

***Report on***

**Compiler Front-End for the Go programming language**

*Submitted in partial fulfilment of the requirements for* ***Semester VI***

***Compiler Design (UE18CS351)***

**Bachelor of Technology**

**in**

**Computer Science & Engineering**

***Submitted by:***

|  |  |
| --- | --- |
| **Aronya Baksy**  **Suhas Vasisht**  **Aditeya Baral** | **PES1201800002**  **PES1201800212**  **PES1201800366** |

*Under the guidance of*

|  |
| --- |
| **Mr. Kiran P**  Assistant Professor  PES University, Bengaluru |

**January – May 2021**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

FACULTY OF ENGINEERING

**PES UNIVERSITY**

(Established under Karnataka Act No. 16 of 2013)

100ft Ring Road, Bengaluru – 560 085, Karnataka, India

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **Chapter No.** | **Title** | **Page No.** |
|  | **INTRODUCTION** | **03** |
|  | **ARCHITECTURE OF LANGUAGE:** | **03** |
|  | **LITERATURE SURVEY** | **04** |
|  | **CONTEXT FREE GRAMMAR** | **04** |
|  | **DESIGN STRATEGY**   * **SYMBOL TABLE CREATION** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING** | **09** |
|  | **IMPLEMENTATION DETAILS (TOOL AND DATA STRUCTURES USED**   * **SYMBOL TABLE CREATION** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING** |  |
|  | **RESULTS AND possible shortcomings of your Mini-Compiler** |  |
|  | **SNAPSHOTS (of different outputs)** |  |
|  | **CONCLUSIONS** |  |
|  | **FURTHER ENHANCEMENTS** |  |
| **REFERENCES/BIBLIOGRAPHY** | |  |

# Introduction

The compiler is designed for the Go programming language (also sometimes referred to as Golang). Go is a statically typed, compiled programming language that features a C-like syntax but aims to be as readable and usable as more modern languages like Python.

The compiler designed takes in a valid path to a file containing source code in Golang. The compiler reads the source code, and undertakes the following steps on it:

* **Lexical Analysis:** Converts the source code into a stream of tokens. Tokens are defined in the lexer file lexer.py
* **Syntax Analysis:** Matches the input stream of tokens with the grammar rules defined in the file parser.py.
* **Semantic Analysis:** Using actions attached with the above defined grammar rules, transform the source code into a tree representation called an **Abstract Syntax Tree** and the associated **Three Address Code** (a machine-independent intermediate representation of the code)
* **Three Address Code:** Collect the Three Address Code generated by the semantic rules in the above step and put into a data structure that stores it in *quadruple* format.
* **Optimization:** Apply constant folding, common subexpression elimination and packing temporary optimizations to reduce the length and complexity of the generated Three Address Code.

# Architecture of the Language

The following subset of the syntax of Go has been handled by the grammar (syntax and semantic rules) implemented in this mini project:

* Short variable declarations (using type inference and the shorthand operator :=) are supported. Explicit type declarations (of the form var x int = 5) are *not* supported.
* Assignment statements involving shorthand operators (+=, -=, \*= and so on) are supported
* Arithmetic expressions involving the +, -, \* ,/ and % operators are supported
* Switch case expressions involving single variables are supported
* For loop statements containing a logical condition are supported
* Import statements, type definitions, constant declarations, array expressions are *not* handled

# Literature Survey

Sources:

* PLY (Python Lex Yacc) official documentation and examples (<https://ply.readthedocs.io/en/latest/>)
* BNF (Backus-Naur Form) representation of the Context-Free Grammar of the Go language (<https://golang.org/ref/spec>)
* Operator precedence rules in Go programming language (<https://www.tutorialspoint.com/go/go_operators_precedence.htm>)

