Extracted from:

The Definitive ANTLR 4 Reference

The Pragmatic Bookshelf

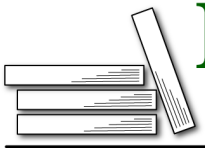
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

The Definitive ANTLR 4 Reference

Terence Parr

Edited by Susannah Davidson Pfalzer



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-93435-699-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—September 19, 2012

3.3 Building a Translator with a Listener

Imagine your boss assigns you to build a tool that generates a Java interface file from the methods in a Java class definition. Panic ensues if you're a junior programmer. As an experienced Java developer, you might suggest using the Java reflection API or the javap tool to extract method signatures. If your Java tool building kung fu is very strong you might even try using a bytecode library such as ASM.[2] Then your boss says, "Oh yeah. Preserve whitespace and comments within the bounds of the method signature." There's no way around it now: We have to parse Java source code. For example, we'd like to read in Java code like this:

tour/Demo.java

```
import java.util.List;
import java.util.Map;
public class Demo {
        void f(int x, String y) { }
        int[ ] g(/*no args*/) { return null; }
        List<Map<String, Integer>>[] h() { return null; }
}
```

and generate an interface with the method signatures, preserving the whitespace and comments:

tour/IDemo.java

```
interface IDemo {
        void f(int x, String y);
        int[ ] g(/*no args*/);
        List<Map<String, Integer>>[] h();
}
```

Believe it or not, we're going to solve the core of this problem in about 15 lines of code by listening to "events" fired from a Java parse tree walker. The Java parse tree will come from a parser generated from an existing Java grammar included in the source code for this book. We'll derive the name of the generated interface from the class name and grab method signatures (return type, method name, and argument list) from method definitions. For a similar but more thoroughly explained example, see Section 7.3, *Generating a Call Graph*, on page ?.

The key "interface" between the grammar and our listener object is called JavaListener and ANTLR automatically generates it for us. It defines all of the methods we can trigger via class ParseTreeWalker (from ANTLR's runtime) as it traverses the parse tree. In our case, we need to respond to three events:

2. http://asm.ow2.org

When the walker enters and exits a class definition and when it encounters a method definition. Here are the relevant methods from the generated listener interface:

```
public interface JavaListener extends ParseTreeListener<Token> {
    void enterClassDeclaration(JavaParser.ClassDeclarationContext ctx);
    void exitClassDeclaration(JavaParser.ClassDeclarationContext ctx);
    void enterMethodDeclaration(JavaParser.MethodDeclarationContext ctx);
    ...
}
```

The biggest difference between the listener and visitor mechanisms is that listener methods are called independently by an ANTLR-provided walker object, whereas visitor methods must walk their children with explicit visit calls. Forgetting to invoke visitor methods on a node's children, means those subtrees don't get visited.

To build our listener implementation, we need to know what rules classDeclaration and methodDeclaration look like because listener methods have to grab phrase elements matched by the rules. File Java.g4 is a complete grammar for Java, but here are the two methods we need to look at for this problem:

tour/Java.g4

```
classDeclaration
    :   'class' Identifier typeParameters? ('extends' type)?
        ('implements' typeList)?
        classBody
    ;
```

tour/Java.g4

```
methodDeclaration
    :   type Identifier formalParameters ('[' ']')* methodDeclarationRest
    |   'void' Identifier formalParameters methodDeclarationRest
    ;
```

So that we don't have to implement all 200 or so interface methods, ANTLR generates a blank default implementation called JavaBaseListener. Our interface extractor can then subclass JavaBaseListener and override the methods of interest.

Our basic strategy will be to print out the interface header when we see the start of a class definition. Then, we'll print a terminating } at the end of the class definition. Upon each method definition, we'll spit out its signature. Here's the complete implementation:

