LAB MANUAL

of

Compiler Design Laboratory

(CSE606)

Bachelor of Technology (CSE)

By

Vedant Gandhi (22000788)

Department of Computer Science and Engineering

School Engineering and Technology

Navrachana University, Vadodara

Autumn Semester

(2024-2025)

TABLE OF CONTENT

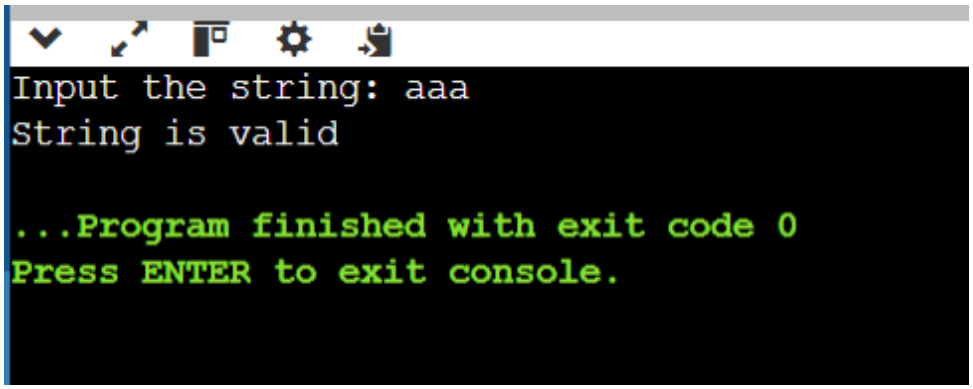
EXPERIMENT 1

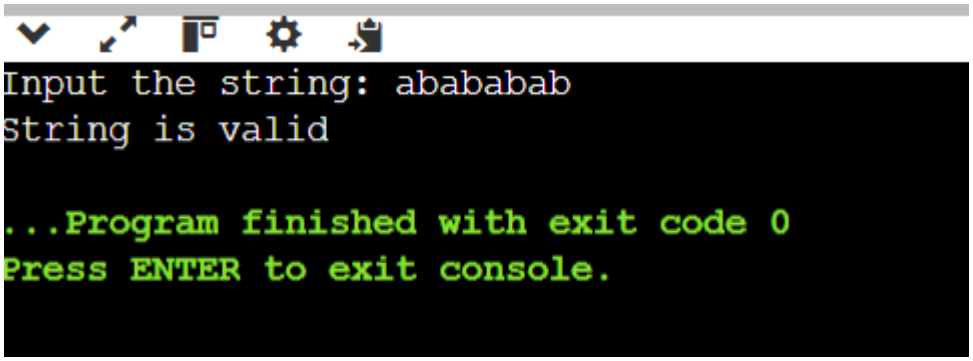
a)
Write a program to recognize strings starts with ‘a’ over {a, b}.

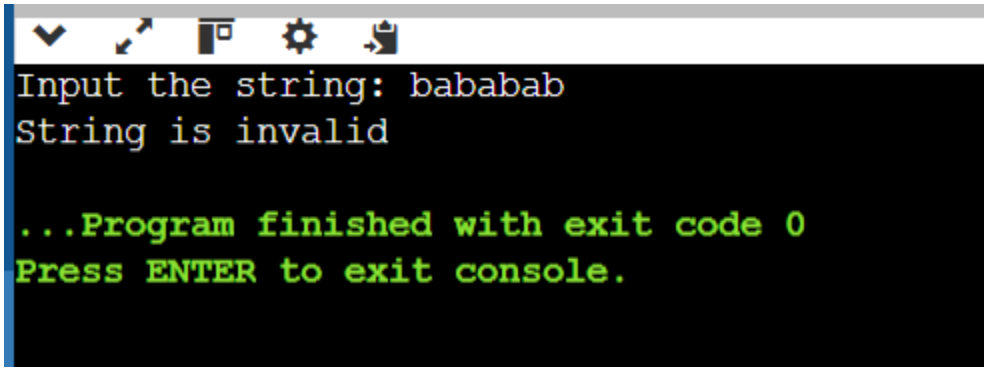
SOURCE CODE:

