Lexical Analyzer for the C Language

National Institute of Technology Karnataka, Surathkal

Date: JAN 30, 2020

Submitted To:

Dr. P. Santhi Thilagam

Dept. of Computer Science, NITK Surathkal

Group Members:

1. **Adarsh Kumar (17CO204)**
2. **Saurabh Singhal (17CO240)**
3. **Rahul Kumar (17CO232)**

Abstract:

A compiler is computer software that transforms computer code written in one programming language (the source language) into another programming language (the target language). The name compileris primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language, object code, or machine code) to create an executable program.

Phases of Compiler

Conceptually, a compiler operates in phases, each of which transforms the source program from one representation to another. The phases are as below :

The phases are as below :

Analysis

1. Lexical Analysis:
2. Parsing:
3. Semantic Analysis:
4. Intermediate Code Generation:

Synthesis

1. Code Optimization:
2. Code Generation:

Objectives:

This project aims to undertake a sequence of experiments to design and implement various phases of a compiler for the C programming language. Following constructs will be handled by the mini-compiler :

1. Data Types: int, char data types with all its sub-types. Syntax :

```
int a=3;
```

2. Comments: Single line and multiline comments,

3. Keywords:

```
char, else, for, if, int, long, return, short, signed, struct, unsigned, void,
while, main
```

4. Identification of valid identifiers used in the language,

5. Looping Constructs: It will support nested for and while loops. Syntax:

```
int i;
for(i=0;i<n;i++){ } int x; while(x<10){ ... x++}
```

6. Conditional Constructs: if...else-if...else statements,

7. Operators: ADD(+), MULTIPLY(*), DIVIDE(/), MODULO(%), AND(&), OR(|)

8. Delimiters: SEMICOLON(;), COMMA(,)

9. Structure construct of the language, Syntax:

```
struct pair{ int a; int b};
```

10. Function construct of the language, Syntax:

```
int func(int x)
```

11. Support of nested conditional statement,

12. Support for a 1-Dimensional array. Syntax :

```
char s[20];
```

Contents: **Page No.**

Introduction

Lexical Analysis

The Lexical Analyzer is the first phase of the Analysis (front end) stage of a compiler. In layman's terms, the Lexical Analyzer (or Scanner) scans through the input source program character by character, and identifies 'Lexemes' and categorizes them into 'Tokens'. These 'tokens' are represented as a symbol table, and is given as input to the Parser (second phase of the front end of a compiler).

Tokens

Tokens are essentially just a group of characters which have some meaning or relation when put together.

The Lexical Analyzer detects these tokens with the help of 'Regular Expressions'. While writing the Lexical Analyzer, we have to specify rules for each Token type using Regular Expression. These rules are used to check whether a certain group of characters fall under a given token category or not.

An example, in this case, would be an 'Identifier' token. We specify the rules for an identifier as follows: Any string of characters, that start with an _ or an alphabet, followed by any number of _'s, alphabets or numbers. The regular expression for Identifiers is `{S}({S}|{D})*` where S is `[a-zA-z_]` and D is `[0-9].`

Lexemes

Lexemes are instances of Tokens. An example would be ' `long int` ', which is a Lexeme of 'Keyword' Token.

Symbol Table

A symbol table is generated in the Lexical Analyzer stage, which is basically a table with the columns '**Symbol**', '**Type**' and '**Token ID**'. The symbol is the Lexime itself, the 'Type' is the token category and the 'Token ID' is a unique ID given to a token, which is used in the parser stage. There are no duplicate entries in the symbol table. Each symbol is recorded only once, even if there are multiple instances.

A Lexical Analyzer is internally implemented based on the concept of FSMs (Finite State Machines). A DFA (Deterministic Finite State Automata) is internally built for each Token based on the Regular Express provided. This is used to identify Lexemes and categorize them into Tokens.

Flex Script

The script written by us is a program 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.

The structure of our flex script is intentionally similar to that of a yacc file; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

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.

C Program

This section describes the input C program which is fed to the flex script in order to generate the lex file after taking all the rules mentioned into account. Finally, a file called lex.yy.c is generated, which when executed recognizes the tokens present in the C program which was given as an input.

The script also has an option to take standard input instead of taking input from a file.

```
/* Declaration section */
%option noyywrap
%option yylineno

/* Declarations*/
%{
   #include "lib/header.h"
   #define MAX_NODES 1000
   symbol_node *symbolTable[100];
   symbol_node *constantTable[100];
   symbol_node *headerTable[100];
   void printToken(char *info, char *token, int line_number);
%}

/* Definitions */
letter              [a-zA-Z]
digit               [0-9]
escape_sequences    0|a|b|f|n|r|t|v|"\\"|"\""|"\'"
keyword
signed|while|for|continue|break|if|else|return|struct|int|char|void|m
ain|float|double|short|long|unsigned
operator
"&"|"^"|\||\|\||"="|">"|"<"|">="|"<="|"=="|"!="|"+"|"-"|"++"|"--"|"!"
|"~"|"*"|"/"|"%"|">>"|"<<"|
alphanum            {letter}|{digit}
function            ((_|{letter})({alphanum}|_)*)/[ ]*[(]

/* Rules */
%%
\n                                             {}
" "                                            {}
"["                                            { printToken("LEFT
BRACKET", yytext, yylineno); }
"]"                                            { printToken("RIGHT
BRACKET", yytext, yylineno); }
"{"                                            { printToken("LEFT
BRACE", yytext, yylineno); }
```

```
"}"                                          { printToken("RIGHT
BRACE", yytext, yylineno); }
","                                          { printToken("COMMA",
yytext, yylineno); }
";"                                          {
printToken("SEMICOLON", yytext, yylineno); }

"#include"[ ]*"<"{letter}({alphanum})*".h>"   { printToken("HEADER",
yytext, yylineno);insert(headerTable, yytext, "Header" , yylineno);}

"#define"[ ]+(_|{letter})({alphanum})*[ ]*(.)+   {
printToken("PREPROCESSOR DIRECTIVE", yytext, yylineno); }

"//".*                                       { printToken("SINGLE
LINE COMMENT", yytext, yylineno); }

("/*")(([^*]*[*]+[^*/])*([^*]*[*]+[/]))      { printToken("MULTI LINE
COMMENT", yytext, yylineno); }

("/*")(([^*]*([*]+[^/])*)*)*                 { printToken(COL_RED
"ERROR: UNMATCHED COMMENT", yytext, yylineno); }

"("                                          { printToken("LEFT
PARENTHESIS", yytext, yylineno); }

")"                                          { printToken("RIGHT
PARENTHESIS", yytext, yylineno); }

("\"")[^\n\"]*("\"")                         { printToken("STRING",
yytext, yylineno); insert(constantTable, yytext, "String",
yylineno);}

("\"")[^\n\"]*                               { printToken(COL_RED
"ERROR: UNCLOSED STRING", yytext, yylineno); }

("\'")(("\\"({escape_sequences}))|.)("\'")   {
printToken("CHARACTER", yytext, yylineno); insert(constantTable,
yytext, "Character", yylineno);}
```

```
("\'")(((("\\")[^0abfnrtv\\\"\'][^\n\']*))|[^\n\''][^\n\'']+)("\'") {
printToken(COL_RED "ERROR: NOT A CHARACTER", yytext, yylineno); }

{keyword}/[ ]*[(]?                        { printToken("KEYWORD",
yytext, yylineno);}

#include[/"<][ ]*{letter}{letter}*\.h[/">]   { printToken("HEADER",
yytext, yylineno);}

{operator}                                { printToken("OPERATOR",
yytext, yylineno); }

{function}                                { printToken("FUNCTION",
yytext, yylineno); insert(symbolTable, yytext, "Function",
yylineno);}

(_|{letter})({alphanum}|_)*               {
printToken("IDENTIFIER", yytext, yylineno);insert(symbolTable,
yytext, "Identfier", yylineno);}

"-"?{digit}+                              { printToken("INTEGER",
yytext, yylineno);insert(constantTable, yytext, "Integer",
yylineno);}

"-"?{digit}+\.({digit}+)?                 { printToken("FLOATING
POINT", yytext, yylineno);insert(constantTable, yytext, "Floating
point", yylineno);}

%%

void printToken(char *info, char *token, int line_number){
printf(COL_YEL "%-30s\t\t%-30s\t\t%-30d\n" RESET, info, token,
line_number);}

int main()
{
   printf(COL_MAG "\n" DASHES RESET);
```

```
    printf(COL_CYN "\t\t\t\t\t\t\tLexical Analyser for C \n" RESET);
    printf(COL_MAG DASHES "\n" RESET);
    printf(COL_GRN "%-30s\t\t%-30s\t\t%-30s\n", "TOKEN TYPE", "TOKEN
VALUE", "TOKEN LINE NUMBER" RESET);
    yylex();
    print(symbolTable, "Symbol Table");
    print(constantTable, "Constant Table");
    print(headerTable, "Header Table");
    return 0;
}
```

Explanation:

In the definition section of the program, all necessary header files were included. Apart from that structure declaration for both the symbol table and constant table were made. In order to convert a string of the source program into a particular integer value a hash function was written that takes a string as input and converts it into a particular integer value. Standard table operations like look-up and insert were also written. Linear Probing hashing technique was used to implement the symbol table i.e. if there is a collision, then after the point of collision, the table is searched linearly in order to find an empty slot. Functions to print the symbol table and constant table was also written.

Rules section:

In this section rules related to the specification of the C language were written in the form of valid regular expressions. E.g. for a valid C identifier the regex written was `[A-Za-z_][A-Za-z_0-9]*` which means that a valid identifier need to start with an alphabet or underscore followed by 0 or more occurrence of alphabets, numbers or underscores. In order to resolve conflicts we used lookahead method of scanner by which a scanner decides whether an expression is valid token or not by looking at its adjacent character. E.g. in order to differentiate between comments and division operator lookahead characters of a valid operator were also given in the regular expression to resolve a conflict. If none of the patterns matched with the input, we said it is a lexical error as it does not match with any valid pattern of the source language. Each character/pattern along with its token class was also printed.

C code section:

In this section both the tables (symbol and constant) were initialised to 0 and `yylex( )` function was called to run the program on the given input file. After that, both the symbol table and constant table were printed in order to show the result.

