Compiler labs: ANTLR

23/11/2020

Hamid EL MAAZOUZ, Dalila CHIADMI

1 Introduction

ANTLR (ANother Tool for Language Recognition) is a parser generator. It's a tool that helps you create lexers and parsers, usually for textual languages, but also for data formats, diagrams, or any structure in the form of text. A parser takes a piece of input text and transforms it in an organized structure, such as an AST (Abstract Syntax Tree). You can think of the AST as a story describing the content of the input text [1].

ANTLR is made up of two main modules. The first module is the *tool* which is used to generate the lexer and parser from a given ANTLR grammar specification. The second module is the *runtime* which is necessary to run the generated lexer and parser on actual input texts.

ANTLR supports generation of lexers and parsers in many target languages such as Java, JavaScript, C++, and C#. Every target is bundled as a separate jar, and because Java is the default target, it is also bundled along with the *tool* module.

The goal of this lab is to get you informed on how languages can be implemented using ANTLR. To do this you will first setup the ANTLR *tool* and use it on the provided example grammar. This will get you warmed up enough to tackle the main challenge of the lab which is implementing the *calculang* language. In this challenge, you will especially learn how ANTLR can be integrated in Java Gradle projects.

2 Generating lexer and parser

The ANTLR *tool* is bundled along with the Java target as a single jar. The class Tool in this jar is the main entry class to use for generating lexers and parsers. What arguments does the class Tool take ? Explain what every argument means. How do you invoke it on a given grammar ?

The *nice* language is a the language of nice people. It's a set of English statements that are usually used in greetings and social introductions. The following is an example grammar for such a language.

```
// parser
grammar Nice;
program: statements* EOF;
statement: greeting | introduction;
greeting: 'hello ' ID;
introduction: 'My name is ' ID;

// lexer
WS : [ \t\r\n\f]+ -> skip;
LETTER : [A-Z] | [a-z];
ID : LETTER+;
```

Generate a lexer and parser from the given Nice grammar using the ANTLR Tool class. What are the generated files ? What is the purpose of each file ?

The class TestRig, commonly called *grun* tool, can be used to test grammar production rules against input texts. It can also optionally output a graphical representation of the input text. What arguments does the class TestRig take ? Explain what every argument means. Invoke it on the different production rules of the above grammar and try out the gui option.

3 ANTLR integration in Gradle projects

Gradle is a build automation system that extends over the concepts and traditions of Ant and Maven. Gradle is written in Java and can be used to manage the build of many types of projects. Gradle is flexible, well supported, and adopted by a large community of users.

When ANTLR projects grow, working on the command line interface might not be practical. For this reason, Gradle provides a plugin called *antlr* to save us from invoking the Tool class.

The *calculang* is a simple language for writing calculus expressions. It features the usual arithmetic operations, i.e, addition, subtraction, multiplication, and division.

1. Install the Gradle build tool. Make sure to install Java first
2. Set up a new Gradle project and call it *calculang*
3. Under the project *calculang*, create a new Java module and call it *language*. Make sure to select support for Groovy and refresh your Gradle project.
4. Within the module *language*, declare ANTLR as a dependency and apply the Gradle *antlr* plugin
5. Under directory "language", create the directory tree "src/antlr/ma/emi/cs/calculang"
6. Write a grammar for the *calulang* language. Your grammar needs to support expressions such as:

   ```
   a = 2019
   b = a + 1
   c = a + b * (a - 1)
   a + b + c
   ```

   Make sure to place your grammar file under the innermost directory "calculang"
7. Configure parser generation by configuring the 'generateGrammarSource' Gradle task:

   ```
   generateGrammarSource {
       arguments += ['-package', 'ma.emi.cs.calculang.parser', "-no-listener"]
   }
   ```

8. Generate the lexer and parser for your grammar using by calling the 'generateGrammarSource' Gradle task

Note that it's challenging to come up with a whole grammar first and then write an implementation for it next. For this reason, and especially in TDD (Test-Driven Developement) settings, it makes more sense to iterate between writing the grammar for the language, implementing it, and running tests. Therefore you will need seamless parser generation process that doesn't interfere with the more important language implementation.

4 Conclusion

ANTLR [...] is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It's widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees.

References

[1] Terence Parr. *The definitive ANTLR 4 reference.* The Pragmatic Programmers, Frisco, TX, 2014.