ANTLR Tutorial for Project #1

Michael McThrow
San José State University
CS 152 – Programming Paradigms

August 31, 2021

Note: This tutorial is a basic introduction to ANTLR that covers the features that are necessary for students to complete Project #1. This is not a full description of ANTLR; please refer to `https://github.com/antlr/antlr4/blob/4.9.2/doc/index.md` for a full-fledged guide to the latest version of ANTLR (v4.9.2) at this time of writing.

1 Introduction

ANTLR is a parser generator that is used in many real-world applications. Given a grammar written in ANTLR's custom grammar description language, ANTLR generates code that is capable of lexing and parsing strings written in the language defined by the grammar. Those implementing compilers, interpreters, or other programming language tools (e.g., style checkers, syntax highlighters, cheating-detection software, etc.) could adopt some of this generated code to traverse derivation trees generated by ANTLR. While in this project we will be using Java code generated by ANTLR, ANTLR is capable of generating code in other languages, including (but not limited to) C++, C#, and Python.

Using ANTLR requires having version 7 or higher of the Java Development Kit installed. To install ANTLR, simply download the file `https://www.antlr.org/download/antlr-4.9-complete.jar`. Using ANTLR requires that this `*.jar` file be included in your Java classpath. You can do this by either modifying your system's `CLASSPATH` variable (the details on how to do so depend on the operating system and the programming environment you are using) or by manually specifying the classpath while building and compiling your Java source code files that use ANTLR. On the home page of the ANTLR website under the "Quick Start" section, you can view instructions for how to set up the `CLASSPATH` variables and aliases for the `antlr4` and `grun` tools for macOS, Linux, and Windows.

2 How to Define a Grammar

ANTLR's syntax is a bit more complex than that of extended Backus-Naur form, but this complexity allows ANTLR to support many features that would be difficult to express using EBNF.

An ANTLR grammar is specified in a file ending with the `.g4` extension. The first non-commented line of the grammar is `grammar Name;` (note the ending semicolon), where `Name` is the name of the file without the `.g4` extension. These matching file and grammar names are required by ANTLR. For example, suppose we had a grammar named `QBasic` that describes the QBasic programming language. It would be stored in a file called `QBasic.g4`.

ANTLR supports the same type of comment syntax that Java and C++ does.

After specifying the name of the grammar using `grammar Name;`, we need to specify the rules of the grammar. Unlike in BNF/EBNF, in ANTLR there are two types of rules: lexer rules (which start with an uppercase letter) and parser rules (which start with a lowercase letter). The fundamental difference between lexer rules and parser rules is that when we write Java code that handles the derivation tree that ANTLR creates, that Java code interacts with the results of the parser rules, while there is no such interaction between our Java code and the lexer rules.

Here is how we specify terminal symbols in both lexer and parser rules:

- For string literals, we surround them with single quotes (e.g., `'+'`, `'*'`, `'var'`, and `'else'`).
- ANTLR supports various regular expressions for specifying terminal symbols. For example, if we want to match multi-digit integers, we can do so using `[0-9]+`, which means that the rule matches one or more occurrences of the digits 0, 1, ..., 9.

For specifying non-terminal symbols, we simply use those symbols without additional syntax. For example, suppose we have the parser rule `expr`. We could then define a part of the rule as follows: `expr '*' expr` (note the whitespace between the symbols; this is how ANTLR distinguishes between symbols).

For defining rules with just one part, we can use `name : definition ;`, where `name` is the name of the rule and `definition` is the definition of the rule.

