Report on

"Mini GoLang Compiler"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Avinash V K	**PES2201800114**
Shreya Yuvraj Panale	**PES2201800117**
Sanskriti Pattanayak	**PES2201800329**

Under the guidance of

Dr. Mehala N
Professor of Department of Computer
Science and Engineering
PES University

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

● INTRODUCTION

This project, being a Mini Compiler for the GoLang programming language, focuses on generating an intermediate code for the language for specific constructs. It works for constructs such as arithmetic statements, relational conditional expressions, print statements, if-else conditional statements, conditional for loops, and basic function declarations.

The main functionality of the project is to generate an optimized intermediate code for the given source code. This is done using the following steps:

- ➔ Lexical and Syntax Analysis
- ➔ Lexical and Syntax Error Handling
- ➔ 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 predefined 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. Methods such as Copy Propagation, Dead Code Removal, Constant Folding, and Constant Propagation is used to optimize the intermediate code generated by the parser

● ARCHITECTURE OF LANGUAGE

- ➔ GoLang constructs implemented:
    1. Arithmetic Operations
    2. Relational Operations
    3. Simple If
    4. If-else
    5. For-loop
- ➔ Arithmetic expressions with +, -, *, /, ++, -- are handled
- ➔ Boolean expressions with >,<,>=,<=,== are handled
- ➔ Error handling reports undeclared variables, re-defined variables and type mismatch.
- ➔ Error handling also reports syntax errors with line numbers
- ➔ Error handling reports lexical errors such as unidentified tokens, badly terminated strings, and extra-long identifiers.

Hello World Program in GOLANG to get a Basic Construct Idea

```
package main
import "fmt"
func main() {
    fmt.Println("Hello World");
}
```

LITERATURE SURVEY

1. https://www.oreilly.com/library/view/lex-yacc/9781565920002/ch01.html
2. https://golang.org/ref/spec#Introduction
3. https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf
4. http://pages.cs.wisc.edu/~fischer/cs701.f07/lectures/Lecture02.pdf
5. https://cs.gmu.edu/~white/CS540/Slides/Semantic/CS540-2-lecture6.pdf

CONTEXT FREE GRAMMAR

➔ `Program :: = PackageSpecs ImportDeclList TopLevelDeclaration`

➔ `PackageSpecs :: = PACKAGE PackageName`

➔ `ImportDeclList ::= IMPORT PackageName | IMPORT ( PackNameList)`

➔ `PackageNameList ::= PackageNameList PackageName`

➔ `PackageName ::= IDENTIFIER | String`

➔ `Statement ::= Declaration | SimpleStmt | ReturnStmt | Block | IfStmt | ForStmt | PrintSmt`

➔ `SimpleStmt ::= IncDecStmt | Assignment`

➔ `Assignment ::= ExpressionList '=' ExpressionList`

➔ `Declaration ::= ConstDecl | VarDecl`

➔ `ConstDecl ::= CONST IdentifierList Type | CONST IdentifierList Type = Expression`

➔ `VarDecl ::= VAR IdentifierList Type | VAR IdentifierList Type = Expression`

➔ `IfStmt ::= IF Expression Block | IF Expression Block Else Block | IF Expression Block ELSE IfStmt`

➔ `ForStmt ::= FOR SimpleStmt ; Cond ; SimpleStmt`

➔ `Expression ::= Expression math_op Expression | Expression rel_op Expression | Operand`

➔ `Operand ::= Literal | IDENTIFIER | ( Expression )`

➔ `Literal ::= INT | FLOAT | STRING | BOOL`

- ## DESIGN STRATEGY

a. SYMBOL TABLE CREATION

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

COLUMN NAME	DATA TYPE	USE	UPDATIONS
token_id	char *	Give Unique ID to each Token read by lexer.	Only at the time of creation.
symbol	char [100]	Store the yytext of the token read as it is.	Only at the time of creation.
line_no	int	Store the last line number where the token has been encountered.	Updated every time the token is encountered.
value	char *	Store value of the identifier. Specific to variable filed and left as NULL for others.	Updated every time variable is either initialised / assigned to a new value. Computed values are stored in case of arithmetic computations.
datatype	char *	Stores the datatype of the constant or the variable. data types include string, int , float32, bool.	Updated only at the time of creation.

b. INTERMEDIATE CODE GENERATION

Intermediate code is generated in two formats :

1. Quadruple format with each line as
   < Operator , Argument 1 , Argument 2, Result >
2. TAC format generated simultaneously while generating Quadruples.
   - ➔ All tokens read from the input pertaining to variables, constants and operators / newly generated temporary variables are pushed into a global stack.
   - ➔ Intermediate computation in the RHS is represented as temporary variables which are then used to assign values to the LHS, where Quadruple follows the format < Operator , Argument1, Argument2 , Temporary Variable >

- ➔ Basic blocks are identified and label statements to mark their beginning are generated using intermediate code generation. *Basic Block* is a straight line code sequence which has no branches in and out branches except to the entry and at the end respectively.
- ➔ The branches where flow of the code changes are represented using if statements and goto statements to change the course of code execution.
- ➔ Special functions are written to implement increment and decrement statements code generation which focus on generating temporary variables to store the intermediate output.
- ➔ Another focus point is for assignment statements where the Quadruple format follows the format < "=" , "Arg1" , NULL , "Res" >

c. CODE OPTIMIZATION:

1. **Copy Propagation:** After the assignment of one variable to another, a reference to one variable may be replaced with the value of the other variable (until one or the other of the variables is reassigned).

   temp = a   =>   ~~temp = a~~
   result = temp * temp   result = a * a

2. **Dead Code Removal:** Expressions or statements whose values or effects are unused may be eliminated.

   x = 5   =>   ~~x = 5~~
   result = 1 + a   result = 1 + a

3. **Constant Propagation:** Constant assigned to a variable is substituted when the variable is encountered during compile time till the variable is reassigned/recomputed.

   temp = 2   =>   ~~temp = 2~~
   result = temp + 3   result = 2 + 3

4. **Constant Folding:** An expression involving constant (literal) values may be evaluated and simplified to a constant result value.

   result = 4 * 5   =>   result = 20

d. ERROR HANDLING

The compiler reports the lexical, syntax and semantic errors to the user in case it occurs. Phase level error recovery strategy is implemented on strings and identifiers during lexical error handling. Another handling strategy implemented in the compiler is using error productions.

1. **Phase level error recovery**: Perform local correction on the input to repair the error. But error correction is difficult in this strategy. For strings and identifiers, in case the length of the string is greater than the allowed size, it replaces the name with the modified valid name.
2. **Error productions**: Augmented Grammars are used as production rules that generate erroneous constructs when these errors are encountered to execute some action defined. Common syntax errors are caught using this handling method

• IMPLEMENTATION DETAILS

The majority of the code is written in LEX and YACC, with C++ used to describe the symbol table and abstract syntax tree structures.
The Context Free grammar provided by the lex and yacc files is used for the SYNTAX analysis.
This is used for scanning the basic syntax-based production rules that must be followed while writing code.

The Lex file is divided into three parts as shown below. The first part defines all the necessary constants, variables, regular definitions as well as header files as required by the program. This is written in the C language.

```
%{
    #include <bits/stdc++.h>
    using namespace std;
    #include <stdio.h>
    #include<string.h>
    #include<errno.h>
    #include "y.tab.h"    /* token codes from the parser */
    int yylex();
    void yyerror (const char* s);
    extern int errno;
    int yycolno  = 0;
    int prevleng = 0;
    #define YY_USER_ACTION {yylloc.first_line = yylloc.last_line = yylineno; yylloc.last_column =
yycolno;}
%}
```

The above code defines two important global variables called **yycolno** and **prevleng** which help extract the column number of the tokens which are required while displaying syntax errors.
yylex() returns a value in the code above that indicates the type of token that was obtained.
If the token has a value, it is stored in an external variable called **yylval** (or a representation of the value, such as a pointer to a string containing the value).
Similarly, the lex and yacc library function **yyerror()** simply prints a text string argument to stderr.

Next a list of regular expressions is defined consisting of all possible valid entries that can be done and recognized by the compiler, such as keywords and constants.

```
UNICODE_LETTER            [a-zA-Z]
LETTER                    [a-zA-Z_]
DIGIT                     [0-9]
UNICODE_CHAR              [^\n]
NEW_LINE                  [\n]
VAR_TYPE                  "bool"|"error"|"float32"|"float64"|"int"|"string"
BOOL_CONST                "true"|"false"
NIL_VAL                   "nil"
WHITESPACE                [\t\r\f\v]+
```

The second section of the Lex file enclosed in %% contains the rules that are used to construct the programming language.These rules specify how each of the entities mentioned above is expected to function and what roles they perform in the code.
It also defines the types of keywords or identifiers that have been previously and concurrently studied in the lexical analysis process.

A small subsection of the same is shown below.

➔ Arithmetic Operators (Similar definitions for Relational Operators)

```
"="       {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_ASSIGN;}
"+"       {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_ADD;}
"-"       {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_MINUS;}
"*"       {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_MULTIPLY;}
"/"       {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_DIVIDE;}
"%"       {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_MOD;}
"++"      {yylval.sval= strdup(yytext);yycolno+=prevleng;prevleng=yyleng;return T_INCREMENT;}
"--"      {yylval.sval= strdup(yytext);yycolno+=prevleng;prevleng=yyleng;return T_DECREMENT;}
```

➔ Punctuations and Brackets

```
"("       {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_LEFTPARANTHESES;}
")"       {yylval.sval= strdup(yytext);yycolno+=prevleng;prevleng=yyleng;return T_RIGHTPARANTHESES;}
"{"       {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_LEFTBRACE;}
"}"       {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_RIGHTBRACE;}
"["       {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_LEFTBRACKET;}
"]"       {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_RIGHTBRACKET;}
","       {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_COMMA;}
";"       {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_SEMICOLON;}
"."       {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_PERIOD;}
```

➔ Keywords

```
"package"        {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_PACKAGE;}
"import"         {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_IMPORT;}
"func"           {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_FUNC;}
"break"          {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_BREAK;}
"const"          {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_CONST;}
"continue"       {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_CONTINUE;}
"else"           {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_ELSE;}
"for"            {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_FOR;}
"if"             {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_IF;}
"return"         {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_RETURN;}
"var"            {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_VAR;}
{VAR_TYPE}       {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_VAR_TYPE;}
{BOOL_CONST}     {yylval.sval= strdup(yytext);yycolno+=prevleng;prevleng=yyleng;return
T_BOOL_CONST;}
{NIL_VAL}        {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return T_NIL_VAL;}
```

➔ Handling Whitespaces and newlines

```
\/\/{UNICODE_CHAR}*\n       {yycolno=0; prevleng=0;}
"\n"                        {yycolno=0;prevleng=0;}
"\t"                        {yycolno+=prevleng; prevleng=4;};
" "                         {yycolno+=prevleng; prevleng=1;}
{WHITESPACE}                ;
```

As shown in the code snippets above, each of the keywords defined in the above list has a set of rules defined with it to make the compiler's lexical analyzer understand what it is reading.

yylval.sval = strdup(yytext)

yytext is a feature that returns the text that the lexical analyzer is currently analysing.
The strdup() function copies the keyword's value into yylval, which has a string value and is later used for symbol table and AST generation.

The yywrap() function is used to wrap up the entire syntax analysis process and move it on to the Yacc file for semantic analysis, as shown below. This is defined in the last part of the LEX file.

```
int yywrap(){
    printf("Input Exhausted!\n");
    return 1;
}
```

YACC File Rules and Corresponding Code Example

Production Rule	Example Code Snippet

```
StartFile:
    T_PACKAGE PackageName ImportDeclList TopLevelDeclList {
    };

PackageName:
    T_IDENTIFIER {}
    | T_STRING {}
    | T_STRING T_SEMICOLON {};

//Can be a list of imports, single import or no imports.
ImportDeclList:
      ImportDeclList ImportDecl {}
    | ImportDecl {}
    | /*empty*/ {};

ImportDecl:
    T_IMPORT PackageName {}
    | T_IMPORT T_LEFTPARANTHESES ImportSpecList
    T_RIGHTPARANTHESES {};

ImportSpecList:
    ImportSpecList PackageName {}
    | PackageName {};
```
	```
	package main
	import "fmt"
	//Any type of top level
	declaration
	func main() {}
	```

```
TopLevelDeclList:
    TopLevelDeclList TopLevelDecl  {}
    | TopLevelDeclList T_SEMICOLON TopLevelDecl  {}
    | TopLevelDecl  {};

TopLevelDecl:
    Declaration {}
    | FunctionDecl {};
```

```
//Following Rules elaborate on
the code snippets corresponding
to top level declaration.
```

```
FunctionDecl:
    T_FUNC T_IDENTIFIER T_LEFTPARANTHESES T_RIGHTPARANTHESES
    Block {};
```

```
Declaration:
    T_VAR Expression Type assign_op ExpressionList {}
    | T_VAR Expression Type {};
    | T_CONST Expression Type assign_op ExpressionList
{codegen_assign();}
    | T_CONST Expression Type {};
    ;
```

```
func main() {
    const a string = "Hello";
    var b int;
    var c int = 2;
}
```

```
Block:
    T_LEFTBRACE StatementList T_RIGHTBRACE {}
    | /*empty*/ {};
StatementList:
    StatementList Statement T_SEMICOLON {}
    | Statement T_SEMICOLON {}
    | StatementList Statement {}
    | Statement {};
Statement:
    Declaration {}
    | SimpleStmt {}
    | Block {}
    | IfStmt {}
    | ForStmt {}
    | PrintStmt {};
```

```
//Following Rules elaborate on
the code snippets corresponding
to statements.
```

```
SimpleStmt:
    Expression assign_op Expression {}
    | Expression T_INCREMENT {}
    | Expression T_DECREMENT {}
    | ExpressionList assign_op ExpressionList {}
    | /*empty*/{};
```

```
    b = a+c*2;
    b++;
```

```
Expression:
    Expression math_op Expression{}
    | Expression rel_op Expression{}
    | Expression bin_op Expression {}
    | unary_op Operand {}
    | Operand {};
```

```
    b = a+c*2;
    var e bool = a==b;
    var f bool = 2<3;
```

```
bin_op:
    T_LOR {}
    | T_LAND {};
rel_op:
    T_EQL {}
    | T_NEQ {}
    | T_LSR {}
    | T_LEQ {}
    | T_GTR {}
    | T_GEQ {};
math_op:
    T_ADD {}
    | T_MINUS {}
    | T_MULTIPLY {}
    | T_DIVIDE {}
    | T_MOD {};
unary_op:
    T_ADD {}
    | T_MINUS {}
    | T_NOT {};
assign_op:
    T_ASSIGN {};
```

```
//The Rules here just group
together operator types which
are used in the expressions
above.
```

```
Operand:
    BasicLit {}
    | T_IDENTIFIER {}
    | T_LEFTPARANTHESES Expression T_RIGHTPARANTHESES {};
BasicLit:
    T_INTEGER {}
    | T_FLOAT {}
    | T_STRING {}
    | T_BOOL_CONST {};
```

```
//Above Rules make use of
operands in the expressions.
```

```
IfStmt:
    T_IF Expression Block {}
    | T_IF Expression Block T_ELSE  Block {}
    | T_IF Expression Block T_ELSE IfStmt {};
```

```
 //If else statement
    if 7 == 0 {}
    else {}
 //If statement
    if 8 >= 0 {}
    var num int = 9;
 //If - elseif - else statement
    if num < 0 {}
    else if num < 10 {}
    else {}
```

```
ForStmt:
  T_FOR SimpleStmt{} T_SEMICOLON Expression{} T_SEMICOLON
SimpleStmt{} Block{} {};
```

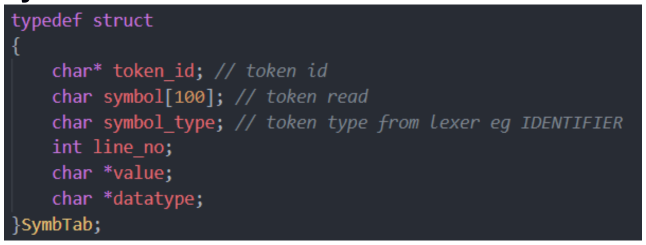
```
    for i = 0 ; i < 10; i++ {
        sum++;
    };
```

```
PrintStmt:
    T_PRINT T_LEFTPARANTHESES T_STRING T_RIGHTPARANTHESES {};
```

```
fmt.Println("Hello World");
```

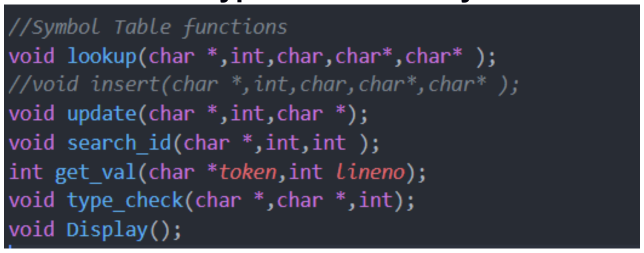
a. SYMBOL TABLE CREATION

➔ Symbol Table Structure



```
typedef struct
{
    char* token_id; // token id
    char symbol[100]; // token read
    char symbol_type; // token type from lexer eg IDENTIFIER
    int line_no;
    char *value;
    char *datatype;
}SymbTab;
```

➔ Function Prototypes defined in Symbol Table Header File



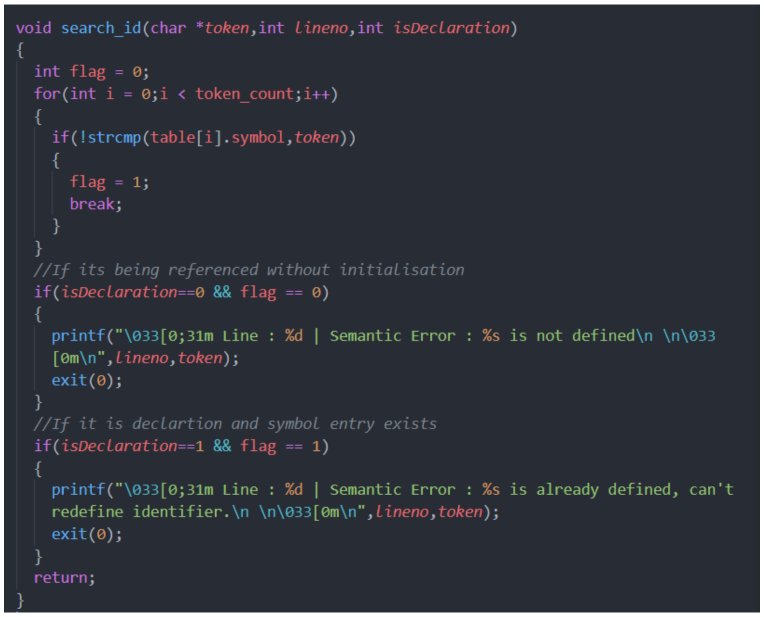
```
//Symbol Table functions
void lookup(char *,int,char,char*,char* );
//void insert(char *,int,char,char*,char* );
void update(char *,int,char *);
void search_id(char *,int,int );
int get_val(char *token,int lineno);
void type_check(char *,char *,int);
void Display();
```

➔ search_id Function

This function is responsible for checking if the passed identifier token is already present in the symbol table.

