ANTLR Tutorial

Compilers - L.EIC026 - 2022/2023

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

University of Porto/FEUP
Department of Informatics Engineering

version 1.3, February 2023

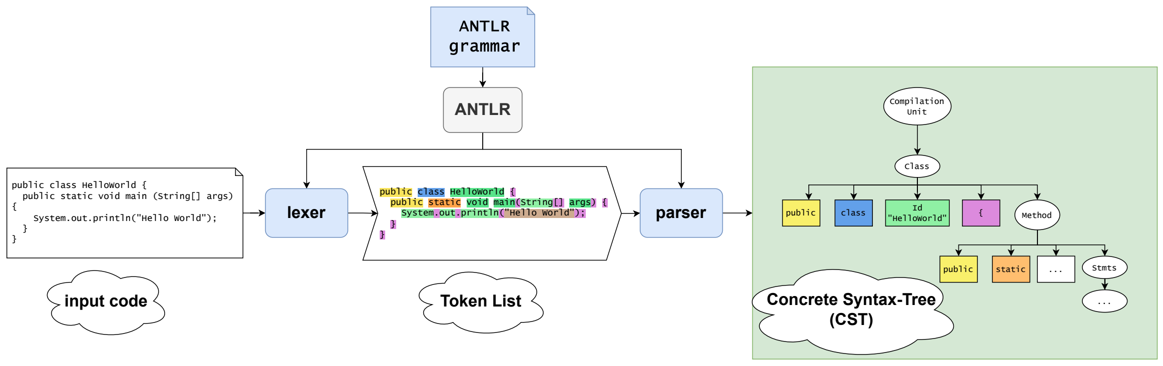
Contents

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 in this repository. There are three important subfolders inside the main folder. First, inside the subfolder *src/main/antlr/comp2023/grammar*, you will find the initial grammar definition. Then, inside the subfolder named *src/main/pt/up/fe/comp2023*, you will find the entry point of the application. Finally, the subfolder named *tutorial* contains code solutions for several steps of the tutorial.

Once you've downloaded the materials, and the dependencies are 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,



```
public class HelloWorld {
  public static void main (String[] args)
{
    System.out.println("Hello World");
  }
}
```

