Bart's blog

A blog about software development.

SUNDAY, MARCH 06, 2011

5. Building an AST

*This is part 5/9 of **Creating your own programming language with ANTLR**.*

Index

Introduction

In this part we're going to continue with the combined grammar file that already contains all lexer- and parser rules. If you haven't already got the necessary files, you can download the project **here**, which contains all the source code, ANTLR jar, build script etc.

AST construction in ANTLR

By default, ANTLR's generated parser emits tokens of type **CommonToken** as a 1 dimensional list. To tell our parser it should create **CommonTree** tokens that can have parent- and child tokens, we need to set the option `output=AST` in the `options` section of our grammar. This section must come directly *after* the grammar declaration:

view plain print ?

```
grammar TL;

options {
  output=AST;
}

@parser::header {
  package tl.parser;
}

@lexer::header {
  package tl.parser;
}

parse
  :  block EOF
  ;

// ...
```

Now edit the **test.tl** file and paste the following in it:

view plain print ?

```
a = 2 + 5;
b = a * 2;
```

Also edit the **Main.java** file to look like this:

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 ?

```java
package tl;

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

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);

    // invoke the entry point of our parser and generate a DOT image of the tree
    CommonTree tree = (CommonTree)parser.parse().getTree();
    DOTTreeGenerator gen = new DOTTreeGenerator();
    StringTemplate st = gen.toDOT(tree);
    System.out.println(st);
  }
}
```

The code above will cause a **DOT file** to be printed to the console of the generated AST. It will produce the following output:

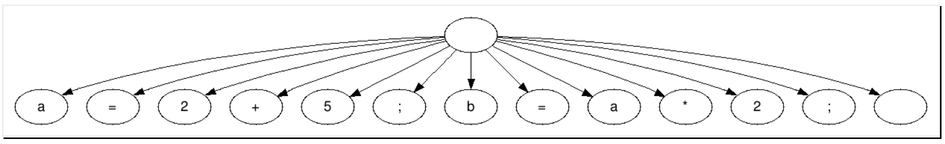
```
digraph {

  n0 [label="A"];
  n1 [label="B"];
  n2 [label="C"];
  n3 [label="2"];
  n4 [label="+"];
  n5 [label="5"];
  n6 [label=";"];
  n7 [label="b"];
  n8 [label="="];
  n9 [label="a"];
  n10 [label="*"];
  n11 [label="2"];
  n12 [label=";"];
  n13 [label=""];

  n0 -> n1 // "" -> "a"
  n0 -> n2 // "" -> "="
  n0 -> n3 // "" -> "2"
  n0 -> n4 // "" -> "+"
  n0 -> n5 // "" -> "5"
  n0 -> n6 // "" -> ";"
  n0 -> n7 // "" -> "b"
  n0 -> n8 // "" -> "="
  n0 -> n9 // "" -> "a"
  n0 -> n10 // "" -> "*"
  n0 -> n11 // "" -> "2"
  n0 -> n12 // "" -> ";"
  n0 -> n13 // "" -> ""

}
```

There are many viewers to generate images from such DOT files. I like **graphviz-dev.appspot.com** as a quick online tool. After you have pasted the output from Main in the **graphviz-dev.appspot.com** tool, you'll see an image like this appear:



This is still just a 1 dimensional list of tokens (but now of type `CommonTree`). We'll have to do a bit more: ANTLR does not magically know what rules need to be the root, and what tokens will need to be omitted from our tree.

Tree operators and rewrite rules

There are two ways to create a proper AST in ANTLR:

1. by using tree operators: `^` and `!`
2. or by using rewrite rules: `... -> ^(...)`

1. Tree operators

You can tell ANTLR to make a certain node the root by placing a `^` after it in your parser rules, and to exclude certain nodes from the tree, place a `!` after it.

Take the following parser rule for example:

view plain print ?

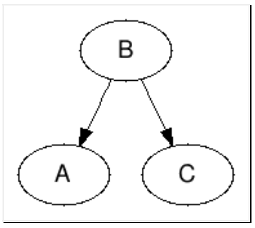
```
bar
  :  A B C D
  ;
```

If we wanted to let `B` become the root, and wanted to exclude token `D` from the tree, we'd do that like this:

view plain print ?

```
bar
  :  A B^ C D!
  ;
```

This will produce the tree:



2. Rewrite rules

The second option to create an AST, is to use rewrite rules. A rewrite rule is placed after a parser rule (but before the `';'`!). It consists of an arrow, `->`, followed by `^( ... )` where the first token (or rule) inside the parenthesis will become the root of the tree. Tokens that need to be removed from the AST will simply be omitted from the rewrite rule.

