    (< TIMES > | < DIVIDE >)
    Factor
  )?
;

Factor :
  < INTEGER > |
  < MINUS> Factor |
  < OPEN_PAREN > AdditiveExpression < CLOSE_PAREN >
;

INJECT Node :
{
    default int eval() {throw new UnsupportedOperationException();}
}

```

```
INJECT Start :
{
    public int eval() {
        return getChild(0).eval();
    }
}

INJECT AdditiveExpression :
{
    public int eval() {
        int result = getChild(0).eval();

        if (getChildCount() == 3) {

            boolean isAdd = getChild(1) instanceof PLUS;
            int nextOperand = getChild(2).eval();

            if(isAdd) {
                result = result + nextOperand;
            } else {
                result = result - nextOperand;
            }
        }

        return result;
    }
}

INJECT MultiplicativeExpression :
{
    public int eval() {
        int result = getChild(0).eval();

        if (getChildCount() == 3) {

            boolean isMult = getChild(1) instanceof TIMES;
            int nextOperand = getChild(2).eval();

            if(isMult) {
                result = result * nextOperand;
            } else {
                result = result / nextOperand;
            }
        }

        return result;
    }
}

INJECT Factor :
{
    public int eval() {

        switch (getChildCount()) {
        case 1: // just an integer, e.g., 2
            return getChild(0).eval();
        case 2: // a negated integer, e.g., -2
            return -1 * getChild(1).eval();
        case 3: // and expression surrounded by parenthesis, e.g. (2+3)
            return getChild(1).eval();
        default:
            throw new RuntimeException("Wrong no. of nodes in Factor.");
        }
    }
}

INJECT INTEGER :
{
    public int eval() {
        return Integer.parseInt(getImage());
    }
}
```

Finally, we can change the code that actually uses the parser to call this new evaluation function. Study the `Launcher` and `CalculatorParser` classes. In the latter class, you can add a call to the `eval` function at the root node. The code of the new `CalculatorParser` class should look like this:

```
package pt.up.fe.comp;

```

```java
import java.util.Arrays;
import java.util.Collections;

import pt.up.fe.comp.jmm.ast.JmmNode;
import pt.up.fe.comp.jmm.parser.JmmParser;
import pt.up.fe.comp.jmm.parser.JmmParserResult;
import pt.up.fe.comp.jmm.report.Report;
import pt.up.fe.comp.jmm.report.Stage;
import pt.up.fe.specs.util.SpecsIo;
import pt.up.fe.specs.util.SpecsLogs;


public class CalculatorParser implements JmmParser {

  @Override
  public JmmParserResult parse(String jmmCode) {

    try {

      SimpleCalculator parser = new SimpleCalculator(SpecsIo.toInputStream(jmmCode));
      parser.Start();

      Node root = parser.rootNode();
      root.dump("");
      System.out.println("eval: " + root.eval()); // new code

      if (!(root instanceof JmmNode)) {
        SpecsLogs.info("JmmNode interface not yet implemented, returning null root node");
        return new JmmParserResult(null, Collections.emptyList());
      }

      return new JmmParserResult((JmmNode) root, Collections.emptyList());

    } catch (Exception e) {
      return new JmmParserResult(null,
          Arrays.asList(
            Report.newError(Stage.SYNTATIC, -1, -1, "Exception during parsing", e)));
    }
  }
}
```

You can now experiment with other arithmetic expressions and test if the calculator parser is evaluating them correctly.

Note: The new code needed for this section can be found in `./tutorial/3/`.

4 Introduction to Jmm Interfaces

We will now see how we can start using the developed program code in a way that closely resembles a traditional compiler. Other similar tutorials start with a `main` function that is injected into the parser class to serve as an entry point. In the provided code, we deliberately did not do this. We separated the application launcher, the command line arguments parsing, the actual code parser and other tasks that will be implemented, as a compiler performs multiple tasks, of which parsing (the front end) is just one.

We built several interfaces, some of them presented here in this tutorial, that will be used in the main project of the course. The ones used in this tutorial will act as an introduction, and starting point, for the main structure of the project. During the semester you will learn more about the other interfaces. The launcher code, as well as other provided support code, will use several of the Jmm interfaces, which have the `JmmNode` interface in common.

4.1 JmmNode

`JmmNode` is an interface to a generic tree node, which is used internally in the provided support classes. Since this represents a generic node, it knows nothing about the parser that was used initially, providing decoupling and independence from specific tools. Therefore, all information of a specific node must be self-contained and it is accessed with `get` and `put` methods. The main `JmmNode` functions are:

```java
public interface JmmNode {

    String getKind();
    List<String> getAttributes();
    void put(String attribute, String value);
    String get(String attribute);
    List<JmmNode> getChildren();
    void add(JmmNode child, int index);
    /* ... */
}
```

The `getKind` function returns a string that represents the kind, or type, of a node, for instance `AdditiveExpression` or `Factor`. The other functions are used to deal with the attributes of each node, and children management. The easiest way to implement the `JmmNode` interface is to extend a default implementation, `AJmmNode` in this case, and then complete the remaining unimplemented methods. To achieve this, we will again rely on JavaCC21's injection mechanism, which we can use to define super classes as well as which interfaces are implemented. Add the following code at the end of your grammar file:

```
// Injections for the implementation of JmmNode
// Since this is injected into BaseNode, it affects only non-terminals
INJECT BaseNode :
    import pt.up.fe.comp.jmm.ast.JmmNode;
    import pt.up.fe.comp.jmm.ast.AJmmNode;
    import pt.up.fe.specs.util.SpecsCollections;
    extends AJmmNode
{
    @Override
    public List<JmmNode> getChildren() {

        return SpecsCollections.cast(children(), JmmNode.class);
    }

    @Override
    public void add(JmmNode child, int index) {

        if (child instanceof Node) {

            addChild(index, (Node) child);
        } else {

            throw new RuntimeException("Node " + child + " is not a Node.");
        }

    }
}

// Injections for the implementation of JmmNode
// Since this is injected into Token, it affects only terminals
INJECT Token :
    import pt.up.fe.comp.jmm.ast.JmmNode;
    import pt.up.fe.comp.jmm.ast.AJmmNode;
    import pt.up.fe.specs.util.SpecsCollections;
    extends AJmmNode
{
    @Override
    public List<JmmNode> getChildren() {

        return SpecsCollections.cast(children(), JmmNode.class);
    }

    @Override
    public void add(JmmNode child, int index) {

    throw new RuntimeException("Cannot add child nodes to a Token.");
    }

    @Override
    public List<String> getAttributes() {
        return SpecsCollections.concat("image", super.getAttributes());
    }

    @Override
    public String get(String attribute) {
    if(attribute.equals("image")) {
      return getImage();
```

```
    }

        return super.get(attribute);
    }
}
```

4.2 Visitors

Visitor is a well-known software design behavioral pattern[6]. It is used when we want to perform the same sort of operation on a group of similar objects. When we apply the pattern, we move that functionality from those objects to another class, which implements the visit function for each relevant object type.

The `JmmVisitor` interface, and some provided implementations, give you access to ready-to-use visitors that will ease certain tree-walking tasks. Our implementation is somewhat different from the original description of the pattern. First, we do not have an accept method in each relevant target class and instead we use a generic visit method. Second, we do not have a visit method for each type, instead we map node types (using the `getKind` function) to functions that will perform the desired operation.

We will reimplement the evaluation functionality using visitors. For each node kind we will describe what to do and how to proceed with the evaluation, that is, which nodes to visit afterwards. We can remove the evaluation functions from the grammar. The code for the new evaluation class is provided below and should be put inside the `src` folder, next to the `Launcher` and `CalculatorParser` classes.

