Bart's blog

A blog about software development.

FRIDAY, MARCH 04, 2011

3. Lexical analysis of *TL*

*This is part 3/9 of **Creating your own programming language with ANTLR**.*

Index

Introduction

In this part we're going to write a lexical analyzer, or just lexer, for our *TL* language. Since *TL* is an uncomplicated language, we're going to write the lexer- and parser grammar in one combined ANTLR grammar as shown in **part 2. Introduction to ANTLR**.

Looking at **part 1. *TL* specification**, here's the list of tokens in the combined grammar file **TL.g**:

```
view plain print ?
01. grammar TL;
02.
03. parse
04.   :  (t=.
05.           {System.out.printf("text: \%-7s  type: \%s \n",
06.            $t.text, tokenNames[$t.type]);}
07.     )*
08.     EOF
09.   ;
10.
11. Println  : 'println';
12. Print    : 'print';
13. Assert   : 'assert';
14. Size     : 'size';
15. Def      : 'def';
16. If       : 'if';
17. Else     : 'else';
18. Return   : 'return';
19. For      : 'for';
20. While    : 'while';
21. To       : 'to';
22. Do       : 'do';
23. End      : 'end';
24. In       : 'in';
25. Null     : 'null';
26.
27. Or       : '||';
28. And      : '&&';
29. Equals   : '==';
30. NEquals  : '!=';
31. GTEquals : '>=';
32. LTEquals : '<=';
33. Pow      : '^';
34. Excl     : '!';
35. GT       : '>';
36. LT       : '<';
37. Add      : '+';
38. Subtract : '-';
39. Multiply : '*';
40. Divide   : '/';
41. Modulus  : '%';
42. OBrace   : '{';
```

```
43.  CBrace   : '}';
44.  OBracket : '[';
45.  CBracket : ']';
46.  OParen   : '(';
47.  CParen   : ')';
48.  SColon   : ';';
49.  Assign   : '=';
50.  Comma    : ',';
51.  QMark    : '?';
52.  Colon    : ':';
53.
54.  Bool
55.    :  'true'
56.    |  'false'
57.    ;
58.
59.  Number
60.    :  Int ('.' Digit*)?
61.    ;
62.
63.  Identifier
64.    :  ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | Digit)*
65.    ;
66.
67.  String
68.  @after {
69.    setText(getText().substring(1, getText().length()-1).replaceAll("\\\\(.)", "$1"));
70.  }
71.    :  '"'  (~('"' | '\\')  | '\\' .)* '"'
72.    |  '\'' (~('\'' | '\\') | '\\' .)* '\''
73.    ;
74.
75.  Comment
76.    :  '//' ~('\r' | '\n')* {skip();}
77.    |  '/*' .* '*/'         {skip();}
78.    ;
79.
80.  Space
81.    :  (' ' | '\t' | '\r' | '\n' | '\u000C') {skip();}
82.    ;
83.
84.  fragment Int
85.    :  '1'..'9' Digit*
86.    |  '0'
87.    ;
88.
89.  fragment Digit
90.    :  '0'..'9'
91.    ;
```

There are a couple of things that might be confusing, and/or things I haven't mentioned before. I'll address those first before testing our lexer.

DOT meta character

In the parser rule `parse`, I've used the `.` (dot):

```
view plain  print  ?
01.  parse
02.    :  .* EOF
03.    ;
```

A dot can mean two things:

- inside a lexer rule, it matches any character in the range `\u0000..\u0FFFF`;
- inside a parser rule, it matches any lexer rule.

Many people new to ANTLR think that when doing:

```
view plain  print  ?
01.  foo
02.    :  .
03.    ;
04.
05.  A : 'a';
06.
07.  B : 'b';
```

the rule `foo` will match any character, which is not the case. The rule `foo` matches either rule `A` or `B` (so only the character `'a'` or `'b'`).

The rule `C` in:

```
view plain  print  ?
01.  C : . ;
```

does match any character in the range `\u0000..\u0FFFF`.

Order of lexer rules

Notice that the first few lexer rules (`Println` to `Null`) are some of the so called reserved words. ANTLR's lexer will try to match the first lexer rule it encounters, and if it can't, it will go down the list from top to bottom. So it's important that such reserved words come before a rule like `Identifier`, which would cause no reserved word token to be created since it too matches the string `'println'` like the token `Println` does.

Custom code

Note that I used `{skip();}` in both the `Comment` and `Space` rules. Whenever the lexer matches either of those tokens, it will be discarded immediately. These tokens will therefor not be available in parser rules. This `skip()` method is inherited from the **`Lexer`** class, which our `TLLexer` automatically extends.

I also used:

```
view plain print ?
01. setText(getText().substring(1, getText().length()-1).replaceAll("\\\\(.)", "$1"));
```

in the `String` rule. This is just a short notation for:

```
view plain print ?
01. // get the text this token matched
02. String matched = getText();
03.
04. // remove the leading and trailing quote
05. matched = matched.substring(1, matched.length()-1);
06.
07. // replace all `\X` with `X`
08. matched = matched.replaceAll("\\\\(.)", "$1");
09.
10. // set the new contents of this token
11. setText(matched);
```

where `getText()` and `setText(String)` are methods of the **`CommonToken`** class available in our `String` rule.

