Report on

“Lexical Analyzer for C Language”

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

Diploma

In

Computer Science & Engineering

Submitted by:

Sudhanshu Chaudhary	**E19481335500001**
Adarsh Yadav	**E19481335500004**
Amarjeet	**E19481335500005**
Ashutosh Kumar Mauriya	**E19481335500010**

Under the guidance of

Mr.Rakesh Kumar Mauriya

January – May 2022

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MAHAMAYA POLYTECHNIC OF INFORMATION TECHNOLOGY BANSI SIDDHARTHNAGAR

TABLE OF CONTENTS

Chapter No.	Title	Page No.

1. INTRODUCTION

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or code that a computer's processors use. The file used for writing a C-language contains what are called the source statements. The programmer then runs the appropriate language compiler, specifying the name of the file that contains the source statements. When executing, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages, builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. The output of the compilation is called object code or sometimes an object module.

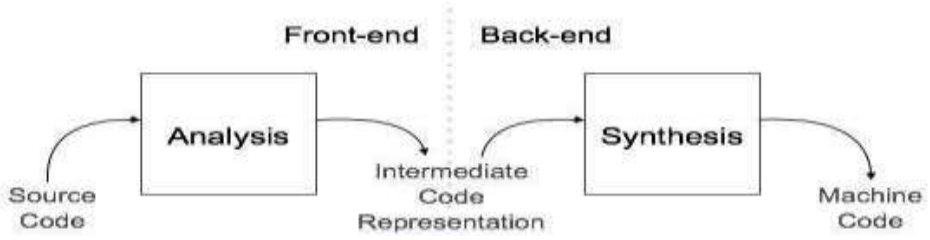
Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. Symbol table is an 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, etc.

Symbol table is used by both the analysis and the synthesis parts of a compiler. We have designed a lexical analyzer for the C language using lex. It takes as input a C code and outputs a stream of tokens. The tokens displayed as part of the output include keywords, identifiers, signed/unsigned integer/floating point constants, operators, special characters, headers, data-type specifiers, array, single-line comment, multi-line comment, preprocessor directive, pre-defined functions (printf and scanf), user-defined functions and the main function. The token, the type of token and the line number of the token in the C code are being displayed. The line number is displayed so that it is easier to debug the code for errors. Errors in single-line comments, multi-line comments are displayed along with line numbers. The output also contains the symbol table which contains tokens and their type. The symbol table is generated using the hash organisation.

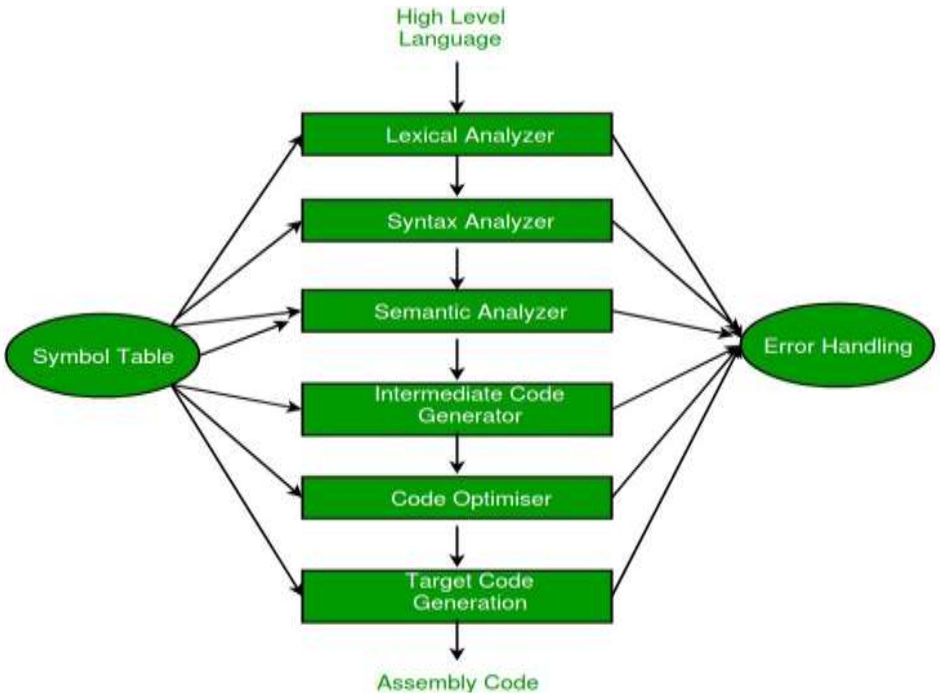
2. ARCHITECTURE OF LANGUAGE

1. Analysis phase : Known as the front-end of the compiler, the analysis phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors.The analysis phase generates an intermediate representation of the source program and symbol table.

This phase consists of:
➢ Lexical Analysis
➢ Syntax Analysis
➢ Semantic Analysis



2. *Synthesis phase:* Known as the back-end of the compiler, the synthesis phase generates the target program with the help of intermediate source code representation and symbol table. This phase consists of: ➢ Code Optimization
➢ Intermediate Code Generation



Lexical Analysis

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

Syntax Analysis

Syntax analysis or parsing is the second phase of a compiler. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree).

Semantic Analysis

Semantic analysis is the third phase of a compiler. Semantic analyzer checks whether the parse tree constructed by the syntax analyzer follows the rules of language.

Intermediate Code Generator

It generates intermediate code, that is a form which can be readily executed by machine. We have many popular intermediate codes. Example – Three address code etc. Intermediate code is converted to machine language using the last two phases which are platform dependent.

Till intermediate code, it is the same for every compiler out there, but after that, it depends on the platform. To build a new compiler we don’t need to build it from scratch. We can take the intermediate code from the already existing compiler and build the last two parts.

Code Optimizer

It transforms the code so that it consumes fewer resources and produces more speed. The meaning of the code being transformed is not altered. Optimisation can be categorized into two types: machine dependent and machine independent.

In our project, we have handles the following constructs:

- Looping construct: while, for, do-while
- Data types: (signed/unsigned) int, float
- Arithmetic and Relational Operators
- Data structure: Arrays
- User defined functions
- Keywords of C language
- Single and Multi-line comments
- Identifiers and Constant errors
- Selection statement: (nested) if-else, while

3. CONTEXT FREE GRAMMAR

begin
: external_declaration
| begin external_declaration
| Define begin
;
primary_expression : IDENTIFIER {
insertToHash($<str>1,
data_type , yylineno); }
| CONSTANT
| STRING_LITERAL
| '(' expression ')'
;
Define
: DEFINE
;
postfix_expression :
primary_expression
| postfix_expression '[' expression ']'
| postfix_expression '(' ')' |
postfix_expression '('
argument_expression_list ')' |
postfix_expression '.' IDENTIFIER
| postfix_expression PTR_OP IDENTIFIER
| postfix_expression INC_OP
| postfix_expression DEC_OP
;
argument_expression_list :
assignment_expression |
argument_expression_list ','
assignment_expression
;
unary_expression :
postfix_expression |

INC_OP
unary_expre
ssion
| DEC_OP unary_expression
| unary_operator cast_expression
| SIZEOF unary_expression
| SIZEOF '(' type_name ')'
;
unary_operator
: '&'
| '*'
| '+'
| '-'
| '~'
| '!'
;
cast_expression :
unary_expression
| '(' type_name ')' cast_expression
;
multiplicative_expression :
cast_expression |
multiplicative_expression '*'
cast_expression
| multiplicative_expression '/'
cast_expression |
multiplicative_expression '%'
cast_expression
;
additive_expression :
multiplicative_expression |
additive_expression '+'
multiplicative_expression |
additive_expression '-'
multiplicative_expression

logical_and_expression
;
shift_expression : inclusive_or_expression : additive_expression |
logical_and_expression AND_OP
| shift_expression LEFT_OP inclusive_or_expression
additive_expression ;
| shift_expression RIGHT_OP logical_or_expression additive_expression :
logical_and_expression

; | logical_or_expression OR_OP
relational_expression logical_and_expression
: shift_expression ;
| relational_expression '<' conditional_expression
shift_expression : logical_or_expression
| relational_expression '>' | logical_or_expression '?' expression ':' shift_expression
conditional_expression
| relational_expression LE_OP ; shift_expression assignment_expression
| relational_expression GE_OP : conditional_expression shift_expression |
unary_expression assignment_operator
; assignment_expression
equality_expression ;
: relational_expression assignment_operator
| equality_expression EQ_OP : '='
relational_expression
| equality_expression NE_OP | MUL_ASSIGN| DIV_ASSIGN relational_expression |
MOD_ASSIGN
| ADD_ASSIGN
; | SUB_ASSIGN and_expression | LEFT_ASSIGN
: equality_expression | RIGHT_ASSIGN
| and_expression '&' equality_expression | AND_ASSIGN
; | XOR_ASSIGN
exclusive_or_expression | OR_ASSIGN
: and_expression ;
| exclusive_or_expression '^' expression
and_expression: assignment_expression ; | expression ',' assignment_expression
inclusive_or_expression ;
: exclusive_or_expression constant_expression | inclusive_or_expression
'|' : conditional_expression exclusive_or_expression ;
; declaration
: declaration_specifiers ';' | : init_declarator
declaration_specifiers | init_declarator_list ',' init_declarator
init_declarator_list ';' ;
; init_declarator :
declaration_specifiers : declarator
storage_class_specifier | | declarator '=' initializer
storage_class_specifier ;
declaration_specifiers | type_specifier storage_class_specifier
{ strcpy(data_type, : TYPEDEF
$<str>1); } | EXTERN
| type_specifier declaration_specifiers | STATIC
; | AUTO
init_declarator_list | REGISTER

```
;
type_specifier :
VOID
| CHAR
| SHORT
| INT
| LONG
| FLOAT
| DOUBLE
| SIGNED
| UNSIGNED
| struct_or_union_specifier
;
specifier_qualifier_list
: type_specifier specifier_qualifier_list
| type_specifier
| CONST specifier_qualifier_list
| CONST
;
struct_or_union_specifier :
struct_or_union IDENTIFIER '{'
struct_declaration_list '}' ';' |
struct_or_union '{'
struct_declaration_list '}' ';'
| struct_or_union IDENTIFIER ';'
;
struct_or_unio
n : STRUCT
| UNION
;
struct_declaration_list
: struct_declaration |
struct_declaration_list
struct_declaration
;
struct_declaration :
specifier_qualifier_list
struct_declarator_list ';'
;
struct_declarator_list
: declarator
| struct_declarator_list ',' declarator
;
declarator : pointer
direct_declarator
| direct_declarator
;
direct_declarato
r : IDENTIFIER
| '(' declarator ')'
| direct_declarator '['
constant_expression ']'
| direct_declarator '['
']'
| direct_declarator '(' parameter_list ')'
| direct_declarator '(' identifier_list ')'
| direct_declarator '(' ')'
;
pointer
: '*'
| '*' pointer
;
parameter_list :
parameter_declaration
| parameter_list ',' parameter_declaration
;
parameter_declaration :
declaration_specifiers declarator
| declaration_specifiers
;
identifier_list
: IDENTIFIER
| identifier_list ',' IDENTIFIER
;
type_name
: specifier_qualifier_list
| specifier_qualifier_list declarator
;

initializer
: assignment_expression
```

| '{' initializer_list '}'
| '{' initializer_list ',' '}'
;
initializer_list
: initializer
| initializer_list ',' initializer
;
statement :
compound_statement
| expression_statement
| selection_statement
| iteration_statement
| jump_statement
;
compound_statement
: '{' '}'
| '{' statement_list '}'
| '{' declaration_list '}'
| '{' declaration_list statement_list '}'
;
declaration_list
: declaration
| declaration_list declaration
;
statement_list
: statement
| statement_list statement
;
expression_statement
;
function_definition
: declaration_specifiers declarator declaration_list
compound_statement
| declaration_specifiers declarator
compound_statement

: ';'
| expression ';'
;
selection_statement
: IF '(' expression ')' statement %prec
NO_ELSE
| IF '(' expression ')' statement ELSE
statement

;
iteration_statement : WHILE '('
expression ')' statement
| DO statement WHILE '(' expression ')' ';' |
FOR '(' expression_statement
expression_statement ')' statement | FOR
'(' expression_statement
expression_statement expression ')'
statement
;
jump_statemen
t : CONTINUE ';'
| BREAK ';'
| RETURN ';'
| RETURN expression ';'
;
external_declaration
: function_definition
| declaration

| declarator declaration_list
compound_statement
| declarator compound_statement

4. DESIGN STRATEGY

4.1 Symbol Table Creation

The symbol table will be created for lexical analysis, semantic analysis, syntax analysis and Intermediate code generation separately. The data structure that would be used in order to create the symbol table is HashTable and the method that would be used is hashing.

