# Mini Golang Compiler

Ashwin R Bharadwaj*, Athraya*, Sahazeer*
PES1201700003, PES1201700949, PES1201700951
* Department of Computer Science and Engineering, PES University

***Abstract*—In this project we aim to understand how a high level code is converted into machine level code that can be understood by a computer. The above process is carried out by a compiler. In this project we present a simple compiler. The code also builds a visual representation of the steps involved in the compilation process. The final output of the compiler is an assembly level code that is similar to ARM code.**

***Index Terms*—Golang, Compiler, switch, while**

## I. Introduction

A compiler is a computer program that translates computer code written in one programming language (the source language) into another language (the target language). The name compiler is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program.

Go is a statically typed, compiled programming language designed by Google. Go is syntactically similar to C, but with memory safety, garbage collection, structural typing and CSP-style concurrency.The language is often referred to as ”Golang” because of its domain name, golang.org. Go is an open source programming language that makes it easy to build simple, reliable, and efficient software. The project is split up into to six phases. Each phase is a step in the compilation process and is further discussed below.

## II. Phase 1

This is the first stage of the compiler. This phase deals with the generation of lexemes which are used to compile the code. The lexer, also called lexical analyzer or tokenizer, is a program that breaks down the input source code into a sequence of lexemes. It reads the input source code character by character, recognizes the lexemes and outputs a sequence of tokens describing the lexemes Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions. In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

## III. Phase 2

Programs submitted to a compiler often have errors of various kinds So, good compiler should be able to detect as many errors as possible in various ways and also recover from them (i.e) even in the presence of errors ,the compiler should scan the program and try to compile all of it.(error recovery). When the scanner or parser finds an error and cannot proceed , the compiler must modify the input so that the correct portions of the program can be pieced together and successfully processed in the syntax analysis phase. To achieve this functionality yacc is used to parse. The grammar which is used to verify if the given code is syntactically sound is obtained from ”https://golang.org/ref/spec”. The grammar obtained is modified as the project only focuses on ”while” loop and ”switch” statements. The project doesn’t concern itself on the ”import” statements but assumes all required libraries are already present. In the project an error recovery technique is also implemented called panic mode error recovery. If there is an inconsistency in the grammar the parser assumes that all the program skips all symbols till it reaches the symbol it was looking for called the synchronization symbol. The output of this phase is a rudimentary AST (abstract syntax tree) if the code is syntactically sound and an error message with the relevant details to find and rectify the error if the code given has errors present.

## IV. Phase 3

An abstract syntax tree (AST) is a way of representing the syntax of a programming language as a hierarchical tree-like structure. This structure is used for generating symbol tables for compilers and later code generation. The tree represents all of the constructs in the language and their subsequent rules.

An AST has several properties that aid the further steps of the compilation process:

- An AST can be edited and enhanced with information such as properties and annotations for every element it contains. Such editing and annotation is impossible with the source code of a program, since it would imply changing it.
- Compared to the source code, an AST does not include inessential punctuation and delimiters
- An AST usually contains extra information about the program, due to the consecutive stages of analysis by the compiler. For example, it may store the position of each element in the source code, allowing the compiler to print useful error messages.

In this project the syntax tree generated is displayed as a interactive web page. The data structure to store the AST was built in C++ and is a n array tree. Once the code has been successfully parsed the AST generated by the parser is read by a python script and is converted into a web page for easy viewing

## V. Phase 4

During the translation of a source program into the object code for a target machine, a compiler may generate a middle-level language code, which is known as intermediate code or intermediate text. There are many reasons why this is the case, but mainly it is done to streamline the process of machine-independent optimization. If the code is directly converted into machine code, there will be delays in cured as to execute a piece of code all the steps of a compiler must be performed, but if an intermediate code is generated only the final few steps need to repeated for each machine. The intermediate code keeps the analysis portion same for all the compilers that's why it doesn't need a full compiler for every unique machine. Intermediate code generator receives input from its predecessor phase and semantic analyzer phase. It takes input in the form of an annotated syntax tree. There are many different type of intermediate codes and they can be represented in many ways. In this project the IR is represented as Quadruples. Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. This phase outputs a un-optimized intermediate code.

## VI. Phase 5

Code optimization is an approach to enhance the performance of the code. The process involves eliminating the unwanted code lines and rearranging the statements of the code. Doing this decreases the execution time and also the memory needed by the program. As programmers are human, they are fallible and cannot be expected to write perfect code. Thus this step helps programmers write code worry free. In this project Constant folding and constant propagation are implemented as part of code optimization.

- *Constant Folding:* Expressions with constant operands can be evaluated at compile time, thus improving run-time performance and reducing code size by avoiding evaluation at compile-time.
- *Constant Propagation:* Constants assigned to a variable can be propagated through the flow graph and substituted at the use of the variable.

