An Introduction To ANTLR

Terence Parr

Introduction

During the 1980s I built many many recognizers and translators by hand and finally got disgusted enough to try to automate the process; whence my motto:

> "*Why program by hand in five days what you can spend five years of your life automating.*"

The benefit of building so many projects by hand is that you can see the commonality and what can reasonably expected to be formalized and automated. I didn't understand `yacc` too well back then and wanted something that reproduced what I built by hand anyway. ANTLR is the result (originally called PCCTS actually). I've been working on it for well over a decade now. [See **a quick history** for more details].

ANTLR, ANother Tool for Language Recognition, is a tool that accepts grammatical language descriptions and generates programs that recognize sentences in those languages. As part of a translator, you may augment your grammars with simple operators and actions to tell ANTLR how to build ASTs and how to generate output. ANTLR knows how to generate recognizers in Java, C++, C#, and soon Python.

ANTLR knows how to build recognizers that apply grammatical structure to three different kinds of input: (i) character streams, (ii) token streams, and (iii) two-dimensional trees structures. Naturally these correspond to lexers, parsers, and tree walkers. The syntax for specifying these grammars, the *meta-language*, is nearly identical in all cases.

Once you are comfortable with ANTLR or a similar tool, you will start to see programming in a new light. Many tasks cry out for language solutions well outside the stereotypical programming language genre. For example, these course notes are written in TML, Terence's Markup Language. I hate typing HTML and so I built a trivial translator using ANTLR to convert text (with a few extra goodies and conventions) into HTML or PDF or whatever I bother to write a generator for.

Finally, let me point out that ANTLR is just a tool--that's all. It helps you build software by automating the well-understood tedious components,

but does not attempt to let you specify an entire compiler, for example, in a single description. Tools that claim this sort of thing are great for writing amazing "one liners" for publishing journal papers, but fail miserably on real projects.

There are, as of early 2003, almost 5,000 ANTLR downloads a month. ANTLR is completely in the public domain without even a copyright and comes with complete source code.

These notes assume you are familiar with basic recognition and translation concepts. For now, you need to get familiar with ANTLR's metalanguage and what it generates. Later, we will focus on building complicated translators.

A Gentle Introduction to ANTLR Syntax

Getting to know ANTLR is best done via example. A simple calculator is often used to get started and with good reason: it's easy to understand and simple. There are a number of similar examples and tutorials for ANTLR, but I will describe a calculator here in my own words. First we will build something that directly evaluates simple expressions. Then we will generate trees and evaluate the trees to get the same answer.

While you know that eventually you have to break up an input stream of characters into tokens, thinking about the grammatical structure of an expression is good place to start.

Syntax-directed execution

Recognition

Let's accept arithmetic expressions with operators plus, minus, and multiply such as `3+4*5-1` or expressions with parentheses such as `(3+4)*5` to enforce an evaluation order.

All ANTLR grammars are subclasses of `Lexer`, `Parser`, or `TreeParser` and, since you should start thinking about this at the syntactic level, you will build a `Parser` subclass. After the class declaration, you will specify the rules in EBNF notation:

```
class ExprParser extends Parser;

expr:   mexpr ((PLUS|MINUS) mexpr)*
    ;

mexpr
```

```
    :   atom (STAR atom)*
    ;

atom:   INT
    |   LPAREN expr RPAREN
    ;
```

The lexer follows a similar pattern and only needs to define some operators and whitespace. Putting the lexer into the same file, say `expr.g`, is the easiest thing to do:

```
class ExprLexer extends Lexer;

options {
    k=2; // needed for newline junk
    charVocabulary='\u0000'..'\u007F'; // allow ascii
}

LPAREN: '(' ;
RPAREN: ')' ;
PLUS  : '+' ;
MINUS : '-' ;
STAR  : '*' ;
INT   : ('0'..'9')+ ;
WS    : ( ' '
        | '\r' '\n'
        | '\n'
        | '\t'
        )
        {$setType(Token.SKIP);}
      ;
```

To generate a program (in Java) from this grammar, `expr.g`, run ANTLR on it:

```
$ java antlr.Tool expr.g
ANTLR Parser Generator   Version 2.7.2   1989-2003 jGuru.com
$
```

What does ANTLR generate?