4.1.1 Lexical Analysis:

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens (strings with an identified "meaning"). A program that performs lexical analysis may be called a lexer, tokenizer, or scanner (though "scanner" is also used to refer to the first stage of a lexer). Such a lexer is generally combined with a parser, which together analyze the syntax of programming languages, web pages, and so forth. The script written by us is a computer program called the “lex” program, is the one that generates lexical analyzers ("scanners" or "lexers"). Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

4.1.2 Syntax Analysis:

The syntax analyser for the C language by writing two scripts, one that acts as a lexical analyzer (lexer) and outputs a stream of tokens, and the other one that acts as a parser. The lexer is known as the lex program. Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

4.1.3 Semantic Analysis:

Semantic analysis is the task of ensuring that the declarations and statements of a program are Semantically correct, i.e that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used.

Implementing the semantic actions is conceptually simpler in recursive descent parsing because they are simply added to the recursive procedures. Some of the functions of Semantic analysis are that it maintains and updates the symbol table, check source programs for semantic errors and warnings like type mismatch, global and local scope of a variable, re-definition of variables, usage of undeclared variables.

4.2 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, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generators assume an unlimited number of memory storage (register) to generate code.

A three-address code has at most three address locations to calculate the expression. Hence, the intermediate code generator will divide any expression into sub-expressions and then generate the corresponding code. A three-address code can be represented in two forms : quadruples and triples.

Quadruples : Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result.

4.3 Code Optimization

To optimize the obtained Intermediate code, we have resorted to using three strategies:

1. **Elimination of dead code** - In compiler theory, dead code elimination is a compiler optimization to remove code which 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.
2. **Constant folding -** Constant folding is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime
3. **Common Subexpression elimination** - it is a compiler optimization technique that searches for instances of identical expressions (i.e., they all evaluate to the same value), and analyzes whether it is worthwhile replacing them with a single variable holding the computed value.

These strategies are implemented in a python script that takes the output of the ICG file as its input and then implements the above strategies and gives the optimized code as the output.

4.4 Error Handling

When the compiler encounters an error, firstly, it displays the kind of error it encountered and secondly, it tries to handle those errors such that flow of the compiler doesn’t break.

A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process. A program may have the following kinds of errors at various stages:

- Lexical : name of some identifier typed incorrectly
- Syntactical : missing semicolon or unbalanced parenthesis
- Semantical : incompatible value assignment ● Logical : code not reachable, infinite loop

5. IMPLEMENTATION DETAILS

5.1 Symbol Table Creation

5.1.1 Lexical Analysis

Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.
Hash tables and stacks were used to implement the Symbol table at this phase.
The structure of the lex program consists of three sections:
{definition section}
%%
{rules section}
%%
{C code section}
The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.
The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time. The lex program, when compiled using the

lex command, generates a file called lex.yy.c, which when executed recognizes the tokens present in the input C program.

5.1.2 Syntax Analysis

The Yacc program specifies productions for the following:

- Looping construct: while, for, do-while
- Data types: (signed/unsigned) int, float
- Arithmetic and Relational Operators
- Data structure: Arrays
- User defined functions
- Keywords of C language
- Single and Multi-line comments
- Identifiers and Constant errors
- Selection statement: (nested) if-else, while

After parsing, if there are errors then the line numbers of those errors are displayed along with a 'parsing failed' message on the terminal. Otherwise, a 'parsing complete' message is displayed on the console.
Hash tables and stacks were used to implement the Symbol table.

5.1.3 Semantic Analysis

A YACC source program is structurally similar to a LEX one.
Declarations
%%
Rules
%%
Routines

- The declaration section may be empty. Moreover, if the routines section is omitted, the second %% mark may be omitted also.
- Blanks, tabs and newlines are ignored except that they may not appear in names.

Declarations Section
- Declarations of tokens. Yacc requires token names to be declared as such using the keyword %token
- Declaration of the start symbol using the keyword %start.
- C declarations: included files, global variables, types.

● C code between %{ and % }

Rules Section

A rule has the form:

Nonterminal : sentential form

| sentential form

………………

| sentential form

;

Actions may be associated with rules and are executed when the associated sentential form is matched.

Hash tables and stacks were used to implement the Symbol table.

5.2 Intermediate Code Generation

To implement the ICG code, we used data structures like stacks and arrays. Extensive use of pointers and structures was also made to implement it.

parser.l : Lex file which defines all the terminals of the productions stated in the yacc file. It contains regular expressions.

parser.y : Yacc file is where the productions for all the loops, selection and conditional statements and expressions are mentioned. This file also contains the semantic rules defined against every production necessary. Rules for producing three address code are also present against the productions and in functions.

symbolTable.c : It is the C file which generates the symbol table. It is included along with the yacc file

test.c : The input C code which will be parsed and checked for semantic correctness by executing the lex and yacc files along with it.

Yacc file has productions to check the following functionalities:

● Preprocessor directives

● Function definition

● Compound statements

● Nested compound statements

● if else

● Nested while

● Variable definition and declaration

● Assignment operation

● Arithmetic operations

5.3 Code Optimization

The language used to write the optimization phase was Python. The three strategies were then implemented using sets, stack, lists, dictionaries and arrays. A main function was used to call the various strategies and the subsequent results were printed on the screen.

5.4 Error Handling

When the compiler encounters an error, firstly, it displays the kind of error it encountered and secondly, it tries to handle those errors such that flow of the compiler doesn't break.

The following strategies may be used for handling the errors:
There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic mode

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semicolon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement mode

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of the statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing a comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions

Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

5.5 Instructions on how to build and run your program.

5.5.1Lexical Analysis

The lex file (lexer.l) is compiled using following commands:

```
lex lexer.l
./a.out
```

5.5.2 Syntax Analysis

The lex file (parser.l) and yacc (parser.y) are compiled using following commands:

```
lex parser.l yacc -d parser.y gcc y.tab.c -ll -w ./a.out
test1.c
```

5.5.3 Semantic Analysis

The lex file (parser.l) and yacc (parser.y) are compiled using following commands:

```
lex parser.l
yacc parser.y
cc y.tab.h -ll
./a.out test2.c
```

5.5.4 Abstract Syntax Tree

```
yacc -d AST.y
lex AST.l
gcc -g y.tab.c lex.yy.c graph.c -ll -o AST
./AST < input.c
```

5.5.5 Intermediate Code Generation

The lex (parser.l) and yacc (parser.y) codes are compiled with input file (test.c)

```
lex parser.l yacc parser.y gcc y.tab.c -ll -ly ./a.out
test.c
```

5.5.6 ICG in quadruple format

```
lex icg_quad.l yacc
icg_quad.y gcc y.tab.c
-ll -ly -w ./a.out
```

5.5.7 Code Optimization

The python file(OptimizeICG.py) is compiled and executed by the following terminal commands to parse the given input file (output_file.txt).

```
python3 OptimizeICG.py output_file.txt
```

6. RESULTS

6.1 Lexical Analysis

Lexemes or tokens are generated after the code is passed through the lexer.l file. These tokens are then updated in the symbol table.

6.2 Syntax Analysis

The lexical analyzer and the syntax analyzer for a subset of C language, which include selection statements, compound statements, iteration statements, jumping statements, user defined functions and primary expressions, are generated. It is important to note that conflicts (shift-reduce and reduce-reduce) may occur in case of syntax analyzer if proper care is not taken while specifying the context-free grammar for the language. We should always specify unambiguous grammar for the parser to work properly.

6.3 Semantic Analysis

After parsing, if there are errors then the line numbers of those errors are displayed along with a ‘parsing failed’ on the terminal. Otherwise, a ‘parsing complete’ message is displayed on the console. The symbol table with stored & updated values is always displayed, irrespective of errors.

6.4 Intermediate Code Generation

After parsing, if there are errors then the line numbers of those errors are displayed along with a ‘parsing failed’ on the terminal. Otherwise, a ‘parsing complete’ message is displayed on the console. The symbol table with stored & updated values is always displayed, irrespective of errors. Also the three address codes along with the temporary variables are also displayed along with the flow of the conditional and iterative statements.

6.5 Code Optimization

It removed redundant code without changing the meaning of the program.
Accomplished our main objectives:

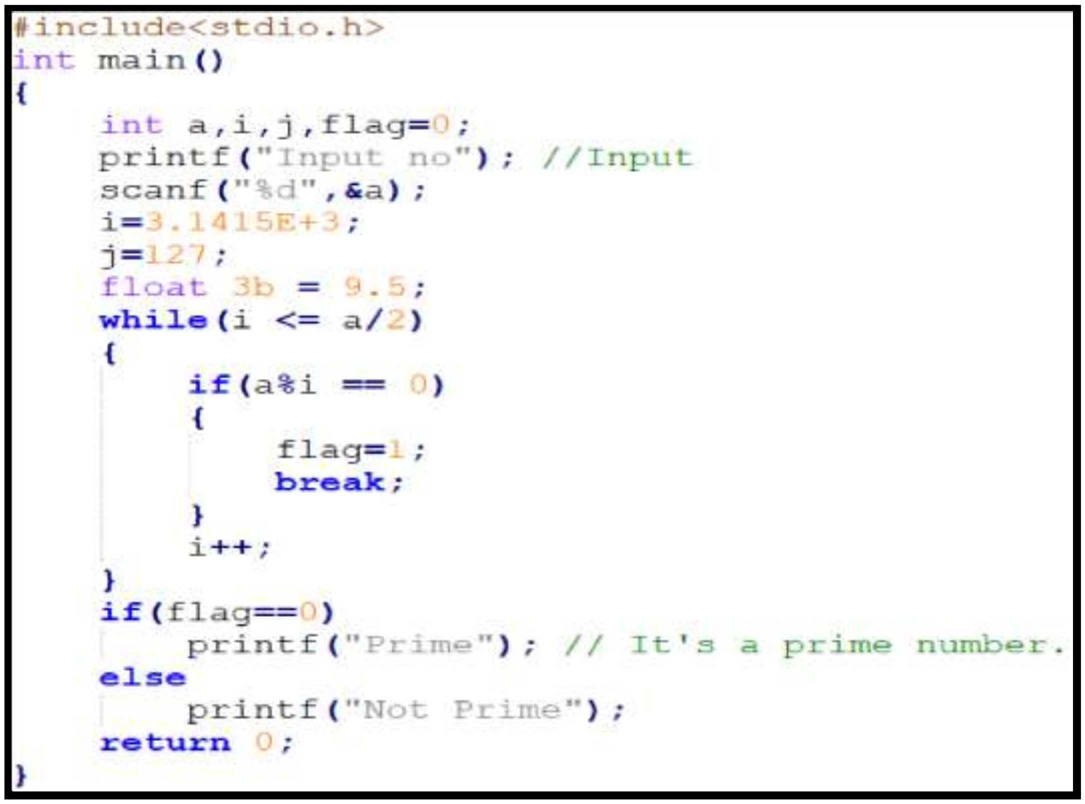
1. Reduce execution speed
2. Reduce code size

We achieved this through code transformation while preserving semantics.

