*Report on*

# Mini Compiler for GoLang

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

***Compiler Design***

**Bachelor of Technology**
**in**
**Computer Science & Engineering**

***Submitted by:***

|  |  |
|---|---|
| **GBS Akhil** | **PES1201800188** |
| **Sreejesh Saya** | **PES1201800293** |
| **Navaneeth M** | **PES1201801597** |

*Under the guidance of*

**H.B. Mahesh**
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 focuses on designing a mini-compiler for the Go Lang programming language. The end-product of this project, a compiler generates **optimized** intermediate code for the given Go code.
It works for Go Lang's looping constructs – for and while loop.

This is done using the following steps:

- Generate symbol table after performing expression evaluation
- Generate Abstract Syntax Tree for the code
- Generate 3 address code followed by corresponding quadruples
- Perform Code Optimization

The main tools used in the project include LEX which identifies pre-defined patterns and generates tokens for the patterns matched and YACC which parses the input for semantic meaning and generates an abstract syntax tree and intermediate code for the source code.

Appropriate screenshots of the input program files and associated output at every stage of the compiler have been provided at the end of this document.

## 2. ARCHITECTURE OF LANGUAGE

Go Lang constructs implemented

- For loop
- While loop

We have handled compiling Golang programs having basic assignment statements, arithmetic expressions. Updating of the variables from these expressions take place in the symbol table. Undeclared variables and redeclaration of variables will be flagged as errors along with the line numbers of the error.

## 3. LITERATURE SURVEY AND REFERENCES

- https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf
- https://www.epaperpress.com/lexandyacc/attr.html
- https://cs.gmu.edu/~white/CS540/Slides/Semantic/CS540-2-lecture6.pdf

