Introduction to the ANTLR4 Parser Generator

Robert Soulé

Version of September 15, 2015

Abstract

ANTLR4 is a parser generator developed and maintained by Terence Parr at the University of San Francisco. This document describes a subset of ANTLR4, including the features needed for a Compilers course taught by Robert Soulé at USI. The full, official documentation for ANTLR4 is here:

`http://www.antlr.org`

This document frequently refers to the Dragon book:

Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*, Second Edition. Addison Wesley, 2007.

1 Getting Started

Figure 1 shows the ANTLR4 implementation of a simple translator from infix notation to postfix notation. For now, we will not explain how it works; instead, we will just focus on learning how to compile and run it. The code corresponds to the running example in Chapter 2 of the Dragon book, in particular, Figure 2.23.

```
grammar Simple;

// productions for syntax analysis
program returns [String s]: e=expr EOF {$s = $e.s; } ;
expr returns [String s]: t=term r=rest {$s = $t.s + $r.s;} ;
rest returns [String s]
 : PLUS t=term r=rest                  {$s = $t.s + "+" + $r.s;}
 | MINUS t=term r=rest                 {$s = $t.s + "-" + $r.s;}
 | /* empty */                         {$s = "";} ;
term returns [String s] : DIGIT        {$s = $DIGIT.text;};

// productions for lexical analysis
PLUS : '+' ;
MINUS : '-' ;
DIGIT : [0-9];
```

Figure 1: Source code for `Simple.g4`

Figure 2 shows the Java code for using the translator from Figure 1. The input comes from the command line argument `args[0]`, and is assumed to be an infix expression. The code `new SimpleLexer(..)` creates an instance of the lexer. The code `new SimpleParser(..)` creates an instance of the parser.

Calling the `program` method runs the translator starting at nonterminal `program`. The result is the string stored in attributes `s` of the return from the `program` call.

```
import java.io.*;
import org.antlr.v4.runtime.*;

public class SimpleMain {
  public static void main(final String[] args) throws IOException {
    String printSource = null, printSymTab = null,
      printIR = null, printAsm = null;
    SimpleLexer lexer = new SimpleLexer(new ANTLRInputStream(args[0]));
    SimpleParser parser = new SimpleParser(new CommonTokenStream(lexer));
    String postfix = parser.program().s;
    System.out.println(postfix);
  }
}
```

Figure 2: Source code for `SimpleMain.java`

Make sure you have the JDK for Java 8 in your `PATH`, including `javac` (the Java compiler) and `java` (the Java virtual machine).

You will also need to download the file `antlr-4.5.1-complete.jar` from here: `http://www.antlr.org/download/antlr-4.5.1-complete.jar`. Now, you can try out the translator as follows:

- Install `antlr-4.5.1-complete.jar` in `/usr/local/lib`. Use ANTLR4 on `Simple.g4`, which will generate the new files: `SimpleLexer.java`, `SimpleParser.java`, `Simple.tokens`, and `SimpleLexer.tokens`.

  ```
  java -jar /usr/local/lib/antlr-4.5.1-complete.jar
  -no-listener Simple.g4
  ```

- Use the Java compiler to compile the `.java` files to `.class` files:

  ```
  javac -g -cp /usr/local/lib/antlr-4.5.1-complete.jar
  *.java
  ```

- Run the translator on the infix expression `9-5+2`. This should print the postfix expression `95-2+`:

module → *grammarDecl productions*
grammarDecl → `grammar` **id** `;`
productions → *productions production* | ε

Figure 3: ANTLR4 top-level grammar

```
java -ea -cp .:/usr/local/lib/antlr-4.5.1-complete.jar
SimpleMain '9-5+2'
```

Note that the example only works with single digits and no whitespace, because the grammar is so simple.

