Report on

"C Mini Compiler"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Shashank S Byakod	**PES2201800412**
Aditya Bhimesha	**PES2201800124**
Nanda Kumar N	**<SRN 3>**

Under the guidance of

Dr.N Mehala
Faculty, CSE Department
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

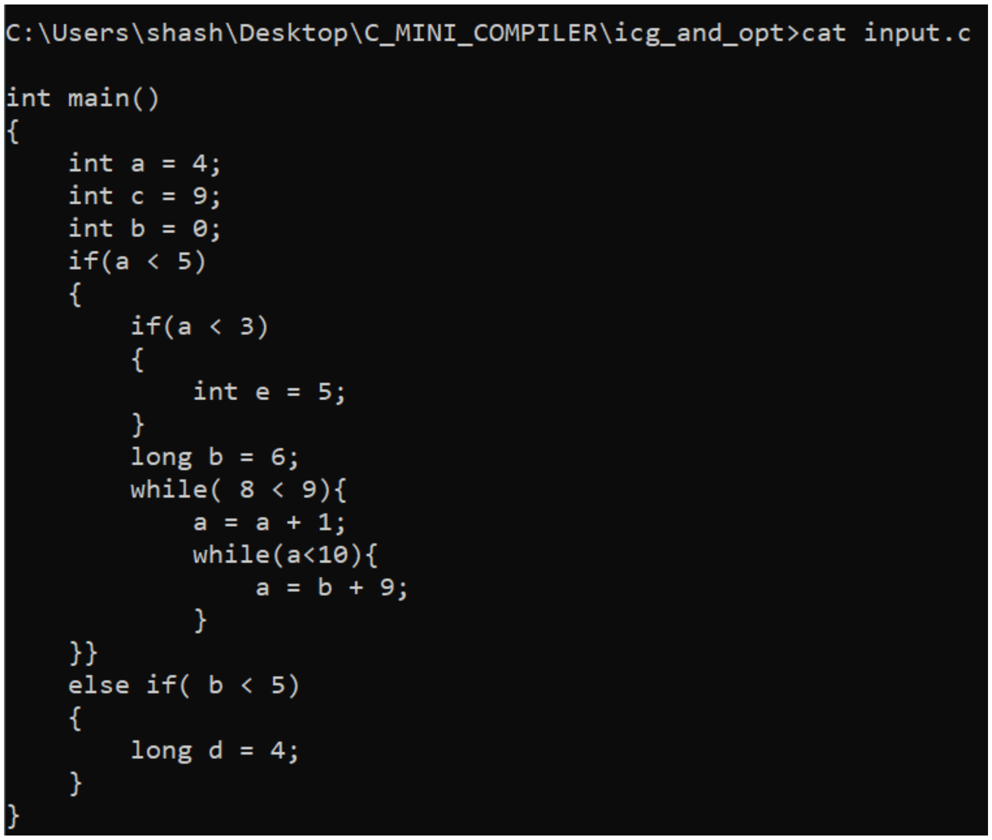
Introduction

The objective of this project is to build a Mini C compiler using Lex and Yacc.
The output of the project is being code optimised intermediate code using python programming language.
It works for constructs which include While and constructs such as conditional statements such as If, Else , Else If.
The various steps included in generating the optimised intermediate code includes :

- Generating symbol table after evaluating the expressions in the input file.
- Generate Abstract Syntax Tree for the code generated.
- Generate 3 address codes which are not optimised and are in the form of quadruples.
- Generate optimised code by performing basic code optimisation techniques.

The various tools used in building this Mini C compiler include LEX which helps in identification of predefined patterns and helped in generating tokens accordingly. YACC was used for parsing the input for semantic meaning and generating abstract syntax trees and intermediate code generation.

Sample Input For the C Mini Compiler:



```
C:\Users\shash\Desktop\C_MINI_COMPILER\icg_and_opt>cat input.c

int main()
{
    int a = 4;
    int c = 9;
    int b = 0;
    if(a < 5)
    {
        if(a < 3)
        {
            int e = 5;
        }
        long b = 6;
        while( 8 < 9){
            a = a + 1;
            while(a<10){
                a = b + 9;
            }
    }}
    else if( b < 5)
    {
        long d = 4;
    }
}
```

Sample Output:

Architecture of Language

The following constructs are handled in our implementation:

- Constructs like ‘**if-else**’ and ‘**while**’ and the required indentation for these loops.
- Nested loops
- Single and multi-line comments.
- Integer and float data types

Corresponding error messages are displayed based on the type of error. Syntax errors are handled using the yyerror() function, while the semantic errors are handled by making a call to a function that searches for a particular identifier in the symbol table. The line number is displayed as part of the error message. Also the fact that a Trie is implemented to handle errors such as ‘whil’ and ‘while’.