## 4. CONTEXT FREE GRAMMAR

```
PROG : T_Package T_Main Stmts MAIN Stmts
    {
             cout << "\nValid Program\n\n";
             $$.code = $3.code + "\n" + $4.code + "\n" + $5.code + "\n";
             ofstream ICG("./ICG_QUAD/ICG.txt");
             ICG << $$.code;
             ICG.close();
    }
    ;

MAIN : T_Func T_Main T_Paren '{' { ++scope; } Stmts '}' { --scope; }
    {
             $$.code = $6.code + "\n";
    }
    ;

Stmts : { $$.code = ""; }
    | DECL Stmts
      { $$.code = $1.code + "\n" + $2.code + "\n"; }
    | ASSIGN ';' Stmts
      { $$.code = $1.code + "\n" + $3.code + "\n"; }
    | UNARY_EXPR ';' Stmts
      { $$.code = $1.code + "\n" + $3.code + "\n"; }
    | LOOP Stmts
      { $$.code = $1.code + "\n" + $2.code + "\n"; }
    ;

TYPE : T_Int
    | T_Float
    | T_Double
    | T_Bool
    | T_String
    ;

VALUE : EXPR
      {
             $$.val = $1.val;
             $$.addr = $1.addr;
             $$.code = $1.code + "\n";
      }
    | UNARY_EXPR
      {
             $$.val = $1.val;
```

```
       }
    | T_String
       {
             $$.strval = $1.strval;
       }
    ;

DECL : T_Var LISTVAR TYPE '=' LISTVALUE ';'
       {
             if(vars.size() != vals.size())
             {
                   yyerror("Mismatch in line " + to_string(yylineno) + ": " +
to_string(vars.size()) + " variables " + to_string(vals.size()) + " values");
                   exit(1);
             }
             for(int i = 0; i < vars.size(); i++)
             {
                   int lineno;
                   if(lineno = check_decl(vars[i], scope))
                   {
                         yyerror(vars[i] + " redeclared in line " + to_string(yylineno)
+ "\nPrevious declaration in line " + to_string(lineno));
                         exit(1);
                   }
                   $$.code += codes[i] + "\n" + vars[i] + " = " + addrs[i] + "\n";
                   insert(vars[i], $3.strval, yylineno, vals[i], scope);
             }
             vars.clear();
             vals.clear();
             codes.clear();
             addrs.clear();
    }
    | T_Var LISTVAR TYPE ';'
       {
             for(int i = 0; i < vars.size(); i++)
             {
                   int lineno;
                   if(lineno = check_decl(vars[i], scope))
                   {
                         yyerror(vars[i] + " redeclared in line " + to_string(yylineno)
+ "\nPrevious declaration in line " + to_string(lineno));
                         exit(1);
                   }
                   insert(vars[i], $3.strval, yylineno, 0, scope);
             }
             vars.clear();
    }
    ;

LISTVAR    : T_Id    { vars.push_back($1.strval); }
       | T_Id { vars.push_back($1.strval); } ',' LISTVAR
       ;

LISTVALUE : VALUE { vals.push_back($1.val); codes.push_back($1.code);
addrs.push_back($1.addr); }
             | VALUE { vals.push_back($1.val); codes.push_back($1.code);
addrs.push_back($1.addr); } ',' LISTVALUE
             ;

ASSIGN : T_Id '=' VALUE
             {
                   if(not check_decl($1.strval, scope))
                   {
                         yyerror($1.strval + " not declared in line " +
to_string(yylineno));
                         exit(1);
                   }
                   $$.code = $3.code + "\n" + $1.strval + " = " + $3.addr + "\n";
             }
       | T_Id T_Assgnop VALUE
             {
                   if(not check_decl($1.strval, scope))
                   {
                         yyerror($1.strval + " not declared in line " +
to_string(yylineno));
                         exit(1);
                   }
                   $$.code = $1.strval + " = " + $1.strval + " " + $2.strval + " " +
to_string($3.val) + "\n";
             }
       ;

EXPR : BOOL_EXPR
       {
             $$.code = $1.code + "\n";
       }
    | ARITH_EXPR
       {
             $$.val = $1.val;
             $$.addr = $1.addr;
             $$.code = $1.code + "\n";
       }
       ;
```

```
ARITH_EXPR     : ARITH_EXPR '+' T
       {
             $$.val = $1.val + $3.val;
             $$.addr = newtemp();
             $$.code = $1.code + "\n" + $3.code + "\n" + $$.addr + " = " + $1.addr + "
+ " + $3.addr + "\n";
       }
     | ARITH_EXPR '-' T
       {
             $$.val = $1.val - $3.val;
             $$.addr = newtemp();
             $$.code = $1.code + "\n" + $3.code + "\n" + $$.addr + " = " + $1.addr + "
- " + $3.addr + "\n";
       }
     | T
       {
             $$.val = $1.val;
             $$.addr = $1.addr;
             $$.code = $1.code + "\n";
       }
       ;

T : T '*' F
       {
             $$.val = $1.val * $3.val;