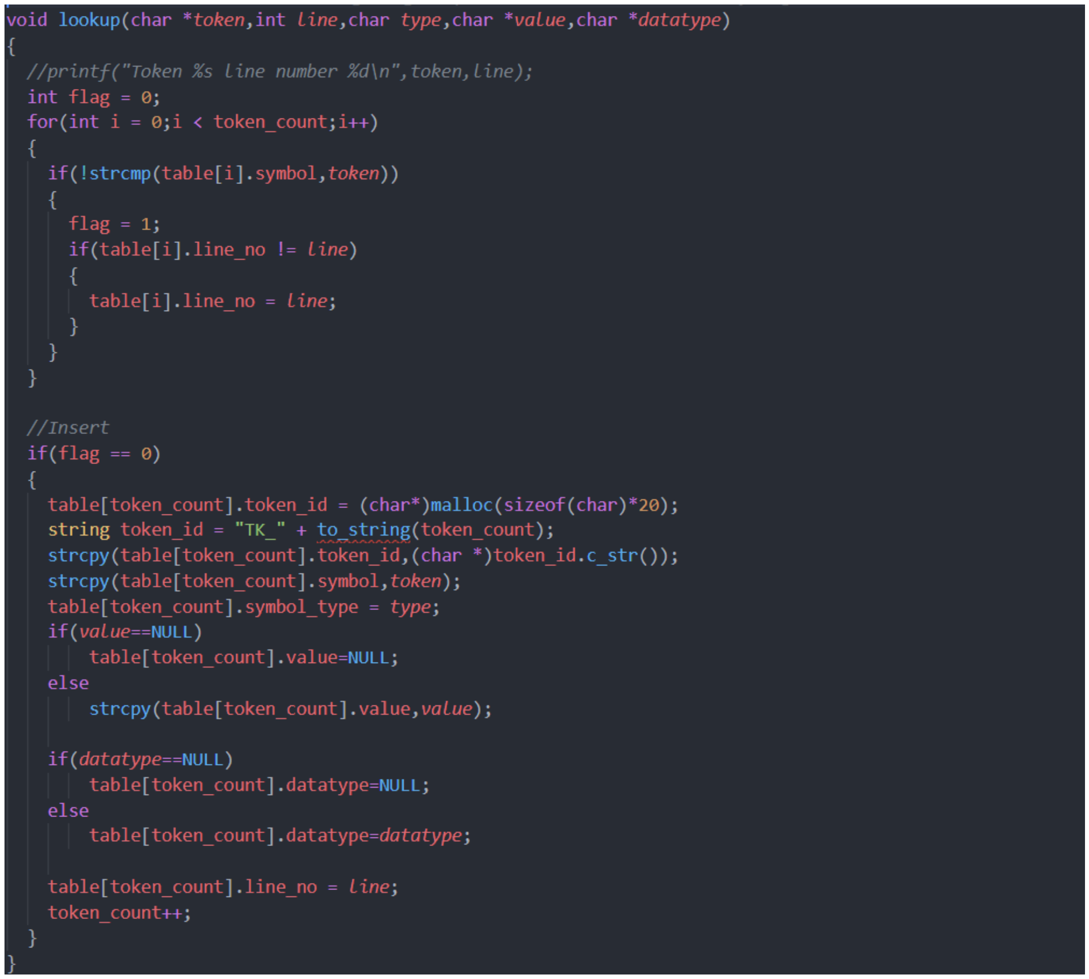
The *isDeclaration* parameter passed is used to identify if the statement the identifier occurs is a declaration or not. If it is a declaration and the symbol entry exists this implies a semantic error for redeclaration of variables.

If it is not a declaration and the symbol entry doesn't exist it implies that an undefined identifier is being referenced which is also an indication of a semantic error.



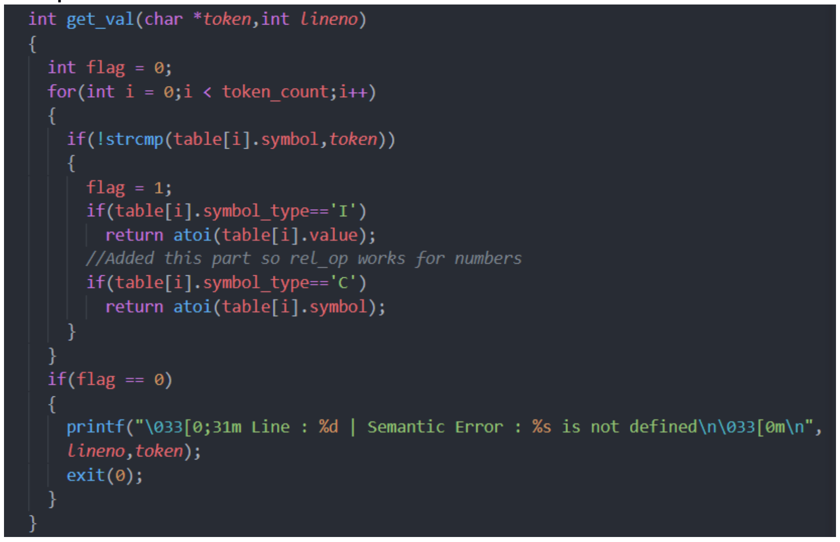
```
void search_id(char *token,int lineno,int isDeclaration)
{
  int flag = 0;
  for(int i = 0;i < token_count;i++)
  {
    if(!strcmp(table[i].symbol,token))
    {
      flag = 1;
      break;
    }
  }
  //If its being referenced without initialisation
  if(isDeclaration==0 && flag == 0)
  {
    printf("\033[0;31m Line : %d | Semantic Error : %s is not defined\n \n\033
    [0m\n",lineno,token);
    exit(0);
  }
  //If it is declartion and symbol entry exists
  if(isDeclaration==1 && flag == 1)
  {
    printf("\033[0;31m Line : %d | Semantic Error : %s is already defined, can't
    redefine identifier.\n \n\033[0m\n",lineno,token);
    exit(0);
  }
  return;
}
```

➔ **lookup Function (Also accommodate Inserting functionality)**
This function is responsible for checking if the passed token is already present in the symbol table. If the symbol already exists then only the line_no is updated. Otherwise a new symbol table entry is created to accommodate for the new symbol read.



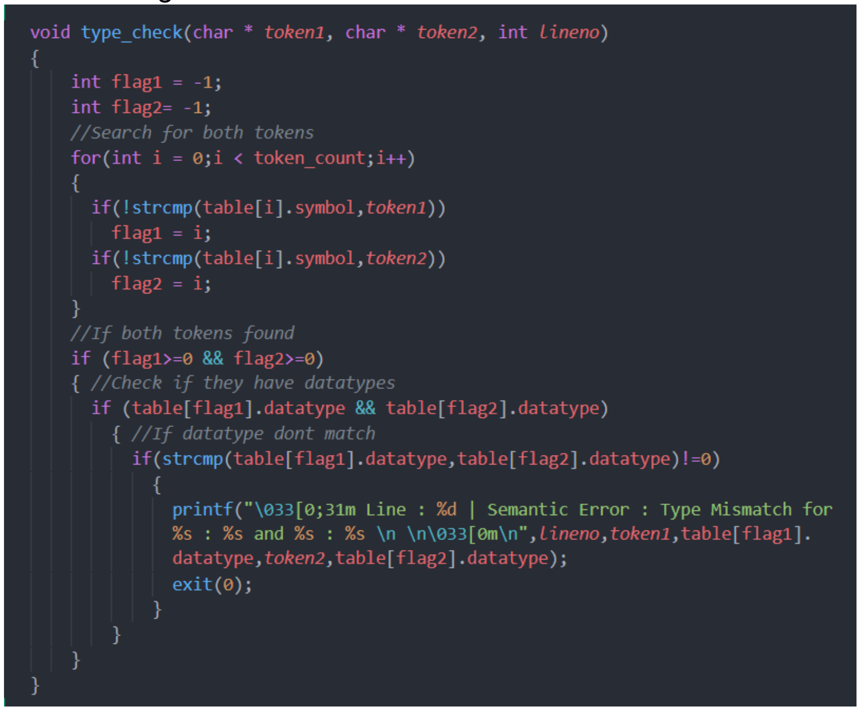
```
void lookup(char *token,int line,char type,char *value,char *datatype)
{
  //printf("Token %s line number %d\n",token,line);
  int flag = 0;
  for(int i = 0;i < token_count;i++)
  {
    if(!strcmp(table[i].symbol,token))
    {
      flag = 1;
      if(table[i].line_no != line)
      {
        table[i].line_no = line;
      }
    }
  }

  //Insert
  if(flag == 0)
  {
    table[token_count].token_id = (char*)malloc(sizeof(char)*20);
    string token_id = "TK_" + to_string(token_count);
    strcpy(table[token_count].token_id,(char *)token_id.c_str());
    strcpy(table[token_count].symbol,token);
    table[token_count].symbol_type = type;
    if(value==NULL)
        table[token_count].value=NULL;
    else
        strcpy(table[token_count].value,value);

    if(datatype==NULL)
        table[token_count].datatype=NULL;
    else
        table[token_count].datatype=datatype;

    table[token_count].line_no = line;
    token_count++;
  }
}
```

➔ **get_val Function**
This function is used to get the value of the identifier so it can be used during computations.



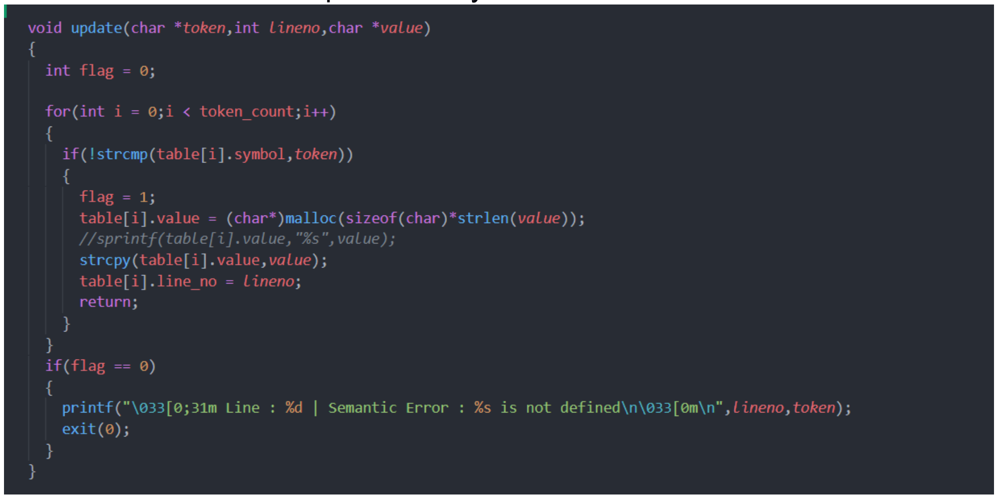
```
int get_val(char *token,int lineno)
{
  int flag = 0;
  for(int i = 0;i < token_count;i++)
  {
    if(!strcmp(table[i].symbol,token))
    {
      flag = 1;
      if(table[i].symbol_type=='I')
        return atoi(table[i].value);
      //Added this part so rel_op works for numbers
      if(table[i].symbol_type=='C')
        return atoi(table[i].symbol);
    }
  }
  if(flag == 0)
  {
    printf("\033[0;31m Line : %d | Semantic Error : %s is not defined\n\033[0m\n",
    lineno,token);
    exit(0);
  }
}
```

➔ **type_check Function**
This function compares the data type of the identifiers being used in an expression to check for type mismatch which is a semantic error. If semantic error is found an error message is thrown to indicate the same.



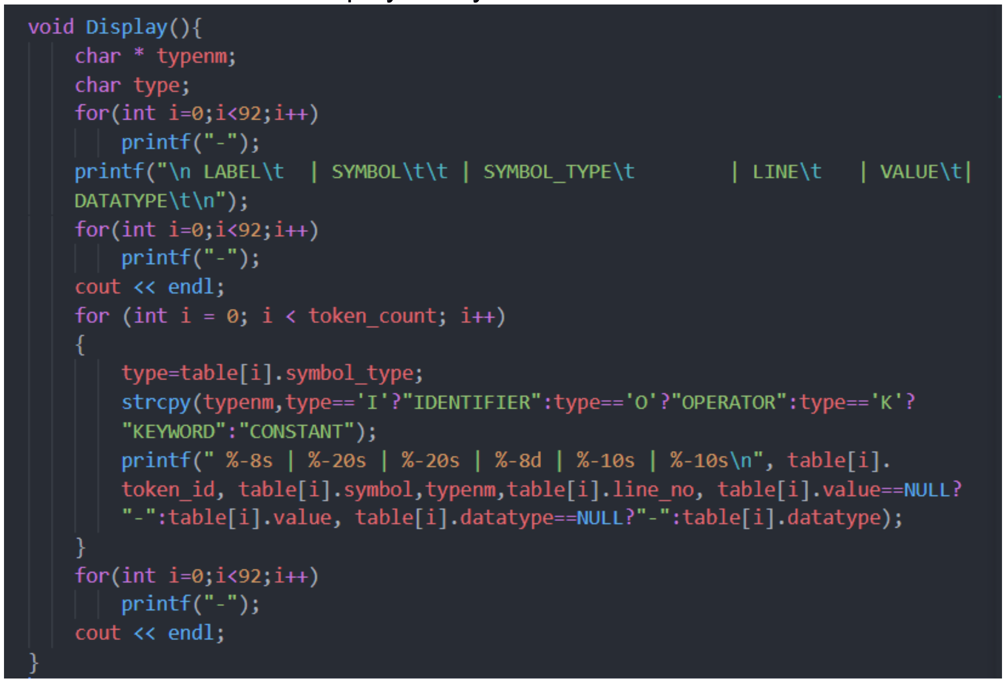
```
void type_check(char * token1, char * token2, int lineno)
{
    int flag1 = -1;
    int flag2= -1;
    //Search for both tokens
    for(int i = 0;i < token_count;i++)
    {
      if(!strcmp(table[i].symbol,token1))
        flag1 = i;
      if(!strcmp(table[i].symbol,token2))
        flag2 = i;
    }
    //If both tokens found
    if (flag1>=0 && flag2>=0)
    { //Check if they have datatypes
      if (table[flag1].datatype && table[flag2].datatype)
        { //If datatype dont match
          if(strcmp(table[flag1].datatype,table[flag2].datatype)!=0)
            {
              printf("\033[0;31m Line : %d | Semantic Error : Type Mismatch for
              %s : %s and %s : %s \n \n\033[0m\n",lineno,token1,table[flag1].
              datatype,token2,table[flag2].datatype);
              exit(0);
            }
        }
    }
}
```

- ➔ **update Function**
  This function is used to update the symbol table with an identifier values .



```
void update(char *token,int lineno,char *value)
{
  int flag = 0;

  for(int i = 0;i < token_count;i++)
  {
    if(!strcmp(table[i].symbol,token))
    {
      flag = 1;
      table[i].value = (char*)malloc(sizeof(char)*strlen(value));
      //sprintf(table[i].value,"%s",value);
      strcpy(table[i].value,value);
      table[i].line_no = lineno;
      return;
    }
  }
  if(flag == 0)
  {
    printf("\033[0;31m Line : %d | Semantic Error : %s is not defined\n\033[0m\n",lineno,token);
    exit(0);
  }
}
```

- ➔ **Display Function**
  This function is used to display the symbol table.



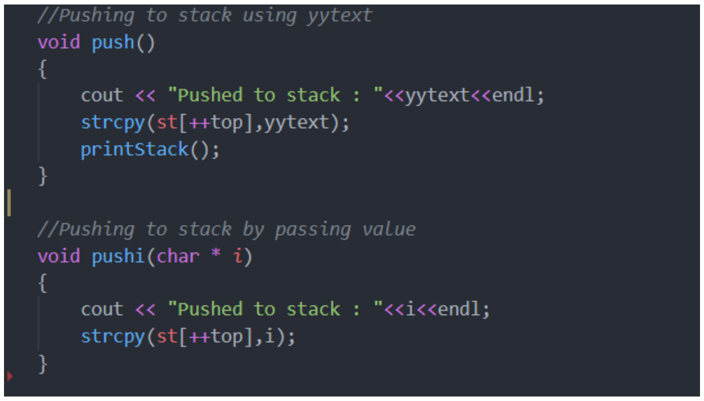
```
void Display(){
    char * typenm;
    char type;
    for(int i=0;i<92;i++)
        printf("-");
    printf("\n LABEL\t  | SYMBOL\t\t | SYMBOL_TYPE\t        | LINE\t   | VALUE\t|
    DATATYPE\t\n");
    for(int i=0;i<92;i++)
        printf("-");
    cout << endl;
    for (int i = 0; i < token_count; i++)
    {
        type=table[i].symbol_type;
        strcpy(typenm,type=='I'?"IDENTIFIER":type=='O'?"OPERATOR":type=='K'?
        "KEYWORD":"CONSTANT");
        printf(" %-8s | %-20s | %-20s | %-8d | %-10s | %-10s\n", table[i].
        token_id, table[i].symbol,typenm,table[i].line_no, table[i].value==NULL?
        "-":table[i].value, table[i].datatype==NULL?"-":table[i].datatype);
    }
    for(int i=0;i<92;i++)
        printf("-");
    cout << endl;
}
```

b. INTERMEDIATE CODE GENERATION

A global stack is maintained to store all the tokens read by the lexer. This stack is maintained. This stack stores the tokens in character string format.

A quadruple structure is defined for 4 character string members for Operator, Argument 1, Argument 2 and Result respectively. An array of quadruples is maintained to store the three address codes.

A function is defined for **Pushing tokens** into the stack as they are read in order of their positions as interpreted by the Parser.
eg. For the expression a = b, the pushing order is a , = , b with b on top of the stack

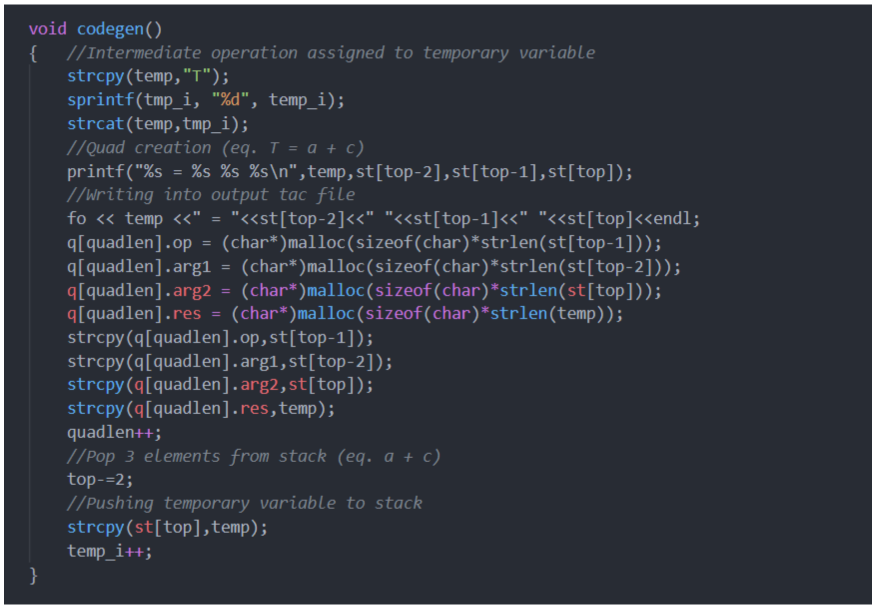


```
//Pushing to stack using yytext
void push()
{
    cout << "Pushed to stack : "<<yytext<<endl;
    strcpy(st[++top],yytext);
    printStack();
}

//Pushing to stack by passing value
void pushi(char * i)
{
    cout << "Pushed to stack : "<<i<<endl;
    strcpy(st[++top],i);
}
```

- **Code generation for temporary variables.**
  For an expression such as a = 7 * 9 , the value computed by 7*9 must be stored in a temporary variable that would be assigned to variable a.
  The current state of the stack would have the top three positions as 9 , * , 7 with 9 on top of the stack.
  This part of the code generates a new temporary variable, eg. T0 and generates a quadruple of the following format :

Operator	* (Using top-1 position in the stack)
Arg1	7 (Using top-2 position in the stack)
Arg2	9 (Using top position in the stack)
Result	T0 (Generated)

  After this 7, *, 9 are popped from the stack and replaced by temporary variable T0, which is pushed into the stack.



