

Getting Started with ANTLR

October 29, 2010

Tobias Gutzmann
Software Technology Group
School of Computer Science, Physics, and Mathematics,
Linnaeus University

(email: tobias.gutzmann@lnu.se)

Abstract

This short text is targeted to students who want to start using ANTLR. The aim is to bridge the gap from “never used ANTLR” to being able to read the instructions provided at <http://www.antlr.org>. It also contains a simple exercise suitable for making you more comfortable with ANTLR.

Our start kit is available at

<http://w3.msi.vxu.se/users/tgumsi/antlr>

Introduction

ANTLR (ANother Tool for Language Recognition) is a parser generator. A parser generator is a tool that reads a grammar specification and converts it to a program that can recognize matches to the grammar. ANTLR can generate code for many target programming languages, but this tutorial focuses on generating Java code. In addition to the parser generator itself, ANTLR provides other standard capabilities related to parser generation such as tree building, actions, error recovery, debugging, etc.

Getting Started

First of all, make sure that you have Java installed. ANTLR requires version 5.0 or higher, if you don't have it, you can download it from <http://java.sun.com>. You have to install a full *Java Development Kit* (JDK) – **having Eclipse installed is not sufficient**. It's also a good idea to update your environment variables according to:¹

```
CLASSPATH: "other paths"; .../jdk1.6.0_22/jre/lib/rt.jar
PATH: "other paths"; ... /jdk1.6.0_22/bin
```

The exact syntax depends on what operating system you are using. Verify that they work by typing `java -version`, and `javac -version` in a terminal window.

Next, download ANTLRWorks. You can always get the latest version from

<http://www.antlr.org/download.html>

Our starter kit (see below) contains version 1.4, which is the most recent one as of writing. Since this version sometimes makes trouble (at least for me), 1.3.1 is also included – use it in case something does not work properly. ANTLRWorks is a GUI development environment for ANTLR, which runs “out of the box” and includes ANTLR itself. You can usually start ANTLRWorks by double-clicking on it, if that does not work, type

```
java -jar antlrworks-1.4.jar
```

in a command line prompt.

ANTLRWorks should start up now. It asks you a couple of questions about yourself for a survey. Simply ignore it (“Don't Send”). If you have a personal firewall installed, you should grant ANTLRWorks access to the local internet, i.e., your computer, which is required for debugging.

Some visualization functions of ANTLRWorks make use of the *graphviz* software. Download and install it². Then, add the `bin` folder of the *graphviz* package to your path, or manually set the path to the `dot tool` (contained in the *graphviz* software) in ANTLRWorks/File/Preferences/Dot path.

If you run into trouble or strange behavior when using ANTLRWorks, confer “Troubleshooting ANTLRWorks” at the very end of this document.

¹Note: on Linux/MacOS, you separate paths by ':' instead of ','

²<http://www.graphviz.org/Download.php>

Getting Started using ANTLR

We have put together a tiny getting started kit that you can download from

<http://w3.msi.vxu.se/users/tgumsi/antlr/>

The kit contains the following files³:

```
Expr.g TreeExpr.g Test.java DFSPrinter.java
antlr-runtime-3.2.jar antlrworks-1.3.jar testing/
```

The files are:

1. `Expr.g`: This is a simple parser specification. This file can be used to *generate* a set of Java source files that implement a parser for a very simple expression language containing `+`, `-`, `*` and variable assignments.
2. `TreeExpr.g`: A simple parser specification which does the same as `Expr.g`, but creates an abstract syntax tree (AST). We go into detail later.
3. `Test.java`: A simple driver class (containing the main method) that initiates and starts the generated parser.
4. `DFSPrinter.java`: A class that prints an intended version of the resulting syntax tree
5. `antlr-runtime-3.2.jar`: The runtime classes for ANTLR, which are required to compile and run the parser generated by ANTLR.
6. `antlrworks-1.4.jar`: ANTLRWorks. See above on how to start this tool.
7. `antlrworks-1.3.1.jar`: ANTLRWorks, older version. See above on how to start this tool.

The directory `testing` contains a few very simple test programs that can be used for testing when you will work on the exercise that we present later in this tutorial.

The Example Application

We start with an example to get you used to using ANTLR and ANTLRWorks. You do not have to understand all the details of the ANTLR grammar files (`.g`) files in this section; we explain the single parts in the next section.