Of course, the latter code is far more readable, but personally, I'd like to keep my ANTLR grammar as empty as possible: I don't like to scan through many lines of Java code to see where the ANTLR rules are. But, if you feel more comfortable with the latter code, by all means, use that.

Fragment rules

Lastly, there are two rules that have the keyword `fragment` in front of it. You create fragment rules when you don't want such a rule to become a token of its own. These rules are therefor only available from other lexer rules. See of it as some sort of macro: you define the fragment rule `Digit` to match a single digit and you can then use this `Digit` fragment rule from any other lexer rule instead of duplicating `'0'..'9'` in many lexer rules.

The example of such a `Digit` fragment might not be a very convincing one, but let's say we wanted to support hexadecimal character literals inside out lexer that'd look like this `\u005A` (`\u` followed by four hexadecimal digits). We could do that by doing:

```
view plain print ?
01. HexChar
02.   :  '\\' 'u' ('0'..'9' | 'a'..'f' | 'A'..'F') ('0'..'9' | 'a'..'f' | 'A'..'F')
03.              ('0'..'9' | 'a'..'f' | 'A'..'F') ('0'..'9' | 'a'..'f' | 'A'..'F')
04.   ;
```

But the following is far better:

```
view plain print ?
01. HexChar
02.   :  '\\' 'u' HexDigit HexDigit HexDigit HexDigit
03.   ;
04.
05. fragment HexDigit
06.   :  '0'..'9' | 'a'..'f' | 'A'..'F'
07.   ;
```

And if we didn't make `HexDigit` a fragment, there would be the risk that a `HexDigit` would become a token itself, which is probably not desired.

Also notice that I placed two backslashes inside `'\\'`. Inside literals, the `\` and `'` need to be escaped with a backslash if you want to match them.

Testing the lexer

Let's create a test class for our lexer, called **Main.java**, and copy the following contents in it:

```
view plain print ?
01. import org.antlr.runtime.*;
02.
03. public class Main {
04.   public static void main(String[] args) throws Exception {
05.     // create an instance of the lexer
06.     TLLexer lexer = new TLLexer(new ANTLRFileStream(args[0]));
07.
08.     // wrap a token-stream around the lexer
09.     CommonTokenStream tokens = new CommonTokenStream(lexer);
10.
11.     // create the parser
12.     TLParser parser = new TLParser(tokens);
13.
14.     // invoke the 'parse' rule
15.     parser.parse();
16.   }
17. }
```

And also create a small test script called **test.tl** containing:

```
view plain print ?
01. aaa = 1.333 + 'a\'b"';
02. b = /* comment 1 */ aaa^aaa;
03. println('b=' + b); // comment 2
```

Now generate the lexer:

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

And since it's a combined grammar, the parser will also be generated, but the parser will only have a single rule, `parse`, that matches the entire token stream and prints the text and type of each token.

Compile all Java source files:

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

And run the main class:

```
java -cp .:antlr-3.2.jar Main test.tl # *nix/Mac
java -cp .;antlr-3.2.jar Main test.tl # Windows
```

which will produce the following output:

```
text: aaa      type: Identifier
text: =        type: Assign
text: 1.333    type: Number
text: +        type: Add
text: a'b"     type: String
text: ;        type: SColon
text: b        type: Identifier
text: =        type: Assign
text: aaa      type: Identifier
text: ^        type: Pow
text: aaa      type: Identifier
text: ;        type: SColon
text: println  type: Println
text: (        type: OParen
text: b=       type: String
text: +        type: Add
text: b        type: Identifier
text: )        type: CParen
text: ;        type: SColon
```

As you can see, the white spaces and comments are no longer part of the token stream and the rest of the input is correctly tokenized.

Okay, that concludes the lexical analysis of our language. Next up is the more challenging part: the syntactic analysis, or less formally, the parsing of the tokens produced by the lexer. Continue reading: **4. Syntactic analysis of *TL***

Posted by Bart Kiers at **9:51 AM**
Labels: **antlr**, **dsl**, **java**, **parsing**

2 comments:

Mateo said...

I'm not sure why, but when I do the test harness at the bottom I get the correct result with an exception at the end. "Exception in thread 'main' java.lang.ArrayIndexOutOfBoundsException: -1 at Main.main".

Doesn't that seem odd?

6:44 PM

Bart said...

Hi Mateo, yeah, that is odd. I could not reproduce this with ANTLR 3.2, 3.3 or 3.4. However, I did find something that ANTLR versions 3.3 and 3.4 did slightly different than 3.2. So I changed the grammar and driver class slightly so that all 3.x version produce the same output.

1:02 AM

Post a Comment

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

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