7. SNAPSHOTS

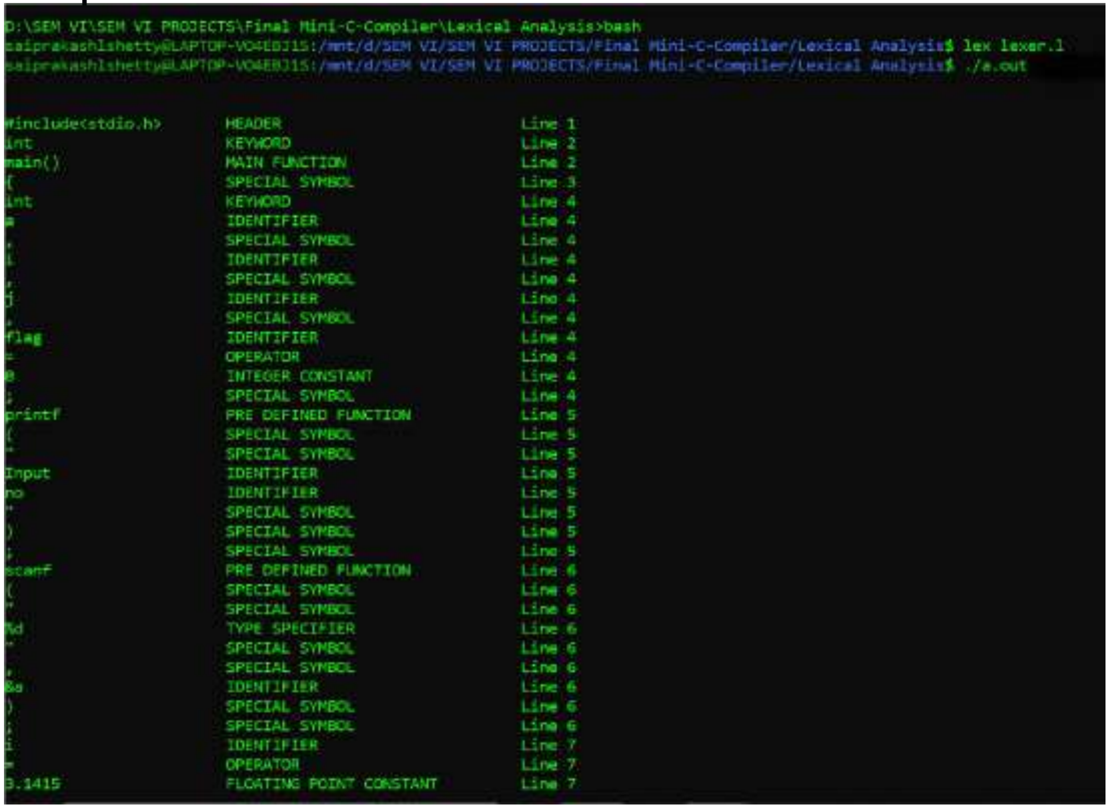
7.1 Lexical Analysis

Test case : **isPrime.c**

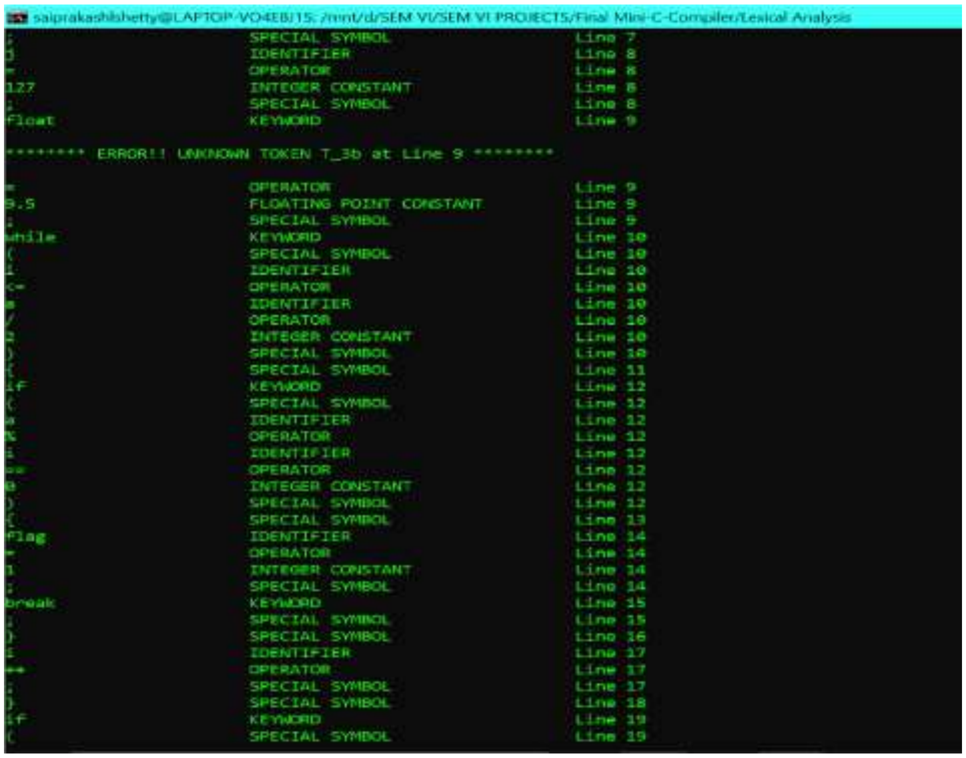


```c
#include<stdio.h>
int main()
{
    int a,i,j,flag=0;
    printf("Input no"); //Input
    scanf("%d",&a);
    i=3.1415E+3;
    j=127;
    float 3b = 9.5;
    while(i <= a/2)
    {
        if(a%i == 0)
        {
            flag=1;
            break;
        }
        i++;
    }
    if(flag==0)
        printf("Prime"); // It's a prime number.
    else
        printf("Not Prime");
    return 0;
}
```

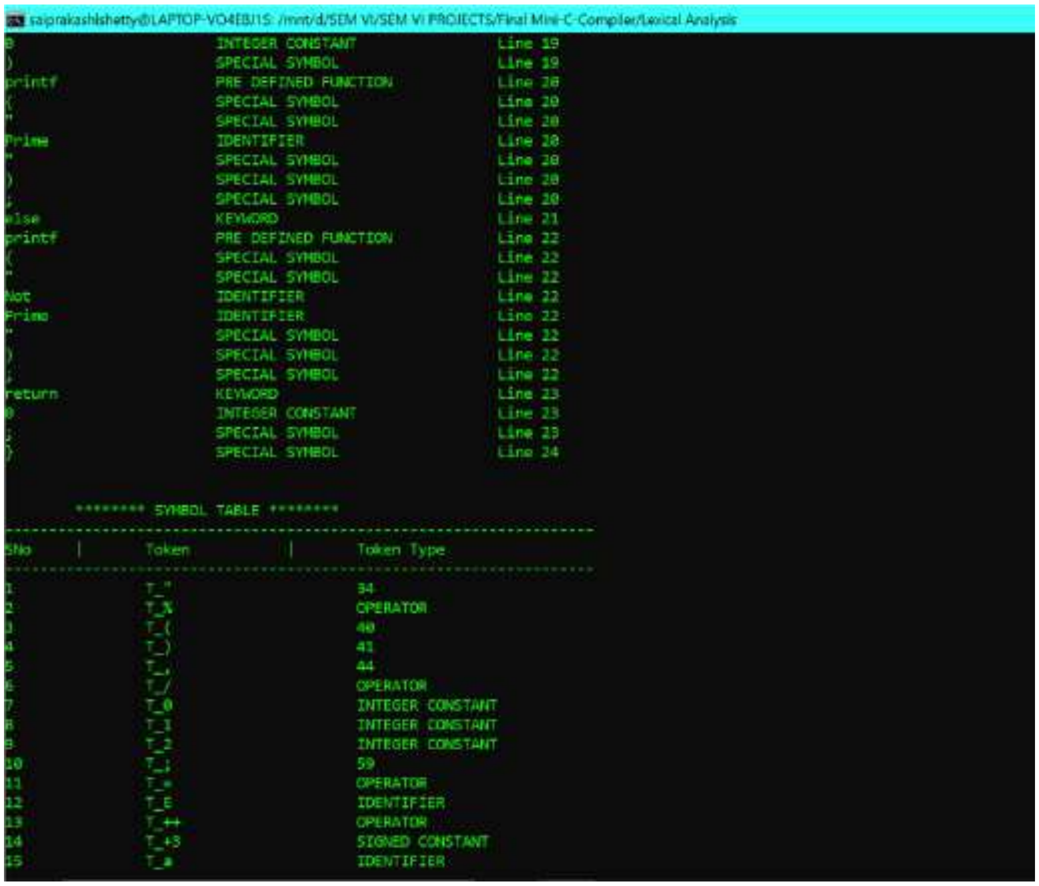
Output :



