*Report on*

# "Mini C Compiler for 'If-Else-For-While' Constructs"

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

***Compiler Design Laboratory***

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

***Submitted by:***

| | |
|---|---|
| **Srujan V** | **PES1201700132** |
| **Shreyas Mavanoor** | **PES1201700837** |
| **Vihan S Murthy** | **PES1201701603** |

*Under the guidance of*

**Suhas G K**
Assistant Professor
PES University, Bengaluru

**January – May 2020**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**
FACULTY OF ENGINEERING
**PES UNIVERSITY**
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

# TABLE OF CONTENTS

# Chapter 1
**Introduction:**

- This project being a Mini Compiler for the C programming language, focuses on generating an intermediate code for the language for specific constructs.It works for constructs such as conditional statements, loops and the ternary operator.

- The main functionality of the project is to generate an optimized intermediate code for the given C source code and generating Target code. Which is done using :

  i) Generate symbol table after performing expression evaluation

  ii) Generate Abstract Syntax Tree for the code

  iii) Generate 3addresscodefollowedby corresponding quadruples

  iv) Perform Code Optimization

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

- C is used to optimize the intermediate code generated by the parser.

# Chapter 2
**Architecture:**

- Used Lex to create the scanner for our language, Yacc to implement grammar rules to the token generated in the scanner phase.

- C constructs implemented:

  1. If

  2. If-else

  3. Ternary operator 4. While loop

  5. For-loop

- Arithmetic expressions with +, -, *, /, ++, -- are handled.
- Boolean expressions with >, <, >=, <=, == are handled.

- Ignore comments and white-spaces.
- Types: int, float, char, void.
- Error handling reports undeclared variables.
- Error handling also reports syntax errors with line numbers and Error handling related to scope and declaration.
- Code Optimisation Technique: Common Subexpression Elimination and Dead Code Elimination

**Phase 1: (a)Lexical Analysis**

● LEX tool was used to create a scanner for C language

● The scanner transforms the source file from a stream of bits and bytes into a series of meaningful tokens containing information that will be used by the later stages of the compiler.

● The scanner also scans for the comments (single-line and multi- line comments) and writes the source file without comments onto an output file which is used in the further stages.

● All tokens included are of the form <token,token-name>.Eg: <5,identifier>,<for ,keyword>
● A global variable 'yylavl' is used to record the value of each lexeme scanned. 'yytext' is the lex variable that stores the matched string.
● Skipping over white spaces and recognizing all keywords, operators, variables and constants are handled in this phase.
● Scanning error is reported when the input string does not match any rule in the lex file.
● The rules are regular expressions which have corresponding actions that execute on a match with the source input.

**Phase 1: (b)Syntax Analysis**

● Syntax analysis is only responsible for verifying that the sequence of tokens forms a valid sentence given the definition of your Programming Language Grammar.
● The design implementation supports:

**1.** Variable declarations and initialization
**2.** Variables of type int,float and char
**3.** Arithmetic and boolean expressions
**4.** Postfix and prefix expressions
**5.** Constructs **-if-else, ternary, while loop and for loop**

- Yacc tool is used for parsing. It reports shift-reduce and reduce- reduce conflicts on parsing an ambiguous grammar.

# Chapter 3

## Literature Survey and References:

- Lex and Yacc Usage and terminology:
  A website that explained compilers or interpreters for a programming language and its decomposed parts. It also helped us to understand the basic concepts required in building each phase of the compiler from Lexical and Syntax Phase to generating the target assembly language code.
  http://dinosaur.compilertools.net

- Lex and Yacc Tutorial:
  Lex and yacc tutorial by Tom Neimann that helped us to understand how to use the lex and yacc tools to build each phase of the compiler from Lexical Phase to the final phase.
  https://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf

- Reference for C Grammar:
  The most important part that has to be done before starting to build the different phases of the compiler is the context free grammar for the language. It is this grammar that has rules and productions for the language and on the basis of this context free grammar we start building the compiler phase by phase.
  https://www.lysator.liu.se/c/ANSI-C-grammar-y.html

# Chapter 4

## Context Free Grammar:

**The following is the CFG used and storing symbol table in structures -**

S->#include<Header>S|Type main(){Statements;}

Header->stdio.h|stdlib.h|string.h

Type->int|char|double|void|longint

Statements->(Var_dec|Cond_stat|Loop_stat|Operation|Rel_Operation|Inc|Print_stat)Statements | λ

Var_dec->Type Variable

Variable->Var|Var,Variable

Var->id|Assign_exp

Assign_exp->id=const;

Operation->A oper A

A->A+B|A-B|B

B->A*B|A/B|c

C->id|const

Oper->+|-|*|/

Cond_stat->if(Cond){Statements;}else{Statements;}

Rel_Operation->A Rel A

Rel-> >=|<=|>|<|!=|==

Loop_stat->while(Cond) ob statements cb ;|for(cond) ob statements; cb| λ

Cond->Rel_Operation|λ

Inc->Inc_dec|λ

Inc_dec->c++|c--|--c|++c

Print_stat -> printf("String");Print_stat | printf("String %Dt String",Var);Print_stat|λ