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World");
  }
}
```

Figure 1: Flow of a parser specified in ANTLR.

Idea, or Visual Studio Code, support the Gradle build system, albeit with the possible need for 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.

1 Parser Generation using ANTLR

The goal of this tutorial is to introduce you to parser technology (lexical and syntactic analyzer) in Java, namely ANTLR, in the context of the Compilers 2022/2023 project. To this end, you will build a tool to parse simple Java-based expressions. Previous knowledge about predictive top-down syntax analysis is helpful, but strictly not required.

ANTLR is a parser generator that, given a grammar specification, will generate a Java[1] program to parse a so-called input string, matching it to the specified language, and, if successful, building the corresponding parse tree with respect to the specific grammar. This flow can be seen in Figure 1.

ANTLR requires the description of the tokens and grammar rules in a specific text file (with a *g4* extension) to generate a set of Java classes. For instance, for a grammar named *Example*, defined in *Example.g4*, ANTLR will generate files *ExampleLexer.java* and *ExampleParser.java*. The lexer, generated based on the grammar, verifies the input file or text (e.g. a Java code as in Figure 1) and converts the contents into a set of tokens.

Unlike other parser generators (such as JavaCC), the parser generated by ANTLR builds a concrete-syntax tree (CST), with nodes for both rules and for the individual tokens. These nodes have attributes, namely *text*, which can be used to retrieve the token's text (its character sequence), for instance, the name of an identifier or even the text representation of an integer.

As for the grammar specification, it is defined inside a *g4* text file, with four main sections:

- **Grammar name**: must be the same as the file name and is used as a prefix of every generated class name;
- **Options and imports**: where we define global options and can import grammars (similar to grammar inheritance);
- **Token rules**: where we define each token, either literally or through a regular expression;
- **Parser rules**: where we define the production rules of the grammar (accepts the symbols +, *, and ? with the same meaning as in regular expressions).

These can come in any order, but we recommend that you follow this order to keep your code organized. The only mandatory elements are the grammar name and at least 1 (parser) rule definition. Everything else is optional, although a fully working compiler will likely need all those features.

[1]ANTLR can also generate parser code in other languages, including Python, C#, and JavaScript.

Parser rule names must start with a lowercase letter and lexer rules must start with a capital letter. A good standard to follow is to use camelCase for parser rules and UPPERCASE for lexer rules. It is also possible to include arbitrary target-language code (Java, in our case) within curly brackets.

For more information you can check ANTLR's website, documentation, and FAQ.

2 Simple Expression Example

In this example we will develop a simple parser for expressions such as those found in multiple imperative languages. This parser will be automatically generated based on the grammar specification as ANTLR rules.

Consider the following EBNF(Extended Backus–Naur Form)[2] definition of this example's token and grammar rules. Note that concatenation is performed by simple juxtaposition of the elements and not with the comma operator.

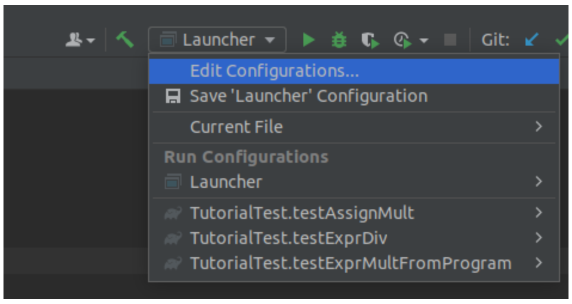
INT	::=	`[0-9]+`
ID	::=	`[a-zA-Z_][a-zA-Z_0-9]*`
statement	::=	expression ';'
	|	ID '=' INT ';'
expression	::=	expression '+' expression
	|	expression '-' expression
	|	expression '*' expression
	|	expression '/' expression
	|	INT
	|	ID

This is a simple grammar that accepts two types of statements: an assignment or a single expression (usually called an expression statement). The expression rule specifies simple arithmetic (binary) operations, and the use of a constant value (INT) or a variable (ID).

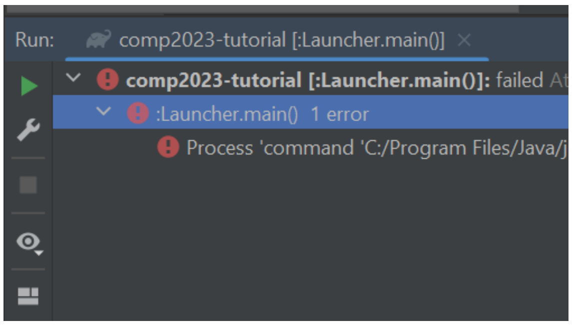
We specify this grammar in a file with the *g4* extension and in addition specify the name of the grammar and the package where the new classes will be generated. The following code contains the previous grammatical rules and it is sufficient to generate all the code for the parser and tree nodes. If desired, we can associate arbitrary Java code sections, between brackets, to each production. Then, this code will run during execution of the function associated with that specific production.

```
grammar Javamm;

@header {
    package pt.up.fe.comp2023;
}

INTEGER : [0-9]+ ;
ID: [a-zA-Z_][a-zA-Z_0-9]* ;

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

program
    : statement EOF
    ;

statement
    : expression ';'
    | ID '=' INTEGER ';'
    ;

expression
    : expression '+' expression
    | expression '-' expression
    | expression '*' expression
    | expression '/' expression
    | INTEGER
    | ID
    ;
```

[2] https://en.wikipedia.org/wiki/Extended_Backus-Naur_form

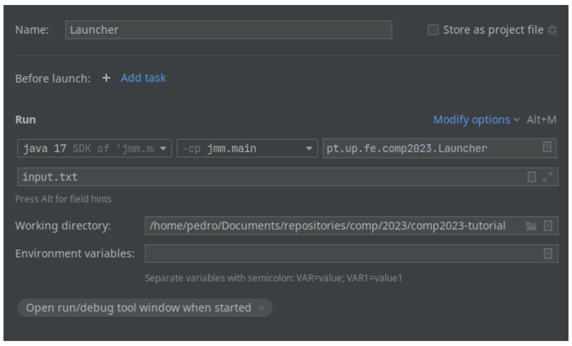


(a) How to edit the configuration