To create the same tree as the example above using a rewrite rule, we'd write:

view plain print ?

```
bar
  :  A B C D -> ^(B A C)
  ;
```

I.e.: `B` will be the root, and `A` and `C` its children (`D` is removed).

What to use?

The first option using tree operators may look easier to use. However, when your parser rules become larger, the (small) operators will become less conspicuous. Also, the rewrite rules are more flexible. If we wanted to let token `C` become the first child of parent token `B` in the previous example, tree operators wouldn't be able to do this, but a rewrite rule would simply look like this:

view plain print ?

```
bar
  :  A B C D -> ^(B C A)
  ;
```

Another advantage is that the translation of a parser grammar to a tree grammar (the next part of this tutorial) is more straight forward. But you'll see that for yourself later on.

But sometimes it *is* more convenient to use tree operators than rewrite rules. For example, with an add expression:

view plain print ?

```
addExpr
  :  multiplyExpr (('+' | '-') multiplyExpr)*
  ;
```

we would like to make the + or - the root of the expression. With rewrite rules, that would look like this:

view plain print ?

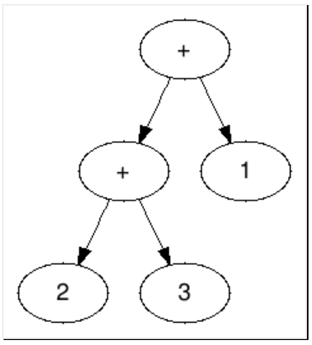
```
addExpr
  :  (multiplyExpr -> multiplyExpr)
     ( '+' e=multiplyExpr -> ^('+' $addExpr $e)
     | '-' e=multiplyExpr -> ^('-' $addExpr $e)
     )*
  ;
```

which is no trivial rewrite syntax! And I'll leave it unexplained because in our grammar, we'll go for the far less complex:

view plain print ?

```
addExpr
  :  multiplyExpr (('+' | '-')^ multiplyExpr)*
  ;
```

Yes, that is correct: the single ^ produces the exact tree as the complex looking rewrite rule! For the expression 1+2+3, both produce the tree:



Imaginary tokens

Sometimes a parser rule doesn't have an obvious root candidate. In that case we can insert a so called *imaginary token*. Take the block rule for example:

view plain print ?

```
block
  :  (statement | functionDecl)* (Return expression ';')?
  ;
```

which matches zero or more statements or function declarations, optionally ending with a return statement. To mark the first statement as the root would make little sense. What we're going to do is create a couple of *imaginary tokens* to make a *block tree* always look the same. The first we need to do is put a tokens { ... } section directly below the options { ... } section (and above the header sections!):

view plain print ?

```
grammar TL;

options {
  output=AST;
}

tokens {
  BLOCK;
  RETURN;
  STATEMENTS;
}

@parser::header {
  package tl.parser;
}

// ...
```

Imaginary tokens should start with a capital, just like lexer rules. I find it convenient to let them be distinguishable from lexer rules by using *only* capitals instead of just capitalizing the first letter like I did with lexer rules.

Our `block` rule including a rewrite rule would look like:

view plain print ?

```
block
  :  (statement | functionDecl)* (Return expression ';')?
     -> ^(BLOCK ^(STATEMENTS statement*) ^(RETURN expression?))
  ;
```

This makes sure every *block-tree* will have exactly two children: `STATEMENTS` and `RETURN`.

If we didn't use `STATEMENTS`, all zero or more statements would be direct children of the root `BLOCK` node, making it harder to evaluate since we'd always have to inspect if we're at the end, which could possibly be the return statement. Now we know that all zero or more statements are in the left tree, and the optional return statement is in the right tree.

You may have noticed that I removed the `functionDecl` from the tree. This is no accident: at the parsing stage, we're going to create a `java.uti.Map` containing all methods so that when evaluating our language, we can invoke functions before they've been defined in the source. We'll handle that at a later stage of this tutorial.

Note that you can't use a rewrite rule *and* tree operators in a single parser rule: it's always just one of the two.