String->[A-Za-z0-9]*

Dt-> d|f|c|s|ld

# Chapter 5

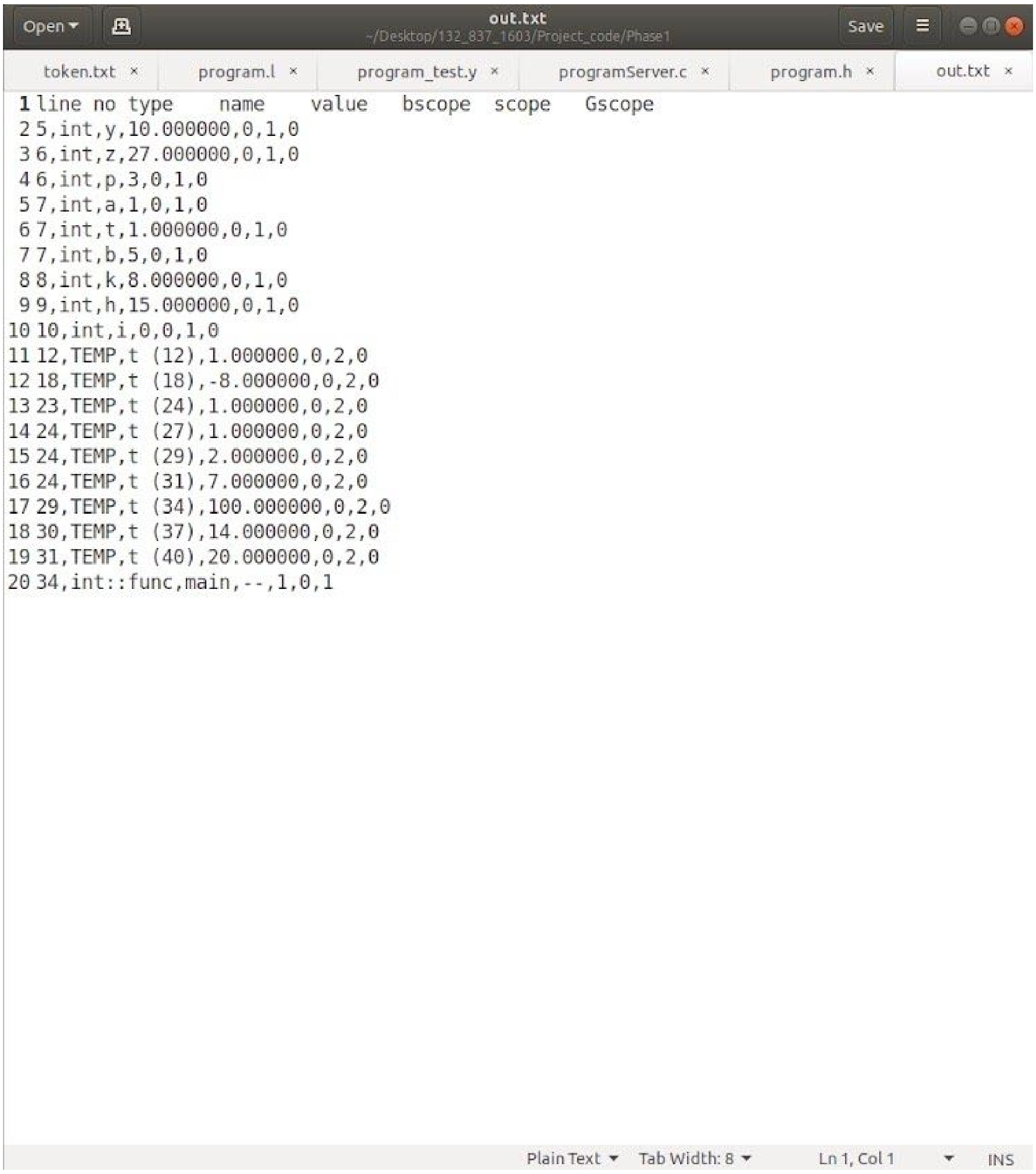
## Design Strategy:

- **Symbol Table:** This is a structure maintained to keep track of the variables, constants, operators and the keywords in the input. The parameters of the structure are the name of the token, the line number of occurence, the category of the token (constant, variable,keyword,operator),the value that it holds the datatype.

```
struct table
        {

             char name[20];
              char type[20];
              char value[20]; //for function name
              int lineNO;

             //for scope
              int scope;
              int Bscope;
              int Gscope;


             }arr[100];

       struct temporary{
            char type[20];
            char name[20];

           char value[20];
           int Cflag;

           int dflag;
         }Tarr[20];

bool funcFlag;

int funcCount = 0;

       struct dtemp{
            char name[20];

       }dtemparr[100];
```

- As each line is parsed, the actions associated with the grammar rules are executed. Symbol tables functions such as lookup, search_id, update and get_val are called appropriately with each production rule.

- Expressions are evaluated and the values of the used variables are updated accordingly.
- At the end of the parsing, the updated symbol table is displayed.



```
line no type    name    value   bscope  scope   Gscope
5,int,y,10.000000,0,1,0
6,int,z,27.000000,0,1,0
6,int,p,3,0,1,0
7,int,a,1,0,1,0
7,int,t,1.000000,0,1,0
7,int,b,5,0,1,0
8,int,k,8.000000,0,1,0
9,int,h,15.000000,0,1,0
10,int,i,0,0,1,0
12,TEMP,t (12),1.000000,0,2,0
18,TEMP,t (18),-8.000000,0,2,0
23,TEMP,t (24),1.000000,0,2,0
24,TEMP,t (27),1.000000,0,2,0
24,TEMP,t (29),2.000000,0,2,0
24,TEMP,t (31),7.000000,0,2,0
29,TEMP,t (34),100.000000,0,2,0
30,TEMP,t (37),14.000000,0,2,0
31,TEMP,t (40),20.000000,0,2,0
34,int::func,main,--,1,0,1
```

