Exercise – ANTLRv4

Patryk Kiepas

March 25, 2017

Our task is to learn ANTLR – a parser generator. This tool generates parser and lexer for any language described using a *context-free grammar*. With this parser we can perform further analysis of the texts written in the language. Help: https://github.com/antlr/antlr4/blob/master/doc/index.md

1 Installation

With IntelliJ IDEA

1. Install *IntelliJ IDEA Community Edition* – https://www.jetbrains.com/idea/
2. Open *IntelliJ IDEA Community Edition*
3. Click *File* → *Settings...* (Ctrl + Alt + S)
4. In the *Settings* window select *Plugins* and click *Install JetBrains plugin...*
5. Find and install plugin *ANTLR v4 grammar plugin*

Without IntelliJ IDEA

1. Download ANTLR v4.6 java binary – http://www.antlr.org/download/antlr-4.6-complete.jar
2. Put the JAR in any directory

2 First project

We will start with grammar *Hello*. Create *Hello.g4* file and copy the text below to the file:

```
// Define a grammar called Hello
grammar Hello;
r  : 'hello' ID ;         // match keyword hello followed by an identifier
ID : [a-z]+ ;             // match lower-case identifiers
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

This grammar allows to parse any string consists of word *hello* followed by a space, and any valid identificator. This identificator is defined as token `ID` with regular expression `[a-z]+`. This means: *match any characters from* ***a*** *to* ***z*** *one or more times.*

Note **The grammar's name and the name of *g4* file must be the same. The grammar must contain at least one parser rule.**

With IntelliJ IDEA

1. Click *File* → *New* → *Project...*
2. Then create an *Empty Project*
3. Include *Hello.g4* grammar in the project
4. Open tab with the plugin *ANTLR v4 grammar plugin*
5. In the tab write down the test code (e.g. `hello aaaaaaaa`)
6. Select initial rule *r* from *Structure* sidebar (on the left)
7. Test the grammar (look over AST)

Without IntelliJ IDEA

1. Open terminal and go to the directory with the JAR
2. Put *Hello.g4* in that directory
3. Type `java -cp "antlr-4.6-complete.jar;." org.antlr.v4.Tool Hello.g4` (this generate parser/lexer)
4. Compile parser/lexer files with `javac Hello*.java`
5. Prepare file with code to test `TEST_FILE.txt` (e.g. `hello aaaaaaaa`)
6. Test the grammar `java -cp "antlr-4.6-complete.jar;." org.antlr.v4.gui.TestRig Hello r -gui TEST_FILE.txt` – where *Hello* is the name of the grammar, and *r* is the rule to test (look over AST)

3 Arithmetic

Let's say our language is an arithmetic expression. This language consists of numbers mixed with arithmetic operators (e.g. `+`, `-`, `/`, `*`) and a pair of paranthesis (`()`). Example expressions from this language are: `3*2+(103-3)`, `3*3*3*3` or `(((3-3)))`. We start by fiding all the tokens – a basic blocks of our language. Then we prepare parser rules that describe how these tokens are connected.

Tokens

Token name starts with capital letter. By convention all token name is written in CAPS LOCK. Token is defined using *regular grammar* (well known *regular expression*). In our language there are numbers, let's say integers: `NUMBER : [0-9]+`. This means: *any sequence of one or more digits.*

Apart from named tokens we can have in-lined tokens that are used directly in the parser rules (as we see later). So instead of defining tokens for each operator (e.g. `MUL : '*'`) we will just use `'*'`. Apostrophes means that whatever is in between of them, is a part of the described language.

Parser rules

Having initial token we can move to parser rules. We have one rule which is expression `expr`. Expression can be a *number*, or an *addition of numbers*, *multiplicaiton of numbers*, ... and so on. Also an expression can be *nested in parentheses*. All of this can be written down using one parser rule with many alternatives (subrules):

```
expr : '(' expr ')'
     | expr ('*'|'/') expr
     | expr ('+'|'-') expr
     | NUMBER
     ;
```

When parsing an expression, the rule `expr` is match from top to the bottom. So the order in which rule's alternatives are written matters. If a subrule **A** is before subrule **B** then **A** will be match if it is correct, even though subrule **B** might be correct. This behaviour is also correct for tokens.

Note **Usually we define subrules/tokens starting from the more specific one to the least specific.**

Using this subrule hierarchy we can mimic operators precedence. For our rule `expr` multiplication and division have the same priority (`('*'|'/')`) but also a higher priority than addition and substraction (`('+'|'-')`)

Full example

Now we can test our grammar. We added an additional token `WS` (whitespaces) which matches all white characters (newline, tabulator, space). This token has an annotaion `-> skip` which means to *skip all of these whitespaces and don't use them when matching parser rules*. This makes our rules easy, because otherwise it would be necessary to put the tokens for each of these whitespaces whenever they could appear.