The flex script recognises the following classes of tokens from the input:

- Pre-processor instructions
  - Statements processed : `#include<stdio.h>, #define var1 var2`
  - Token generated : Preprocessor Directive
- Errors in pre-processor instructions
  - Statements processed : `#include<stdio.h>, #include<stdio.?`
  - Token generated : Error with line number
- Single-line comments
  - Statements processed : `//...........`
  - Token generated : Single Line Comments
- Multi-line comments
  - Statements processed : `/*...........*/, /*.../*...*/`
  - Token generated : Multi Line Comment
- Errors for unmatched comments
  - Statements processed : `/*..........`
  - Token generated : Error with line number
- Errors for nested comments
  - Statements processed : `/*....../*....*/....*/`
  - Token generated : Error with line number
- Parentheses (all types)
  - Statements processed : `(..), {..}, [..]`
  - Token generated : Parenthesis

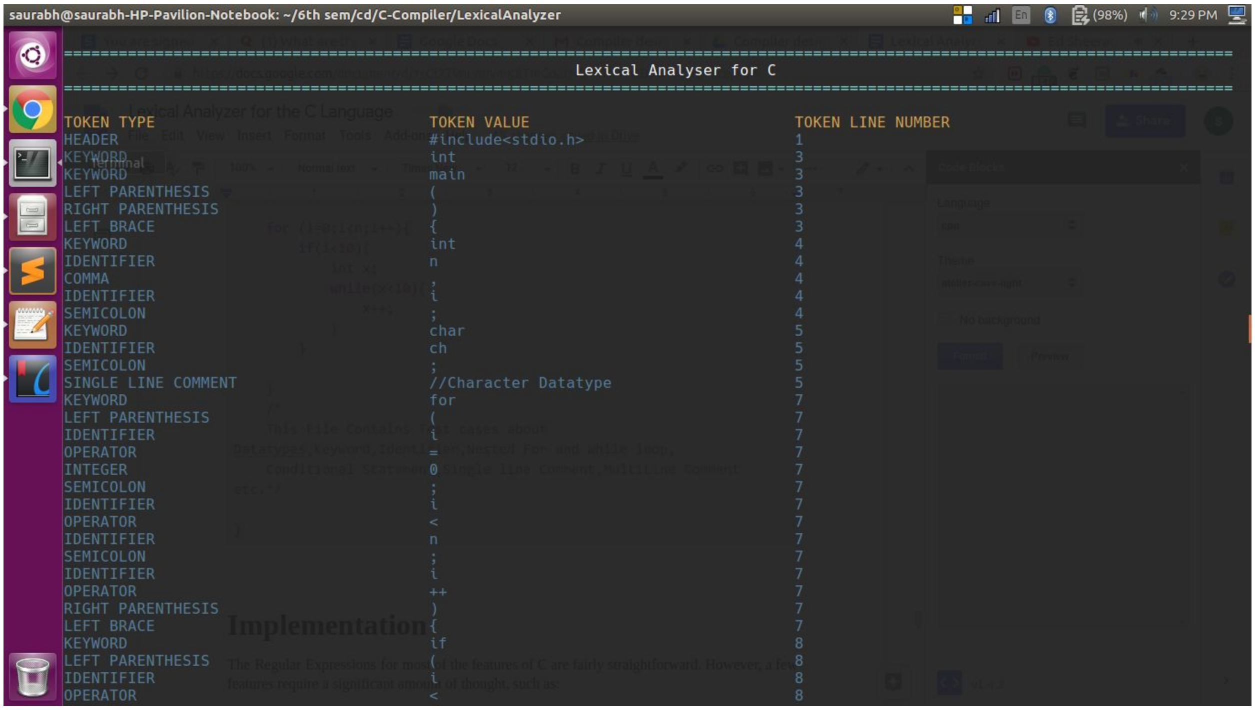
- Operators
- Literals (integer, float, string)
    - Statements processed : `int, float, char`
    - Tokens generated : Keywords
- Errors for unclean integers and floating point numbers
    - Statements processed : `123rf`
    - Tokens generated : Error
- Errors for incomplete strings
    - Statements processed : `char a[]= "abcd`
    - Tokens generated : Error Incomplete string and line number
- Keywords
    - Statements processed : `if, else, void, while, do, int, float, break and so on.`
    - Tokens generated : Keyword
- Identifiers
    - Statements processed : `a, abc, a_b, a12b4`
    - Tokens generated : Identifier
- Errors for any invalid character used that is not in C character set.
    - Keywords accounted for:

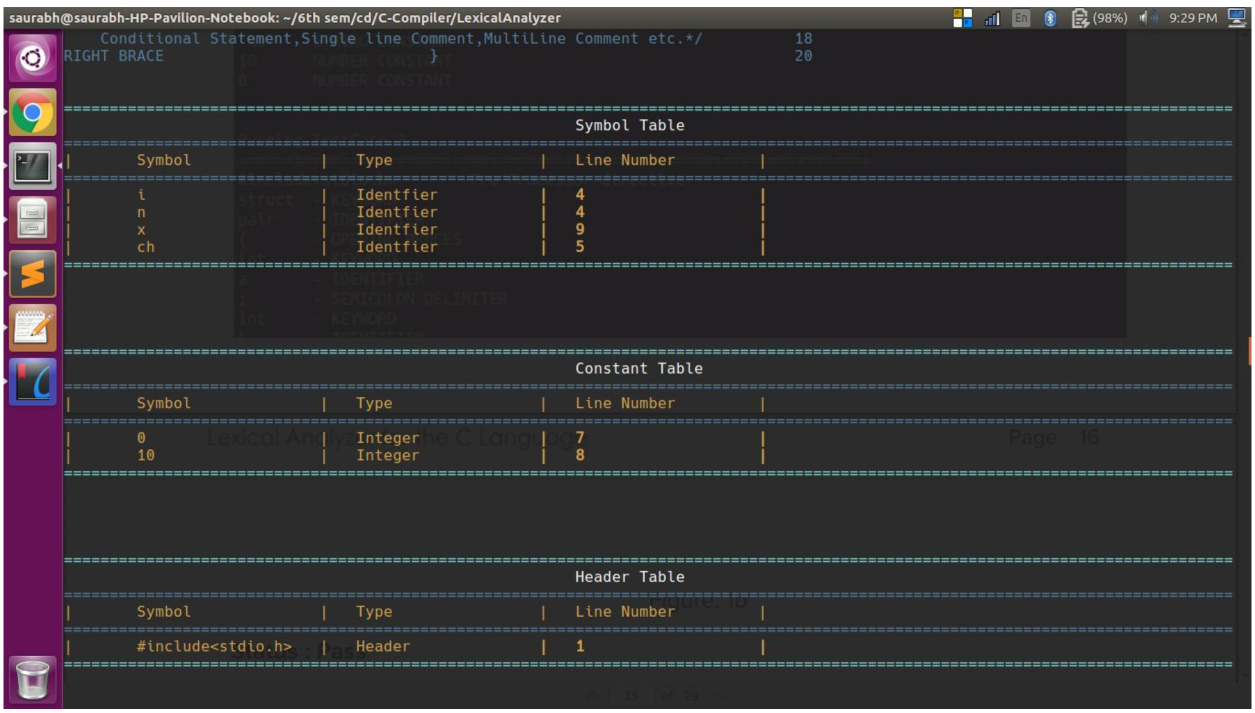
```
auto, break, case, char, const, continue, default, do, double,
else,enum, extern, float, for, goto, if, int, long, register,
return,short, signed, sizeof, static, struct, switch, typedef,
union,unsigned, void, volatile, while, main
```

Test Cases

Test Case 1: Error Free Code

```
#include<stdio.h>

int main(){
    int n,i;
    char ch;//Character Datatype

    for (i=0;i<n;i++){
        if(i<10){
            int x;
            while(x<10){
                x++;
            }
        }

    }
    /*
    This File Contains Test cases about
Datatypes,Keyword,Identifier,Nested For and while loop,
    Conditional Statement,Single line Comment,MultiLine Comment
etc.*/

}
```

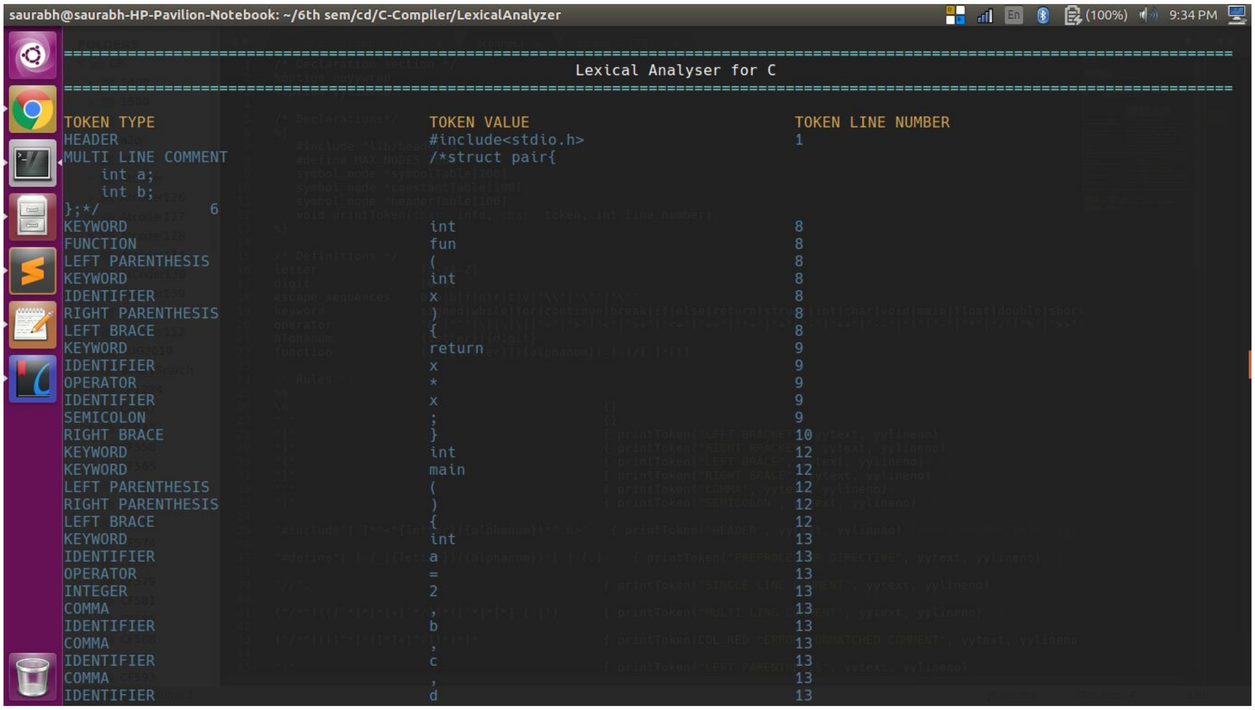

saurabh@saurabh-HP-Pavilion-Notebook: ~/6th sem/cd/C-Compiler/LexicalAnalyzer
Lexical Analyser for C
TOKEN TYPE TOKEN VALUE TOKEN LINE NUMBER
HEADER #include<stdio.h> 1
KEYWORD int 3
KEYWORD main 3
LEFT PARENTHESIS (3
RIGHT PARENTHESIS) 3
LEFT BRACE { 3
KEYWORD int 4
IDENTIFIER n 4
COMMA , 4
IDENTIFIER i 4
SEMICOLON ; 4
KEYWORD char 5
IDENTIFIER ch 5
SEMICOLON ; 5
SINGLE LINE COMMENT //Character Datatype 5
KEYWORD for 7
LEFT PARENTHESIS (7
IDENTIFIER i 7
OPERATOR = 7
INTEGER 0 7
SEMICOLON ; 7
IDENTIFIER i 7
OPERATOR < 7
IDENTIFIER n 7
SEMICOLON ; 7
IDENTIFIER i 7
OPERATOR ++ 7
RIGHT PARENTHESIS) 7
LEFT BRACE { 7
KEYWORD if 8
LEFT PARENTHESIS (8
IDENTIFIER i 8
OPERATOR < 8

