Report on

"Mini Compiler for GoLang"

*Submitted in partial fulfillment of the requirements for **Sem VI***

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Vishal Bharadwaj	**PES1201800014**
Aravind M Subramanya	**PES1201800033**
Rithik R Mali	**PES1201800300**

Under the guidance of

Prof. Madhura V
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

1. Introduction

This project is a mini compiler for Go Language. The intermediate code generation is being done with the help of LEX and YACC tools. The compiled LEX and YACC files takes the input from a “.go” file, parses it, generates a symbol table and finally gives out a “.txt” file containing the intermediate code.

This intermediate code is then passed to the code optimisation module which is coded in python.

The optimization is done by identifying the tokens and acting based on the matched regex.

We perform the following code optimization techniques in order and the output of the previous operation is passed on the next operation:

1. Strength Reduction
2. Constant Propagation
3. Constant Folding
4. Copy Propagation
5. Dead Code elimination

2. Architecture of Language

We have implemented the following constructs in our compiler:

- For loop
- Nested for loop.
- Switch Statements
- Nested Switch Statements

In addition to those we have also taken care of the following

- Variable assignment (including const)
- Multiple variable assignment in a single line
- Arithmetic operations
- Boolean expressions
- Error if const variable is redeclared.
- Error if number of values do not match the number of variables in multiple variable assignment.
- If any other error is there it will give the corresponding line number along with “syntax error” message

Syntax is handled by YACC where grammar rules are specified for the entire language.

3. LITERATURE SURVEY

We referred to the official GoLang documentation [1] for building our lexer and for the grammar.

They have provided the lexical elements and the rules for the grammar followed in GoLang.