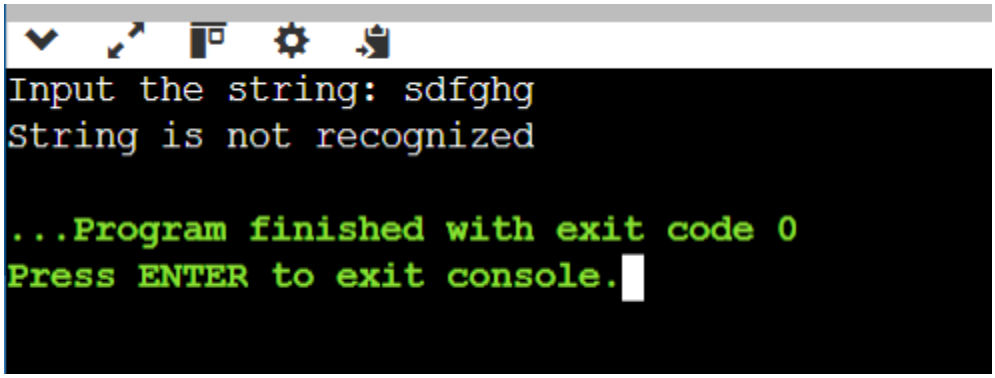
```c
#include <stdio.h>

int main(){
   char input[10];
   int i=0, state=0;
   printf("Input the string: ");
   scanf("%s", input);

   while(input[i]!='\0'){
       switch(state) {
           case 0:
               if(input[i]=='a') state = 1;
               else if (input[i] == 'b') state = 2;
               else state = 3;
               break;

           case 1:
               if (input[i] == 'a' || input[i] == 'b') state = 1;
               else state = 3;
               break;

           case 2:
               if (input[i] == 'a' || input[i] == 'b') state = 2;
               else state = 3;
               break;

           case 3:
               state = 3;
               break;
       }

       i++;
   }

   if (state==1) printf("String is valid");
   else if (state==2 || state==0) printf("String is invalid");
   else if (state==3) printf("String is not recognized");
   return 0;
}
```

OUTPUT:


Input the string: aaa
String is valid
...Program finished with exit code 0
Press ENTER to exit console.


Input the string: abababab
String is valid
...Program finished with exit code 0
Press ENTER to exit console.


Input the string: bababab
String is invalid
...Program finished with exit code 0
Press ENTER to exit console.

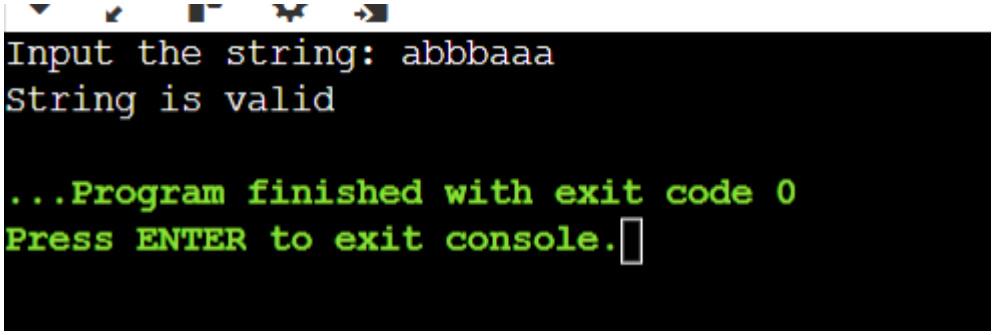

Input the string: sdfghg
String is not recognized
...Program finished with exit code 0
Press ENTER to exit console.

b)
Write a program to recognize strings end with ‘a’.

SOURCE CODE:

```c
#include <stdio.h>

int main(){
   char input[10];
   int i=0, state=0;
   printf("Input the string: ");
   scanf("%s", input);

   while(input[i]!='\0'){
       switch(state) {
           case 0:
               if(input[i]=='a') state = 1;
               else state = 0;
               break;

           case 1:
               if (input[i] == 'a') state = 1;
               else state = 0;
               break;

       }

       i++;
   }

   if (state==1) printf("String is valid");
   else if (state==0) printf("String is invalid");
   return 0;
}
```

OUTPUT:



```
Input the string: babbaaaa
String is valid

...Program finished with exit code 0
Press ENTER to exit console.
```

```
Input the string: abbbbbb
String is invalid

...Program finished with exit code 0
Press ENTER to exit console.
```

```
Input the string: fdghjn
String is invalid

...Program finished with exit code 0
Press ENTER to exit console.
```

c)

Write a program to recognize strings end with ‘ab’. Take the input from text file.

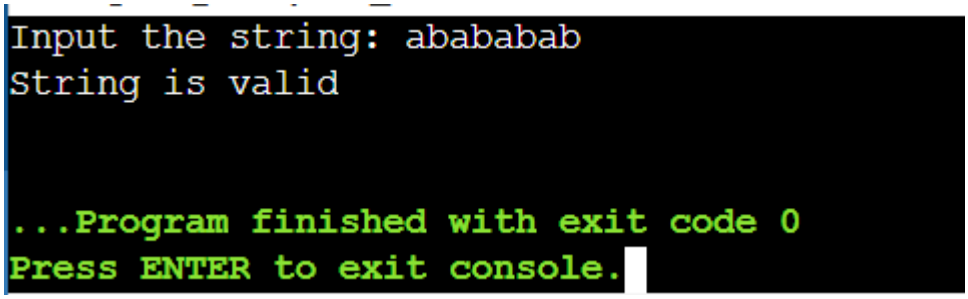
SOURCE CODE:

```c
#include <stdio.h>

int main(){
   char input[10];
   int i=0, state=0;
   printf("Input the string: ");
   scanf("%s", input);

   int len = strlen(input);

   if (len >= 2 && input[len - 2] == 'a' && input[len - 1] == 'b') {
       printf("String is valid\n");
   } else {
       printf("String is invalid\n");
   }

   return 0;
}
```

OUTPUT:

```
Input the string: dgfhab
String is valid


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Input the string: gsfdghb
String is invalid


...Program finished with exit code 0
Press ENTER to exit console.
```


Input the string: abababab
String is valid
...Program finished with exit code 0
Press ENTER to exit console.