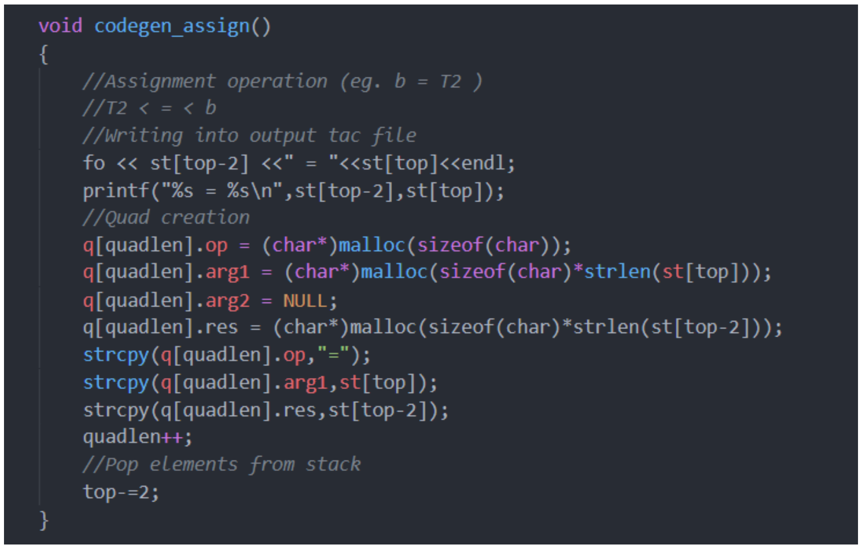
```
void codegen()
{   //Intermediate operation assigned to temporary variable
    strcpy(temp,"T");
    sprintf(tmp_i, "%d", temp_i);
    strcat(temp,tmp_i);
    //Quad creation (eq. T = a + c)
    printf("%s = %s %s %s\n",temp,st[top-2],st[top-1],st[top]);
    //Writing into output tac file
    fo << temp <<" = "<<st[top-2]<<" "<<st[top-1]<<" "<<st[top]<<endl;
    q[quadlen].op = (char*)malloc(sizeof(char)*strlen(st[top-1]));
    q[quadlen].arg1 = (char*)malloc(sizeof(char)*strlen(st[top-2]));
    q[quadlen].arg2 = (char*)malloc(sizeof(char)*strlen(st[top]));
    q[quadlen].res = (char*)malloc(sizeof(char)*strlen(temp));
    strcpy(q[quadlen].op,st[top-1]);
    strcpy(q[quadlen].arg1,st[top-2]);
    strcpy(q[quadlen].arg2,st[top]);
    strcpy(q[quadlen].res,temp);
    quadlen++;
    //Pop 3 elements from stack (eq. a + c)
    top-=2;
    //Pushing temporary variable to stack
    strcpy(st[top],temp);
    temp_i++;
}
```

- **Code Generation for assignment expressions**

  For an expression such as a = 7, the value computed by 7 must be assigned to variable a.
  The current state of the stack would have the top three positions as 7, =, a with 7 on top of the stack.
  This part of the code generates a quadruple of the following format :

Operator	=
Arg1	7 (Using top-2 position in the stack)
Arg2	NULL
Result	a (Using top position in the stack)

  All elements are popped from the stack.



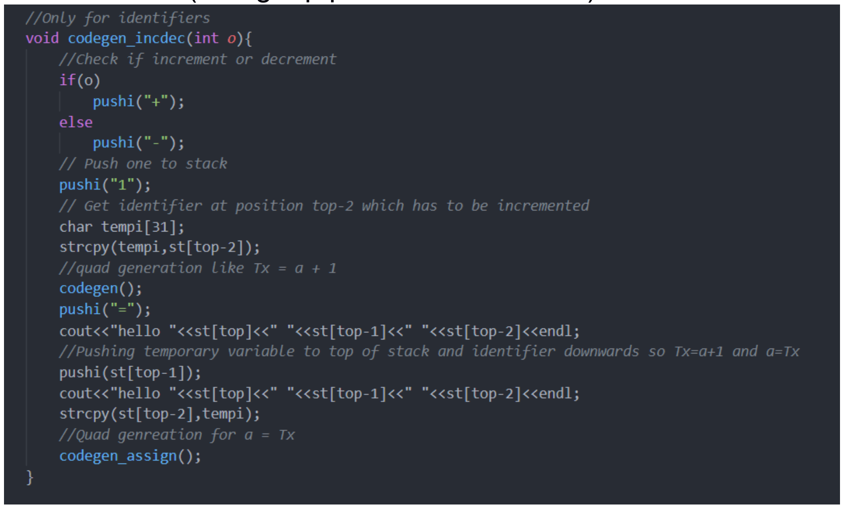
```
void codegen_assign()
{
    //Assignment operation (eg. b = T2 )
    //T2 < = < b
    //Writing into output tac file
    fo << st[top-2] <<" = "<<st[top]<<endl;
    printf("%s = %s\n",st[top-2],st[top]);
    //Quad creation
    q[quadlen].op = (char*)malloc(sizeof(char));
    q[quadlen].arg1 = (char*)malloc(sizeof(char)*strlen(st[top]));
    q[quadlen].arg2 = NULL;
    q[quadlen].res = (char*)malloc(sizeof(char)*strlen(st[top-2]));
    strcpy(q[quadlen].op,"=");
    strcpy(q[quadlen].arg1,st[top]);
    strcpy(q[quadlen].res,st[top-2]);
    quadlen++;
    //Pop elements from stack
    top-=2;
}
```

- **Code Generation for increment / decrement error**
  When a statement like a++ is encountered the following actions are taken :
  - ➔ Operator “+” (or “-” for decrement) pushed into the stack.
  - ➔ Constant value “1” pushed into the stack.
  - ➔ As the identifier a is already stored in the stack (at position top-2 by the pushing function described above) this is saved in a local variable for later use.
  - ➔ *Code generation for the temporary variable* is called to generate quadruple of the format:

Operator	+ (Using top-1 position in the stack
Arg1	a (Using top-2 position in the stack)
Arg2	1 (Using top position in the stack)
Result	T1 (Generated)

  - ➔ Now the using the previous local variable storing the value to be incremented (a), the newly created T1 for storing the incremented value computed the next step is to call *Code Generation for assignment expression* to create quadruple of the format:

Operator	=
Arg1	T1 (Using top-2 position in the stack)
Arg2	NULL
Result	a (Using top position in the stack)



```
//Only for identifiers
void codegen_incdec(int o){
    //Check if increment or decrement
    if(o)
        pushi("+");
    else
        pushi("-");
    // Push one to stack
    pushi("1");
    // Get identifier at position top-2 which has to be incremented
    char tempi[31];
    strcpy(tempi,st[top-2]);
    //quad generation like Tx = a + 1
    codegen();
    pushi("=");
    cout<<"hello "<<st[top]<<" "<<st[top-1]<<" "<<st[top-2]<<endl;
    //Pushing temporary variable to top of stack and identifier downwards so Tx=a+1 and a=Tx
    pushi(st[top-1]);
    cout<<"hello "<<st[top]<<" "<<st[top-1]<<" "<<st[top-2]<<endl;
    strcpy(st[top-2],tempi);
    //Quad genreation for a = Tx
    codegen_assign();
}
```

<u>FOR LOOP INTERMEDIATE CODE GENERATION</u>

eg. **for i=0 ; i<10 ; i++**

```
        {
                sum=sum+1;
        }
```

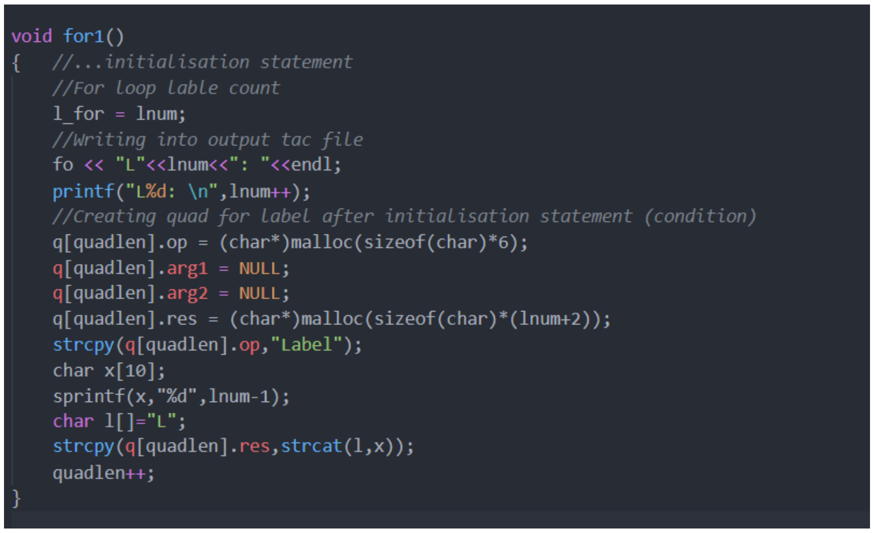
- **Code Generation for the initialization statement** happens using *Code Generation for assignment statements* described above.

- **Label Creation for Condition Statement of For Loop**

  As the condition statement has to be revisited again and again it marks the beginning of a Basic block and hence a Label is generated to mark the beginning of the condition.

Operator	Label
Arg1	
Arg2	
Result	L0

  New Label no.s are generated for every variable and a global count is maintained. Also, a global variable is used to store the label for this condition statement so that it can be referenced later.



```
void for1()
{   //...initialisation statement
    //For loop lable count
    l_for = lnum;
    //Writing into output tac file
    fo << "L"<<lnum<<": "<<endl;
    printf("L%d: \n",lnum++);
    //Creating quad for label after initialisation statement (condition)
    q[quadlen].op = (char*)malloc(sizeof(char)*6);
    q[quadlen].arg1 = NULL;
    q[quadlen].arg2 = NULL;
    q[quadlen].res = (char*)malloc(sizeof(char)*(lnum+2));
    strcpy(q[quadlen].op,"Label");
    char x[10];
    sprintf(x,"%d",lnum-1);
    char l[]="L";
    strcpy(q[quadlen].res,strcat(l,x));
    quadlen++;
}
```

- **Code generation for Condition statements** happens at this stage using Code generation for temporary variables.

  The top of the stack now stores the temporary variable(Let's say T0) corresponding to the output for the condition check.

  ➔ A temporary variable T1 is generated to store the output of "not T0" corresponding to the situation when the condition is false.
  A quadruple of the following format is generated:

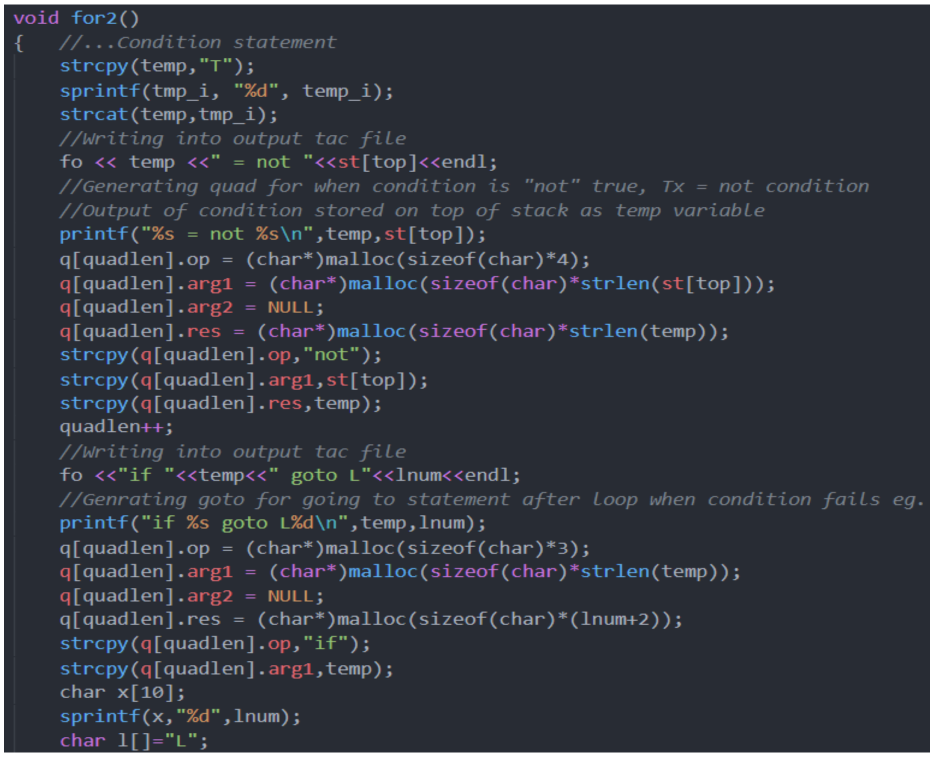
Operator	not
Arg1	T0
Arg2	
Result	T1

➔ Next, a goto statement is generated which corresponding to "if T1 goto L1" where the situation is handled when the condition evaluates as false and the goto points to label for instruction after for loop(L1)

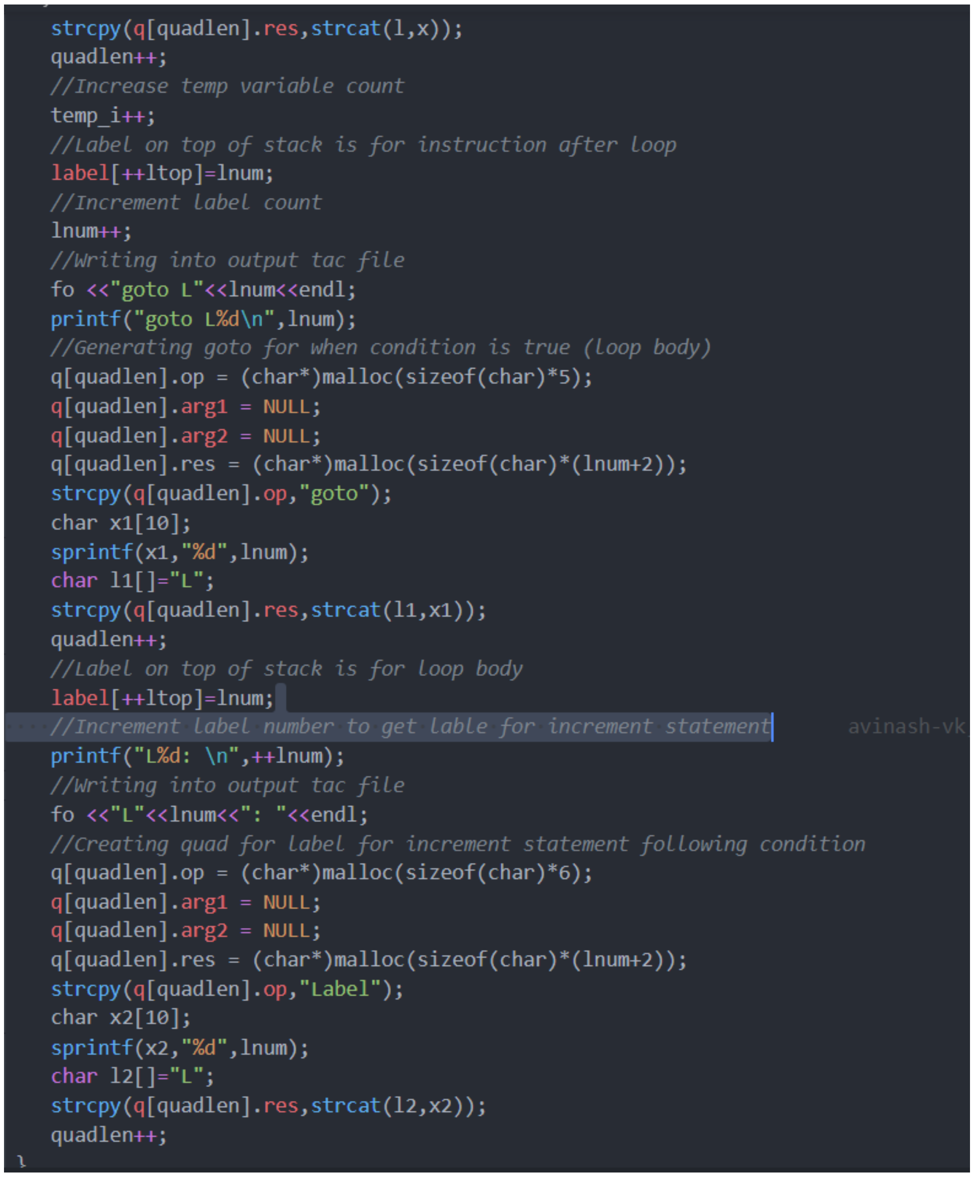
Operator	if
Arg1	T1
Arg2	
Result	L1

➔ The third step is to make a label for the increment statement following the condition statement.