4. CONTEXT FREE GRAMMAR

```
s:
    PackageClause ImportDeclarations TopLevelDeclarations
;
PackageClause:
    KEYWORD_PACKAGE IDENTIFIER
;
ImportDeclarations:
    ImportDeclaration ImportDeclarations %prec NORMAL
    | empty %prec EMPTY
;
ImportDeclaration:
    KEYWORD_IMPORT ImportSpecification
    | KEYWORD_IMPORT '(' ImportSpecificationList ')'
;
ImportSpecificationList:
    ImportSpecification ImportSpecificationList2
;

ImportSpecificationList2:
    ImportSpecification ImportSpecificationList2 %prec NORMAL
    | %empty %prec EMPTY
;

ImportSpecification:
    STRING_LITERAL
    | '.' STRING_LITERAL
    | IDENTIFIER STRING_LITERAL
;

/* Top Level Declaration */
TopLevelDeclarations:
    TopLevelDeclaration TopLevelDeclarations %prec NORMAL
    | %empty %prec EMPTY
;

TopLevelDeclaration:
```

```
    Declaration
    | KEYWORD_FUNC IDENTIFIER '(' ')' CodeBlock
;

Declaration:
    ConstDeclaration
    | TypeDeclaration
    | VariableDeclaration
;

ConstDeclaration:
    KEYWORD_CONST ConstSpecification
    | KEYWORD_CONST '(' ConstSpecifications ')'
;

ConstSpecifications:
    ConstSpecification
    | ConstSpecifications ConstSpecification
;

ConstSpecification:
    IdentifierList ConstIdList
;

ConstIdList:
    PreConstIdList '=' ExpressionList
;
PreConstIdList:
    Type %prec NORMAL
    | %empty %prec EMPTY
;
TypeDeclaration:
    KEYWORD_TYPE TypeSpecifications
;

TypeSpecifications:
    TypeSpecification
    | '(' TypeSpecificationList ')'
;
```

```
TypeSpecificationList:
    TypeSpecification TypeSpecificationList %prec NORMAL
    | %empty %prec EMPTY
;

TypeSpecification:
    AliasDeclaration
    | TypeDefinition
;

AliasDeclaration:
    IDENTIFIER '=' Type
;

TypeDefinition:
    IDENTIFIER Type
;

VariableDeclaration:
    KEYWORD_VAR VariableSpecification
    | KEYWORD_VAR '(' VariableSpecifications ')'
;

VariableSpecifications:
    VariableSpecification
    | VariableSpecifications VariableSpecification
;

VariableSpecification:
    IdentifierList VariableIdList
;

VariableIdList:
    Type VariableIdListType
    | '=' ExpressionList
;
```

```
VariableIdListType:
    '=' ExpressionList %prec NORMAL
    | %empty %prec EMPTY
;

IdentifierList:
    IDENTIFIER
    | IDENTIFIER ',' IdentifierList
;

Type:
    TypeName
    | '(' Type ')'
;

TypeName:
    P_TYPE
    | QualifiedID
;

QualifiedID:
    IDENTIFIER '.' IDENTIFIER
;

ExpressionList:
    Expression
    | ExpressionList ',' Expression
;

Expression:
    Expression LOGICAL_OR Expression
    | Expression LOGICAL_AND Expression
    | Expression RelationalOperation Expression %prec REL_EQUAL
    | Expression AddOperation Expression %prec '+'
    | Expression MultipyOperation Expression %prec '-'
    | UnaryExpression %prec P_UNARY
;
```

```
RelationalOperation:
    REL_EQUAL
    | REL_NEQUAL
    | REL_LT
    | REL_LEQ
    | REL_GT
    | REL_GEQ
;

AddOperation:
    '+'
    | '-'
    | '|'
    | '^'
;

MultipyOperation:
    '*'
    | '/'
    | '%'
    | LSHIFT
    | RSHIFT
    | '&'
    | AMPXOR
;

UnaryOperation:
    '+'
    | '-'
    | '!'
    | '^'
    | '*'
    | '&'
;

AssignOperation:
    ADD_ASSIGN
    | SUB_ASSIGN
    | OR_ASSIGN
```

```
    | XOR_ASSIGN
    | MUL_ASSIGN
    | DIV_ASSIGN
    | MOD_ASSIGN
    | AND_ASSIGN
    | LSHIFT_ASSIGN
    | RSHIFT_ASSIGN
    | AMPXOR_ASSIGN
;

UnaryExpression:
    CHANNEL_ASSIGN UnaryExpression
    | UnaryOperation UnaryExpression %prec P_UNARY
    | PrimaryExpression
;

PrimaryExpression:
    Operand
    | PrimaryExpression Index
;

Index:
    '[' Expression ']'
;

Operand:
    Literal
    | OperandName
    | '(' Expression ')'
    | P_NIL
    | P_BOOL
;

Literal:
    BasicLiteral
;

BasicLiteral:
    INT_LITERAL
```

```
    | FLOAT_LITERAL
    | STRING_LITERAL
;

OperandName:
    IDENTIFIER
    | P_FUNCTION
    | QualifiedID
;

CodeBlock:
    '{'
    Statements
    '}'
;

Statements:
    Statements Statement %prec NORMAL
    | %empty %prec EMPTY
;

Statement:
    SimpleStatement
    | Declaration
    | ForStatement
    | SwitchStatement
;

SimpleStatement:
    Expression
    | IncrementDecrement
    | Assignment
    | ShortDeclaration
;

IncrementDecrement:
    Expression INC_ASSIGN
    | Expression DEC_ASSIGN
;
```

```
Assignment:
    IdentifierList '=' ExpressionList
    | Expression AssignOperation Expression
;

ShortDeclaration:
    IdentifierList SHORT_DECLARATION ExpressionList
;

ForStatement:
    KEYWORD_FOR
    ForClauseParent
    CodeBlock
;

ForClauseParent:
    ForCondition %prec NORMAL
    | ForClause %prec NORMAL
    | RangeClause %prec NORMAL
    | empty %prec EMPTY
;

ForCondition:
    Expression
;

ForClause:
    ForClauseInit
    ';'
    ForClauseCondition
    ';'
    ForClauseUpdation
;

ForClauseInit:
    InitializeStatement %prec NORMAL
    | empty %prec EMPTY
;
```

```
ForClauseCondition:
    ForCondition
;

ForClauseUpdation:
    UpdationStatement %prec NORMAL
    | empty %prec EMPTY
;

InitializeStatement:
    SimpleStatement
;

UpdationStatement:
    SimpleStatement
;

RangeClause:
    PreForRange KEYWORD_RANGE Expression
;

PreForRange:
    ExpressionList '=' %prec NORMAL
    | IdentifierList SHORT_DECLARATION %prec NORMAL
    | empty %prec EMPTY
;

SwitchStatement:
    ExprSwitchStmt
    | TypeSwitchStmt
;

ExprSwitchStmt:
    KEYWORD_SWITCH SimpleStatement '{' ExprCaseClauses '}'
;
ExprCaseClauses:
    ExprCaseClauses ExprCaseClause
    | ExprCaseClause
;
```

```
ExprCaseClause:
    ExprSwitchCase ':' Statements
;

ExprSwitchCase:
    KEYWORD_CASE ExpressionList
    | KEYWORD_DEFAULT
;

TypeSwitchStmt:
    KEYWORD_SWITCH TypeSwitchGuard '{' TypeCaseClauses '}'
;

TypeSwitchGuard:
   IDENTIFIER SHORT_DECLARATION IDENTIFIER '.' '(' KEYWORD_TYPE ')'
   | IDENTIFIER SHORT_DECLARATION Literal '.' '(' KEYWORD_TYPE ')'
;

TypeCaseClauses:
    TypeCaseClauses TypeCaseClause
    | TypeCaseClause
;

TypeCaseClause:
    TypeSwitchCase ':' Statements
;

TypeSwitchCase:
    KEYWORD_CASE TypeList
    | KEYWORD_DEFAULT
;

TypeList:
    Type
    | Type ',' TypeList
;

empty: %empty;
```