# Context-Free Grammar

SourceFile : PACKAGE IDENTIFIER SEMICOLON ImportDeclList TopLevelDeclList

ImportDeclList : ImportDecl SEMICOLON ImportDeclList

| empty

TopLevelDeclList : TopLevelDecl SEMICOLON TopLevelDeclList

| empty

TopLevelDecl : Declaration

| FunctionDecl

ImportDecl : IMPORT LROUND ImportSpecList RROUND

| IMPORT ImportSpec

ImportSpecList : ImportSpec SEMICOLON ImportSpecList

| empty

ImportSpec : DOT string\_lit

| IDENTIFIER string\_lit

| empty string\_lit

Block : LCURLY ScopeStart StatementList ScopeEnd RCURLY

ScopeStart : empty

ScopeEnd : empty

StatementList : Statement SEMICOLON StatementList

| empty

Statement : Declaration

| SimpleStmt

| ReturnStmt

| Block

| IfStmt

| SwitchStmt

| ForStmt

| PrintIntStmt

| PrintStrStmt

Declaration : VarDecl

IdentifierList : IDENTIFIER COMMA IdentifierBotList

IdentifierBotList : IDENTIFIER COMMA IdentifierBotList

| IDENTIFIER

ExpressionList : Expression COMMA ExpressionBotList

ExpressionBotList : Expression COMMA ExpressionBotList

| Expression

Type : StandardTypes

StandardTypes : PREDEFINED\_TYPES

FunctionType : FUNC Signature

Signature : Parameters

| Parameters Result

Result : Parameters

| Type

Parameters : LROUND RROUND

| LROUND ParameterList RROUND

ParameterList : ParameterDecl

| ParameterList COMMA ParameterDecl

ParameterDecl : IdentifierList Type

| IDENTIFIER Type

| Type

SimpleStmt : Expression

| Assignment

| ShortVarDecl

| IncDecStmt

ShortVarDecl : ExpressionList ASSIGN\_OP ExpressionList

| Expression ASSIGN\_OP Expression

Assignment : ExpressionList assign\_op ExpressionList

| Expression assign\_op Expression

assign\_op : EQ

| PLUS\_EQ

| MINUS\_EQ

| OR\_EQ

| CARET\_EQ

| STAR\_EQ

| DIVIDE\_EQ

| MODULO\_EQ

| LS\_EQ

| RS\_EQ

| AMP\_EQ

| AND\_OR\_EQ

SwitchStmt : ExprSwitchStmt

ExprSwitchStmt : SWITCH SimpleStmt SEMICOLON LCURLY ScopeStart ExprCaseClauseList ScopeEnd RCURLY

| SWITCH SimpleStmt SEMICOLON Expression LCURLY ScopeStart ExprCaseClauseList ScopeEnd RCURLY

| SWITCH LCURLY ScopeStart ExprCaseClauseList ScopeEnd RCURLY

| SWITCH Expression LCURLY ScopeStart ExprCaseClauseList ScopeEnd RCURLY

ExprCaseClauseList : empty

| ExprCaseClauseList ExprCaseClause

ExprCaseClause : ExprSwitchCase COLON StatementList

ExprSwitchCase : CASE ExpressionList

| DEFAULT

| CASE Expression

ForStmt : FOR Expression Block

| FOR Block

ReturnStmt : RETURN

| RETURN Expression

| RETURN ExpressionList

Expression : UnaryExpr

| Expression OR\_OR Expression

| Expression AMP\_AMP Expression

| Expression EQ\_EQ Expression

| Expression NOT\_EQ Expression

| Expression LT Expression

| Expression LT\_EQ Expression

| Expression GT Expression

| Expression GT\_EQ Expression

| Expression PLUS Expression

| Expression MINUS Expression

| Expression OR Expression

| Expression CARET Expression

| Expression STAR Expression

| Expression DIVIDE Expression

| Expression MODULO Expression

| Expression LS Expression

| Expression RS Expression

| Expression AMP Expression

| Expression AND\_OR Expression

UnaryExpr : PrimaryExpr

| unary\_op UnaryExpr

unary\_op : PLUS

| MINUS

| NOT

| CARET

| STAR

| AMP

| LT\_MINUS

PrimaryExpr : Operand

| IDENTIFIER

| PrimaryExpr Selector

| PrimaryExpr Index

| PrimaryExpr Arguments

Operand : Literal

| LROUND Expression RROUND

Literal : BasicLit

BasicLit : decimal\_lit

| float\_lit

| string\_lit

decimal\_lit : DECIMAL\_LIT

float\_lit : FLOAT\_LIT

Arguments : LROUND RROUND

| LROUND ExpressionList RROUND

| LROUND Expression RROUND

| LROUND Type RROUND

| LROUND Type COMMA ExpressionList RROUND

| LROUND Type COMMA Expression RROUND

string\_lit : STRING\_LIT

# Design Strategy

## 5.1 Symbol Table Creation

The Symbol Table entries are created by the lexer. At this stage, the only field that is not empty is the symbol name.

The parser adds additional information about the identifier, such as its type, and its scope.

At the semantic stage, the expression result is evaluated, and the values are stored in the symbol table.

## 5.2 Intermediate Code Generation

The generation of three-address code is handled by action rules that are incorporated into the grammar. The scheme used for TAC generation is the **syntax directed translation scheme (SDTS)**. This means that the rules for generating a TAC are written in terms of the actual parser stack implementation and depend on the structure of the parsing stack.

## 5.3 Optimization

## The three-address code is optimized in a separate script once the entire intermediate code is generated from the input. The optimization script takes the complete symbol table, as well as the current generated three address code as input, and outputs the appropriate optimized three address code.