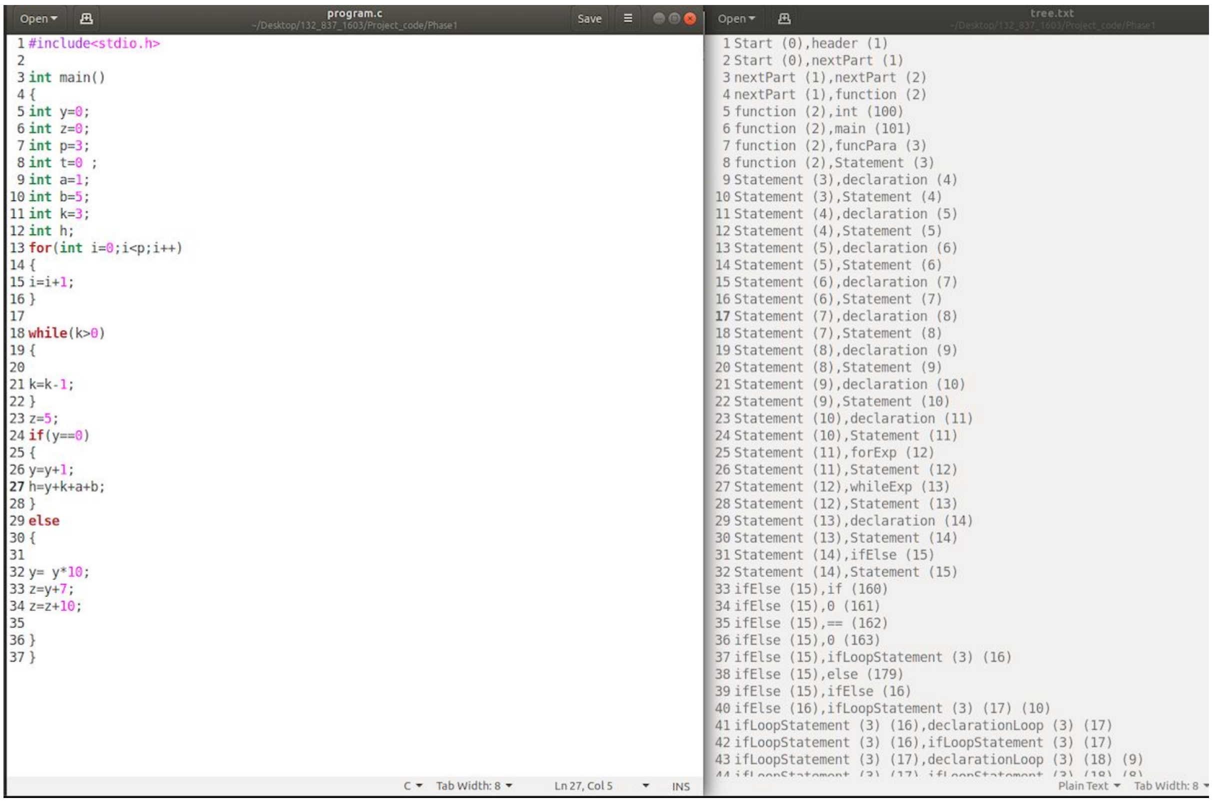
- **Abstract Syntax Tree:** A tree structure representing the syntactical flow of the code is generated in this phase. For expressions associativity is indicated using the %left and %right fields. Precedence of operations - last rule gets higher precedence and hence it is:
  %left lt gt
  %left add sub

%left mul div

To build the tree, a structure is maintained which has pointers to its children and a container for its data value.

When every new token is encountered during parsing, the build Tree function takes in the value of the token , creates a node of the tree and attaches it to its parent(head of the reduced production)

*Sample input* *sample output*


Open
program.c
~/Desktop/132_837_1603/Project_code/Phase1
Save
C
Tab Width: 8
Ln 27, Col 5
INS
Open
tree.txt
~/Desktop/132_837_1603/Project_code/Phase1
Plain Text
Tab Width: 8

```c
#include<stdio.h>

int main()
{
int y=0;
int z=0;
int p=3;
int t=0 ;
int a=1;
int b=5;
int k=3;
int h;
for(int i=0;i<p;i++)
{
i=i+1;
}

while(k>0)
{

k=k-1;
}
z=5;
if(y==0)
{
y=y+1;
h=y+k+a+b;
}
else
{

y= y*10;
z=y+7;
z=z+10;

}
}
```

```
Start (0),header (1)
Start (0),nextPart (1)
nextPart (1),nextPart (2)
nextPart (1),function (2)
function (2),int (100)
function (2),main (101)
function (2),funcPara (3)
function (2),Statement (3)
Statement (3),declaration (4)
Statement (3),Statement (4)
Statement (4),declaration (5)
Statement (4),Statement (5)
Statement (5),declaration (6)
Statement (5),Statement (6)
Statement (6),declaration (7)
Statement (6),Statement (7)
Statement (7),declaration (8)
Statement (7),Statement (8)
Statement (8),declaration (9)
Statement (8),Statement (9)
Statement (9),declaration (10)
Statement (9),Statement (10)
Statement (10),declaration (11)
Statement (10),Statement (11)
Statement (11),forExp (12)
Statement (11),Statement (12)
Statement (12),whileExp (13)
Statement (12),Statement (13)
Statement (13),declaration (14)
Statement (13),Statement (14)
Statement (14),ifElse (15)
Statement (14),Statement (15)
ifElse (15),if (160)
ifElse (15),0 (161)
ifElse (15),== (162)
ifElse (15),0 (163)
ifElse (15),ifLoopStatement (3) (16)
ifElse (15),else (179)
ifElse (15),ifElse (16)
ifElse (16),ifLoopStatement (3) (17) (10)
ifLoopStatement (3) (16),declarationLoop (3) (17)
ifLoopStatement (3) (16),ifLoopStatement (3) (17)
ifLoopStatement (3) (17),declarationLoop (3) (18) (9)
```

