CHAPTER 4

A Quick Tour

So far, we have learned how to install ANTLR and have looked at the key processes, terminology, and building blocks needed to build a language application. In this chapter, we're going to take a whirlwind tour of ANTLR by racing through a number of examples that illustrate its capabilities. We'll be glossing over a lot of details in the interest of brevity, so don't worry if things aren't crystal clear. The goal is just to get a feel for what you can do with ANTLR. We'll look underneath the hood starting in Chapter 5, *Designing Grammars*, on page 57. For those with experience using previous versions of ANTLR, this chapter is a great way to retool.

This chapter is broken down into four broad topics that nicely illustrate the feature set. It's a good idea to download the code[1] for this book (or follow the links in the ebook version) and work through the examples as we go along. That way, you'll get used to working with grammar files and building ANTLR applications. Keep in mind that many of the code snippets you see interspersed in the text aren't complete files so that we can focus on the interesting bits.

First, we're going to work with a grammar for a simple arithmetic expression language. We'll test it initially using ANTLR's built-in test rig and then learn more about the boilerplate main program that launches parsers shown in Section 3.3, *Integrating a Generated Parser into a Java Program*, on page 26. Then, we'll look at a nontrivial parse tree for the expression grammar. (Recall that a parse tree records how a parser matches an input phrase.) For dealing with very large grammars, we'll see how to split a grammar into manageable chunks using grammar imports. Next, we'll check out how ANTLR-generated parsers respond to invalid input.

1. http://pragprog.com/titles/tpantlr2/source_code

Second, after looking at the parser for arithmetic expressions, we'll use a visitor pattern to build a calculator that walks expression grammar parse trees. ANTLR parsers automatically generate visitor interfaces and blank method implementations so we can get started painlessly.

Third, we'll build a translator that reads in a Java class definition and spits out a Java interface derived from the methods in that class. Our implementation will use the tree listener mechanism that ANTLR also generates automatically.

Fourth, we'll learn how to embed *actions* (arbitrary code) directly in the grammar. Most of the time, we can build language applications with visitors or listeners, but for the ultimate flexibility, ANTLR allows us to inject our own application-specific code into the generated parser. These actions execute during the parse and can collect information or generate output like any other arbitrary code snippets. In conjunction with *semantic predicates* (Boolean expressions), we can even make parts of our grammar disappear at runtime! For example, we might want to turn the `enum` keyword on and off in a Java grammar to parse different versions of the language. Without semantic predicates, we'd need two different versions of the grammar.

Finally, we'll zoom in on a few ANTLR features at the lexical (token) level. We'll see how ANTLR deals with input files that contain more than one language. Then we'll look at the awesome `TokenStreamRewriter` class that lets us tweak, mangle, or otherwise manipulate token streams, all without disturbing the original input stream. Finally, we'll revisit our interface generator example to learn how ANTLR can ignore whitespace and comments during Java parsing but retain them for later processing.

Let's begin our tour by getting acquainted with ANTLR grammar notation. Make sure you have the `antlr4` and `grun` aliases or scripts defined, as explained in Section 1.2, *Executing ANTLR and Testing Recognizers*, on page 6.

4.1 Matching an Arithmetic Expression Language

For our first grammar, we're going to build a simple calculator. Doing something with expressions makes sense because they're so common. To keep things simple, we'll allow only the basic arithmetic operators (add, subtract, multiply, and divide), parenthesized expressions, integer numbers, and variables. We'll also restrict ourselves to integers instead of allowing floating-point numbers.

Here's some sample input that illustrates all language features:

tour/t.expr
```
193
a = 5
b = 6
a+b*2
(1+2)*3
```

In English, a *program* in our expression language is a sequence of statements terminated by newlines. A statement is either an expression, an assignment, or a blank line. Here's an ANTLR grammar that'll parse those statements and expressions for us:

tour/Expr.g4
```
grammar Expr;

/** The start rule; begin parsing here. */
prog:   stat+ ;

stat:   expr NEWLINE
    |   ID '=' expr NEWLINE
    |   NEWLINE
    ;

expr:   expr ('*'|'/') expr
    |   expr ('+'|'-') expr
    |   INT
    |   ID
    |   '(' expr ')'
    ;

ID  :   [a-zA-Z]+ ;      // match identifiers
INT :   [0-9]+ ;         // match integers
NEWLINE:'\r'? '\n' ;     // return newlines to parser (is end-statement signal)
WS  :   [ \t]+ -> skip ; // toss out whitespace
```

Without going into too much detail, let's look at some of the key elements of ANTLR's grammar notation.

