Project Report Summer 2025

Aaron Rogers

August 8, 2025

1 Overview

In this report you will find my experiences this past summer from studying compiler design. A compiler is a program that transforms higher level language to lower level assembly language. The use of a program that performs this function allows for programming to be quick and accessible. Of course these types of programs are no magic. They are instead in modern implementations, programs themselves.

Compilers are not to be confused with assemblers or linkers. Assemblers turn human readable assembly into machine code where as linkers organize that code allowing for it to be properly loaded into memory. I include this distinction in this report because many modern compilers automatically call a linker and assembler in there function but are only called a compiler such as the gcc and zig compilers.

A compiler performs a series of 'passes' on a static text file, performing different tasks on each pass. The first pass handles whats called pre-processor which handles macros and other language features that are to be handled before the main parsing process. The next pass would involve Lexical analysis and parsing which performs syntactic analysis and generates a parse tree based on a context-free grammar. Following this pass is code generation where stepping through a parse tree the compiler creates intermediary language. These first steps are referred to as the frontend where as the backend focuses on converting intermediary code into assembly for the target architecture.

Natural languages have grammar or syntactic rules that make them make sense. In a similar sense programming languages also have grammars that determine hierarchies that allows them to make sense in an exact manner these hierarchical grammars are called context free grammar. In a context free grammar literal components of the grammar are called terminals where as a collection of terminals or non-terminals are called non-terminals. Another approach to grammars are regular grammars which use regular expressions to recognize it's syntax but regular grammars are to limiting to use to implement a fully featured language.

2 Lexing and Parsing

For the remainder of this report I am going to focus on frontend components of a compiler starting with lexical analysis. A Lexer is a component which takes a series of symbols and converts them into tokens which are more useful to a computer then the symbols its converting from. The original string that becomes a token is called a lexeme. Tokens tend to have some kind of type identifier and can also contain meta data such as a number value if it is a literal, debug information such as the location of the lexeme in the source file, and even the lexeme itself. This token data is then useful later when parsing and generating code.

A parser steps through or creates a parse tree based on a given context-free grammar. Each non-terminal set is called a production rule as a non-terminal is then translated into its set. By analyzing a string of tokens a parser is able to follow a set of production rules to create a parse tree. Example: [some example here]

While stepping through a parse tree, the compiler can then generate intermediary code depending on which production rule to follow. In this way out grammar can allow our high level language to be easily translated by the compiler. In each rule we can also perform tasks like type checking and other syntactical analyses rules.

I wrote my own LL(1) parser from scratch which is a predictive parser that reads the next token and based off the grammar it can predict what the next production rule should be based off that token. My lexer tokenises strings and stores them in a queue that passes to the parser where one by one it determines the next production rule to step into based off of the previous token. This style of parser is called a recursive decent parser as the production rules are functions which call each other depending on the next outlining a parse tree.

3 YACC and LEX

It was at this point in my study I started messing with tools like YACC and Lex. YACC is an acronym for "yet another compiler compiler" that acts as a tool which streamlines compiler creation. YACC allows for you to write in your grammar to a *.y file and directly in your grammar you can write C code. This makes YACC a fast way to get up and running with creating a compiler as it embraces both a declarative and imperative style as you don't have to write your own parse tree and instead just focus on the grammar. But for each production rule you can write your own logic to be executed when the parser reaches that rule.

Lex works in tandem with YACC to create something fully functional. Lex allows you to write C code right into regular expressions which will determine how the lexer will behave when it reaches a match for that regular expression. This allows for a very customizable lexer which can keep track of lines and columns and save meta data about tokens while also being very easy to work

with.

Since the YACC and Lex combo is very programmable, I also write my own C files to include to make my compiler prototype more complete. I create my own symbol table to help with syntactic analyses as well as a stack to keep track of register allocation when generating code to make sure registers do not get clobbered.

4 Conclusion

Overall as someone who is interested in lower level systems programming this project has been a good experience for me. I feel that at the end of this I have A better understanding of parsers and compilers in general. With this knowledge I know that with some more work I could definitely make a feature rich programming language. Learning about parsers was also valuable for me as I imagine parsers can be generally useful for programs that need to interpret some kind of user input such as creating a debug command shell for a video game similar to the one in quake or doom.

Currently my project is at a place where all the groundwork is set out for code generation. All that's remaining to be done is filling out the logic for the remaining production rules and then a backend implementation. I imagine the backend should be simple to implement since my intermediate code style is very similar to assembly language already I would just need to swap out symbols based on the target architecture.