- **Intermediate Code Generation:**

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

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

Example – The three address code for the expression a + b * c + d :

T 1 = b* c

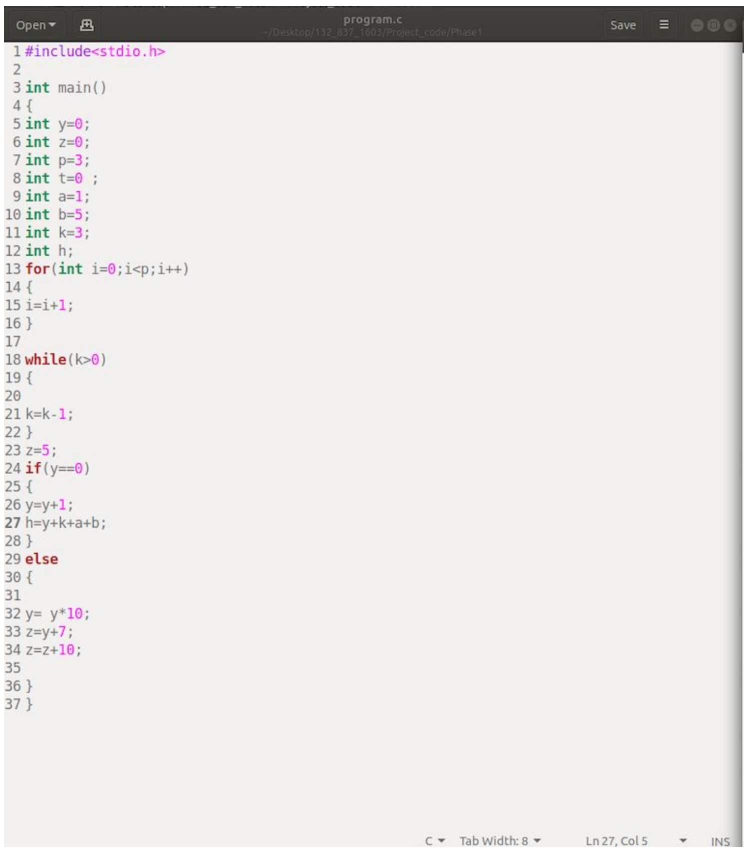
T 2 = a+ T 1

T 3 = T2 + d

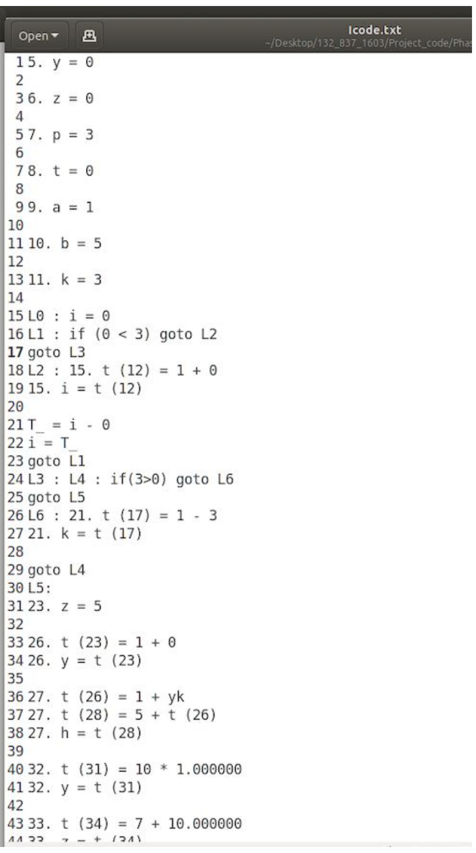
T1, T2, T3 are temporary variables.