- Grammars consist of a set of rules that describe language syntax. There are rules for syntactic structure like stat and expr as well as rules for vocabulary symbols (tokens) such as identifiers and integers.
- Rules starting with a lowercase letter comprise the parser rules.
- Rules starting with an uppercase letter comprise the lexical (token) rules.
- We separate the alternatives of a rule with the | operator, and we can group symbols with parentheses into *subrules*. For example, subrule ('*'|'/') matches either a multiplication symbol or a division symbol.

We'll tackle all of this stuff in detail when we get to Chapter 5, *Designing Grammars*, on page 57.

One of ANTLR v4's most significant new features is its ability to handle (most kinds of) left-recursive rules. A left-recursive rule is one that invokes itself at the start of an alternative. For example, in this grammar, rule expr has alternatives on lines 11 and 12 that recursively invoke expr on the left edge. Specifying arithmetic expression notation this way is dramatically easier than what we'd need for the typical top-down parser strategy. In that strategy, we'd need multiple rules, one for each operator precedence level. For more on this feature, see Section 5.4, *Dealing with Precedence, Left Recursion, and Associativity*, on page 69.

The notation for the token definitions should be familiar to those with regular expression experience. We'll look at lots of lexical (token) rules in Chapter 6, *Exploring Some Real Grammars*, on page 83. The only unusual syntax is the -> skip operation on the WS whitespace rule. It's a directive that tells the lexer to match but throw out whitespace. (Every possible input character must be matched by at least one lexical rule.) We avoid tying the grammar to a specific target language by using formal ANTLR notation instead of an arbitrary code snippet in the grammar that tells the lexer to skip.

OK, let's take grammar Expr out for a joy ride. Download it either by clicking the tour/Expr.g4 link on the previous code listing, if you're viewing the ebook version, or by cutting and pasting the grammar into a file called Expr.g4.

The easiest way to test grammars is with the built-in TestRig, which we can access using alias grun. For example, here is the build and test sequence on a Unix box:

```
$ antlr4 Expr.g4
$ ls Expr*.java
ExprBaseListener.java   ExprListener.java
ExprLexer.java          ExprParser.java
$ javac Expr*.java
$ grun Expr prog -gui t.expr # launches org.antlr.v4.runtime.misc.TestRig
```

Because of the -gui option, the test rig pops up a window showing the parse tree, as shown in Figure 2, *Window showing the parse tree*, on page 35.

The parse tree is analogous to the function call tree our parser would trace as it recognizes input. (ANTLR generates a function for each rule.)

It's OK to develop and test grammars using the test rig, but ultimately we'll need to integrate our ANTLR-generated parser into an application. The main

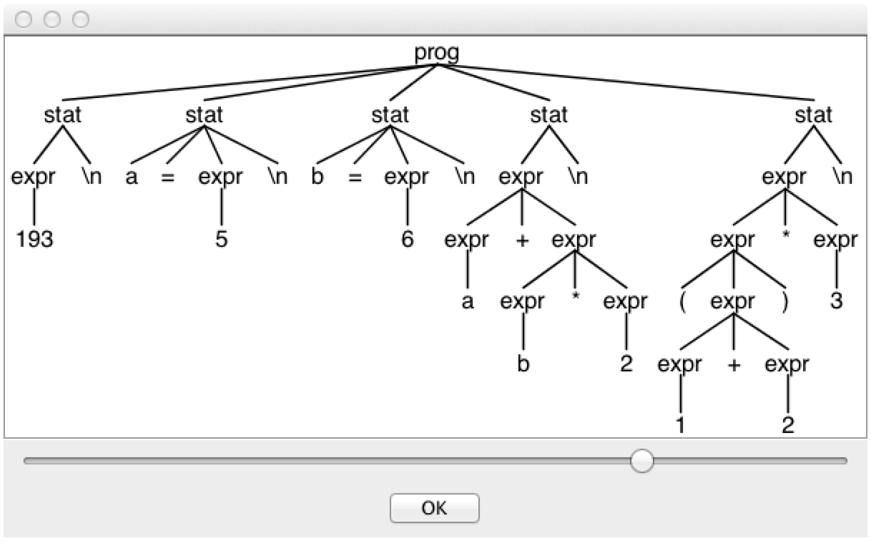


Figure 2—Window showing the parse tree

program given below shows the code necessary to create all necessary objects and launch our expression language parser starting at rule prog:

tour/ExprJoyRide.java

```
Line 1 import org.antlr.v4.runtime.*;
     - import org.antlr.v4.runtime.tree.*;
     - import java.io.FileInputStream;
     - import java.io.InputStream;
     5 public class ExprJoyRide {
     -     public static void main(String[] args) throws Exception {
     -         String inputFile = null;
     -         if ( args.length>0 ) inputFile = args[0];
     -         InputStream is = System.in;
    10         if ( inputFile!=null ) is = new FileInputStream(inputFile);
     -         ANTLRInputStream input = new ANTLRInputStream(is);
     -         ExprLexer lexer = new ExprLexer(input);
     -         CommonTokenStream tokens = new CommonTokenStream(lexer);
     -         ExprParser parser = new ExprParser(tokens);
    15         ParseTree tree = parser.prog(); // parse; start at prog
     -         System.out.println(tree.toStringTree(parser)); // print tree as text
     -     }
     - }
```

Lines 7..11 create an input stream of characters for the lexer. Lines 12..14 create the lexer and parser objects and a token stream "pipe" between them. Line 15 actually launches the parser. (Calling a rule method is like invoking that rule; we can call any parser rule method we want.) Finally, line 16 prints out the parse tree returned from the rule method prog() in text form.

Here is how to build the test program and run it on input file t.expr:

```
⇒ $ javac ExprJoyRide.java Expr*.java
⇒ $ java ExprJoyRide t.expr
❮ (prog
    (stat (expr 193) \n)
    (stat a = (expr 5) \n)
    (stat b = (expr 6) \n)
    (stat (expr (expr a) + (expr (expr b) * (expr 2))) \n)
    (stat (expr (expr ( (expr (expr 1) + (expr 2)) )) * (expr 3)) \n)
  )
```

The (slightly cleaned up) text representation of the parse tree is not as easy to read as the visual representation, but it's useful for functional testing.

This expression grammar is pretty small, but grammars can run into the thousands of lines. In the next section, we'll learn how to keep such large grammars manageable.

Importing Grammars

It's a good idea to break up very large grammars into logical chunks, just like we do with software. One way to do that is to split a grammar into parser and lexer grammars. That's not a bad idea because there's a surprising amount of overlap between different languages lexically. For example, identifiers and numbers are usually the same across languages. Factoring out lexical rules into a "module" means we can use it for different parser grammars. Here's a lexer grammar containing all of the lexical rules:

tour/CommonLexerRules.g4

```
lexer grammar CommonLexerRules; // note "lexer grammar"

ID  :   [a-zA-Z]+ ;      // match identifiers
INT :   [0-9]+ ;         // match integers
NEWLINE:'\r'? '\n' ;     // return newlines to parser (end-statement signal)
WS  :   [ \t]+ -> skip ; // toss out whitespace
```

Now we can replace the lexical rules from the original grammar with an import statement.

tour/LibExpr.g4

```
grammar LibExpr;         // Rename to distinguish from original
import CommonLexerRules; // includes all rules from CommonLexerRules.g4
/** The start rule; begin parsing here. */
prog:   stat+ ;
```

```
stat:   expr NEWLINE
    |   ID '=' expr NEWLINE
    |   NEWLINE
    ;

expr:   expr ('*'|'/') expr
    |   expr ('+'|'-') expr
    |   INT
    |   ID
    |   '(' expr ')'
    ;
```

The build and test sequence is the same as it was without the import. We do not run ANTLR on the imported grammar itself.

```
⇒ $ antlr4 LibExpr.g4 # automatically pulls in CommonLexerRules.g4
⇒ $ ls Lib*.java
❮ LibExprBaseListener.java        LibExprListener.java
  LibExprLexer.java               LibExprParser.java
⇒ $ javac LibExpr*.java
⇒ $ grun LibExpr prog -tree
⇒ 3+4
⇒ EOF
❮ (prog (stat (expr (expr 3) + (expr 4)) \n))
```

So far, we've assumed valid input, but error handling is an important part of almost all language applications. Let's see what ANTLR does with erroneous input.