Error recovery strategies such as panic mode recovery have been implemented for the lexer. Thus recovering from errors in variable declaration. In case of identifiers, when the name begins with a digit, the compiler neglects the digit and considers the rest as the identifier name. A Trie based implementation auto-suggests the next token for tokens that can be known well at compile time

Literature Survey

1) Full Grammar Specification Link:
https://www.lysator.liu.se/c/ANSI-C-grammar-y.html
Abstract: This is the full C grammar, it contains the CFG for our language which includes while for conditional statements and arithmetic operation.

2) Introduction to Lex Link:
http://dinosaur.compilertools.net/lex/index.html
Abstract: Gives a glimpse of how Lex is used to parse regular expressions of a language and generate the respective output files.

3) Introduction to Yacc Link:
https://www.inf.unibz.it/~artale/Compiler/intro-yacc.pdf
Abstract: Introduces working of Yacc using examples of calculator, roman numbers and parenthesis. Gives details about the input files and output files and discusses how to parse conflicts.

4) Intermediate Code Generation Link:
https://2k8618.blogspot.com/2011/06/intermediate-code-generatorfor.html?m=0
Abstract: This has an example of an intermediate code generator for arithmetic expressions in the form of a yacc program.

Context Free Grammar

```
Start : T_int T_main T_openParenthesis T_closedParanthesis
openflower block_end_flower

block_end_flower : stmt Multiple_stmts
                 | closeflower

block :  openflower block_end_flower
         | stmt
         | T_Semicolon
         ;

Multiple_stmts : stmt Multiple_stmts
               |closeflower
               ;

stmt : expr T_Semicolon
               | if_stmt
               | while_stmt
               | Assignment_stmt T_Semicolon
               | error T_Semicolon
               ;

while_stmt : T_while T_openParenthesis expr T_closedParanthesis
block

if_stmt : T_if T_openParenthesis expr T_closedParanthesis block
elseif_else_empty

elseif_else_empty : T_else T_if T_openParenthesis expr
T_closedParanthesis block elseif_else_empty
                                     | T_else Multiple_stmts_not_if
                                     | T_else openflower block_end_flower
                                     ;
```

```
Multiple_stmts_not_if : stmt_without_if Multiple_stmts
                                        |T_Semicolon
                                        ;

stmt_without_if : expr T_Semicolon
                                        | Assignment_stmt T_Semicolon
                                        | while_stmt
                                        ;

Assignment_stmt:       T_identifier T_AssignmentOperator expr
                                        | T_identifier T_shortHand expr
                                        |                T_type               T_identifier
T_AssignmentOperator expr_without_constants
                                        |                T_type               T_identifier
T_AssignmentOperator T_stringLiteral
                                        |                T_type               T_identifier
T_AssignmentOperator T_numericConstants
                                        | T_int T_identifier T_AssignmentOperator
expr_without_constants
                                        | T_int T_identifier T_AssignmentOperator
T_numericConstants
                                ;


expr_or_empty_with_semicolon_and_assignment:          expr_or_empty
T_Semicolon
        | Assignment_stmt T_Semicolon

expr_or_empty_with_assignment_and_closed_parent:    expr_or_empty
T_closedParanthesis
        | Assignment_stmt T_closedParanthesis

expr_without_constants:  T_identifier
                | expr T_plus expr
                | expr T_minus expr
                | expr T_divide expr
                | expr T_multiply expr
                | expr T_mod expr
                | expr T_LogicalAnd expr
                | expr T_LogicalOr expr
                | expr T_less expr
                | expr T_less_equal expr
                | expr T_greater expr
                | expr T_greater_equal expr
                | expr T_equal_equal expr
                | expr T_not_equal expr
                ;


expr:  T_numericConstants
                | T_stringLiteral
                | T_identifier
                | expr T_plus expr
                | expr T_minus expr
                | expr T_divide expr
                | expr T_multiply expr
                | expr T_mod expr
                | expr T_LogicalAnd expr
                | expr T_LogicalOr expr
                | expr T_less expr
                | expr T_less_equal expr
                | expr T_greater expr
                | expr T_greater_equal expr
                | expr T_equal_equal expr
                | expr T_not_equal expr
                ;

expr_or_empty: expr
                |
                ;
```

