Bart's blog

A blog about software development.

THURSDAY, MARCH 03, 2011

2. Introduction to ANTLR

*This is part 2/9 of **Creating your own programming language with ANTLR**.*

Index

Introduction

ANTLR is a so called ***parser generator***, or, sometimes called a *compiler-compiler*. In a nutshell: given a grammar of a language, ANTLR can generate a lexer and parser for said language. I realize that a one line definition isn't worth much, so we'll create a small **CSV** parser using ANTLR before tackling our more complex *TL*. To keep things simple, we won't be creating a CSV parser that adheres strictly to its **RFC**, but it's not too hard to rewrite it so that it does. I'll leave that as an exercise for the reader (sorry, I just *had* to cram that remark in this tutorial somewhere: I won't do it again. Promised!).

The first step is deciding what the smallest building blocks of the language are. In a CSV **file**, a single **value** is one of:

- **a simple value**: zero or more characters other than a **comma**, **line break** or a **double quote**;
- **a quoted value**: a single double quote followed by zero or more **characters other than a double quote** and ending with a single double quote. If a double quote is to be part of the value, it can be escaped by placing a double quote before it (so that's a *double* double quote: `""`).

Each **row** in a CSV file is one or more values as described above, delimited by a comma.

As you might have noticed, I underlined some of the words in my specification of a CSV file. These are the building blocks of the language. As you can see, some blocks are made up from other blocks, like:

- file (one ore more rows)
- row (one or more values separated by comma's)
- simple value (see the list above)
- quoted value (see the list above)

These are also known as *non-terminals* in parser theory. While the other blocks can't be broken down into smaller blocks:

- comma
- line break
- double quote
- character other than a double quote

which are, not surprisingly, called *terminals*.

However, in this example we'll be parsing **value**s as terminals because it's easier to do so (as you'll see later on).

Lexing

The first step is to create a lexer (or scanner) for our CSV language. A lexer is a thing that, given a stream of single characters, produces a stream of tokens. For example, the string `AA,bBb,"c""C"` would be presented to the lexer as follows:

```
A A , b B b , " c " " C "
```

i.e., just an array of characters. It should (ideally) produce a stream of tokens that looks like:

- `AA`
- `,`
- `bBb`
- `,`
- `"c""C"`

ANTLR lexer

First create a file called **CSVLexer.g**, which is the ANTLR grammar file. Every ANTLR grammar file must start with a grammar-type and -name. The name should correspond to the name of the file. So, inside **CSVLexer.g**, this is the first you type:

view plain print ?

```
01. // CSVLexer.g
02. lexer grammar CSVLexer;
03.
04. /*
05. the rest of the file here
06. */
```

As you can see, the same single- and multi-line comments as in Java are allowed in ANTLR grammar files.

Now we're going to define our three token types (value, comma and line-break) in our grammar file (which I'll call rules from now on). In a lexer grammar, you always define your rules starting with a capital. This is no convention, but a must!

Okay, let start with the easiest: the comma. The lexer rule for a comma is this:

view plain print ?

```
01. Comma
02.   :  ','
03.   ;
```

Note that the indentation is my personal preference, but the following is equivalent:

view plain print ?

```
01. Comma: ',';
```

The first just looks better :).

So, to emphasize, an ANTLR rule must always look like this:

```
RULE_NAME : RULE_CONTENTS ;
```

Now the rule for line breaks. A line break is one of the following:

- **A)** `"\r\n"` on Windows
- **B)** `"\n"` on Macs/*nix
- **C)** `"\r"` on old Macs

This results in the following rule:

view plain print ?

```
01. LineBreak
02.   :  '\r' '\n'  // A
03.   |  '\n'       // B
04.   |  '\r'       // C
05.   ;
```

or, when using the operator `?` to make something optional, you could use the slightly more compact rule:

view plain print ?

```
01. LineBreak
02.   :  '\r'? '\n'  // matches '\r\n' or '\n'
03.   |  '\r'
```

```
04.   ;
```

And the lexer rules for a CSV value could look like this:

```
view plain print ?
01. SimpleValue
02.   :  ~(',' | '\r' | '\n' | '"')+
03.   ;
04.
05. QuotedValue
06.   :  '"' ('""' | ~'"')* '"'
07.   ;
```

As you might have made up from the example above, some characters are special inside ANTLR grammars. Here's a table of the ones you've seen so far:

character	meaning	example	matches
|	logical *OR*	`'a' | 'b'`	either `'a'` or `'b'`
?	optional	`'a' 'b'?`	either `'ab'` or `'a'`
*	none or more	`'a'*`	nothing, `'a'`, `'aa'`, `'aaa'`, ...
+	once or more	`'a'+`	`'a'`, `'aa'`, `'aaa'`, ...
~	negation	`~('a' | 'b')`	any character (in the range \u0000..\uFFFF) except `'a'` and `'b'`
(...)	grouping	`('a' 'b')+`	`'ab'`, `'abab'`, `'ababab'`, ...