tour/ExtractInterfaceListener.java

```
import org.antlr.v4.runtime.TokenStream;
import org.antlr.v4.runtime.misc.Interval;

public class ExtractInterfaceListener extends JavaBaseListener {
```

```
    JavaParser parser;
    public ExtractInterfaceListener(JavaParser parser) { this.parser = parser; }

    /** Listen to matches of classDeclaration */
    @Override
    public void enterClassDeclaration(JavaParser.ClassDeclarationContext ctx) {
        System.out.println("interface I"+ctx.Identifier()+" {");
    }

    @Override
    public void exitClassDeclaration(JavaParser.ClassDeclarationContext ctx) {
        System.out.println("}");
    }

    /** Listen to matches of methodDeclaration */
    @Override
    public void enterMethodDeclaration(JavaParser.MethodDeclarationContext ctx) {
        TokenStream tokens = parser.getTokenStream(); // need parser to get tokens
        String type = "void";
        if ( ctx.type()!=null ) {
            type = tokens.getText(ctx.type().getSourceInterval());
        }
        String args = tokens.getText(ctx.formalParameters());
        System.out.println("\t"+type+" "+ctx.Identifier()+args+";");
    }
}
```

To fire this up, we need a main program, which looks almost the same as the others in this chapter. Our application code starts after we've launched the parser:

tour/ExtractInterfaceTool.java

```
JavaLexer lexer = new JavaLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
JavaParser parser = new JavaParser(tokens);
ParserRuleContext<Token> tree = parser.compilationUnit(); // parse

ParseTreeWalker walker = new ParseTreeWalker(); // create standard walker
ExtractInterfaceListener extractor = new ExtractInterfaceListener(parser);
walker.walk(extractor, tree); // initiate walk of tree with listener
```

We also need to add `import org.antlr.v4.runtime.tree.*;` at the top of the file.

Given grammar Java.g4 and our `main()` in `ExtractInterfaceTool`, here's the complete build and test sequence:

```
⇒ $ antlr4 Java.g4
⇒ $ ls Java*.java ExtractInterface*.java
❮ ExtractInterfaceListener.java  JavaBaseListener.java   JavaListener.java
  ExtractInterfaceTool.java      JavaLexer.java          JavaParser.java
⇒ $ javac Java*.java Extract*.java
```

```
⇒ $ java ExtractInterfaceTool Demo.java
❮ interface IDemo {
         void f(int x, String y);
         int[ ] g(/*no args*/);
         List<Map<String, Integer>>[] h();
}
```

This implementation isn't quite complete because it doesn't include in the interface file the import statements for the types referenced by the interface methods such as List. As an exercise, try handling the imports. It should convince you that it's easy to build these kinds of extractors or translators using a listener. We don't even need to know what the importDeclaration rule looks like because enterImportDeclaration() should simply print the text matched by the entire rule: parser.getTokenStream().getText(ctx).

The visitor and listener mechanisms work very well and promote the separation of concerns between parsing and parser application. Sometimes, though, we need extra control and flexibility.

3.4 Making Things Happen During the Parse

Listeners and visitors are great because they keep application-specific code out of grammars, making grammars easier to read and preventing them from getting entangled with a particular application. For the ultimate flexibility and control, however, we can directly embed code snippets (actions) within grammars. These actions are copied into the recursive-descent parser code ANTLR generates. In this section, we'll implement a simple program that reads in rows of data and prints out the values found in a specific column. After that, we'll see how to make special actions, called semantic predicates, dynamically turn parts of a grammar on and off.

Embedding Arbitrary Actions in a Grammar

We can compute values or print things out on-the-fly during parsing if we don't want the overhead of building a parse tree. On the other hand, it means embedding arbitrary code within the expression grammar, which is harder; we have to understand the effect of the actions on the parser and where to position those actions.

To demonstrate actions embedded in a grammar, let's build a program that prints out a specific column from rows of data. This comes up all the time for me because people send me text files from which I need to grab, say, the name or email column. For our purposes, let's use the following data:

tour/t.rows

```
parrt   Terence Parr    101
```

```
tombu   Tom Burns       020
bke     Kevin Edgar     008
```

The columns are tab-delimited and each row ends with a newline character. To match this kind of input is pretty simple grammatically:

```
file : (row '\n')+ ;
row  : STUFF+ ;
```

It gets mucked up, though, when we add actions. We need to create a constructor so that we can pass in the column number we want (counting from 1) and we need an action inside the (...)+ loop in rule row:

tour/Rows.g4

```
grammar Rows;

@parser::members {
    int col;
    public RowsParser(TokenStream input, int col) {
        this(input);
        this.col = col;
    }
}

file: (row '\n')+ ;

row
locals [int i=0]
    : (   STUFF
          {
          $i++;
          if ( $i == col ) System.out.println($STUFF.text);
          }
      )+
    ;

TAB   :  '\t' -> skip ;   // match but don't pass to the parser
NL    :  '\n' ;           // match and pass to the parser
STUFF:  ~[\t\n]+ ;        // match any chars except tab, newline
```

The STUFF lexical rule matches anything that's not a tab or newline, which means we can have space characters in a column.