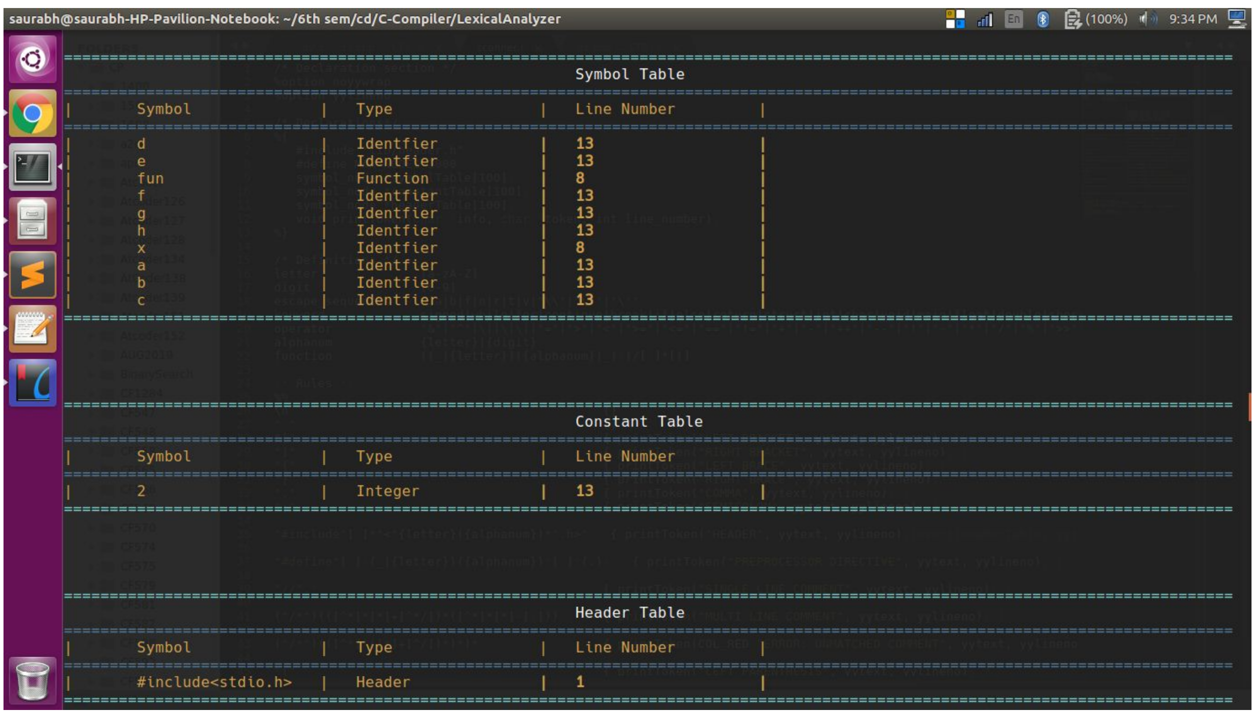

saurabh@saurabh-HP-Pavilion-Notebook: ~/6th sem/cd/C-Compiler/LexicalAnalyzer
Conditional Statement,Single line Comment,MultiLine Comment etc.*/ 18
RIGHT BRACE } 20
Symbol Table
Symbol	Type	Line Number
i	Identfier	4
n	Identfier	4
x	Identfier	9
ch	Identfier	5
Constant Table
Symbol	Type	Line Number
0	Integer	7
10	Integer	8
Header Table
Symbol	Type	Line Number
#include<stdio.h>	Header	1

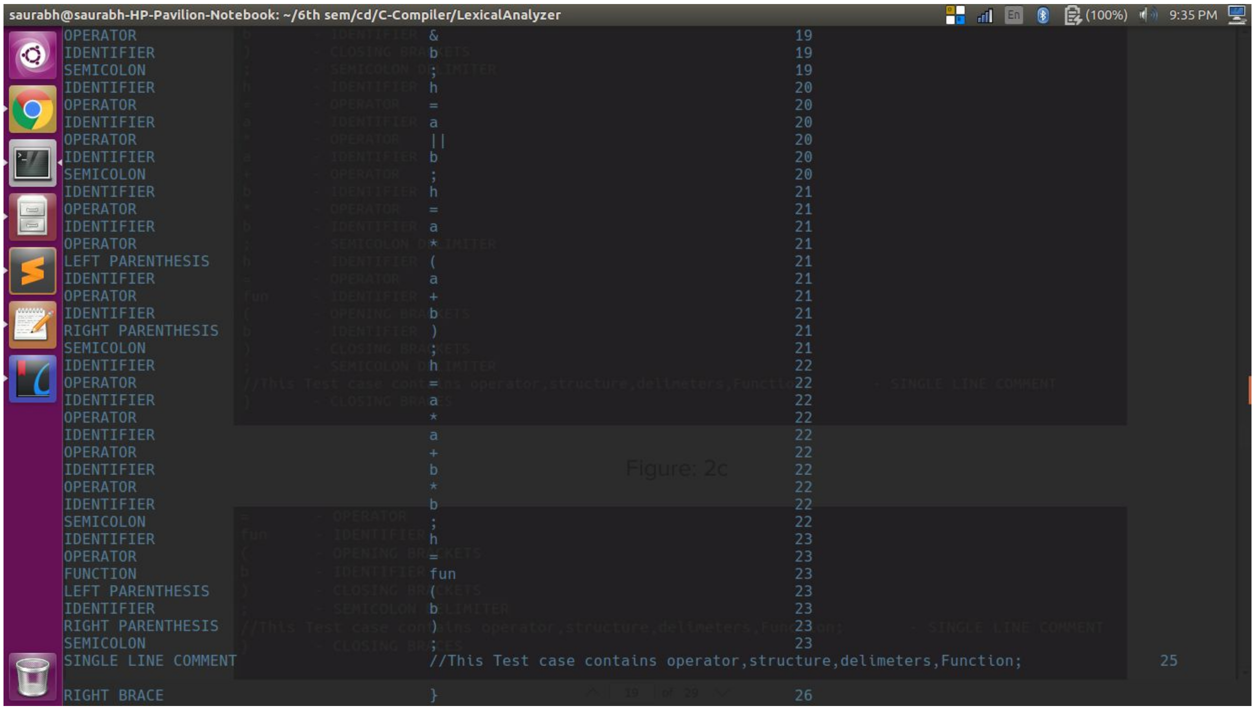
Status : PASS

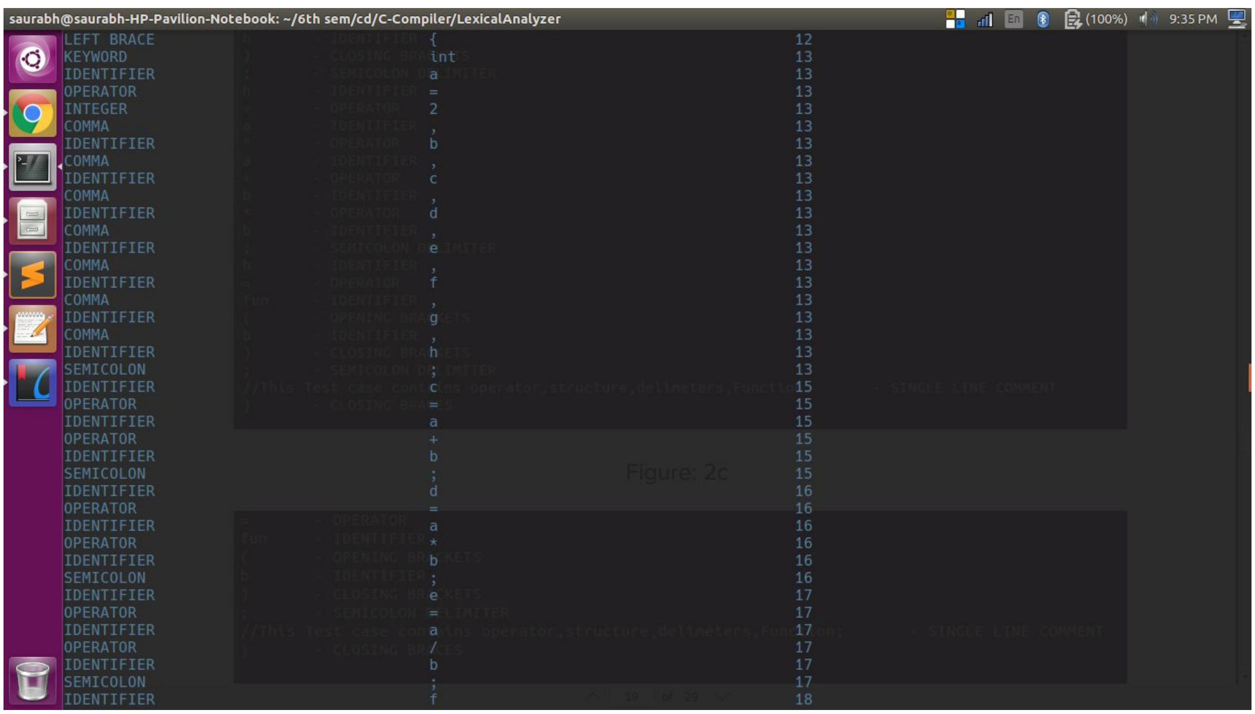
Test Case 2:

```
#include<stdio.h>

/*struct pair{
    int a;
    int b;
};*/

int fun(int x){
    return x*x;
}

int main(){
    int a=2,b,c,d,e,f,g,h;

    c=a+b;
    d=a*b;
    e=a/b;
    f=a%b;
    g=a&&b;
    h=a||b;
    h=a*(a+b);
    h=a*a+b*b;
    h=fun(b);
    //This Test case contains operator,structure,delimeters,Function;
}
```


saurabh@saurabh-HP-Pavilion-Notebook: ~/6th sem/cd/C-Compiler/LexicalAnalyzer
9:34 PM
Lexical Analyser for C
TOKEN TYPE
TOKEN VALUE
TOKEN LINE NUMBER
HEADER #include<stdio.h> 1
MULTI LINE COMMENT /*struct pair{
int a;
int b;
};*/ 6
KEYWORD int 8
FUNCTION fun 8
LEFT PARENTHESIS (8
KEYWORD int 8
IDENTIFIER x 8
RIGHT PARENTHESIS) 8
LEFT BRACE { 8
KEYWORD return 9
IDENTIFIER x 9
OPERATOR * 9
IDENTIFIER x 9
SEMICOLON ; 9
RIGHT BRACE } 10
KEYWORD int 12
KEYWORD main 12
LEFT PARENTHESIS (12
RIGHT PARENTHESIS) 12
LEFT BRACE { 12
KEYWORD int 13
IDENTIFIER a 13
OPERATOR = 13
INTEGER 2 13
COMMA , 13
IDENTIFIER b 13
COMMA , 13
IDENTIFIER c 13
COMMA , 13
IDENTIFIER d 13


saurabh@saurabh-HP-Pavilion-Notebook: ~/6th sem/cd/C-Compiler/LexicalAnalyzer
9:34 PM
Symbol Table
Symbol	Type	Line Number
d	Identfier	13
e	Identfier	13
fun	Function	8
f	Identfier	13
g	Identfier	13
h	Identfier	13
x	Identfier	8
a	Identfier	13
b	Identfier	13
c	Identfier	13
Constant Table
Symbol	Type	Line Number
2	Integer	13
Header Table
Symbol	Type	Line Number
#include<stdio.h>	Header	1