2 Structure of an ANTLR4 File

Figure 3 shows a simplified "grammar for the grammar": the syntax in which you can write a ANTLR4 file. The notation follows the conventions in the Dragon book: the arrow '→' separates the head and body of a production; nonterminals are in *italics*; and tokens are in **bold**. The vertical bar | separates alternatives, and the letter ε denotes an empty sequence.

The start symbol of the ANTLR4 grammar is *grammarDecl*. A *grammarDecl* specifies the grammar name. ANTLR4 is case-sensitive and whitespace-insensitive. It supports single-line comments, which start with `//` and end at the next newline, and multi-line comments, which start with `/*` and end with `*/`.

With ANTLR4, productions for lexical analysis and productions for syntax analysis are written in a single file. However, it is still useful to distinguish the two kinds of productions. Lexer rule names (i.e, terminals) are written in capital letters with underscores, such as `STRING_LIT`. Parser rule names (i.e., non-terminals) are written in in camelCase, such as `addExpr`. These requirements are illustrated in Figure 1. Furthermore, we will make use of different ANTLR4 features for lexical and syntax analysis, as described in the following two sections.

3 Lexical Analysis

Figure 4 shows the rest of the simplified "grammar for the grammar", elaborating on the *production* nonterminal from Figure 3.

Consider the example lexical production:

```
DIGIT : [0-9];
```

In this production *id* is `DIGIT`, *return* is ε, and the choice contains a single sequence with a single item `[0-9]`. This item is a **charClass**, denoting any digit between `0` and `9`. In other words, when the lexical analysis finds a digit from `0-9`, it recognizes a `DIGIT` token. In practice, you would usually also have whitespace after the token, as described below.

production → **id** *return* `:` *choice* `;`
return → `returns [` *attrList* `]` | ε
attrList → *attr attrListTail*
attr → *type* **id**
attrListTail → `,` *attr attrListTail* | ε
type → **id**
choice → *choice* `|` *sequence action*
| *sequence action*
sequence → *sequence item* | ε
item → *primary suffix*
suffix → `?` | `*` | `+` | ε
primary → **id** | *literal*
literal → `_` | **stringLit** | **charLit** | **charClass**
action → { **javaCode** }

Figure 4: ANTLR4 production grammar

For the remaining examples, we will show assorted productions typical for many programming languages, which go beyond the `Simple.g4` grammar in Figure 1.

Most realistic grammars will use a lexical production to define a whitespace token:

```
WS : ( [ \r\t\n] | NEWLINE | COMMENT )+ -> skip ;
```

As defined, whitespace consists of either one of the characters space, tab, form-feed, and newline, or a comment. The *suffix* operators `?`, `*`, and `+` indicate the item is optional, repeated zero or more times, or repeated one or more times, respectively. In the `WS` production, `+` means there can be an arbitrary amount of whitespace.

Because ANTLR4 is an LL(*) parser, it is slightly more powerful than predicated-LL(k) from the lectures, and can do some actual parsing in the lexer. One of the common uses of this feature is handling whitespace. Unlike other tools, ANTLR4 does not require you to specify whitespace explicitly wherever it should match. You can simply specify the `WS` rule once in the grammar. Note that the example `WS` rule uses a special command, `-> skip`. The `-> skip` command tells the lexer not to produce tokens for whitespace.

4 Syntax Analysis

For consistency with the Dragon book, we will restrict ourselves to use ANTLR4 with simple context-free grammars. In particular, we will avoid predicate *prefix* operators and any *suffix* operators (`?`, `*`,

+) in the productions for syntax analysis. The example grammar in Figure 1 illustrates these restrictions. By following these conventions, the skills you learn building a parser for ANTLR4 are easier to transfer to other parser generators.