Handling Erroneous Input

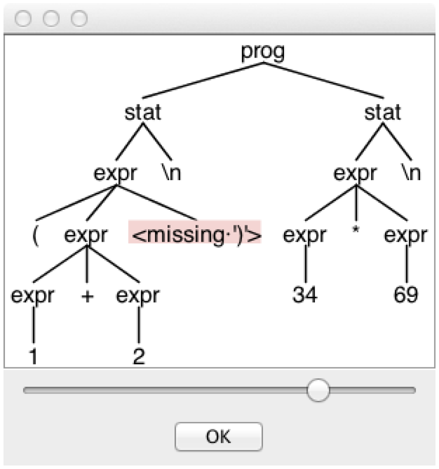
ANTLR parsers automatically report and recover from syntax errors. For example, if we forget a closing parenthesis in an expression, the parser automatically emits an error message.

```
⇒ $ java ExprJoyRide
⇒ (1+2
⇒ 3
⇒ EOF
❮ line 1:4 mismatched input '\n' expecting {')', '+', '*', '-', '/'}
  (prog
    (stat (expr ( (expr (expr 1) + (expr 2)) <missing ')'>) \n)
    (stat (expr 3) \n)
  )
```

Equally important is that the parser recovers to correctly match the second expression (the 3).

When using the -gui option on grun, the parse-tree dialog automatically highlights error nodes in red.

```
⇒ $ grun LibExpr prog -gui
⇒ (1+2
⇒ 34*69
⇒ EOF
```



Notice that ANTLR successively recovered from the error in the first expression again to properly match the second.

ANTLR's error mechanism has lots of flexibility. We can alter the error messages, catch recognition exceptions, and even alter the fundamental error handling strategy. We'll cover this in Chapter 9, *Error Reporting and Recovery*, on page 149.

That completes our quick tour of grammars and parsing. We've looked at a simple expression grammar and how to launch it using the built-in test rig and a sample main program. We also saw how to get text and visual representations of parse trees that show how our grammar recognizes input phrases. The import statement lets us break up grammars into *modules*. Now, let's move beyond language recognition to interpreting expressions (computing their values).

4.2 Building a Calculator Using a Visitor

To get the previous arithmetic expression parser to compute values, we need to write some Java code. ANTLR v4 encourages us to keep grammars clean and use parse-tree visitors and other walkers to implement language applications. In this section, we'll use the well-known visitor pattern to implement our little calculator. To make things easier for us, ANTLR automatically generates a visitor interface and blank visitor implementation object.

Before we get to the visitor, we need to make a few modifications to the grammar. First, we need to label the alternatives of the rules. (The labels can be any identifier that doesn't collide with a rule name.) Without labels on the alternatives, ANTLR generates only one visitor method per rule. (Chapter 7, *Decoupling Grammars from Application-Specific Code*, on page 109 uses a similar grammar to explain the visitor mechanism in more detail.) In our case, we'd like a different visitor method for each alternative so that we can get different "events" for each kind of input phrase. Labels appear on the right edge of alternatives and start with the # symbol in our new grammar, LabeledExpr.

tour/LabeledExpr.g4
```
stat:   expr NEWLINE                # printExpr
    |   ID '=' expr NEWLINE         # assign
    |   NEWLINE                     # blank
    ;

expr:   expr op=('*'|'/') expr      # MulDiv
    |   expr op=('+'|'-') expr      # AddSub
    |   INT                         # int
    |   ID                          # id
    |   '(' expr ')'                # parens
    ;
```

Next, let's define some token names for the operator literals so that, later, we can reference token names as Java constants in the visitor.

tour/LabeledExpr.g4
```
MUL :   '*' ; // assigns token name to '*' used above in grammar
DIV :   '/' ;
ADD :   '+' ;
SUB :   '-' ;
```

Now that we have a properly enhanced grammar, let's start coding our calculator and see what the main program looks like. Our main program in file Calc.java is nearly identical to the main() in ExprJoyRide.java from earlier. The first difference is that we create lexer and parser objects derived from grammar LabeledExpr, not Expr.

