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 1.0, 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, 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.

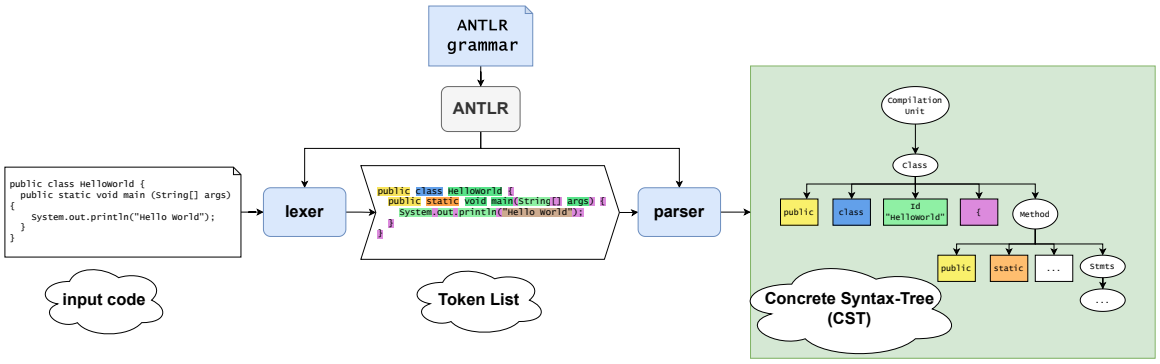


Figure 1: Flow of a parser specified in ANTLR.

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. 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.

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

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).

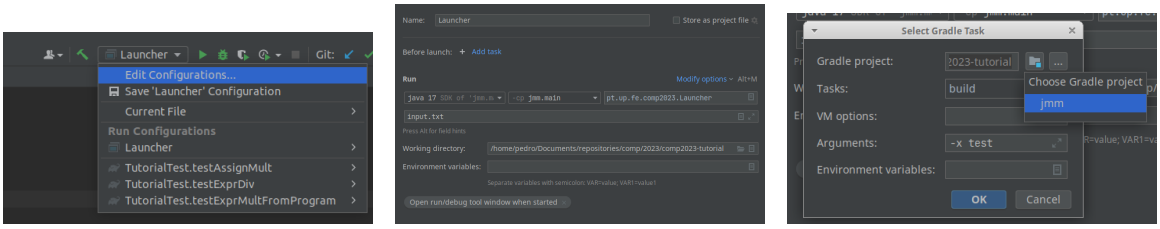
We specify this grammar in a file with the *g4* extension and in addition specify the name of the grammar and the package where the new classes will be generated. The following code contains the previous grammatical rules and it is sufficient to generate all the code for the parser and tree nodes. If 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
    ;
```

To test the generated parser proceed as follows:

1. Go to the root directory of the provided code;

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



(a) How to edit the configuration (b) Main class definition (c) Gradle build dependency

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

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 >
```

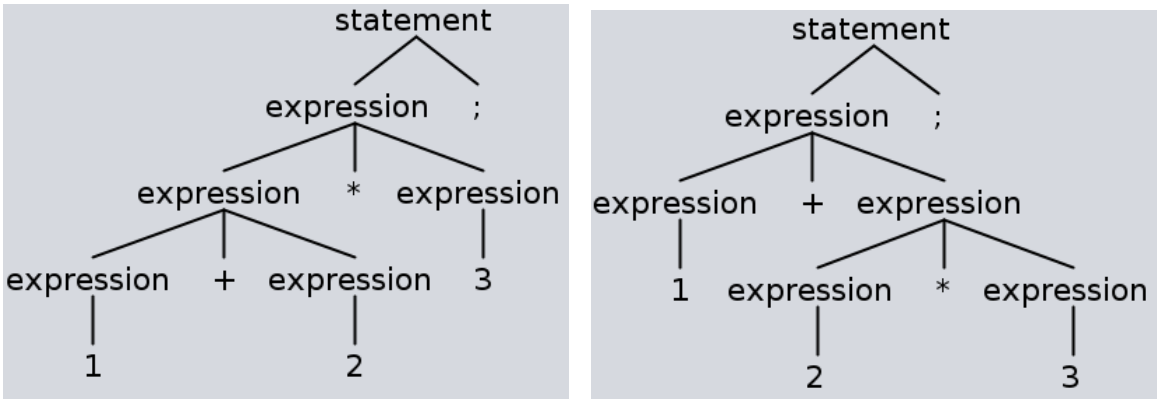
The script takes a single argument, which is a file with the input code. In the above example, the input file had as its content the expression `1+2*3`. This example program prints the parse tree to the terminal and, additionally, creates a GUI preview of the tree. You can change the input in the file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

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 no longer need to use the `installDist` Gradle task, nor the *jmm* running script. Instead you can setup a running configuration such as the one presented in Figure 2b. 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)
```



(a) incorrect tree

(b) correct tree

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

This is expected since, as the errors 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 2b. Then, you just need to add "input.txt" to the "Program Arguments" field and press "Ok". The execution should work fine now!

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 2c. 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.

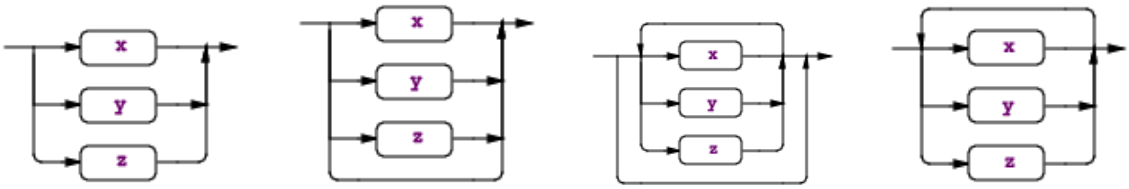
3 Precedences

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, 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.



(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.

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
    ;
```

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
    ;
```

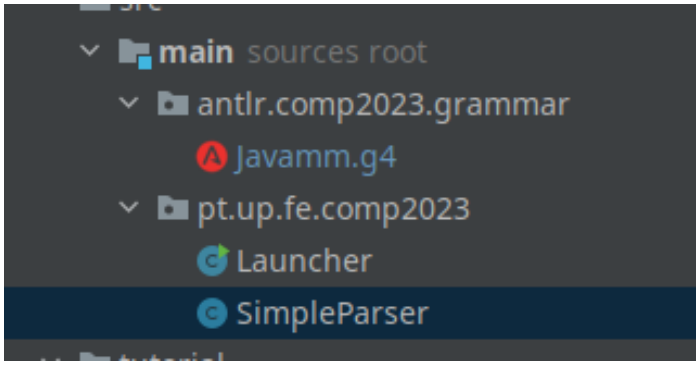


Figure 5: Project directory after copying and replacing the files in *src* with the files in *tutorial/src/jmm*.

5 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. 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:

```
1  public interface JmmNode {
2
3      String getKind();
4      List<String> getAttributes();
5      void put(String attribute, String value);
6      String get(String attribute);
7      List<JmmNode> getChildren();
8      void add(JmmNode child, int index);
9      /* ... */
10 }
```

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.

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)
```