ANTLR — ANother TutoRiaL

Karl Stroetmann

March 29, 2007

Contents

1 Introduction

This tutorial introduces ANTLR, which is short for another tool for language recognition. ANTLR [PQ95] is a scanner and parser generator. There are other tools offering similar functionality, for example LEX [LS75] generates scanners and YACC [Joh75] generates parsers. LEX and YACC, or their modern variants FLEX [Nic93] and BISON [DS90], are often used in conjunction. The book by Levine, Mason, and Brown [LMB92] shows LEX and *Yacc* working together. In contrast to LEX and YACC, which can only be used for `C` and `C++` programs, ANTLR works with a number of different languages. In particular, ANTLR works with *Java*. There is even an *Eclipse* [Ecl03] plugin for ANTLR available at `http://antlreclipse.sourceforge.net/`. ANTLR itself is available from its homepage `http://www.antlr.org`. ANTLR creates top-down parsers, also known as `LL`(k)-parsers [ASU86]. In contrast, YACC generates bottom-up parsers, technically known as LALR(1)-parsers. While bottom-up parsers can be more powerful than top-down parsers, the latter are, in general, easier to understand and to maintain.

ANTLR is a complex tool offering many features. Our intent is not to introduce all of them. Instead, we just show how simple parsers can be readily build with ANTLR. Towards this goal, the next section shows how a simple scanner can be developed with ANTLR. We proceed in Section 3 to develop a simple parser for arithmetical expressions. This parser will do nothing more than regognize an arithmetical expression. In Section 5 we refine this parser to actually do something with these expressions: For every arithmetical expression parsed the parser will construct a *Java* object representing this expression. To make this example interesting we show how this expression can then be used to differentiate arithmetical expressions symbolically.

2 Implementing a Simple Scanner

In this section we develop a scanner that can extract one line comments from *Java* programs, i.e. the scanner will extract all text from a file that starts with two slashes "`//`" and reaches to the end of the line. Figure 1 on page 2 shows the ANTLR grammar file `comments.g` specifiying the scanner. All ANTLR keywords have been underlined. We discuss this grammar file line by line.

```
1   class CommentsLexer extends Lexer;
2
3   options {
4        k              = 2;
5        charVocabulary = '\u0000'..'\u00FF';
6   }
7
8   COMMENT : "//" (~'\n')* ;
9
10  REST    : . ;
```

Figure 1: A scanner extracting one line comments.

1. Line 1 specifies that this file declares a scanner named `CommentsLexer`. This line tells ANTLR that its task is to generate a *Java* file `CommentsLexer.java` defining the class `CommentsLexer` which will contain a *scanner*. Here a scanner is an object that has, among others, the methods

   nextToken()

   which reads the next available token from the input stream.

2. Line 3 – 6 contains the options section. We have used two options:

   (a) The first option "`k = 2`" specifies that the scanner uses a lookahead of two characters. We will discuss the topic of lookahead later.

   (b) The keyword `charVocabulary` defines the set of characters that our scanner understands. The default vocabulary consists of just the ASCII characters. In order to be able to process German umlauts we have extended the vocabulary to contain all unicode characters representable by one byte.

3. Line 8 contains the first of two grammar rules. The first rule is named `COMMENT`. The body of the rule is the text to the right of the colon "`:`". This definition states that a comment should be any text beginning with the character pair "`//`" and followed by any number of characters that are different from the newline character. Here, `'\n'` denotes the newline character. The character "`~`" serves as negation operator, so that the string "`~'\n'`" denotes any character different from the newline character. Since we want to admit any number of non-newline characters, we have enclosed this string in parentheses, followed by a star, so that "`(~'\n')*`" specifies any number of non-newline characters.

   Note the space character between the first string "`"//"`" and the second string "`(~'\n')*`". In ANTLR, white space has no significance, so this space does not influence the interpretation of the rule. Therefore, whitespace can be used to format rules in order to increase their readability. This behaviour is different from LEX and can cause confusion for users accustomed to LEX.

4. Line 10 contains the second rule. The name of this rule is `Rest`. Its body is the string "`.`". It matches any single character.