```
grammar ArithmeticExpression;

NUMBER : [0-9]+;
WS : [ \r\n\t] -> skip;

expr : '(' expr ')'
     | expr ('*'|'/') expr
     | expr ('+'|'-') expr
     | NUMBER
     ;
```

We can test this grammar with an example given before `3*2+(103-3)`. The output AST (abstract-syntax tree) for this expression is as follow:

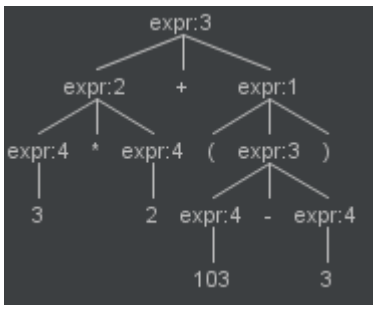

Figure 1: An AST for $3 * 2 + (103 - 3)$

4 C++ function

Now we try to parse a simple C++ function. Let's look at our language example:

```
// function example
#include <iostream>
using namespace std;

int addition (double a, float b)
{
    int r;
    r=a+b;
    return r;
}
```

Our example consists of a few language constructs that are good candidates for parser rules:

- Include directive – `#include <iostream>`
- Namespace directive – `using namespace std`
- Function definition – `int addition(double ...`
- Argument definition – `double a`
- Variable definition – `int r`
- Assignment statement – `r = a+b`
- Expression – `a+b`
- Return statement – `return r`

Here are a few proposal for our tokens

- Identificator – `iostream`, `std`, `addition`, `a`, `r` etc.
- Type – `float`, `int`, `double`
- Keywords – `using`, `namespace`, `return`
- Comment – `// function example`
- Operators – `+`
- Special characters – `<`, `>`, `;`

Tokens

We start with tokens. We define `TYPE` with three subrules for each data type name. `ID` is a classic sequence of lower and upper latin characters. `COMMENT` token matches comments and skip them. We know that comments start with `//` characters. After this characters everything belong to the comment. Term `.*?` means *match every character but do it in a non-greedy matter* (we don't want to match the rest of the program with this token). Comment ends with a new line.

```
TYPE : 'int' | 'double' | 'float';
ID : [a-zA-Z]+;
COMMENT : '//' .*? '\n' -> skip;
WS : [ \r\n\t] -> skip;
```

Parser rules

We define `program` rule as our main rule. Each program consists of an *include directive*, a *namespace directive* and a *function definitions*. The *include directive* is a `#include` token with name of the library... and so on.

```
program : include* namespace* function_def+;

include : '#include' '<' ID '>';
namespace : 'using' 'namespace' ID ';';

arg : TYPE ID;
function_def : TYPE ID '(' arg (',' arg)* ')' '{' statements '}';

statements : (stmt ';')+;
stmt : TYPE ID
     | ID '=' expr
     | 'return' ID
     ;

expr : expr '+' expr
     | ID
     ;
```

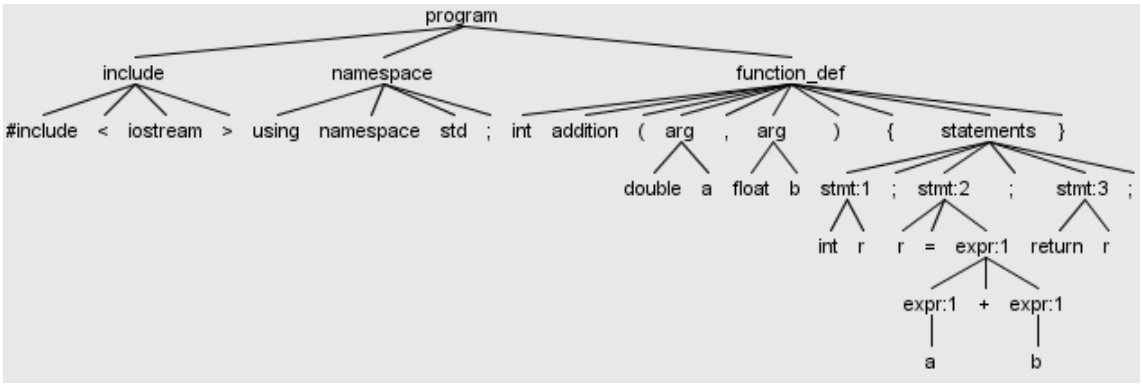

Figure 2: An AST for our C++ program

5 Project

Go to course webpage and open PDF with a project description.