## VII. Phase 6

The code generation phase of a compiler translates the intermediate form into the target language. Given the information contained in the syntax tree, generating correct code is usually not a difficult task, but to generate code that will minimize the time and memory required is quite convoluted. The optimization of assembly code is not talked about in this project. In the project the optimized intermediate code is converted into assembly code that can be executed on a real machine. The project produces a sudo ARM assembly code where some op codes are different from those that are used in the machine. The code generator assumes there are 16 general purpose registers that are available to run the given code and the generator judicially uses the registers (any program can be exicuted using only three registers and unlimited memory and time). If all the registers are currently being used the generator performs a cntact switch and stores the code in main memory and retrieves it when it is needed.

The final output will be the assembly code along with the memory allocations for the declared varibles.

## VIII. *Challenges faced

This project was a good opportunity to learn the steps of a compiler and what actually happens when one runs a piece of code. In the first two phases the main challenges that were faced were to maintain the grammar as it was described and to manage all the corner cases in the grammar and error handling. The next main difficulty that was encountered was to build the AST. Many different types of data structures were implemented to build the AST with most of them having small bugs. To ameliorated the problem a n-array tree data structure was built to store the AST. The next main difficulty was to represent the AST, to do so a simple web page was implemented which would convert the AST into a visual format for easy viewing. The final problem that was faced was in the final stage of the compiler where the target code was being generated. To judiciously allocate the registers we used a simple algorithm. Using the algorithm any sized code can be executed using only 16 registers. The algorithm check for existence of free register, if found uses else frees a register based on its use age in the future. These were some the main problems we faced during the course of the project

[1] [2] [3] [4]

## References

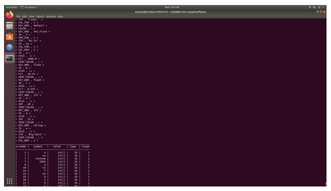
[1] G. Lang and S. DeBenedetti, "Angular correlation of annihilation radiation in various substances," *Physical Review*, vol. 108, no. 4, p. 914, 1957.

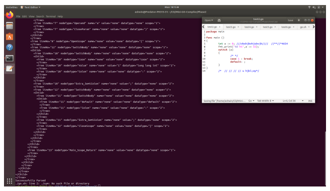
[2] J. R. Levine, J. Mason, J. R. Levine, T. Mason, D. Brown, J. R. Levine, and P. Levine, *Lex & yacc.* " O'Reilly Media, Inc.", 1992.

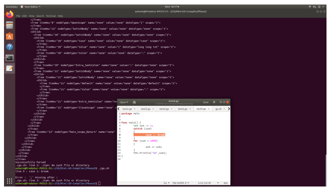
[3] M. E. Lesk and E. Schmidt, "Lex: A lexical analyzer generator," 1975.

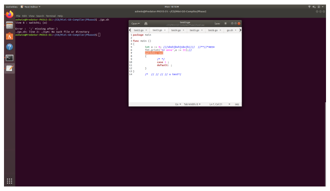
[4] B. Almende, B. Thieurmel, and T. Robert, "visnetwork: Network visualization using "vis. js" library," 2016.

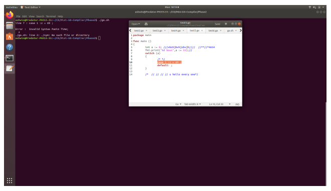
## IX. Output Screen shots

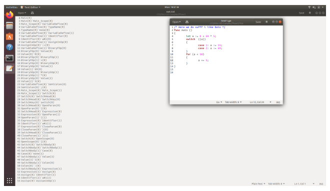

(a) Lexer

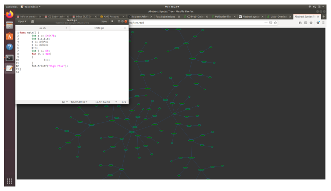

(b) Syntax Analyzer 1

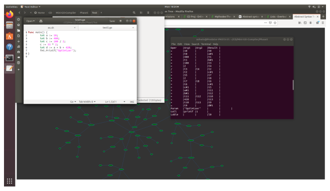

(c) Syntax Analyzer 2

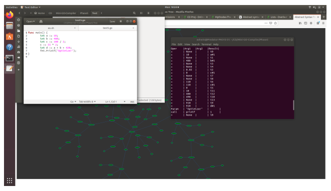

(d) Syntax Analyzer 3

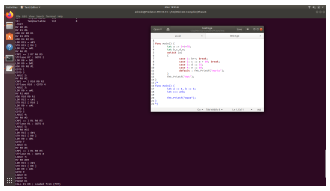

(e) Syntax Analyzer


(f) AST Beta


(g) AST


(h) AST


(i) Intermediate code


(j) Intermediate Optimize code

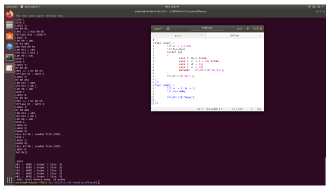

(k) Assembly Code

Fig. 1. Scatter Plots of various neural network models