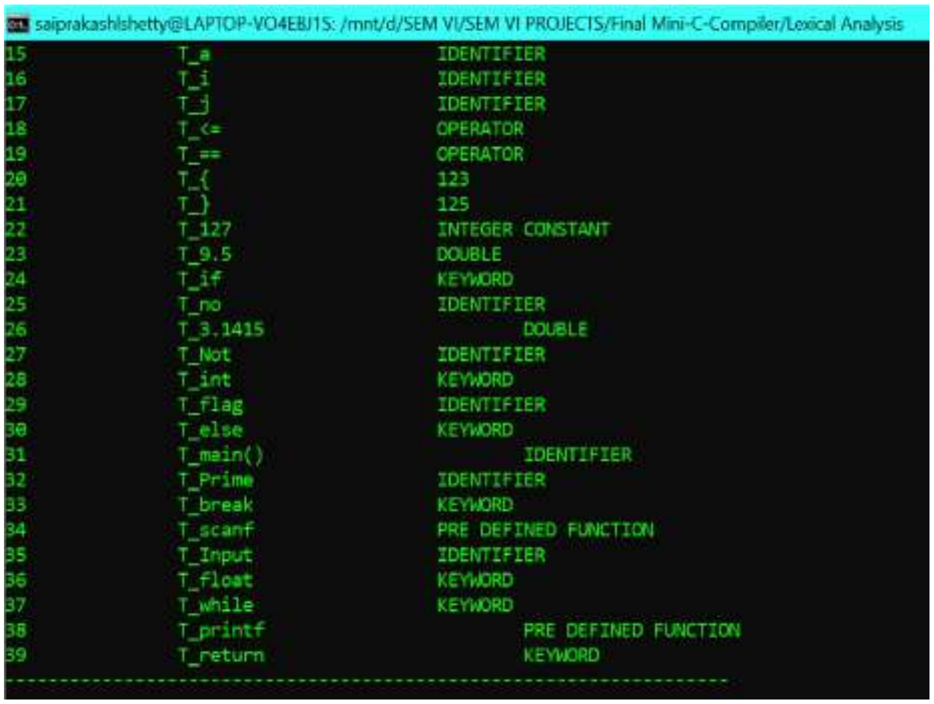
```
D:\SEM VI\SEM VI PROJECTS\Final Mini-C-Compiler\Lexical Analysis>bash
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Lexical Analysis$ lex lexer.l
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Lexical Analysis$ ./a.out


#include<stdio.h>        HEADER                     Line 1
int                      KEYWORD                    Line 2
main()                   MAIN FUNCTION              Line 2
{                        SPECIAL SYMBOL             Line 3
int                      KEYWORD                    Line 4
a                        IDENTIFIER                 Line 4
,                        SPECIAL SYMBOL             Line 4
i                        IDENTIFIER                 Line 4
,                        SPECIAL SYMBOL             Line 4
j                        IDENTIFIER                 Line 4
,                        SPECIAL SYMBOL             Line 4
flag                     IDENTIFIER                 Line 4
=                        OPERATOR                   Line 4
0                        INTEGER CONSTANT           Line 4
;                        SPECIAL SYMBOL             Line 4
printf                   PRE DEFINED FUNCTION       Line 5
(                        SPECIAL SYMBOL             Line 5
"                        SPECIAL SYMBOL             Line 5
Input                    IDENTIFIER                 Line 5
no                       IDENTIFIER                 Line 5
"                        SPECIAL SYMBOL             Line 5
)                        SPECIAL SYMBOL             Line 5
;                        SPECIAL SYMBOL             Line 5
scanf                    PRE DEFINED FUNCTION       Line 6
(                        SPECIAL SYMBOL             Line 6
"                        SPECIAL SYMBOL             Line 6
%d                       TYPE SPECIFIER             Line 6
"                        SPECIAL SYMBOL             Line 6
,                        SPECIAL SYMBOL             Line 6
&a                       IDENTIFIER                 Line 6
)                        SPECIAL SYMBOL             Line 6
;                        SPECIAL SYMBOL             Line 6
i                        IDENTIFIER                 Line 7
=                        OPERATOR                   Line 7
3.1415                   FLOATING POINT CONSTANT    Line 7
```



```
saiprakashlshetty@LAPTOP-VO4EBJ15: /mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Lexical Analysis
;                        SPECIAL SYMBOL                      Line 7
j                        IDENTIFIER                          Line 8
=                        OPERATOR                            Line 8
127                      INTEGER CONSTANT                    Line 8
;                        SPECIAL SYMBOL                      Line 8
float                    KEYWORD                             Line 9

******** ERROR!! UNKNOWN TOKEN T_3b at Line 9 ********

=                        OPERATOR                            Line 9
9.5                      FLOATING POINT CONSTANT             Line 9
;                        SPECIAL SYMBOL                      Line 9
while                    KEYWORD                             Line 10
(                        SPECIAL SYMBOL                      Line 10
i                        IDENTIFIER                          Line 10
<=                       OPERATOR                            Line 10
a                        IDENTIFIER                          Line 10
/                        OPERATOR                            Line 10
2                        INTEGER CONSTANT                    Line 10
)                        SPECIAL SYMBOL                      Line 10
{                        SPECIAL SYMBOL                      Line 11
if                       KEYWORD                             Line 12
(                        SPECIAL SYMBOL                      Line 12
a                        IDENTIFIER                          Line 12
%                        OPERATOR                            Line 12
i                        IDENTIFIER                          Line 12
==                       OPERATOR                            Line 12
0                        INTEGER CONSTANT                    Line 12
)                        SPECIAL SYMBOL                      Line 12
{                        SPECIAL SYMBOL                      Line 13
flag                     IDENTIFIER                          Line 14
=                        OPERATOR                            Line 14
1                        INTEGER CONSTANT                    Line 14
;                        SPECIAL SYMBOL                      Line 14
break                    KEYWORD                             Line 15
;                        SPECIAL SYMBOL                      Line 15
}                        SPECIAL SYMBOL                      Line 16
i                        IDENTIFIER                          Line 17
++                       OPERATOR                            Line 17
;                        SPECIAL SYMBOL                      Line 17
}                        SPECIAL SYMBOL                      Line 18
if                       KEYWORD                             Line 19
(                        SPECIAL SYMBOL                      Line 19
```



```
saiprakashlshetty@LAPTOP-VO4EBJ15: /mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Lexical Analysis
0                        INTEGER CONSTANT                    Line 19
)                        SPECIAL SYMBOL                      Line 19
printf                   PRE DEFINED FUNCTION                Line 20
(                        SPECIAL SYMBOL                      Line 20
"                        SPECIAL SYMBOL                      Line 20
Prime                    IDENTIFIER                          Line 20
"                        SPECIAL SYMBOL                      Line 20
)                        SPECIAL SYMBOL                      Line 20
;                        SPECIAL SYMBOL                      Line 20
else                     KEYWORD                             Line 21
printf                   PRE DEFINED FUNCTION                Line 22
(                        SPECIAL SYMBOL                      Line 22
"                        SPECIAL SYMBOL                      Line 22
Not                      IDENTIFIER                          Line 22
Prime                    IDENTIFIER                          Line 22
"                        SPECIAL SYMBOL                      Line 22
)                        SPECIAL SYMBOL                      Line 22
;                        SPECIAL SYMBOL                      Line 22
return                   KEYWORD                             Line 23
0                        INTEGER CONSTANT                    Line 23
;                        SPECIAL SYMBOL                      Line 23
}                        SPECIAL SYMBOL                      Line 24


        ******** SYMBOL TABLE ********
--------------------------------------------------------------------
SNo     |         Token            |        Token Type
--------------------------------------------------------------------
1                 T_"                       34
2                 T_%                       OPERATOR
3                 T_(                       40
4                 T_)                       41
5                 T_,                       44
6                 T_/                       OPERATOR
7                 T_0                       INTEGER CONSTANT
8                 T_1                       INTEGER CONSTANT
9                 T_2                       INTEGER CONSTANT
10                T_;                       59
11                T_=                       OPERATOR
12                T_E                       IDENTIFIER
13                T_++                      OPERATOR
14                T_+3                      SIGNED CONSTANT
15                T_a                       IDENTIFIER
```

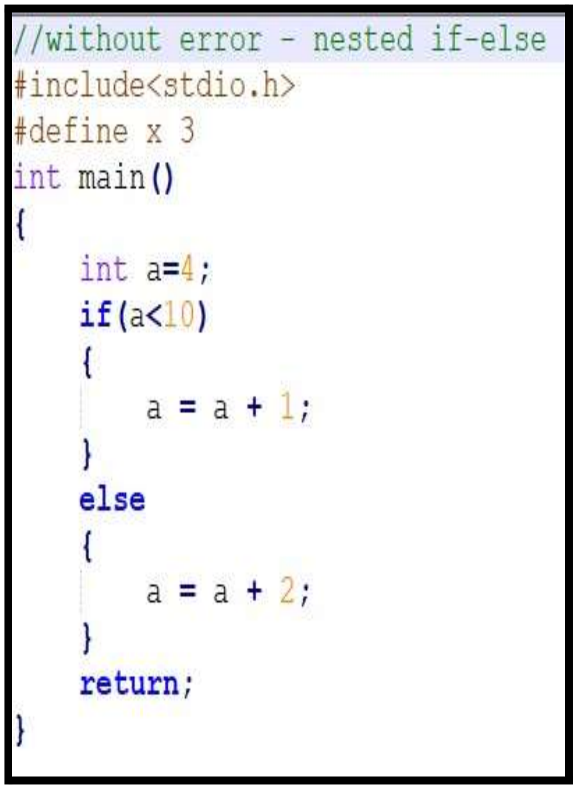


```
saiprakashlshetty@LAPTOP-VO4EBJ1S: /mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Lexical Analysis
15              T_a                     IDENTIFIER
16              T_i                     IDENTIFIER
17              T_j                     IDENTIFIER
18              T_<=                    OPERATOR
19              T_==                    OPERATOR
20              T_{                     123
21              T_}                     125
22              T_127                   INTEGER CONSTANT
23              T_9.5                   DOUBLE
24              T_if                    KEYWORD
25              T_no                    IDENTIFIER
26              T_3.1415                        DOUBLE
27              T_Not                   IDENTIFIER
28              T_int                   KEYWORD
29              T_flag                  IDENTIFIER
30              T_else                  KEYWORD
31              T_main()                        IDENTIFIER
32              T_Prime                 IDENTIFIER
33              T_break                 KEYWORD
34              T_scanf                 PRE DEFINED FUNCTION
35              T_Input                 IDENTIFIER
36              T_float                 KEYWORD
37              T_while                 KEYWORD
38              T_printf                        PRE DEFINED FUNCTION
39              T_return                        KEYWORD
--------------------------------------------------------------------
```

7.2 Syntax Analysis

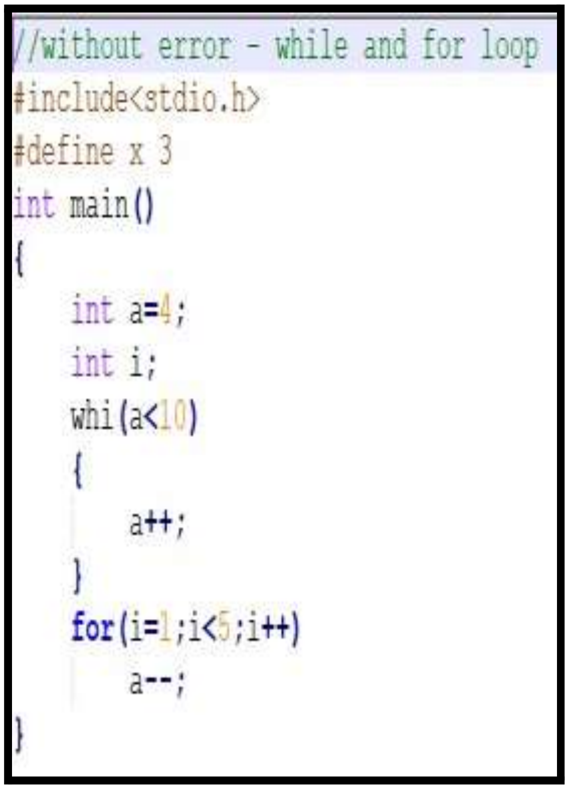
Test Cases:

test1.c



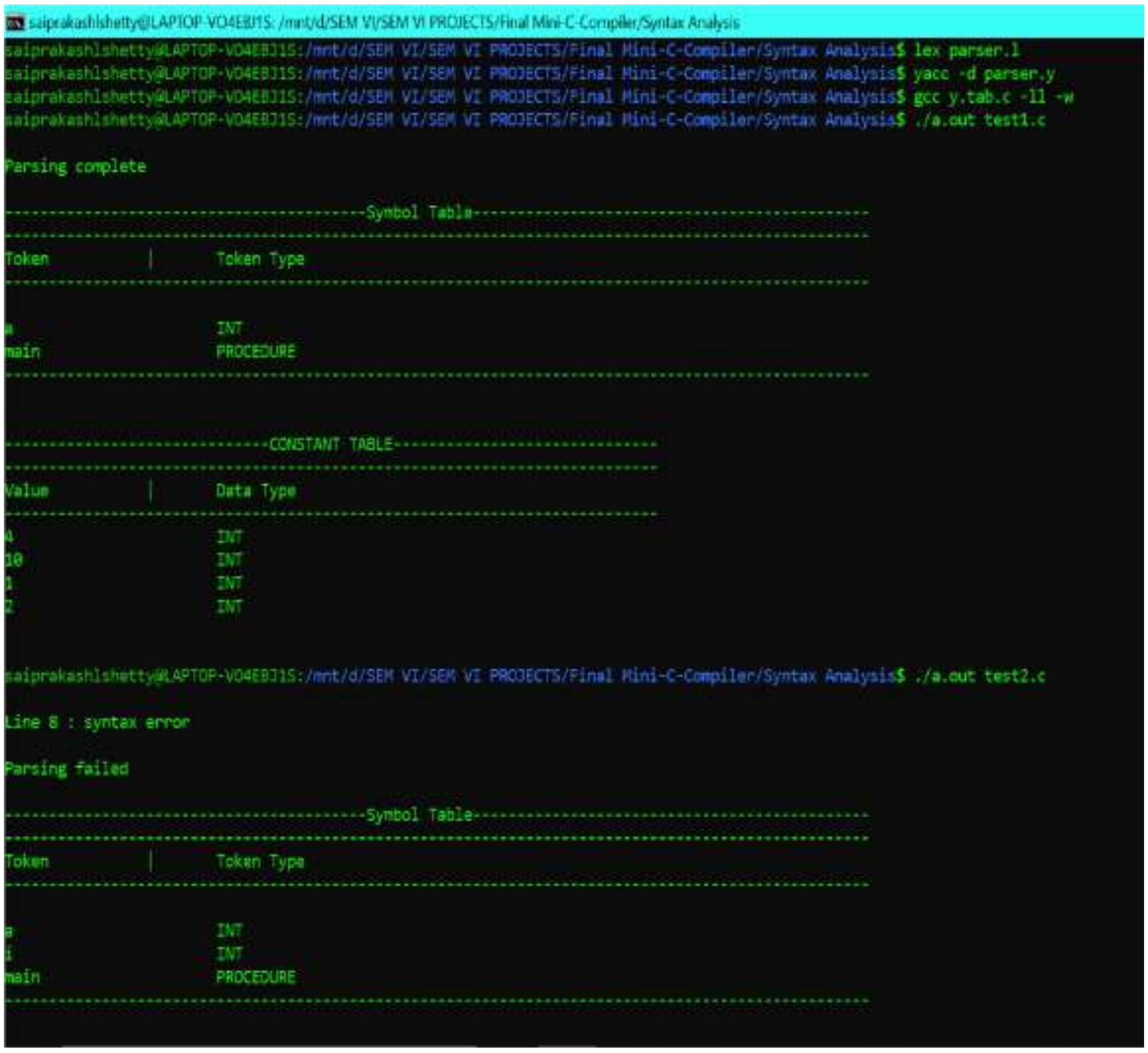
```c
//without error - nested if-else
#include<stdio.h>
#define x 3
int main()
{
    int a=4;
    if(a<10)
    {
        a = a + 1;
    }
    else
    {
        a = a + 2;
    }
    return;
}
```