5. DESIGN STRATEGY

- **SYMBOL TABLE CREATION**

  The Symbol Table in our compiler stores the name of the declared variable, its data type (inferred if not defined), its value (most updated value), the details about the scope in which the variable was declared.

  We have implemented our symbol table with the help of hashing.

  We hash the token name to a table by applying the hash function character wise to obtain the final position. Collision resolution is done by linear probing.

```
 currValue=(currValue*128+inputToHash[i])%1000000000; // for each
char
 currValue = currValue % SYMBOL_TABLE_MAX;
```

- **INTERMEDIATE CODE GENERATION**

  We have functions to create labels and temporary variables and have maintained global variables to keep track of the most recently created temp and label. For the constructs, switch and for, we have used a stack and queue structure respectively for the generation of intermediate code.

- **CODE OPTIMIZATION**

  For the ICG generated, we have implemented algebraic identity, strength reduction, constant propagation, constant folding, copy propagation and dead code elimination. We perform these optimizations in that order and the final output optimized code is printed.

- **ERROR HANDLING**

We are handling syntax errors, which are generated during parsing. We are also handling re-declaration of variables in the relevant scope, and are showing the appropriate error messages. We have also handled imbalanced assignment errors, type mismatch errors, and reassignment of constant variables. We stop parsing the input on encountering these errors and display the line number of the errors, and a short description of the error.

6. IMPLEMENTATION DETAILS

Symbol Table

```
struct symbolTableStructure {
  char name[31];
  char declaredVariableType[10];
  char type;
  char value[20];
  int scope_depth;
  int hcode;
  int scope_id;
};
stack iStack;
stack vStack;
stack scopeStack;
stack tStack;
```

Functions Implemented:

```
int firstHashFunction(char *inputToHash)

int hashTableIndexReturn(char *inputToHash)

void appendHashTable(char type, char *inputToHash, char
*declaredVariableType, char *value, int scope_depth, int scope_id)

char *findFromHashTable(char *inputToHash)
```

```
void updateHashTable(char *inputToHash, char *declaredVariableType,
char *value)

void PresentIdentifierAssignment(Node *LeftHandSide, Node
*RightHandSide)

void IdentifierAssignment(char declaredType, Node *LeftHandSide, Node
*RightHandSide)

int getNextScope(int scope_depth)

int getPreviousScope(int scope_depth)

int scopeOfParentCheck(char *inputToHash)

void displaySymbolTable()
```

Intermediate Code Generation

```
char resString[32];
char tempArray[20] = "";
int tempVariableCount = 0;
int labelCount = 0;
char tempVariableStore[32];
char labelStore[32];
FILE* intermediateCodeFile;
queue forLoopQueue
stack switchConditionStack
```