Operator	Label
Arg1	
Arg2	
Result	L3



```
void for2()
{   //...Condition statement
    strcpy(temp,"T");
    sprintf(tmp_i, "%d", temp_i);
    strcat(temp,tmp_i);
    //Writing into output tac file
    fo << temp <<" = not "<<st[top]<<endl;
    //Generating quad for when condition is "not" true, Tx = not condition
    //Output of condition stored on top of stack as temp variable
    printf("%s = not %s\n",temp,st[top]);
    q[quadlen].op = (char*)malloc(sizeof(char)*4);
    q[quadlen].arg1 = (char*)malloc(sizeof(char)*strlen(st[top]));
    q[quadlen].arg2 = NULL;
    q[quadlen].res = (char*)malloc(sizeof(char)*strlen(temp));
    strcpy(q[quadlen].op,"not");
    strcpy(q[quadlen].arg1,st[top]);
    strcpy(q[quadlen].res,temp);
    quadlen++;
    //Writing into output tac file
    fo <<"if "<<temp<<" goto L"<<lnum<<endl;
    //Genrating goto for going to statement after loop when condition fails eg.
    printf("if %s goto L%d\n",temp,lnum);
    q[quadlen].op = (char*)malloc(sizeof(char)*3);
    q[quadlen].arg1 = (char*)malloc(sizeof(char)*strlen(temp));
    q[quadlen].arg2 = NULL;
    q[quadlen].res = (char*)malloc(sizeof(char)*(lnum+2));
    strcpy(q[quadlen].op,"if");
    strcpy(q[quadlen].arg1,temp);
    char x[10];
    sprintf(x,"%d",lnum);
    char l[]="L";
```



```
    strcpy(q[quadlen].res,strcat(l,x));
    quadlen++;
    //Increase temp variable count
    temp_i++;
    //Label on top of stack is for instruction after loop
    label[++ltop]=lnum;
    //Increment label count
    lnum++;
    //Writing into output tac file
    fo <<"goto L"<<lnum<<endl;
    printf("goto L%d\n",lnum);
    //Generating goto for when condition is true (loop body)
    q[quadlen].op = (char*)malloc(sizeof(char)*5);
    q[quadlen].arg1 = NULL;
    q[quadlen].arg2 = NULL;
    q[quadlen].res = (char*)malloc(sizeof(char)*(lnum+2));
    strcpy(q[quadlen].op,"goto");
    char x1[10];
    sprintf(x1,"%d",lnum);
    char l1[]="L";
    strcpy(q[quadlen].res,strcat(l1,x1));
    quadlen++;
    //Label on top of stack is for loop body
    label[++ltop]=lnum;
    //Increment label number to get lable for increment statement       avinash-vk
    printf("L%d: \n",++lnum);
    //Writing into output tac file
    fo <<"L"<<lnum<<": "<<endl;
    //Creating quad for label for increment statement following condition
    q[quadlen].op = (char*)malloc(sizeof(char)*6);
    q[quadlen].arg1 = NULL;
    q[quadlen].arg2 = NULL;
    q[quadlen].res = (char*)malloc(sizeof(char)*(lnum+2));
    strcpy(q[quadlen].op,"Label");
    char x2[10];
    sprintf(x2,"%d",lnum);
    char l2[]="L";
    strcpy(q[quadlen].res,strcat(l2,x2));
    quadlen++;
}
```

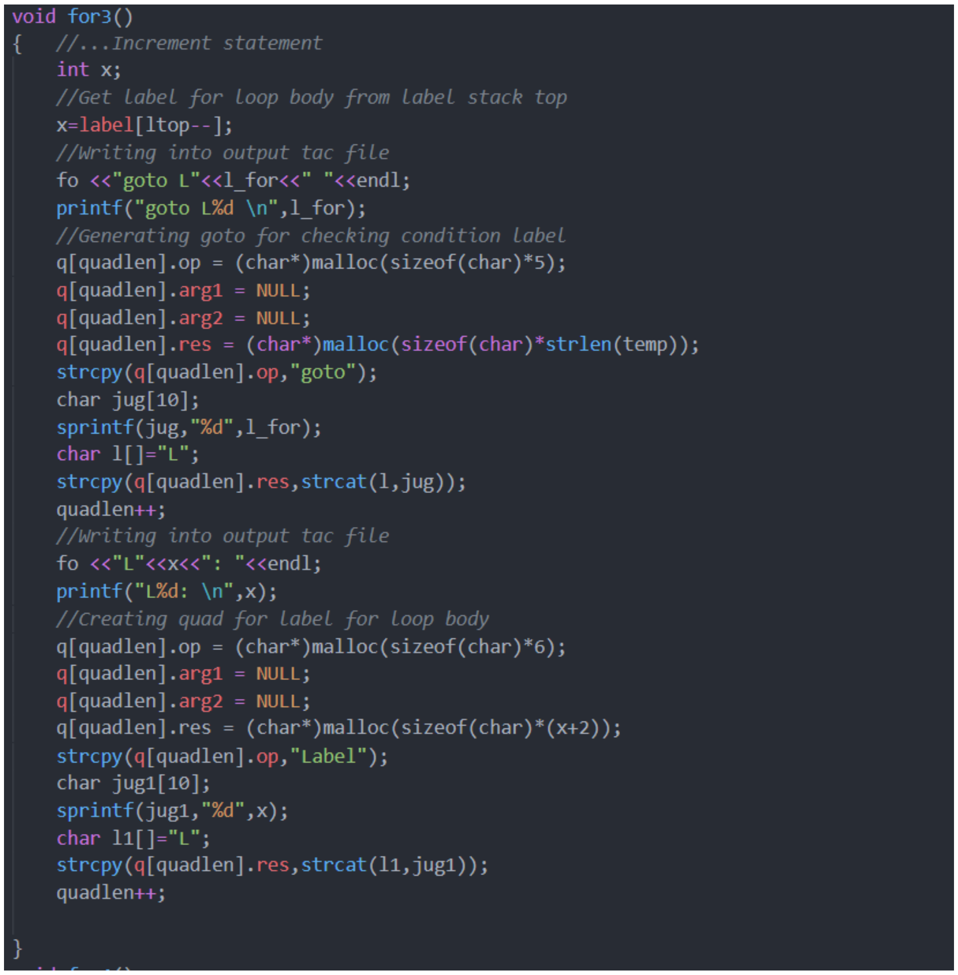
- **Code Generation for Increment Statements** happens at this stage using Code generation for increment /decrement expressions.

    ➔ Next the goto statement is made to go to the for loop checking condition (Label made by Label generation for condition statement)

Operator	goto
Arg1	
Arg2	
Result	L0

    ➔ Next Label is made for the statements following the increment statement that is the label is made for the loop body.

Operator	Label
Arg1	
Arg2	
Result	L2



```
void for3()
{   //...Increment statement
    int x;
    //Get label for loop body from label stack top
    x=label[ltop--];
    //Writing into output tac file
    fo <<"goto L"<<l_for<<" "<<endl;
    printf("goto L%d \n",l_for);
    //Generating goto for checking condition label
    q[quadlen].op = (char*)malloc(sizeof(char)*5);
    q[quadlen].arg1 = NULL;
    q[quadlen].arg2 = NULL;
    q[quadlen].res = (char*)malloc(sizeof(char)*strlen(temp));
    strcpy(q[quadlen].op,"goto");
    char jug[10];
    sprintf(jug,"%d",l_for);
    char l[]="L";
    strcpy(q[quadlen].res,strcat(l,jug));
    quadlen++;
    //Writing into output tac file
    fo <<"L"<<x<<": "<<endl;
    printf("L%d: \n",x);
    //Creating quad for label for loop body
    q[quadlen].op = (char*)malloc(sizeof(char)*6);
    q[quadlen].arg1 = NULL;
    q[quadlen].arg2 = NULL;
    q[quadlen].res = (char*)malloc(sizeof(char)*(x+2));
    strcpy(q[quadlen].op,"Label");
    char jug1[10];
    sprintf(jug1,"%d",x);
    char l1[]="L";
    strcpy(q[quadlen].res,strcat(l1,jug1));
    quadlen++;

}
```

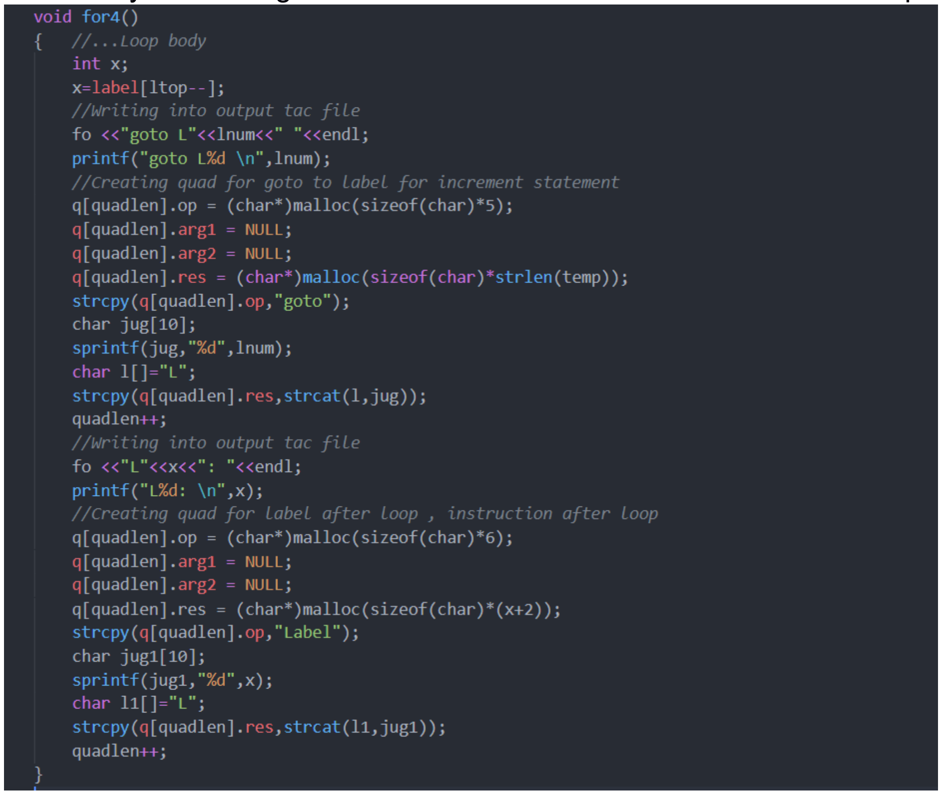
- **Code Generation for loop body and the following statements.**

  The quadruples corresponding to the loop body are generated using the above explained above.

  ➔ Next goto statement is made to goto the label for the increment statement which must be executed after for loop body execution.

Operator	goto
Arg1	
Arg2	
Result	L3

➔ Lastly a label is generated to mark the statement after the for loop.

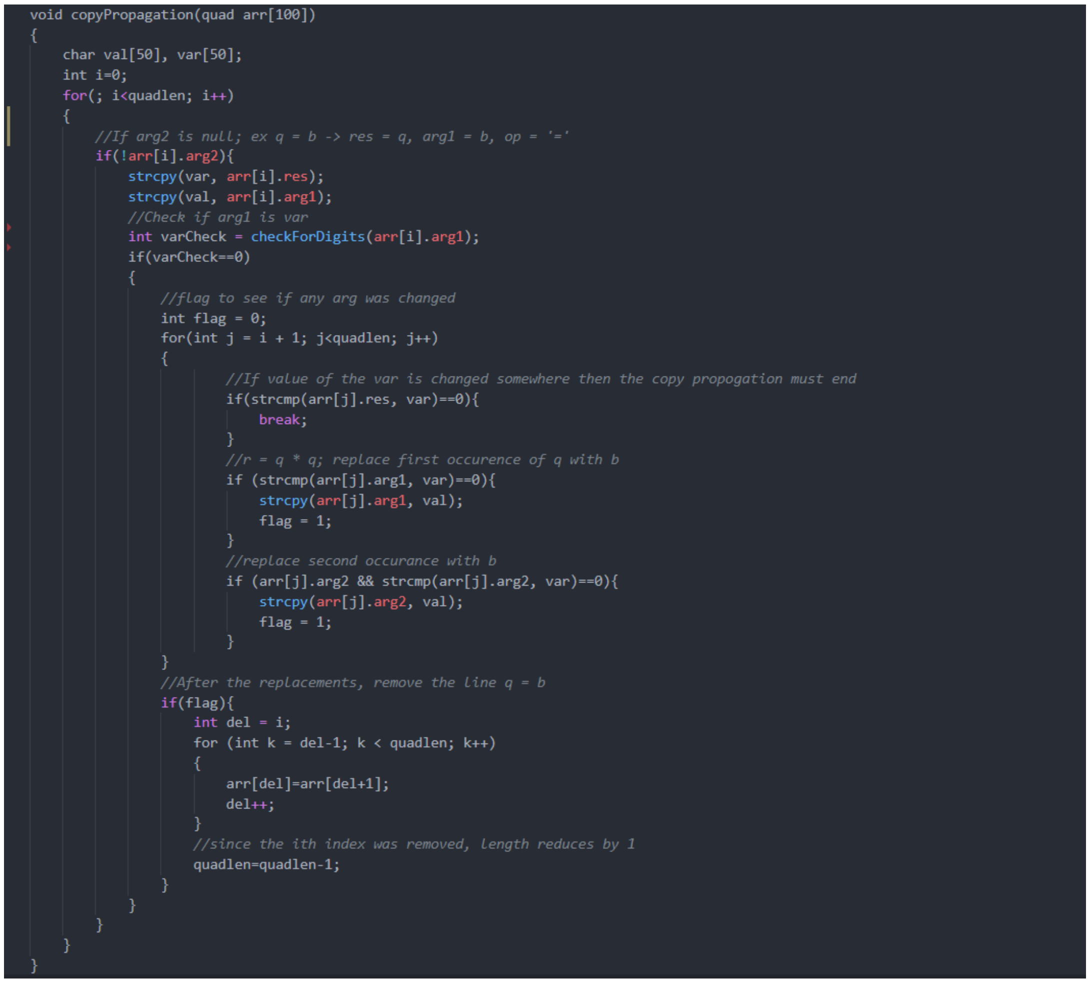


```
void for4()
{   //...Loop body
    int x;
    x=label[ltop--];
    //Writing into output tac file
    fo <<"goto L"<<lnum<<" "<<endl;
    printf("goto L%d \n",lnum);
    //Creating quad for goto to label for increment statement
    q[quadlen].op = (char*)malloc(sizeof(char)*5);
    q[quadlen].arg1 = NULL;
    q[quadlen].arg2 = NULL;
    q[quadlen].res = (char*)malloc(sizeof(char)*strlen(temp));
    strcpy(q[quadlen].op,"goto");
    char jug[10];
    sprintf(jug,"%d",lnum);
    char l[]="L";
    strcpy(q[quadlen].res,strcat(l,jug));
    quadlen++;
    //Writing into output tac file
    fo <<"L"<<x<<": "<<endl;
    printf("L%d: \n",x);
    //Creating quad for label after loop , instruction after loop
    q[quadlen].op = (char*)malloc(sizeof(char)*6);
    q[quadlen].arg1 = NULL;
    q[quadlen].arg2 = NULL;
    q[quadlen].res = (char*)malloc(sizeof(char)*(x+2));
    strcpy(q[quadlen].op,"Label");
    char jug1[10];
    sprintf(jug1,"%d",x);
    char l1[]="L";
    strcpy(q[quadlen].res,strcat(l1,jug1));
    quadlen++;
}
```

c. CODE OPTIMIZATION

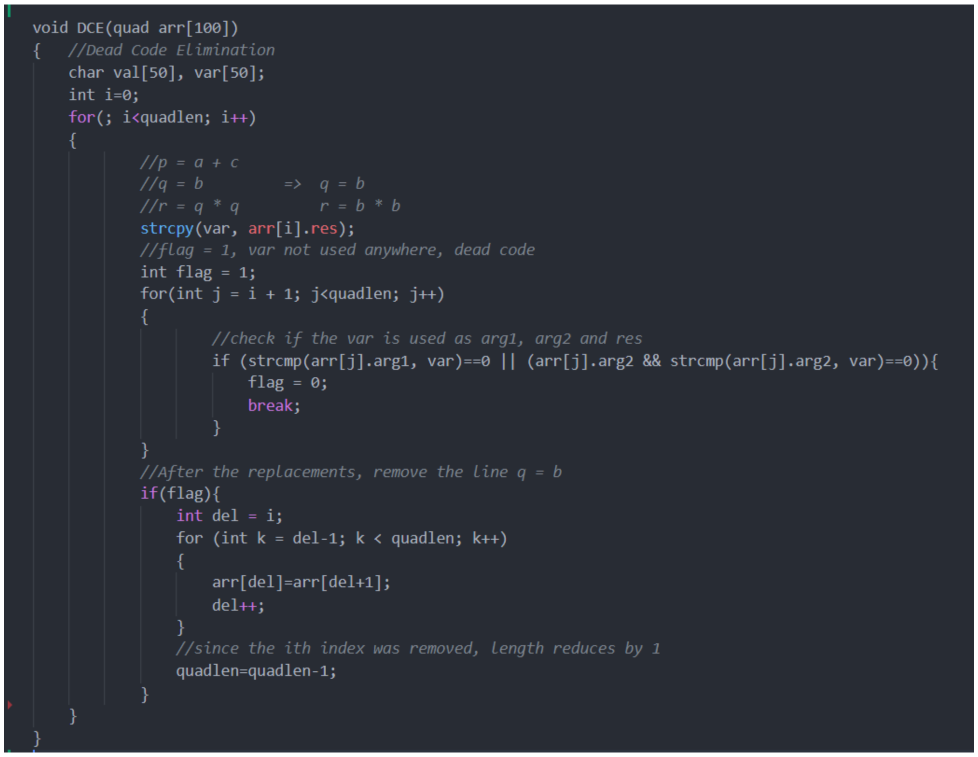
The following methods are used for Code optimization:

1. **Copy Propagation:**
   a. For each statement of type q = *variable*, lines following that statement are checked to see whether variable q is being used.
   b. If the variable q is being used as arg1 or arg2 in any statement then that occurrence of q is replaced with *variable*
   c. If the value of variable q is changed in any of the statements, i.e if q is used as the res, then the current copy propagation is stopped
   d. After changing all the valid occurrences of variable q, the statement q = *variable* is removed



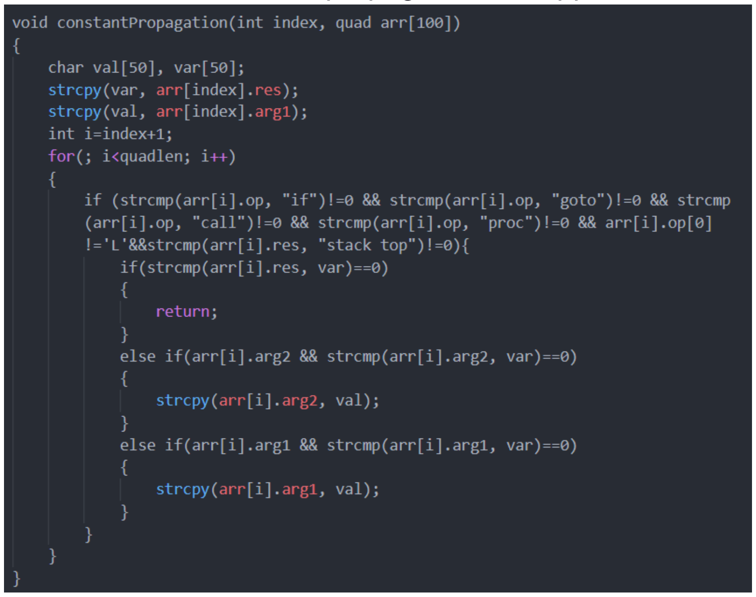
```
void copyPropagation(quad arr[100])
{
    char val[50], var[50];
    int i=0;
    for(; i<quadlen; i++)
    {
        //If arg2 is null; ex q = b -> res = q, arg1 = b, op = '='
        if(!arr[i].arg2){
            strcpy(var, arr[i].res);
            strcpy(val, arr[i].arg1);
            //Check if arg1 is var
            int varCheck = checkForDigits(arr[i].arg1);
            if(varCheck==0)
            {
                //flag to see if any arg was changed
                int flag = 0;
                for(int j = i + 1; j<quadlen; j++)
                {
                        //If value of the var is changed somewhere then the copy propogation must end
                        if(strcmp(arr[j].res, var)==0){
                            break;
                        }
                        //r = q * q; replace first occurence of q with b
                        if (strcmp(arr[j].arg1, var)==0){
                            strcpy(arr[j].arg1, val);
                            flag = 1;
                        }
                        //replace second occurance with b
                        if (arr[j].arg2 && strcmp(arr[j].arg2, var)==0){
                            strcpy(arr[j].arg2, val);
                            flag = 1;
                        }
                }
                //After the replacements, remove the line q = b
                if(flag){
                    int del = i;
                    for (int k = del-1; k < quadlen; k++)
                    {
                        arr[del]=arr[del+1];
                        del++;
                    }
                    //since the ith index was removed, length reduces by 1
                    quadlen=quadlen-1;
                }
            }
        }
    }
}
```