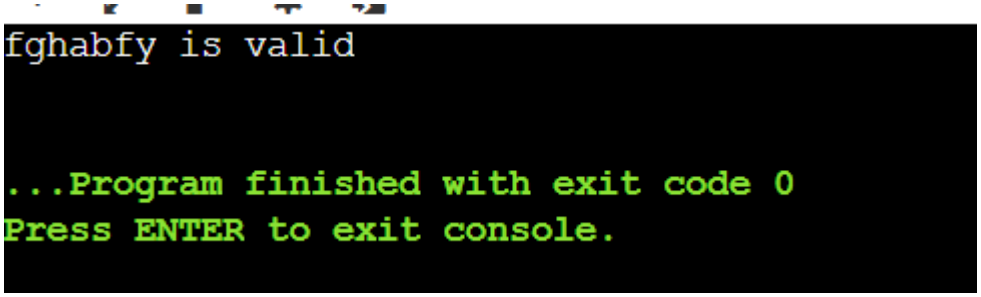
d)

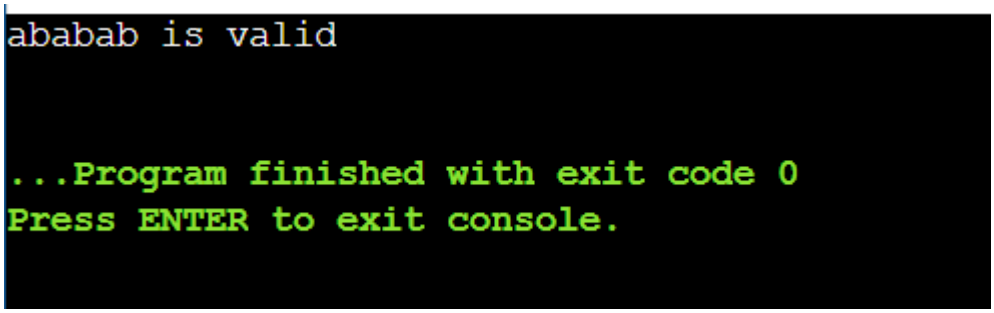
Write a program to recognize strings contains ‘ab’. Take the input from text file.

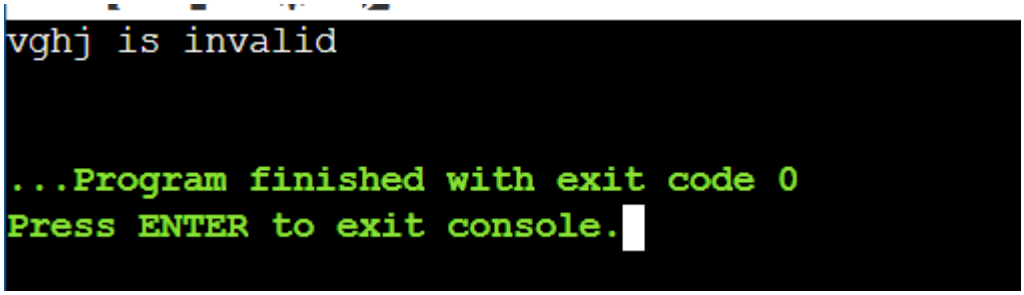
SOURCE CODE:

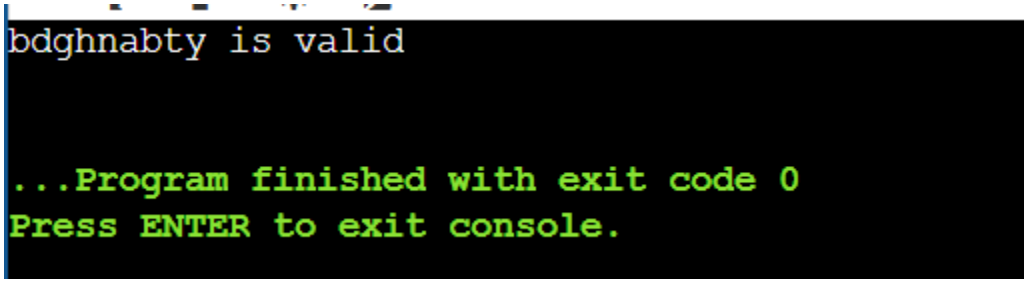
```c
#include <stdio.h>
#include <string.h>

int main() {
    char input[100];
    FILE *file = fopen("input.txt", "r");

    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    fscanf(file, "%s", input);
    fclose(file);

    if (strstr(input, "ab") != NULL) {
        printf("%s is valid\n",input);
    } else {
        printf("%s is invalid\n",input);
    }

    return 0;
}
```