Functions Implemented

```
char *popFromStack(stack *stackPointer)

void pushToStack(stack *stackPointer, char *stackItem)

int isStackEmpty(stack st)
```

```
void insertToQueue(queue *stackPointer, char *stackItem)

char *removeFromQueue(queue *stackPointer)

void createTempVariable()

void createLabel()
```

Code Optimization

re - for regex matching in python

Functions Implemented

```
def algID(tokens):
def constantFP(list_of_lines):
def constantFolding(list_of_lines, comp=[]):
def constantProp(list_of_lines,comp=[]):
def strengthRed(list_of_lines, comp=[]):
def copyProp(list_of_lines, comp=[]):
def variableAssgnCheck(list_of_lines, token, line_no):
def deadCodeRemove(list_of_lines):
```

7. RESULTS AND POSSIBLE SHORTCOMINGS OF YOUR MINI-COMPILER

We were able to successfully build the Mini Compiler for GoLang for the constructs: for and switch. Our grammar covers all different possibilities and cases we have also handled multiple different types of possible errors that could occur. We generate the ICG and have performed various code optimizations to generate the final code.

The shortcomings of our mini-compiler include:

- not recording temp variables used for intermediate code in the symbol table

- not taking care of function identification / function calls
- not converting function calls like print to intermediate code

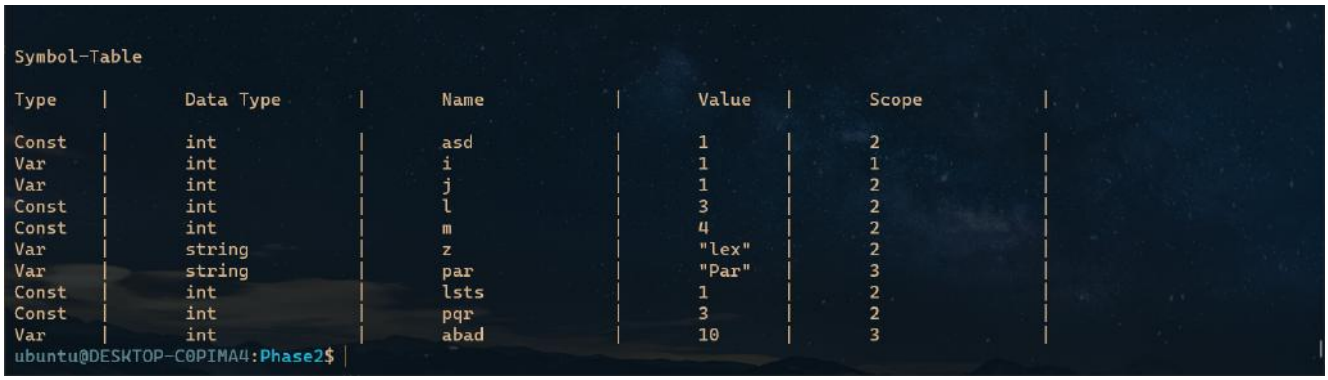
8. SNAPSHOTS

For Loop and Switch

```
// lskjgdmsfsld.mkfs
// dfjlsdm
// sjldkmv
// kjnlkm
// kbjnlmk

/*sdkjlnvkm
jhkbjnk;ml
hgjkm
gvjhbkjl
ghkbjnlkm
jgvhbkjnl
jhbkjnl*/
package main

import "fmt"

func main()
{
    // var b, c int = 1, 2,12,12
     for i := 1; i < 5; i++
     {
         var z = "lex"
         const l,m,pqr,lsts,asd= 1,3,4,3,1
         for j := 1; j < 5; j++
        {
            fmt.Println("I'm a Nested for")
            abad:=10
            par:="Par"
        }                       //Grammer Accepted

    }
```

```
    switch i {
    case 1:
        fmt.Println("one")
    case 2:
        fmt.Println("two")
    case 3:
        fmt.Println("three")
    }

    switch t := ooo.(type) {
    case bool:
        fmt.Println("I'm a bool")
        switch i {
        case 1:
            fmt.Println("one")
        case 2:
            fmt.Println("two")
        case 3:
            fmt.Println("three")
        }
    case int:
        fmt.Println("I'm an int")
    default:
        fmt.Printf("Don't know type")
    }                                                   //Grammar Accepted
}
```