             $$.addr = newtemp();
             $$.code = $1.code + "\n" + $3.code + "\n" + $$.addr + " = " + $1.addr + "
* " + $3.addr + "\n";
       }
     | T '/' F
       {
             $$.val = $1.val / $3.val;
             $$.addr = newtemp();
             $$.code = $1.code + "\n" + $3.code + "\n" + $$.addr + " = " + $1.addr + "
/ " + $3.addr + "\n";
       }
     | T '%' F
     | F
       {
             $$.val = $1.val;
             $$.addr = $1.addr;
             $$.code = $1.code + "\n";
       }
       ;

F : '-' T_Num
       {
             $$.val = -$2.val;
             $$.addr = newtemp();
             $$.code = $$.addr + " = - " + to_string($2.val) + "\n";
       }
    | '-' T_Id
       {
             if(not check_decl($2.strval, scope))
             {
                   yyerror($2.strval + " not declared in line " + to_string(yylineno));
                   exit(1);
             }
             $$.val = -$2.val;
             $$.addr = newtemp();
             $$.code = $$.addr + " = - " + $2.strval + "\n";
       }
    | T_Num
       {
             $$.val = $1.val;
             $$.addr = newtemp();
             $$.code = $$.addr + " = " + to_string($1.val) + "\n";
       }
    | T_Id
       {
             if(not check_decl($1.strval, scope))
             {
                   yyerror($1.strval + " not declared in line " + to_string(yylineno));
                   exit(1);
             }
             $$.val = $1.val;
             $$.addr = newtemp();
             $$.code = $$.addr + " = " + $1.strval + "\n";
       }
    | '(' ARITH_EXPR ')'
       {
             $$.val = $2.val;
             $$.addr = $2.addr;
             $$.code = $2.code + "\n";
       }
       ;

BOOL_EXPR : LOGICAL
    | RELATIONAL
       {
             $$.addr = $1.addr;
             $$.code = $1.code + "\n";
       }
       ;
```

```
RELATIONAL : ARITH_EXPR T_Relop ARITH_EXPR
    {
       $$.addr = newtemp();
       $$.code = $1.code + "\n" + $3.code + "\n" + $$.addr + " = " + $1.addr + " " +
$2.strval + " " + $3.addr + "\n";
    }
    | '(' RELATIONAL ')'
    ;

LOGICAL    : LOGICAL T_Or X | X
    ;

X : X T_And Y | Y
    ;

Y : '!' Y | Z
    ;

Z : T_True
    | T_False
    ;

UNARY_EXPR : T_Id T_Inc
       {
             if(not check_decl($1.strval, scope))
             {
                   yyerror($1.strval + " not declared in line " + to_string(yylineno));
                   exit(1);
             }
             $$.code = $1.strval + " = " + $1.strval + " + 1\n";
       }
    | T_Inc T_Id
       {
             if(not check_decl($2.strval, scope))
             {
                   yyerror($2.strval + " not declared in line " + to_string(yylineno));
                   exit(1);
             }
             $$.code = $1.strval + " = " + $1.strval + " + 1\n";
       }
    | T_Id T_Dec
       {
             if(not check_decl($1.strval, scope))
             {
                   yyerror($1.strval + " not declared in line " + to_string(yylineno));
                   exit(1);
```

```
            }
            $$.code = $1.strval + " = " + $1.strval + " - 1\n";
      }
    | T_Dec T_Id
      {
            if(not check_decl($2.strval, scope))
            {
                  yyerror($2.strval + " not declared in line " + to_string(yylineno));
                  exit(1);
            }
            $$.code = $1.strval + " = " + $1.strval + " - 1\n";
      }
    ;

LOOP : FOR
      {
            $$.code = $1.code + "\n";
      }
    | WHILE
      {
            $$.code = $1.code + "\n";
      }
      ;

WHILE    : T_For '(' BOOL_EXPR ')' '{' { ++scope; } Stmts '}'
      {
            --scope;
            string start = newlabel();
            $3.True = newlabel();
            $3.False = newlabel();
            $$.code = start + ": " + "\n" + $3.code + "if " + $3.addr + " goto " +
$3.True + "\n" + "goto " + $3.False + "\n" + $3.True + ": " + "\n" + $7.code + "\n" +
"goto " + start + "\n" + $3.False + ": " + "\n";
      }
      ;

POST : UNARY_EXPR
      {
            $$.code = $1.code + "\n";
      }
    | ASSIGN
      {
            $$.code = $1.code + "\n";
      }
    ;

FOR    : T_For T_Id T_For_Init VALUE { insert($2.strval, "int", yylineno, $4.val,
++scope); } ';' BOOL_EXPR ';' POST '{' Stmts '}'
      {
            --scope;
            string x = newlabel();
            string y = newlabel();
            string z = newlabel();
            $$.code = $2.strval + " = " + to_string($4.val) + "\n" + z + ": " + "\n" +
$7.code + "if " + $7.addr + " goto " + x + "\n" + "goto " + y + "\n" + x + ": " + "\n"
+ $11.code + "\n" + $9.code + "\n" + "goto " + z + "\n" + y + ": " + "\n";
      }
      ;
```