```
Run:   comp2023-tutorial [:Launcher.main()]
comp2023-tutorial [:Launcher.main()]: failed At
  :Launcher.main()  1 error
    Process 'command 'C:/Program Files/Java/j
```

(b) Filtering output



Name: Launcher

Store as project file

Before launch: + Add task

Run

Modify options Alt+M

java 17 SDK of 'jmm.m | -cp jmm.main | pt.up.fe.comp2023.Launcher

input.txt

Press Alt for field hints

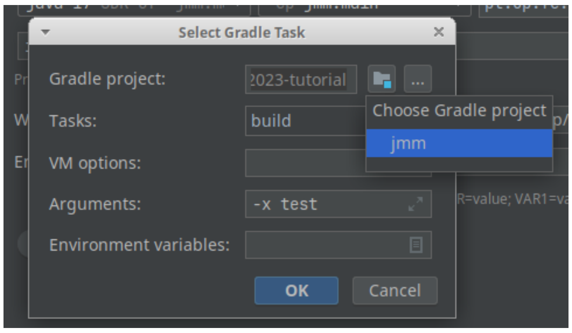
Working directory: /home/pedro/Documents/repositories/comp/2023/comp2023-tutorial

Environment variables:

Separate variables with semicolon: VAR=value; VAR1=value1

Open run/debug tool window when started

(c) Main class definition



Select Gradle Task

Gradle project: 2023-tutorial

Tasks: build

VM options:

Arguments: -x test

Environment variables:

Choose Gradle project
jmm

(d) Gradle build dependency

Figure 2: Adding an execution configuration for this project in IDEA.

We have now two ways of working with our project: via an IDE and via command line. If you prefer to use an IDE, we suggest the use of IntelliJ IDEA, as it is able to automatically detect the projects compilation configuration and it also provides a plugin for ANTLR. For IntelliJ IDEA instructions, please refer to Section 2.1. If you prefer to execute the program in a terminal, please refer to Section 2.2.

2.1 Working with IntelliJ Idea

Although this setup should work out-of-the-box and is ready for development, we strongly encourage you to use an IDE such as **IntelliJ Idea**. This IDE provides full integration with the Gradle build system and good plugin for ANTLR.

If you decide to work with IDEA, you can setup a running configuration such as the one presented in Figure 2c. To obtain that configuration you just need to go to the main class (Launcher.java) and you will see a "play" (▷) button beside the main method. When you click it, it will create a new running configuration for that main method. In principle, IntelliJ will understand the compilation flow (by means of the gradle configuration) and automatically compile everything for you, and it will automatically recompile every time you make changes in the code (or even in the grammar). When running for the first time, it will give an error similar to the following:

```
> Task :Launcher.main() FAILED
Exception in thread "main" Executing with args: []
java.lang.RuntimeException: Expected a single argument, a path to an existing input file.
  at pt.up.fe.comp2023.Launcher.parseArgs(Launcher.java:56)
  at pt.up.fe.comp2023.Launcher.main(Launcher.java:21)
```

If you do not see this error, it might have to do with the way IntelliJ filters the output. If you look at the bottom-left corner, you should see a window similar to Figure 2b. Depending on what item is selected, you will see more or less output. The further down the item tree, the less output appears. Select the item that says `Launcher.main()` and you should see the error message.

This error is expected since, as the message says, we need to pass a input file to the compiler to process it. This is as simple as editing the running configuration you have created. To do that, simply go to the top right toolbox with the name "Launcher" in it (see Figure 2a), press the drop-down button, and click "Edit Configurations". It will then open a window similar to Figure 2c. Then, you just need to add "input.txt" to the "Program Arguments" field and press "Ok". The execution should work fine now!

The program will print the parse tree to the terminal related to the input file and, additionally, creates a GUI preview of that same tree. If input file had as its content the expression `1+2*3`, you will see a tree similar to the present in Figure 3a. You can change the input in the file *input.txt* and check if the generated tool is able to parse it and how the tree looks.

Note: If for some reason the compilation fails, or during execution the program gives problems related to the compilation, you can try the following simple fix: Click on *Before launch: Add Task* to add a task to be performed before this program is execute, then specify the task as shown in Figure 2d. This way it will always use the most up-to-date version of the code to compile our project.

Although we only provide this configuration for IntelliJ IDEA, feel free to use any other IDE you feel comfortable with, as they all should support this sort of execution configuration.

2.2 Working in a Command Line

It is also possible to work through the command line interface. To test the generated 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 JAR and a launcher script in the folder *./build/install/jmm/bin/*. For convenience, there are two running script files (in the root directory), one for Windows (*jmm.bat*) and another for Linux (*jmm*), that call the launcher script;
3. After compilation, it is possible that 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`. **Note:** Only ignore the tests in sporadic situations. For normal development keep the tests running.

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

