Introduction to ANTLR (ANother Tool for Language Recognition)

Jon Eyolfson

University of Waterloo

September 27 - October 1, 2010

Outline

- Introduction
- Usage
- Example
- Demonstration
- Conclusion

Introduction

- **ANTLR** accepts a language description in EBNF grammar and creates a recognizer for that language
- The recognizers handle three types of input: character streams (lexer), token streams (parser) and node streams
- This tool can generate recognizers in may languages, the most popular being Java and C++
- We will focus on the first two which you will need for your first lab

Input

ANTLR Grammar

```
[grammar_type] grammar NAME

options { variable = value; ... }

tokens { TOKEN = 'string'; ... }

@header { /* Header of generated Java file */ }

@lexer::header { /* Copied to NAMELexer.java */ }

@members { /* Member section of generated Java file */ }

rulename : ruledefinition ...
```

These are just **ANTLR** settings, we define our actions in the rules section

Rules

Convention

Lexer non-terminals (token names) contain only upper case letters
Parser non-terminals contain only lower case letters

ANTLR Grammar (Rules)

```
rulename [args] [returns T val]
    : firstchoice { /* Optional Java code */ }
    | secondchoice { /* Optional Java code */ }
    ;
```

There are 4 EBNF operators

- X|Y matches X or Y
- X* matches X zero or more times
- X+ matches X one or more times
- X? optionally matches X

Using ANTLR

- Download **ANTLR** from http://www.antlr.org
- Include the JAR files in your classpath which may include antlr.jar, antlr3.jar, stringtemplate.jar
- Run it using java org.antlr.Tool Grammar.g
- Since our grammar is small, we can define our lexer and parser in the same grammar file
- In this case the tool generates GrammarLexer.java, GrammarPaser.java and Grammar.tokens for us

Example

Consider a subset of your lab, the *Simple Datatype Language* which only handles the following:

- Read, print and assignments
- Atom integer values
- Map operator with + and -

How would we use **ANTLR** to help us?

Example Header

SDL.g

```
grammar SDL;

options {
  language = Java;
}

@header {package ca.uwaterloo.ece251;}

@lexer::header {package ca.uwaterloo.ece251;}

@members {Interp interp = new Interp();}
```

[Optional] You may also define keywords and symbols here to be used in the parsing rules by using tokens

Example Lexer

SDL.g

```
EXT: 'atom';
fragment LETTER: ('a'..'z' | 'A'..'Z');
VAR: LETTER (LETTER | '0'..'9' | '_')*;
LITERAL: ('0'..'9')+;
NEWLINE: ('\r'? '\n')+ {skip();};
WHITESPACE: (' ' | '\t') {skip();};
```

[Optional] We could also use $channel = HIDDEN; instead of skip();

Example Parser

SDL.g

```
prog
  : stmt* EOF
  ;

stmt
  : (read | print | assign)
  ;

read
  : 'read' VAR '.' EXT
  ;

print
  : 'print' VAR
  ;
```

```
assign
  : VAR ':=' exp
  ;

exp
  : VAR
  | LITERAL
  | 'map' transformer exp
  ;

transformer
  : '+' LITERAL
  | '-' LITERAL
  ;
```

Almost Done?

We can generate a lexer and parser for us and use them

SDL.java

```java
package ca.uwaterloo.ece251;

import org.antlr.runtime.*;

public class SDL {
  public static void main(String[] args) throws Exception {
    CharStream input = null;
    if (args.length == 1) { input = new ANTLRFileStream(args[0]); }
    else { System.err.println("You must provide an input file");
           return; }
    SDLLexer lexer = new SDLLexer(input);
    TokenStream tokenStream = new CommonTokenStream(lexer);
    SDLParser parser = new SDLParser(tokenStream);
    parser.prog();
  }
}
```

Now valid input files will parse without throwing an Exception, however our program does not interpret and process our language

Modifying the Parser

Remember we can insert our own Java code onto the end of each rule using braces?

- We also have an defined an Interp object in our Parser class, how might we use it?

Print Rule

```
print
  : 'print' VAR {interp.print($VAR.text);};
```

This will insert code after this rule matches which calls interp.print passing it a String argument with the text of *VAR*

- Tokens have a text field which is a String, but we can also access fields of rules with return values

A More Complex Case

Consider the the map matching of the exp rule

Exp Rule

```
exp returns [Expr e]
  ...
  | 'map' transformer e1=exp
    {$e = new MapExpr($transformer.t, $e1.e);}
  ;
```

- We can set the return value by assigning the variable (e) to a new object
- We may also access return values of other rules, in this case the transformer rule returns an Expr.Transformer
- We must provide an alias for the last exp to be able to refer to that specific rule, since we are already in the exp rule just having *$exp.e* is ambiguous

Trees

- There are ways to generate ASTs in **ANTLR** using their built in tree structure
- However the lab is not complex and this is not required, you can use your own simple data structure in order to represent the tree (see the example JavaDoc)

Demonstration

That's all there is to it! Observe...

Conclusion

You should now be well prepared to begin your lab

- Create your grammar file and insert code to interact with your interpreter
- Download **ANTLR**, set your classpath and run it
- ???
- Profit