The getting starting kit includes the complete code of a parser for a simple calculator programming language. It implements a parser for the following grammar:

```
prog          ::= stat+
stat          ::= expr <NL> | <ID> '=' expr <NL> | <NL>
expr         ::= multExpr ( '+' multExpr | '-' multExpr )*
multExpr     ::= atom ( '*' atom )*
atom         ::= <INT> | <ID> | '(' expr ')'
```

³.g files kindly taken from <http://www.antlr.org/wiki/display/ANTLR3/Expression+evaluator>

Here `prog` is the start symbol and everything enclosed in `<>`-brackets are terminals (ID=Identifier, NL=newline, INT=Integer). The AntLR specification for this grammar can be found in the file `Expr.g`. The specification can be used, together with `Test.java`, to create a simple calculator application:

Open `Expr.g` in ANTLRWorks and generate the Java-files by running “Generate/Generate Code” (or press Ctrl-Shift-G). You will get a notification where the generated files, `ExprLexer.java` and `ExprParser.java`, have been written to (you can change this in the File/Preferences dialog). Copy `Test.java` to the same directory and compile them by typing

```
javac -cp antlr-runtime-3.2.jar;. Test.java ExprLexer.java ExprParser.java
```

on a command line (make sure `antlr-runtime-3.2.jar` is in the same directory; if you use for example Eclipse, add it to your current project). You can then run the parser by typing

```
java -cp antlr-runtime-3.2.jar;. Test
```

The program expects input following the grammar above. When you’re done, press CTRL-Z (on Linux/MacOS: CTRL-D) in order to finish the input. The calculator will then parse the grammar and evaluate each expression. A sample program input may look like this:

```
java -cp antlr-runtime-3.2.jar;. Test
4+5*5
(4+5)*5
x=5
y=10
x*y
~Z
```

The output will look like this:

```
29
45
50
```

Try a little bit around, e.g., use undefined variables or create syntax errors to see what happens. You will notice that ANTLR does not bail out at the first error, but rather uses automated error recovery.

The application entry point is the file `Test.java`. The connection to the parser part of the program looks like:

```
import org.antlr.runtime.*;

public class Test {
    public static void main(String[] args) throws Exception {
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        ExprLexer lexer = new ExprLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ExprParser parser = new ExprParser(tokens);
        parser.prog();
    }
}
```

First, we create a new `ANTLRInputStream` which takes a regular `java.io.InputStream` as parameter; in our case, we read from the console (`System.in`). Creating code that reads from a file is a viable alternative.

We pass this stream to the lexer (class `ExprLexer`, which is one of the generated files). The lexer creates a token stream (class `org.antlr.runtime.CommonTokenStream`) which is then passed to the constructor of class `ExprParser`, the other generated class.

The actual parsing is done by invoking a method corresponding to the start symbol (`prog`) of the grammar. Program interpretation is done during parsing, so the method `prog()` does not return any value.

Now that you know about the generated files, the requirement of using the antlr runtime library, and how to connect Java code to the generated code, you can learn about some nice features which ANTLRWorks provides: the Interpreter and the Debugger. The Interpreter takes a given input and creates a *parse tree*, while the debugger is a convenient and easy-to-use frontend which does what `Test.java` does (and more)⁴. A tutorial providing step-by-step instructions on how to use the Interpreter and the Debugger is available on the web⁵. Follow this tutorial, and play around with both the debugger and the interpreter - they will be very useful to you in the future!

Building Abstract Syntax Trees

So far, the calculator application interprets the user input on-the-fly while parsing. However, for more sophisticated applications like compilers, an abstract syntax tree (AST) is much more convenient to use.

Start ANTLRWorks, and open the file `TreeExpr.g`. It is similar to `Expr.g`, but the generated parser creates an AST instead of interpreting the simple expression grammar. Generate the code, and run the debugger (CTRL-D). Provide some proper input to the grammar, and make sure that “prog” is selected in the drop-down box “start rule”. Once the debugger run is over, you can see the AST by clicking on the “AST” icon on the bottom of ANTLRWorks. It is much more “cleaned up” than the parse tree which you have seen before.

ANTLR Grammar Files

This is just a brief introduction to parser specification using ANTLR. Further details can be found at the ANTLR home page at www.antlr.org. The actual specification takes place in ANTLR files with postfix `.g`.

Our ANTLR grammar files from above can be roughly divided into three parts:

1. A **header** where you decide the properties (e.g. name, options, etc.) of the parser class to be generated. In our example it looks like this:

```
grammar TreeExpr;
options {
    output=AST;
    ASTLabelType=CommonTree;
}
```

⁴if you get a `Compiler Exception`, then `javac` is not found on the path; add it to your system’s path, or set it in ANTLRWorks under `File/Preferences/Compiler`.

⁵<http://www.antlr.org/works/help/tutorial/calculator.html>

```
}
```

2. A section containing the **lexical specification** (the tokens to be recognized). Here you specify the token names and their corresponding regular expressions. Our example specification looks like:

```
ID : ('a'..'z'|'A'..'Z')+ ;
INT : '0'..'9'+ ;
NEWLINE: '\r'? '\n' ;
WS : (' '|'\t')+ { $channel=HIDDEN; } ;
```

This means that integers have the token INT with corresponding regular expression $([0-9])^+$ and that whitespace (WS) consists of regular “space” and/or “tabs”. `$channel=HIDDEN` tells the lexer not to generate tokens for the parser. An identifier ID consists of one or more letters, and a newline is specified either as `'\n'` or `'\r\n'`, to meet both Unix and Windows standards.