If the specification shown in Figure 1 is saved in a file `comments.g`, we can construct a lexer by issueing the command

```
java antlr.Tool comments.g
```

This call generates the following files:

1. `CommentsLexer.java` implements the scanner.

2. `CommentsLexerTokenTypes.java` defines the token types. These are integers, representing the different types of tokens that the scanner recognizes. In this case, the file looks as follows:

```
1  public interface CommentsLexerTokenTypes {
2      int EOF = 1;
3      int NULL_TREE_LOOKAHEAD = 3;
4      int COMMENT = 4;
5      int REST = 5;
6  }
```

   We can ignore the definition of `NULL_TREE_LOOKAHEAD`. The other token types should be obvious: `EOF` signals *end of file* and the token types `COMMENT` and `REST` correspond to the lexical rules with the same name.

3. `CommentsLexerTokenTypes.txt` contains more or less the same information as the file `CommentsLexerTokenTypes.java`. The file looks as follows:

```
1  // $ANTLR : comments.g -> CommentsLexerTokenTypes.txt$
2  CommentsLexer    // output token vocab name
3  COMMENT=4
4  REST=5
```

Next, we need a driver for our scanner. Figure 2 on page 4 shows the implementation of such a simple driver.

1. Line 1 imports everything from the package `antlr`, in particular the definition of the class `Token`.
2. Line 5 creates a lexer by feeding an `InputStream` into the constructor *CommentsLexer*.
3. Line 8 reads a token.
4. Line 9 checks the type of this token. If the token is of type `COMMENT`, then line 10 prints the token using the method `getText()`. This method returns the string that was matched by the corresponding rule.
5. Line 8 to 10 are repeated until an *end-of-file* token is encountered.

3 A Parser for Arithmetic Expressions

Our next example deals with arithmetic expressions. We build a parser recognizing arithmetical expressions built from constants and variables using the operator symbols "`+`", "`-`", "`*`", and "`/`". Furthermore, the function symbols "`exp`" and "`ln`" can be used in arithmetical expressions.

We start our construction with the scanner. Figure 3 on page 4 shows its specification.

1. Lines 3 to 8 define rules for recognizing operator symbols and parentheses.
2. Line 10 contains the rule for recognizing identifiers, i.e. variables. Its body states that an identifier starts with a letter and is followed by any number of letters and digits. In ANTLR, the operator "`|`" is used to denote disjunctions. Therefore, the expression

   ```
   LETTER | DIGIT
   ```

   denotes any character that is either a letter or a digit.

```
1  import antlr.*;
2
3  public class Comments {
4      public static void main(String args[]) throws Exception {
5          CommentsLexer lexer = new CommentsLexer(System.in);
6          Token         token;
7          do {
8              token = lexer.nextToken();
9              if (token.getType() == CommentsLexerTokenTypes.COMMENT) {
10                 System.out.println(token.getText());
11             }
12         } while (token.getType() != CommentsLexerTokenTypes.EOF);
13     }
14 }
```

Figure 2: A driver for the scanner.

```
1  class ExpressionLexer extends Lexer;
2
3  LPAREN : '(';
4  RPAREN : ')';
5  PLUS   : '+';
6  MINUS  : '-';
7  STAR   : '*';
8  SLASH  : '/';
9
10 IDENTIFIER : LETTER (LETTER | DIGIT)* ;
11
12 DOUBLE : (DIGIT)+ ('.' (DIGIT)+)? ;
13
14 protected DIGIT  : '0'..'9' ;
15 protected LETTER : 'a'..'z' ;
16
17 WS   :
18      ( ' '
19      | '\t'
20      | '\r' '\n'
21      | '\n'
22      ) { $setType(Token.SKIP); }
23      ;
```

Figure 3: A simple scanner for arithmetical expressions.

3. Line 12 specifies the rule for recognizing floating point numbers. According to this rule, these consist of a positive number of digits, followed by an optional fractional part which starts with a dot followed by a positive number of digits. This definition shows two ANTLR operators:

   (a) For an expression e, the expression $(e)+$ denotes a positive number of es.

   (b) For an expression e, the expression $(e)?$ denotes an optional occurrence of e, so either

one occurrence or no occurrence.