```
comp2023-tutorial > gradle installDist

BUILD SUCCESSFUL in 998ms
5 actionable tasks: 5 executed
comp2023-tutorial >
```

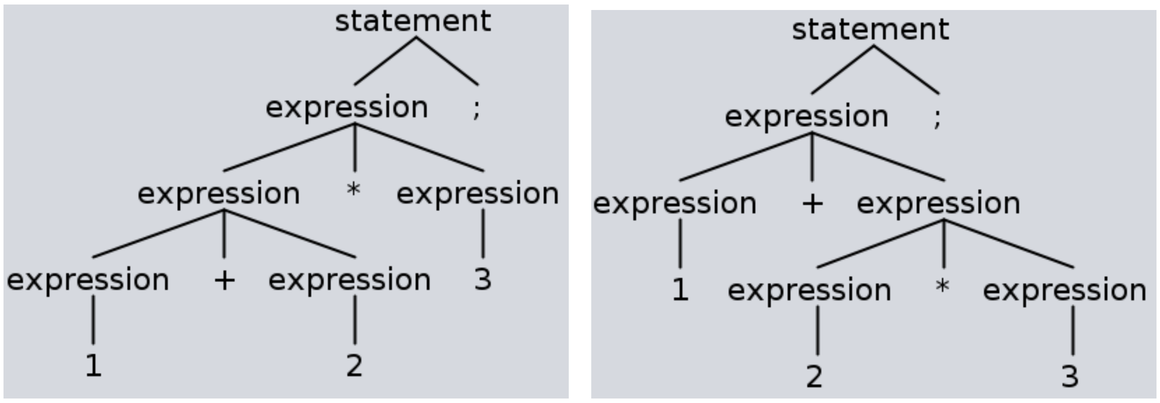
To execute the generated parser you can call the running script. For instance, in Linux, you can do:

```
comp2023-tutorial > ./jmm input.txt
Executing with args: [input.txt]
(statement (expression (expression (expression 1) + (expression 2)) * (expression 3)) ;)
comp2023-tutorial >
```

The script takes a single argument, which is a file with the input code. In the above example, the input file had as its content the expression `1+2*3`. This example program prints the parse tree to the terminal and, additionally, creates a GUI preview of the tree. You can change the input in the file *input.txt* and check if the generated tool is able to parse it and how the tree looks.

3 Precedences

As you can see from the tree generated from the initial example, shown in Figure 3a, this grammar does not take care of operator precedences. As the evaluation of the tree is performed bottom-up,



(a) incorrect tree

(b) correct tree

Figure 3: Parse trees for the input `1+2*3;`.

you can see that the addition will be evaluated before the multiplication, that is, in order in which they appeared in the input.

ANTLR gives priority to alternatives that are defined earlier, which means that if we want multiplication to have precedence, that specific alternative needs to be defined at the top. Change your grammar to reflect this and check how it affects the generated parse tree. It should look like the one in Figure 3b.

```
expression
    : expression '*' expression
    | expression '/' expression
    | expression '+' expression
    | expression '-' expression
    | INTEGER
    | ID
    ;
```

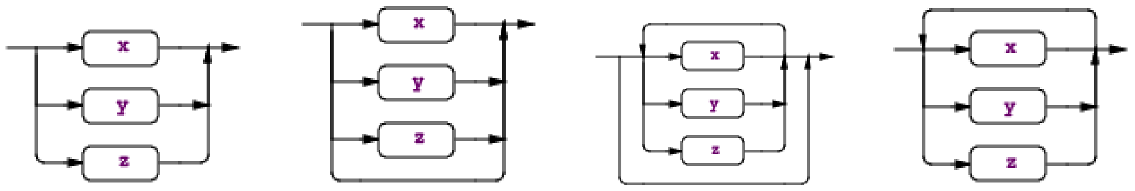
Now consider the input expression `1/2*3;`. What do you think will happen with the current grammar? And what should happen? Although the division and multiplication operations have the same precedence, our grammar prioritizes the multiplication (since it is specified before the division). This means that the tree prioritizes the multiplication (as it is closer to the bottom of the tree) and will perform it first, even though the division appears before in the input. This means that one of the operators has priority over the other, depending on their relative position, even though they should have the same precedence.

4 Subrules

Like EBNF, ANTLR has the concept of subrules. These are unnamed alternative blocks within a rule, and provide 1 or more alternatives surrounded by parenthesis. For instance, `(x|y|z)`, is a subrule that presents three alternatives, x, y, and z. It is important to note that the alternatives within a subrule have the **same priority** and only one will occur.

With subrules we can fix our grammar so that certain groups of operators have priority over others, but share the same priority within the group. Change the *expression* rule in your grammar with this in mind.