While not necessary for the completion of this tutorial, you may find it illuminating to see what ANTLR generates inside the recognizer files. ANTLR generates recognizers that mimic what you would build by hand--recursive-descent parsers; `yacc` and friends, on the other hand, generate tables full of integers as they simulate push-down-automata.

ANTLR will generate the following files:

```
ExprLexer.java
ExprParser.java
ExprParserTokenTypes.java
ExprParserTokenTypes.txt
```

If you take a look inside, for example, `ExprParser.java`, you will see a method for every rule defined in the parser grammar in `expr.g`. For example, the code for rules `mexpr` and `atom` look very much like this:

```
public void mexpr() {
  atom();
  while ( LA(1)==STAR ) {
    match(STAR);
    atom();
  }
}

public void atom() {
  switch ( LA(1) ) { // switch on lookahead token type
    case INT :
      match(INT);
      break;
    case LPAREN :
      match(LPAREN);
      expr();
      match(RPAREN);
      break;
    default :
      // error
  }
}
```

Notice that rule references are translated to method calls and token references are translated to `match(TOKEN)` calls. The only hard thing about building a parser from a grammar is computing the lookahead information.

The token types class defines all the constant token type numbers your lexer and parser use:

```
// $ANTLR 2.7.2: "expr.g" -> "ExprParser.java"$

public interface ExprParserTokenTypes {
        int EOF = 1;
        int NULL_TREE_LOOKAHEAD = 3;
        int PLUS = 4;
        int MINUS = 5;
        int STAR = 6;
        int INT = 7;
        int LPAREN = 8;
        int RPAREN = 9;
        int WS = 10;
}
```

Testing the lexer/parser

To actually use the resulting parser, in `ExprParser.java`, use a `main()` such as the following:

```
import antlr.*;
public class Main {
    public static void main(String[] args) throws Exception {
        ExprLexer lexer = new ExprLexer(System.in);
```

```
        ExprParser parser = new ExprParser(lexer);
        parser.expr();
    }
}
```

```
$ java Main
3+(4*5)
$
```

Or for bad input:

```
$ java Main
3++
line 1:3: unexpected token: +
$
```

or

```
$ java Main
3+(4
line 1:6: expecting RPAREN, found 'null'
$
```

Expression evaluation

To actually evaluate the expressions on the fly as the tokens come in, just add actions to the parser:

```
class ExprParser extends Parser;

expr returns [int value=0]
{int x;}
    :   value=mexpr
        ( PLUS x=mexpr  {value += x;}
        | MINUS x=mexpr {value -= x;}
        )*
    ;

mexpr returns [int value=0]
{int x;}
    :   value=atom ( STAR x=atom {value *= x;} )*
    ;

atom returns [int value=0]
    :   i:INT {value=Integer.parseInt(i.getText());}
    |   LPAREN value=expr RPAREN
    ;
```

The lexer is still the same, but add a print statement in the main program:

```
import antlr.*;

public class Main {
        public static void main(String[] args) throws Exception {
                ExprLexer lexer = new ExprLexer(System.in);
                ExprParser parser = new ExprParser(lexer);
                int x = parser.expr();
                System.out.println(x);
        }
```

```
}
```

Now, when you run the program you get the resulting computations:

```
$ java Main
3+4*5
23
$ java Main
(3+4)*5
35
$
```

How ANTLR translates actions

Actions are generally placed into the generated parser verbatim at the spot corresponding to the position in the grammar.