For defining multi-part rules, we can use the `|` character to specify each part:

```
name : definition1
     | definition2
    ...
     | definitionN
     ;
```

For multi-part rules, we can choose to give each definition a label that may be very useful when writing code that traverses the derivation tree. Without these labels, it is more challenging to distinguish between different parts of the rule. Keep in mind that the decision to label definitions of a rule is an all-or-nothing affair; either none of the definitions within a rule are labeled, or all of them are. Note that label names may not conflict with rule names. Below is an example of what the syntax would look like for labeling definitions:

```
name : definition1 #label1
     | definition2 #label2
    ...
     | definitionN #labelN
     ;
```

Note that the ordering of definitions matters: `definition1` is checked before `definition2`, `definition2` is checked before `definition3`, and so on. This makes it easy to define order of operations without having to created recursively-nested rules as you would in BNF or EBNF.

Finally, there are two special rules in ANTLR that must be mentioned. The `WS` rule specifies how ANTLR should deal with whitespace. Since in Project #1 whitespace is insignificant, we can ignore whitespace by defining `WS` was follows:

```
WS : [ \t\r\n]+ -> skip ;
```

The second special rule is `Error`. Unfortunately error handling in ANTLR is a complex topic and is rather convoluted for an undergraduate assignment, in my opinion, though if you were writing production code, then you'd have to deal with this. How I deal with errors is including this line, which punts errors from the lexer to the parser:

```
Error : . ;
```

Let's conclude this section with an example grammar: an ANTLR implementation of the simple arithmetic expression grammar defined in the "Syntax in Programming Languages" handout:

```
/*
 * SimpleArith.g4 -- ANTLR grammar for simple arithmetic expression language
 *
 * Michael McThrow
 * CS 152 -- Section 06
```

```
 * San Jose State University
 * Fall 2021
 */
grammar SimpleArith;

WS          : [ \t\r\n]+ -> skip ;

AddOp       : '+' | '-' ;
MultOp      : '*' | '/' ;

Integer     : '-'? [0-9]+ ;

expr        : '(' expr ')'       # parensExpr
            | expr MultOp expr   # multExpr
            | expr AddOp expr    # addExpr
            | Integer            # intExpr
            ;

Error       : . ;
```

3 Generating Java Files from an ANTLR Grammar

Now that we have a grammar file defined, the next thing we need to do is generate Java files that perform lexing and parsing using that grammar. To generate these files, we use the `antlr4` command as follows:

```
antlr4 -no-listener -visitor SimpleExpr.g4
```

The `-no-listener` and `-visitor` flags inform ANTLR to create visitor classes for traversing the derivation tree based on the parse rules in the grammar. (Listeners are an alternative way of traversing the derivation tree, but in my opinion they are more complex to use and thus I won't be covering them.)

After running the above command, you will now see a bunch of generated Java files that all begin with the name of the grammar. Our next step is to compile them all; a convenient way of doing so is `javac *.java`.

4 Visualizing Derivation Trees with ANTLR

Once we have a grammar file, we can view derivation trees of strings parsed by ANTLR. Suppose we have the `SimpleArith.g4` grammar file and we want to see the derivation tree for `3 * (4 + 5 * 6) + -7 / 8`. To view the derivation tree, we use the `grun` command as follows:

```
echo "3 * (4 + 5 * 6) + -7 / 8" | grun SimpleArith expr -gui
```

(Please note that this command will fail if you did not compile the Java files that ANTLR generated.)

Let's pay attention to the arguments of `grun`. The first argument is the name of the grammar. The second argument is the start rule for parsing the input. The third argument is the `-gui` flag, which tells `grun` to display the derivation tree in a new window titled "Parse Tree Inspector."

Figure 1 shows what the parse tree for `3 * (4 + 5 * 6) + -7 / 8` looks like. This is a handy tool for checking to see if your grammar file is parsing expressions as you intended.

Another option for viewing parse trees is to replace the `-gui` flag with the `-tree` flag. The resulting output looks like this (slightly modified for formatting purposes):

```
(expr (expr (expr 3)
            *
            (expr ( (expr (expr 4) + (expr (expr 5) * (expr 6))) )))
```

```
+
(expr (expr -7) / (expr 8)))
```

This style of syntax is known as an *S-expression*, which is a convenient way of representing tree-like data structures. I won't expound on S-expressions in this ANTLR tutorial, but I promise you this is definitely not the last time we'll be covering S-expressions in this course.