So, the part: `('""' | ~'"')*` from the lexer rule `QuotedValue` matches 2 quotes or any character other than a quote, which is then repeated zero or more times.

Our lexer grammar now looks like this:

```
view plain print ?
01. lexer grammar CSVLexer;
02.
03. Comma
04.   :  ','
05.   ;
06.
07. LineBreak
08.   :  '\r'? '\n'
09.   |  '\r'
10.   ;
11.
12. SimpleValue
13.   :  ~(',' | '\r' | '\n' | '"')+
14.   ;
15.
16. QuotedValue
17.   :  '"' ('""' | ~'"')* '"'
18.   ;
```

and let's say our CSV file look like this:

```
value1,value2,"value3.1,"",value3.2"
"line
break",Bbb,end
```

i.e. 2 rows containing each 3 values (note that `"line` and `break"` are one value: quoted values can contain line breaks!).

Now let's generate a lexer from the grammar we've created. To do this, download **ANTLR v3.2** and put it in the same directory as the **CSVLexer.g** file. Generating a lexer from our **CSVLexer.g** file can be done by issuing the following command on the command line of your OS:

```
java -cp antlr-3.2.jar org.antlr.Tool CSVLexer.g
```

After doing so, two files have been generated: **CSVLexer.java** and **CSVLexer.tokens**. The first one is your actual lexer class that is able to tokenize the source into tokens. You can test this lexer with the following class:

```
view plain print ?
01. import org.antlr.runtime.*;
02.
03. public class Main {
04.   public static void main(String[] args) throws Exception {
05.     // the input source
06.     String source =
07.         "value1,value2,\"value3.1,\"\",value3.2\"" + "\n" +
08.         "\"line\nbreak\",Bbb,end";
09.
10.     // create an instance of the lexer
11.     CSVLexer lexer = new CSVLexer(new ANTLRStringStream(source));
12.
13.     // wrap a token-stream around the lexer
```

```
    CommonTokenStream tokens = new CommonTokenStream(lexer);

    // when using ANTLR v3.3 or v3.4, un-comment the next line:
    //tokens.fill();

    // traverse the tokens and print them to see if the correct tokens are created
    int n = 1;
    for(Object o : tokens.getTokens()) {
      CommonToken token = (CommonToken)o;
      System.out.println("token(" + n + ") = " + token.getText().replace("\n", "\\n"))
;
      n++;
    }
  }
}
```

which, after compiling all **.java** files and running the **Main** class:

```
javac -cp antlr-3.2.jar *.java
java -cp .:antlr-3.2.jar Main
```

(on Windows, substitute the `':'` for a `';'` in the last command)

results in the following output:

```
token(1) = value1
token(2) = ,
token(3) = value2
token(4) = ,
token(5) = "value3.1,"",value3.2"
token(6) = \n
token(7) = "line\nbreak"
token(8) = ,
token(9) = Bbb
token(10) = ,
token(11) = end
```

Parsing

After the lexer has created the tokens from the source, the tokens are then passed to the parser. The parser performs the syntactic analysis. Some source might my lexically correct, but syntactically incorrect. Take the input `a\n\nb` is lexically correct: it will be tokenized as `Value`, `LineBreak`, `LineBreak` and `Value` but empty lines are not allowed in CSV files, so syntactically the input is not correct.

ANTLR parser

To create an ANTLR parser grammar, we could create a file called **CSVParser.g**, but in case of rather small grammars, it's easier to create a so called *combined grammar* in which you can mix lexer- and parser rules. Instead of declaring either **lexer grammar ...** or **parser grammar ...** at the start, simply create a file called **CSV.g** and declare the grammar like this:

```
view plain print ?
grammar CSV;

// ... parser- and lexer rules
```

in which case a **CSVParser.java** and **CSVLexer.java** are automatically generated ("Lexer" and "Parser" are automatically appended to the grammar name).

