Our compiler project is our capstone project for our bachelor’s degree in computer science. The project is to create a compiler for the computer language called “LITTLE”. A very general definition of a compiler is something that translates a language into another. Our compiler takes source code and will translate it into assembly code. Our compiler will do this in 4 main steps. The scanner, parser, symbol table, and semantic routines.

Our first step was to create the scanner. The scanner is to read the source file and it tokenizes the file. As long as all the tokens are within the language, it will output the token’s type and the value. For our compiler we started off using C# in the Visual Studio IDE. We were using C# with the ANTLR framework but after some struggling with ANTLR, decided we would change to Python and Lex. We already had some experience with Lex from our lab, so it was easier to get the scanner figured out. We created our grammar using Lex and regular expressions. Once we had it recognizing the tokens in the different source files, we just had to make our output match the output files.

Our second step is to create the parser. To create our parser, we used PLY. Ply is an implementation of lex/yacc parsing tools for Python which uses LR-parsing. An LR parser is a bottom up parser. So it looks at the string and creates the parse tree from the string using a specific grammar. One of the problems we ran into along the way was some of our token names were not specific enough to create our parse rules. It would accept strings that were not actually in the language. We fixed this by creating more specific token names. For example, we had all the +, -, \*, / just under the operator token, we changed this by creating a unique token name for each operator (i.e. addOp, subOp, mulOp, divOp). Once we created more specific token names, we were able to write our parse rules so they would only accept strings if they were actually recognized by the LITTLE language.

In step three we created our symbol table. To do this we used a dictionary. We found this to be the best way to go about the symbol table rather than using five different arrays. After we determined that the best route for the symbol table was a dictionary we found that we needed to account for scope of each of the variables. Obviously this scope could be global or local. We then added the variables to our stack and then checked their scope. After we checked their scope we added them to our main array to get ready to print. After we added the global and local variables correctly to be printed, we had to do something about the blocks. Specifically the if then statements and the while statements. The way we saw to do this was to add almost “breakpoints” in the if and while statements. We added the block method to our symbol table class and then if we read that there was an if or while statement then we would break out of the statement and add it to our array and symbol table accordingly. After adding all of these correctly we formatted to printed out our symbol table.