**CPSC 323 Project Documentation**

Project 3: Jason Haselle and Andrew Pham

1. **Problem Statement**

Part 1: Compilers need to read through symbols and characters to properly function and carry out the intended operations. To do this, we propose analyzing the sequence of characters in our lexer and assigning them tokens. These tokens are pre-identified and assigns the input meaning.

Part 2: Compilers also need to check for proper grammar and syntax in the code. To do this, we use a top down parser, specifically a predictive recursive descent parser. The parser takes the tokens generated by the lexer and processes them by calling functions matched to certain syntax rules. Functions will call new functions or recurse until hitting a terminal. The parser accepts and checks one token at a time. While processing, it prints the production rules used. Unexpected tokens or end of file will result in an error indicating a syntax error to the programmer.

Part 3: Finally, compilers generate assembly code for the given machine from the inputted code. These are basic steps with logic the machine is built to handle. Our machine is stack based, understands basic operations, and has memory address starting at 2000. Our compiler needs to create assembly code that can access memory, and perform basic arithmetic operations. We do this by creating a symbol table with our identifiers, and using the table to manipulate the data in memory to perform our desired operations. In the special case for jumps we need to modify the assembly with the jump lines, because of this we need to hold our assembly instructions in a data structure and we cannot print them out real time. Once we find the jump, we can modify the instructions with the proper line. Assembly code is all outputted at the end.

1. **How to use your program** 
   1. Unzip the folder
   2. Navigate into the Debug file
   3. Modify input.txt if desired
   4. Run Lexer.exe
   5. Open and examine output files
2. **Design of your program**

Part 1: The main function contains a while loop, which extracts an individual string. This string is sent to the lexer function, which uses a while loop to parse the individual characters. The lexer function contains a switch statement that determines the case or state. Each case/state has a series of if statements to add the character to the lexeme string and or change the state if necessary. Once an accepting state is reached, the token is assigned, the lexeme and token is returned, and both are printed to the “Token\_List.txt until the end of file is reached.

Part 2: Next the main function declares the parser and calls the Begin function. The constructor for the parser opens the generated “Token\_List.txt” as input. The Analyzer will take from the token list one at a time and recursively compute the parse tree. Each function call first prints to the “Parse\_Tree.txt” its grammar rule and calls the next rule in search of a terminal. The functions are called in order from the grammar and based on what lexeme is current. Eventually the function calls will reach a terminal where it will increment the parser or all terminals are reached.

Part 3: Assembly code is generated at the same time as the parse tree. Each identifier is stored in a memory address and continues incrementally. As the lexer finds identifiers, we check if it is already in the symbol table. If it’s not in the table and we are assigning it, we give it the next memory address and add it to the table; if it is in the table and we are assigning it, we return an error. If we are using the identifier and it is in the table, we access the memory address with it; else, error. Upon hitting key lexemes, our parser calls Generate\_Instructions to add the proper operation and memory location to the instruction\_table. This is done by checking for the identifiers in the table, accessing their memory locations, and pushing the memory locations and operations to the instruction table. If identifiers are not present in the table, we return an error. Upon completion, the instruction\_table is outputted as “Assembly\_Code.txt”

1. **Any Limitation**

*none*

1. **Any shortcomings**

*no shortcomings*