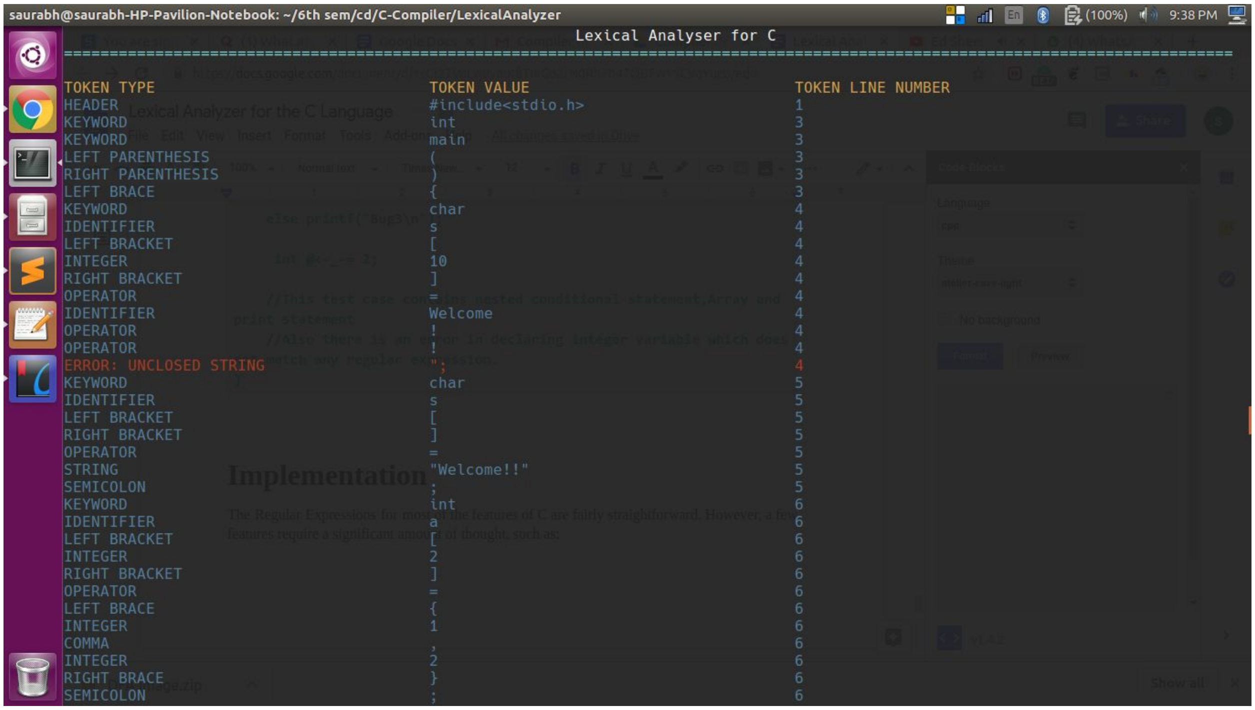

saurabh@saurabh-HP-Pavilion-Notebook: ~/6th sem/cd/C-Compiler/LexicalAnalyzer
OPERATOR & 19
IDENTIFIER b 19
SEMICOLON ; 19
IDENTIFIER h 20
OPERATOR = 20
IDENTIFIER a 20
OPERATOR || 20
IDENTIFIER b 20
SEMICOLON ; 20
IDENTIFIER h 21
OPERATOR = 21
IDENTIFIER a 21
OPERATOR * 21
LEFT PARENTHESIS (21
IDENTIFIER a 21
OPERATOR + 21
IDENTIFIER b 21
RIGHT PARENTHESIS) 21
SEMICOLON ; 21
IDENTIFIER h 22
OPERATOR = 22
IDENTIFIER a 22
OPERATOR * 22
IDENTIFIER a 22
OPERATOR + 22
IDENTIFIER b 22
OPERATOR * 22
IDENTIFIER b 22
SEMICOLON ; 22
IDENTIFIER h 23
OPERATOR = 23
FUNCTION fun 23
LEFT PARENTHESIS (23
IDENTIFIER b 23
RIGHT PARENTHESIS) 23
SEMICOLON ; 23
SINGLE LINE COMMENT //This Test case contains operator,structure,delimeters,Function; 25
RIGHT BRACE } 26

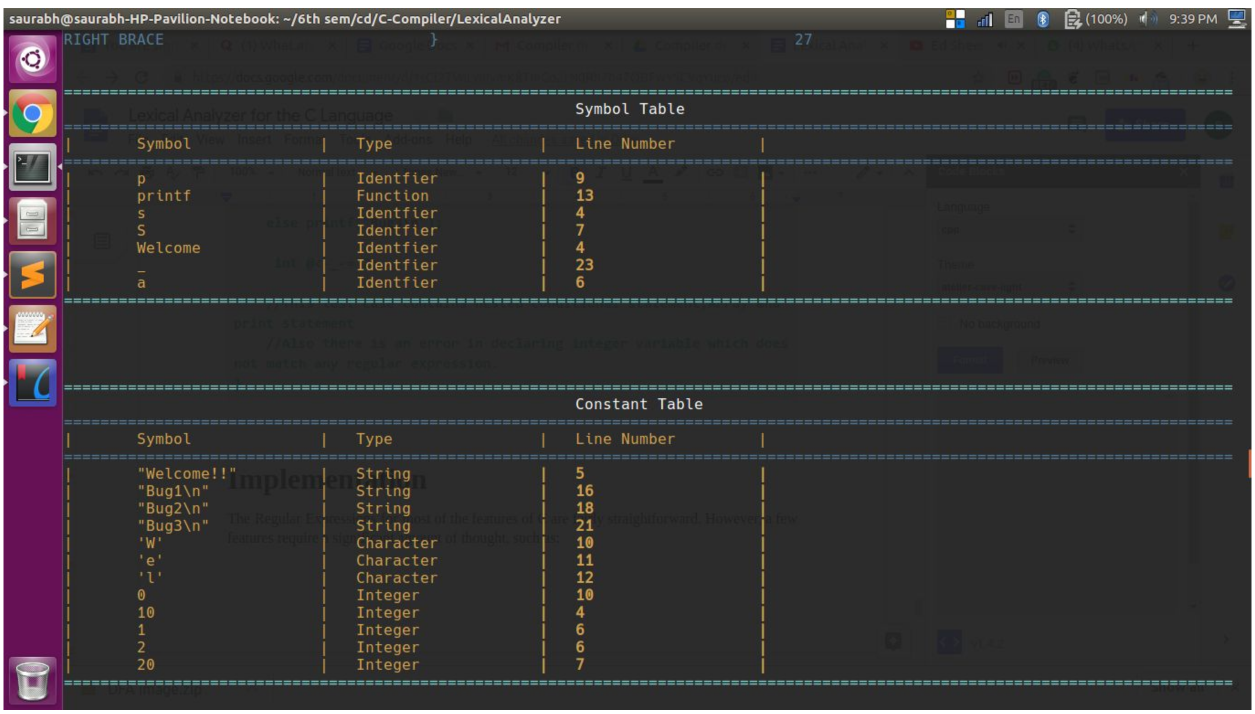

saurabh@saurabh-HP-Pavilion-Notebook: ~/6th sem/cd/C-Compiler/LexicalAnalyzer
LEFT BRACE { 12
KEYWORD int 13
IDENTIFIER a 13
OPERATOR = 13
INTEGER 2 13
COMMA , 13
IDENTIFIER b 13
COMMA , 13
IDENTIFIER c 13
COMMA , 13
IDENTIFIER d 13
COMMA , 13
IDENTIFIER e 13
COMMA , 13
IDENTIFIER f 13
COMMA , 13
IDENTIFIER g 13
COMMA , 13
IDENTIFIER h 13
SEMICOLON ; 13
IDENTIFIER c 15
OPERATOR = 15
IDENTIFIER a 15
OPERATOR + 15
IDENTIFIER b 15
SEMICOLON ; 15
IDENTIFIER d 16
OPERATOR = 16
IDENTIFIER a 16
OPERATOR * 16
IDENTIFIER b 16
SEMICOLON ; 16
IDENTIFIER e 17
OPERATOR = 17
IDENTIFIER a 17
OPERATOR / 17
IDENTIFIER b 17
SEMICOLON ; 17
IDENTIFIER f 18

Status:PASS

Test Case 3: Error Code

```
#include<stdio.h>

int main(){
    char s[10]=Welcome!!";
    char s[]="Welcome!!";
    int a[2] = {1, 2};
    char S[20];

    int p;
    if(s[0]=='W'){
        if(s[1]=='e'){
            if(s[2]=='l'){
                printf("Welcome!!");
            }

            else printf("Bug1\n");
        }
        else printf("Bug2\n");
    }

    else printf("Bug3\n");

     int @<-_-= 2;

    //This test case contains nested conditional statement,Array and
print statement
    //Also there is an error in declaring integer variable which does
not match any regular expression.
}
```