A proper AST

Okay, having said all that, here's how our grammar looks like with tree operators and rewrite rules:

view plain print ?

```
grammar TL;

options {
  output=AST;
}

tokens {
  BLOCK;
  RETURN;
  STATEMENTS;
  ASSIGNMENT;
  FUNC CALL;
  EXP;
  EXP LIST;
  ID LIST;
  IF;
  TERNARY;
  UNARY MIN;
  NEGATE;
  FUNCTION;
  INDEXES;
  LIST;
  LOOKUP;
}

@parser::header {
  package tl.parser;
}

@lexer::header {
  package tl.parser;
}

parse
  :  block EOF -> block
  ;

block
  :  (statement | functionDecl)* (Return expression ';')?
     -> ^(BLOCK ^(STATEMENTS statement*) ^(RETURN expression?))
  ;

statement
  :  assignment ';'   -> assignment
  |  functionCall ';' -> functionCall
  |  ifStatement
  |  forStatement
  |  whileStatement
  ;

assignment
  :  Identifier indexes? '=' expression
     -> ^(ASSIGNMENT Identifier indexes? expression)
  ;

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

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

ifStat
  :  If expression Do block -> ^(EXP expression block)
  ;

elseIfStat
  :  Else If expression Do block -> ^(EXP expression block)
  ;

elseStat
  :  Else Do block -> ^(EXP block)
  ;

functionDecl
  :  Def Identifier '(' idList? ')' block End {/* implemented later */}
  ;

forStatement
  :  For Identifier '=' expression To expression Do block End
     -> ^(For Identifier expression expression block)
  ;

whileStatement
  :  While expression Do block End -> ^(While expression block)
  ;

idList
  :  Identifier (',' Identifier)* -> ^(ID_LIST Identifier+)
  ;

exprList
  :  expression (',' expression)* -> ^(EXP_LIST expression+)
  ;

expression
  :  condExpr
  ;

condExpr
  :  (orExpr -> orExpr)
     ( '?' a=expression ':' b=expression -> ^(TERNARY orExpr $a $b)
     | In expression                     -> ^(In orExpr expression)
     )?
  ;

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

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

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

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

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

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

powExpr
  :  unaryExpr ('^'^ unaryExpr)*
  ;

unaryExpr
  :  '-' atom -> ^(UNARY_MIN atom)
  |  '!' atom -> ^(NEGATE atom)
  |  atom
  ;

atom
  :  Number
  |  Bool
  |  Null
  |  lookup
  ;

list
  :  '[' exprList? ']' -> ^(LIST exprList?)
  ;

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

indexes
  :  ('[' expression ']')+ -> ^(INDEXES expression+)
  ;

Println  : 'println';
Print    : 'print';
Assert   : 'assert';
Size     : 'size';
Def      : 'def';
If       : 'if';
Else     : 'else';
Return   : 'return';
For      : 'for';
While    : 'while';
To       : 'to';
Do       : 'do';
End      : 'end';
In       : 'in';
Null     : 'null';

Or        : '||';
And       : '&&';
Equals    : '==';
NEquals   : '!=';
GTEquals  : '>=';
LTEquals  : '<=';
Pow       : '^';
Excl      : '!';
GT        : '>';
LT        : '<';
Add       : '+';
Subtract  : '-';
Multiply  : '*';
Divide    : '/';
Modulus   : '%';
OBrace    : '{';
CBrace    : '}';
OBracket  : '[';
CBracket  : ']';
OParen    : '(';
CParen    : ')';
SColon    : ';';
Assign    : '=';
Comma     : ',';
QMark     : '?';
Colon     : ':';

Bool
  :  'true'
  |  'false'
  ;

Number
  :  Int ('.' Digit*)?
  ;

Identifier
  :  ('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | Digit)*
  ;

String
@after {
  setText(getText().substring(1, getText().length()-1).replaceAll("\\\\(.)", "$1"));
}
  :  '"'  (~('"' | '\\')  | '\\' ('\\' | '"'))* '"'
  |  '\'' (~('\'' | '\\') | '\\' ('\\' | '\''))* '\''
  ;

Comment
  :  '//' ~('\r' | '\n')* {skip();}
  |  '/*' .* '*/'         {skip();}
  ;

Space
  :  (' ' | '\t' | '\r' | '\n' | '\u000C') {skip();}
  ;

fragment Int
  :  '1'..'9' Digit*
  |  '0'
  ;

fragment Digit
  :  '0'..'9'
  ;
```

And to test this, run the Ant target using the command line parameter `-emacs`, which casues Ant *not* to print all the `[java]` etc. stuff to the console, making it easier to copy and paste the DOT source.

Okay, the tree that was generated is a heck of a lot easier to interpret than the flat list of nodes! But before evaluating, we'll first have to create some sort of iterator that traverses our freshly constructed AST, which will be the subject of the next part **6. Creating a tree grammar**.

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 **7:58 AM**
Labels: **antlr**, **dsl**, **java**, **parsing**

1 comment:

oleg said...

Thanks for this wonderful post !!! It really makes things clear.

Keep up the good work :p

7:28 PM

Post a Comment

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

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

Powered by **Blogger**.