```
package pt.up.fe.comp;

import pt.up.fe.comp.jmm.ast.JmmNode;
import pt.up.fe.comp.jmm.ast.AJmmVisitor;

public class VisitorEval extends AJmmVisitor<Object, Integer> {

    public VisitorEval() {

        addVisit("AdditiveExpression", this::addExprVisit);
        addVisit("MultiplicativeExpression", this::mulExprVisit);
        addVisit("Factor", this::factorVisit);
        addVisit("INTEGER", this::integerVisit);

        setDefaultVisit(this::defaultVisit);
    }

    private Integer addExprVisit(JmmNode node, Object dummy) {

        int result = visit(node.getJmmChild(0));

        if (node.getNumChildren() == 3) {

            boolean isAdd = node.getJmmChild(1).getKind().equals("PLUS");
            int nextOperand = visit(node.getJmmChild(2));

            if (isAdd) {
                result = result + nextOperand;
            } else {
                result = result - nextOperand;
            }
        }

        return result;
    }

    private Integer mulExprVisit(JmmNode node, Object dummy) {

        int result = visit(node.getJmmChild(0));

        if (node.getNumChildren() == 3) {

            boolean isMult = node.getJmmChild(1).getKind().equals("TIMES");
            int nextOperand = visit(node.getJmmChild(2));

```

[6] For further reading check *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma *et al.*

```
            if (isMult) {
                result = result * nextOperand;
            } else {
                result = result / nextOperand;
            }
        }

        return result;
    }

    private Integer factorVisit(JmmNode node, Object dummy) {

        switch (node.getNumChildren()) {
        case 1: // just an integer, e.g., 2
            return visit(node.getJmmChild(0));
        case 2: // a negated integer, e.g., -2
            return -1 * visit(node.getJmmChild(1));
        case 3: // and expression surrounded by parenthesis, e.g. (2+3)
            return visit(node.getJmmChild(1));
        default:
            throw new RuntimeException("Wrong no. of nodes in Factor.");
        }
    }

    private Integer integerVisit(JmmNode node, Object dummy) {

        return Integer.parseInt(node.get("image"));
    }

    private Integer defaultVisit(JmmNode node, Object dummy) {

        if (node.getNumChildren() != 2) { // Start has 2 children because <LF> is a Token Node
            throw new RuntimeException("Illegal number of children in node " + node.getKind() + ".");
        }

        return visit(node.getJmmChild(0));
    }
}
```

After this, we also change the `parse` function in the `CalculatorParser` class to remove the call to the evaluation function. This will be a task of the `Launcher` class, and its new code should look like this:

```
public class Launcher {
    /* ... */
    public static void main(String[] args) {
        SpecsSystem.programStandardInit();

        SpecsLogs.info("Executing with args: " + Arrays.toString(args));

        String input = parseArgs(args);
        SpecsLogs.info("Input code: " + input);

        // Instantiate JmmParser
        CalculatorParser parser = new CalculatorParser();

        // Parse stage
        JmmParserResult parserResult = parser.parse(input);
        /* visitor code */
        var eval = new VisitorEval();
        JmmNode root = parserResult.getRootNode();

        System.out.println("visitor eval: " + eval.visit(root, null));
        /* visitor code */

        // Check if there are parsing errors
        TestUtils.noErrors(parserResult.getReports());

        // ... add remaining stages
    }
    /* ... */
}
```

While this new evaluation class has more code than the previous `eval` function it is much more manageable and scalable than the previous alternative. As the grammar gets increasingly more complex and the number of types of tree node grows, the previous solution would result in a single large function with large switches and other tests to decide what to perform based on the node type.

The current implementation, although having a larger overhead, especially for small grammars, will result in more clean and maintainable code in the future, where the operations for each kind of node are clearly separated.

Note: The new code needed for this section can be found in `./tutorial/4/`.

5 Parse Tree Simplification and Abstract Syntax Trees

The syntax trees generated in the presented examples are designated as concrete syntax trees (they represent the derivations by the grammar productions faithfully). The simplification of these trees results in abstract syntax trees (ASTs).

The `#void` directive is used to avoid the generation of a node for each grammar production specified in the grammar file. These directives are placed after the names of the procedures which we do not want to represent as a node in the syntax tree. Other directives, such as `#BinOp(2)` and `#IntegerLiteral`, are used inside the grammar rules to generate nodes conditionally, for instance, due to arbitrary boolean expressions. This method allows us to automatically generate ASTs without requiring the manual transformation of the concrete syntax tree into an AST.

The following file presents a new version of the calculator grammar that generates ASTs. Note the new option at the top of the grammar file, which disables the generation of tree nodes for the tokens. Token nodes representing the binary operations, for instance `+`, and the integer literals are no longer needed, since we define exactly which nodes are on the tree. Also, we now annotate the new nodes with information that we will need at later stages, using the arbitrary Java code blocks to store that information using the `JmmNode` interface.