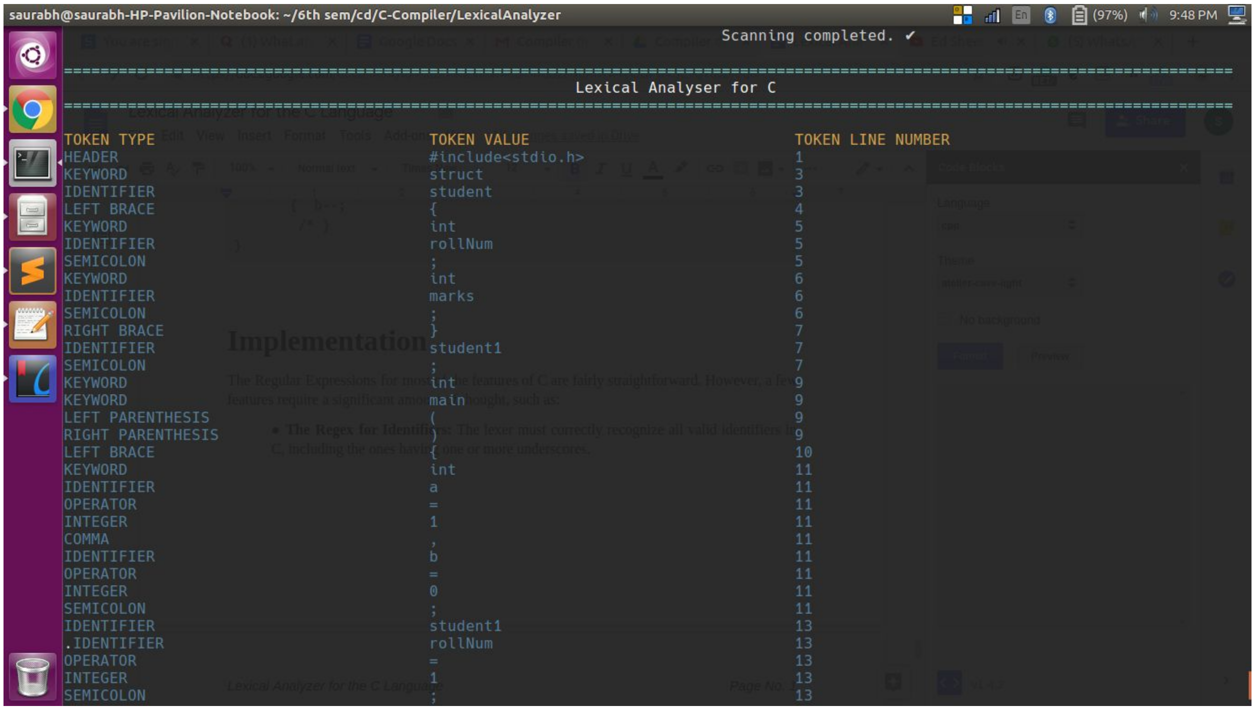

saurabh@saurabh-HP-Pavilion-Notebook: ~/6th sem/cd/C-Compiler/LexicalAnalyzer
9:38 PM
Lexical Analyser for C
TOKEN TYPE
TOKEN VALUE
TOKEN LINE NUMBER
HEADER #include<stdio.h> 1
KEYWORD int 3
KEYWORD main 3
LEFT PARENTHESIS (3
RIGHT PARENTHESIS) 3
LEFT BRACE { 3
KEYWORD char 4
IDENTIFIER s 4
LEFT BRACKET [4
INTEGER 10 4
RIGHT BRACKET] 4
OPERATOR = 4
IDENTIFIER Welcome 4
OPERATOR ! 4
OPERATOR ! 4
ERROR: UNCLOSED STRING "; 4
KEYWORD char 5
IDENTIFIER s 5
LEFT BRACKET [5
RIGHT BRACKET] 5
OPERATOR = 5
STRING "Welcome!!" 5
SEMICOLON ; 5
KEYWORD int 6
IDENTIFIER a 6
LEFT BRACKET [6
INTEGER 2 6
RIGHT BRACKET] 6
OPERATOR = 6
LEFT BRACE { 6
INTEGER 1 6
COMMA , 6
INTEGER 2 6
RIGHT BRACE } 6
SEMICOLON ; 6

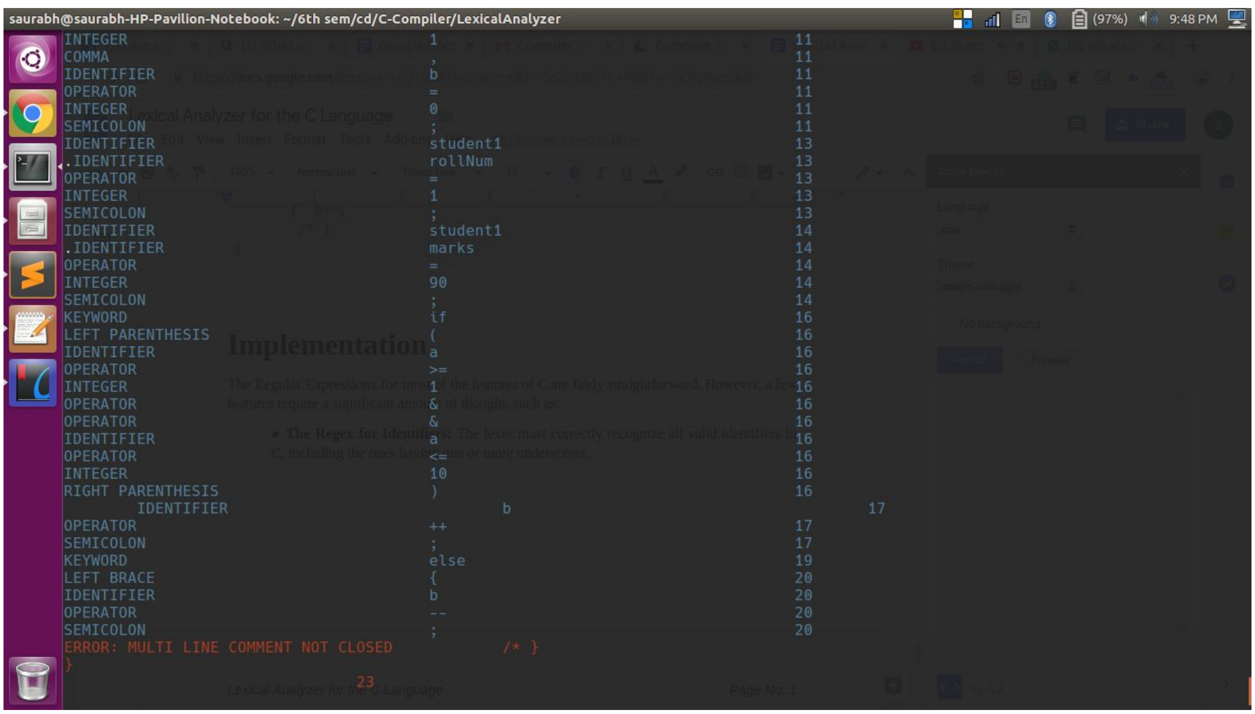

saurabh@saurabh-HP-Pavilion-Notebook: ~/6th sem/cd/C-Compiler/LexicalAnalyzer
9:39 PM
RIGHT BRACE } 27
Symbol Table
Symbol	Type	Line Number
p	Identfier	9
printf	Function	13
s	Identfier	4
S	Identfier	7
Welcome	Identfier	4
_	Identfier	23
a	Identfier	6
Constant Table
Symbol	Type	Line Number
"Welcome!!"	String	5
"Bug1\n"	String	16
"Bug2\n"	String	18
"Bug3\n"	String	21
'W'	Character	10
'e'	Character	11
'l'	Character	12
0	Integer	10
10	Integer	4
1	Integer	6
2	Integer	6
20	Integer	7

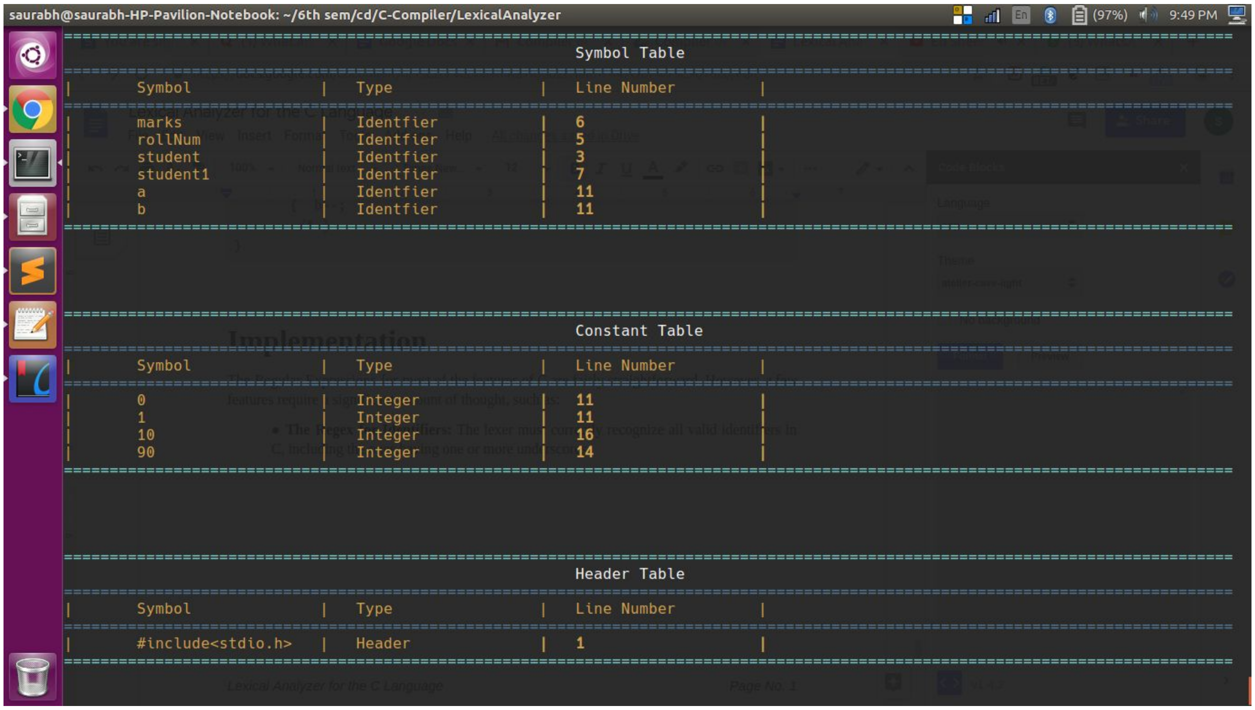
Status: PASS

Test Case 4:

```
#include<stdio.h>

struct student
{
  int rollNum;
  int marks;
}student1;

int main()
{
 int a = 1, b=0;

 student1.rollNum = 1;
 student1.marks = 90;

 if(a >= 1 && a <= 10)
      b++;

 else
       {  b--;
        /* }
}
```


saurabh@saurabh-HP-Pavilion-Notebook: ~/6th sem/cd/C-Compiler/LexicalAnalyzer
(97%) 9:48 PM
Scanning completed. ✔
Lexical Analyser for C
TOKEN TYPE TOKEN VALUE TOKEN LINE NUMBER
HEADER #include<stdio.h> 1
KEYWORD struct 3
IDENTIFIER student 3
LEFT BRACE { 4
KEYWORD int 5
IDENTIFIER rollNum 5
SEMICOLON ; 5
KEYWORD int 6
IDENTIFIER marks 6
SEMICOLON ; 6
RIGHT BRACE } 7
IDENTIFIER student1 7
SEMICOLON ; 7
KEYWORD int 9
KEYWORD main 9
LEFT PARENTHESIS (9
RIGHT PARENTHESIS) 9
LEFT BRACE { 10
KEYWORD int 11
IDENTIFIER a 11
OPERATOR = 11
INTEGER 1 11
COMMA , 11
IDENTIFIER b 11
OPERATOR = 11
INTEGER 0 11
SEMICOLON ; 11
IDENTIFIER student1 13
.IDENTIFIER rollNum 13
OPERATOR = 13
INTEGER 1 13
SEMICOLON ; 13