2. **Dead Code Removal:**
   a. For each statement of type q = expression or q = constant, lines following that statement are checked to see whether q is being used or not and if that statement is reachable
   b. A flag = 1 is declared to do so
   c. If the value of q is being used either as arg1 or arg2 in the valid scope then the flag is changed to 0
   d. After checking all the statements, if the flag is 1 then the statement is a *dead code* and hence removed



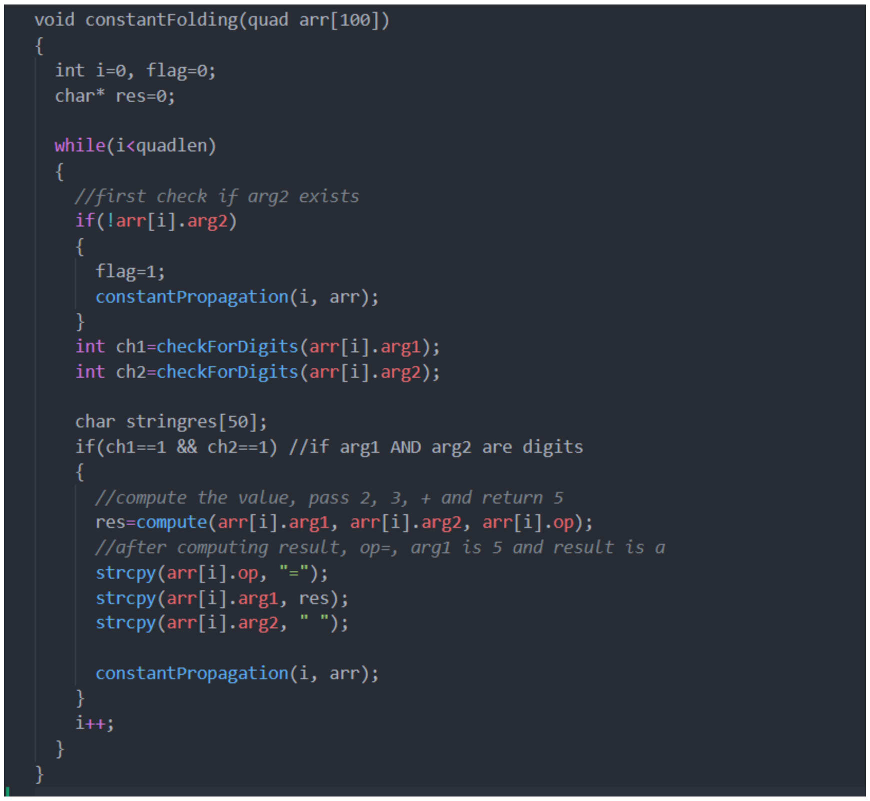
```
void DCE(quad arr[100])
{   //Dead Code Elimination
    char val[50], var[50];
    int i=0;
    for(; i<quadlen; i++)
    {
            //p = a + c
            //q = b         =>  q = b
            //r = q * q         r = b * b
            strcpy(var, arr[i].res);
            //flag = 1, var not used anywhere, dead code
            int flag = 1;
            for(int j = i + 1; j<quadlen; j++)
            {
                //check if the var is used as arg1, arg2 and res
                if (strcmp(arr[j].arg1, var)==0 || (arr[j].arg2 && strcmp(arr[j].arg2, var)==0)){
                    flag = 0;
                    break;
                }
            }
            //After the replacements, remove the line q = b
            if(flag){
                int del = i;
                for (int k = del-1; k < quadlen; k++)
                {
                    arr[del]=arr[del+1];
                    del++;
                }
                //since the ith index was removed, length reduces by 1
                quadlen=quadlen-1;
            }
    }
}
```

3. **Constant Propagation:**
    a. For each statement q = *constant*, the following statements are checked to see whether variable q is being used
    b. If the variable q is being used as arg1 or arg2 in any statement then that occurrence of q is replaced with *constant*
    c. If the value of variable q is changed in any of the statements, i.e if q is used as the res, then the current constant propagation is stopped



```
void constantPropagation(int index, quad arr[100])
{
    char val[50], var[50];
    strcpy(var, arr[index].res);
    strcpy(val, arr[index].arg1);
    int i=index+1;
    for(; i<quadlen; i++)
    {
        if (strcmp(arr[i].op, "if")!=0 && strcmp(arr[i].op, "goto")!=0 && strcmp
        (arr[i].op, "call")!=0 && strcmp(arr[i].op, "proc")!=0 && arr[i].op[0]
        !='L'&&strcmp(arr[i].res, "stack top")!=0){
            if(strcmp(arr[i].res, var)==0)
            {
                return;
            }
            else if(arr[i].arg2 && strcmp(arr[i].arg2, var)==0)
            {
                strcpy(arr[i].arg2, val);
            }
            else if(arr[i].arg1 && strcmp(arr[i].arg1, var)==0)
            {
                strcpy(arr[i].arg1, val);
            }
        }
    }
}
```

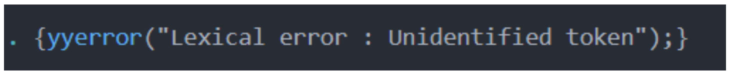
4. **Constant Folding:**
   a. For each statement q = a OP b, check if a and b are digits
   b. If both are constants then the compute() function is called to calculate the value of q
   c. Since the statement is changed to q = *constant*, constant propagation is done for the statement



```
void constantFolding(quad arr[100])
{
  int i=0, flag=0;
  char* res=0;

  while(i<quadlen)
  {
    //first check if arg2 exists
    if(!arr[i].arg2)
    {
      flag=1;
      constantPropagation(i, arr);
    }
    int ch1=checkForDigits(arr[i].arg1);
    int ch2=checkForDigits(arr[i].arg2);

    char stringres[50];
    if(ch1==1 && ch2==1) //if arg1 AND arg2 are digits
    {
      //compute the value, pass 2, 3, + and return 5
      res=compute(arr[i].arg1, arr[i].arg2, arr[i].op);
      //after computing result, op=, arg1 is 5 and result is a
      strcpy(arr[i].op, "=");
      strcpy(arr[i].arg1, res);
      strcpy(arr[i].arg2, " ");

      constantPropagation(i, arr);
    }
    i++;
  }
}
```

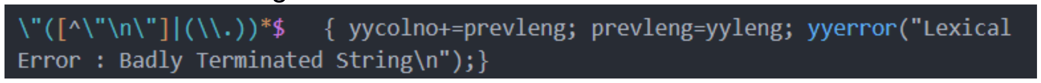
d. ERROR HANDLING

1. **Lexical Errors:**



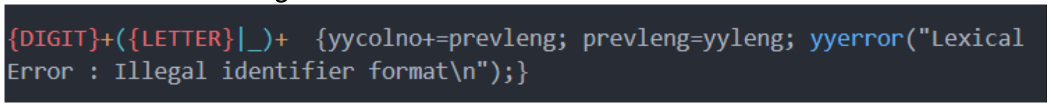
```
. {yyerror("Lexical error : Unidentified token");}
```

   a. **Badly terminated string** - Regex used to catch badly terminated string input from the Scanner stage.



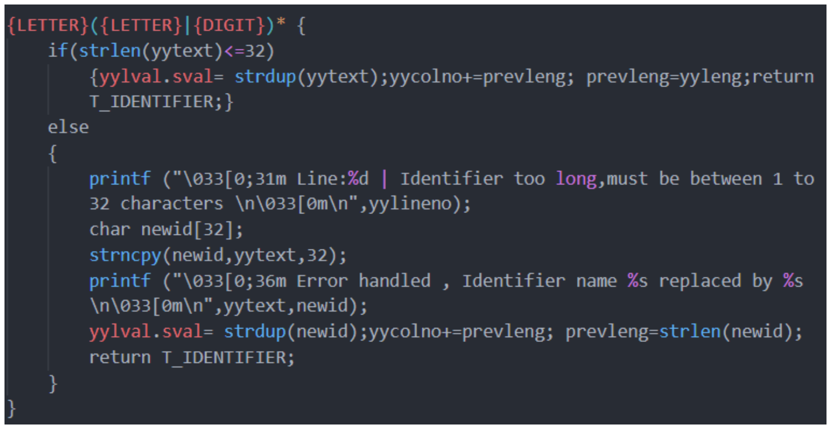
```
\"([^\"\n\"]|(\\.))*$   { yycolno+=prevleng; prevleng=yyleng; yyerror("Lexical
Error : Badly Terminated String\n");}
```

   b. **Badly formatted identifier** - Regex used to catch badly formatted identifiers from the Scanner stage.



```
{DIGIT}+({LETTER}|_)+  {yycolno+=prevleng; prevleng=yyleng; yyerror("Lexical
Error : Illegal identifier format\n");}
```

c. **Extra Long identifiers -** Length of identifier checked using basic string operations. Error handling performed by cutting shirt identifier to not exceed 32 characters.

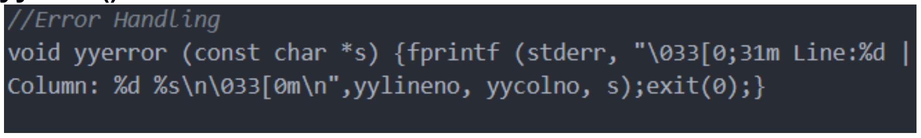


```
{LETTER}({LETTER}|{DIGIT})* {
    if(strlen(yytext)<=32)
        {yylval.sval= strdup(yytext);yycolno+=prevleng; prevleng=yyleng;return
        T_IDENTIFIER;}
    else
    {
        printf ("\033[0;31m Line:%d | Identifier too long,must be between 1 to
        32 characters \n\033[0m\n",yylineno);
        char newid[32];
        strncpy(newid,yytext,32);
        printf ("\033[0;36m Error handled , Identifier name %s replaced by %s
        \n\033[0m\n",yytext,newid);
        yylval.sval= strdup(newid);yycolno+=prevleng; prevleng=strlen(newid);
        return T_IDENTIFIER;
    }
}
```

2. **Syntax Errors:**

a. Error in syntax is highlighted with a respective message to indicate where the error occurred using line number and column number. Error Verbose is used to get customised error messages to help the programmer better understand the error. Syntax errors are caught based on the rules defined for the language. If a statement fails to match any of the production rules it is counted as an error.

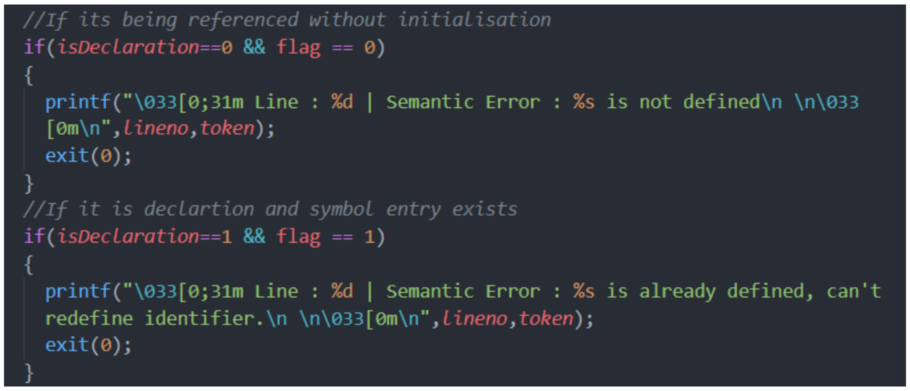
yyerror() definition:



```
//Error Handling
void yyerror (const char *s) {fprintf (stderr, "\033[0;31m Line:%d |
Column: %d %s\n\033[0m\n",yylineno, yycolno, s);exit(0);}
```

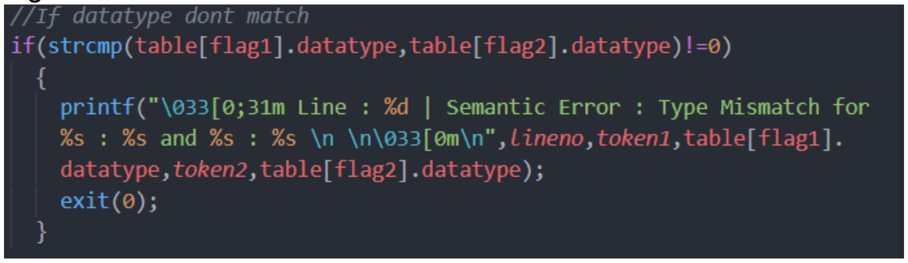
3. **Semantic Errors:**

a. Variable existential validation: The isDeclaration parameter passed is used to identify if the statement the identifier occurs is a declaration or not. If it is a declaration and the symbol entry exists this implies a semantic error for redeclaration of variables. If it is not a declaration and the symbol entry doesn't exist it implies that an undefined identifier is being referenced which is also an indication of a semantic error.



```
//If its being referenced without initialisation
if(isDeclaration==0 && flag == 0)
{
  printf("\033[0;31m Line : %d | Semantic Error : %s is not defined\n \n\033
  [0m\n",lineno,token);
  exit(0);
}
//If it is declartion and symbol entry exists
if(isDeclaration==1 && flag == 1)
{
  printf("\033[0;31m Line : %d | Semantic Error : %s is already defined, can't
  redefine identifier.\n \n\033[0m\n",lineno,token);
  exit(0);
}
```

b. **Type checking**: Every time a math operation is executed in the parser, the variables involved in the operation are sent to the type_check() function to check if the data type on the left hand side of the '=' is the same as that of the right hand side. Else a semantic error is raised.



```
//If datatype dont match
if(strcmp(table[flag1].datatype,table[flag2].datatype)!=0)
  {
    printf("\033[0;31m Line : %d | Semantic Error : Type Mismatch for
    %s : %s and %s : %s \n \n\033[0m\n",lineno,token1,table[flag1].
    datatype,token2,table[flag2].datatype);
    exit(0);
  }
```

e. Instructions to build and run the code

➔ To ensure the code works well, you need make, g++, flex/lex and bison installed in your system.

➔ Clone the repository https://github.com/sanskritip/mini-golang-compiler into your local system. The project has a single global lex file, while has separate yacc files for each of the phases to ensure code cleanliness and ease of maintenance.

➔ To build any of the phases, from a terminal, go into the directory and execute the makeFile using the `make` command. It then builds the compiler into an executable in the name of `gocompiler` in the folder.

➔ To run the compiler with an input file testinput.go, from the same directory as the previous step run `./gocompiler testinput.go`

➔ To help execute and test the compiler quickly, there are a set of prebuilt scripts in the scripts/ folder which will help build and execute predefined test cases using the compiler. Run `sh <script-name>.sh`

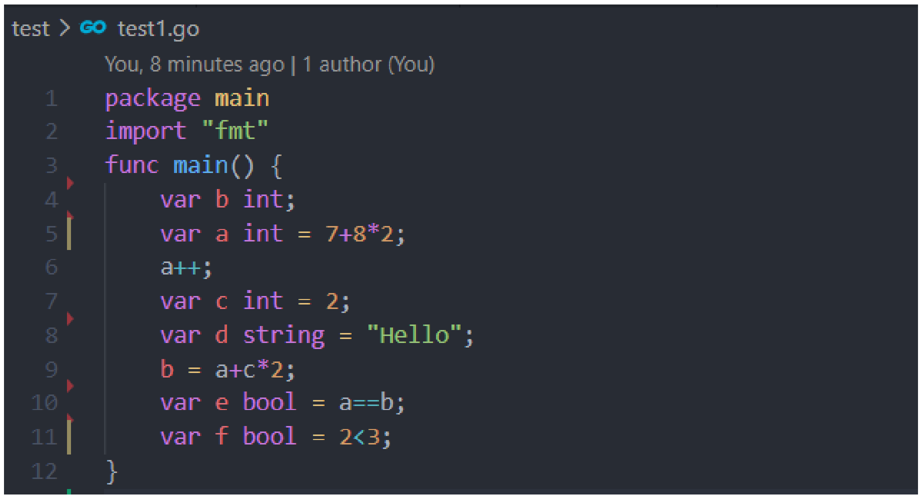
● RESULTS AND SHORTCOMINGS

- Milestones achieved:
  - Scanning to recognise Tokens
  - Parsing to identify Code construct based on CFG
  - Arithmetic and Relational Expression Computation
  - Arithmetic and Relational Expression AST Generation
  - Symbol Table Generation
  - Intermediate Code Generation for Arithmetic and Relational Expressions
  - Intermediate Code Generation for FOR loops
  - Code Optimisation for Arithmetic and Relational Expressions
- Shortcomings
  - Code generation and Optimisation for IF Statements
  - Shift/Reduce and Reduce/Reduce Conflicts
  - Symbol Table updation for Scope, References etc.