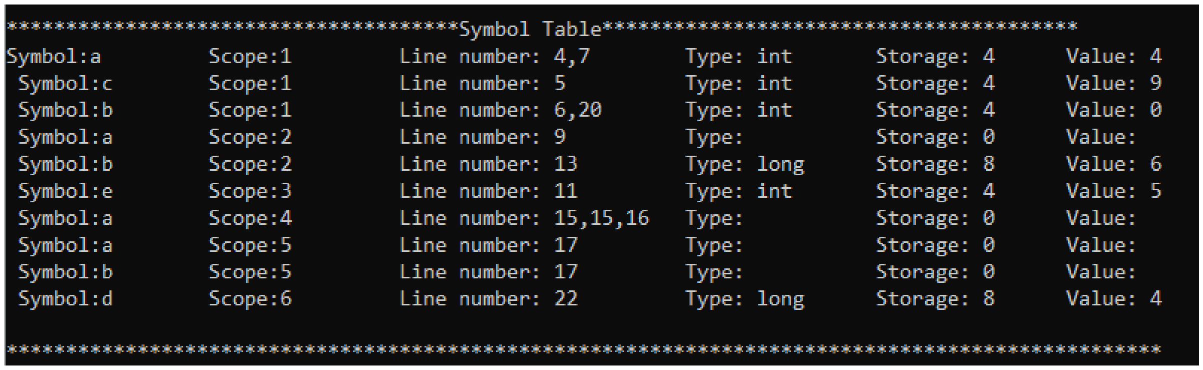
openflower: T_openFlowerBracket {};
closeflower: T_closedFlowerBracket {};

Design Strategy / IMPLEMENTATION DETAILS

1) SYMBOL TABLE CREATION

The following snapshot and data structure shows the structure declaration for symbol table:

```
struct var
{
    char var_name[20];
    char Line_t[100];
    char type[100];
    char value[100];
    int scope;
    int storage;
};
struct scope
{
    struct var arr[20];
    int up;
};
```

