Bart's blog

A blog about software development.

SATURDAY, MARCH 05, 2011

4. Syntactic analysis of *TL*

*This is part 4/9 of **Creating your own programming language with ANTLR**.*

Index

- Introduction
- Project structure
- Parser rules
- Next

Introduction

In this part, we're going to extend our combined grammar file from part 3 with parser rules. After the lexer has provided us with tokens, we're going to let the parser validate the order of these tokens, which is called the syntactic analysis of the language.

Project structure

Before we continue, we're going to setup a very basic project structure to keep things a bit orderly, and to make generating, compiling and running code easier.

The project will look like:

```
TL_ROOT
   |
   +-- build/
   |     |
   |     +-- classes/
   +-- lib/
   |    |
   |    +-- antlr-3.2.jar
   +-- src/
   |    |
   |    +-- grammar/
   |    |      |
   |    |      +-- TL.g
   |    +-- main/
   |          |
   |          +-- tl/
   |               |
   |               +-- parser/
   |               +-- Main.java
   +-- build.xml
   +-- test.tl
```

The names that end with a `/` are directories, the others, not surprisingly, are files. You can re-use the files we used in **3. Lexical analysis of TL**. There are some small modifications however:

In the combined grammar file **TL.g** we will tell ANTLR to put both the generated lexer and parser in the package `lt.parser` by adding this package declaration in their respective header sections:

ARCHIVE

- 2012 (5)
- 2011 (11)
  - May (1)
  - March (10)
    - 9. Room for improvement
    - 8. Interpreting and evaluating TL II
    - 7. Interpreting and evaluating TL I
    - 6. Creating a tree grammar
    - 5. Building an AST
    - 4. Syntactic analysis of TL
    - 3. Lexical analysis of TL
    - 2. Introduction to ANTLR
    - 1. TL specification
    - Creating your own programming language with ANTLR....

```
view plain print ?
01. grammar TL;
02.
03. @parser::header {
04.   package tl.parser;
05. }
06.
07. @lexer::header {
08.   package tl.parser;
09. }
10.
11. parse
12.   :  .* EOF
13.   ;
14.
15. // the lexer rules here
```

The class `Main.java` will be in the package `tl` and it will import `lt.parser.*`, which will be the place where our generated lexer and parser are going to be placed:

```
view plain print ?
01. package tl;
02.
03. import tl.parser.*;
04. import org.antlr.runtime.*;
05.
06. public class Main {
07.   public static void main(String[] args) throws Exception {
08.     // create an instance of the lexer
09.     TLLexer lexer = new TLLexer(new ANTLRFileStream("test.tl"));
10.
11.     // wrap a token-stream around the lexer
12.     CommonTokenStream tokens = new CommonTokenStream(lexer);
13.
14.     // create the parser
15.     TLParser parser = new TLParser(tokens);
16.
17.     // invoke the entry point of our parser
18.     parser.parse();
19.     System.out.println("Done!");
20.   }
21. }
```

We'll use **Ant** to do all the boring tasks such as generating, compiling and running (source) code. I assume you, as a Java developer (or Java enthusiast), either already have Ant installed, or are able to do so on your own.

The Ant build script we'll be using looks like this:

```
view plain print ?
01. <?xml version="1.0" encoding="UTF-8"?>
02. <project name="TinyLanguage" default="run">
03.
04.   <path id="classpath">
05.     <pathelement location="build/classes/" />
06.     <pathelement location="src/main/" />
07.     <fileset dir="lib">
08.       <include name="*.jar" />
09.     </fileset>
10.   </path>
11.
12.   <target name="clean">
13.     <delete dir="build/classes/" />
14.     <mkdir dir="build/classes/" />
15.   </target>
16.
17.   <target name="compile" depends="clean">
18.     <javac srcdir="src/main/" destdir="build/classes/" includeantruntime="false">
19.       <classpath refid="classpath" />
20.     </javac>
21.   </target>
22.
23.   <target name="generate" depends="clean">
24.     <echo>Generating the lexer and parser...</echo>
25.     <java classname="org.antlr.Tool" fork="true" failonerror="true">
26.       <arg value="-fo" />
27.       <arg value="src/main/tl/parser/" />
28.       <arg value="src/grammar/TL.g" />
29.       <classpath refid="classpath" />
30.     </java>
31.     <antcall target="compile" />
32.   </target>
33.
34.   <target name="run" depends="generate">
35.     <echo>Running the main class...</echo>
36.     <java classname="tl.Main">
37.       <classpath refid="classpath" />
38.     </java>
39.   </target>
40.
41. </project>
```