```
PARSER_PACKAGE=pt.up.fe.comp;
PARSER_CLASS=SimpleCalculator;
TOKENS_ARE_NODES=false; // tokens are no longer nodes in the tree


SKIP : " " | "\t" | "\r" ;

TOKEN :
  < PLUS : "+" > |
  < MINUS : "-" > |
  < TIMES : "*" > |
  < DIVIDE : "/" > |
  < OPEN_PAREN : "(" > |
  < CLOSE_PAREN : ")" > |
  < INTEGER : (["0" - "9"])+ > |
  < LF : "\n" >
;

Start : AdditiveExpression <LF> ;

AdditiveExpression #void :
  MultiplicativeExpression
  (
    (< PLUS > MultiplicativeExpression { jjtThis.put("op", "ADD"); }) #BinOp(2) |
    (< MINUS > MultiplicativeExpression { jjtThis.put("op", "SUB"); }) #BinOp(2)
  )?
;

MultiplicativeExpression #void :
  UnaryOp
  (
    (< TIMES > UnaryOp { jjtThis.put("op", "MUL"); }) #BinOp(2) |
    (< DIVIDE > UnaryOp { jjtThis.put("op", "DIV"); }) #BinOp(2)
  )?
;

void UnaryOp #void :
  (< MINUS > Factor { jjtThis.put("op", "NEG"); }) #UnaryOp(1) |
  Factor
;

Factor #void :
  (< INTEGER > { jjtThis.put("image", lastConsumedToken.getImage()); }) #IntegerLiteral |
  < OPEN_PAREN > AdditiveExpression < CLOSE_PAREN >
;
```

```
46
47 // Injections for the implementation of JmmNode
48 // Since this is injected into BaseNode, it affects only non-terminals
49 INJECT BaseNode :
50     import pt.up.fe.comp.jmm.ast.JmmNode;
51     import pt.up.fe.comp.jmm.ast.AJmmNode;
52     import pt.up.fe.specs.util.SpecsCollections;
53     extends AJmmNode
54 {
55     @Override
56     public List<JmmNode> getChildren() {
57
58         return SpecsCollections.cast(children(), JmmNode.class);
59     }
60
61     @Override
62     public void add(JmmNode child, int index) {
63
64         if (child instanceof Node) {
65
66             addChild(index, (Node) child);
67         } else {
68
69             throw new RuntimeException("Node " + child + " is not a Node.");
70         }
71
72     }
73 }
```

We will also change the evaluation visitor class and the reason is twofold. First, we now have new types of tree nodes and the visit function map needs to be updated to reflect that. Second, since we lost the access to the token nodes and their images (where we previously got the data), we now have to use the mechanisms of `JmmNode` to store and retrieve such information.

```
1  package pt.up.fe.comp;
2
3  import pt.up.fe.comp.jmm.ast.JmmNode;
4  import pt.up.fe.comp.jmm.ast.AJmmVisitor;
5
6  public class VisitorEval extends AJmmVisitor<Object, Integer> {
7
8      public VisitorEval() {
9
10         addVisit("IntegerLiteral", this::integerVisit);
11         addVisit("UnaryOp", this::unaryOpVisit);
12         addVisit("BinOp", this::binOpVisit);
13
14         setDefaultVisit(this::defaultVisit);
15     }
16
17     private Integer integerVisit(JmmNode node, Object dummy) {
18
19         if (node.getNumChildren() == 0) {
20
21             return Integer.parseInt(node.get("image"));
22         }
23
24         throw new RuntimeException("Illegal number of children in node " + node.getKind() + ".");
25     }
26
27     private Integer unaryOpVisit(JmmNode node, Object dummy) {
28
29         String opString = node.get("op");
30         if (opString != null) {
31
32             if (node.getNumChildren() != 1) {
33                 throw new RuntimeException("Illegal number of children in node " + node.getKind() + ".");
34             }
35
36             SimpleCalculatorOps op = SimpleCalculatorOps.fromName(opString);
37             switch (op) {
38             case NEG:
39                 return -1 * visit(node.getJmmChild(0));
40
41             default:
42                 throw new RuntimeException("Illegal operation '" + op + "' in " + node.getKind() + ".");
43             }
44         }
45
```