Note also that the dot occurring in this rule must be enclosed in single quotes since otherwise it would be interpreted as a wildcard denoting an arbitrary character.

4. Line 14 shows the definition of digits, using the range operator "`..`". In Lex, this range would have been denoted as "`[0-9]`". The name of the rule is preceded by the keyword "`protected`". This keyword specifies that the token `DIGIT` is used only locally. In our example, `DIGIT` is used in the definition of `DOUBLE`. Since `DIGIT` is declared as protected, the scanner will never return a token of type `DIGIT`. Rather, `DIGIT`s will be joined to make `DOUBLE`s.

5. Line 15 defines letters.

6. The rule defining *white space* in lines 17 – 23 has a *semantic action*, given by the program text that is enclosed in curly braces. In this case,

   ```
   { $setType(Token.SKIP); }
   ```

   sets the token type to `SKIP`. This is a dummy type, which is never returned, but just SKIPped.

```
1   class ExpressionParser extends Parser;
2
3   expr    : product ( "+" product | "-" product )* ;
4
5   product : factor ( "*" factor | "/" factor )* ;
6
7   factor  : "(" expr ")"
8           | "exp" "(" expr ")"
9           | "ln"  "(" expr ")"
10          | DOUBLE
11          | IDENTIFIER
12          ;
```

Figure 4: A simple parser

We proceed to specify the parser for arithmetical expressions. Figure 4 on page 5 shows the definition of a suitable solution.

1. The keywords "`extends Parser`" in Line 1 specifies that a parser, rather than a scanner is defined.

   Next, there are three rules defining the nonterminals *expr*, *product*, and *factor*.

2. Line 3 defines an expression as a product followed by any number of products preceded by one of the operators "`+`" or "`-`".

3. Line 5 specifies a product as a factor followed by any number of factors preceded by one of the operators "`*`" or "`/`".

4. Line 7 specifies a factor as one of the following:

   (a) An opening parenthesis "`(`", followed by an expression, followed by a closing parenthesis "`)`".

   (b) The string "`exp`", followed by an opening parenthesis "`(`", followed by an expression, followed by a closing parenthesis "`)`".

(c) The string "`ln`", followed by an opening parenthesis "`(`", followed by an expression, followed by a closing parenthesis "`)`".

(d) A token `DOUBLE` denoting a floating point number.

(e) A token `IDENTIFIER` denoting a variable.

We know that the strings `DOUBLE` and `IDENTIFIER` refer to tokens and not to non-terminals because the names of tokens are made up from uppercase letters, while the names of non-terminals consist solely of lowercase letters.

5. Looking back at the scanner in Figure 3 we notice that there are no lexical rules to recognize the strings "`exp`" or "`ln`". Since these strings are longer than one character, the parser automatically generates rules for these tokens. Of course, it would be nice if ANTLR could also generate lexical rules for one character tokens. It remains one of the mysteries of ANTLR that this convenience is not offered.

```
1  public class ArithmeticalExpression {
2      public static void main(String args[]) throws Exception {
3          ExpressionLexer  lexer  = new ExpressionLexer(System.in);
4          ExpressionParser parser = new ExpressionParser(lexer);
5          parser.expr();
6      }
7  }
```

Figure 5: A driver for the parser recognizing arithmetical expressions.

Finally, Figure 5 on page 6 shows a driver that can be used to test the parser. If we put both the code of Figure 3 and Figure 4 into one file, we can generate a parser to recognize arithmetical expressions. After compiling, we run it on an example input string such as "`echo "x * exp(x)"` by issueing the command:

```
echo "x * exp(x)" | java ArithmeticalExpression
```

Since the string "`x * exp(x)`" is a valid expression according to our grammar, the parser will silently accept it. If instead we issue the command

```
echo "exp(x--)" | java ArithmeticalExpression
```

the parser will produce the error message

```
line 1:7:  unexpected token:  -
```

informing us that our input does not conform to the grammar. In fact, the parser realizes that after the second dash "`-`", there is no way to complete the input that yields a correct arithmetical expression.