OUTPUT:




ababab is valid
...Program finished with exit code 0
Press ENTER to exit console.


vghj is invalid
...Program finished with exit code 0
Press ENTER to exit console.


bdghnabty is valid
...Program finished with exit code 0
Press ENTER to exit console.

EXPERIMENT 2

a)

Write a program to recognize the valid identifiers and keywords

SOURCE CODE:

Input.txt:

if

else

my_var

1variable

_underscore

class

def

function_name

99bottles

helloWorld

int

float

__init__

try

except

lambda

whileTrue

True

False

None

CODE:

```
def is_valid_identifier(token):
    state = 0
    for char in token:
        if state == 0:
            if char.isalpha() or char == '_':
                state = 1
            else:
                return False
        elif state == 1:
            if char.isalnum() or char == '_':
                state = 1
            else:
                return False
    return state == 1

def is_keyword(token):
    keywords = {"if", "else", "while", "return", "for", "def", "class",
"import", "from", "as", "with", "try", "except", "finally", "raise", "lambda",
"pass", "break", "continue", "in", "not", "or", "and", "is", "None", "True",
"False", "global", "nonlocal", "assert", "yield"}
    return token in keywords

def tokenize_and_check(input_string):
    tokens = input_string.split()
    results = []
    for token in tokens:
        identifier = is_valid_identifier(token)
        keyword_check = is_keyword(token)
        status = "Both Identifier and Keyword" if identifier and keyword_check
else \
                 "Valid Identifier" if identifier else \
                 "Keyword" if keyword_check else "Invalid"
        results.append((token, status))

    return results

if __name__ == "__main__":
    with open("input.txt", "r") as file:
        input_string = file.read().strip()

    results = tokenize_and_check(input_string)

    with open("output.txt", "w") as file:
```

```
        file.write("Tokenized Output:\n")
        for token, status in results:
            file.write(f"Token: '{token}', Status: {status}\n")

    print("\nTokenized Output:")
    for token, status in results:
        print(f"Token: '{token}', Status: {status}")
```

OUTPUT:

```
Tokenized Output:
Token: 'if', Status: Both Identifier and Keyword
Token: 'else', Status: Both Identifier and Keyword
Token: 'my_var', Status: Valid Identifier
Token: '1variable', Status: Invalid
Token: '_underscore', Status: Valid Identifier
Token: 'class', Status: Both Identifier and Keyword
Token: 'def', Status: Both Identifier and Keyword
Token: 'function_name', Status: Valid Identifier
Token: '99bottles', Status: Invalid
Token: 'helloWorld', Status: Valid Identifier
Token: 'int', Status: Valid Identifier
Token: 'float', Status: Valid Identifier
Token: '__init__', Status: Valid Identifier
Token: 'try', Status: Both Identifier and Keyword
Token: 'except', Status: Both Identifier and Keyword
Token: 'lambda', Status: Both Identifier and Keyword
Token: 'whileTrue', Status: Valid Identifier
Token: 'True', Status: Both Identifier and Keyword
Token: 'False', Status: Both Identifier and Keyword
Token: 'None', Status: Both Identifier and Keyword
```

b)

Write a program to recognize the valid operators.

SOURCE CODE:

```
operators = {'+', '-', '*', '/', '%', '=', '==', '!=', '>', '<', '>=', '<='}

def is_identifier(token):
    return token.isalnum() and not token.isdigit()

expression = input("Enter a string: ")

for char in expression:
    if char in operators:
        print(f"{char} is an operator.")
    elif char.isalnum():
        print(f"{char} is an identifier.")
```

OUTPUT:

```
Enter a string: a+b-c=e*q
a is an identifier.
+ is an operator.
b is an identifier.
- is an operator.
c is an identifier.
= is an operator.
e is an identifier.
* is an operator.
q is an identifier.
```

c)

Write a program to recognize the valid number.