```
        if (node.getNumChildren() != 1) {
            throw new RuntimeException("Illegal number of children in node " + node.getKind() + ".");
        }

        return visit(node.getJmmChild(0));
    }

    private Integer binOpVisit(JmmNode node, Object dummy) {

        String opString = node.get("op");
        if (opString != null) {

            if (node.getNumChildren() != 2) {
                throw new RuntimeException("Illegal number of children in node " + node.getKind() + ".");
            }

            SimpleCalculatorOps op = SimpleCalculatorOps.fromName(opString);
            switch (op) {
            case MUL:
                return visit(node.getJmmChild(0)) * visit(node.getJmmChild(1));

            case DIV:
                return visit(node.getJmmChild(0)) / visit(node.getJmmChild(1));

            case ADD:
                return visit(node.getJmmChild(0)) + visit(node.getJmmChild(1));

            case SUB:
                return visit(node.getJmmChild(0)) - visit(node.getJmmChild(1));

            default:
                throw new RuntimeException("Illegal operation '" + op + "' in " + node.getKind() + ".");
            }
        }

        if (node.getNumChildren() != 1) {
            throw new RuntimeException("Illegal number of children in node " + node.getKind() + ".");
        }

        return visit(node.getJmmChild(0));
    }

    private Integer defaultVisit(JmmNode node, Object dummy) {

        if (node.getNumChildren() != 1) {
            throw new RuntimeException("Illegal number of children in node " + node.getKind() + ".");
        }

        return visit(node.getJmmChild(0));
    }
}
```

Finally, we need to add the enumeration that the evaluation class is using to represent the types of operations, *e.g.*, in the binary operation nodes, `BinOp`. This `enum` has the following code:

```
package pt.up.fe.comp;

import pt.up.fe.specs.util.SpecsEnums;

public enum SimpleCalculatorOps {

    ADD("+"),
    SUB("-"),
    MUL("*"),
    DIV("/"),
    NEG("-");

    private final String code;

    SimpleCalculatorOps(String code) {
        this.code = code;
    }

    @Override
    public String toString() {
        return code;
    }

    static SimpleCalculatorOps fromName(String name) {
        return SpecsEnums.fromName(SimpleCalculatorOps.class, name);
```

```
    }
}
```

Note: The new code needed for this section can be found in `./tutorial/5/`.

6 Optional Exercises

You can try these optional exercises as a way to both practice the concepts you've learned so far, and advance the state of the parser towards what will be the future project you will be developing.

6.1 Chained Expressions

The presented grammar does not accept arithmetic expressions that contain sequences of operations of the same type (e.g., 2+3+4 or 2*3*4). The only way to introduce expressions of this type is to use parenthesis to group the operations such as, for example, 2+(3+4) or 2*(3*4). The grammar modified to accept sequences of operations of the same type ("{...}" indicates 0 or more) is illustrated below.

Start	→	AddExpression LF
AddExpression	→	MulExpression { ('+' | '-') MulExpression }
MulExpression	→	Factor { ('*' | '/') Factor }
Factor	→	INTEGER
	|	'-' Factor
	|	'(' AddExpression ')'

Identify the problems that this grammar might introduce in the computation of the value of the expressions using the syntax tree.

6.2 Symbol Assignments

Modify the previous grammar so that it accepts an arbitrary number of integer assignments to symbols and an arbitrary number of expressions. The symbols should be used in the evaluation of the arithmetic expressions, as illustrated by the following example:

```
A = 2; B = 3; A * B; A + B;
```

The evaluation of the previous input should print 6 and 5. First, consider that these symbols are defined by the regular expression `[A-Za-z][0-9A-Za-z]*`. Then, update the evaluation function to correctly compute the value of the arithmetic expressions.