4 Symbolic Differentiation

The preceeding example is rather dull, since nothing is done with the recognized input. In order to make it more interesting, we will next store, represent and process the recognized expressions as objects. As a showcase, we will write a *symbolic differentiator*, i.e. a program that reads an arithmetical expression, interprets it as a mathematical function and then differentiates it with respect to the variable `x`. For example, when given the input "`x * exp(x)`" the program will produce the output

```
1 * exp(x) + x * exp(x).
```

In order to carry out this plan, we first need to decide how expressions can be represented as objects. We will develop an abstract class *Expr* representing arithmetical expressions as objects.

Figure 6 shows the definition of this class. An *Expr* is either a *Sum* or a *Product* or $\cdots$, so we represent it by an abstract class *Expr* with concrete subclasses *Sum*, *Product*, $\cdots$, *Logarithm*.

```
1  public abstract class Expr {
2      public abstract Expr diff(Variable x);
3  }
```

Figure 6: Implementation of the abstract class *Expr*.

```
1  class ExpressionParser extends Parser;
2
3  expr returns [Expr r = null]
4      { Expr p; }
5      : r = product
6        ( "+" p = product { r = new Sum(r, p);        }
7        | "-" p = product { r = new Difference(r, p); }
8        )*
9      ;
10
11 product returns [Expr r = null]
12     { Expr f; }
13     : r = factor
14       ( "*" f = factor { r = new Product(r, f);  }
15       | "/" f = factor { r = new Quotient(r, f); }
16       )*
17     ;
18
19 factor returns [Expr r = null]
20     { Expr a; }
21     : "(" r = expr ")"
22     | "exp" "(" a = expr ")" { r = new Exponential(a); }
23     | "ln"  "(" a = expr ")" { r = new Logarithm(a);   }
24     | d  : DOUBLE        { r = new Constant(Double.valueOf(d.getText())); }
25     | id : IDENTIFIER    { r = new Variable(id.getText());                }
26     ;
```

Figure 7: A parser recognizing and constructing expressions.

In order not to be distracted from our main goal we proceed with the specification of the parser which is shown in Figure 7. The grammar is the same as before, but now the parser constructs also objects representing the expressions that have been parsed.

1. The rule to recognize *expr* is now considerable more complex:

   (a) In line 3, the string "`returns [Expr r = null]`" states that every time the parser sees an `expr` it will construct an object of type *Expr*. The variable `r` will reference this object. We refer to this variable as *result variable*. It needs to be initialized, so it is set to `null`.

   Each grammar rule specifying an object that we want to capture and manipulate, begins with specifying such a *result variable*.

(b) In line 4, the string "{ `Expr p;` }" declares `p` as a local variable of this rule.

As before, an `expr` is specified as a `product` folllowed by any number of procucts that are either preceded by an operator "`+`" or by an operator "`-`". However, the grammar rules are now enriched with variable assignments and semantic actions.

(c) In line 5, the result variable `r` is initialized to the expression that is constructed when the first `product` is seen.

(d) In line 6 the assignment

```
p = product
```

sets the local variable `p` to the object that is constructed when the accompanying `product` is parsed. At the end of this line, the action

```
{ r = new Sum(r, p); }
```

specifies that the result variable is updated by constructing a new object of class *Sum* using the original value of the result variable `r` as first summand and the value of the local variable `p` as the second summand.

The next line deals in the same way with differences, and the remaining code similarly specifies the non-terminals `product` and `factor`.

(e) Line 24 shows how tokens are dealt with. First, the string

```
d:  DOUBLE
```

declares `d` to represent a token forming a floating point number. Then, the action

```
{ r = new Constant(Double.valueOf(d.getText())); }
```

extracts the string attached to this token, turns it into an object of type `Double`, which is then used to construct an object of type `Constant`. Finally, the result variable `r` is set to this object.

Note that with tokens the syntax for extracting the value is different from the syntax used for non-terminals. For non-terminals it is

var = `non-terminal`

while for tokens ANTLR uses the following syntax:

var : `token`

```
1  public class Symbolic {
2      public static void main(String args[]) throws Exception {
3          ExpressionLexer  lexer  = new ExpressionLexer(System.in);
4          ExpressionParser parser = new ExpressionParser(lexer);
5          try {
6              Expr     expr  = parser.expr();
7              Variable x     = new Variable("x");
8              Expr     dExpr = expr.diff(x);
9              System.out.println("d (" + expr + ") / dx = " + dExpr);
10         } catch (Exception e) {}
11     }
12 }
```

Figure 8: A driver program for the parser.