A suitable main program should be looking pretty familiar by now. The only thing different here is that we're passing in a column number to the parser using a custom constructor:

tour/Col.java

```
RowsLexer lexer = new RowsLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
int col = Integer.valueOf(args[0]);
```

```
RowsParser parser = new RowsParser(tokens, col); // pass column number!
parser.file(); // parse
```

There are a lot of details in there that we'll explore in Chapter 9, *Attributes and Actions*, on page ?. For now, actions are code snippets surrounded by curly braces. The members action injects that code into the member area of the generated parser class. The action within rule row accesses $i, the local variable defined with the locals clause. It also uses $STUFF.text to get the text for the most recently matched STUFF token.

Here's the build and test sequence, one test per column:

```
⇒ $ antlr4 -no-listener Rows.g4  # don't need the listener
⇒ $ javac Rows*.java Col.java
⇒ $ java Col 1 < t.rows          # print out column 1, reading from file t.rows
❮ parrt
  tombu
  bke
⇒ $ java Col 2 < t.rows
❮ Terence Parr
  Tom Burns
  Kevin Edgar
⇒ $ java Col 3 < t.rows
❮ 101
  020
  008
```

These actions extract and print values matched by the parser, but they don't alter the parse itself. Actions can also finesse how the parser recognizes input phrases. In the next section, we'll take the concept of embedded actions one step further.

Altering the Parse with Semantic Predicates

Until we get to Chapter 10, *Altering the Parse with Semantic Predicates*, on page ?, we can demonstrate the power of semantic predicates with a simple example. Let's look at a grammar that reads in sequences of integers. The trick is that part of the input specifies how many integers to group together. We don't know until runtime how many integers to match. Here's a sample input file:

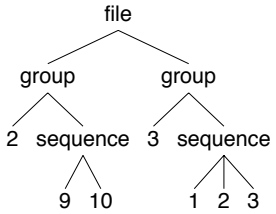
tour/t.data

```
2 9 10 3 1 2 3
```

The first number says to match the two subsequent numbers, 9 and 10. The 3 following the 10 says to match 3 more as a sequence. Our goal is a grammar called Data that groups 9 and 10 together then 1, 2, and 3 like this:

```
⇛ $ antlr4 -no-listener Data.g4
⇛ $ javac Data*.java
⇛ $ grun Data file -tree t.data
❮ (file (group 2 (sequence 9 10)) (group 3 (sequence 1 2 3)))
```

The parse tree clearly identifies the groups:

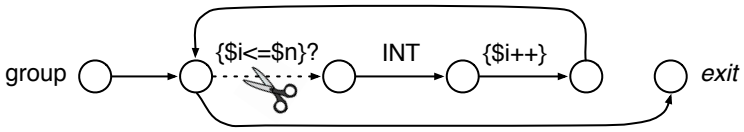


The key in the following Data grammar is semantic predicate {$i<=$n}?. That predicate evaluates to true until we surpass the number of integers requested by the sequence rule parameter n. False predicates make the associated alternative "disappear" from the grammar and, hence, from the generated parser. In this case, a false predicate makes the (...)* loop terminate and return from rule sequence.

tour/Data.g4

```
grammar Data;

file : group+ ;

group: INT sequence[$INT.int] ;

sequence[int n]
locals [int i = 1;]
     : ( {$i<=$n}? INT {$i++;} )* // match n integers
     ;

INT :   [0-9]+ ;             // match integers
WS  :   [ \t\n\r]+ -> skip ; // toss out all whitespace
```

Visually, the internal grammar representation of rule group used by the parser looks something like this:



The scissors and dashed line indicate that the predicate can snip that path, leaving the parser with only one choice: the path to the exit.

Most of the time we won't need such micromanagement, but it's nice to know we've got a weapon for handling pathological parsing problems.

During our tour so far, we've focused on parsing features, but there's lots of interesting stuff going on at the lexical level. Let's take a look.