test2.c

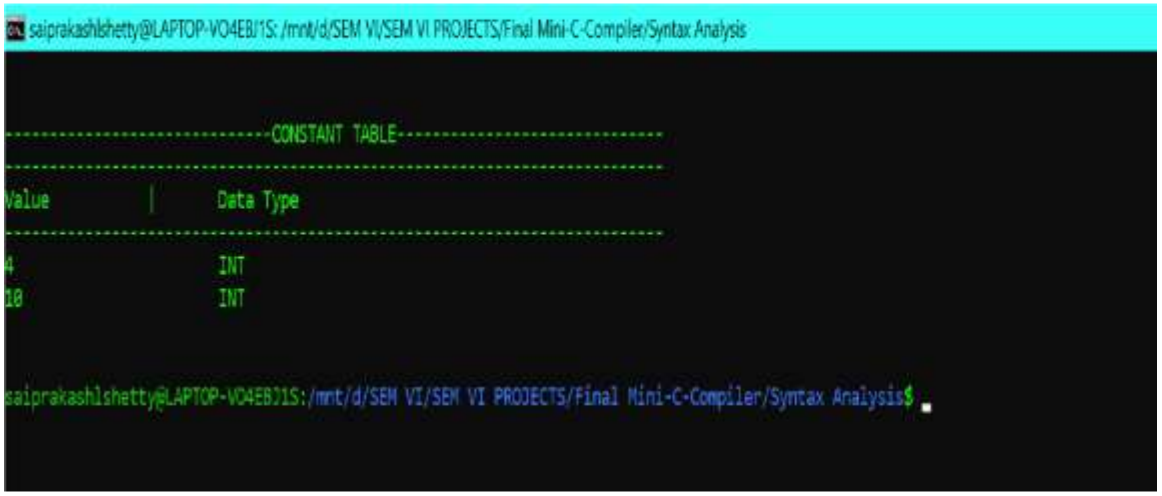


```c
//without error - while and for loop
#include<stdio.h>
#define x 3
int main()
{
    int a=4;
    int i;
    whi(a<10)
    {
        a++;
    }
    for(i=1;i<5;i++)
        a--;
}
```

Output :



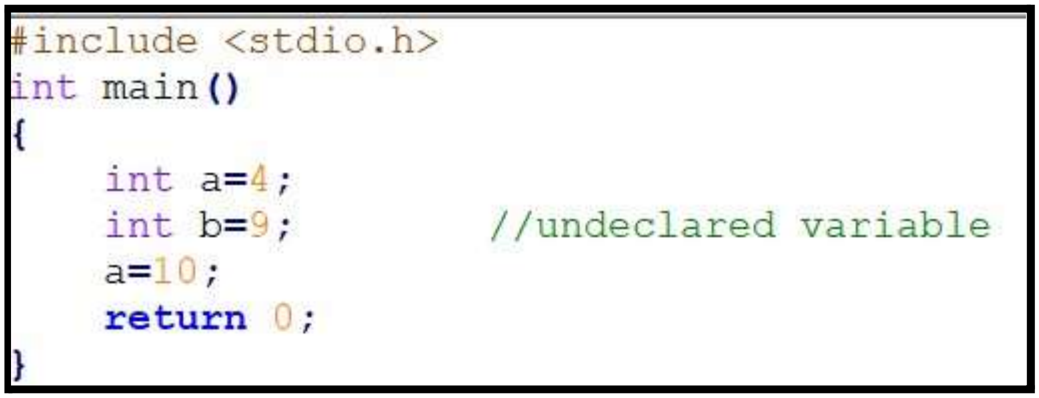
```
saiprakashlshetty@LAPTOP-VO4EBJ1S: /mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Syntax Analysis
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Syntax Analysis$ lex parser.l
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Syntax Analysis$ yacc -d parser.y
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Syntax Analysis$ gcc y.tab.c -ll -w
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Syntax Analysis$ ./a.out test1.c

Parsing complete

-----------------------------------------Symbol Table-------------------------------------------
------------------------------------------------------------------------------------------------
Token           |         Token Type
------------------------------------------------------------------------------------------------

a                         INT
main                      PROCEDURE
------------------------------------------------------------------------------------------------


-----------------------------CONSTANT TABLE-----------------------------
------------------------------------------------------------------------
Value           |         Data Type
------------------------------------------------------------------------
4                         INT
10                        INT
1                         INT
2                         INT

saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Syntax Analysis$ ./a.out test2.c

Line 8 : syntax error

Parsing failed

-----------------------------------------Symbol Table-------------------------------------------
------------------------------------------------------------------------------------------------
Token           |         Token Type
------------------------------------------------------------------------------------------------

a                         INT
i                         INT
main                      PROCEDURE
------------------------------------------------------------------------------------------------
```



```
saiprakashlshetty@LAPTOP-VO4EBJ1S: /mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Syntax Analysis


-----------------------------CONSTANT TABLE-----------------------------
------------------------------------------------------------------------
Value           |         Data Type
------------------------------------------------------------------------
4                         INT
10                        INT


saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Syntax Analysis$
```

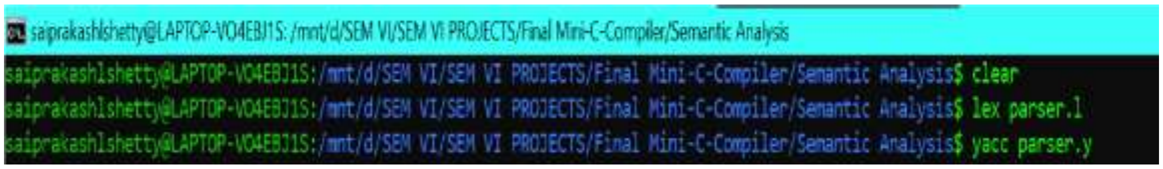
7.3 Semantic Analysis

Test case :

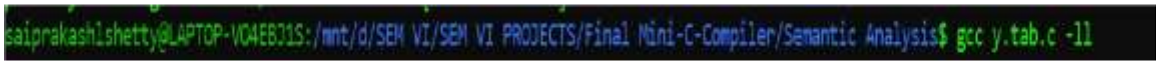


```
#include <stdio.h>
int main()
{
    int a=4;
    int b=9;        //undeclared variable
    a=10;
    return 0;
}
```

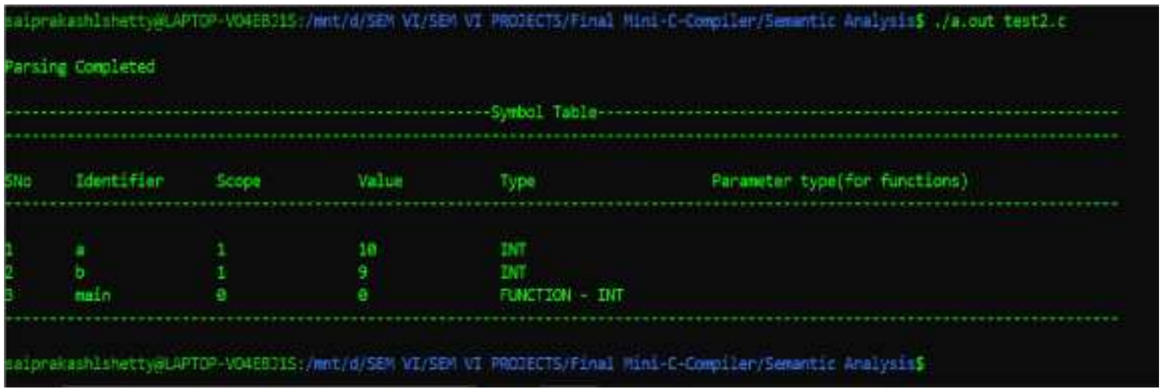
Output :



```
saiprakashlshetty@LAPTOP-VO4EBJ1S: /mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Semantic Analysis
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Semantic Analysis$ clear
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Semantic Analysis$ lex parser.l
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Semantic Analysis$ yacc parser.y
```



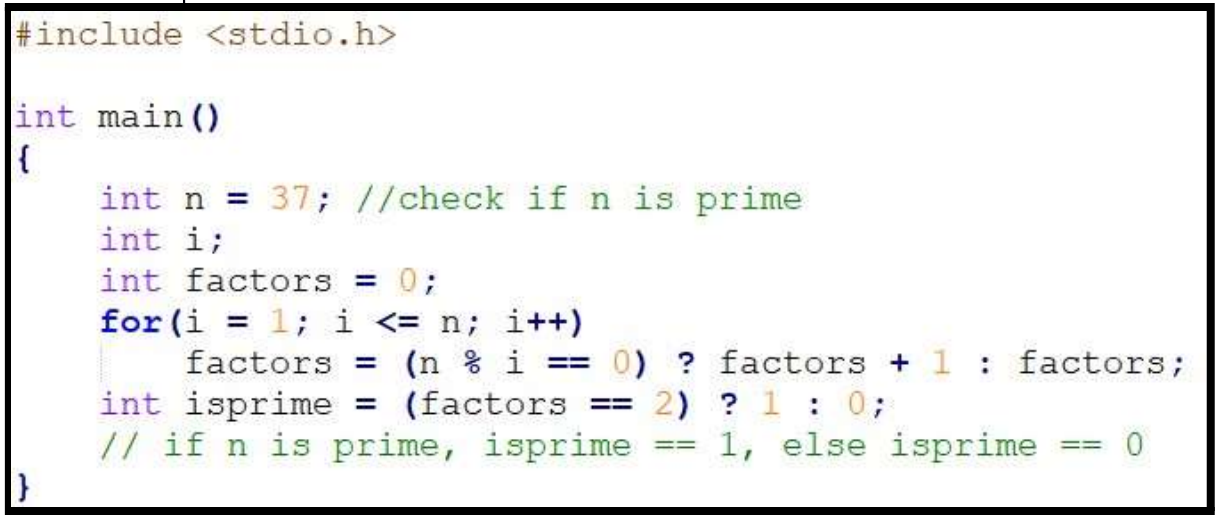
```
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Semantic Analysis$ gcc y.tab.c -ll
```



```
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Semantic Analysis$ ./a.out test2.c

Parsing Completed

---------------------------------------------------Symbol Table---------------------------------------------------
------------------------------------------------------------------------------------------------------------------

SNo    Identifier    Scope        Value        Type                Parameter type(for functions)
------------------------------------------------------------------------------------------------------------------

1      a             1            10           INT
2      b             1            9            INT
3      main          0            0            FUNCTION - INT
------------------------------------------------------------------------------------------------------------------

saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/Semantic Analysis$
```