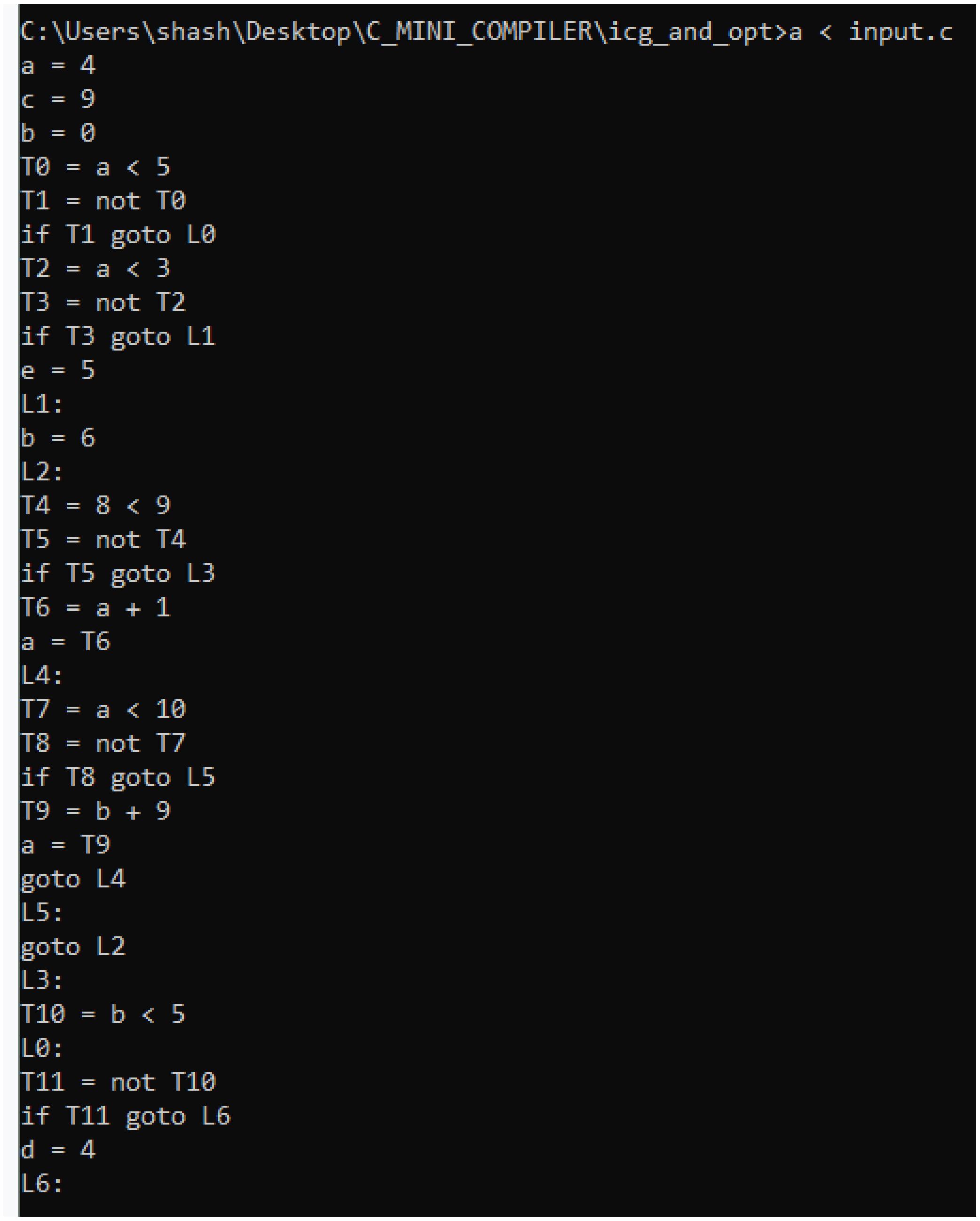


```
***************************************Symbol Table*****************************************
Symbol:a         Scope:1         Line number: 4,7        Type: int       Storage: 4      Value: 4
 Symbol:c        Scope:1         Line number: 5          Type: int       Storage: 4      Value: 9
 Symbol:b        Scope:1         Line number: 6,20       Type: int       Storage: 4      Value: 0
 Symbol:a        Scope:2         Line number: 9          Type:           Storage: 0      Value:
 Symbol:b        Scope:2         Line number: 13         Type: long      Storage: 8      Value: 6
 Symbol:e        Scope:3         Line number: 11         Type: int       Storage: 4      Value: 5
 Symbol:a        Scope:4         Line number: 15,15,16   Type:           Storage: 0      Value:
 Symbol:a        Scope:5         Line number: 17         Type:           Storage: 0      Value:
 Symbol:b        Scope:5         Line number: 17         Type:           Storage: 0      Value:
 Symbol:d        Scope:6         Line number: 22         Type: long      Storage: 8      Value: 4

*********************************************************************************************
```

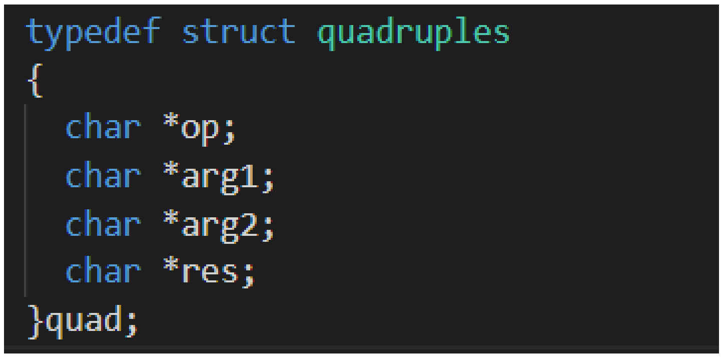
2) INTERMEDIATE CODE GENERATION (QUADRUPLES)