- **SNAPSHOTS :**

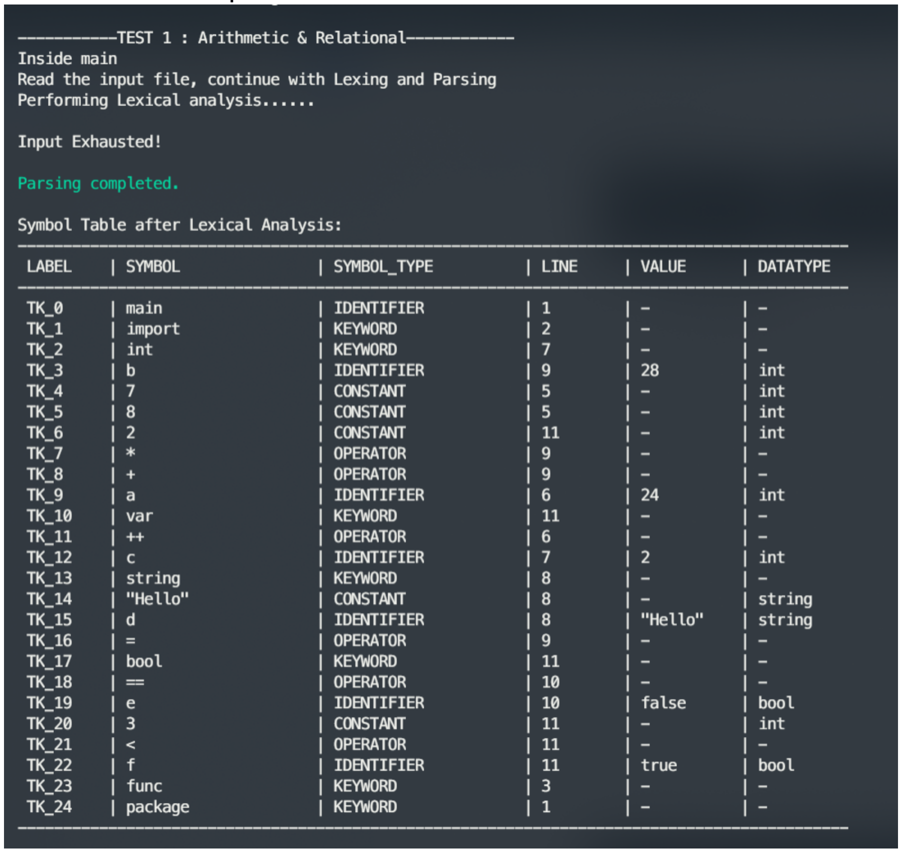
1. SYMBOL TABLE CREATION

- Test Case 1 (Arithmetic and Relational Operations):
  - Test Case GO File



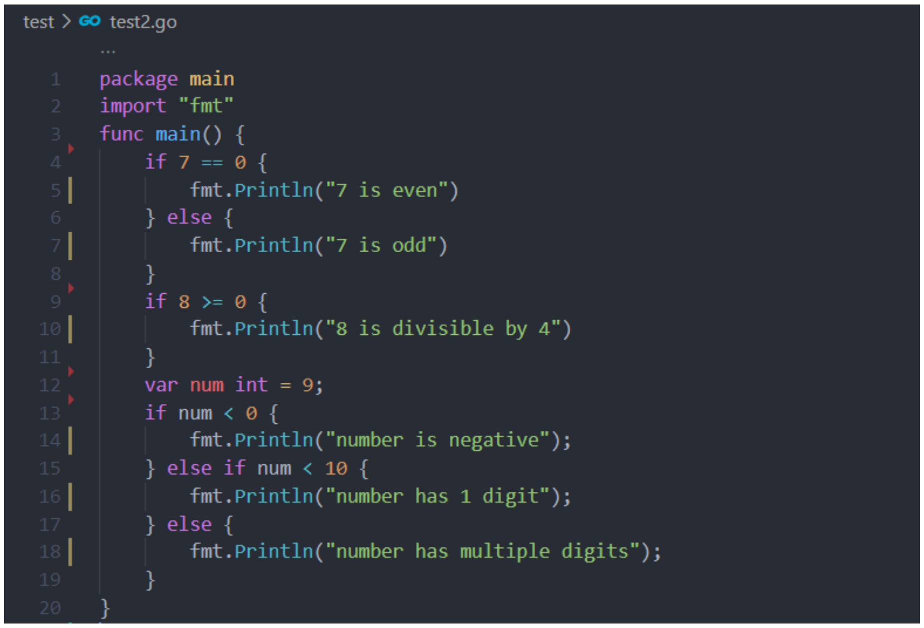
```
test > test1.go
You, 8 minutes ago | 1 author (You)
package main
import "fmt"
func main() {
    var b int;
    var a int = 7+8*2;
    a++;
    var c int = 2;
    var d string = "Hello";
    b = a+c*2;
    var e bool = a==b;
    var f bool = 2<3;
}
```

  - Test Case Output



```
-----------TEST 1 : Arithmetic & Relational-----------
Inside main
Read the input file, continue with Lexing and Parsing
Performing Lexical analysis......

Input Exhausted!

Parsing completed.

Symbol Table after Lexical Analysis:
-------------------------------------------------------------------------------------------
 LABEL    | SYMBOL                 | SYMBOL_TYPE            | LINE        | VALUE        | DATATYPE
-------------------------------------------------------------------------------------------
 TK_0     | main                   | IDENTIFIER             | 1           | -            | -
 TK_1     | import                 | KEYWORD                | 2           | -            | -
 TK_2     | int                    | KEYWORD                | 7           | -            | -
 TK_3     | b                      | IDENTIFIER             | 9           | 28           | int
 TK_4     | 7                      | CONSTANT               | 5           | -            | int
 TK_5     | 8                      | CONSTANT               | 5           | -            | int
 TK_6     | 2                      | CONSTANT               | 11          | -            | int
 TK_7     | *                      | OPERATOR               | 9           | -            | -
 TK_8     | +                      | OPERATOR               | 9           | -            | -
 TK_9     | a                      | IDENTIFIER             | 6           | 24           | int
 TK_10    | var                    | KEYWORD                | 11          | -            | -
 TK_11    | ++                     | OPERATOR               | 6           | -            | -
 TK_12    | c                      | IDENTIFIER             | 7           | 2            | int
 TK_13    | string                 | KEYWORD                | 8           | -            | -
 TK_14    | "Hello"                | CONSTANT               | 8           | -            | string
 TK_15    | d                      | IDENTIFIER             | 8           | "Hello"      | string
 TK_16    | =                      | OPERATOR               | 9           | -            | -
 TK_17    | bool                   | KEYWORD                | 11          | -            | -
 TK_18    | ==                     | OPERATOR               | 10          | -            | -
 TK_19    | e                      | IDENTIFIER             | 10          | false        | bool
 TK_20    | 3                      | CONSTANT               | 11          | -            | int
 TK_21    | <                      | OPERATOR               | 11          | -            | -
 TK_22    | f                      | IDENTIFIER             | 11          | true         | bool
 TK_23    | func                   | KEYWORD                | 3           | -            | -
 TK_24    | package                | KEYWORD                | 1           | -            | -
-------------------------------------------------------------------------------------------
```

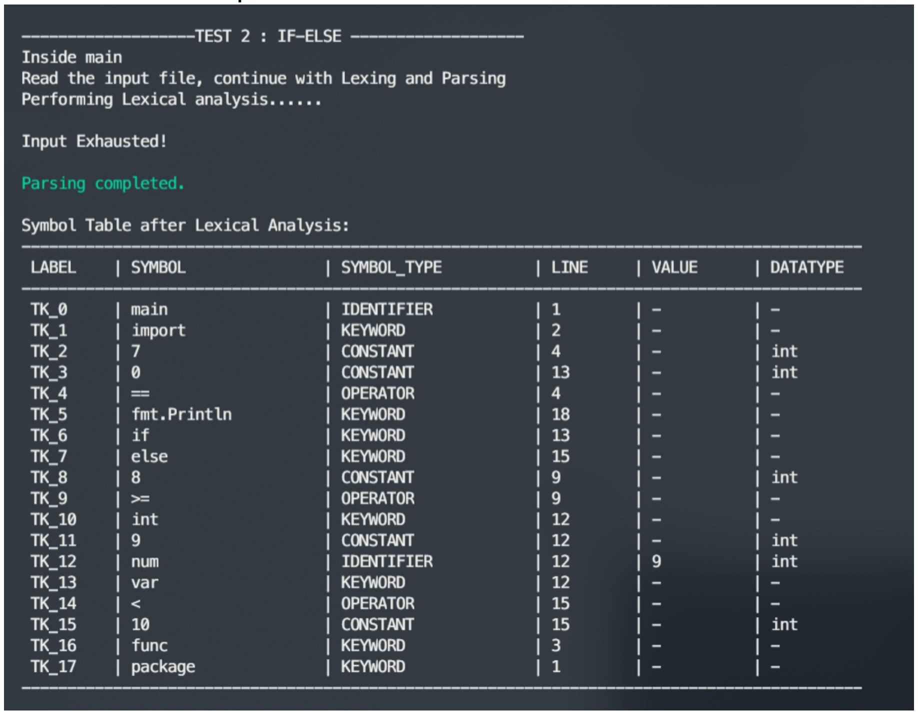
- Test Case 2 (If, If-else, and If-Else-IF Statements):
    - Test Case GO File



test > GO test2.go

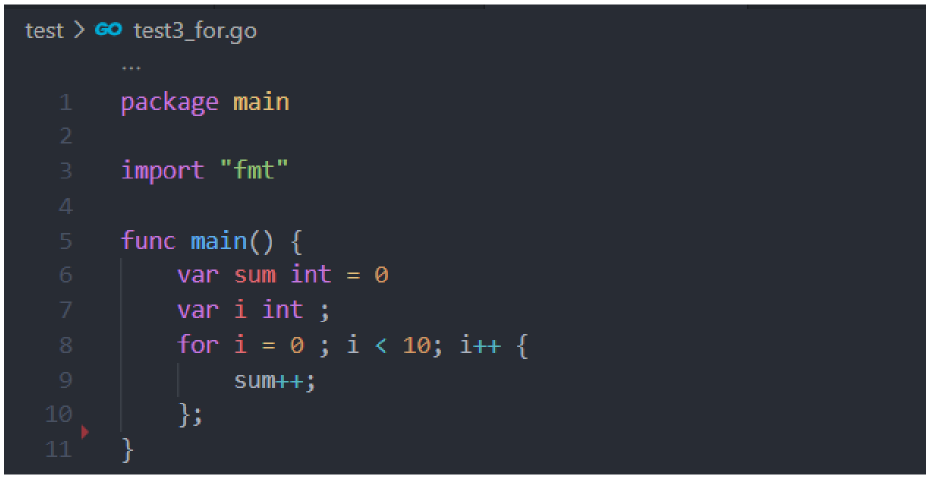
```go
...
package main
import "fmt"
func main() {
    if 7 == 0 {
        fmt.Println("7 is even")
    } else {
        fmt.Println("7 is odd")
    }
    if 8 >= 0 {
        fmt.Println("8 is divisible by 4")
    }
    var num int = 9;
    if num < 0 {
        fmt.Println("number is negative");
    } else if num < 10 {
        fmt.Println("number has 1 digit");
    } else {
        fmt.Println("number has multiple digits");
    }
}
```

    - Test Case Output



```
-------------------TEST 2 : IF-ELSE -------------------
Inside main
Read the input file, continue with Lexing and Parsing
Performing Lexical analysis......

Input Exhausted!

Parsing completed.

Symbol Table after Lexical Analysis:
--------------------------------------------------------------------------------------------
 LABEL     | SYMBOL                 | SYMBOL_TYPE           | LINE      | VALUE       | DATATYPE
--------------------------------------------------------------------------------------------
 TK_0      | main                   | IDENTIFIER            | 1         | -           | -
 TK_1      | import                 | KEYWORD               | 2         | -           | -
 TK_2      | 7                      | CONSTANT              | 4         | -           | int
 TK_3      | 0                      | CONSTANT              | 13        | -           | int
 TK_4      | ==                     | OPERATOR              | 4         | -           | -
 TK_5      | fmt.Println            | KEYWORD               | 18        | -           | -
 TK_6      | if                     | KEYWORD               | 13        | -           | -
 TK_7      | else                   | KEYWORD               | 15        | -           | -
 TK_8      | 8                      | CONSTANT              | 9         | -           | int
 TK_9      | >=                     | OPERATOR              | 9         | -           | -
 TK_10     | int                    | KEYWORD               | 12        | -           | -
 TK_11     | 9                      | CONSTANT              | 12        | -           | int
 TK_12     | num                    | IDENTIFIER            | 12        | 9           | int
 TK_13     | var                    | KEYWORD               | 12        | -           | -
 TK_14     | <                      | OPERATOR              | 15        | -           | -
 TK_15     | 10                     | CONSTANT              | 15        | -           | int
 TK_16     | func                   | KEYWORD               | 3         | -           | -
 TK_17     | package                | KEYWORD               | 1         | -           | -
--------------------------------------------------------------------------------------------
```

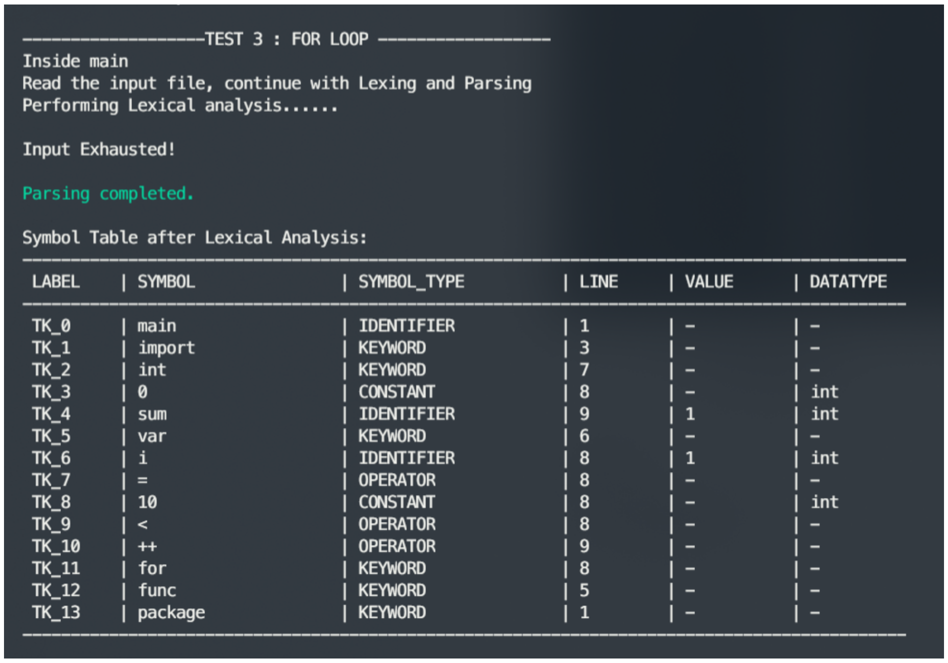
- Test Case 3 (For Loop):
  - Test Case GO File



test > test3_for.go

```go
package main

import "fmt"

func main() {
    var sum int = 0
    var i int ;
    for i = 0 ; i < 10; i++ {
        sum++;
    };
}
```

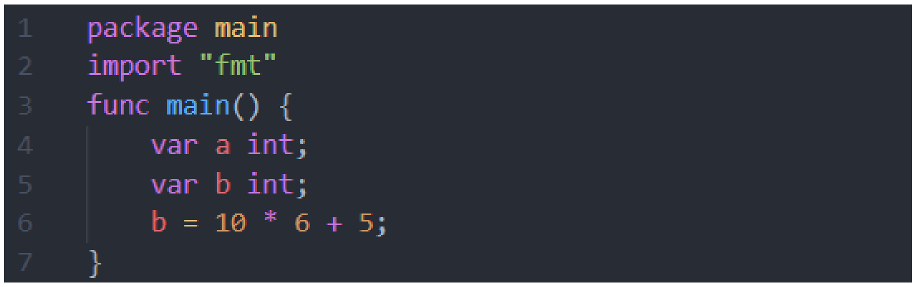
  - Test Case Output



```
------------------TEST 3 : FOR LOOP -----------------
Inside main
Read the input file, continue with Lexing and Parsing
Performing Lexical analysis......

Input Exhausted!

Parsing completed.

Symbol Table after Lexical Analysis:
-------------------------------------------------------------------------------------------
 LABEL     | SYMBOL                | SYMBOL_TYPE           | LINE      | VALUE       | DATATYPE
-------------------------------------------------------------------------------------------
 TK_0      | main                  | IDENTIFIER            | 1         | -           | -
 TK_1      | import                | KEYWORD               | 3         | -           | -
 TK_2      | int                   | KEYWORD               | 7         | -           | -
 TK_3      | 0                     | CONSTANT              | 8         | -           | int
 TK_4      | sum                   | IDENTIFIER            | 9         | 1           | int
 TK_5      | var                   | KEYWORD               | 6         | -           | -
 TK_6      | i                     | IDENTIFIER            | 8         | 1           | int
 TK_7      | =                     | OPERATOR              | 8         | -           | -
 TK_8      | 10                    | CONSTANT              | 8         | -           | int
 TK_9      | <                     | OPERATOR              | 8         | -           | -
 TK_10     | ++                    | OPERATOR              | 9         | -           | -
 TK_11     | for                   | KEYWORD               | 8         | -           | -
 TK_12     | func                  | KEYWORD               | 5         | -           | -
 TK_13     | package               | KEYWORD               | 1         | -           | -
-------------------------------------------------------------------------------------------
```

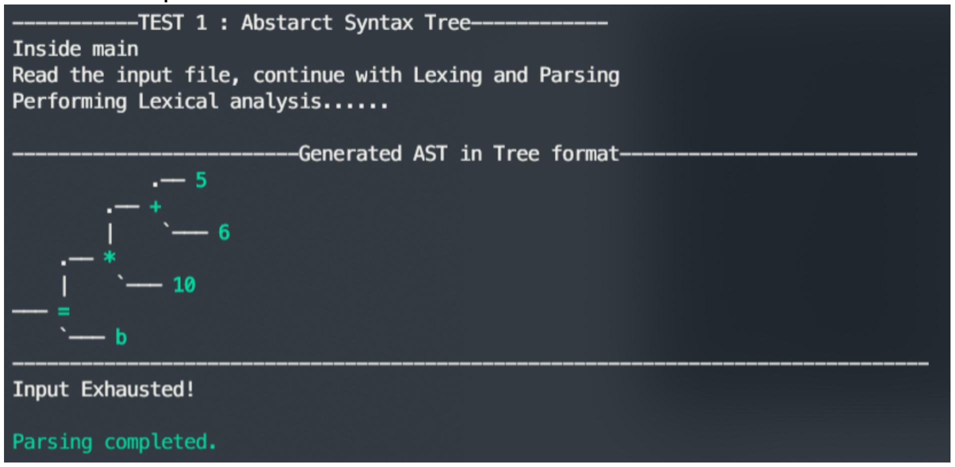
2. AST

- Test Case GO File



```go
package main
import "fmt"
func main() {
    var a int;
    var b int;
    b = 10 * 6 + 5;
}
```

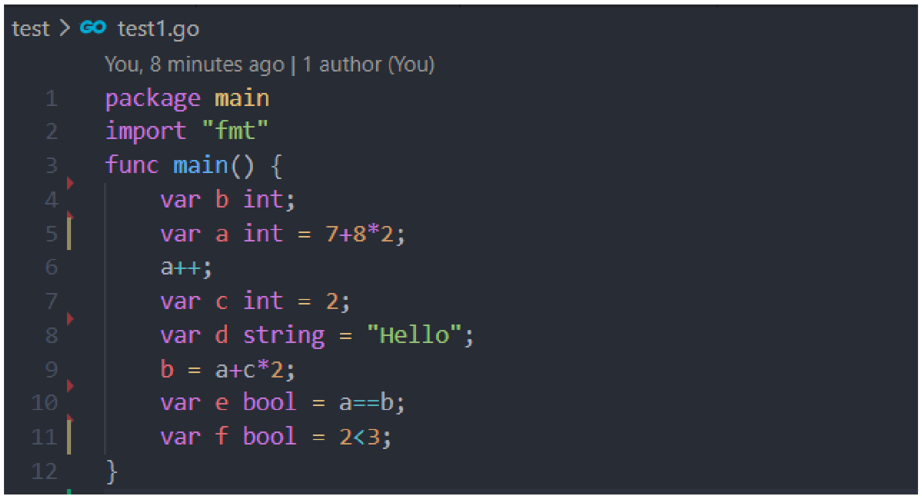
- Test Case Output



```
-----------TEST 1 : Abstarct Syntax Tree------------
Inside main
Read the input file, continue with Lexing and Parsing
Performing Lexical analysis......

------------------------Generated AST in Tree format-------------------------
            .—— 5
        .—— +
        |   `—— 6
    .—— *
    |   `—— 10
——— =
    `—— b
-----------------------------------------------------------------------------
Input Exhausted!

Parsing completed.
```