5 Visitor Classes

After running `antlr4`, you will notice a few Java source code files that start with the name of the grammar. One of them ends with `BaseVisitor.java`. This is the file that we are interested in exploring. The `BaseVisitor<T>` class is a generic class that has a method for each parse rule of the grammar. (If the parse rule is a multi-part rule with labels for each part, then instead of the parse rule having a method, there is a method for each labeled part.) Each method has a parameter that represents the context of the

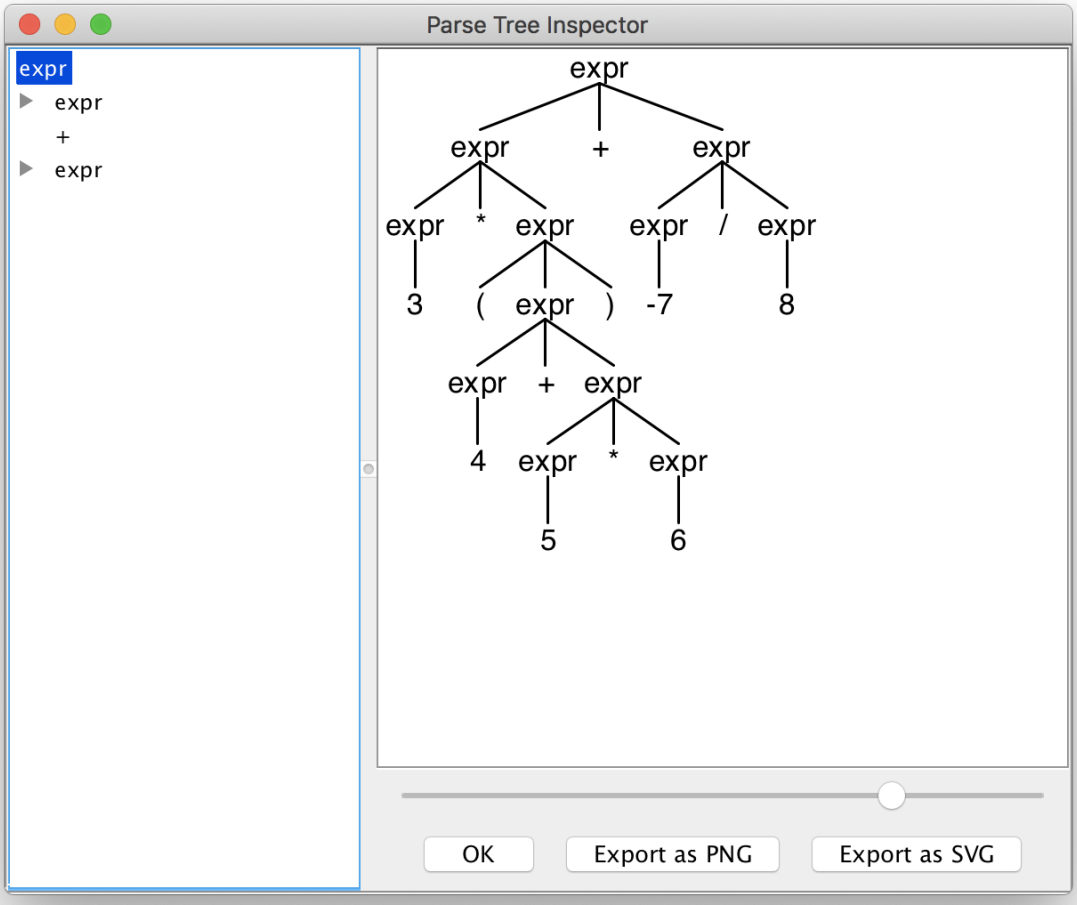

Figure 1: The resulting derivation tree when parsing the expression `3 * (4 + 5 * 6) + -7 / 8` using the `SimpleArith` grammar.

corresponding rule (or labeled part). The class definitions of these context objects can be found in the generated `Parser` class. Each context has methods that correspond to the names of lexer and parser rules referred to by the rule the context represents. (If the same lexer/parser rule is referenced multiple times, then provide a index parameter [starting from 0] referring to the specific symbol being referenced.) The return values of each method are all of the same type `T`.