```
expression
    : expression ('*' | '/') expression
    | expression ('+' | '-') expression
    | INTEGER
    | ID
    ;
```



(a) `(x|y|z)` (b) `(x|y|z)?` (c) `(x|y|z)*` (d) `(x|y|z)+`

Figure 4: The types of subrules supported by ANTLR. Images sourced from here.

Now we have two grammars that syntactically accept the same input text, but provide different parse trees. In the latter one, we have alternatives within alternatives, which allows us to group operator precedence and generate parse trees that reflect this semantic constraint.

ANTLR subrules, like in EBNF, support modifiers to express optionals and repetitions. These modifiers are summarized in Figure 4.

In ANTLR, if a subrule has a single alternative, the parenthesis become optional. That means that `(alt)+` is equivalent to `alt+`. Let us use this knowledge to make a small change to the grammar. Suppose we want our program to accept several statements instead of a single one. We can change the *program* rule to say that it has 1 or more statements. The complete grammar so far looks like this:

```
grammar Javamm;

@header {
    package pt.up.fe.comp2023;
}

INTEGER : [0-9]+ ;
ID : [a-zA-Z_][a-zA-Z_0-9]* ;

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

program
    : statement+ EOF
    ;

statement
    : expression ';'
    | ID '=' INTEGER ';'
    ;

expression
    : expression ('*' | '/') expression
    | expression ('+' | '-') expression
    | INTEGER
    | ID
    ;
```

5 Introduction to Jmm Interfaces

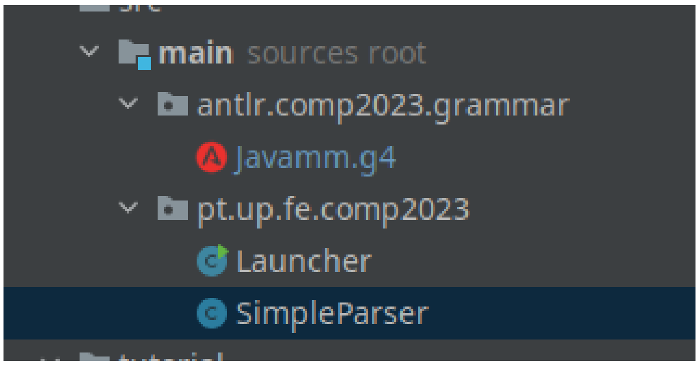


Figure 5: Project directory after copying and replacing the files in *src* with the files in *tutorial/src/jmm*.

We will now see how we can start using the developed program code in a way that closely resembles a traditional compiler. We separated the application launcher, the command line arguments parsing, the actual code parser and other tasks that will be implemented, of which parsing is just (the front end) 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 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.

Find the code inside the directory *./tutorial/src/2.jmm/*. Replace the launcher that you have been using so far with these two new files. Your project directory should look similar to the one in Figure 5. This launcher calls *SimpleParser*, a concrete implementation of the *JmmParser* interface, which has all the code necessary to call the parser files generated by ANTLR. Additionally, you can also see a call to `AntlrParser.parse(parser, startingRule)`. Inside that method we have the actual call to the parsing class with the defined starting rule and, importantly, the conversion from ANTLR tree nodes to *JmmNodes*.

5.1 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 *statement* or *expression*. The other functions are used to deal with the attributes of each node and children management.

During the development of your project you will use *JmmNode*, but you do not need to provide a concrete implementation of this interface, as the nodes are automatically built for you.

5.2 JmmParser

JmmParser provides a generic parser interface with simple methods for parsing and another method to define the default starting rule. Certain parser generators, like ANTLR, provide methods to start parsing at any grammar rule, not just the top-most one. By default, our implementation of *JmmParser* starts at the *program* rule. You can either change this or call the parsing function with a specific rule name.

One of the key features here is that the parse methods return an instance of *JmmParserResult*, another interface that should help your development. The key point here is that this interface codifies a generic parsing result, with a list of reports (which can be errors or warnings) and a root node. You can see how this result is built differently depending on whether the parsing was successful or not.

6 From Concrete Syntax Trees to Abstract Syntax Trees

The parse trees generated in the examples up to Section 5 are designated as concrete syntax trees (CSTs), as they represent the derivations by the grammar productions faithfully. The simplification of these trees results in abstract syntax trees (ASTs).

The conversions from ANTLR nodes to JmmNode in Section 5 purposefully drop certain nodes from the CST, namely the terminal nodes. For instance, the tokens representing the operations, such as `+` or `/` are no longer present in the tree. Likewise for the nodes representing integers or identifiers. In order to maintain this information, we need to annotate the nodes with rule element labels.

Consider the multiplication expression alternative. We can label the element of the rule that has the operator token as such: `expression:  expression op=('*' | '/') expression`. The node that is generated for this expression will have an attribute named *op* that has a string with the specific operator. Change the grammar to annotate the tree with operator information and rerun the previous example. See how the tree has changed.