So rename **CSVLexer.g** into **CSV.g** and copy the following into it:

```
view plain print ?
grammar CSV;

file
  :  row+ EOF
  ;

row
  :  value (Comma value)* (LineBreak | EOF)
  ;

value
  :  SimpleValue
  |  QuotedValue
  ;

```

```
Comma
  :  ','
  ;

LineBreak
  :  '\r'? '\n'
  |  '\r'
  ;

SimpleValue
  :  ~(',' | '\r' | '\n' | '"')+
  ;

QuotedValue
  :  '"' ('""' | ~'"')* '"'
  ;
```

As you can see, the four lexer rules are still the same and there are now three parser rules (which **must** begin with a lower case letter!). The **file** rule, which is the entry point of our grammar, simply states that a file is one ore more rows followed by the reserved ANTLR keyword **EOF** (meaning the *end-of-file*). And a **row** is one ore more values separated by a comma ending with either a line break, or the *end-of-file*.

Now edit **Main.java** with the following contents:

```
view plain  print  ?
import org.antlr.runtime.*;

public class Main {
  public static void main(String[] args) throws Exception {
    // the input source
    String source =
        "value1,value2,\"value3.1,\"\",value3.2\"" + "\n" +
        "\"line\nbreak\",Bbb,end";

    // create an instance of the lexer
    CSVLexer lexer = new CSVLexer(new ANTLRStringStream(source));

    // wrap a token-stream around the lexer
    CommonTokenStream tokens = new CommonTokenStream(lexer);

    // create the parser
    CSVParser parser = new CSVParser(tokens);

    // invoke the entry point of our grammar
    parser.file();
  }
}
```

and generate a lexer and parser from the grammar file:

```
java -cp antlr-3.2.jar org.antlr.Tool CSV.g
```

Then compile all Java source files:

```
javac -cp antlr-3.2.jar *.java
```

and run the main class:

```
java -cp .:antlr-3.2.jar Main
```

(on Windows, substitute the `':'` for a `';'` in the last command)

When running the main class, there shouldn't be any output printed to the console afterwards. This means that the parser didn't find any errors. Go ahead and edit the `String source` in the **Main.java** class so that it's no longer valid CSV, like:

```
view plain  print  ?
String source = "a,\"b,c";
```

for example, and compile and run it. You will see the following error being printed to the console:

```
line 1:6 mismatched character '<EOF>' expecting '"'
```

which is pretty self-explanatory: the parser encountered the *EOF* while it expected a closing quote.

Actions in an ANTLR grammar

Okay, now that we have an ANTLR grammar that produces a lexer and parser, and we're able to instantiate these classes in our Java test class, it's time to let the parser do some actual work. We'd like the `file` rule return some 2 dimensional collection of strings. A `row` would then be a likely candidate to return a 1 dimensional collection of

strings, and the `value` rule would then return, not surprisingly, a single string. You can let an ANTLR rule return an object by placing `returns [AnObject obj]` after the rule name. Let's apply that to our parser rules:

```
view plain print ?
01. file returns [List<List<String>> data]
02.   :  row+ EOF
03.   ;
04.
05. row returns [List<String> row]
06.   :  value (Comma value)* (LineBreak | EOF)
07.   ;
08.
09. value returns [String val]
10.   :  SimpleValue
11.   |  QuotedValue
12.   ;
```

All the return values: `data`, `row` and `val`, are initialized with `null`, so we'll have to do a bit of work to assign actual values to them.

Let's start with the `value` rule. You can embed custom Java code in your grammar by wrapping braces around that code. So, if we wanted to set the value of `val` to `"XYZ"`, we'd do that like this:

```
view plain print ?
01. value returns [String val]
02.   :  SimpleValue {val = "XYZ";}
03.   |  QuotedValue {val = "XYZ";}
04.   ;
```

Notice that I did that twice: in case of a `SimpleValue` and for a `QuotedValue` too. But we really want to get a hold of the contents these tokens have matched, of course. You can get their contents in your Java code by typing a `$` sign followed by their rule name and then invoking the token's `.text` attribute:

```
view plain print ?
01. value returns [String val]
02.   :  SimpleValue {val = $SimpleValue.text;}
03.   |  QuotedValue {val = $QuotedValue.text;}
04.   ;
```