saurabh@saurabh-HP-Pavilion-Notebook: ~/6th sem/cd/C-Compiler/LexicalAnalyzer
(97%) 9:48 PM
INTEGER 1 11
COMMA , 11
IDENTIFIER b 11
OPERATOR = 11
INTEGER 0 11
SEMICOLON ; 11
IDENTIFIER student1 13
.IDENTIFIER rollNum 13
OPERATOR = 13
INTEGER 1 13
SEMICOLON ; 13
IDENTIFIER student1 14
.IDENTIFIER marks 14
OPERATOR = 14
INTEGER 90 14
SEMICOLON ; 14
KEYWORD if 16
LEFT PARENTHESIS (16
IDENTIFIER a 16
OPERATOR >= 16
INTEGER 1 16
OPERATOR & 16
OPERATOR & 16
IDENTIFIER a 16
OPERATOR <= 16
INTEGER 10 16
RIGHT PARENTHESIS) 16
IDENTIFIER b 17
OPERATOR ++ 17
SEMICOLON ; 17
KEYWORD else 19
LEFT BRACE { 20
IDENTIFIER b 20
OPERATOR -- 20
SEMICOLON ; 20
ERROR: MULTI LINE COMMENT NOT CLOSED /* }
}
23



Symbol Table

Symbol	Type	Line Number
marks	Identfier	6
rollNum	Identfier	5
student	Identfier	3
student1	Identfier	7
a	Identfier	11
b	Identfier	11

Constant Table

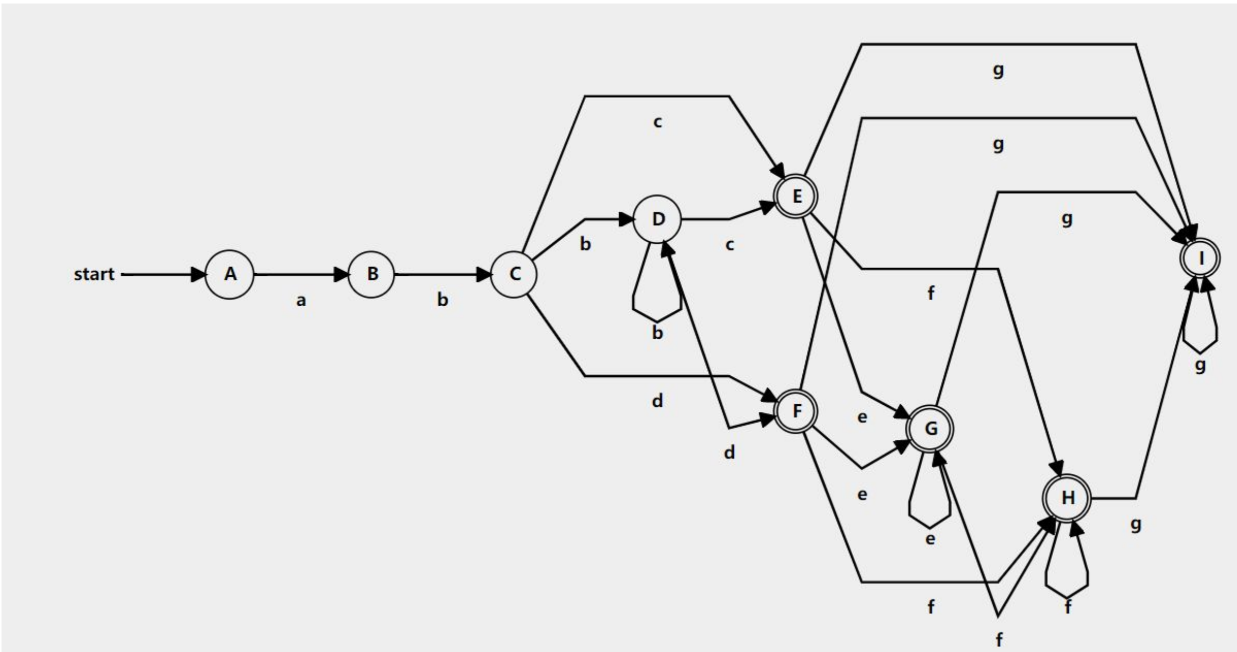
Symbol	Type	Line Number
0	Integer	11
1	Integer	11
10	Integer	16
90	Integer	14

Header Table

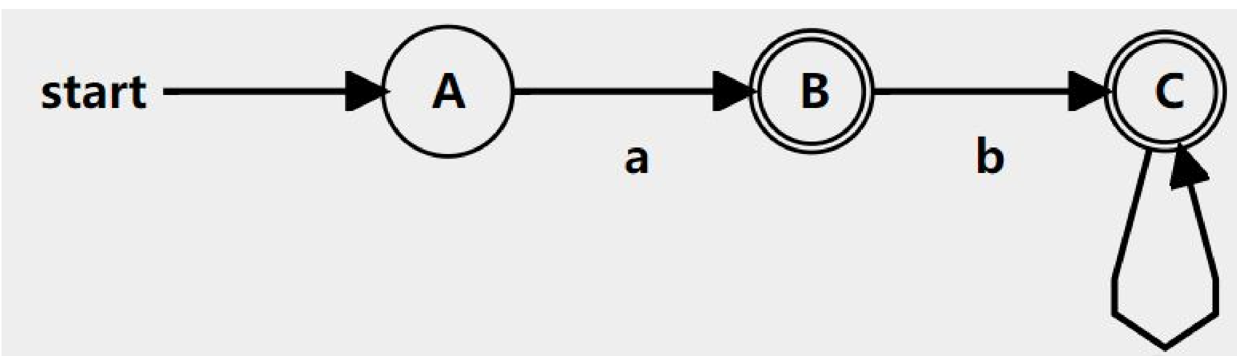
Symbol	Type	Line Number
#include<stdio.h>	Header	1

Status: PASS

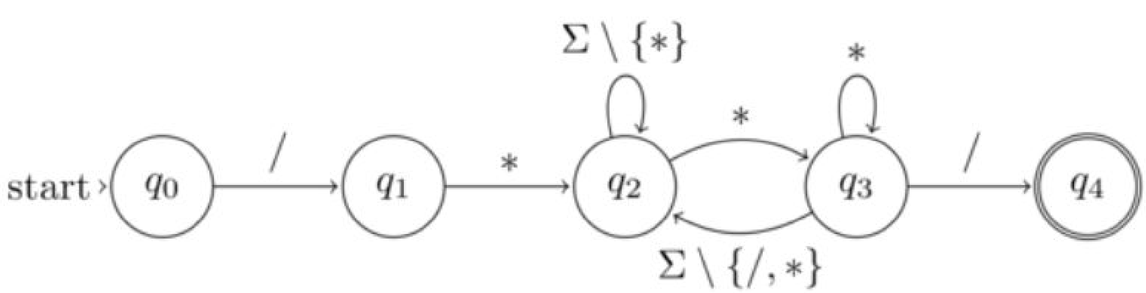
DFAs:



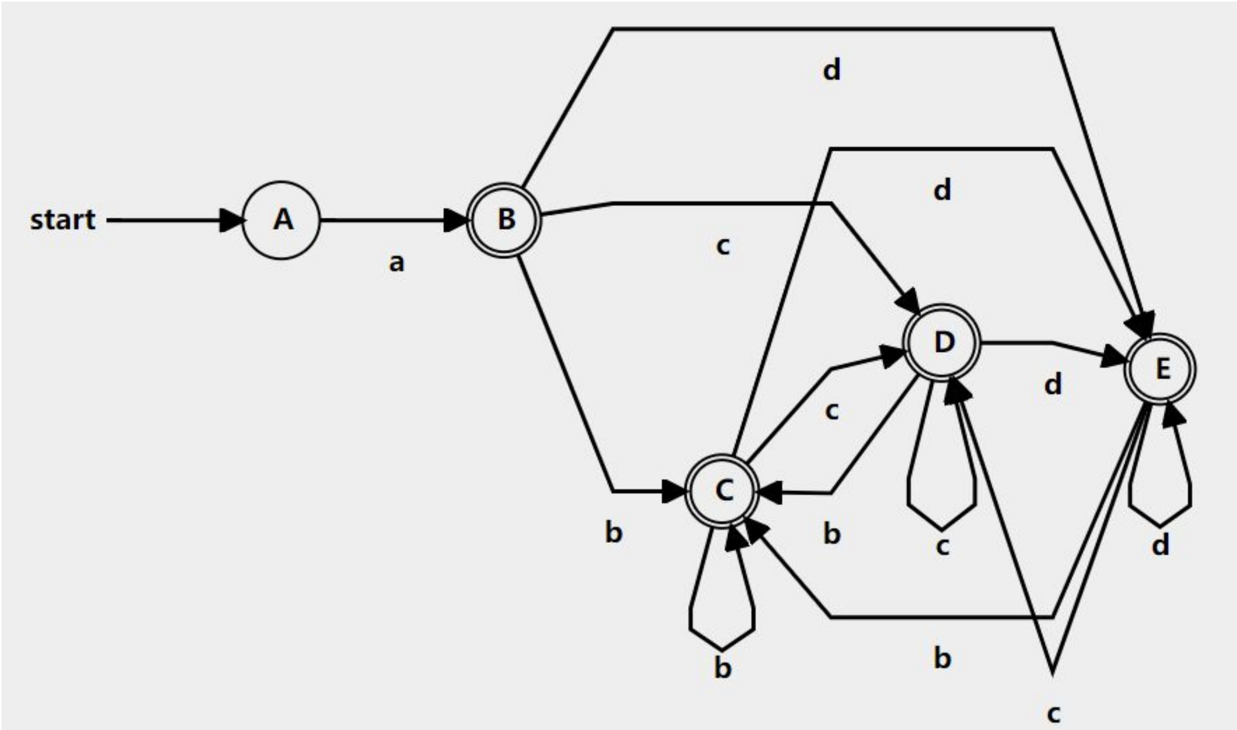
start
A
B
C
D
E
F
G
H
I
a
b
c
d
e
f
g

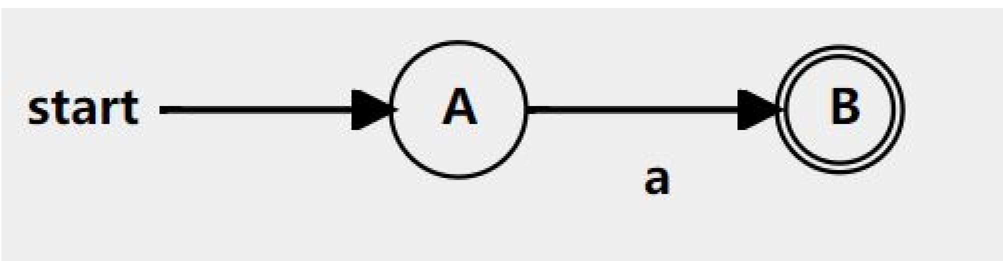


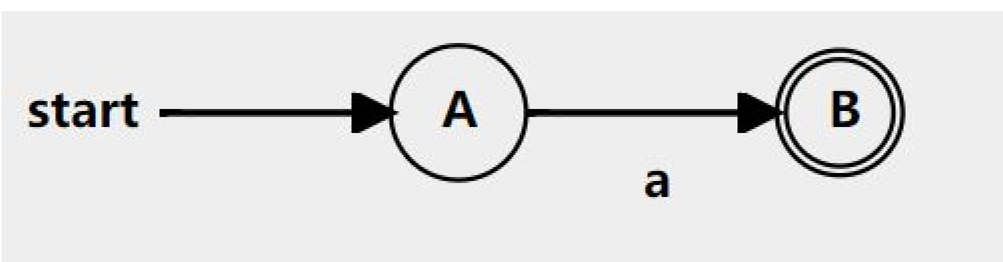
start
A
B
C
a
b

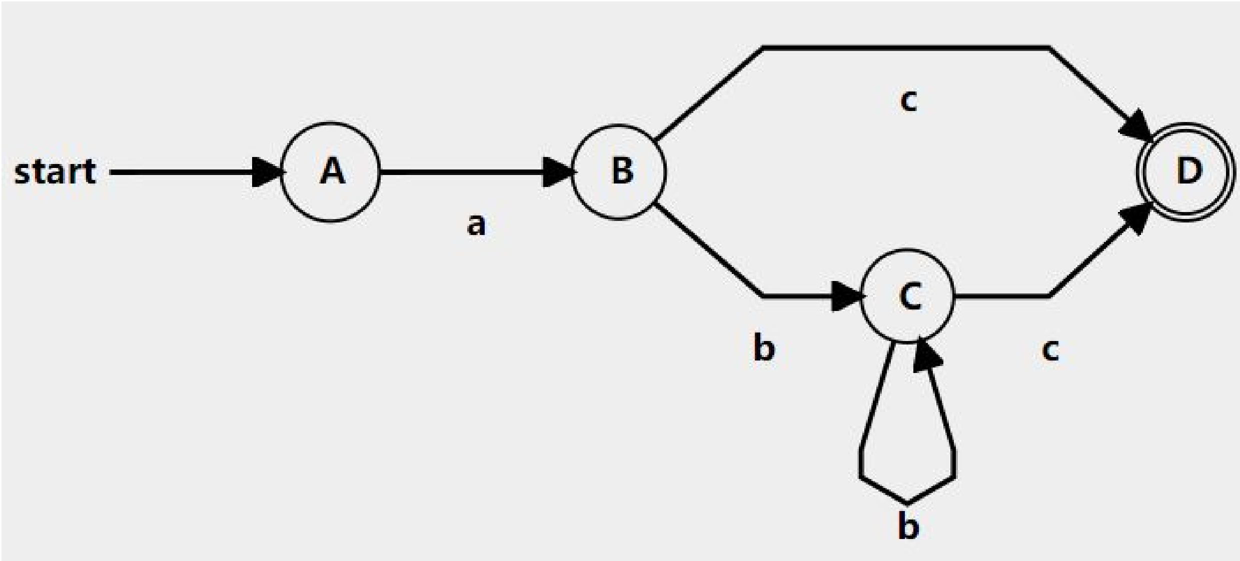


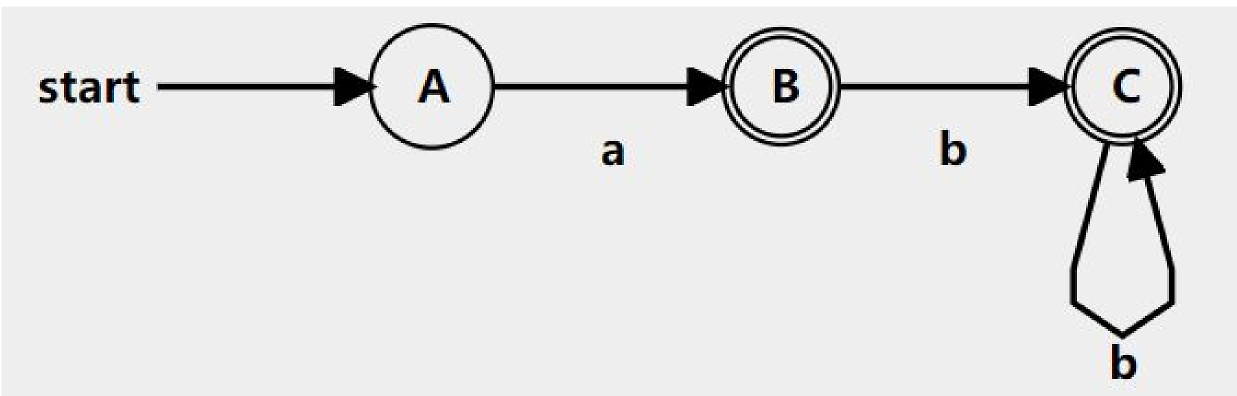
start
q_0
q_1
q_2
q_3
q_4
/
$*$
$\Sigma \setminus \{*\}$
$\Sigma \setminus \{/, *\}$