```
expression
    : expression op=('*' | '/') expression
    | expression op=('+' | '-') expression
    | value=INTEGER
    | value=ID
    ;
```

The tree node of the expression now has the information (the *op* attribute) about the operation to perform. Additionally, we also annotate the elements of the last two alternatives, for integers and identifiers.

There is one more thing that we can change in our grammar to further improve our final tree by making it more tailored to our needs. Consider the expression rule, which has four different alternatives. An input such as the following:

```
1+2;
3*4;
a;
```

will generate a tree that looks like this:

```
Program
   Statement
      Expression (op: '+')
         Expression (value: 1)
         Expression (value: 2)
   Statement
      Expression (op: '*')
         Expression (value: 3)
         Expression (value: 4)
   Statement
      Expression (value: a)
```

Note how every operation and even the single identifier in the last line of the input are all expressions. ANTLR has a feature, called alternative labels, that allows us to label each alternative in a rule with a name of our choice. Be careful because you must either label all alternatives or no alternative at all. The following grammar labels all the expression alternatives. The first two are binary operations (which are differentiated based on the *op* attribute), and the last two are the integers and identifiers, respectively.

```
grammar Javamm;

@header {
    package pt.up.fe.comp2023;
}

INTEGER : [0-9]+ ;
ID : [a-zA-Z_][a-zA-Z_0-9]* ;

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

program
    : statement+ EOF
    ;

statement
    : expression ';'
    | ID '=' INTEGER ';'
    ;

expression
    : expression op=('*' | '/') expression #BinaryOp
    | expression op=('+' | '-') expression #BinaryOp
    | value=INTEGER #Integer
    | value=ID #Identifier
    ;
```

Note how the tree now looks for the same input. It looks cleaner and a lot easier to navigate, as each node has a more specific type and all the needed information.

```
Program
   Statement
      BinaryOp (op: '+')
         Integer (value: 1)
         Integer (value: 2)
   Statement
      BinaryOp (op: '*')
         Integer (value: 3)
         Integer (value: 4)
   Statement
      Identifier (value: a)
```

7 Working with JmmNode and JmmVisitors

Up to this moment, we have been looking at how to obtain an AST (with nodes of type *JmmNode*) from a given input file. Obviously the work of a compiler is more than simply obtain an AST. A compiler can have multiple stages, such as semantic analysis, code transformation, code optimization, code generation and/or synthetization, binary production, etc. During the development of the main project of the course you will have experience with some of those stages, and, for about half of the project, you will work with the *JmmNode* interface. You will have to know how to use it, how to access a node's attributes, and how to visit its children nodes.

In order for you to have a quick hands-on experience with *JmmNode*, and to have a full compilation flow in this compiler, we will now build a new stage. We will add a component that receives the AST as input and prints a Java class that contains code to execute the contents of the original input file. The idea is to convert:

1. the assignments into variable declarations with initializations;
2. the expressions into prints to the console (`System.out.println`) that outputs the actual value of the operation.

So when we run that generated Java class, it will output the operations written in the input file. First of all, the grammar that we will be using is the one below. The main difference to the previous grammar is that we now store the first production of *statement* as an *ExprStmt* node, and the second one as a *Assignment*. This allows us to easily differentiate the type of statements in the tree. Also, in the *Assignment* rule, we are also storing the *ID* in a *var* property, and the integer value in a *value* property.

```
grammar Javamm;

@header {
    package pt.up.fe.comp2023;
}


INTEGER : [0-9]+ ;
ID : [a-zA-Z_][a-zA-Z_0-9]* ;

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

```

program
    : statement+ EOF
    ;

statement
    : expression ';' #ExprStmt
    | var=ID '=' value=INTEGER ';' #Assignment
    ;

expression
    : expression op=('*' | '/') expression #BinaryOp
    | expression op=('+' | '-') expression #BinaryOp
    | value=INTEGER #Integer
    | value=ID #Identifier
    ;
```

To generate the code, we will have to traverse the AST, from top to bottom. This means that we will have to go node by node, and access their children until we are at the leaves of the tree (that is, the number of children of the node is 0). To help traverse any *JmmNode* tree, we provide the *JmmVisitor* interface, and three implementations: *AJmmVisitor*, *PreorderJmmVisitor* and *PostorderJmmVisitor*. *PreorderJmmVisitor* and *PostorderJmmVisitor* automatically traverse the AST completely, with a pre-order[3] or post-order[4] algorithm. In this example, we will be using the abstract class *AJmmVisitor*, which requires the user to visit the child nodes manually.