3. A final section containing the **context-free grammar** defining the syntax analysis, as well as, for example, *actions* to be taken (in case of the Expr parser), or *tree rewrite rules* (in case of the TreeExpr parser). ANTLR uses the Extended Backus-Nauer Form (EBNF) where the standard notation (BNF) is extended with `*`, `+`, `?`. A piece of our example specification looks like:

```
stat:  expr NEWLINE {System.out.println($expr.value);}
      | ID '=' expr NEWLINE
        {memory.put($ID.text, new Integer($expr.value));}
      | NEWLINE
      ;
expr returns [int value]
:  e=multExpr {$value = $e.value;}
   ( '+' e=multExpr {$value += $e.value;}
   | '-' e=multExpr {$value -= $e.value;}
   )*
;

```

The code in curly brackets `{ }` is called *actions*. The code is executed once the parser has decided to take the corresponding production; in the first *stat* alternative, the result of the expression *expr* is printed to the console once a NEWLINE is encountered. The “variable” `$expr.value` is declared in the definition of the production *expr return [int value]*. The actual computation of the value of an expression is, again, performed through actions. You won’t need actions in this course, but it is good to know that they exist and how they work. Maybe they come in handy one day...

In case of the TreeExpr grammar, our example specification looks like this:

```
stat:  expr NEWLINE      -> expr
      | ID ASSIGN expr NEWLINE -> ^(ASSIGN ID expr)
      | NEWLINE          ->
      ;
expr:  multExpr ((PLUS^|MINUS^) multExpr)*
;

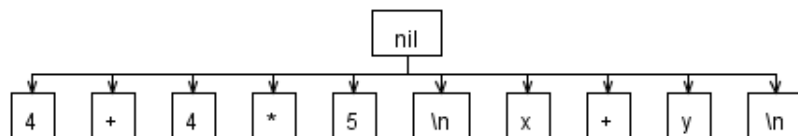
```

Here, we have replaced '=' etc. by the lexical tokens, e.g. ASSIGN. This is only for “cosmetic” purposes and makes programming the DFSPrinter easier.

Then, we use *operators* and *rewrite rules*. If we did not use these, our target AST for the program

```
4+4*5
x+y
```

would look like this:

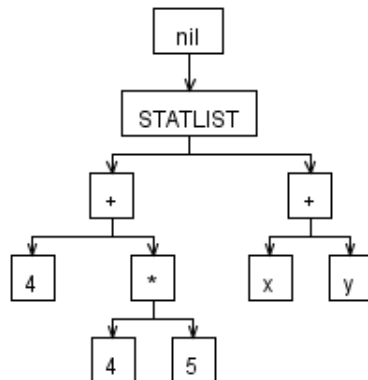


The first rewrite rule, `stat: expr NEWLINE -> expr`, simply tells the parser not to add the newline to the AST.

`ID ASSIGN expr NEWLINE -> ^(ASSIGN ID expr)` tells the parser to create a node for ASSIGN and add *ID* and *expr* as subnodes to the assignment operator ASSIGN (and, again, do not add the newline to the AST).

In the production `multExpr ((PLUS^|MINUS^) multExpr)*`, we use the *operator* ^ to tell that, once we encounter a '+' or a '-', this terminal becomes a subtree root-node in the AST. Note that, unlike in rewrite rules, the ^ stands *behind* the terminal.

When using these rewrite rules, our target AST has this nice form:



More information on tree construction can be found on the web⁶.

A general overview of how ANTLR grammar files look like is, suprise, on the web⁷.

⁶<http://www.antlr.org/wiki/display/ANTLR3/Tree+construction>

⁷<http://www.antlr.org/wiki/display/ANTLR3/Grammars>

Lookahead Handling

As of version 3.0, ANTLR creates LL(*) parsers, i.e., there is no need to take care of lookahead handling at all. While some gurus may call that a sin, ANTLR does this *infinite lookahead handling* in a rather performant way. You can read more about this on the web if you are interested ⁸.

Sources of Information

ANTLR has a home page (<http://www.antlr.org>) where everything you need can be found. There you can:

- Download ANTLR
- Read tutorials
- Find examples
- Look at grammars for the complete Java and many other languages

Troubleshooting ANTLR

Some common mistakes that are made when using ANTLR are listed here.

- Do not use Java keywords as production names, otherwise the generated code will not compile.
- Lexer rules start with capital letters, parser rules with a lower case letter.