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

*Input* *Output*



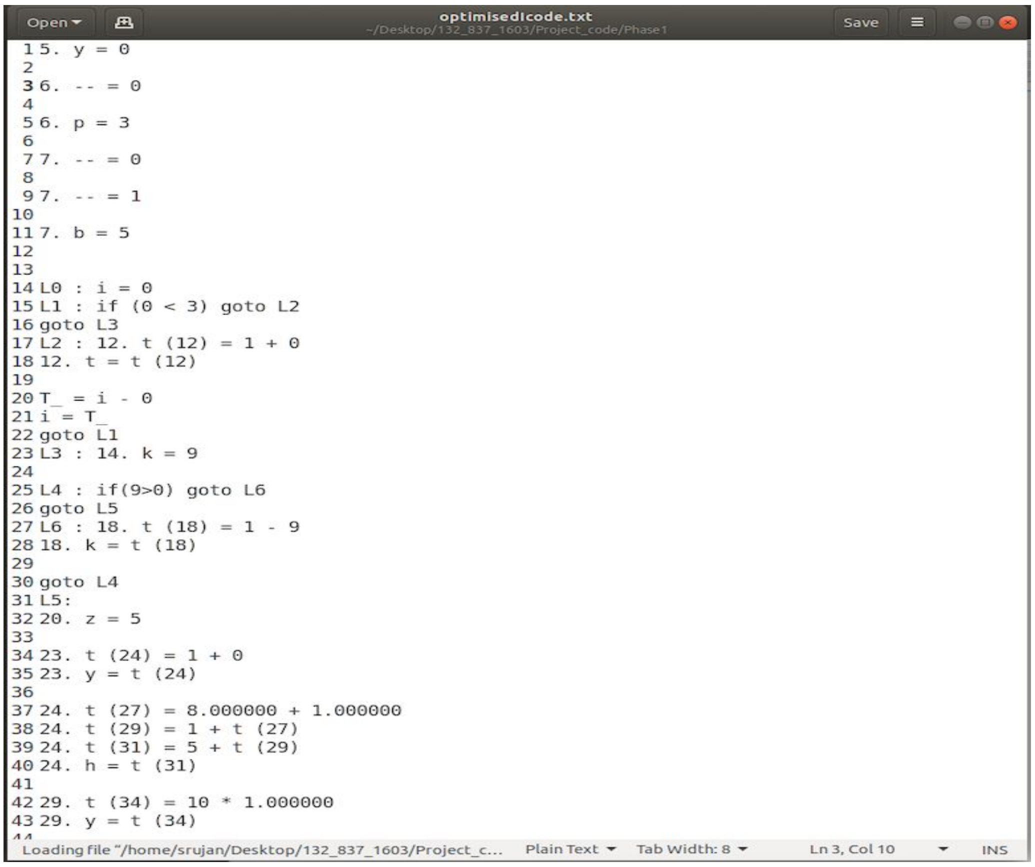
```
#include<stdio.h>

int main()
{
int y=0;
int z=0;
int p=3;
int t=0 ;
int a=1;
int b=5;
int k=3;
int h;
for(int i=0;i<p;i++)
{
i=i+1;
}

while(k>0)
{

k=k-1;
}
z=5;
if(y==0)
{
y=y+1;
h=y+k+a+b;
}
else
{

y= y*10;
z=y+7;
z=z+10;

}
}
```



```
5. y = 0

6. z = 0

7. p = 3

8. t = 0

9. a = 1

10. b = 5

11. k = 3

L0 : i = 0
L1 : if (0 < 3) goto L2
goto L3
L2 : 15. t (12) = 1 + 0
15. i = t (12)

T_ = i - 0
i = T_
goto L1
L3 : L4 : if(3>0) goto L6
goto L5
L6 : 21. t (17) = 1 - 3
21. k = t (17)

goto L4
L5:
23. z = 5

26. t (23) = 1 + 0
26. y = t (23)

27. t (26) = 1 + yk
27. t (28) = 5 + t (26)
27. h = t (28)

32. t (31) = 10 * 1.000000
32. y = t (31)

33. t (34) = 7 + 10.000000
33. z = t (34)
```

- **Code Optimization:**

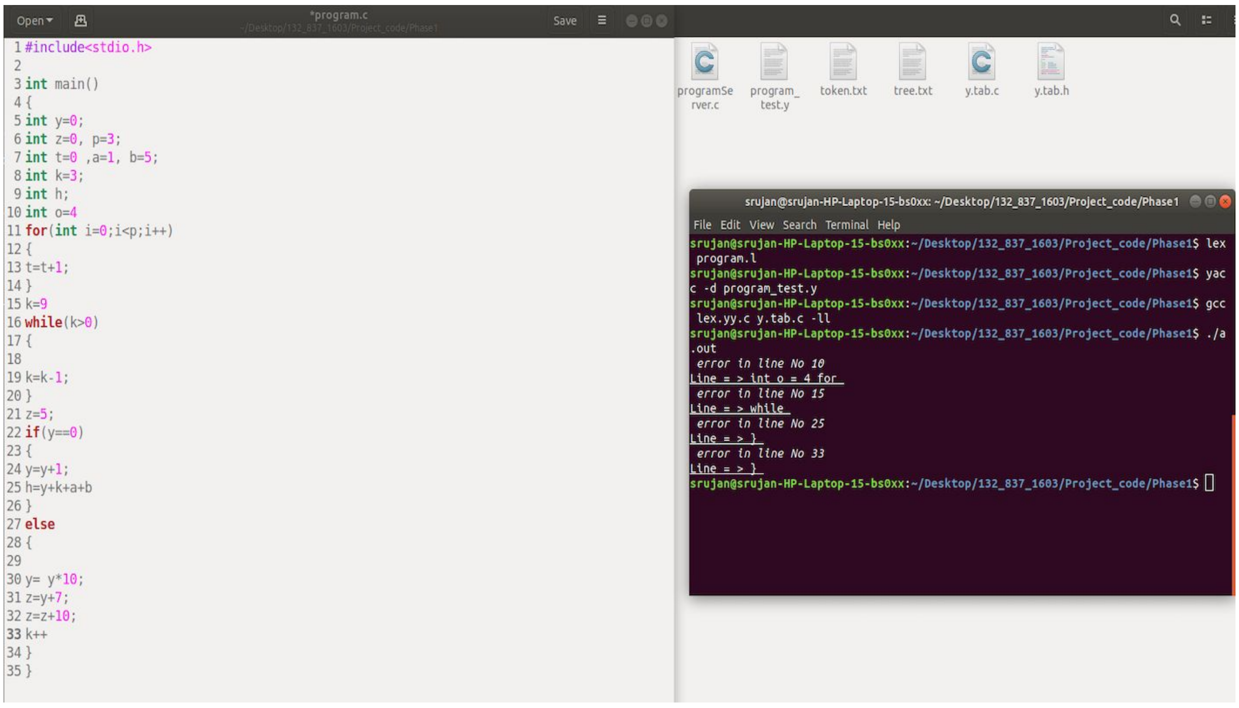
The technique used in code optimisation is common sub-expression elimination which means when all the variables declared multiple times, it considers only the recently declared variable value and avoids all other declared variables which were declared earlier.