# 5. DESIGN STRATEGY

## Phase 1: Lexical Analysis

- LEX tool was used to create a scanner for Golang
- The scanner transforms the source file into a series of meaningful tokens containing information.
- Comments (single and multi-line) are removed.
- Tokens are classified into various forms as they appear. (example keywords, identifiers, numbers and operators)
- The rules are regexes which have corresponding actions that execute on a match with the source input.
- ‘yytext’ is the lex variable that stores the matched string. A global variable ‘yylval’ is used to record the value of each lexeme scanned. The val attribute of ‘yylval’ stores the integer value of the number token
- Skipping white spaces is handled in this phase.
- Scanning error is reported when the input string does not match any rule in the lex file.
- Generating symbol table with details such as variable names, line number where it is used, type of the variable, and scope of the respective variable.

## Phase 2: Syntax Analysis

Syntax analysis is responsible for verifying that the sequence of tokens forms a valid sentence given the definition of the Programming Language grammar.

- Yacc tool is used for parsing.
- Syntax analysis is only responsible for verifying that the sequence of tokens forms a valid sentence given the definition of your Programming Language grammar.
- The design implementation supports
    a. Variable declarations and initializations
    b. Variable assignments
    c. Variables of type int, float32, float64 and string
    d. Arithmetic and boolean expressions
    e. Postfix and prefix expressions
    f. Constructs - while loop and for loop

## Phase 3: Semantic Analysis

Semantic analysis phase is implemented in the yacc file as rules and it includes

- Checking for variable redeclaration in the same scope
- Checking if variables are used without prior declaration in the scope
- Checking if the number of variables and values match on either side of the ‘=’ symbol during variable declaration
- Checking if lvalue is not a constant

## Phase 4: Intermediate Code Generation (ICG)

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax the tree then can be converted into a linear representation. Intermediate code tends to be machine independent code.

**Three-Address Code**

A statement involving no more than three references (two for operands and one for result) is known as three address statements. A sequence of three address statements is known as three address code. Three address statements are of the form x = y op z, here x, y, z will have an address (memory location).

Example – The three address code for the expression a + b * c :
t1 = b
t2 = c
t3 = t1 * t2
t4 = a
t5 = t4 + t3
t1, t2, t3, t4 and t5 are temporary variables.

The data structure used to represent Three address Code is the Quadruples. It is shown with 4 columns- operator, operand1, operand2, and result.

## Phase 5: Code Optimization

The code optimizer maintains a key-value mapping that resembles the symbol table structure to keep track of variables and their values (possibly after expression evaluation). This structure is used to perform constant propagation and constant folding in sequential blocks followed by dead code elimination.

Methods implemented

- Constant folding
- Constant propagation
- Eliminate Dead code/ unreachable code

**Constant folding**

- is the process of recognizing and evaluating **constant** expressions at compile time rather than computing them at runtime.
- These are typically calculations that only reference constant values or expressions that reference variables whose values are constant.

  Ex1:

      y = 2 * 3 => y = 6
      y = 2 + 6 => y = 8

**Constant Propagation**

- Constant assigned to a variable is propagated to places where the variable is referenced in the code.
- This is usually called Constant Propagation.

Ex1:

```
y = 10
x = x + y => x = x + 10
```

Ex2:

```
x = 30
t = t * x   =>   t = t * 30
```

**Dead Code Elimination**

- Dead Code Elimination is an optimization that removes code which does not affect the program results.
- It shrinks program size and it allows the running program to avoid executing irrelevant operations, which reduces its running time of the program.