The scanner can be used unchanged from the previous section. All that is missing now is a driver for the parser. Figure 8 shows the implementation of class *Symbolic*. The method `main()` reads and parses an arithmetical expression `expr` and then calls the method `diff()` for this expression.

This method returns the expression `dExpr` representing the result of differentiating `expr` with respect to `x`.

In order to make this tutorial self contained we proceed to discuss the implementation of the classes derived from *Expr*. These classes shown in the figures 9, 10, 11, 12, 13, 14, 15, and 16 on the following pages. Figure 9 shows the implementation of *Constant* which is used to represent constant functions. These can be represented by their fixed outcome, which we store in the member variable `mValue`. Since

$$\frac{d\,c}{dx} = 0$$

for any constant c, the method *diff()* in class *Constant* returns the *constant* zero.

```
1  public class Constant extends Expr {
2      Double mValue;
3
4      public Constant(Double value) { mValue = value; }
5
6      //  d c / dx  =  0
7      public Constant diff(Variable x) { return new Constant(0.0); }
8
9      public String toString() { return mValue.toString(); }
10 }
```

Figure 9: The class *Constant*.

Figure 10 shows the implementation of class *Variable*. A variable is fully specified by its name, which is therefore stored in the member variable `mName`. When differentiating a variable y with respect to another variable x, there are two cases: If x and y are identical, the result is 1, otherwise 0.

```
1  public class Variable extends Expr {
2      String mName;
3
4      public Variable(String name) { mName = name; }
5
6      public Constant diff(Variable x) {
7          if (mName.equals(x.mName)) {
8              //  d x / dx  =  1
9              return new Constant(1.0);
10         }
11         //  x ≠ y → d y / dx  =  0
12         return new Constant(0.0);
13     }
14
15     public String toString() { return mName; }
16 }
```

Figure 10: The class *Variable*.

Figure 11 shows the implementation of *Sum*. The member variable `mLhs` and `mRhs` store the summands, so that objects of this class represent the sum

`mLhs` + `mRhs`.

In order to differentiate a sum, we use the familiar rule:

$$\frac{d}{dx}(u+v) = \frac{d\,u}{dx} + \frac{d\,v}{dx}.$$

It is easy to see that the method *diff()* of class *Sum* constructs this expression.

```
1  public class Sum extends Expr {
2      Expr mLhs, mRhs;
3
4      public Sum(Expr lhs, Expr rhs) {
5          mLhs = lhs;
6          mRhs = rhs;
7      }
8
9      //  d/dx(u+v) = du/dx + dv/dx
10     public Sum diff(Variable x) {
11         return new Sum(mLhs.diff(x), mRhs.diff(x));
12     }
13
14     public String toString() {
15         return mLhs.toString() + " + " + mRhs.toString();
16     }
17 }
```

Figure 11: The class *Sum*.

The remaining classes are quite similar to the class *Sum*:

1. The class *Diff* in Figure 12 implements the rule $\frac{d}{dx}(u-v) = \frac{d\,u}{dx} - \frac{d\,v}{dx}$.
2. The class *Product* in Figure 13 implements the rule $\frac{d}{dx}(u*v) = \frac{d\,u}{dx} * v + u * \frac{d\,v}{dx}$.
3. The class *Quotient* in Figure 14 implements the rule $\frac{d}{dx}(u/v) = \frac{\frac{d\,u}{dx} * v - u * \frac{d\,v}{dx}}{v * v}$.
4. The class *Exponential* in Figure 15 implements the rule $\frac{d}{dx}\exp(u) = \frac{d\,u}{dx} * \exp(u)$.
5. The class *Logarithm* in Figure 16 implements the rule $\frac{d}{dx}\ln(u) = \frac{\frac{d}{dx}u}{u}$. c

