Bart's blog

A blog about software development.

MONDAY, MARCH 07, 2011

6. Creating a *tree grammar*

*This is part 6/9 of **Creating your own programming language with ANTLR**.*

Index

Introduction

So far we've created a lexer to chop up he character stream into tokens, called the lexical analysis. Using the parser we've checked the syntactic validness of the lexer's tokens stream, and let the parser create a proper AST structure. It's now time to prepare for the semantic analysis, which in our case is simply evaluating the input, or throwing up an exception. For example, take the statement:

```
view plain print ?
01. if 1+1 > true do
02.   // ...
03. end
```

which is lexically correct (they're all valid tokens), and is syntactically correct (it adheres to the grammar of our `relExpr` rule), but semantically, it makes no sense. We can't compare an integer value with a boolean value.

It is now time to create some sort of walker (or iterator) that traverses the AST and can then evaluate the expressions/statements it encounters. You can do that by writing an own iterator to walk the AST, but ANTLR also provides a mechanism to do this. You can let ANTLR generate such an iterator by writing a so called `tree grammar` in which you declare how the tree built by your parser can look like. In practice, you pretty much copy all parser rules from the combined grammar into a new tree grammar file, and only leave the rewrite rules in place.

If you haven't already got the necessary files, you can download the project **here**, which contains all the source code developed so far, the ANTLR jar, build script etc.

Preparation

A tree grammar is used to create an iterator that walks the tree produced by the parser. Therefor, it resembles the parser grammar quite a bit, only that it is a more compact version of it: it does not contain the tokens that have been omitted from the tree, and also does not contain any lexer rules.

You begin a tree grammar file with the keywords `tree grammar` followed by its name, which will be `TLTreeWalker`. After that, you declare what tokens are to be used and the type of tree-tokens that it can expect. These are both defined in the `options { ... }` section of the tree grammar. And like the combined grammar, we will also need to tell ANTLR to put the generated walker in the `tl.parser` package.

```
view plain print ?
01. // file: TLTreeWalker.g
02.
03. tree grammar TLTreeWalker; // the generated file is TLTreeWalker.java
04.
```


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....

```
options {
  tokenVocab=TL; // this means to use the TL.tokens file
  ASTLabelType=CommonTree;
}

@header {
  package tl.parser;
}

// rules here ...
```

Our Ant build file, **build.xml** will need to be extended so that it also generates the tree walker. Change the `generate` target into the following:

```
view plain print ?
<target name="generate" depends="clean" description="Generates the lexer and parser.">

  <echo>Generating the lexer and parser...</echo>
  <java classname="org.antlr.Tool" fork="true" failonerror="true">
    <arg value="-fo" />
    <arg value="src/main/tl/parser/" />
    <arg value="src/grammar/TL.g" />
    <classpath refid="classpath" />
  </java>
  <echo>Generating the tree walker...</echo>
  <java classname="org.antlr.Tool" fork="true" failonerror="true">
    <arg value="-fo" />
    <arg value="src/main/tl/parser/" />
    <arg value="src/grammar/TLTreeWalker.g" />
    <classpath refid="classpath" />
  </java>
  <antcall target="compile" />
</target>
```

And of course, our main class will need to be adjusted so that the tree walker class get instantiated and used:

```
view plain print ?
package tl;

import tl.parser.*;
import org.antlr.runtime.*;
import org.antlr.runtime.tree.*;

public class Main {
  public static void main(String[] args) throws Exception {
    // create an instance of the lexer
    TLLexer lexer = new TLLexer(new ANTLRFileStream("test.tl"));

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

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

    // walk the tree
    CommonTree tree = (CommonTree)parser.parse().getTree();
    CommonTreeNodeStream nodes = new CommonTreeNodeStream(tree);
    TLTreeWalker walker = new TLTreeWalker(nodes);
    walker.walk();
  }
}
```

