Leon Zeltser

Professor Wilkes

COMP 4810

April 29, 2025

Parse Tree Visualizer

My project is a program that shows how a compiler builds a parse tree. Parsing is the second step in the process where code is compiled from a human-readable form to machine code. It place just after tokenization, where the code is turned into tokens, which are the smallest pieces that have meaning, such as numbers, operators, or variables. Parsing is the process where a compiler discovers the meaning of the code and the tokens' relationship with each other and then places it into a data structure like a tree. The type of information that needs to be encoded includes operations being done with numbers and variables and order of operations, parameters that belong to certain functions, and if code belongs to a certain subroutine, to name a few. Every modern programming language is defined in terms of a context-free grammar, and a parse tree shows how a string is derived from the rules of the grammar.

There are two classes of parsing algorithms, top-down and bottom-up, and both use a pushdown automaton (PDA). Top-down algorithms build the tree starting with the root node and work its way down to the leaf nodes, which are the tokens. They are also called LL, for reading tokens from the left and discovering rules from the left, which is done by predicting what rule the parser expects to see based on where in the tree it is and what the next token is. The LL parser's PDA consists of a single state, and on each entry it pops the top item off the parse stack, and acts based on if it is a terminal or not. If the symbol is a non-terminal, the parser will look up the symbol name and front token on a table, and if it matches a certain rule, it will push the symbols in the rule to the stack, otherwise it will throw a parse error. If the symbol is a terminal, it will compare the symbol with the first token in the token

stream, and if they match it will consume the token, otherwise throw a parse error. Another type of top-down algorithm is recursive descent, which uses handwritten code rather than a table made by a parser generator. Every symbol is a function, which corresponds with rows on the table-driven parer's table, while the front token is instead put in a case statement, which corresponds with columns on the table. In case of a non-terminal rule it will call another function, while in the case of a terminal it will attempt to match the token at the front of the token stream in the same way. The call stack for the recursive recursive descent routine corresponds with the parse stack on the table-driven parser.

Bottom-up parsers are another type of algorithm, and they build the tree starting with the leaf nodes on the bottom and work their way up. They are called LR because read tokens from the left but derive rules from the right by recognizing rules instead of predicting. Their PDA has multiple states and has two types of actions, shift and reduce. On a shift the parser shifts to another state and pushes the top token along with the new state onto the top of the stack. In some cases it may be a token, and in others a partially-built tree. When the parser recognizes a rule it reduces, meaning it pops several tokens off the top of the stack and places them under a new node with the name of the rule it recognized and pushes this tree to the top of the token stream, then it goes to the state that is on top of the stack. Unlike the LL parser, the LR parser's table looks up the state and the first symbol in the token stream, which could be a token or another rule. If no action is found it throws a parse error. This project uses a variant of the LR parser called SLR, short for simple LR, which differs from other LR parsers in how it resolves ambiguities when making the parse table.

The tool I was working with was originally made as a Java applet in the late 1990s, and ported to newer versions by two honors students. Professor Wilkes currently uses the newer Java version in his organization of programming languages class. My goal was first to port it to Python, and then make a parser generator so it can work with any grammar a user enters. The first thing I did is look for graphics libraries, since I never did any graphics or GUI work before. I decided to use PyQt6. I found a tutorial

online, and after a day I felt I was comfortable with the library. It was very easy to use and I found I never was fighting against it. The library also has a tool called QtCreator which lets the user drag and drop elements on a screen to create the GUI, and then export the code in XML, while the Python library has a script to turn that into Python code. This allowed me to make the entire GUI in a matter of minutes, and making any change to it was also easy. The rest of the semester I made the three parsing algorithms, with the calculator language from *Programming Language Pragmatics* by Michael L. Scott hard coded into the program. The simple calculator language has three types of statements, reading a number from an input, writing an expression to an output, assigning the value of an expression to a variable. The extended calculator language has two more, an if statement that runs some code of a condition is met, and a while loop that continues while a condition is met, and those two make the language Turing complete. It took some time but I was eventually able to get the three algorithms to work. During the winter break I spent a lot of time cleaning up the code.

In the spring semester I worked on the parser generators. I found that the more difficult part was understanding how the algorithms worked, but once I figured that out I was able to quickly turn it into code. The LL parsers work using the first and follow sets of each symbol. The first sets consist of the tokens that could be at the start of every rule, and the follow sets consist of any token that can come after the rule. From these it generates the predict sets, which say if the parser is in a certain symbol, what rule it should follow based on the token it sees in front of it. The recursive descent code and LL table are generated from these sets. The LR parser generator works differently, it generates a series of states based on the rules, then uses the states and follow sets to build a table.

The program takes a programming language's grammar and code as an input. It generates the parse table or rules then generates tokens from the code. The user could either press the step button for the program to do one step, or just let it run until completion. There is also a button to reset the parse tree from the start. The program displays the token stream and the parse stack in the bottom right

corner, while in the bottom left it either shows the parse table or the recursive descent code, with an option for different programming languages. On the parse tree section on top, nodes are on the parse stack (or call stack in the case of recursive descent) are green, while nodes that are not are red.

I have a list of features I planned to add but did not have time to in the readme, some being from the original proposal and some not. Most are small changes, like adding tooltips or settings to change how the tree is displayed. Two features on it were some of the stretch goals from the proposal, which are allowing the tree to be exported as an image file and to allow grammars or code to be uploaded from a file. Another possible addition new languages for recursive descent, and new context-free grammars for programming languages. The former just involves adding a configuration file in a directory, while for the latter, grammars have to meet certain restrictions for the parsers to be able to run them. I did attempt to add multiple different programming languages towards the end of the semester although I did not as I ran into difficulty refactoring the grammar to meet these restrictions. Two potential larger features to add is more parsing algorithms and error handling. There are more types of parsers besides the three in my project, for example LALR which is similar to SLR but better at handling ambiguities when generating a table, and LR is also similar and even better than LALR in dealing with ambiguities although it generates much more states than LALR and SLR do, and because of that it is generally not used. There are also versions of all the algorithms that look ahead multiple tokens instead of just one. Finally the Earley and CYK algorithm are ones which can parse any grammar, although do not scale as well as the others. Error handling is something most parsers do, rather than stop parsing entirely like this program does in case there are more errors. There are multiple ways of doing it, for example deleting tokens from the token stream until a “safe” token is reached or inserting a token if one is missing, though this requires extra inputs.

There is a lot I learned from developing this project besides the different parsers and parser generators, and one thing is that developing features often takes longer than expected. There were

plenty of times I said I will make something by the end of the week and end up not, often because of a single bug that was hard to track down. While the main goals of the project were done, I did have a number of stretch goals I outlined in my proposal and the readme file that I did not have the time to get to. Had I been working on the project full time or been given more time I would have been able to finish them. Related to this is feature creep. There were many times over the course of the project where I thought about adding new features to add that were not in the original project proposal. Each time I stepped back and reminded myself that I am constrained by time, and reminded myself of the purpose of the project, and what I had in in the proposal is enough to meet the purpose, a tool used in a programming languages or compiler class. Something else I learned is the importance of clean code especially when it comes to debugging. I worked to keep different components separate, which helped when I was struggling with some bugs. There was one time when the tree did not display properly, and knowing the parser works correctly meant I could just focus on the grid the tree is placed on before it is displayed. After looking at the code for days, I ended up figuring out the cause of the bug just after I woke up one morning. Overall I did enjoy working on this project, and am happy to have the chance to make something that will be used in classes.

Project GitHub repository link:

https://github.com/lzeltser/ParseTreeVisualizer-Python