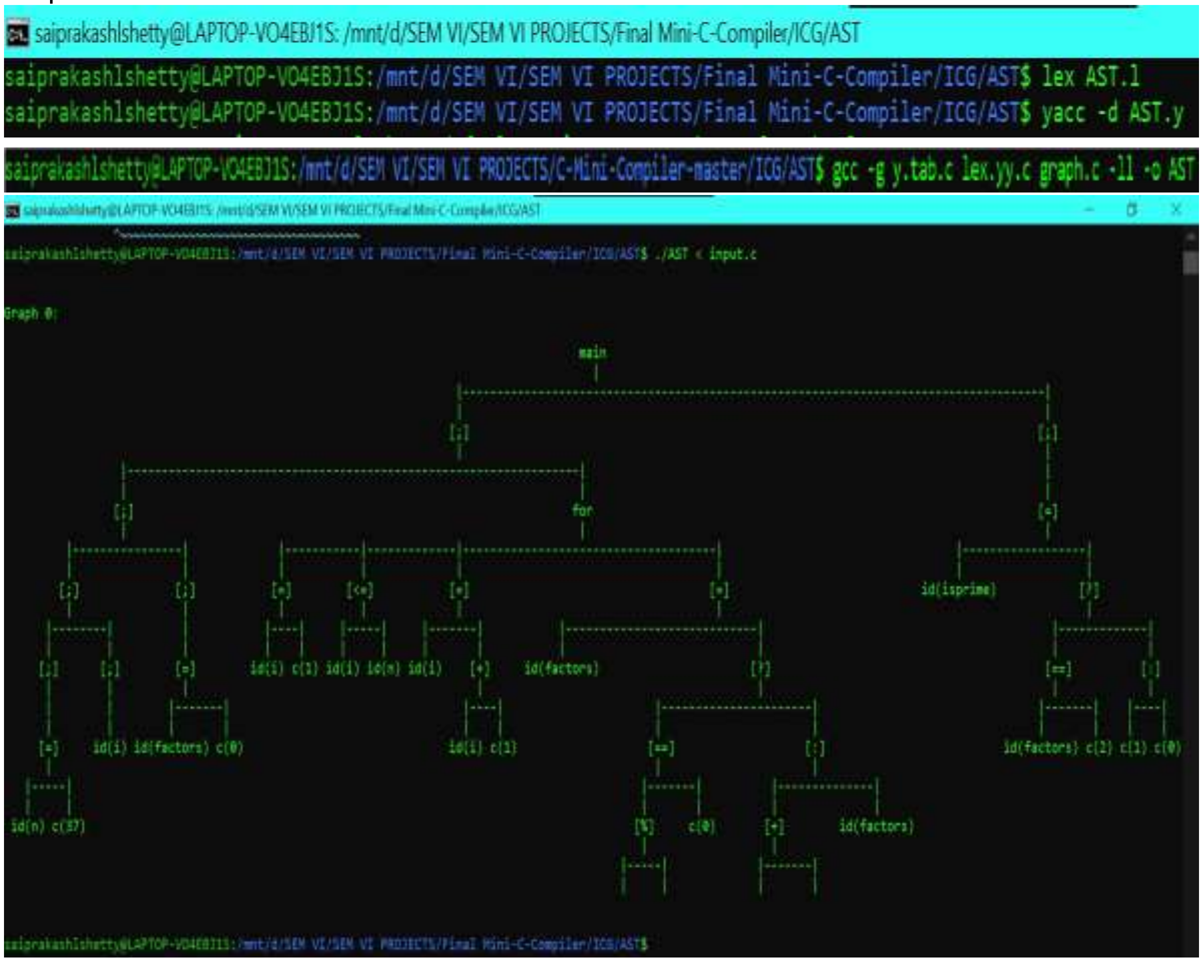
7.4 Abstract Syntax Tree

Test Case : input.c



```c
#include <stdio.h>

int main()
{
    int n = 37; //check if n is prime
    int i;
    int factors = 0;
    for(i = 1; i <= n; i++)
        factors = (n % i == 0) ? factors + 1 : factors;
    int isprime = (factors == 2) ? 1 : 0;
    // if n is prime, isprime == 1, else isprime == 0
}
```

Output :



```
saiprakashlshetty@LAPTOP-VO4EBJ1S: /mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/ICG/AST
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/ICG/AST$ lex AST.l
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/ICG/AST$ yacc -d AST.y
```

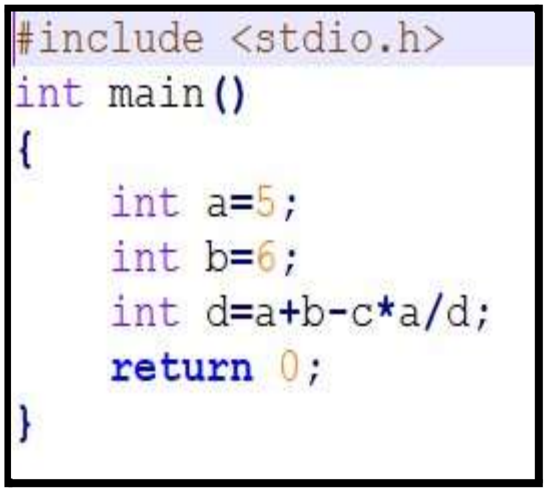
```
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/C-Mini-Compiler-master/ICG/AST$ gcc -g y.tab.c lex.yy.c graph.c -ll -o AST
```

```
saiprakashlshetty@LAPTOP-VO4EBJ1S: /mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/ICG/AST
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/ICG/AST$ ./AST < input.c

Graph 0:

                                                                            main
                                                           [;]                                                                         [;]
               [;]                                                        for                                                         [=]
        [;]            [;]          [=]       [<=]         [+]                              [=]                        id(isprime)          [?]
      [;]     [;]     [=]     id(i) c(1) id(i) id(n) id(i) [+]   id(factors)                    [?]                                   [==]        [:]
      [=]  id(i) id(factors) c(0)                        id(i) c(1)                  [==]                [:]                     id(factors) c(2) c(1) c(0)
 id(n) c(37)                                                                     [%]   c(0)      [+]      id(factors)

saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/ICG/AST$
```

7.5 Intermediate Code Generation :

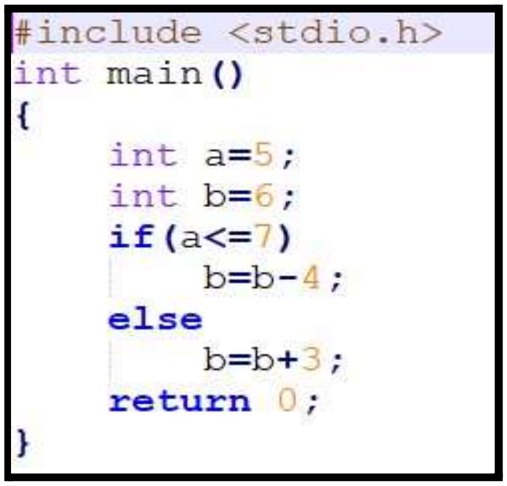
Test Cases :

test1.c :



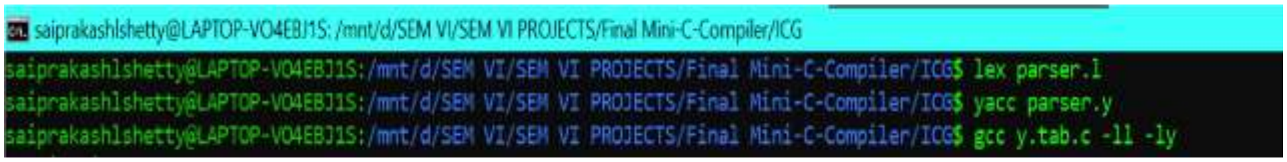
```
#include <stdio.h>
int main()
{
    int a=5;
    int b=6;
    int d=a+b-c*a/d;
    return 0;
}
```

test2.c :

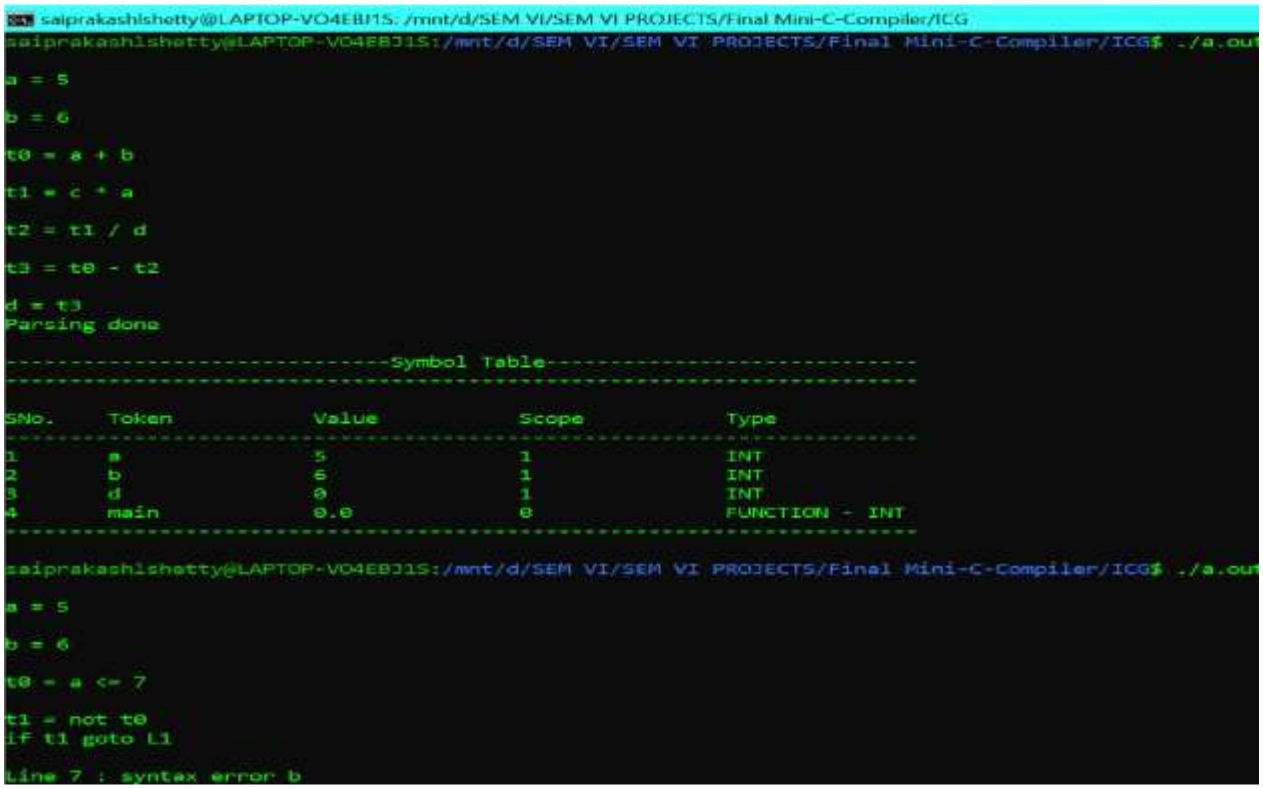


```
#include <stdio.h>
int main()
{
    int a=5;
    int b=6;
    if(a<=7)
        b=b-4;
    else
        b=b+3;
    return 0;
}
```

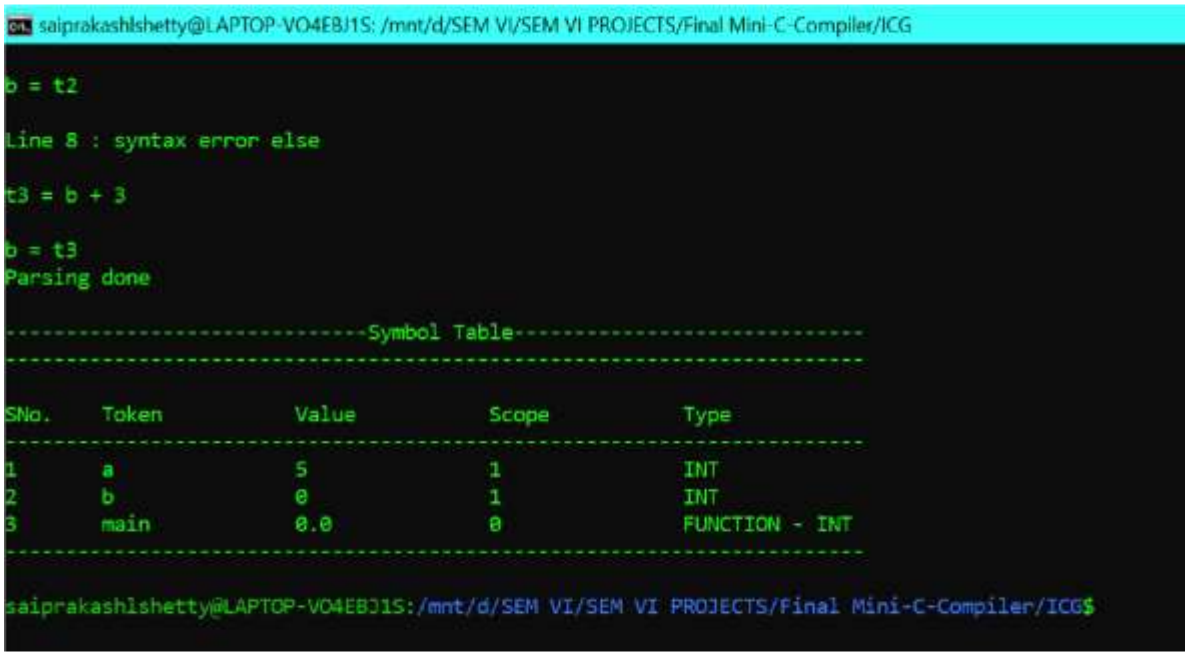
Output :


saiprakashlshetty@LAPTOP-VO4EBJ1S: /mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/ICG

```
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/ICG$ lex parser.l
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/ICG$ yacc parser.y
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/ICG$ gcc y.tab.c -ll -ly
```


saiprakashlshetty@LAPTOP-VO4EBJ1S: /mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/ICG