## Error Handling

- Checking for variable redeclaration in the same scope
- Checking if variables are used without prior declaration in the scope
- Checking if the number of variables and values match on either side of the '=' symbol during variable declaration
- Checking if lvalue is not a constant

# 6. IMPLEMENTATION DETAILS

## 6.1 Symbol Table Creation

**Symbol table with expression evaluation**

A structure is maintained to keep track of the variables in the input. The parameters of the structure are the name, the line number of occurrences, data type and scope of variables. Scope is clearly checked using the check_decl function.

```
class Node
{
      private:
            string id;
            string type;
            int lineno;
            int scope;
      public:
            Node(string, string, int, double, int);
            friend int check_decl(string, int);
            friend ostream& operator<<(ostream &o, const Node& n);
};
```

## 6.2 INTERMEDIATE CODE GENERATION

The constructs given being while loop and for loop, intermediate code generation has been implemented. Lex and yacc files have been used where the regular expressions have been defined in the lex file and the grammar along with the intermediate code generation which are mainly three address codes has been implemented in the yacc file.

**Three-Address Code**

A statement involving no more than three references (two for operands and one for result) is known as three address statements. A sequence of three address statements is known as three address code. Three address statements are of the form x = y op z, here x, y, z will have an address (memory location).

- The three-address code generation of while loop is done by using the code for boolean condition, the label for true is pointed at the statements of the while loop, followed by a goto statement which points to the beginning of the whole code, and finally the false label pointing at the next statement after the while loop
- For the for-loop construct, the codes for the initialiser of the loop and for the boolean condition are generated, followed by the code for the for-loop body, and the code for the post increment/decrement of the loop variable, and finally the appropriate true and false labels are written

**Quadruples**

A quadruple is an optimised representation of a 3-address code, in the format
(op, arg1, arg2, result)

Ex:

```
a = b + c
Quadruple format (op, arg1, arg2, res)
                   +    b    c    a
```

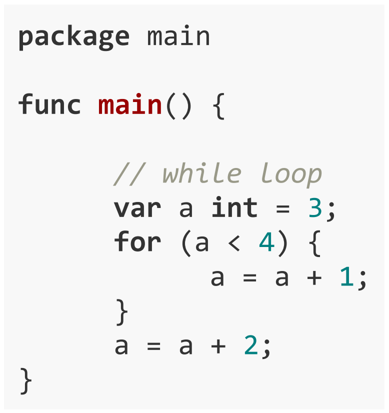
The intermediate three-address code generated is given as an input to a python script which converts the ICG to Quadruple format and redirects the output to a text file.

## 6.3 CODE OPTIMIZATION

Python was used for code optimisation which takes an input file containing intermediate code generated in the form of quadruples. The following were the code optimization techniques implemented in the project

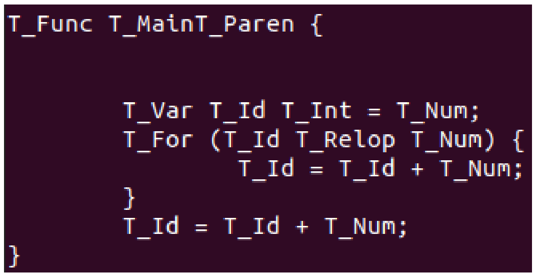
- Constant Folding
- Constant Propagation
- Dead Code Elimination

## 7. SNAPSHOTS



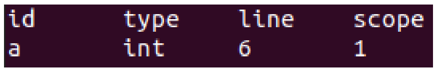
```
package main

func main() {

	// while Loop
	var a int = 3;
	for (a < 4) {
		a = a + 1;
	}
	a = a + 2;
}
```

*Input Go Program*



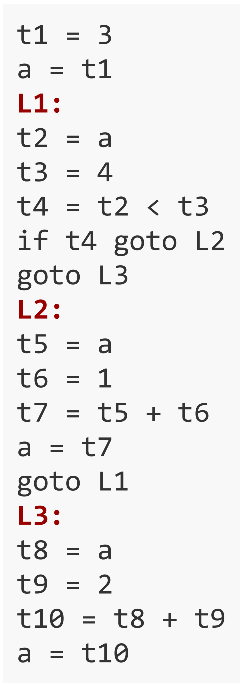
```
T_Func T_MainT_Paren {


	T_Var T_Id T_Int = T_Num;
	T_For (T_Id T_Relop T_Num) {
		T_Id = T_Id + T_Num;
	}
	T_Id = T_Id + T_Num;
}
```