We have to extend the *AJmmVisitor* abstract class, and this class requires two generic types to be specified, the input type and the output type. The input type is the second argument that the *visit rule* will receive (the first one being the visited node), and the output type is what the visit rule returns upon visit. For your work you are free to use it as you see fit in your project. In the case of this tutorial, we will use strings for both input and output types. The input is the initial space, for indentation purposes, and the output will be the actual code to be generated for that node. The following code is the initial content of our visitor class, which we called *JavaCalcGenerator*:

```java
package pt.up.fe.comp2023;

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

public class JavaCalcGenerator extends AJmmVisitor<String, String> {
}
```

As expected, there are errors because this class does not yet implement the abstract methods required by the abstract class. We will solve this in the next steps.

The main difference from the *JmmVisitor* interface to the *AJmmVisitor* abstract class is that we can create rules to map a node kind to a method that should execute when such a node appears. For instance, if we want to specify a method to deal with nodes of the *Program* kind, then we can add code similar to the following:

```java
addVisit("Program", this::dealWithProgram);
...
private String dealWithProgram(JmmNode jmmNode, ...) {
    //code that deals with a "program" node
}
```

This rule ensures that, whenever *Program* node is found, the method *dealWithProgram* is executed. The definition of these rules is performed by filling a map in the `buildVisitor` method of the visitor class. If we add mapping rules for each kind of node that can result from parsing, our visitor should now look similar to the following:

```java
package pt.up.fe.comp2023;

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

public class JavaCalcGenerator extends AJmmVisitor<String, String> {
    private String className;
```

[3] https://en.wikipedia.org/wiki/Tree_traversal#Pre-order,_NLR
[4] https://en.wikipedia.org/wiki/Tree_traversal#Post-order,_LRN

```java
    public JavaCalcGenerator(String className) {
        this.className = className;
    }

    protected void buildVisitor() {
        addVisit("Program", this::dealWithProgram);
        addVisit("Assignment", this::dealWithAssignment);
        addVisit("Integer", this::dealWithLiteral);
        addVisit("Identifier", this::dealWithLiteral);
    }
}
```

We include an argument in the constructor, simply to give a name to the class we will be generating.

Now that we mapped each kind of node to a specific method, we have to implement each one of those methods, adding the corresponding logic for that node. For instance, what we want to do in *dealWithProgram* is to generate the body of the class and the main method that will execute the program. Inside that main method, we want to add all the statements following the rules we defined previously. See the example below for that implementation, where the `s` argument is the space we will use for indentation.

Between lines 7 and 10 we are manually visiting the children of the current node, as this abstract class does not visit them automatically. Here, we expect the visits to return the code of these nodes, and we concatenate that code to the current code. At the end, we return the code corresponding to the complete class.

```java
1  private String dealWithProgram(JmmNode jmmNode, String s) {
2      s = (s!=null?s:"");
3      String ret = s+"public class "+this.className+" {\n";
4      String s2 = s+"\t";
5      ret += s2+"public static void main(String[] args) {\n";
6
7      for(JmmNode child: jmmNode.getChildren()){
8          ret += visit(child,s2 + "\t");
9          ret += "\n";
10     }
11     ret += s2 + "}\n";
12     ret += s + "}\n";
13     return ret;
14 }
```

This example shows how to work with children nodes. Besides children, a node can have its own information provided as *attributes*. These attributes can be accessed via the "get" method that all *JmmNodes* have. To exemplify its use, consider the case of the *Assignment* node. We know that the assignment has two attributes, *var* and *value*, which we need to generate the code we want for assignments. In our case, we want to convert assignments into declaration of variables.

```java
private String dealWithAssignment(JmmNode jmmNode, String s) {
    return s+"int "+jmmNode.get("var")
        + " = "+jmmNode.get("value")
        +";";
}
```

After defining each one of the expected methods, we will have our class with a code similar to the code below. You can see that there are rules not yet present in this class (e.g. for the `BinaryOp`). These are the rules you have to implement by yourself. Nevertheless, the current class is already able to generate code for assignments, such as $a = 10$;. You can try it out first (if so go now to the next paragraph), then complete it with what is missing.

```java
package pt.up.fe.comp2023;
import pt.up.fe.comp.jmm.ast.AJmmVisitor;
import pt.up.fe.comp.jmm.ast.JmmNode;

public class JavaCalcGenerator extends AJmmVisitor<String, String> {
    private String className;
    public JavaCalcGenerator(String className) {
        this.className = className;
    }

    protected void buildVisitor() {
```

```
a = 2;
b = 3;
a* b;
c = 4;
a + 2 * b / c;
```