Symbol Table:



Symbol-Table

Type	Data Type	Name	Value	Scope
Const	int	asd	1	2
Var	int	i	1	1
Var	int	j	1	2
Const	int	l	3	2
Const	int	m	4	2
Var	string	z	"lex"	2
Var	string	par	"Par"	3
Const	int	lsts	1	2
Const	int	pqr	3	2
Var	int	abad	10	3

ubuntu@DESKTOP-C0PIMA4:Phase2$

Intermediate Code Generation

```
package main

import "fmt"

func main() {
    maya:="Aravind"
    maya="Hello"
    var a,b, pqr int = 1,10, 1
    b=30*2/92+10
    b=20
    aravind:=100
    aravind=20
    for pqr=1;pqr<10;pqr++
    {
        a = 4 + 1
        pqr=3+1
    }
    i:=2
    cas:="Cas0"
    switch i
    {
    case 1:
        cas="Cas1"
    case 2:
        cas="Cas2"
    case 3:
        cas="Cas3"
    case 4:
        cas="Cas4"
    case 5:
        cas="Cas5"
    case 6:
        cas="Cas6"
    }
}
```

Intermediate Code

```
maya = "Aravind"
maya = "Hello"
a = 1
b = 10
pqr = 1
_t0 = 2 / 92
_t1 = 30 * _t0
_t2 = _t1 + 10
b = _t2
b = 20
aravind = 100
aravind = 20
pqr = 1
L0:
_t3 = pqr < 10
IFFALSE _t3 GOTO L1
GOTO L2
L3:
_t4 = pqr + 1
pqr = _t4
GOTO L0
L2:
_t5 = 4 + 1
a = _t5
_t6 = 3 + 1
pqr = _t6
GOTO L3
L1:
i = 2
cas = "Cas0"
_t7 = i != 1
IFFALSE _t7 GOTO L4
cas = "Cas1"
L4:
_t8 = i != 2
IFFALSE _t8 GOTO L5
cas = "Cas2"
L5:
_t9 = i != 3
```

```
IFFALSE _t9 GOTO L6
cas = "Cas3"
L6:
_t10 = i != 4
IFFALSE _t10 GOTO L7
cas = "Cas4"
L7:
_t11 = i != 5
IFFALSE _t11 GOTO L8
cas = "Cas5"
L8:
_t12 = i != 6
IFFALSE _t12 GOTO L9
cas = "Cas6"
L9:
```