That's is. Now open a command prompt, navigate to the project root, and test if everything works by executing the `run` target:

```
ant run
```

which should produce the following output:

```
/Path/To/TinyLanguage$ ant run
Buildfile: /Path/To/TinyLanguage/build.xml

clean:
   [delete] Deleting directory /Path/To/TinyLanguage/build/classes
    [mkdir] Created dir: /Path/To/TinyLanguage/build/classes

generate:
     [echo] Generating the lexer and parser...

clean:
   [delete] Deleting directory /Path/To/TinyLanguage/build/classes
    [mkdir] Created dir: /Path/To/TinyLanguage/build/classes

compile:
    [javac] Compiling 3 source files to /Path/To/TinyLanguage/build/classes

run:
     [echo] Running the main class...
     [java] Done!

BUILD SUCCESSFUL
Total time: 1 second
```

Parser rules

Let's start defining the parser rules at the highest level. A *TL* script is essentially a block of of code. A block of code is zero or more statements or function declarations optionally ending with return statement. A statement is either an assignment, a function call, an if-statement, a for-statement or a while-statement. Assignments, function calls and return statements all must end with a semi colon.

If we translate the above into ANTLR parser rules, we could end up with this:

view plain print ?

```
01.  parse
02.    :  block EOF
03.    ;
04.
05.  block
06.    :  (statement | functionDecl)* (Return expression ';')?
07.    ;
08.
09.  statement
10.    :  assignment ';'
11.    |  functionCall ';'
12.    |  ifStatement
13.    |  forStatement
14.    |  whileStatement
15.    ;
16.
17.  functionDecl
18.    :  Def Identifier '(' idList? ')' block End
19.    ;
20.
21.  idList
22.    :  Identifier (',' Identifier)*
23.    ;
```

You can paste the rules into the **TL.g** file. They should come after the `@header { ... }` sections. You can mix lexer- and parser rules, but your grammar is more readable if you keep them separate. People usually write lexer rules at the end of the grammar file (remember that the order of the lexer rules ***is*** important!).

An `assignment` can be just a simple `a = 12*5;` (on the left an identifier, and on the right an expression), but, it could also look like: `X[0][1] = false;` where `X` points to a 2 dimensional list where the first list's second element must be set to `false`. Such a rule would look like this:

view plain print ?

```
assignment
  :  Identifier indexes? '=' expression
  ;

indexes
  :  ('[' expression ']')+
  ;
```

An `expression` has a bit more rules. We start at the operators with the lowest precedence: the ternary condition (`a = 1+2==3 ? true : false`) and the `in` operator (`'foo' in ['bar','foo']`) and we then trickle down towards the operators with the highest precedence (the unary `-` and `!`).

view plain print ?

```
expression
  :  condExpr
  ;

condExpr
  :  orExpr ( '?' expression ':' expression
            | In expression
            )?
  ;

orExpr
  :  andExpr ('||' andExpr)*
  ;

andExpr
  :  equExpr ('&&' equExpr)*
  ;

equExpr
  :  relExpr (('==' | '!=') relExpr)*
  ;

relExpr
  :  addExpr (('>=' | '<=' | '>' | '<') addExpr)*
  ;

addExpr
  :  mulExpr (('+' | '-') mulExpr)*
  ;

mulExpr
  :  powExpr (('*' | '/' | '%') powExpr)*
  ;

powExpr
  :  unaryExpr ('^' unaryExpr)*
  ;

unaryExpr
  :  '-' atom
  |  '!' atom
  |  atom
  ;

atom
  :  Null
  |  Number
  |  Bool
  |  lookup
  ;
```

The `lookup` rule can be various other rules with optional `indexes` after it:

view plain print ?

```
lookup
  :  functionCall indexes?
  |  '(' expression ')' indexes?
  |  list indexes?
  |  Identifier indexes?
  |  String indexes?
  ;
```

The `list` rule shouldn't pose too many problems:

view plain print ?

```
list
  :  '[' exprList? ']'
  ;

```

```
05. exprList
06.   :  expression (',' expression)*
07.   ;
```

A function call would look like:

```
view plain  print  ?
01. functionCall
02.   :  Identifier '(' exprList? ')'
03.   |  Println '(' expression? ')'
04.   |  Print '(' expression ')'
05.   |  Assert '(' expression ')'
06.   |  Size '(' expression ')'
07.   ;
```

As you can see, all of the function calls `print(...)`, `assert(...)` and `size(...)` take exactly one parameter (or `expression`), `println(...)` takes an optional expression as a parameter, and a call to a custom defined function takes zero or more expressions as parameter.