SOURCE CODE:

Numbers.txt:

123

-456.78

3.14159

1E10

-2.5e-3

+100

abc

12.34.56

E45

1.2.3

CODE:

```
def is_valid_number_fsm(number: str) -> bool:
    state = 'a'
    for char in number:
        if state == 'a':
            if char in '+-':
                state = 'h'
            elif char.isdigit():
                state = 'b'
            else:
                return False
        elif state == 'h':
            if char.isdigit():
                state = 'b'
            else:
                return False
        elif state == 'b':
            if char.isdigit():
                state = 'b'
            elif char == '.':
                state = 'c'
            elif char in 'Ee':
```

```
                state = 'e'
            else:
                return False
        elif state == 'c':
            if char.isdigit():
                state = 'd'
            else:
                return False
        elif state == 'd':
            if char.isdigit():
                state = 'd'
            elif char in 'Ee':
                state = 'e'
            else:
                return False
        elif state == 'e':
            if char in '+-':
                state = 'f'
            elif char.isdigit():
                state = 'g'
            else:
                return False
        elif state == 'f':
            if char.isdigit():
                state = 'g'
            else:
                return False
        elif state == 'g':
            if char.isdigit():
                state = 'g'
            else:
                return False
    return state in {'b', 'd', 'g'}

if __name__ == "__main__":
    try:
        with open("numbers.txt", "r") as file:
            for line in file:
                number = line.strip()
                print(f"'{number}' is a valid number:
{is_valid_number_fsm(number)}")
    except FileNotFoundError:
        print("Error: 'numbers.txt' file not found.")
```

OUTPUT:


'123' is a valid number: True
'-456.78' is a valid number: True
'3.14159' is a valid number: True
'1E10' is a valid number: True
'-2.5e-3' is a valid number: True
'+100' is a valid number: True
'abc' is a valid number: False
'12.34.56' is a valid number: False
'E45' is a valid number: False
'1.2.3' is a valid number: False

d)

Write a program to recognize the valid comments

SOURCE CODE:

Comments.txt:

Hello World

// This is a single-line comment

/* This is a multi-line comment */

Not a comment

/* Unclosed comment

CODE:

```
def is_valid_comment(line: str) -> bool:
    state = 'start'
    i = 0
    while i < len(line):
        char = line[i]

        if state == 'start':
            if char == '/':
                state = 'slash'
            else:
                return False

        elif state == 'slash':
            if char == '/':
                return True
            elif char == '*':
                state = 'multi_line'
            else:
                return False

        elif state == 'multi_line':
            if char == '*':
                state = 'multi_line_end'

        elif state == 'multi_line_end':
            if char == '/':
                return True
            elif char != '*':
                state = 'multi_line'
```

```
        i += 1

    return state == 'multi_line_end'

if __name__ == "__main__":
    try:
        with open("comments.txt", "r") as file:
            for line in file:
                line = line.strip()
                print(f"'{line}' is a valid comment:
{is_valid_comment(line)}")
    except FileNotFoundError:
        print("Error: 'comments.txt' file not found.")
```

OUTPUT:

```
'Hello World' is a valid comment: False
'// This is a single-line comment' is a valid comment: True
'/* This is a multi-line comment */' is a valid comment: True
'Not a comment' is a valid comment: False
'/* Unclosed comment' is a valid comment: False
```

e)

Program to implement Lexical Analyzer.

SOURCE CODE:

Input2.txt:

```
// This is a single-line comment
/* This is
  a multi-line comment */
int main() {
   int a = 10;
   float b = 3.14;
   char c = 'A';
   if (a < b) {
      a = a + 1;
   }
   return 0;
}
```

CODE:

```
def check(lexeme):
    keywords = {"auto", "break", "case", "char", "const", "continue",
"default", "do",
                "double", "else", "enum", "extern", "float", "for", "goto",
"if",
                "inline", "int", "long", "register", "restrict", "return",
"short", "signed",
                "sizeof", "static", "struct", "switch", "typedef", "union",
"unsigned", "void", "volatile", "while"}
    if lexeme in keywords:
        print(f"{lexeme} is a keyword")
    else:
        print(f"{lexeme} is an identifier")

def lexer(filename):
    try:
        with open(filename, "r") as f:
            buffer = f.read()
```

```
    except FileNotFoundError:
        print("Error opening file")
        return

    state = 0
    lexeme = ""
    f = 0
    while f < len(buffer):
        c = buffer[f]
        if state == 0:
            if c.isalpha() or c == '_':
                state = 1
                lexeme += c
            elif c.isdigit():
                state = 13
                lexeme += c
            elif c == '/':
                state = 11
            elif c in " \t\n":
                state = 0
            elif c in ";,+-*/%=<>(){}[]":
                print(f"{c} is a symbol")
                state = 0
            else:
                state = 0
        elif state == 1:
            if c.isalnum() or c == '_':
                lexeme += c
            else:
                check(lexeme)
                lexeme = ""
                state = 0
                f -= 1
        elif state == 11:
            if c == '/':
                while f < len(buffer) and buffer[f] != '\n':
                    f += 1
                state = 0
            elif c == '*':
                f += 1
                while f < len(buffer) - 1 and not (buffer[f] == '*' and
buffer[f + 1] == '/'):
                    f += 1
                f += 2
                state = 0
            else:
                print("/ is an operator")
                state = 0
```

```
            f -= 1
        elif state == 13:
            if c.isdigit():
                lexeme += c
            elif c == '.':
                state = 14
                lexeme += c
            elif c in "Ee":
                state = 16
                lexeme += c
            else:
                print(f"{lexeme} is a valid integer")
                lexeme = ""
                state = 0
                f -= 1
        elif state == 14:
            if c.isdigit():
                lexeme += c
                state = 15
            else:
                print("Error: Invalid floating point format")
                lexeme = ""
                state = 0
        elif state == 15:
            if c.isdigit():
                lexeme += c
            elif c in "Ee":
                state = 16
                lexeme += c
            else:
                print(f"{lexeme} is a valid floating point number")
                lexeme = ""
                state = 0
                f -= 1
        elif state == 16:
            if c in "+-":
                state = 17
                lexeme += c
            elif c.isdigit():
                state = 18
                lexeme += c
            else:
                print("Error: Invalid scientific notation")
                lexeme = ""
                state = 0
        elif state == 17:
            if c.isdigit():
                state = 18
```

```
        lexeme += c
            else:
                print("Error: Invalid exponent format")
                lexeme = ""
                state = 0
        elif state == 18:
            if c.isdigit():
                lexeme += c
            else:
                print(f"{lexeme} is a valid scientific notation number")
                lexeme = ""
                state = 0
                f -= 1
        f += 1

lexer("input2.txt")
```