tour/Calc.java
```
LabeledExprLexer lexer = new LabeledExprLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
LabeledExprParser parser = new LabeledExprParser(tokens);
ParseTree tree = parser.prog(); // parse
```

We also can remove the print statement that displays the tree as text. The other difference is that we create an instance of our visitor class, EvalVisitor, which we'll get to in just a second. To start walking the parse tree returned from method prog(), we call visit().

tour/Calc.java
```
EvalVisitor eval = new EvalVisitor();
eval.visit(tree);
```

All of our supporting machinery is now in place. The only thing left to do is implement a visitor that computes and returns values by walking the parse tree. To get started, let's see what ANTLR generates for us when we type.

```
⇒ $ antlr4 -no-listener -visitor LabeledExpr.g4
```

First, ANTLR generates a visitor interface with a method for each labeled alternative name.

```
public interface LabeledExprVisitor<T> {
    T visitId(LabeledExprParser.IdContext ctx);          # from label id
    T visitAssign(LabeledExprParser.AssignContext ctx);  # from label assign
    T visitMulDiv(LabeledExprParser.MulDivContext ctx);  # from label MulDiv
    ...
}
```

The interface definition uses Java generics with a parameterized type for the return values of the visit methods. This allows us to derive implementation classes with our choice of return value type to suit the computations we want to implement.

Next, ANTLR generates a default visitor implementation called LabeledExprBaseVisitor that we can subclass. In this case, our expression results are integers and so our EvalVisitor should extend LabeledExprBaseVisitor<Integer>. To implement the calculator, we override the methods associated with statement and expression alternatives. Here it is in its full glory. You can either cut and paste or save the tour/EvalVisitor link (ebook version).

tour/EvalVisitor.java
```
import java.util.HashMap;
import java.util.Map;

public class EvalVisitor extends LabeledExprBaseVisitor<Integer> {
    /** "memory" for our calculator; variable/value pairs go here */
    Map<String, Integer> memory = new HashMap<String, Integer>();

    /** ID '=' expr NEWLINE */
    @Override
    public Integer visitAssign(LabeledExprParser.AssignContext ctx) {
        String id = ctx.ID().getText();  // id is left-hand side of '='
        int value = visit(ctx.expr());   // compute value of expression on right
        memory.put(id, value);           // store it in our memory
        return value;
    }
    /** expr NEWLINE */
    @Override
    public Integer visitPrintExpr(LabeledExprParser.PrintExprContext ctx) {
        Integer value = visit(ctx.expr()); // evaluate the expr child
        System.out.println(value);         // print the result
        return 0;                          // return dummy value
    }

    /** INT */
    @Override
    public Integer visitInt(LabeledExprParser.IntContext ctx) {
        return Integer.valueOf(ctx.INT().getText());
    }

    /** ID */
    @Override
    public Integer visitId(LabeledExprParser.IdContext ctx) {
        String id = ctx.ID().getText();
        if ( memory.containsKey(id) ) return memory.get(id);
        return 0;
    }

    /** expr op=('*'|'/') expr */
    @Override
    public Integer visitMulDiv(LabeledExprParser.MulDivContext ctx) {
        int left = visit(ctx.expr(0));  // get value of left subexpression
        int right = visit(ctx.expr(1)); // get value of right subexpression
        if ( ctx.op.getType() == LabeledExprParser.MUL ) return left * right;
        return left / right; // must be DIV
    }

    /** expr op=('+'|'-') expr */
    @Override
    public Integer visitAddSub(LabeledExprParser.AddSubContext ctx) {
        int left = visit(ctx.expr(0));  // get value of left subexpression
        int right = visit(ctx.expr(1)); // get value of right subexpression
        if ( ctx.op.getType() == LabeledExprParser.ADD ) return left + right;
        return left - right; // must be SUB
    }

    /** '(' expr ')' */
    @Override
    public Integer visitParens(LabeledExprParser.ParensContext ctx) {
        return visit(ctx.expr()); // return child expr's value
    }
}
```

And here is the build and test sequence that evaluates expressions in t.expr:

```
⇒ $ antlr4 -no-listener -visitor LabeledExpr.g4   # -visitor is required!!!
⇒ $ ls LabeledExpr*.java
❮ LabeledExprBaseVisitor.java     LabeledExprParser.java
  LabeledExprLexer.java           LabeledExprVisitor.java
⇒ $ javac Calc.java LabeledExpr*.java
⇒ $ cat t.expr
❮ 193
  a = 5
  b = 6
  a+b*2
  (1+2)*3
⇒ $ java Calc t.expr
❮ 193
  17
  9
```

The takeaway is that we built a calculator without having to insert raw Java actions into the grammar, as we would need to do in ANTLR v3. The grammar is kept application independent and programming language neutral. The visitor mechanism also keeps everything beyond the recognition-related stuff in familiar Java territory. There's no extra ANTLR notation to learn in order to build a language application on top of a generated parser.