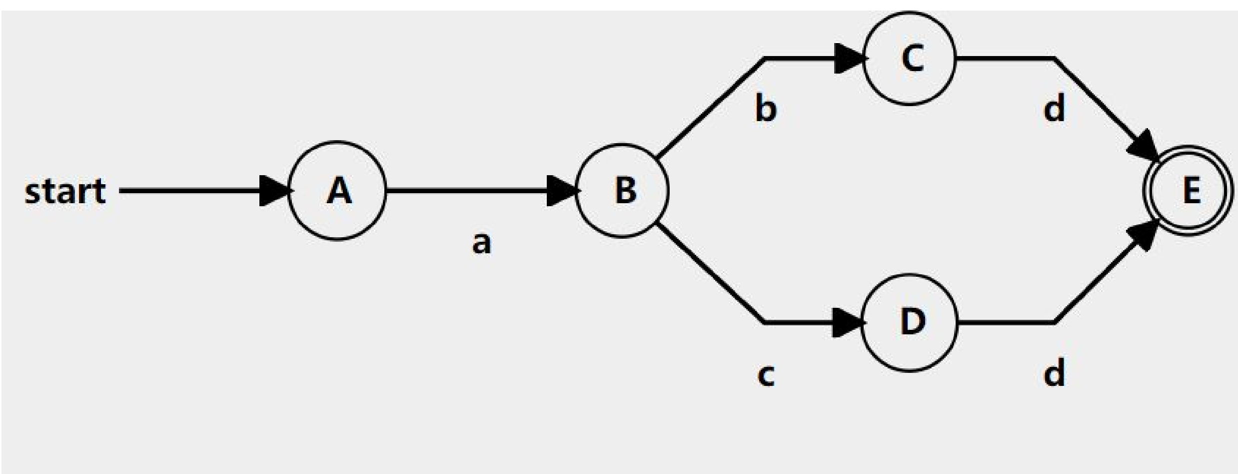

start
A
a
B
C
D
E
d
d
c
c
b
b
b
c
d
d
b
c

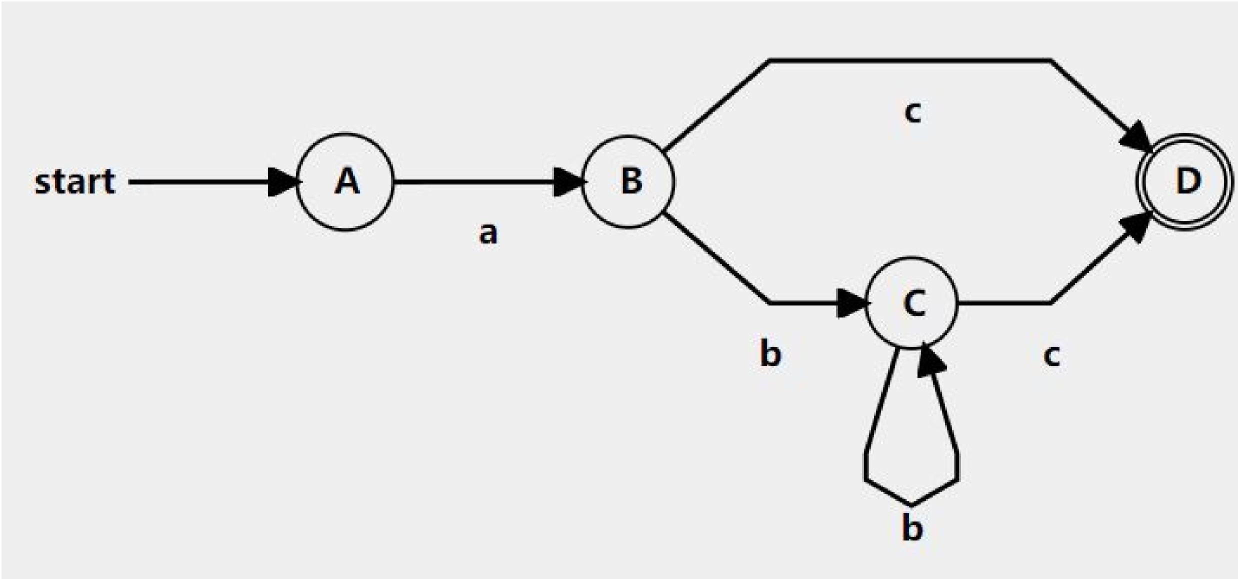

start
A
a
B

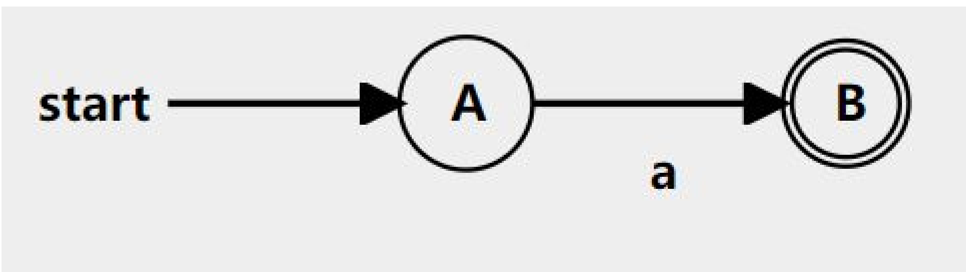

start
A
a
B

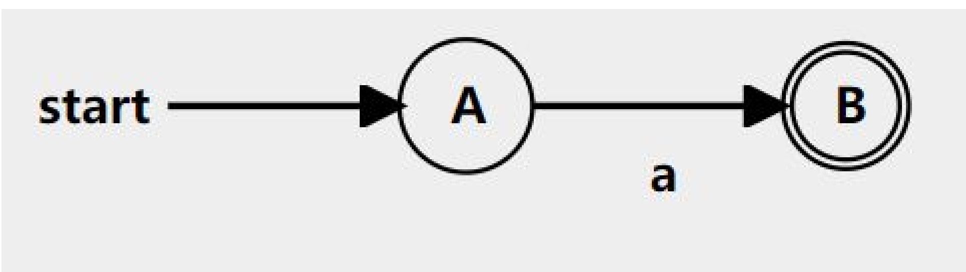

start
A
a
B
c
D
b
C
c
b

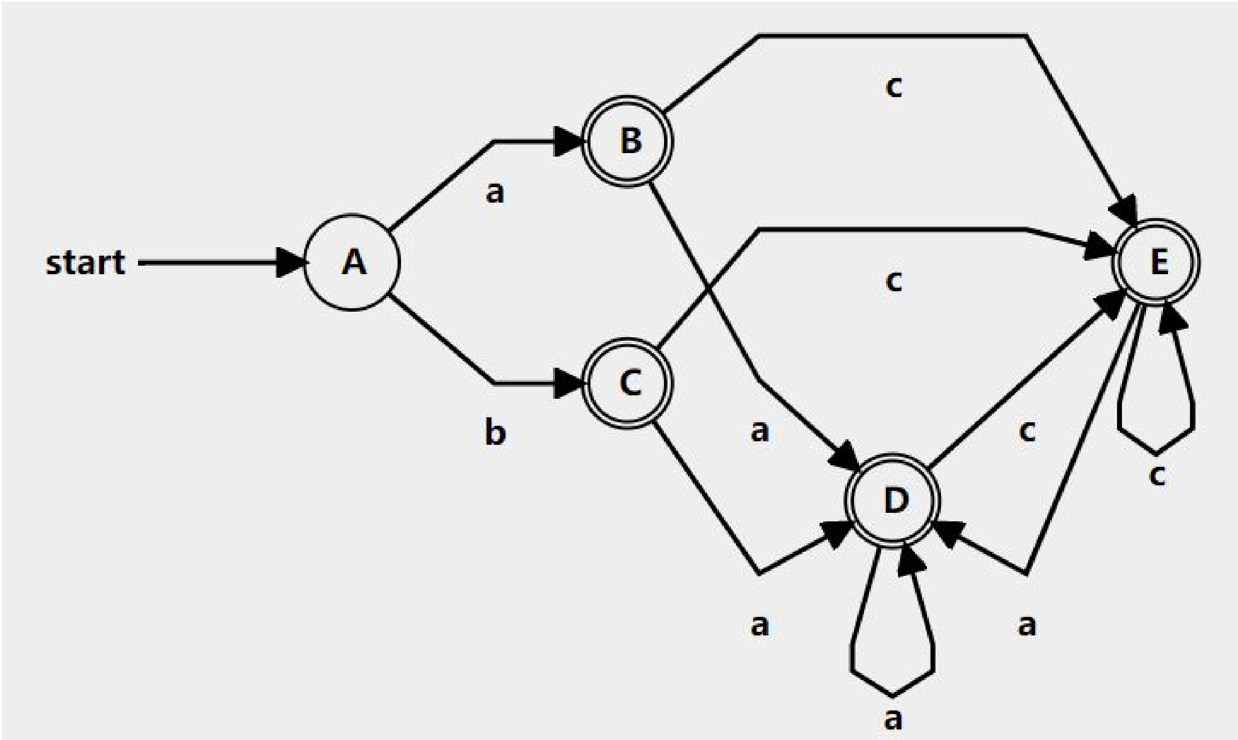

start
A
a
B
b
C
b


start
A
a
B
b
C
d
E
c
D
d

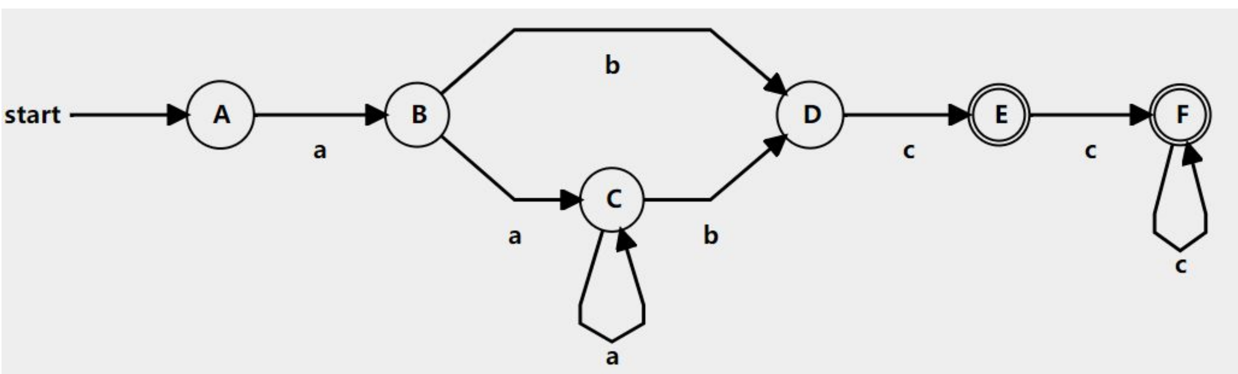

start
A
B
C
D
a
b
c
c
b


start
A
B
a

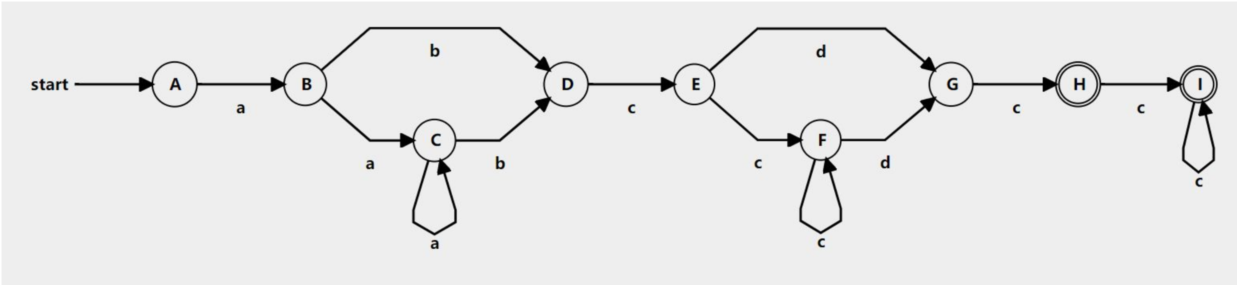

start
A
B
a

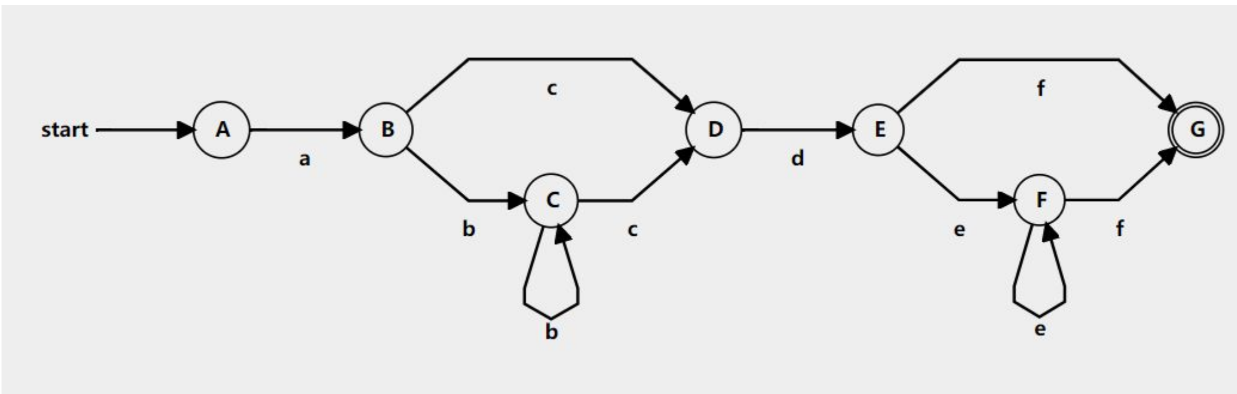


start
A
B
C
D
E
a
b
c
c
a
c
c
a
a
a



start
A
B
C
D
E
F
a
b
a
b
a
c
c
c





Implementation

The Regular Expressions for most of the features of C are fairly straightforward. However, a few features require a significant amount of thought, such as:

- **The Regex for Identifiers:** The lexer must correctly recognize all valid identifiers in C, including the ones having one or more underscores.
- **Multiline comments should be supported:** To implement it a proper regular expression was written along with that lookahead character set for operators were thought so to resolve conflict with the division operator.
- **Literals:** Different regular expressions have been implemented in the code to support all kinds of literals, i.e integers, floats, strings, etc.
- **Error Handling for Incomplete String:** Open and close quote missing, both kind of errors have been handled in the rules written in the script.
- **Error Handling for Unmatched Comments:** This has been handled by adding lookahead characters to operator regular expression. If there is an unmatched comment then it does not match with any of the patterns in the rule. Hence it goes to default state which in turn throws an error.

- **Error Handling for unclean integer constant:** This has been handled by adding appropriate lookahead characters for integer constant. E.g. `int a = 786rt`, is rejected as the integer constant should never follow an alphabet

At the end of the token recognition, the lexer prints a list of all the identifiers and constants present in the program. We use the following technique to implement this:

● We maintain two structures one for symbol table and other for constant tableone corresponding to identifiers and other to constants.

● Four functions have been implemented `lookupST( )`, `lookupCT( )`, these functions return true if the identifier and constant respectively are already present in the table. `InsertST( )`, `InsertCT( )` help to insert identifier/constant in the appropriate table.

● Whenever we encounter an identifier/constant, we call the `insertST()` or `insertCT()` function which in turns call `lookupST( )` or `lookupCT( )` and adds it to the corresponding structure.

● In the end, in `main( )` function, after yylex returns, we call `printST( )` and `printCT( )`, which in turn prints the list of identifier and constants in a proper format.

Results:

1. Token --- Token Class
2. Symbol Table:

   Token --- Attribute
3. Constant Table

   Token --- Attribute

Future work:

The flex script presented in this report takes care of all the rules of the C language, but is not fully exhaustive in nature. Our future work would include making the script even more robust in order to handle all aspects of the C language and making it more efficient.

References:

● Compilers Principles, Techniques and Tool by Alfred V.Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

● http://dinosaur.compilertools.net/lex/index.html

● http://www.csd.uwo.ca/~moreno/CS447/Lectures/Lexical.html/node11.html