## 5.4 Error Handling

The following error handling strategies are in place:

* Lexer handles invalid identifiers by exiting as soon as an invalid identifier token is detected by the lexer. The program exits and no more processing is done after an invalid identifier is detected.
* Parser handles syntax errors using the p\_error production that is defined by PLY.
* Errors handled by the semantic analysis module:
  + Invalid number of identifiers on left- and right-hand sides of assignment statement
  + Invalid number of function arguments

# Implementation Details

## 6.1 Symbol Table Creation

The symbol table is an object of class SymbolTable. The symbol table contains a data member which is a list of objects of type SymbolTableNode. Both classes are defined in the file SymbolTable.py.

The SymbolTableNode class contains 4 data members, namely the Identifier Name, the type, the value, and the scope it was declared in.

The SymbolTable class contains an add method that can add a row to the symbol table, as well as a search method that searches the symbol table given the name of an identifier.

The searching is done using linear search, to avoid the additional overheads of maintaining the table in sorted order for Binary search.

To handle common subexpression elimination, the symbol table contains an expression field which is essentially a reference to an AST node that contains the same expression that has already been encountered. If a symbol table lookup for the expression returns a non-empty result, then the same AST node is reused, instead of a new AST node being generated.

## 6.2 Intermediate Code Generation

The process of intermediate code generation is linked to the process of generating the abstract syntax tree using the action rules included with the grammar productions.

The Intermediate Code is stored in a class called TAC that contains a single data member (a list of size 4 for the operator field, the result, and the two operands).

The TAC class contains methods for inserting a line into the three-address code and contains methods that generate new temporary variables and new label names as needed by the semantic analysis module.

The classes for the TAC and the AST nodes are defined in the file code.py.

The TAC object is associated with a particular AST node that is generated. Using the action rules, the AST is built using these nodes.

Each AST node contains a name, a data field, an input type (for leaf nodes only) and a Boolean variable indicating whether the node denotes an L-value or not.

Each AST node contains an object of type TAC that contains the three-address code corresponding to that node.

## 6.3 Code Optimization

The intermediate code optimization step takes in the symbol table and the final generated three-address code as input.

For constant folding and propagation, the symbol table is used to retrieve the evaluated values and the pointer to the already-evaluated subexpression. The result

## 6.4 Error Handling

Syntax errors in the parsing stage are handled by special rules in the p\_error production. The syntax error can be pointed out and the error message is printed that includes the line number of the syntax error.

In the lexical analysis module, the lexer is capable of detecting invalid identifiers and stopping the compilation process then, printing an error message to the screen with the line number on which the invalid identifier was encountered.

## 6.5 How to run

## On a Linux machine, the compiler can be run using the executable script bundled along with it. Using the command

./go-compile <path-to-source-file.go>

# Results

## 7.1 Final Result

The compiler front-end is shown to generate correct and optimized intermediate representation of the input high level program written in Golang. The three-address code is generated in quadruple format, and the symbol table entries are correctly filled in.

The compiler front-end designed entirely using Python Lex and Yacc is shown to work on both Windows and Linux environments, with minimal number of package dependencies, and with reasonable performance levels for medium-sized input programs.

## 7.2 Possible Shortcomings

## The final optimized code is highly likely to be slower and less efficient than the one generated by the actual reference implementation of the Golang compiler. Also, import statements and the associated features (such as reading from STDIN and writing to STDOUT) are not implemented. The unique features of the Go programming model such as channels and the concurrency model that involve advanced lower-level programming have also not been implemented.

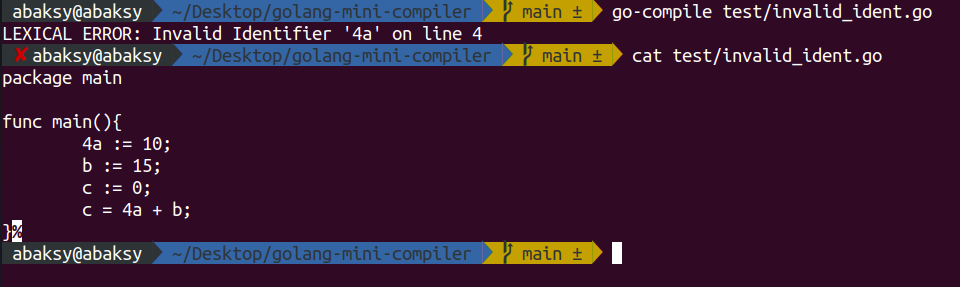
# Screenshots

# 

Output for a program with a switch-case statement

# 

Output for a program with a loop statement



Output for an invalid identifier

# 

Output for a program with common subexpressions

# Conclusions

We can conclude that a satisfactorily accurate compiler can be built using Lex and Yacc for several different languages spreading across multiple genres. We can conclude that the various phases of a standard compiler front-end can be built and implemented using these tools and by following all regulations, a standard compiler can be built for almost any language.

# Further Enhancements

Some possible further enhancements to this mini project are as follows:

* Implementation of syntax and semantic rules for conditional and unconditional branch statements (if, break, goto)
* More optimization techniques based on dead code elimination (implementing live-variable analysis and adding next-use information to the symbol table)