OUTPUT:

```
int is a keyword
main is an identifier
( is a symbol
) is a symbol
{ is a symbol
int is a keyword
a is an identifier
= is a symbol
10 is a valid integer
; is a symbol
float is a keyword
b is an identifier
= is a symbol
3.14 is a valid floating point number
; is a symbol
char is a keyword
c is an identifier
= is a symbol
A is an identifier
; is a symbol
if is a keyword
( is a symbol
a is an identifier
< is a symbol
b is an identifier
) is a symbol
```

```
{ is a symbol
a is an identifier
= is a symbol
a is an identifier
+ is a symbol
1 is a valid integer
; is a symbol
} is a symbol
return is a keyword
0 is a valid integer
; is a symbol
} is a symbol
```

EXPERIMENT 3

To Study about Lexical Analyzer Generator (LEX) and Flex(Fast Lexical Analyzer)

DESCRIPTION:

Lexical analysis is the first phase of a compiler, responsible for converting source code into tokens. This phase is automated using **Lexical Analyzer Generators** like **LEX** and **Flex**.

LEX (Lexical Analyzer Generator)

LEX is a tool used for **generating lexical analyzers** in compiler design. It helps in pattern recognition and tokenizing input text using **regular expressions**. LEX works by defining patterns and corresponding actions in a .l file, which is then processed to generate a C-based scanner.

Key Features of LEX:

- Uses **regular expressions** to match patterns in input text.
- Generates **lex.yy.c**, a C program implementing the scanner.
- Can be compiled using a C compiler to produce an executable lexer.
- Works with **YACC (Yet Another Compiler Compiler)** to build full-fledged compilers.

Working of LEX:

1. **Specification:** The user writes a .l file containing regular expressions and C actions.
2. **Processing:** The lex command processes the .l file and generates lex.yy.c.
3. **Compilation:** The lex.yy.c is compiled with gcc to create an executable scanner.
4. **Execution:** The scanner reads input, matches patterns, and executes the corresponding actions.

Flex (Fast Lexical Analyzer)

Flex is an **enhanced and faster version of LEX**, designed for improved performance and portability. It follows the same working mechanism as LEX but generates more **efficient** and **optimized** C code.

Key Features of Flex:

- Faster and more efficient than LEX.
- Uses **longest match rule** over first match rule.
- Generates **lex.yy.c**, similar to LEX but optimized for better performance.
- Works seamlessly on **Linux, Unix, and Windows** with the required dependencies.

Working of Flex:

1. **Write a `` file** with pattern definitions and C-based actions.
2. **Use the `` command** to generate lex.yy.c.
3. **Compile the file** using gcc.
4. **Run the executable**, which scans the input and processes tokens.

Differences Between LEX and Flex

Feature	LEX	Flex
Speed	Slower	Faster
Portability	Limited	Widely used in Linux & Unix
Memory Usage	Higher	Optimized
Output File	lex.yy.c	lex.yy.c
Default Action	Returns first match	Returns longest match

Procedure

1. Create a .l file (e.g., lexer.l) containing regular expressions and C code.
2. Use the flex command to generate lex.yy.c.
3. Compile the generated C file using GCC.
4. Run the executable and provide input for analysis.