Open
optimisedIcode.txt
~/Desktop/132_837_1603/Project_code/Phase1
Save
Loading file "/home/srujan/Desktop/132_837_1603/Project_c...
Plain Text
Tab Width: 8
Ln 3, Col 10
INS

```
5. y = 0

6. -- = 0

6. p = 3

7. -- = 0

7. -- = 1

7. b = 5


L0 : i = 0
L1 : if (0 < 3) goto L2
goto L3
L2 : 12. t (12) = 1 + 0
12. t = t (12)

T_ = i - 0
i = T_
goto L1
L3 : 14. k = 9

L4 : if(9>0) goto L6
goto L5
L6 : 18. t (18) = 1 - 9
18. k = t (18)

goto L4
L5:
20. z = 5

23. t (24) = 1 + 0
23. y = t (24)

24. t (27) = 8.000000 + 1.000000
24. t (29) = 1 + t (27)
24. t (31) = 5 + t (29)
24. h = t (31)

29. t (34) = 10 * 1.000000
29. y = t (34)
```

- **Error Handling:** In case of syntax error, the compilation is halted, and an error message along with the line number where error occurred is displayed. Semantic errors multiple declaration of the same variable, invalid assignment.

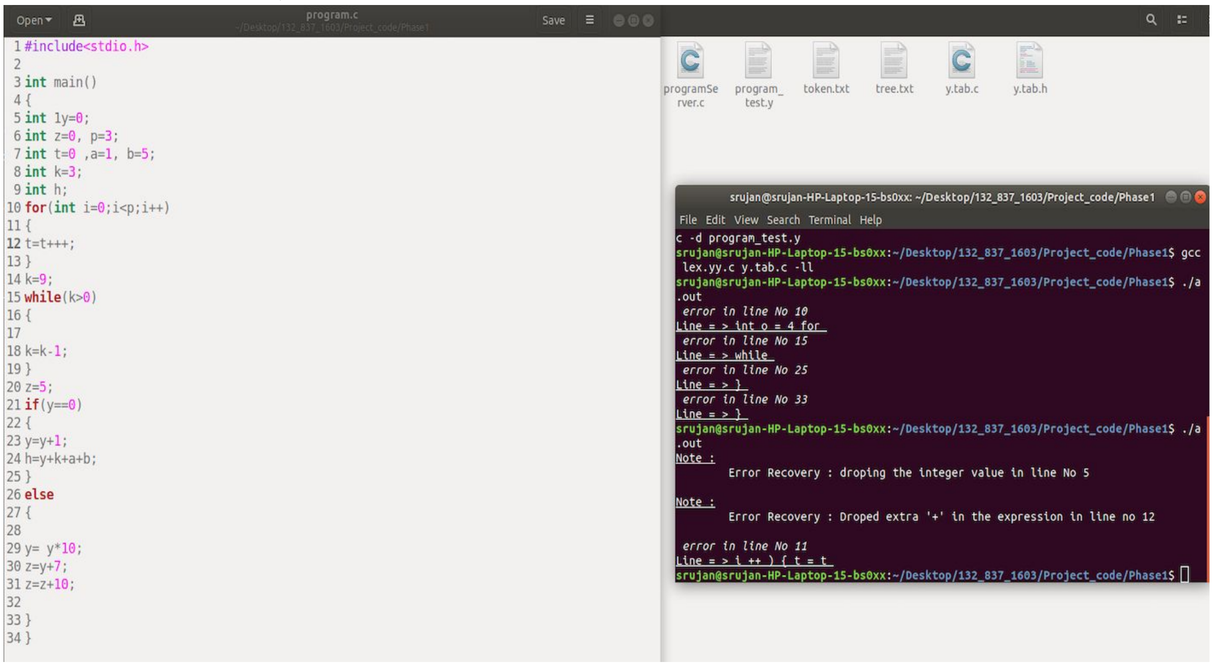
  *Error Handling:*


Open
*program.c
~/Desktop/132_837_1603/Project_code/Phase1
Save
programServer.c
program_test.y
token.txt
tree.txt
y.tab.c
y.tab.h
srujan@srujan-HP-Laptop-15-bs0xx: ~/Desktop/132_837_1603/Project_code/Phase1
File Edit View Search Terminal Help

```
1 #include<stdio.h>
2
3 int main()
4 {
5 int y=0;
6 int z=0, p=3;
7 int t=0 ,a=1, b=5;
8 int k=3;
9 int h;
10 int o=4
11 for(int i=0;i<p;i++)
12 {
13 t=t+1;
14 }
15 k=9
16 while(k>0)
17 {
18
19 k=k-1;
20 }
21 z=5;
22 if(y==0)
23 {
24 y=y+1;
25 h=y+k+a+b
26 }
27 else
28 {
29
30 y= y*10;
31 z=y+7;
32 z=z+10;
33 k++
34 }
35 }
```

```
srujan@srujan-HP-Laptop-15-bs0xx:~/Desktop/132_837_1603/Project_code/Phase1$ lex
 program.l
srujan@srujan-HP-Laptop-15-bs0xx:~/Desktop/132_837_1603/Project_code/Phase1$ yac
c -d program_test.y
srujan@srujan-HP-Laptop-15-bs0xx:~/Desktop/132_837_1603/Project_code/Phase1$ gcc
 lex.yy.c y.tab.c -ll
srujan@srujan-HP-Laptop-15-bs0xx:~/Desktop/132_837_1603/Project_code/Phase1$ ./a
.out
 error in line No 10
Line = > int o = 4 for
 error in line No 15
Line = > while
 error in line No 25
Line = > }
 error in line No 33
Line = > }
srujan@srujan-HP-Laptop-15-bs0xx:~/Desktop/132_837_1603/Project_code/Phase1$
```