Intermediate Code for sample input

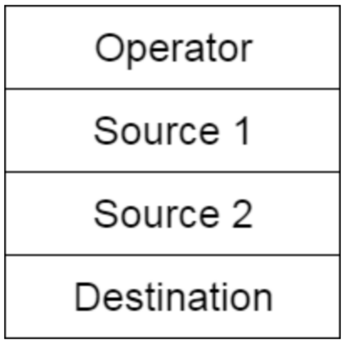


```
C:\Users\shash\Desktop\C_MINI_COMPILER\icg_and_opt>a < input.c
a = 4
c = 9
b = 0
T0 = a < 5
T1 = not T0
if T1 goto L0
T2 = a < 3
T3 = not T2
if T3 goto L1
e = 5
L1:
b = 6
L2:
T4 = 8 < 9
T5 = not T4
if T5 goto L3
T6 = a + 1
a = T6
L4:
T7 = a < 10
T8 = not T7
if T8 goto L5
T9 = b + 9
a = T9
goto L4
L5:
goto L2
L3:
T10 = b < 5
L0:
T11 = not T10
if T11 goto L6
d = 4
L6:
```

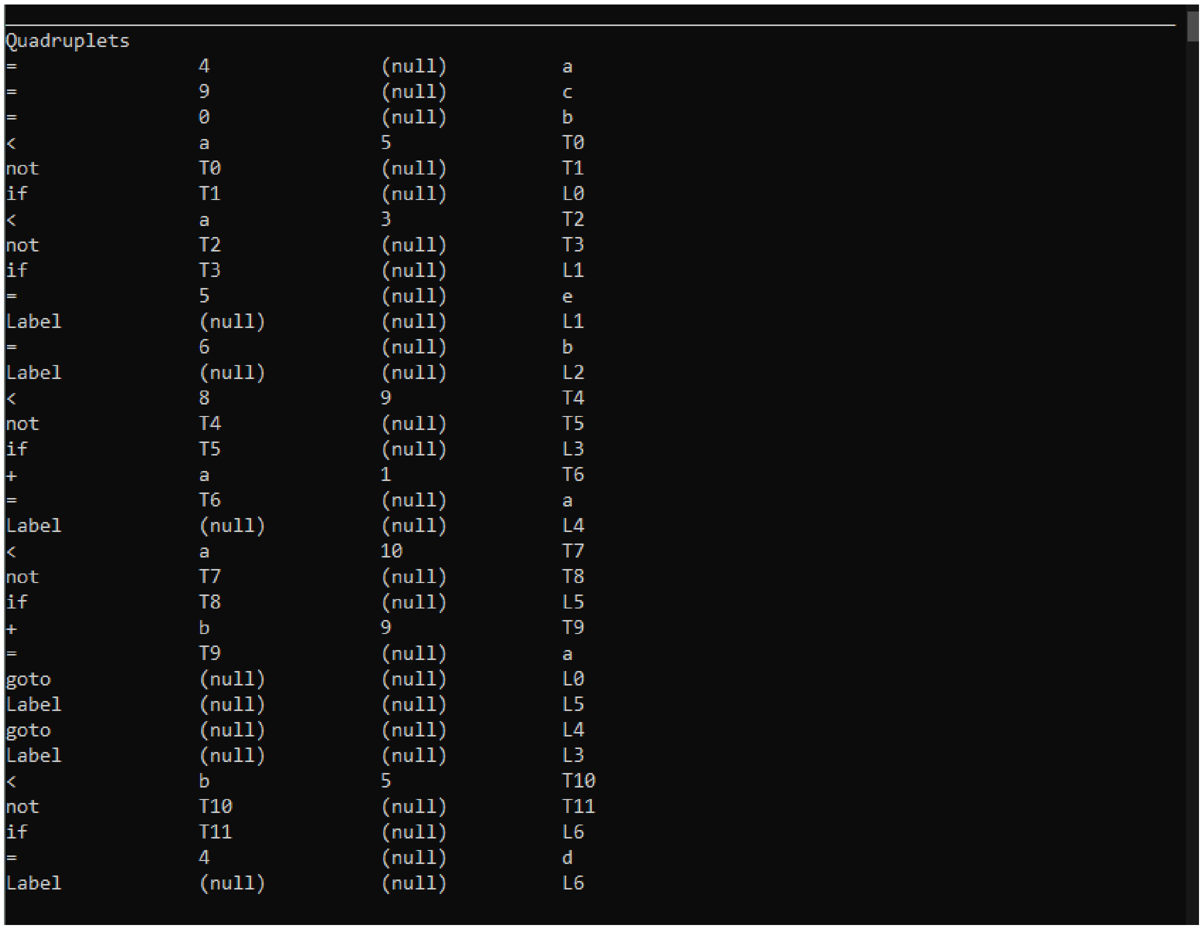
Data structure for Quadruples



```c
typedef struct quadruples
{
  char *op;
  char *arg1;
  char *arg2;
  char *res;
}quad;
```



Sample Output of Quadruples.