Before moving on, you might take a moment to try to extend this expression language by adding a clear statement. It's a great way to get your feet wet and do something real without having to know all of the details. The clear command should clear out the memory map, and you'll need a new alternative in rule stat to recognize it. Label the alternative with # clear and then run ANTLR on the grammar to get the augmented visitor interface. Then, to make something happen upon clear, implement visitor method visitClear(). Compile and run Calc following the earlier sequence.

Let's switch gears now and think about translation rather than evaluating or interpreting input. In the next section, we're going to use a variation of the visitor called a *listener* to build a translator for Java source code.

4.3 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

2. http://asm.ow2.org

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 fifteen 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 8.3, *Generating a Call Graph*, on page 134.

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 that class ParseTreeWalker from ANTLR's runtime can trigger as it traverses the parse tree. In our case, we need to respond to three events by overriding three methods: 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 {
    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 by the ANTLR-provided walker object, whereas visitor methods must walk their children with explicit visit calls. Forgetting to invoke visit() 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 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
    )
    {
        // need parser to get tokens
        TokenStream tokens = parser.getTokenStream();
        String type = "void";
        if ( ctx.type()!=null ) {
            type = tokens.getText(ctx.type());
        }
        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);
ParseTree 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.

4.4 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. Matching this kind of input is pretty simple grammatically.

```
file : (row NL)+ ; // NL is newline token: '\r'? '\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 { // add members to generated RowsParser
    int col;
    public RowsParser(TokenStream input, int col) { // custom constructor
        this(input);
        this.col = col;
    }
}

file: (row NL)+ ;

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   :  '\r'? '\n' ;     // match and pass to the parser
STUFF:  ~[\t\r\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 and telling the parser not to build a tree.

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.setBuildParseTree(false); // don't waste time bulding a tree
parser.file(); // parse
```

There are a lot of details in there that we'll explore in Chapter 10, *Attributes and Actions*, on page 175. 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

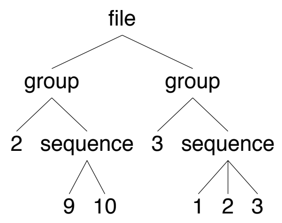
Until we get to Chapter 11, *Altering the Parse with Semantic Predicates*, on page 189, we can demonstrate the power of semantic predicates with a simple example. Let's look at a grammar that reads in sequences of integers. The trick is that part of the input specifies how many integers to group together. 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 three more as a sequence. Our goal is a grammar called Data that groups 9 and 10 together and 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)))
```

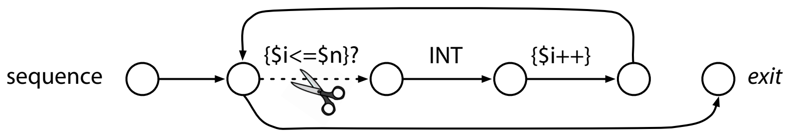
The parse tree clearly identifies the groups.



The key in the following Data grammar is a special Boolean-valued action called a *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 sequence 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 have a weapon for handling pathological parsing problems.

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

4.5 Cool Lexical Features

ANTLR has three great token-related features that are worth demonstrating in our tour. First, we'll see how to deal with formats like XML that have different lexical structures (inside and outside tags) in the same file. Next, we'll learn how to insert a field into a Java class by tweaking the input stream. It'll show how to generate output that is very similar to the input with minimal effort. And, lastly, we'll see how ANTLR parsers can ignore whitespace and comments without having to throw them out.

Island Grammars: Dealing with Different Formats in the Same File

All the sample input files we've seen so far contain a single language, but there are common file formats that contain multiple languages. For example, the `@author` tags and so on inside Java document comments follow a mini language; everything outside the comment is Java code. Template engines such as StringTemplate[3] and Django[4] have a similar problem. They have to treat all of the text surrounding the template expressions differently. These are often called *island grammars*.

ANTLR provides a well-known lexer feature called *lexical modes* that lets us deal easily with files containing mixed formats. The basic idea is to have the lexer switch back and forth between modes when it sees special sentinel character sequences.

XML is a good example. An XML parser treats everything other than tags and entity references (such as `&pound;`) as text chunks. When the lexer sees <, it switches to "inside" mode and switches back to the default mode when it sees > or />. The following grammar demonstrates how this works. We'll explore this in more detail in Chapter 12, *Wielding Lexical Black Magic*, on page 203.

tour/XMLLexer.g4
```
lexer grammar XMLLexer;