*Tokens being generated*



```
id          type        line        scope
a           int         6           1
```

*Symbol Table entries*



```
t1 = 3
a = t1
L1:
t2 = a
t3 = 4
t4 = t2 < t3
if t4 goto L2
goto L3
L2:
t5 = a
t6 = 1
t7 = t5 + t6
a = t7
goto L1
L3:
t8 = a
t9 = 2
t10 = t8 + t9
a = t10
```

*Intermediate Code generated (Three-Address Format)*

```
= 3 NULL t1
int a NULL NULL
= t1 NULL a
Label NULL NULL L1
= a NULL t2
= 4 NULL t3
< t2 t3 t4
if t4 NULL L2
goto NULL NULL L3
Label NULL NULL L2
= a NULL t5
= 1 NULL t6
+ t5 t6 t7
= t7 NULL a
goto NULL NULL L1
Label NULL NULL L3
= a NULL t8
= 2 NULL t9
+ t8 t9 t10
= t10 NULL a
```

*Quadruple Format*

```
------------Quadruple form after Constant Folding and Propagation------------
= 3 NULL t1
int a NULL NULL
= 3 NULL a
Label NULL NULL L1
= 3 NULL t2
= 4 NULL t3
< t2 t3 t4
if t4 NULL L2
goto NULL NULL L3
Label NULL NULL L2
= 3 NULL t5
= 1 NULL t6
= 4 NULL t7
= 4 NULL a
goto NULL NULL L1
Label NULL NULL L3
= 4 NULL t8
= 2 NULL t9
= 6 NULL t10
= 6 NULL a
```

*Constant Folding and Constant Propagation*

```
------------------------Dead Code Elimination-----------------------
int   a     NULL  NULL
=     3     NULL  t2
=     4     NULL  t3
<     t2    t3    t4
if    t4    NULL  L2
goto  NULL  NULL  L3
goto  NULL  NULL  L1
=     6     NULL  a
```

*Dead Code Elimination*

## 8. RESULTS

We have clearly explained the design strategies being used and implemented in the different stages involved in building a mini-compiler and successfully built one that generates intermediate code and optimises given a Golang program as input.

Shortcomings with respect to our implementation of the Golang compiler

- We were unable to implement a literal table to keep track of symbol table entries' updation. Values were directly stored in the symbol table. Hence, we were unable to update the value of a variable in the symbol table.
- While the constant folding and propagation are happening correctly, values of variables should not be propagated to variables which are part of looping constructs as their values will change with different iterations of the loop. This has to be taken care of.

## 9. CONCLUSIONS

We have used the lex and yacc tools provided to build a mini-compiler for Golang. As part of the basic foundation for any compiler, in addition to the constructs assigned to us, basic blocks like declarations, assignments were implemented. This compiler was built according to the various phases of a compiler design namely Lexical Analysis, Syntax Analysis, ICG and Code optimisation. We included the error catching functionality into the compiler. With this, we can catch various syntactic and semantic errors.

We can easily enhance parts of the compiler to handle all constructs of the Golang.

## 10. FURTHER ENHANCEMENTS

- Implementing a literal table to be able to better keep track of updating the values of the variables having entries in the symbol table.

- Improve the compiler's ability to handle more control structures, secondary and user-defined datatypes more efficiently. We shall also include the ability of the compiler to handle user-defined functions and classes.

- The compiler must also handle importing of Go packages

- The compiler must compile different files and have support to link them as well.

- We will be implementing more code optimization techniques and enhance the existing ones to work faster and consume lesser memory.