The GoLite project is a basic compiler built from scratch based on specifications from the Go language developed by Google. The goal of this project is to gain a deeper understanding into how compilers work and how they are built by building very barebone version of Go with some differences for simplification. Golite implements essentially a subset of the basica types, semantics, and grammatical constructs of Go. This report will describe each of the compiler phases of Golite in order, namely the lexer, parser, abstract syntax tree, symbol table, and typechecker, and go over all design decisions that were made throughout the development process.

The implementation language of this project is c++. it is chosen because of its OOP capabilities, and simply because I wanted to be a more proficient c++ programmer. I chose to use CMake to make compiling much simpler, and the testing was done using test scripts that were provided. The lexer and parser were written using the flex and bison tool, and github was used for source control.

The lexer is the first part of the compiler that takes the source file which is just text input and outputs a variety of tokens that is used in the next phase by the parser. It scans through the file based on the regular expressions in the lexer, and will either discard the match or output a unique token and store the value of the match. Though scanning and spitting tokens seem fairly trivial, there were interesting challenges due the Go specifications. For example, in Go, there is an optional semicolon rule where if a line ends with a specific token and it doesn't have a trailing semicolon, the lexer will output that token and an extra semicolon token.

The parser is the next part of the compiler phase after the lexer. It receives the tokens outputted by the lexer and forms an abstract syntax tree using a context free grammar. The context free grammar is essentially a list of top level declarations, which can be variable declarations, type declarations, or function declarations and each function body expands to be a list of statements. These lists are constructed using c++ vectors and helper rules. The inheritance relationships of the AST are as follows:

NDecFunc, NDecVar, NDecType, … ---> NDeclaration ---> NAbstractAstNode
NStmtAssign, NStmtBlock, NStmtDec, ... ---> NStatement ---> NAbstractAstNode
NExpBinary, NExpUnary,NExpLiteral, ... ---> NExpression ---> NAbstractAstNode
NTypeArray, NTypeIdentifier, … ---> NType ---> NAbstractAstNode

The naming of class prefixed by N is highly inspired from the resource https://gnuu.org/2009/09/18/writing-your-own-toy-compiler/. I designed the AST here with using the visitor pattern in mind. This design decision allowed me to add an extra phase very easily, and abstracted the different parts of the compiler into separate components which increased readability and scalability. Here, NAbstractAstNode is the element interface and NDeclaration, NStatement, NExpression, NType being concrete elements. Then I added the class NAbstractDispatcher which is visitor interface and adding a new compiler phase, which is the concrete visitor, will simply require inheriting from NAbstractDispatcher.

After the abstract syntax tree is constructed, a weeding phase is required to weed out all the invalid constructs that would have been too difficult and convoluted to be handled in the parser. These include division by zero, break and continue being outside of loops, switch statements having more than one default case, etc. This is simply done by passing in the root node of the AST and traversing through the tree.

In the next phase of the compiler, symbol tables are used to maps identifiers to the AST node where it was first declared. The symbol phase is done by again passing in the root node of the AST and in the process of traversing through the tree, it creates a cactus stack of symbol tables. When entering a new scope, a new symbol table is created and pushed onto the cactus stack, and when exiting a scope, the top of the cactus stack is popped. When the traversal reaches a declaration node, the identifier for that declaration is added to the current symbol table that maps to that node. In addition, a symbol indicating what kind of identifier it is, function, constant, type, or variable, is created and weaved into the node. When an identifier is used, we can easily see if the identifier has been previously declared by simply traversing up the cactus stack. By doing this, we ensure that all variables follow the scoping rule defined by the symbol phase.

Despite the correct scoping, the identifier might not have been used semantically in the correct context. This is where the typechecking phase comes in. The typechecking phase ensures that semantically, the program makes sense. For example, ensuring that the right hand side of an assignment or declaration is assignable to the left hand side, ensuring that the function returns a value of the type specified in its declaration, etc. This phase is done by passing in the root node of the AST and traversing through the tree. When an expression node is reached, it is recursively called and every one of these expression nodes are evaluated to a type.

In hindsight, I could have done much better on this project, but given the split of our group and having to redo most of milestone 1, and the whole quarantine situation, it largely killed my motivation to invest my time into the project. Regardless, this project was excellent as it really enhanced my understanding of compilers, and my capabilities as a programmer in general.