// Default "mode": Everything OUTSIDE of a tag
OPEN        :   '<'                 -> pushMode(INSIDE) ;
COMMENT     :   '<!--' .*? '-->'    -> skip ;
EntityRef   :   '&' [a-z]+ ';' ;
TEXT        :   ~('<'|'&')+ ;           // match any 16 bit char minus < and &

// ----------------- Everything INSIDE of a tag ---------------------
mode INSIDE;
```

3. http://www.stringtemplate.org
4. https://www.djangoproject.com

```
CLOSE       :   '>'                -> popMode ; // back to default mode
SLASH_CLOSE :   '/>'               -> popMode ;
EQUALS      :   '=' ;
STRING      :   '"' .*? '"' ;
SlashName   :   '/' Name ;
Name        :   ALPHA (ALPHA|DIGIT)* ;
S           :   [ \t\r\n]          -> skip ;

fragment
ALPHA       :   [a-zA-Z] ;

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

Let's use the following XML file as a sample input to that grammar:

tour/t.xml

```
<tools>
        <tool name="ANTLR">A parser generator</tool>
</tools>
```

Here's how to do a build and launch the test rig:

```
⇒ $ antlr4 XMLLexer.g4
⇒ $ javac XML*.java
⇒ $ grun XML tokens -tokens t.xml
❮ [@0,0:0='<',<1>,1:0]
  [@1,1:5='tools',<10>,1:1]
  [@2,6:6='>',<5>,1:6]
  [@3,7:8='\n\t',<4>,1:7]
  [@4,9:9='<',<1>,2:1]
  [@5,10:13='tool',<10>,2:2]
  [@6,15:18='name',<10>,2:7]
  [@7,19:19='=',<7>,2:11]
  [@8,20:26='"ANTLR"',<8>,2:12]
  [@9,27:27='>',<5>,2:19]
  [@10,28:45='A parser generator',<4>,2:20]
  [@11,46:46='<',<1>,2:38]
  [@12,47:51='/tool',<9>,2:39]
  [@13,52:52='>',<5>,2:44]
  [@14,53:53='\n',<4>,2:45]
  [@15,54:54='<',<1>,3:0]
  [@16,55:60='/tools',<9>,3:1]
  [@17,61:61='>',<5>,3:7]
  [@18,62:62='\n',<4>,3:8]
  [@19,63:62='<EOF>',<-1>,4:9]
```

Each line of that output represents a token and contains the token index, the start and stop character, the token text, the token type, and finally the line and character position within the line. This tells us how the lexer tokenized the input.

On the test rig command line, the XML tokens sequence is normally a grammar name followed by the start rule. In this case, we use the grammar name followed by special rule name tokens to tell the test rig it should run the lexer but not the parser. Then, we use test rig option -tokens to print out the list of matched tokens.

Knowledge of the token stream flowing from the lexer to the parser can be pretty useful. For example, some translation problems are really just tweaks of the input. We can sometimes get away with altering the original token stream rather than generating completely new output.

Rewriting the Input Stream

Let's build a tool that processes Java source code to insert serialization identifiers, serialVersionUID, for use with java.io.Serializable (like Eclipse does automatically). We want to avoid implementing every listener method in a JavaListener interface, generated from a Java grammar by ANTLR, just to capture the text and print it back out. It's easier to insert the appropriate constant field into the original token stream and then print out the altered input stream. No fuss, no muss.