Developers of interpreters, compilers, and other programming language tools can use this `BaseVisitor<T>` class by creating a new class that inherits that class and specifies the type `T` that serve as the return values of each method. Every `visit` method in this new class must override its inherited `visit` methods. Finally, in each `visit*` method, you must visit each child element of the context that corresponds to a non-terminal parse rule using the `visit()` function.

Continuing with our `SimpleArith` example, let's explore the contents of `SimpleArithBaseVisitor.java`. The base visitor method for `SimpleArith` has the following methods, each corresponding to the labeled parts of the `expr` parse rule:

- `public T visitIntExpr(SimpleArithParser.IntExprContext ctx)`
- `public T visitAddExpr(SimpleArithParser.AddExprContext ctx)`
- `public T visitParensExpr(SimpleArithParser.ParensExprContext ctx)`
- `public T visitMultExpr(SimpleArithParser.MultExprContext ctx)`

Suppose we create a new class named `SimpleArithEvaluator`:

```
public class SimpleArithEvaluator extends SimpleArithBaseVisitor<Double> {

}
```

Let's implement two methods: `visitIntExpr` and `visitAddExpr`. We can study these examples to see how to use the `ctx` variables and how we can use `visit()` on non-terminal symbols.

```
public Integer visitIntExpr(SimpleArithParser.IntExprContext ctx) {
    return new Integer(ctx.Integer().getText());
}

public Integer visitAddExpr(SimpleArithParser.AddExprContext ctx) {
    String op = ctx.AddOp().getText();

    if (op.equals("+"))
        return visit(ctx.expr(0)) + visit(ctx.expr(1));
    else
        return visit(ctx.expr(0)) - visit(ctx.expr(1));
}
```

After understanding how `visitIntExpr` and `visitAddExpr` works, implementing the other two methods should be trivial.

6 Putting It All Together

The final piece to writing a Java program that parses an expression written in a language specified by an ANTLR grammar and then traverses its parse tree is writing code that uses the generated `*Lexer.java` and `*Parser.java` files.

Below is the source code for `SimpleInfixCalculator.java`, which takes an expression from standard input and outputs the result of evaluating that expression.

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

public class SimpleInfixCalculator {
    public static void main(String[] args) throws IOException {
        // 1. Read input from standard input, creating a character stream.
        BufferedReader in =
         new BufferedReader(new InputStreamReader(System.in));
        CharStream inputStream = CharStreams.fromReader(in);

        // 2. Perform lexical analysis on the character stream.  The purpose
        // of the error listener is to throw an exception whenever the lexer
        // encounters a syntax error.
        SimpleArithLexer lexer = new SimpleArithLexer(inputStream);
        lexer.addErrorListener(new BaseErrorListener() {
            @Override
            public void syntaxError(Recognizer<?, ?> r, Object o, int l, int c,
             String msg, RecognitionException e) {
                throw new RuntimeException(e);
            }
        });
        CommonTokenStream commonTokenStream = new CommonTokenStream(lexer);

        // 3. Parse the stream of tokens generated by the lexer by calling
        // the expr() method, which corresponds to SimpleArith's expr rule.
        SimpleArithParser parser = new SimpleArithParser(commonTokenStream);
        parser.setErrorHandler(new BailErrorStrategy());
        ParseTree tree = parser.expr();

        // 4. Given the resulting parse tree, evaluate it.
        SimpleArithEvaluator evaluator = new SimpleArithEvaluator();
        Integer result = evaluator.visit(tree);

        // 5. Print result
        System.out.println(result);
    }
}
```