Troubleshooting ANTLRWorks

While ANTLRWorks is a very nice and powerful tool, it unfortunately has still some annoying bugs:

- If you get a “Connection Error - cannot launch the debugger - Timeout waiting to connect to the remote debugger”, then restart ANTLRWorks.
- When you start the debugger, it will compile the .g file to the output files in a “special way”, so that you cannot use them from a program of your own, i.e., the output is different from the output that you would get when using the option “Generate/Generate Code”. Simply generate the code again.
- If the menu items for the debugger are grayed-out, i.e., you cannot start the debugger, or you cannot edit the .g file from within ANTLRWorks, then the debugger is most likely still running. Click on the stop-button (filled square box) on the very left of ANTLRWorks.
- When debugging, on the bottom-left you should see the input to your grammar with the single tokens being clickable. If something seems odd there, check that, when providing the input to the debug run, you have the line-endings “Unix (LF)” selected – even on computers running Windows!

⁸<http://www.artima.com/lejava/articles/antlr.3.html>

Goal	:=	"program" Identifier "{" (VarDeclaration)* Statement* "}"
VarDeclaration	:=	Type Identifier ("=" Integer)?";
Type	:=	"int"
Statement	:=	"{" (Statement)* "}" "if" "(" Expression ")" Statement "else" Statement "while" "(" Expression ")" Statement "print" "(" Identifier ")" ";" Identifier "=" Expression ";"
Expression	:=	Expression ("<" "+" "-" "*" "/") Expression Identifier
Identifier	:=	<IDENTIFIER>

Table 1: BNF for Program

Example Exercise

First of all the most important thing: **The exercise described here is for you to try out ANTLR. It is not to be submitted and will therefore not be graded.** You can use it as a “playground” to try out different approaches to AST construction etc., before you get started with the “real” assignment.

In this exercise you should write a new parser `SimpleParser` to be able to parse programs like

```

program Compute {
  //Declarations
  int n = 100;
  int count = 2;
  int fib;
  int prev = 1;
  int prevPrev = 1;

  //Compute fibonacci(n)
  if (n > 2) {
    while (count < n) {
      fib = prev + prevPrev;
      prevPrev = prev;
      prev = fib;
      count = count + 1;
    }
  } else {
    fib = 1;
  }

  if (fib > 1) {
    print(fib);
  }
}

```

We suggest to use `TreeExpr.g` as a base and then perform the following steps. Note that it is a good idea to first write the grammar, and leave out the tree rewrite rules; after your grammar accepts all the input files provided for testing, you can go on and add the rewrite rules. As a good starting point you can try to write the grammar for this particular program using the grammar description presented in Table 1. And then apply the following steps to your solution.

1. **Program declaration:** Require the grammar to be surrounded by 'program' followed by a name for the program

- Comments** Add comments to the lexer rules. If you're unsure on how to do this, just "steal" (yes, this time it's ok to copy!) from a complete Java grammar⁹.
- Variable declaration:** Change the grammar so that variables must be explicitly declared ('int'). They may be initialized; initializations may look like this:

```
int first = 1;
int second = first + 2;
int third;
```

Use the test program `testing\assignments.prgm` to check if it works.

- Boolean variables and expressions:** Introduce a new type `boolean`, the boolean literals `true` and `false`, and the operations `<`, `>` and `&&` (logical AND). Make sure that correct operator priorities are explicit in the resulting syntax tree. The following type of program should be accepted:

```
//Declarations
int a = 1;
boolean v1;
boolean v2 = true;

//Statements
v1 = 8 < a + 1;           //Rhs Priority: 8 < (a+1)
v2 = 4 < a && 5 < a;      //Rhs Priority: (4<a) && (5<a)
print(true && false);
```

Use the test program `testing\boolean.prgm` to check if it works.

- While and If statements:** Introduce `while-` and `if-` statements that look like:

```
while ( ... ) { |   if ( ... ) { |   if ( ... ) {
...           |   ...           |   ...
}             |   }             |   } else {
              |               |   ...
              |               |   }
              |               |   }
```

That is, the `else` part in the `if`-statements should be optional. It should be possible to have nested statements (statements within statements). Use the test program `testing\statements.prgm` to check if it works.

- Semantic Analysis:** The parser checks if the input is syntactically correct and constructs a syntax tree. The next phase of a compiler is called the semantical analysis. This is where we find programming errors not caught by the parser. For example, the following code has correct syntax but is still wrong since we are using a variable `b` that we haven't declared.

```
int a = 1;
b = a+3;
```

Write a class `CheckDeclared.java` that traverses the syntax tree and checks that all variables are declared before they are used. Hint: Take a look in `DFSPrinter.java` to see how to traverse the tree.

- What remains?:** What more should be checked before we can say that the program is correct?

⁹<http://wwwantlr.org/grammar/1152141644268/Java.g>