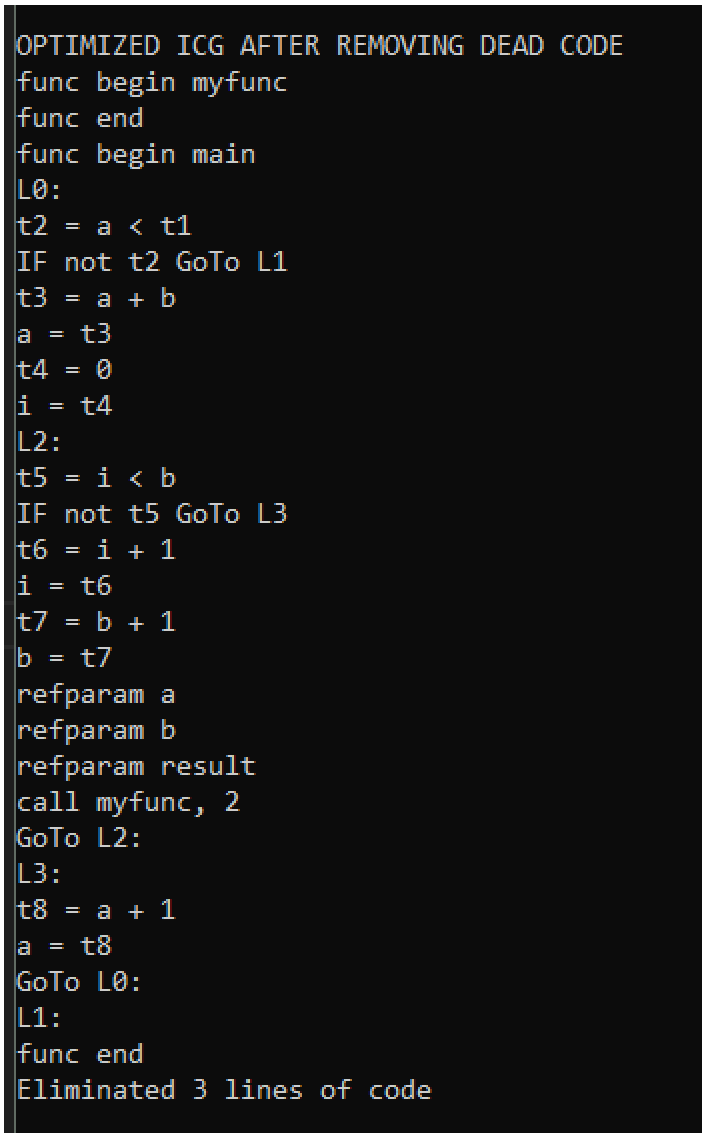


```
Quadruplets
=                4               (null)          a
=                9               (null)          c
=                0               (null)          b
<                a               5               T0
not              T0              (null)          T1
if               T1              (null)          L0
<                a               3               T2
not              T2              (null)          T3
if               T3              (null)          L1
=                5               (null)          e
Label            (null)          (null)          L1
=                6               (null)          b
Label            (null)          (null)          L2
<                8               9               T4
not              T4              (null)          T5
if               T5              (null)          L3
+                a               1               T6
=                T6              (null)          a
Label            (null)          (null)          L4
<                a               10              T7
not              T7              (null)          T8
if               T8              (null)          L5
+                b               9               T9
=                T9              (null)          a
goto             (null)          (null)          L0
Label            (null)          (null)          L5
goto             (null)          (null)          L4
Label            (null)          (null)          L3
<                b               5               T10
not              T10             (null)          T11
if               T11             (null)          L6
=                4               (null)          d
Label            (null)          (null)          L6
```

The quadruples have four fields to implement the three address code. The field of quadruples contains the name of the operator, the first source operand, the second source operand and the result respectively.