Example Code (LEX/Flex Program)

```
%{
#include <stdio.h>
%}
%%
[0-9]+      { printf("NUMBER\n"); }
[a-zA-Z]+   { printf("IDENTIFIER\n"); }
.           { printf("SPECIAL CHARACTER\n"); }
```

```
%%
int main() {
   yylex();
   return 0;
}
```

Conclusion

LEX and Flex are powerful tools for lexical analysis in compilers. They help automate **tokenization** using **regular expressions** and **C functions**, making lexical analysis efficient.

EXPERIMENT 4

a)

Write a Lex program to take input from text file and count no of characters, no. of lines & no. of words.

SOURCE CODE:

input.txt:

Hello

Good Morning

This is my lex program

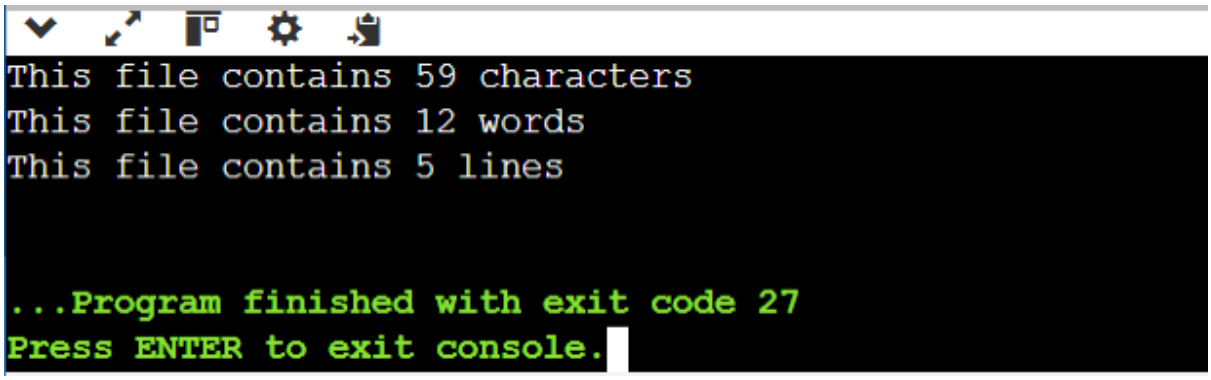
123456 677 34.676

56e56

sample.l:

```
%{
#include<stdio.h>
int char_count=0, word_count=0, line_count=0;
%}
%%
\n  {line_count++;word_count++;}
[\t ]+  word_count++;
. char_count++;
%%
void main(){
yyin=fopen("input.txt","r");
yylex();
printf("This file contains %d characters\n", char_count);
printf("This file contains %d words\n", word_count);
printf("This file contains %d lines\n", line_count);
}
int yywrap(){ return(1);}
```

OUTPUT:


This file contains 59 characters
This file contains 12 words
This file contains 5 lines
...Program finished with exit code 27
Press ENTER to exit console.

b)

Write a Lex program to take input from text file and count number of vowels and consonants.

SOURCE CODE:

input.txt:

Hello

Good Morning

This is my lex program

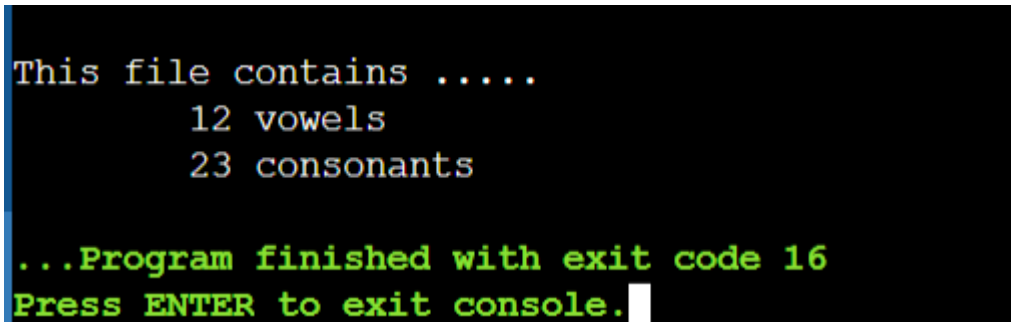
123456 677 34.676

56e56

d2.l:

```
%{
#include<stdio.h>
int consonants=0, vowels=0;
%}
%%
[aeiouAEIOU] {vowels++;}
[a-zA-Z] {consonants++;}
. ;
%%
void main(){
yyin=fopen("input.txt","r");
yylex();
printf("This file contains .....");
printf("\n\t%d vowels ",vowels);
printf("\n\t%d consonants ",consonants);
return 0;
}
int yywrap(){ return(1);}
```

OUTPUT:


This file contains .....
12 vowels
23 consonants
...Program finished with exit code 16
Press ENTER to exit console.

c)

Write a Lex program to print out all numbers from the given file.