Output of Code Optimization

```
('--------------------------', 'ICG', '--------------------------')
maya = "Aravind"
maya = "Hello"
a = 1
b = 10
pqr = 1
_t0 = 2 / 92
_t1 = 30 * _t0
_t2 = _t1 + 10
b = _t2
b = 20
aravind = 100
aravind = 20
pqr = 1
L0:
_t3 = pqr < 10
IFFALSE _t3 GOTO L1
GOTO L2
L3:
_t4 = pqr + 1
pqr = _t4
GOTO L0
L2:
_t5 = 4 + 1
```

```
a = _t5
_t6 = 3 + 1
pqr = _t6
GOTO L3
L1:
i = 2
cas = "Cas0"
_t7 = i != 1
IFFALSE _t7 GOTO L4
cas = "Cas1"
L4:
_t8 = i != 2
IFFALSE _t8 GOTO L5
cas = "Cas2"
L5:
_t9 = i != 3
IFFALSE _t9 GOTO L6
cas = "Cas3"
L6:
_t10 = i != 4
IFFALSE _t10 GOTO L7
cas = "Cas4"
L7:
_t11 = i != 5
IFFALSE _t11 GOTO L8
cas = "Cas5"
L8:
_t12 = i != 6
IFFALSE _t12 GOTO L9
cas = "Cas6"
L9:

('----------------', 'Strength Reduction Done', '----------------')
maya = "Aravind"
maya = "Hello"
a = 1
b = 10
pqr = 1
_t0 = 2 / 92
_t1 = 30 * _t0
_t2 = _t1 + 10
b = _t2
b = 20
aravind = 100
aravind = 20
pqr = 1
L0:
```

```
_t3 = pqr < 10
IFFALSE _t3 GOTO L1
GOTO L2
L3:
_t4 = pqr + 1
pqr = _t4
GOTO L0
L2:
_t5 = 4 + 1
a = _t5
_t6 = 3 + 1
pqr = _t6
GOTO L3
L1:
i = 2
cas = "Cas0"
_t7 = i != 1
IFFALSE _t7 GOTO L4
cas = "Cas1"
L4:
_t8 = i != 2
IFFALSE _t8 GOTO L5
cas = "Cas2"
L5:
_t9 = i != 3
IFFALSE _t9 GOTO L6
cas = "Cas3"
L6:
_t10 = i != 4
IFFALSE _t10 GOTO L7
cas = "Cas4"
L7:
_t11 = i != 5
IFFALSE _t11 GOTO L8
cas = "Cas5"
L8:
_t12 = i != 6
IFFALSE _t12 GOTO L9
cas = "Cas6"
L9:

('-----', 'Constant Propagation and Constant Folding Done', '-----')
maya = "Aravind"
maya = "Hello"
a = 1
b = 10
pqr = 1
```

```
_t0 = 0
_t1 = 0
_t2 = 10
b = 10
b = 20
aravind = 100
aravind = 20
pqr = 1
L0:
_t3 = True
IFFALSE _t3 GOTO L1
GOTO L2
L3:
_t4 = 2
pqr = 2
GOTO L0
L2:
_t5 = 5
a = 5
_t6 = 4
pqr = 4
GOTO L3
L1:
i = 2
cas = "Cas0"
_t7 = True
IFFALSE _t7 GOTO L4
cas = "Cas1"
L4:
_t8 = False
IFFALSE _t8 GOTO L5
cas = "Cas2"
L5:
_t9 = True
IFFALSE _t9 GOTO L6
cas = "Cas3"
L6:
_t10 = True
IFFALSE _t10 GOTO L7
cas = "Cas4"
L7:
_t11 = True
IFFALSE _t11 GOTO L8
cas = "Cas5"
L8:
_t12 = True
IFFALSE _t12 GOTO L9
```

```
cas = "Cas6"
L9:

('--------------------', 'Copy Propagation Done', '--------------------')
maya = "Aravind"
maya = "Hello"
a = 1
b = 10
pqr = 1
_t0 = 0
_t1 = 0
_t2 = 10
b = 10
b = 20
aravind = 100
aravind = 20
pqr = 1
L0:
_t3 = True
IFFALSE True GOTO L1
GOTO L2
L3:
_t4 = 2
pqr = 2
GOTO L0
L2:
_t5 = 5
a = 5
_t6 = 4
pqr = 4
GOTO L3
L1:
i = 2
cas = "Cas0"
_t7 = True
IFFALSE True GOTO L4
cas = "Cas1"
L4:
_t8 = False
IFFALSE False GOTO L5
cas = "Cas2"
L5:
_t9 = True
IFFALSE True GOTO L6
cas = "Cas3"
L6:
_t10 = True
```

```
IFFALSE True GOTO L7
cas = "Cas4"
L7:
_t11 = True
IFFALSE True GOTO L8
cas = "Cas5"
L8:
_t12 = True
IFFALSE True GOTO L9
cas = "Cas6"
L9:

('--------------------', 'Dead Code Elimination Done', '--------------------')
L0:
IFFALSE True GOTO L1
GOTO L2
L3:
GOTO L0
L2:
GOTO L3
L1:
IFFALSE True GOTO L4
L4:
IFFALSE False GOTO L5
L5:
IFFALSE True GOTO L6
L6:
IFFALSE True GOTO L7
L7:
IFFALSE True GOTO L8
L8:
IFFALSE True GOTO L9
L9:
```

9. Conclusions

Throughout the duration of building the mini compiler, we have understood the basics of designing a compiler in LEX & YACC and creating a symbol table and output intermediate code in target language. We have also familiarized ourselves with the regex which helped us in coding the rules in LEX and identifying tokens/expressions for Code optimisation. There was also a good amount of exposure to use of CFG in real world applications.

10. Future Enhancements

This compiler could be extended to account for all the constructs supported by Go Lang such as if, if-else, while, user-defined functions etc.

REFERENCES/BIBLIOGRAPHY

[1] https://golang.org/ref/spec