5 Conclusion

ANTLR provides many more features and options. For those interested, the web site `www.antlr.org` gives pointers to several articles describing additional featuers. Furthermore, the software distribution contains an extensive manual [PLS05] which is also available online.

```
1  public class Difference extends Expr {
2      Expr mLhs, mRhs;
3
4      public Difference(Expr lhs, Expr rhs) {
5          mLhs = lhs;
6          mRhs = rhs;
7      }
8
9      //  d/dx(u - v)  =  du/dx  -  dv/dx
10     public Difference diff(Variable x) {
11         return new Difference(mLhs.diff(x), mRhs.diff(x));
12     }
13
14     public String toString() {
15         return mLhs.toString() + " - (" + mRhs.toString() + ")";
16     }
17 }
```

Figure 12: The class *Difference*.

```
1  public class Product extends Expr {
2      Expr mLhs, mRhs;
3
4      public Product(Expr lhs, Expr rhs) {
5          mLhs = lhs;
6          mRhs = rhs;
7      }
8
9      //  d/dx(u/v)  =  du/dx * v  +  u * dv/dx
10     public Sum diff(Variable x) {
11         return new Sum(new Product(mLhs.diff(x), mRhs),
12                        new Product(mLhs, mRhs.diff(x)));
13     }
14
15     public String toString() {
16         return "(" + mLhs.toString() + ") * (" + mRhs.toString() + ")";
17     }
18 }
```

Figure 13: The class *Product*.

```
1  public class Quotient extends Expr {
2      Expr mLhs, mRhs;
3
4      public Quotient(Expr lhs, Expr rhs) {
5          mLhs = lhs;
6          mRhs = rhs;
7      }
8
9      public Expr getRhs() { return mRhs; }
10
11     //   d/dx (u/v)  =  (du/dx * v  -  u * dv/dx) / (v * v)
12     public Quotient diff(Variable x) {
13         return new
14             Quotient(new Difference(new Product(mLhs.diff(x), mRhs),
15                                     new Product(mLhs, mRhs.diff(x))),
16                      new Product(mRhs, mRhs));
17     }
18
19     public String toString() {
20         return "(" + mLhs.toString() + ") / (" + mRhs.toString() + ")";
21     }
22 }
```

Figure 14: The class *Quotient*.

```
1  public class Exponential extends Expr {
2      Expr mArg;
3
4      public Exponential(Expr arg) { mArg = arg; }
5
6      //   d/dx exp(u)  =  du/dx * exp(u)
7      public Product diff(Variable x) {
8          return new Product(mArg.diff(x), this);
9      }
10
11     public String toString() { return "exp(" + mArg + ")"; }
12 }
```

Figure 15: The class *Exponential*.

```
1  public class Logarithm extends Expr {
2      Expr mArg;
3
4      public Logarithm(Expr arg) { mArg = arg; }
5
6      //  d/dx ln(u)  =  (d/dx u) / u
7      public Quotient diff(Variable x) {
8          return new Quotient(mArg.diff(x), mArg);
9      }
10
11     public String toString() { return "ln(" + mArg + ")"; }
12 }
```

Figure 16: The class *Logarithm*.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[DS90] Charles Donnelly and Richard Stallman. *Bison: the Yacc-compatible parser generator*. pub-FSF, pub-FSF:adr, Bison Version 1.12 edition, December 1990.

[Ecl03] Eclipse Platform: Technical Overview, 2003. http://www.eclipse.org/whitepapers/-eclipse-overview.pdf.

[Joh75] S. C. Johnson. Yacc: Yet another compiler compiler. Computer Science Technical Report #32, Bell Laboratories, Murray Hill, NJ, 1975.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O'Reilly, Sebastopol, 2 edition, 1992.

[LS75] M. E. Lesk and E. Schmidt. Lex — A lexical analyzer generator. Computing Science Technical Report No. 39, Bell Laboratories, Murray Hill, N.J., 1975.

[Nic93] G. T. Nicol. *Flex: The Lexical Scanner Generator, for Flex Version 2.3.7*. pub-FSF, pub-FSF:adr, 1993.

[PLS05] Terence John Parr, John Lilley, and Scott Stanchfield. ANTLR 2.75 reference manual. Technical report, 2005. Available at: `http://www.antlr.org/doc/index.html`.

[PQ95] Terence J. Parr and Russell W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software — Practice and Experience*, 25(7):789–810, 1995.