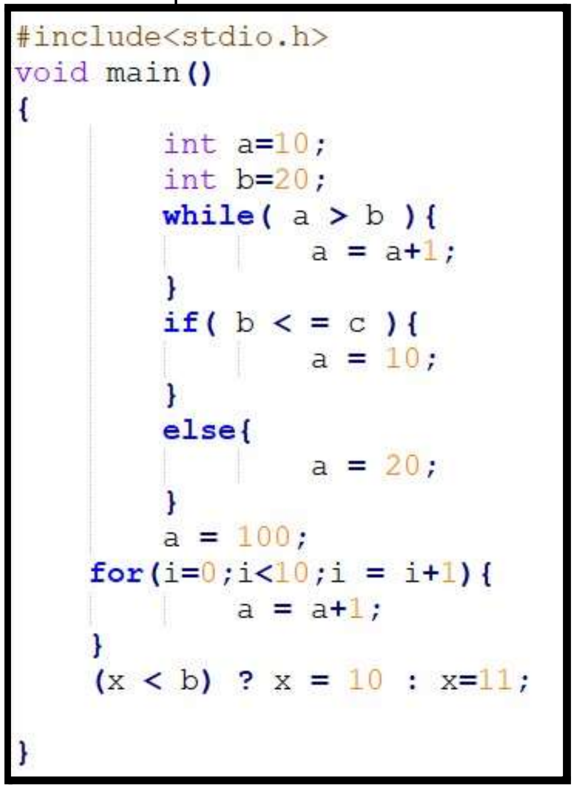
```
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/ICG$ ./a.out

a = 5

b = 6

t0 = a + b

t1 = c * a

t2 = t1 / d

t3 = t0 - t2

d = t3
Parsing done

------------------------------Symbol Table-----------------------------
-----------------------------------------------------------------------

SNo.    Token           Value           Scope           Type
-----------------------------------------------------------------------
1       a               5               1               INT
2       b               6               1               INT
3       d               0               1               INT
4       main            0.0             0               FUNCTION - INT
-----------------------------------------------------------------------

saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/ICG$ ./a.out

a = 5

b = 6

t0 = a <= 7

t1 = not t0
if t1 goto L1

Line 7 : syntax error b
```


saiprakashlshetty@LAPTOP-VO4EBJ1S: /mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/ICG

```
b = t2

Line 8 : syntax error else

t3 = b + 3

b = t3
Parsing done

------------------------------Symbol Table-----------------------------
-----------------------------------------------------------------------

SNo.    Token           Value           Scope           Type
-----------------------------------------------------------------------
1       a               5               1               INT
2       b               0               1               INT
3       main            0.0             0               FUNCTION - INT
-----------------------------------------------------------------------

saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/ICG$
```

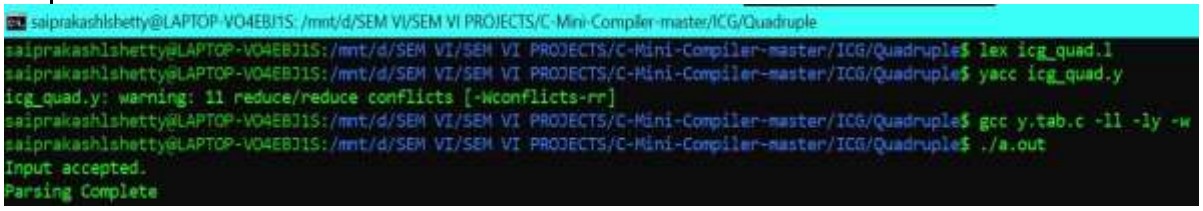
7.6 ICG in Quadruple format

Test Case :input.c

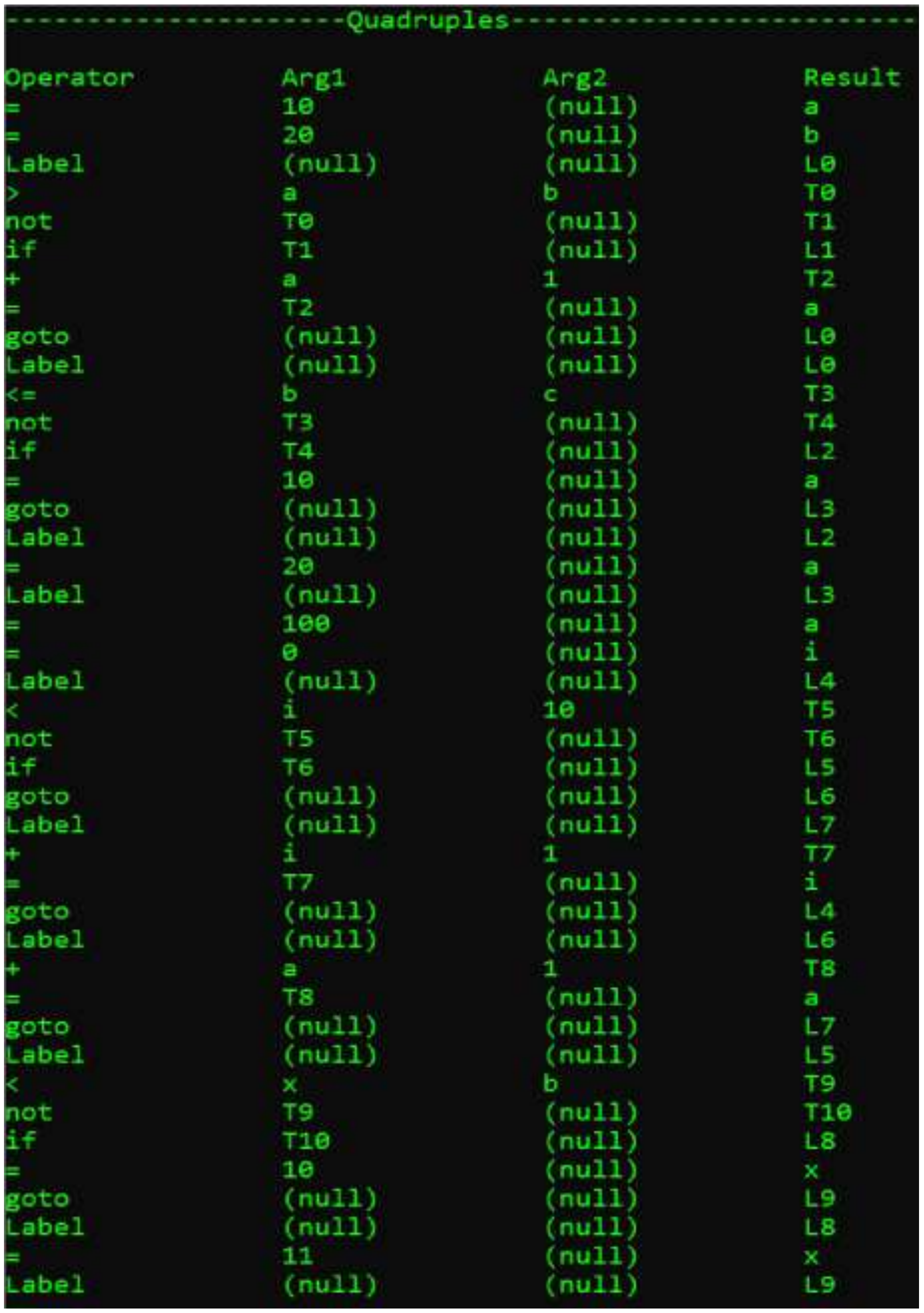


```c
#include<stdio.h>
void main()
{
        int a=10;
        int b=20;
        while( a > b ){
                a = a+1;
        }
        if( b < = c ){
                a = 10;
        }
        else{
                a = 20;
        }
        a = 100;
    for(i=0;i<10;i = i+1){
            a = a+1;
    }
    (x < b) ? x = 10 : x=11;

}
```

Output :



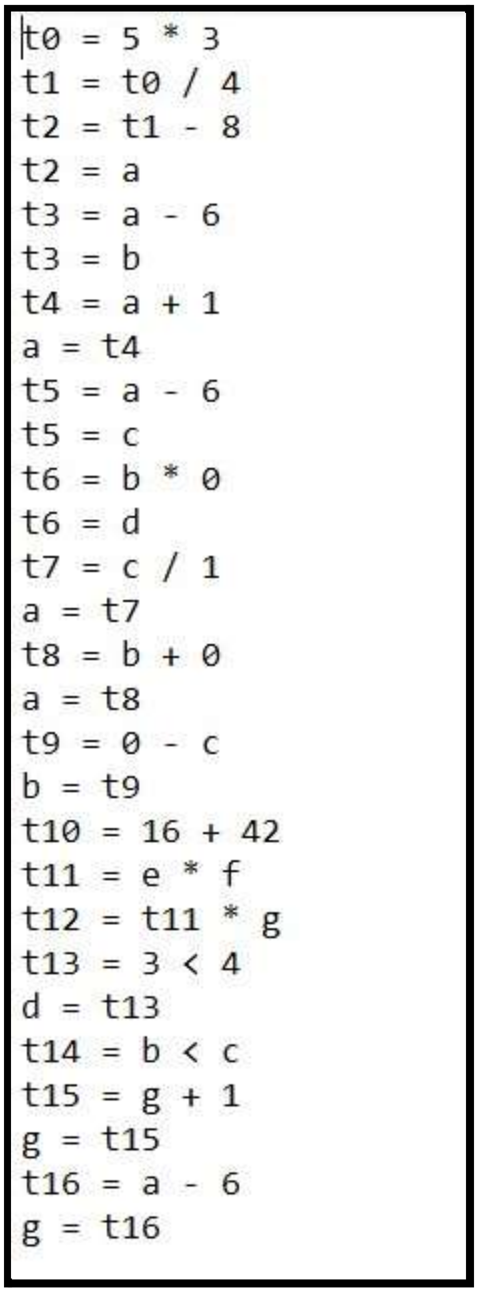
```
saiprakashlshetty@LAPTOP-VO4EBJ1S: /mnt/d/SEM VI/SEM VI PROJECTS/C-Mini-Compiler-master/ICG/Quadruple
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/C-Mini-Compiler-master/ICG/Quadruple$ lex icg_quad.l
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/C-Mini-Compiler-master/ICG/Quadruple$ yacc icg_quad.y
icg_quad.y: warning: 11 reduce/reduce conflicts [-Wconflicts-rr]
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/C-Mini-Compiler-master/ICG/Quadruple$ gcc y.tab.c -ll -ly -w
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/C-Mini-Compiler-master/ICG/Quadruple$ ./a.out
Input accepted.
Parsing Complete
```



```
---------------------Quadruples-------------------------

Operator         Arg1             Arg2             Result
=                10               (null)           a
=                20               (null)           b
Label            (null)           (null)           L0
>                a                b                T0
not              T0               (null)           T1
if               T1               (null)           L1
+                a                1                T2
=                T2               (null)           a
goto             (null)           (null)           L0
Label            (null)           (null)           L0
<=               b                c                T3
not              T3               (null)           T4
if               T4               (null)           L2
=                10               (null)           a
goto             (null)           (null)           L3
Label            (null)           (null)           L2
=                20               (null)           a
Label            (null)           (null)           L3
=                100              (null)           a
=                0                (null)           i
Label            (null)           (null)           L4
<                i                10               T5
not              T5               (null)           T6
if               T6               (null)           L5
goto             (null)           (null)           L6
Label            (null)           (null)           L7
+                i                1                T7
=                T7               (null)           i
goto             (null)           (null)           L4
Label            (null)           (null)           L6
+                a                1                T8
=                T8               (null)           a
goto             (null)           (null)           L7
Label            (null)           (null)           L5
<                x                b                T9
not              T9               (null)           T10
if               T10              (null)           L8
=                10               (null)           x
goto             (null)           (null)           L9
Label            (null)           (null)           L8
=                11               (null)           x
Label            (null)           (null)           L9
```