(a) Input

```
public class Calculator {
    public static void main(String[] args) {
        int a = 2;
        int b = 3;
        System.out.println(a * b);
        int c = 4;
        System.out.println(a + 2 * b / c);
    }
}
```

(b) output

Figure 6: Example of a generated class.

```
        addVisit("Program", this::dealWithProgram);
        addVisit("Assignment", this::dealWithAssignment);
        addVisit("Integer", this::dealWithLiteral);
        addVisit("Identifier", this::dealWithLiteral);
        //add here other rules!

    }

    private String dealWithProgram(JmmNode jmmNode, String s) {
            s = (s!=null?s:"");
            String ret = s+"public class "+this.className+" {\n";
            String s2 = s+"\t";
            ret += s2+"public static void main(String[] args) {\n";

            for(JmmNode child: jmmNode.getChildren()){
                ret += visit(child,s2 + "\t");
                ret += "\n";
            }
            ret += s2 + "}\n";
            ret += s + "}\n";
            return ret;
    }
    private String dealWithAssignment(JmmNode jmmNode, String s) {
        return s+"int "+jmmNode.get("var")
                + " = "+jmmNode.get("value")
                +";";
    }
    private String dealWithLiteral(JmmNode jmmNode, String s) {
            return s+jmmNode.get("value");
    }
}
```

Now that we developed the visitor, we have to use it in our compiler. For that we have to call it in the *Launcher* class, after invoking the parser. The parse operation returns a *JmmParserResult* which, if no errors occurred, will contain the root JmmNode of the resulting AST. We can use our visitor to visit that root node and obtain the generated code which should be able to execute the input operations.

The following code is how we can invoke our visitor with a given input root node. We start by creating a *JavaCalcGenerator* instance, saying that our to be generated class is named ”Calculator”. Then, we simply have to invoke the *visit* method and pass as argument the root node, which can be accessed via the *getRootNode* method of the *JmmParserResult* class. The visit process returns the resulting code.

```
JmmParserResult parserResult = parser.parse(code, config);
TestUtils.noErrors(parserResult.getReports());

JavaCalcGenerator gen = new JavaCalcGenerator("Calculator");
String generatedCode = gen.visit(parserResult.getRootNode(),"");
System.out.println(generatedCode);
```

We now have a simple, but complete, compiler, able to parse an input text and generate executable code. In Figure 6 we have an example of the output generated by running the compiler with the input defined in 6a. After you have completed the code, it should generate the class defined in 6b.

8 Exercise Your Knowledge

We provide in this section a set of exercises for you to apply what you have learn during this tutorial. These are optional exercises but we advise you to try them, as they help you gain more experience and might provide you with a few hints for the actual project to be developed.

1. Change the `statement` rule so that the production for assignments (`<ID>=<INTEGER>`) accepts any expression instead of a single integer. The rule should allow something like: $a = 2 * 3 + 4;$.

2. Adapt the `expression` rule to include parenthesis in operations. The rule should allow one to prioritize operations, such as $(1 + 2) * 3;$, meaning that the sum must be produced before the multiplication.

3. Adapt the visitor of the previous section to include the parenthesis rule. Was it necessary to adapt anything? What and why?

4. Adapt the `expression` rule to include the logical and (&&) operator and the less-than (<) operator. Do not forget to check Java operators precedence. Do you need to change the `JavaCalcVisitor` class?

5. Build a new visitor that reports variable usages, both for write and reads. The output of the visitor should be a message about: number of reads a specific variable had and the number of writes each variable had. For instance, for the following input:

   ```
   a = 2;
   b = a;
   b * a;
   ```

   The visitor would output something like:

   ```
   "a": 1 write and 2 reads
   "b": 1 write and 1 read
   ```

   Some hints for this exercise:

   (a) use one/two map(s) to store this information: key is the name of the variable, value is where you increment usages.

   (b) the output can be done to the console.

   (c) if you don't need the generic types of `JmmVisitor`, you can set them with the `Void` type (note the uppercase in "V"). When putting the argument as `Void` you can simply ignore the second argument of the visit method (e.g. name it as dummy). When putting the return as `Void`, just `return null;` at the end of the method.