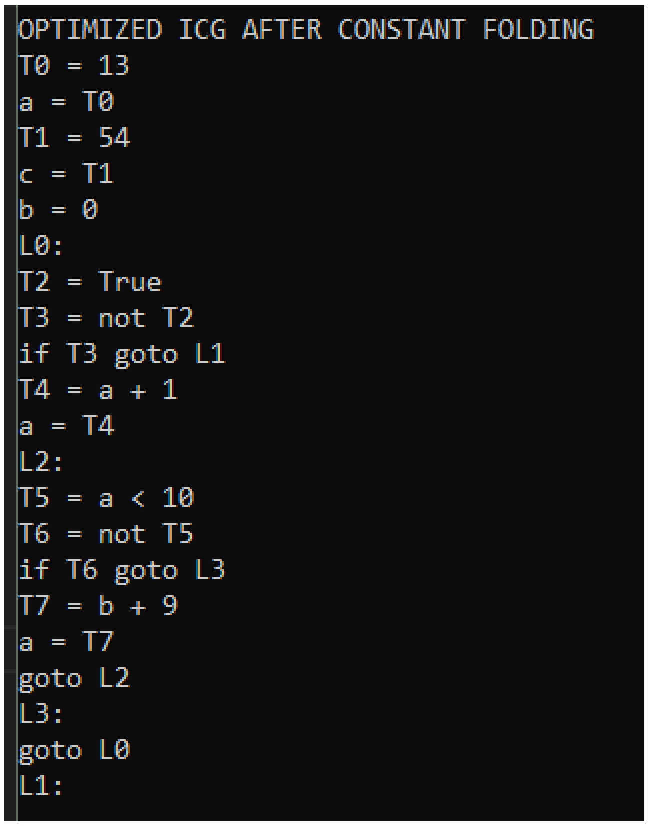
3) CODE OPTIMISATION:

- **Remove Dead Code**
  Temporaries and unreachable code which are not used in runtime are all removed to decrease the load on the used resources.



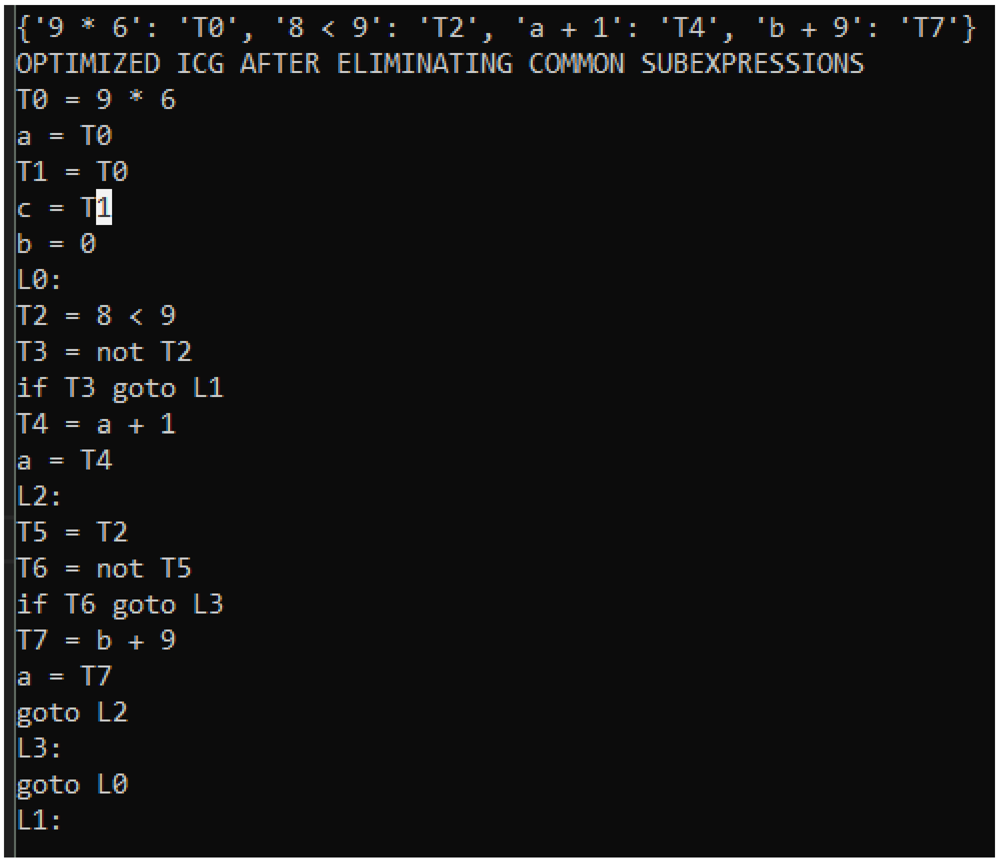
```
OPTIMIZED ICG AFTER REMOVING DEAD CODE
func begin myfunc
func end
func begin main
L0:
t2 = a < t1
IF not t2 GoTo L1
t3 = a + b
a = t3
t4 = 0
i = t4
L2:
t5 = i < b
IF not t5 GoTo L3
t6 = i + 1
i = t6
t7 = b + 1
b = t7
refparam a
refparam b
refparam result
call myfunc, 2
GoTo L2:
L3:
t8 = a + 1
a = t8
GoTo L0:
L1:
func end
Eliminated 3 lines of code
```