ANTLR4 generates top-down, LL(*) parsers. These are most closely related to the LL parsers in the Dragon book, but with some important differences. The parsers generated by ANTLR4 allow for arbitrary lookahead, and ANTLR4 grammars may be ambiguous (i.e., the same string may be recognized in multiple ways). To resolve ambiguities, ANTLR4 uses the order of the productions in the input grammar file. In other words, ANTLR4 uses the first production that matches the input, instead of reporting an ambiguity. For example, consider the following production:

```
relOp :  '<=' | '<' | '>=' | '>' ;
```

The choice "<=" must be defined before the choice `'<'`, otherwise, ANTLR4 will never recognize `'<='`. This production also illustrates the use of **stringLit**s.

This grammar shows **javaCode** as a token; it can contain arbitrary Java code. We will see more details on the syntax for a *production* in later sections.

The Dragon book describes various other techniques for writing top-down parsers, which can be directly applied to ANTLR4. You can implement repetition by recursive grammar rules; for instance, `rest` in Figure 1 is defined recursively. You can make a production optional by supplying an empty alternative; for instance, Figure 1 contains such a case, marked with the comment `/*empty*/`. Finally, you can avoid left-recursion by introducing `rest` productions to your grammar; the example in Figure 1 is the result of this technique.

5 Abstract Syntax Trees

The following example defines a syntax-analysis production with an action for constructing a new abstract syntax tree (AST) node:

```
whileStmt returns [WhileStmt v] :
WHILE c=expr b=blockStmt
{$v = new WhileStmt(
  new Location($ctx), $c.v, $b.v);} ;
```

This could parse `while(i<n) { print(" "); i++; }`, where `c` binds the AST of expression `i<n` and `b` binds the AST for statement `{ print(" "); i++; }`.

The return type from an ANTLR4 production is always a ParseTree object. However, the `returns` statement adds an extra attribute(s) to the ParseTree, which we can be uses as a custom return value. In the example above, the additional attribute has *type* `WhileStmt`, unlike the productions we have seen before, in which the attributes had *type* `String`. In our compiler construction course, you will have to define classes for all AST nodes, such as `WhileStmt`, yourself. ANTLR4 also has a feature for auto-generating generic AST nodes, but we will not use this feature (to get strongly-typed ASTs, to make your skills easier to transfer across parser generators, and for consistency with the Dragon book).

The return value of a production is a natural way to implement synthesized attributes, which are attributes computed by a syntax-directed translator that propagate from child productions to parent productions. In ANTLR4, the child production uses an action to assign the attribute to a special variable (`v` in the above example), and the parent picks up the attribute using the notation **`$ id . id`** (e.g., `$b.v`)

Besides synthesized attributes, the other kind of attributes in a syntax-directed translator is inherited attributes, which propagate from parent productions to child productions. We cannot implement inherited attributes directly in the ANTLR4 grammar. Instead, we need to first construct an AST, and then, we can implement a syntax-directed translation scheme by a tree traversal. For example, after applying the technique for dealing with left-recursion from the Dragon book, we need inherited attributes to construct a left-associative AST. Figure 5.13 in the Dragon book shows such a translation scheme. We would implement it with a separate tree traversal, not during parsing but after parsing.

6 Source Locations

A source location consists of a file name, line number, and column number in the file that is being parsed. Source locations are important for error reporting, both during parsing and later, for instance, during type checking. In ANTLR4, the file name comes from a parameter to the parser of type ParserRuleContext. Semantic actions in productions have access to a special variable `$ctx`, which holds a reference to the tokens in the production. The file name, line number, and column number can be accessed from the ParserRuleContext.

```
Token firstToken = ctx.getStart();
String file = firstToken.getTokenSource().getSourceName();
```

```
int line = firstToken.getLine();
int column = firstToken.getCharPositionInLine() + 1;
```

It is common practice to store locations with each AST node. For example:

```
{$v = new WhileStmt(
new Location($ctx), $c.v, $b.v);} ;
```

Later in the compiler, the location can be used to report an error message, for example, by doing:

```
void print(Location loc, String msg) {
  System.err.println(loc + ": " + msg + ".");
}
```