Rule return specifications like

```
mexpr returns [int value=0]
  : ...
  ;
```

are translated to

```
public int mexpr() {
  int value=0;
  ...
  return value;
}
```

If you were to add an incoming parameter specification, the args are also just copied to the method declaration:

```
mexpr[int x] returns [int value=0]
  : ... {value = x;}
  ;
```

yields

```
public int mexpr(int x) {
  int value=0;
  ...
  value = x;
  return value;
}
```

So, the full translation for the `mexpr` and `atom` rules looks like:

```
public int mexpr() {
  int value=0;
  int x; // local variable def from rule mexpr
  value = atom();
  while ( LA(1)==STAR ) {
    match(STAR);
    x = atom();
    value *= x;
```

```
  }
  return value;
}

public int atom() {
  int value=0;
  switch ( LA(1) ) { // switch on lookahead token type
    case INT :
      Token i = LT(1); // make label i point to next lookahead token object
      match(INT);
      value=Integer.parseInt(i.getText()); // compute int value of token
      break;
    case LPAREN :
      match(LPAREN);
      value = expr(); // return whatever expr() computes
      match(RPAREN);
      break;
    default :
      // error
  }
  return value;
}
```

Evaluation via AST intermediate form

So now you've seen basic syntax-directed translation/computation where the grammar/syntax dictates when to do an action. A more powerful strategy is to build an intermediate representation that holds all or most of the input symbols and has encoded, in the structure of the data, the relationship between those tokens. For example, input "3+4" is represented as an abstract syntax tree (AST) via:

```
  +
 / \
3   4
```

For this kind of tree, you will use a tree walker (generated by ANTLR from a tree grammar) to compute the same values as before, but using a different strategy.

The utility of ASTs becomes clear when you must do multiple walks over the tree to figure out what to compute or to do tree rewrites, morphing the tree towards another language.

AST construction

To get ANTLR to generate a useful AST is pretty simple for many grammars. In our case, turn on the `buildAST` option and then add a few suffix operators to tell ANTLR what tokens should be subtree roots.

```
class ExprParser extends Parser;

options {
```

```
        buildAST=true;
}

expr:   mexpr ((PLUS^|MINUS^) mexpr)*
    ;

mexpr
    :   atom (STAR^ atom)*
    ;

atom:   INT
    |   LPAREN! expr RPAREN!
    ;
```

Again, the lexer doesn't change. The main program asks for the resulting tree and prints it out:

```
import antlr.*;
import antlr.collections.*;

public class Main {
    public static void main(String[] args) throws Exception {
        ExprLexer lexer = new ExprLexer(System.in);
        ExprParser parser = new ExprParser(lexer);
        parser.expr();
        AST t = parser.getAST();
        System.out.println(t.toStringTree());
    }
}
```

```
$ java Main
3+4
 ( + 3 4 )
$ java Main
3+4*5
 ( + 3 ( * 4 5 ) )
$ java Main
(3+4)*5
 ( * ( + 3 4 ) 5 )
$
```

AST parsing and evaluation

The trees built by the above parser are pretty simple. A single rule in the tree parser suffices.

```
class ExprTreeParser extends TreeParser;

options {
    importVocab=ExprParser;
}

expr returns [int r=0]
{ int a,b; }
    :   #(PLUS  a=expr b=expr)  {r = a+b;}
    |   #(MINUS a=expr b=expr)  {r = a-b;}
    |   #(STAR  a=expr b=expr)  {r = a*b;}
    |   i:INT                   {r = (int)Integer.parseInt(i.getText());}
    ;
```

The main program is modified to use the new tree parser for evaluation:

```
import antlr.*;
import antlr.collections.*;

public class Main {
    public static void main(String[] args) throws Exception {
        ExprLexer lexer = new ExprLexer(System.in);
        ExprParser parser = new ExprParser(lexer);
        parser.expr();
        AST t = parser.getAST();
        System.out.println(t.toStringTree());
        ExprTreeParser treeParser = new ExprTreeParser();
        int x = treeParser.expr(t);
        System.out.println(x);
    }
}
```

Now you get the tree structure and the resulting value.

```
$ java Main
3+4
 ( + 3 4 )
7
$ java Main
3+(4*5)+10
 ( + ( + 3 ( * 4 5 ) ) 10 )
33
$
```