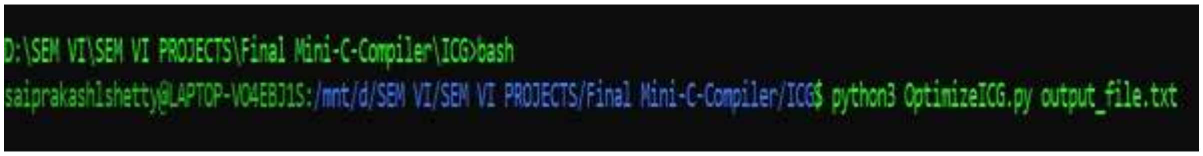
7.7 Code Optimization

Test case : output_file.txt

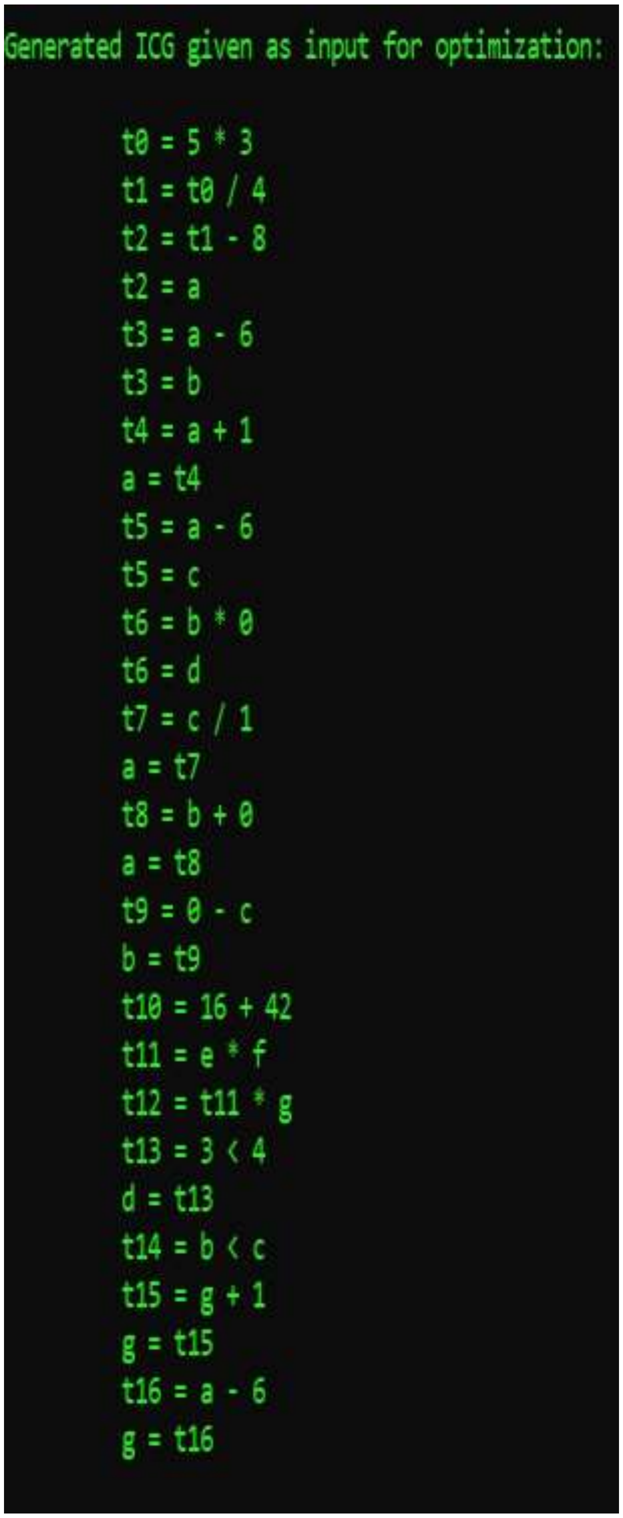


```
t0 = 5 * 3
t1 = t0 / 4
t2 = t1 - 8
t2 = a
t3 = a - 6
t3 = b
t4 = a + 1
a = t4
t5 = a - 6
t5 = c
t6 = b * 0
t6 = d
t7 = c / 1
a = t7
t8 = b + 0
a = t8
t9 = 0 - c
b = t9
t10 = 16 + 42
t11 = e * f
t12 = t11 * g
t13 = 3 < 4
d = t13
t14 = b < c
t15 = g + 1
g = t15
t16 = a - 6
g = t16
```

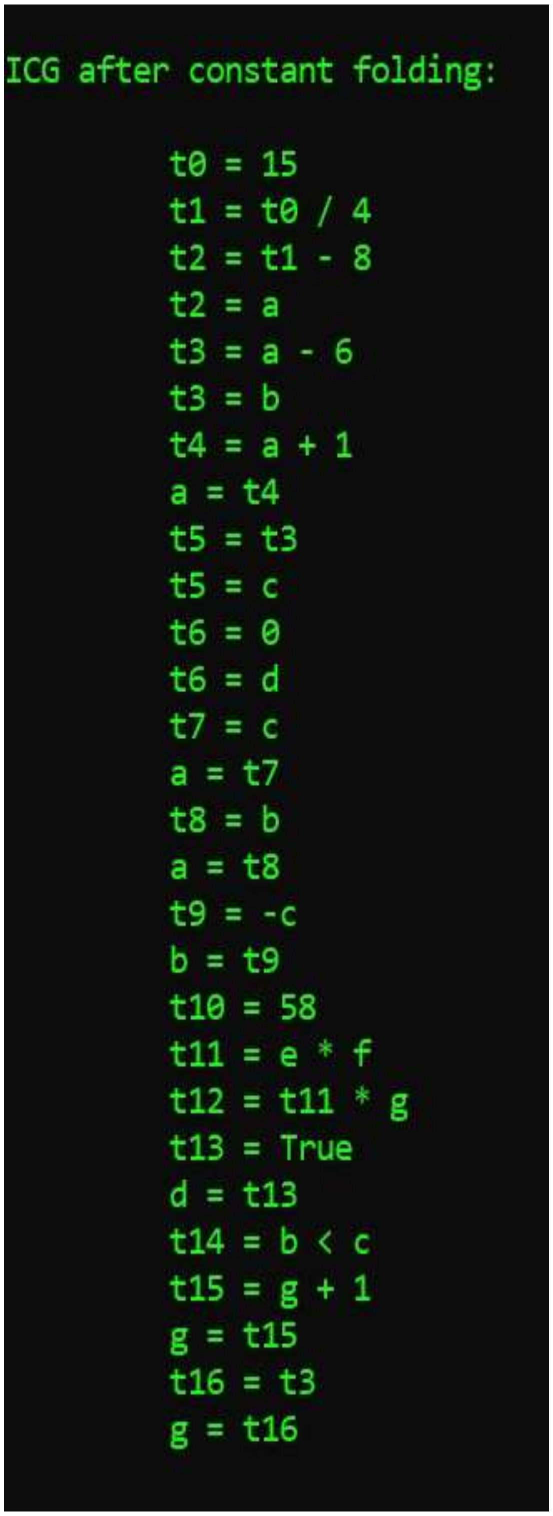
Output :



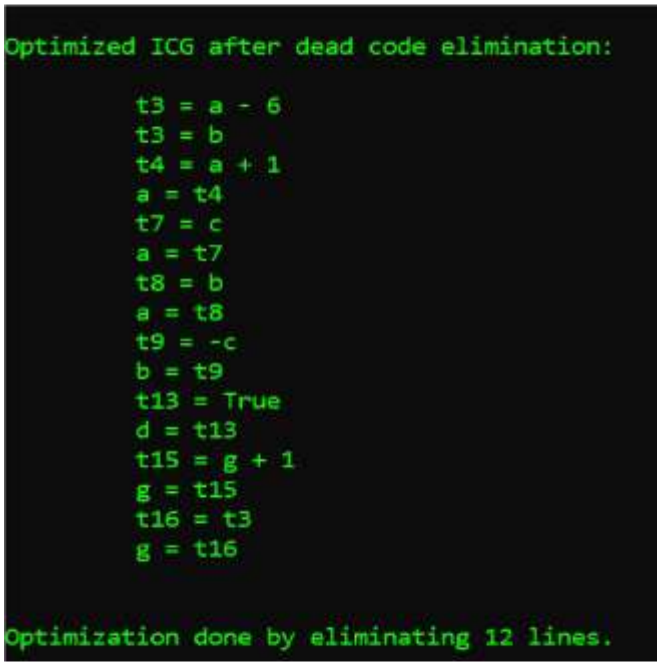
```
D:\SEM VI\SEM VI PROJECTS\Final Mini-C-Compiler\ICG>bash
saiprakashlshetty@LAPTOP-VO4EBJ1S:/mnt/d/SEM VI/SEM VI PROJECTS/Final Mini-C-Compiler/ICG$ python3 OptimizeICG.py output_file.txt
```



```
Generated ICG given as input for optimization:

        t0 = 5 * 3
        t1 = t0 / 4
        t2 = t1 - 8
        t2 = a
        t3 = a - 6
        t3 = b
        t4 = a + 1
        a = t4
        t5 = a - 6
        t5 = c
        t6 = b * 0
        t6 = d
        t7 = c / 1
        a = t7
        t8 = b + 0
        a = t8
        t9 = 0 - c
        b = t9
        t10 = 16 + 42
        t11 = e * f
        t12 = t11 * g
        t13 = 3 < 4
        d = t13
        t14 = b < c
        t15 = g + 1
        g = t15
        t16 = a - 6
        g = t16
```



```
ICG after constant folding:

        t0 = 15
        t1 = t0 / 4
        t2 = t1 - 8
        t2 = a
        t3 = a - 6
        t3 = b
        t4 = a + 1
        a = t4
        t5 = t3
        t5 = c
        t6 = 0
        t6 = d
        t7 = c
        a = t7
        t8 = b
        a = t8
        t9 = -c
        b = t9
        t10 = 58
        t11 = e * f
        t12 = t11 * g
        t13 = True
        d = t13
        t14 = b < c
        t15 = g + 1
        g = t15
        t16 = t3
        g = t16
```



```
Optimized ICG after dead code elimination:

        t3 = a - 6
        t3 = b
        t4 = a + 1
        a = t4
        t7 = c
        a = t7
        t8 = b
        a = t8
        t9 = -c
        b = t9
        t13 = True
        d = t13
        t15 = g + 1
        g = t15
        t16 = t3
        g = t16


Optimization done by eliminating 12 lines.
```

8. CONCLUSION

The lexical analyzer, syntax analyzer and the semantic analyzer for a subset of C language, which include selection statements, compound statements, iteration statements (for, while and do-while) and user defined functions are generated. It is important to define unambiguous grammar in the syntax analysis phase.

The semantic analyzer performs type checking, reports various errors such as undeclared variable, type mismatch, errors in function call (number and datatypes of parameters mismatch) and errors in array indexing.

The syntax analyser for the C language by writing two scripts, one that acts as a lexical analyzer (lexer) and outputs a stream of tokens, and the other one that acts as a parser. The Syntax analyzer generates the various statements and expressions required based on the context free grammar defined. Parse trees for various statements and expressions are generated by the syntax analyzer.

The Intermediate Code Generation phase involves the execution of lex and yacc codes. Once the parsing is complete, if errors are encountered then the errors are displayed along with the line numbers. Along with this, the updated symbol table is displayed.

9. FUTURE ENHANCEMENTS

We have implemented the parser and semantic analyzer for only a subset of C language. The future work may include defining the grammar and specifying the 31 semantics for switch statements, predefined functions (like string functions, file read and write functions), jump statements and enumerations.

In future for 3 address intermediate code generators, we can extend it to support and generate three-address code for pointers, structures and functions

REFERENCES/BIBLIOGRAPHY

1. *Compilers – Principles, Techniques and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman*
2. *https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/*
3. *http://web.cs.wpi.edu/~kal/courses/compilers*
4. *https://www.tutorialspoint.com/compiler_design/compiler_design_intermediate_code_generations.htm*