- **Constant Folding**
  Variables whose values can be computed at compile time are computed by the compiler, thus again reducing the usage of resources.



```
OPTIMIZED ICG AFTER CONSTANT FOLDING
T0 = 13
a = T0
T1 = 54
c = T1
b = 0
L0:
T2 = True
T3 = not T2
if T3 goto L1
T4 = a + 1
a = T4
L2:
T5 = a < 10
T6 = not T5
if T6 goto L3
T7 = b + 9
a = T7
goto L2
L3:
goto L0
L1:
```

- **Elimination of Common Subexpression**
  We find expressions that evaluate to the same value and substitute a temporary variable in place of it, to avoid unnecessary computation by the resources it is running on.



```
{'9 * 6': 'T0', '8 < 9': 'T2', 'a + 1': 'T4', 'b + 9': 'T7'}
OPTIMIZED ICG AFTER ELIMINATING COMMON SUBEXPRESSIONS
T0 = 9 * 6
a = T0
T1 = T0
c = T1
b = 0
L0:
T2 = 8 < 9
T3 = not T2
if T3 goto L1
T4 = a + 1
a = T4
L2:
T5 = T2
T6 = not T5
if T6 goto L3
T7 = b + 9
a = T7
goto L2
L3:
goto L0
L1:
```

HOW TO RUN THE CODE

1. **Download Flex on your system**

2. **For Compiling Yacc File**
   Use the Command ***yacc -d Yacc.y***

3. **For Compiling Lex File**
   Use the Command ***lex lex.l***

4. **Then Construct the Compiler using**
   gcc lex.yy.c Yacc.tab.c -w

5. **Run executable file**
   a < input.c (windows)
   or
   ./a.out < input.c (Unix Based machine)

6. **Generate Optimized ICG Using Python3**
   python3 optimize.py input.txt (ICG code generated by compiler)

RESULTS AND SHORTCOMINGS

The mini-compiler built in this project works perfectly for the ‘if-else’ and ‘while’ constructs of C language. Our compiler can be executed in different phases by building and running the code separated in the various folders. First, the tokens are displayed, followed by a ‘PARSE SUCCESSFUL’ message. The abstract syntax tree is printed next. Next, the symbol table along with the intermediate code is printed without optimisations. Finally, the symbol table and the intermediate code after optimisations is displayed after the quadruples table

This mini-compiler has the following shortcomings:

• User defined functions are not handled.

• Importing libraries and calling library functions is not taken care of.

• Data Types other than integer and float, example strings, lists, tuples, dictionaries, etc have not been considered.

• Constructs other than ‘while’ and ‘if-else’ have not been added in the compiler program.

FURTHER ENHANCEMENTS

This mini-compiler can be enhanced to a complete compiler for the C language by making a few improvements. User defined functions can be handled and the functionality of importing libraries and calling library functions can be taken care of. Data Types other than Integers, example strings, lists, tuples, dictionaries, etc can be included and constructs other than **'while'** and '**if-else**', like **'for'** can be added in the compiler program. The output can be made to look more enhanced and beautiful.

The overall efficiency and speed of the program can be improved by using some other data structures, functions or approaches.

We can also perform more machine dependent code optimisation to improve the compile time of large programs on specific systems.

References:

1) Lex and Yacc
http://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf

2) Introduction about Flex and Bison
http://dinosaur.compilertools.net/

3) Full Grammar Specification
https://www.lysator.liu.se/c/ANSI-C-grammar-y.html

4) Introduction to Yacc
https://www.inf.unibz.it/~artale/Compiler/intro-yacc.pdf

5) Intermediate Code Generation
https://2k8618.blogspot.com/2011/06/intermediate-code-generatorfor.html?m=0

6) Link to our github repository

THANK YOU