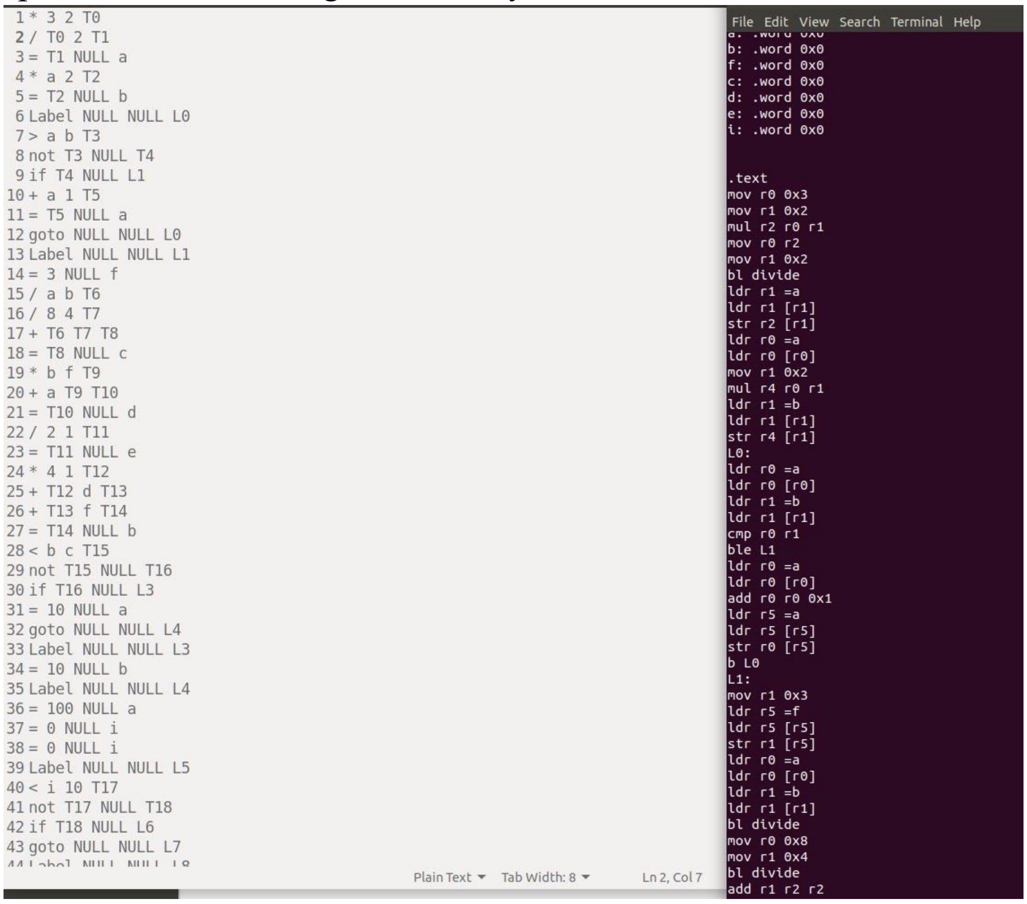
*Error Recovery:*



```
1 #include<stdio.h>
2
3 int main()
4 {
5 int ly=0;
6 int z=0, p=3;
7 int t=0 ,a=1, b=5;
8 int k=3;
9 int h;
10 for(int i=0;i<p;i++)
11 {
12 t=t+++;
13 }
14 k=9;
15 while(k>0)
16 {
17
18 k=k-1;
19 }
20 z=5;
21 if(y==0)
22 {
23 y=y+1;
24 h=y+k+a+b;
25 }
26 else
27 {
28
29 y= y*10;
30 z=y+7;
31 z=z+10;
32
33 }
34 }
```

```
srujan@srujan-HP-Laptop-15-bs0xx: ~/Desktop/132_837_1603/Project_code/Phase1
c -d program_test.y
srujan@srujan-HP-Laptop-15-bs0xx:~/Desktop/132_837_1603/Project_code/Phase1$ gcc
 lex.yy.c y.tab.c -ll
srujan@srujan-HP-Laptop-15-bs0xx:~/Desktop/132_837_1603/Project_code/Phase1$ ./a
.out
 error in line No 10
Line = > int o = 4 for
 error in line No 15
Line = > while
 error in line No 25
Line = > }
 error in line No 33
Line = > }
srujan@srujan-HP-Laptop-15-bs0xx:~/Desktop/132_837_1603/Project_code/Phase1$ ./a
.out
Note :
        Error Recovery : droping the integer value in line No 5

Note :
        Error Recovery : Droped extra '+' in the expression in line no 12

 error in line No 11
Line = > i ++ ) { t = t
srujan@srujan-HP-Laptop-15-bs0xx:~/Desktop/132_837_1603/Project_code/Phase1$
```