And if you want to strip the start- and end-quotes from `QuotedValue`, and replace all `""` with a single double quote, do something like this:

```
view plain print ?
01. value returns [String val]
02.   :  SimpleValue {val = $SimpleValue.text;}
03.   |  QuotedValue
04.      {
05.        val = $QuotedValue.text;
06.        val = val.substring(1, val.length()-1); // remove leading- and trailing quotes
07.        val = val.replace("\"\"", "\""); // replace all `""` with `"`
08.      }
09.   ;
```

Next is the `row` rule. This one returns a list of strings, and there's a `*` in there that matches zero or more `value` rules, so we can't assign a single return value here. At the very start of the rule, we're going to instantiate the `List<String>` that will be returned inside an `@init { ... }` block. The code in such init-blocks are executed before any of rule itself is matched. After that, we'll fill the `List<String>` with the `String val` from the `value` rule:

```
view plain print ?
01. row returns [List<String> list]
02. @init {list = new ArrayList<String>();}
03.   :  a=value {list.add($a.val);} (Comma b=value {list.add($b.val);})* (LineBreak | EOF
    )
04.   ;
```

As you can see in the example above, since there are two `value` sub-rules in there, we can't reference the `String val` by doing `$value.val` because the parser does not know which `value` sub-rule we mean. So I assigned variables `a` and `b` to these sub-rules, and referenced `a` and `b` instead.

Lastly, we will also initialize a `List<List<String>>` for the `file` rule and add the return values of the one or more `row` rules to it:

```
view plain print ?
01. file returns [List<List<String>> data]
02. @init {data = new ArrayList<List<String>>();}
03.   :  (row {data.add($row.list);})+ EOF
04.   ;
```

For completeness sake, here's the final CSV grammar:

```
grammar CSV;

file returns [List<List<String>> data]
@init {data = new ArrayList<List<String>>();}
  :  (row {data.add($row.list);})+ EOF
  ;

row returns [List<String> list]
@init {list = new ArrayList<String>();}
  :  a=value {list.add($a.val);} (Comma b=value {list.add($b.val);})* (LineBreak | EOF)
  ;

value returns [String val]
  :  SimpleValue {val = $SimpleValue.text;}
  |  QuotedValue
     {
       val = $QuotedValue.text;
       val = val.substring(1, val.length()-1); // remove leading- and trailing quotes
       val = val.replace("\"\"", "\""); // replace all `""` with `"`
     }
  ;

Comma
  :  ','
  ;

LineBreak
  :  '\r'? '\n'
  |  '\r'
  ;

SimpleValue
  :  ~(',' | '\r' | '\n' | '"')+
  ;

QuotedValue
  :  '"' ('""' | ~'"')* '"'
  ;
```

which can be tested with the class:

```
import org.antlr.runtime.*;
import java.util.List;

public class Main {
  public static void main(String[] args) throws Exception {
    // the input source
    String source =
        "aaa,bbb,ccc" + "\n" +
        "\"d,\"\"d\",eee,fff";

    // create an instance of the lexer
    CSVLexer lexer = new CSVLexer(new ANTLRStringStream(source));

    // wrap a token-stream around the lexer
    CommonTokenStream tokens = new CommonTokenStream(lexer);

    // create the parser
    CSVParser parser = new CSVParser(tokens);

    // invoke the entry point of our grammar
    List<List<String>> data = parser.file();

    // display the contents of the CSV source
    for(int r = 0; r < data.size(); r++) {
      List<String> row = data.get(r);
      for(int c = 0; c < row.size(); c++) {
        System.out.println("(row=" + (r+1) + ",col=" + (c+1) + ") = " + row.get(c));
      }
    }
  }
}
```

And after running the main class, the following is printed to the console:

```
(row=1,col=1) = aaa
(row=1,col=2) = bbb
(row=1,col=3) = ccc
(row=2,col=1) = d,"d
(row=2,col=2) = eee
(row=2,col=3) = fff
```

which are the expected values.

This concludes our introduction to ANTLR. Next up is part 3 of this tutorial where we'll be looking at the lexical analysis of our *TL* language.

Continue reading **part 3. Lexical analysis of *TL***.

Posted by Bart Kiers at **2:35 PM**
Labels: **antlr**, **dsl**, **java**, **parsing**

No comments:

Post a Comment

Newer Post **Home** **Older Post**

Subscribe to: **Post Comments (Atom)**