Creating a Parser and Lexer in ANTLR v4

Dean Fletcher, Darshan Patel, Charles Read

****

**ABSTRACT**

**Background:** First created in 1989 by Terrance Parr [1], ANTLR was created to help computer science students, and scientists themselves, parse through lines of code, to separate it into tokens for a better understanding. What we tackled in this project, was to gain an understanding of how ANTLR worked and to create a lexer and parser ourselves.

**Results:** A dummy grammar was created to help generate and run the lexer and parser. Our choice in this was an employee database where a user can input a new employee with name, date of birth, and employee ID, as well as delete the employee, and update an existing employee. A separate function was created to showcase ANTLR’s ability to parse through arithmetic equations.

**Conclusion:** We were successfully able to generate successful parse trees using both the arithmetic function and the “setter” functions. Those will be included later in the report.

**1 Background**

The purpose of a parse tree is to show the reader how exactly a certain line of code can be broken down, all the way to its parent header. Its primary purpose is for use in context-free grammars to show how an end symbol can be traced back to a non terminal symbol, which could be the start symbol, but this is not always the case [2]. ANTLR is capable of outputting one of these parse trees for a given program and is used throughout the industry today. ANTLR’s website claims that Twitter uses ANTLR to help parse through queries every day [1]. ANTLR will take in a given grammar, and use that language to help parse through the given code, generating the parse tree for that code.

**2 Our Task**

Throughout completing our project, we looked to multiple directions on how to tackle ANTLR. Our main focus was first to learn what ANTLR was and what it was capable of. Once this was accomplished, we then took a more direct approach to create parse trees with something that could be quite useful in a workplace environment. It also can tie in quite well to either Object-Oriented Programming or Data Structures, depending on which class the student is taking, in how a command line function for an object is broken up. Through our research, it was found that ANTLR itself does not have memory, rather it will generally act upon the respective language it is given and use that code snippet for memory. For instance, in our case, an employee database can prove to be quite useless unless it is stored. In a C++ program, that database can have an output statement which when parsed, all the cout statements can be traced back to the standard library within C++.

**3 Results**

Creating the lexer and parser was not difficult in the sense of writing or making, rather the difficulty lied in setting up our environment to actually run ANTLR. Once that was set up, we needed to read through the ANTLR documents, and then we were able to see exactly what commands were needed to generate the parse tree we wanted. When we first tried running ANTLR, we ran with our “add” command. Afterward, we were able to see exactly how the tree looked.

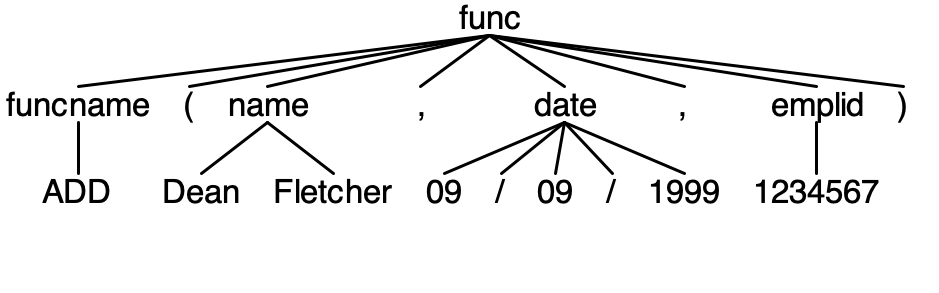


Figure 1: parse tree for ADD function

In this example, the overall function is the command line input, which is broken up into its respective token. Each token has a specific role as outlined in the grammar to the side. For the case above, there is the overall funcname ADD, which in this case, can be denoted as either “Add” or “ADD,” with only the two being acceptable cases. If you then move on to the date token, you can see how it is broken up into three “INT”s and two “FWD\_SLASH”s. The code written by the programmer would be the one to check if it is valid or not. Similarly, name takes two strings.

The other parse tree that was created was one for calculating pay for an employee based on an inputted wage and hours worked. These could either be a double or an integer. That tree is shown at the right.

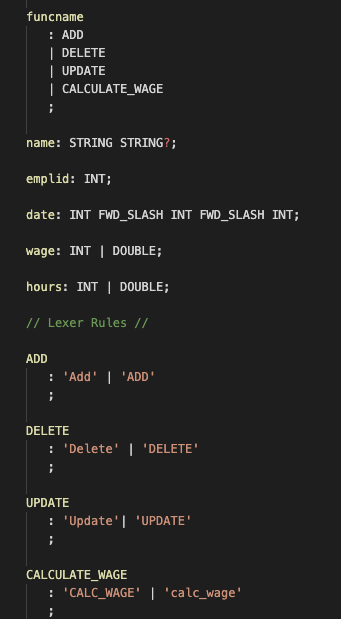


Figure 2: Snippet of code from grammar file

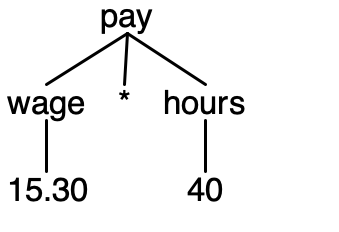


Figure 3: pay parse tree

**4 Conclusion**

This project taught us a lot about how a parse tree works, what is needed to build a parser and lexer, and quite frankly, what they are used for. Learning that something so simple can be used to process millions of searches every day on an app that millions of users log on to regularly, can be quite mind blowing. ANTLR is such a simple yet powerful tool that one would never think twice about, truly making it, Another Tool for Language Recognition. While there are many out there that can accomplish the same goal, none are quite as intuitive and easy to code as ANTLR. Our project merely scratched the surface of what ANTLR is really capable of, and with some more practice, we could really create some cool parse trees with code we’ve written in the past.

**5 For Next Time**

In the future, if we were to work with ANTLR some more, we would dive deeper into how the listener and visitor of ANTLR work. These two files are quite important when it comes to actually generating the parse tree, but involve more focus on each. When ANTLR is run, multiple files get generated, both in the code that you are running ANTLR on, but also some java files. These are the files that we would look at closer and use to help see more of what ANTLR is capable of.

**6 Bibliography**

<span style="font-variant: normal">Anon</span>.https://www.antlr.org/about.html

<span style="font-variant: normal">Anon</span>.https://www.cs.wcupa.edu/rkline/fcs/parse-trees.html