Our main program looks exactly the same as the one in ExtractInterfaceTool.java from Section 4.3, *Building a Translator with a Listener*, on page 42 except that we print the token stream out when the listener has finished (highlighted with an arrow).

tour/InsertSerialID.java
```
ParseTreeWalker walker = new ParseTreeWalker(); // create standard walker
InsertSerialIDListener extractor = new InsertSerialIDListener(tokens);
walker.walk(extractor, tree); // initiate walk of tree with listener

// print back ALTERED stream
➤ System.out.println(extractor.rewriter.getText());
```

To implement the listener, we need to trigger an insertion when we see the start of a class.

tour/InsertSerialIDListener.java
```
import org.antlr.v4.runtime.TokenStream;
import org.antlr.v4.runtime.TokenStreamRewriter;

public class InsertSerialIDListener extends JavaBaseListener {
    TokenStreamRewriter rewriter;
    public InsertSerialIDListener(TokenStream tokens) {
        rewriter = new TokenStreamRewriter(tokens);
    }
    @Override
    public void enterClassBody(JavaParser.ClassBodyContext ctx) {
        String field = "\n\tpublic static final long serialVersionUID = 1L;";
        rewriter.insertAfter(ctx.start, field);
    }
}
```

The key is the TokenStreamRewriter object that knows how to give altered views of a token stream without actually modifying the stream. It treats all of the manipulation methods as "instructions" and queues them up for lazy execution when traversing the token stream to render it back as text. The rewriter executes those instructions every time we call getText().

Let's build and test the listener on the Demo.java test file we used before.

```
⇒ $ antlr4 Java.g4
⇒ $ javac InsertSerialID*.java Java*.java
⇒ $ java InsertSerialID Demo.java
❮ import java.util.List;
  import java.util.Map;
  public class Demo {
          public static final long serialVersionUID = 1L;
          void f(int x, String y) { }
          int[ ] g(/*no args*/) { return null; }
          List<Map<String, Integer>>[] h() { return null; }
  }
```

With only a few lines of code, we were able to tweak a Java class definition without disturbing anything outside of our insertion point. This strategy is very effective for the general problem of source code instrumentation or refactoring. The TokenStreamRewriter is a powerful and extremely efficient means of manipulating a token stream.

One more lexical goodie before finishing our tour involves a mundane issue but one that is a beast to solve without a general scheme like ANTLR's token channels.

Sending Tokens on Different Channels

The Java interface extractor we looked at earlier magically preserves whitespace and comments in method signatures such as the following:

```
int[ ] g(/*no args*/) { return null; }
```

Traditionally, this has been a nasty requirement to fulfill. For most grammars, comments and whitespace are things the parser can ignore. If we don't want to explicitly allow whitespace and comments all over the place in a grammar, we need the lexer to throw them out. Unfortunately, that means the whitespace and comments are inaccessible to application code and any subsequent processing steps. The secret to preserving but ignoring comments and whitespace is to send those tokens to the parser on a "hidden channel." The parser tunes to only a single channel and so we can pass anything we want on the other channels. Here's how the Java grammar does it:

tour/Java.g4
```
COMMENT
    :   '/*' .*? '*/'    -> channel(HIDDEN) // match anything between /* and */
    ;
WS  :   [ \r\t\u000C\n]+ -> channel(HIDDEN)
    ;
```

The -> channel(HIDDEN) is a lexer command like the -> skip we discussed before. In this case, it sets the channel number of these tokens so that it's ignored by the parser. The token stream still maintains the original sequence of tokens but skips over the off-channel tokens when feeding the parser.

With these lexical features out of the way, we can wrap up our ANTLR tour. This chapter covered all of the major elements that make ANTLR easy to use and flexible. We didn't cover any of the details, but we saw ANTLR in action solving some small but real problems. We got a feel for grammar notation. We implemented visitors and listeners that let us calculate and translate without embedding actions in the grammar. We also saw that, sometimes, embedded actions are exactly what we want in order to satisfy our inner control freak. And, finally, we looked at some cool things we can do with ANTLR lexers and token streams.

It's time to slow down our pace and revisit all of the concepts explored in this chapter with the goal of learning all of the details. Each chapter in the next part of the book will take us another step toward becoming language implementers. We'll start by learning ANTLR notation and figuring out how to derive grammars from examples and language reference manuals. Once we have those fundamentals, we'll build some grammars for real-world languages and then learn the details of the tree listeners and visitors we just raced through. After that, we'll move on to some virtuoso topics in Part III.