- **Target Code Generation:** C Program is written which converts the optimized ICG to Target Assembly Code.



```
* 3 2 T0
/ T0 2 T1
= T1 NULL a
* a 2 T2
= T2 NULL b
Label NULL NULL L0
> a b T3
not T3 NULL T4
if T4 NULL L1
+ a 1 T5
= T5 NULL a
goto NULL NULL L0
Label NULL NULL L1
= 3 NULL f
/ a b T6
/ 8 4 T7
+ T6 T7 T8
= T8 NULL c
* b f T9
+ a T9 T10
= T10 NULL d
/ 2 1 T11
= T11 NULL e
* 4 1 T12
+ T12 d T13
+ T13 f T14
= T14 NULL b
< b c T15
not T15 NULL T16
if T16 NULL L3
= 10 NULL a
goto NULL NULL L4
Label NULL NULL L3
= 10 NULL b
Label NULL NULL L4
= 100 NULL a
= 0 NULL i
= 0 NULL i
Label NULL NULL L5
< i 10 T17
not T17 NULL T18
if T18 NULL L6
goto NULL NULL L7
Label NULL NULL L8
```

```
a: .word 0x0
b: .word 0x0
f: .word 0x0
c: .word 0x0
d: .word 0x0
e: .word 0x0
i: .word 0x0


.text
mov r0 0x3
mov r1 0x2
mul r2 r0 r1
mov r0 r2
mov r1 0x2
bl divide
ldr r1 =a
ldr r1 [r1]
str r2 [r1]
ldr r0 =a
ldr r0 [r0]
mov r1 0x2
mul r4 r0 r1
ldr r1 =b
ldr r1 [r1]
str r4 [r1]
L0:
ldr r0 =a
ldr r0 [r0]
ldr r1 =b
ldr r1 [r1]
cmp r0 r1
ble L1
ldr r0 =a
ldr r0 [r0]
add r0 r0 0x1
ldr r5 =a
ldr r5 [r5]
str r0 [r5]
b L0
L1:
mov r1 0x3
ldr r5 =f
ldr r5 [r5]
str r1 [r5]
ldr r0 =a
ldr r0 [r0]
ldr r1 =b
ldr r1 [r1]
bl divide
mov r0 0x8
mov r1 0x4
bl divide
add r1 r2 r2
```

# Chapter 6- Implementation Details:

## Lexical and Syntax Phase:

- The tools used for implementing the code are lex and yacc respectively. The lex file contains the tokens specified with the aid of regular expressions and the yacc file contains grammar rules with corresponding actions.
- The scanner or lexical analyser transforms the source file from a stream of bits and bytes into a chain of meaningful tokens containing information that will be used on the later stages of the compiler. A global variable 'yylavl' is used to record the value of each lexeme scanned. 'yytext' is the lex variable that stores the matched string. Skipping over white spaces and recognizing all keywords, operators, variables and constants is handled in this phase.
- Scanning error gets reported when the input string is not matching any rule in the lex file.
- The rules consist of regular expressions that have corresponding actions which execute on a match with the source input.
- Syntax analysis is solely responsible for ensuring that the sequence of tokens creates a valid sentence provided the definition of your Programming Language grammar.
- While the code is being parsed, the tokens are generated whereas comments and extra spaces are ignored. For each new variable encountered, it is appended into the symbol table along with its attributes.
- Yacc tool that is used for parsing, reports shift-reduce and reduce-reduce conflicts on parsing an ambiguous grammar.
- We have implemented the symbol table as a linked list of structures. The members of the structures include variable name, line of declaration, data type, value, scope. Every new variable encountered in the program is entered into the symbol table.

## Semantic Phase:

- A structure is used as a record to keep track of the variables, constants, operators and the keywords in the input. As each line is parsed, the actions associated with the grammar rules are executed. A union which consists of int, float and char and void type to use it for assigning data type of each variable.

- Semantics Analysis uses information procured via the Symbol table to check for Semantics.
- The scope check takes place by having a variable which increments on every level of nesting. In this manner, the scope is checked for each variable and error messages are displayed if any term that is out of scope is used.
- $1 is used to refer to the first token in the given production and $$ is used to refer to the resultant of the given production. Expressions are evaluated and the values of the used variables are updated accordingly. At the end of the parsing, the updated symbol table is displayed.

**Abstract Syntax Tree:**

- After parsing is successful, we generate an abstract tree that is shown through the pre-order manner. To build the tree, a structure is maintained which has pointers to its children and a container for its data value.
- A tree structure representing the syntactical flow of the code is generated in this phase. The expression associativity is indicated using the %left and %right fields.
- A stack is used to push all the nodes into initially, followed by the popping of the nodes. When an operation is encountered, the left and right pointers of the node (node which has operation as its data in the data field) are assigned to the previously popped nodes and push the result back into the stack.

**Intermediate Code Generation:**

- The Intermediate code generator receives its input from its predecessor phase, the semantic analyser. Intermediate code tends to be machine independent code.
- Three Address Code: A statement involving no more than three references (two for operands and one for result) is known as a three address statement. A sequence of three address statements is known as three address code. Three address statements are of the form x = y op z, here x, y, z will have an address (memory location).
- The intermediate code generated is also displayed in quadruple format. It is shown with 4 columns- operator, operand1, operand2, and result.

**Optimisation of ICG:**

- After generating intermediate code, optimization is done via dead elimination.

- Dead Elimination: It is the process of removal of code that does not affect the program results. Removing such code has several benefits: it shrinks program size, an important consideration in some contexts, and it allows the running program to avoid executing irrelevant operations, which reduces its running time. It can also enable further optimizations by simplifying program structure.

**Target Code Generation:**
Post the generation of optimized ICG, we proceed to generate the Target Assembly Code.

# Chapter 7

## Results and Conclusions:

Thus, we have seen the design strategies and implementation of the different stages involved in building a mini C compiler and successfully built a working compiler that generates symbol table, abstract syntax tree, intermediate code, target code for a given C code as input.

## Shortcomings:

There exist a few shortcomings with respect to our implementation. The symbol table structure is the same across all types of tokens (constants, identifiers and operators). This can lead to some fields being empty for a few of the tokens. This can be optimized by using a better representation.

## Further Enhancements:

Improved memory utilisation with separate structures for the different types of tokens and a declared a union of these structures
Implement switch or do while construct
Handle more data types and keywords.

## References / Bibliography:

Compilers – Principles, Techniques, and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
Course material shared by the department.
https://www.javatpoint.com/compiler-tutorial
https://www.javatpoint.com/code-generation