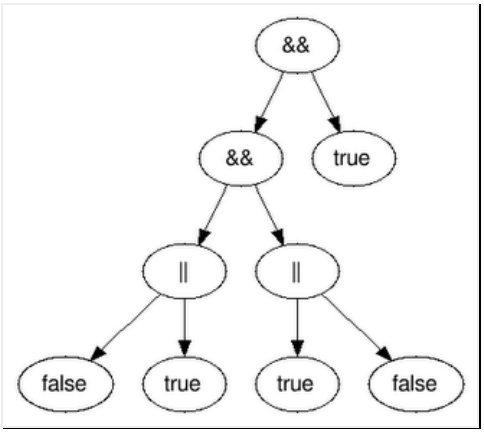
Lastly, there are the `if-`, `for-`, and `while` statements. I've divided the `if` statement into a couple of separate sub-rules otherwise the entire rule would have been too big for my taste:

```
view plain  print  ?
01. ifStatement
02.   :  ifStat elseIfStat* elseStat? End
03.   ;
04.
05. ifStat
06.   :  If expression Do block
07.   ;
08.
09. elseIfStat
10.   :  Else If expression Do block
11.   ;
12.
13. elseStat
14.   :  Else Do block
15.   ;
16.
17. forStatement
18.   :  For Identifier '=' expression To expression Do block End
19.   ;
20.
21. whileStatement
22.   :  While expression Do block End
23.   ;
```

That's the complete parser grammar of *TL*! If you execute your the Ant target `run` so that the file **test.tl** gets parsed, there shouldn't appear any error messages on your screen, which means that the contents of the test script is first properly tokenized, and after that, the parser has correctly parsed these tokens (the file is syntactically valid).

Next

So that's all the parser does: it checks if the token stream is syntactically valid. But our tokens are still just a flat, 1 dimensional list. It also contains tokens that are only needed to check the validity of the source but are making it hard to evaluate the expression(s). Let's say out input source is `true && (false || true && (true || false));`, our parser now just validates it as syntactic valid input, but the token stream is just a 1 dimensional list of tokens while it would be easier to evaluate such input if the token stream was structured like this:



I.e. the parenthesis and semi colon are gone, and every operator node has exactly two children that can be evaluated more easily.

The structuring of our token stream into an AST (abstract syntax tree) will be handled in **part 5. Building an AST**.

P.S. You can download the project containing all source code, build script, ANTLR jar etc. ***here****.*

Posted by Bart Kiers at **7:54 AM**
Labels: **antlr**, **dsl**, **java**, **parsing**

8 comments:

tarasboyko@gmail.com said...

Hi Bart

Many thanks for your great work in explaining ANTLR by example. I am new to ANTLR and I came across your work by searching stackoverflow.com for solution of my own parsing grammar problem (Excel-like expressions). I tried your sources from this page with ANTRLWorks 1.4.2 interpreter and it seems that this expression is not fully parsable (ran it starting from "expression" rule:

(1+(2+-3^2))*4^5

The result was that after the second closing parenthesis parsing was stopped so *4^5 was just ignored.

Still looking for solution... :)

/Taras

1:53 PM

Bart said...

Hi Taras, thanks for your kind words. It's hard to diagnose your problem without seeing the grammar you're working on. I suggest you post a question on Stackoverflow including the grammar and test-classes and explain what's going wrong.

Good luck!

Bart.

2:07 PM

tarasboyko@gmail.com said...

Besides my own grammar, what I actually meant was that grammar from the link above (http://big-o.nl/blog/TinyLanguage_part4.zip) had problems interpreting this expression:
(1+(2+-3^2))*4^5

So, my question is if TL supports such syntax of expressions or not. What I could understand from reading the grammar, it should. But ANTRLWorks 1.4.2 Interpreter fails in the way I described above.

/Taras

1:19 AM

Bart said...

Hi Taras, no, **(1+(2+-3^2))*4^5** by itself is not a valid statement in *TL*. It should be a return statement (**return (1+(2+-3^2))*4^5;**), or an assignment (**n = (1+(2+-3^2))*4^5;**), for example. Don't forget the semi-colon at the end!

2:10 AM

tarasboyko@gmail.com said...

tried

return (1+(2+-3^2))*4^5;

and it works just fine if I start Interpreter from "block" or "parse" rules. Interestingly,

(1+(2+-3^2))*4^5

which normally should correspond to "expression" rule if run in Interpreter starting from "expression" doesn't work properly. Weird...

4:51 AM

Bart said...

This is exactly why I didn't introduce ANTLRWorks in my tutorial: granted, it's a neat tool, but it has its own peculiarities, and a somewhat buggy interpreter, to be frank.

Yes, you are right, the expression is a valid one as you can see when you run this snippet:

```
TLLexer lexer = new TLLexer(new ANTLRStringStream("(1+(2+-3^2))*4^5"));

TLParser parser = new TLParser(new CommonTokenStream(lexer));

parser.expression();
```

ANTLRWorks *should* also accept it...

5:03 AM

tarasboyko@gmail.com said...

Bart said...
This is exactly why I didn't introduce ANTLRWorks in my tutorial: granted

very valuable comment for newcomers...

Thanks, Bart

6:14 AM

Anonymous said...

thank you

12:42 AM

Post a Comment

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

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