Tree grammar

First create a file called **TLTreeWalker.g** and save it in **src/grammar/**. The tree grammar would look like this:

```
view plain print ?
tree grammar TLTreeWalker;

options {
  tokenVocab=TL;
  ASTLabelType=CommonTree;
}

@header {
  package tl.parser;
}

walk
  :  block
  ;

block
  :  ^(BLOCK ^(STATEMENTS statement*) ^(RETURN expression?))
  ;

```

```
statement
  :  assignment
  |  functionCall
  |  ifStatement
  |  forStatement
  |  whileStatement
  ;

assignment
  :  ^(ASSIGNMENT Identifier indexes? expression)
  ;

functionCall
  :  ^(FUNC CALL Identifier exprList?)
  |  ^(FUNC CALL Println expression?)
  |  ^(FUNC CALL Print expression)
  |  ^(FUNC CALL Assert expression)
  |  ^(FUNC_CALL Size expression)
  ;

ifStatement
  :  ^(IF ifStat elseIfStat* elseStat?)
  ;

ifStat
  :  ^(EXP expression block)
  ;

elseIfStat
  :  ^(EXP expression block)
  ;

elseStat
  :  ^(EXP block)
  ;

forStatement
  :  ^(For Identifier expression expression block)
  ;

whileStatement
  :  ^(While expression block)
  ;

idList
  :  ^(ID_LIST Identifier+)
  ;

exprList
  :  ^(EXP_LIST expression+)
  ;

expression
  :  ^(TERNARY expression expression expression)
  |  ^(In expression expression)
  |  ^('||' expression expression)
  |  ^('&&' expression expression)
  |  ^('==' expression expression)
  |  ^('!=' expression expression)
  |  ^('>=' expression expression)
  |  ^('<=' expression expression)
  |  ^('>' expression expression)
  |  ^('<' expression expression)
  |  ^('+' expression expression)
  |  ^('-' expression expression)
  |  ^('*' expression expression)
  |  ^('/' expression expression)
  |  ^('%' expression expression)
  |  ^('^' expression expression)
  |  ^(UNARY MIN expression)
  |  ^(NEGATE expression)
  |  Number
  |  Bool
  |  Null
  |  lookup
  ;

list
  :  ^(LIST exprList?)
  ;

lookup
  :  ^(LOOKUP functionCall indexes?)
  |  ^(LOOKUP list indexes?)
  |  ^(LOOKUP expression indexes?)
  |  ^(LOOKUP Identifier indexes?)
  |  ^(LOOKUP String indexes?)
  ;

indexes
  :  ^(INDEXES expression+)
  ;
```

Notice that the `doEndBlock` is not part of this tree grammar. This is because the `Do` and `End` tokens were removed during the rewrite phase in the previous part of this tutorial. The rules that used the `doEndBlock` rule, are now using the `block` instead.

Test the tree grammar with a couple of different input to see if it all works (no error message(s) means everything is okay). As usual, executing the Ant target `run` will be enough: all necessary files will be generated, compiled and the main class will be executed.

This tree grammar will then be mixed with custom code mixed in it that will evaluate the expressions and statements in the AST. But that will be done in the next part: **7. Interpreting and evaluating *TL* I**.

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

Posted by Bart Kiers at **7:59 AM**
Labels: **antlr**, **dsl**, **java**, **parsing**

2 comments:

Matt Lawrence said...

Very well organized tutorial for beginners. Could your tree/parser grammar be used for complex languages like C/C++ with typedef/includes etc.

1:25 AM

Bart said...

Thanks Matt, I appreciate it! Yes that's possible, although implementing all functionality of C++ (or even "only" C) might be a tall order, my example could fairly easy be extended so that it supports simple classes/structs. An import/include mechanism is even easier since that would pretty much consist of recursively parsing another source file. Both might very well be parts #10 and #9 respectively, in my little ANTLR tutorial.

5:42 AM

Post a Comment

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

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