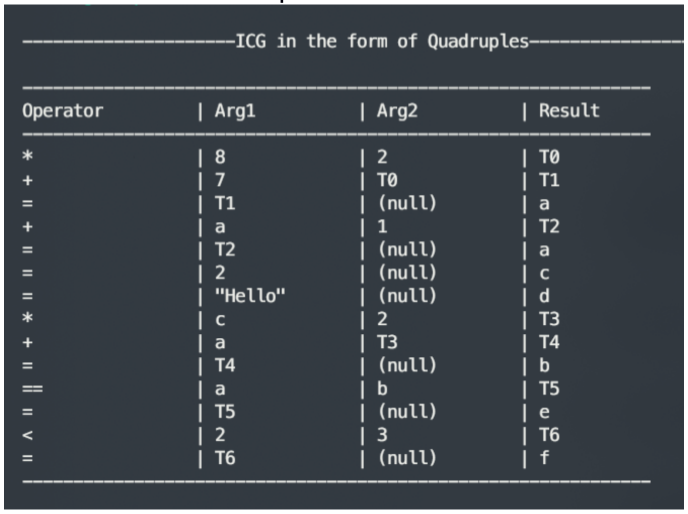
3. INTERMEDIATE CODE GENERATION

- Test 1 (Arithmetic and Relational Operators):
  - Test Case GO File

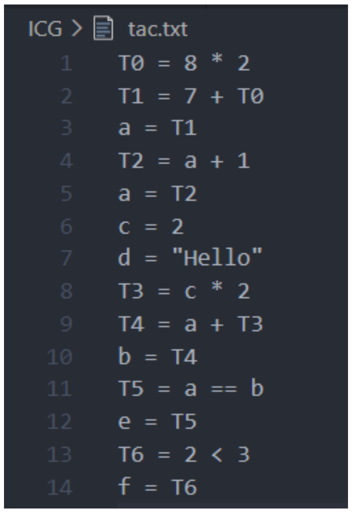


```go
test > test1.go
You, 8 minutes ago | 1 author (You)
package main
import "fmt"
func main() {
    var b int;
    var a int = 7+8*2;
    a++;
    var c int = 2;
    var d string = "Hello";
    b = a+c*2;
    var e bool = a==b;
    var f bool = 2<3;
}
```

- Test Case Output

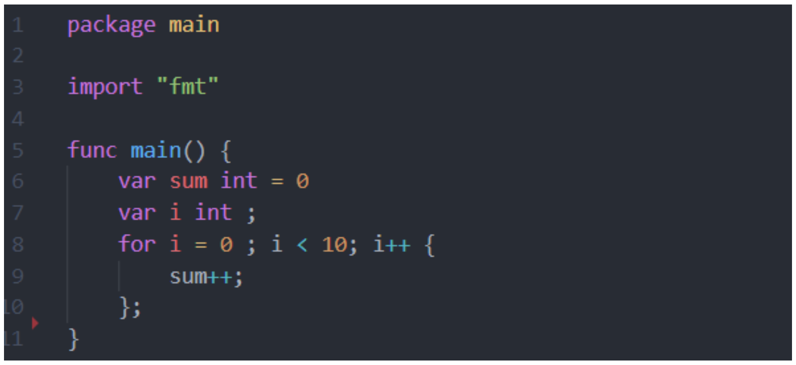


```
--------------------ICG in the form of Quadruples--------------
-------------------------------------------------------------
Operator        | Arg1        | Arg2        | Result
-------------------------------------------------------------
*               | 8           | 2           | T0
+               | 7           | T0          | T1
=               | T1          | (null)      | a
+               | a           | 1           | T2
=               | T2          | (null)      | a
=               | 2           | (null)      | c
=               | "Hello"     | (null)      | d
*               | c           | 2           | T3
+               | a           | T3          | T4
=               | T4          | (null)      | b
==              | a           | b           | T5
=               | T5          | (null)      | e
<               | 2           | 3           | T6
=               | T6          | (null)      | f
-------------------------------------------------------------
```



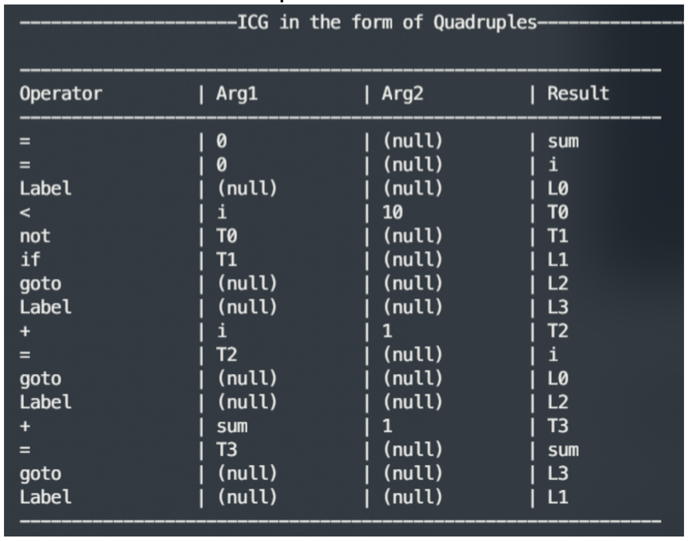
```
ICG > tac.txt
T0 = 8 * 2
T1 = 7 + T0
a = T1
T2 = a + 1
a = T2
c = 2
d = "Hello"
T3 = c * 2
T4 = a + T3
b = T4
T5 = a == b
e = T5
T6 = 2 < 3
f = T6
```

- Test 2 (For Loop):
  - Test Case GO File

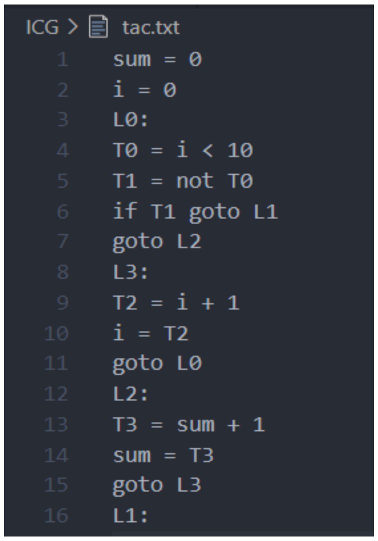


```go
package main

import "fmt"

func main() {
    var sum int = 0
    var i int ;
    for i = 0 ; i < 10; i++ {
        sum++;
    };
}
```

  - Test Case Output



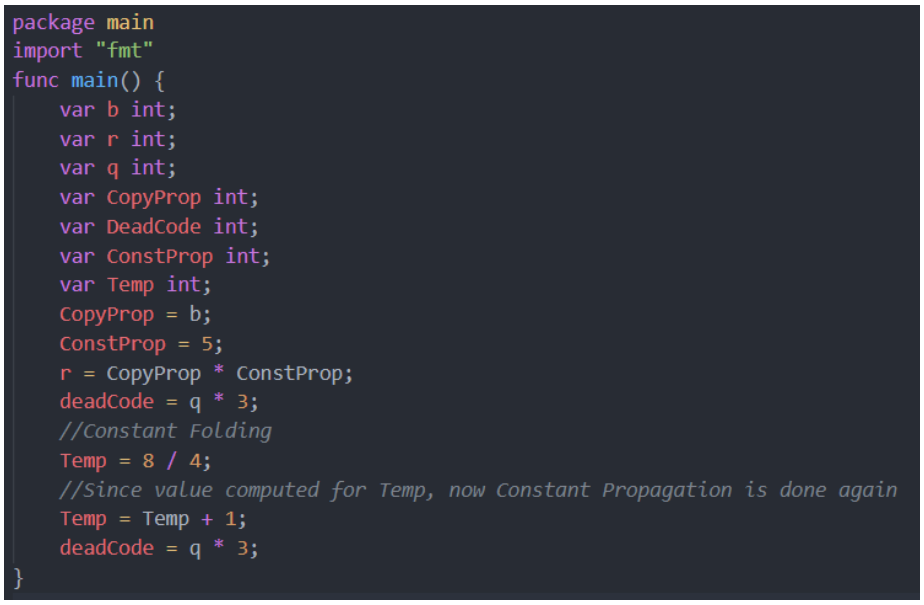
```
--------------------ICG in the form of Quadruples--------------
-------------------------------------------------------------
Operator        | Arg1        | Arg2        | Result
-------------------------------------------------------------
=               | 0           | (null)      | sum
=               | 0           | (null)      | i
Label           | (null)      | (null)      | L0
<               | i           | 10          | T0
not             | T0          | (null)      | T1
if              | T1          | (null)      | L1
goto            | (null)      | (null)      | L2
Label           | (null)      | (null)      | L3
+               | i           | 1           | T2
=               | T2          | (null)      | i
goto            | (null)      | (null)      | L0
Label           | (null)      | (null)      | L2
+               | sum         | 1           | T3
=               | T3          | (null)      | sum
goto            | (null)      | (null)      | L3
Label           | (null)      | (null)      | L1
-------------------------------------------------------------
```



```
ICG > tac.txt
sum = 0
i = 0
L0:
T0 = i < 10
T1 = not T0
if T1 goto L1
goto L2
L3:
T2 = i + 1
i = T2
goto L0
L2:
T3 = sum + 1
sum = T3
goto L3
L1:
```

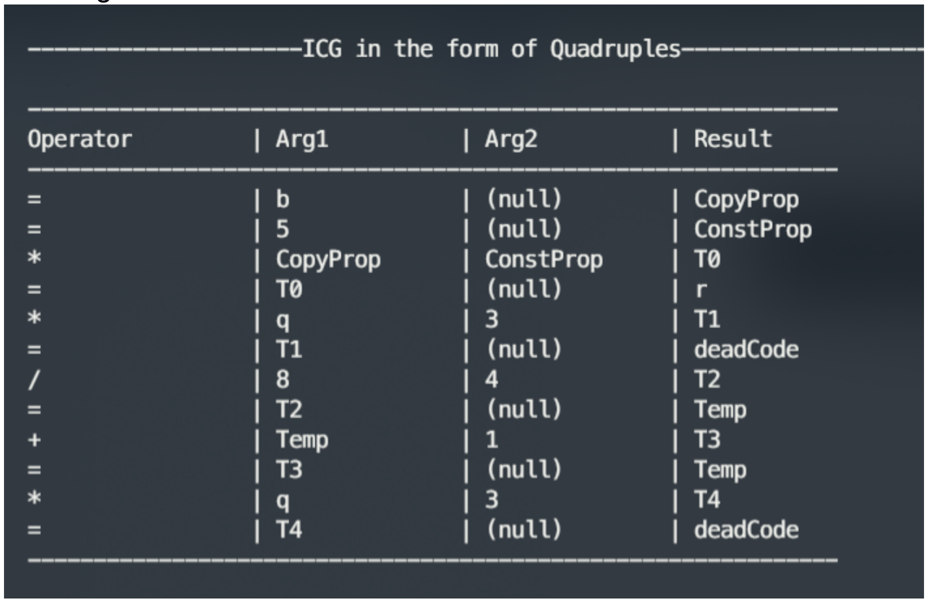
4. CODE OPTIMIZATION

- Test Case Go File:



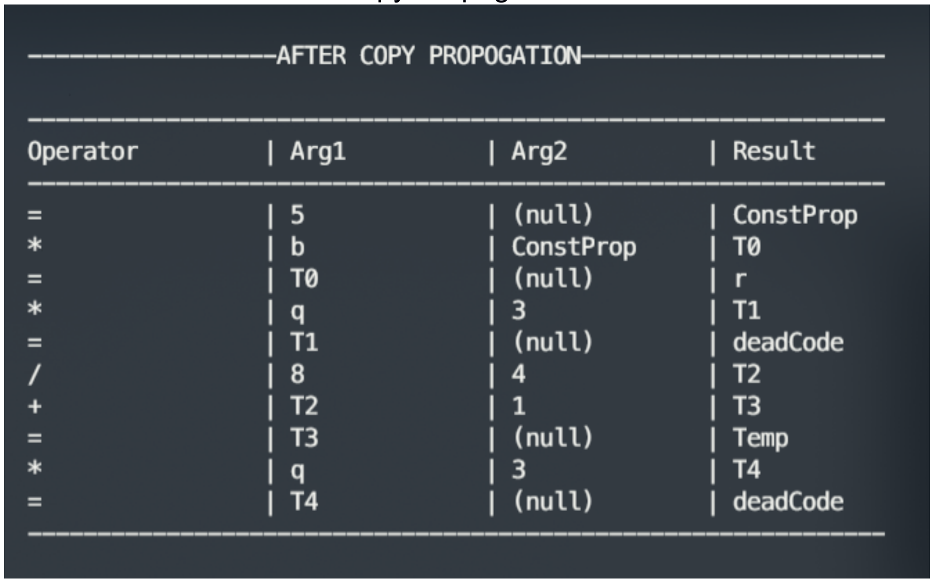
```
package main
import "fmt"
func main() {
    var b int;
    var r int;
    var q int;
    var CopyProp int;
    var DeadCode int;
    var ConstProp int;
    var Temp int;
    CopyProp = b;
    ConstProp = 5;
    r = CopyProp * ConstProp;
    deadCode = q * 3;
    //Constant Folding
    Temp = 8 / 4;
    //Since value computed for Temp, now Constant Propagation is done again
    Temp = Temp + 1;
    deadCode = q * 3;
}
```

- Original Intermediate Code Generated:



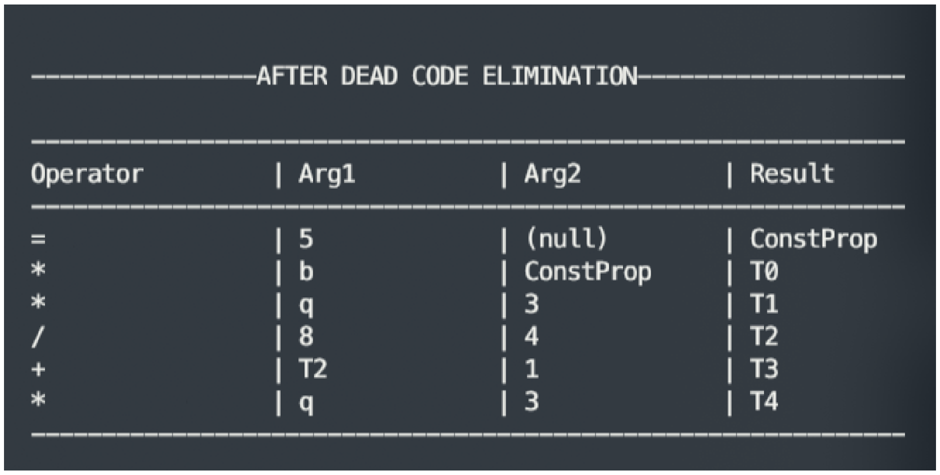
```
--------------------ICG in the form of Quadruples-----------------

---------------------------------------------------------------
Operator         | Arg1          | Arg2          | Result
---------------------------------------------------------------
=                | b             | (null)        | CopyProp
=                | 5             | (null)        | ConstProp
*                | CopyProp      | ConstProp     | T0
=                | T0            | (null)        | r
*                | q             | 3             | T1
=                | T1            | (null)        | deadCode
/                | 8             | 4             | T2
=                | T2            | (null)        | Temp
+                | Temp          | 1             | T3
=                | T3            | (null)        | Temp
*                | q             | 3             | T4
=                | T4            | (null)        | deadCode
---------------------------------------------------------------
```

- Intermediate Code after Copy Propagation:



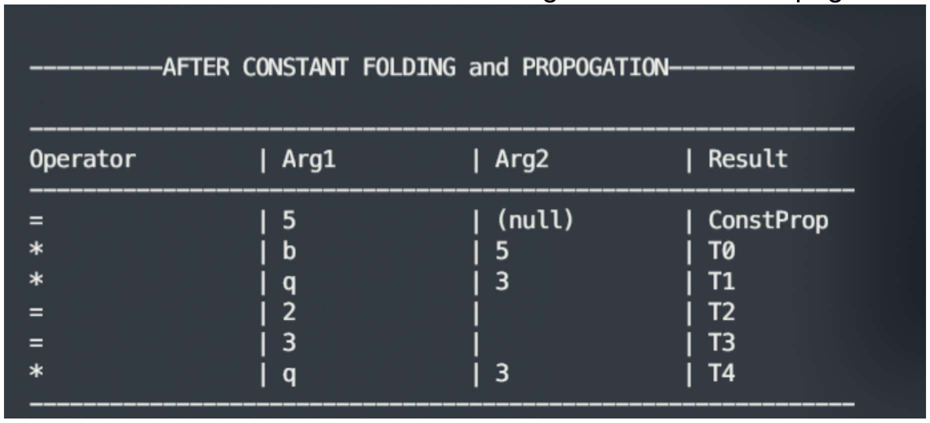
```
-----------------AFTER COPY PROPOGATION---------------------

-------------------------------------------------------------
Operator         | Arg1          | Arg2          | Result
-------------------------------------------------------------
=                | 5             | (null)        | ConstProp
*                | b             | ConstProp     | T0
=                | T0            | (null)        | r
*                | q             | 3             | T1
=                | T1            | (null)        | deadCode
/                | 8             | 4             | T2
+                | T2            | 1             | T3
=                | T3            | (null)        | Temp
*                | q             | 3             | T4
=                | T4            | (null)        | deadCode
-------------------------------------------------------------
```

- Intermediate Code after Dead Code Removal:



```
----------------AFTER DEAD CODE ELIMINATION-------------------

--------------------------------------------------------------
Operator         | Arg1          | Arg2          | Result
--------------------------------------------------------------
=                | 5             | (null)        | ConstProp
*                | b             | ConstProp     | T0
*                | q             | 3             | T1
/                | 8             | 4             | T2
+                | T2            | 1             | T3
*                | q             | 3             | T4
--------------------------------------------------------------
```

- Intermediate Code after Constant Folding and Constant Propagation:

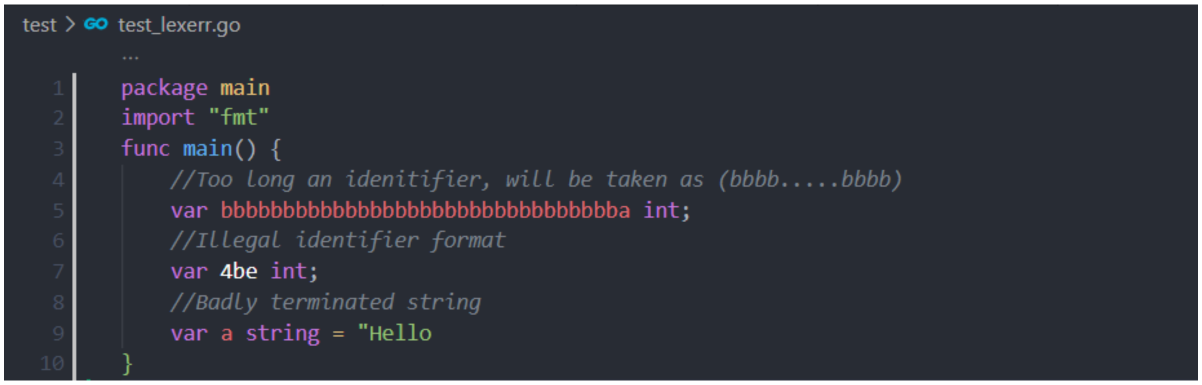


```
----------AFTER CONSTANT FOLDING and PROPOGATION--------------

-------------------------------------------------------------
Operator         | Arg1          | Arg2          | Result
-------------------------------------------------------------
=                | 5             | (null)        | ConstProp
*                | b             | 5             | T0
*                | q             | 3             | T1
=                | 2             |               | T2
=                | 3             |               | T3
*                | q             | 3             | T4
-------------------------------------------------------------
```

5. ERROR HANDLING

1. Lexical Errors

➔ Test Case File :

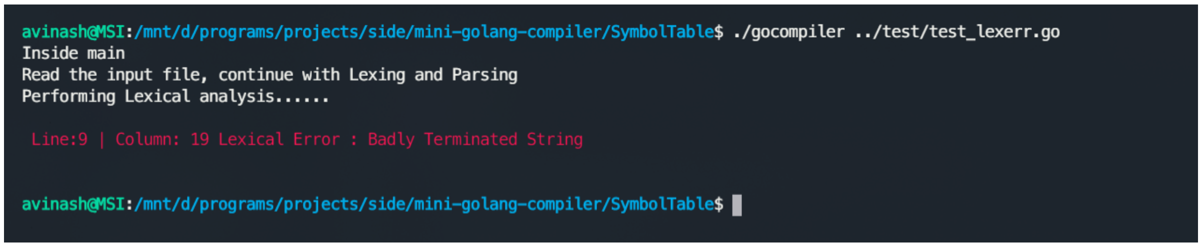


test > test_lexerr.go

```go
package main
import "fmt"
func main() {
    //Too long an idenitifier, will be taken as (bbbb.....bbbb)
    var bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbba int;
    //Illegal identifier format
    var 4be int;
    //Badly terminated string
    var a string = "Hello
}
```

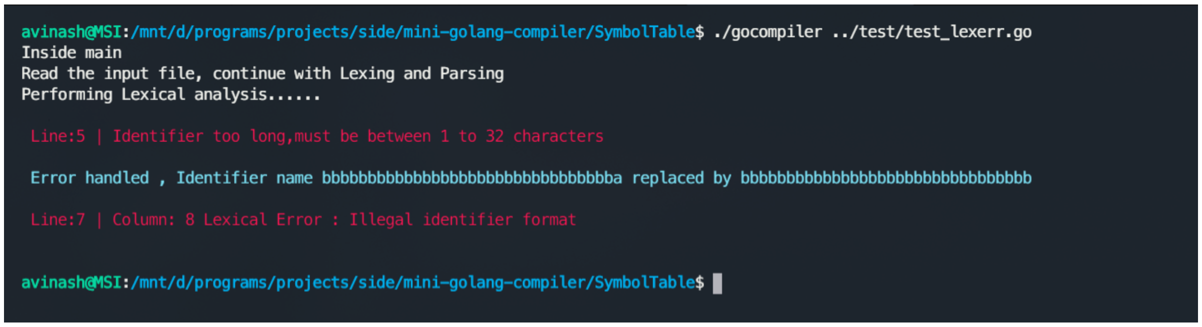
➔ Outputs of Lexical Error Reporting (Taken one at a time)

◆ Badly terminated String



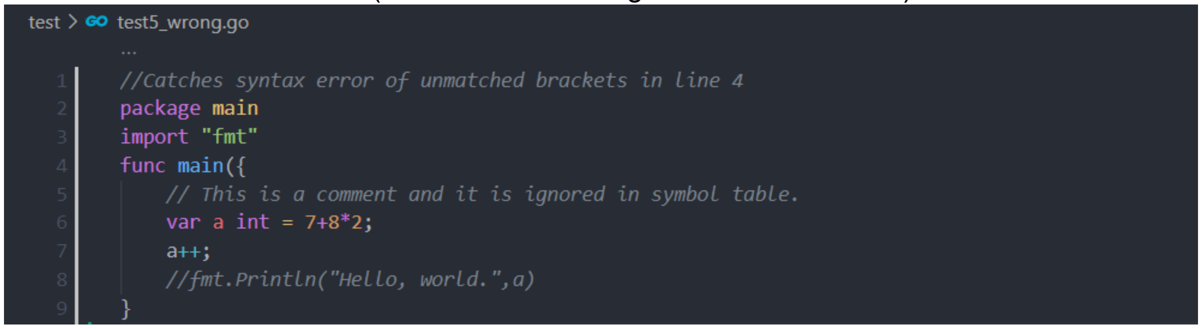
```
avinash@MSI:/mnt/d/programs/projects/side/mini-golang-compiler/SymbolTable$ ./gocompiler ../test/test_lexerr.go
Inside main
Read the input file, continue with Lexing and Parsing
Performing Lexical analysis......

 Line:9 | Column: 19 Lexical Error : Badly Terminated String


avinash@MSI:/mnt/d/programs/projects/side/mini-golang-compiler/SymbolTable$
```

◆ Illegal Identifier format and Extra Long Identifier



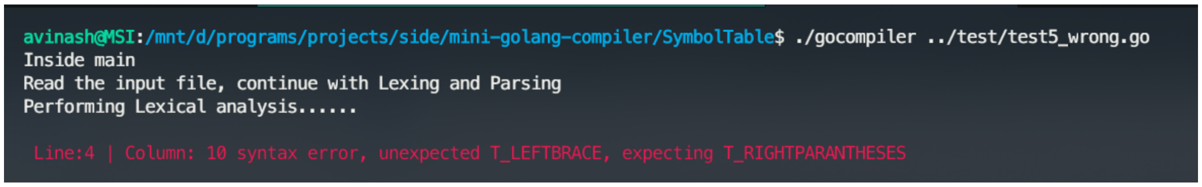
```
avinash@MSI:/mnt/d/programs/projects/side/mini-golang-compiler/SymbolTable$ ./gocompiler ../test/test_lexerr.go
Inside main
Read the input file, continue with Lexing and Parsing
Performing Lexical analysis......

 Line:5 | Identifier too long,must be between 1 to 32 characters

 Error handled , Identifier name bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbba replaced by bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb

 Line:7 | Column: 8 Lexical Error : Illegal identifier format


avinash@MSI:/mnt/d/programs/projects/side/mini-golang-compiler/SymbolTable$
```

2. Syntax Errors
   - ➔ Test Case File 1 (Parenthesis Missing - Bracket Mismatch)



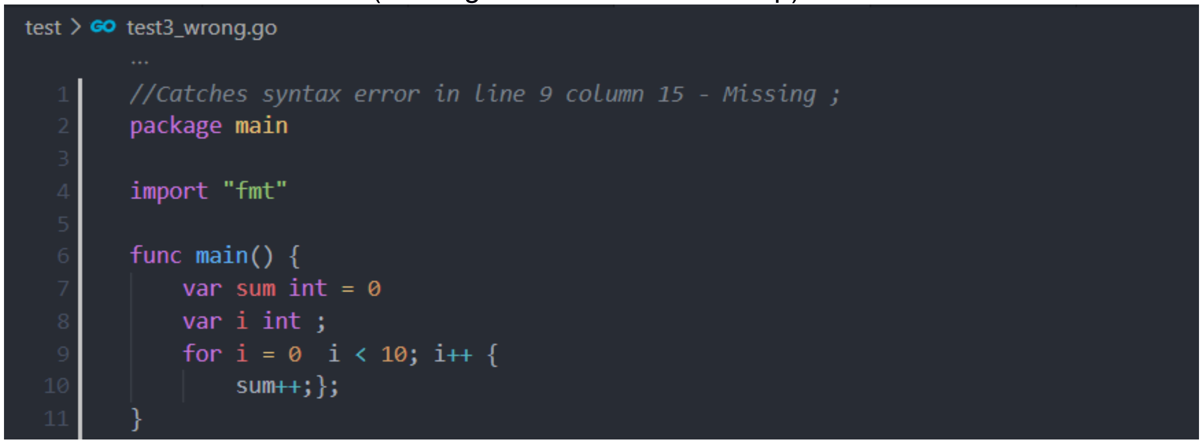
```
test > test5_wrong.go
  ...
1   //Catches syntax error of unmatched brackets in line 4
2   package main
3   import "fmt"
4   func main({
5       // This is a comment and it is ignored in symbol table.
6       var a int = 7+8*2;
7       a++;
8       //fmt.Println("Hello, world.",a)
9   }
```

   - ➔ Reporting Syntax Error



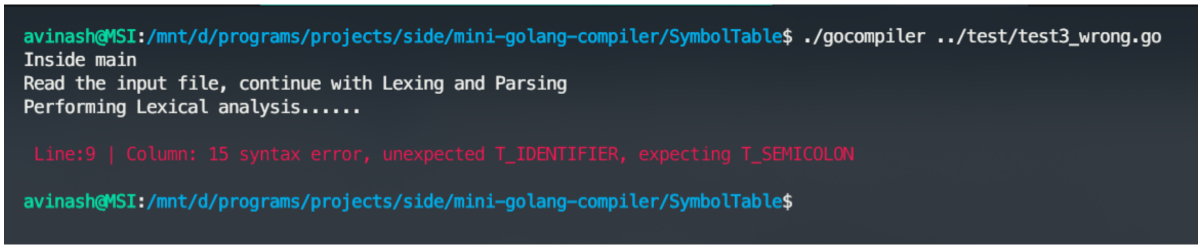
```
avinash@MSI:/mnt/d/programs/projects/side/mini-golang-compiler/SymbolTable$ ./gocompiler ../test/test5_wrong.go
Inside main
Read the input file, continue with Lexing and Parsing
Performing Lexical analysis......

 Line:4 | Column: 10 syntax error, unexpected T_LEFTBRACE, expecting T_RIGHTPARANTHESES
```

   - ➔ Test Case File 2 (Missing Semicolon in FOR loop)



```
test > test3_wrong.go
  ...
1   //Catches syntax error in line 9 column 15 - Missing ;
2   package main
3
4   import "fmt"
5
6   func main() {
7       var sum int = 0
8       var i int ;
9       for i = 0  i < 10; i++ {
10          sum++;};
11  }
```

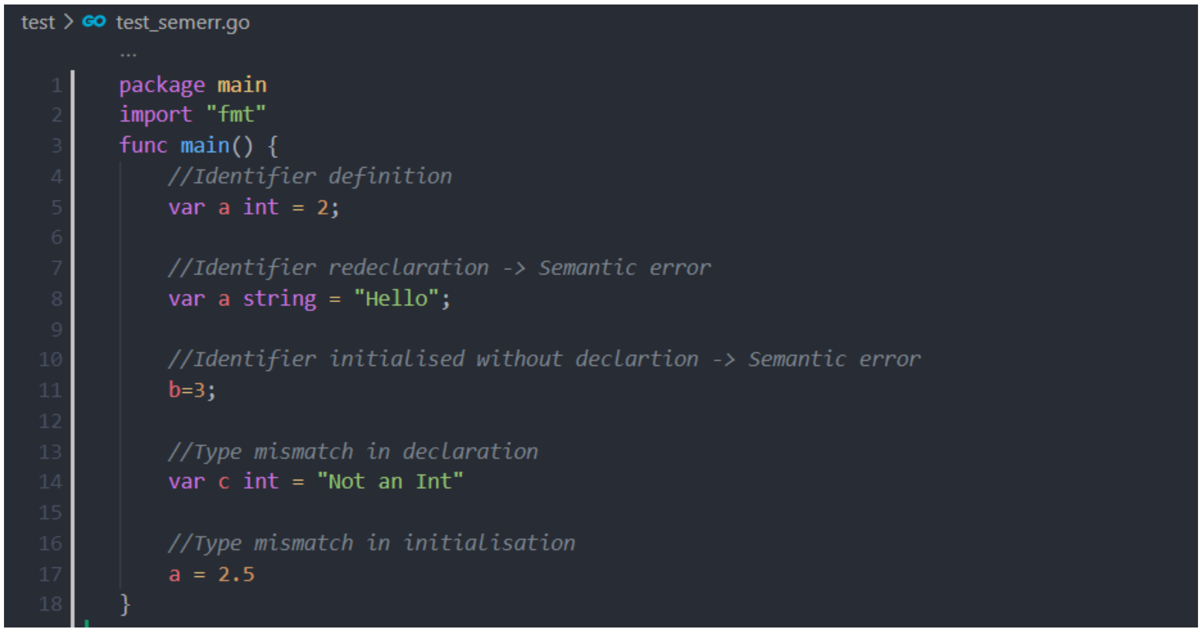
   - ➔ Reporting Syntax Error



```
avinash@MSI:/mnt/d/programs/projects/side/mini-golang-compiler/SymbolTable$ ./gocompiler ../test/test3_wrong.go
Inside main
Read the input file, continue with Lexing and Parsing
Performing Lexical analysis......

 Line:9 | Column: 15 syntax error, unexpected T_IDENTIFIER, expecting T_SEMICOLON

avinash@MSI:/mnt/d/programs/projects/side/mini-golang-compiler/SymbolTable$
```

3. Semantic Errors

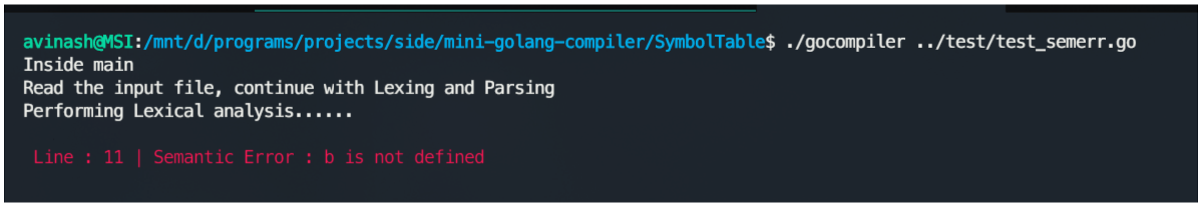
➔ Test Case File :



```
test > test_semerr.go
package main
import "fmt"
func main() {
    //Identifier definition
    var a int = 2;

    //Identifier redeclaration -> Semantic error
    var a string = "Hello";

    //Identifier initialised without declartion -> Semantic error
    b=3;

    //Type mismatch in declaration
    var c int = "Not an Int"

    //Type mismatch in initialisation
    a = 2.5
}
```

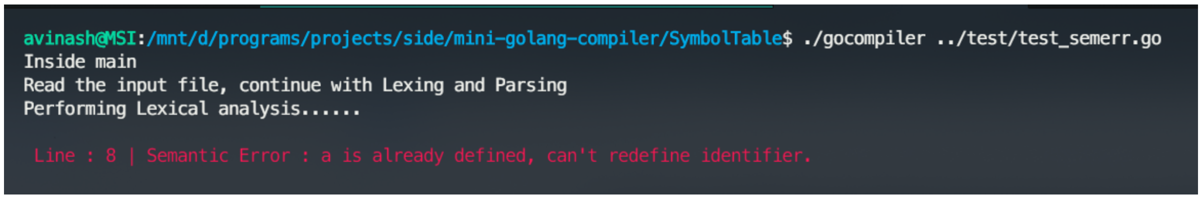
➔ Outputs of Semantic Error Reporting and Handling (Taken one at a time)

◆ Undeclared Identifier



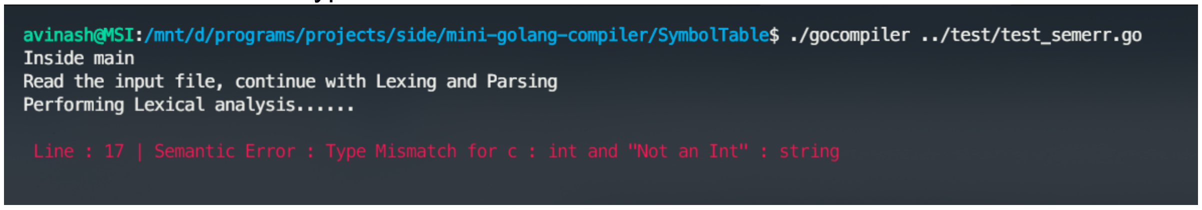
```
avinash@MSI:/mnt/d/programs/projects/side/mini-golang-compiler/SymbolTable$ ./gocompiler ../test/test_semerr.go
Inside main
Read the input file, continue with Lexing and Parsing
Performing Lexical analysis......

 Line : 11 | Semantic Error : b is not defined
```

◆ Redeclaration of Identifier



```
avinash@MSI:/mnt/d/programs/projects/side/mini-golang-compiler/SymbolTable$ ./gocompiler ../test/test_semerr.go
Inside main
Read the input file, continue with Lexing and Parsing
Performing Lexical analysis......

 Line : 8 | Semantic Error : a is already defined, can't redefine identifier.
```

◆ Data Type Mismatch



```
avinash@MSI:/mnt/d/programs/projects/side/mini-golang-compiler/SymbolTable$ ./gocompiler ../test/test_semerr.go
Inside main
Read the input file, continue with Lexing and Parsing
Performing Lexical analysis......

 Line : 17 | Semantic Error : Type Mismatch for c : int and "Not an Int" : string
```

CONCLUSIONS

Hence we have a mini-GoLang compiler that performs the following tasks:

- ➔ For arithmetic, relational and logical operations.
  - ◆ Complete Parsing and symbol tree generation
  - ◆ Computation and updation of Arithmetic and Relational Expressions
  - ◆ Abstract Syntax Tree Generation
  - ◆ Intermediate Code Generation
  - ◆ Code Optimizations
- ➔ For For loops
  - ◆ Complete Parsing and symbol tree generation
  - ◆ Intermediate Code Generation
- ➔ Conditional statements
  - ◆ Complete Parsing and symbol tree generation
- ➔ Basic Error Handling

FURTHER ENHANCEMENTS

There can be the following enhancements in the future as this is only a mini compiler:

- ➔ Implementation of more code constructs
- ➔ Further updation of symbol table like scope etc.
- ➔ Reducing shift/reduce errors and reduce/reduce errors.

At a more ambitious side we would like to predict the best order of optimizations.

REFERENCES

1. https://www.oreilly.com/library/view/lex-yacc/9781565920002/ch01.html
2. https://golang.org/ref/spec#Introduction
3. https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf
4. http://pages.cs.wisc.edu/~fischer/cs701.f07/lectures/Lecture02.pdf
5. https://cs.gmu.edu/~white/CS540/Slides/Semantic/CS540-2-lecture6.pdf