SOURCE CODE:

input.txt:

Hello

Good Morning

This is my lex program

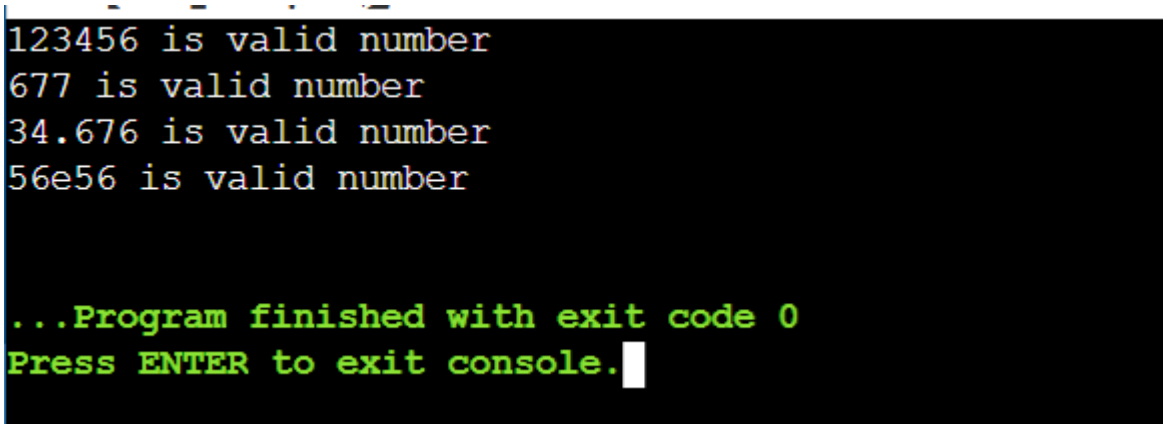
123456 677 34.676

56e56

d3.l:

```
%{
#include<stdio.h>
%}
digits [0-9]+
%%
digits(\.digits)?([eE][+-]?digits)? printf("%s is valid number\n", yytext);
\n ;
. ;
%%
void main(){
yyin=fopen("input.txt","r");
yylex();

}
int yywrap(){ return(1);}
```

OUTPUT:


123456 is valid number
677 is valid number
34.676 is valid number
56e56 is valid number
...Program finished with exit code 0
Press ENTER to exit console.

d)

Write a Lex program which adds line numbers to the given file and display the same into different file.

SOURCE CODE:

input.txt

Hello

Good Morning

This is my lex program

123456 677 34.676

56e56

d4.l:

```
%{
int line_number = 1;
%}
%%
.* {fprintf(yyout, "%d: %s",line_number,yytext);line_number++;}

%%
void main(){
yyin=fopen("input.txt","r");
yyout=fopen("op.txt","w");
yylex();
printf("done");
return 0;
}
int yywrap(){ return(1);}
```

OUTPUT:

```
d4.1: In function 'main':
d4.1:13:8: warning: 'return' with a value, in function returning void
d4.1:8:6: note: declared here
done

...Program finished with exit code 4
Press ENTER to exit console.
```

main.c | input.txt | op.txt

```
1: Hello
2: Good Morning
3: This is my lex program
4: 123456 677 34.676
5: 56e56
6
```

e)

Write a Lex program to printout all markup tags and HTML comments in file.

SOURCE CODE:

input.txt:

<html>

<head> Heer </head>

<body>

<!-- iehhfjs 122 -->

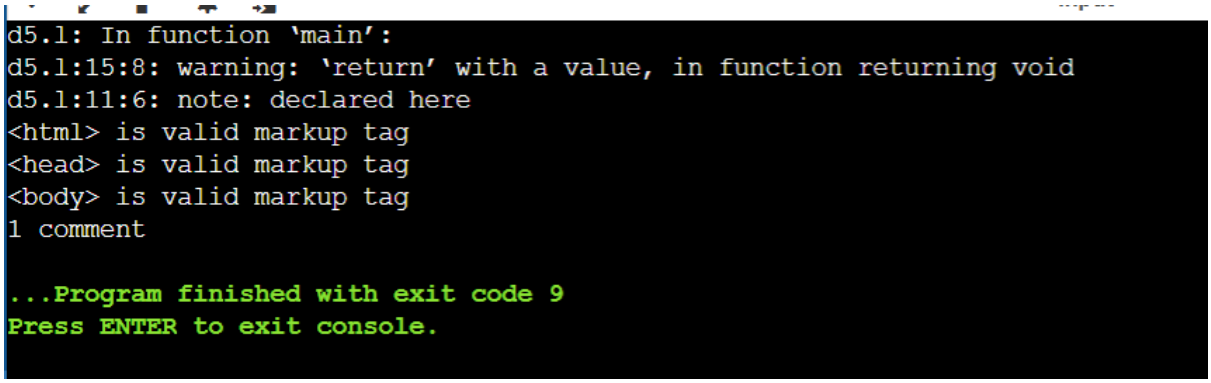
</body>

</html>

d5.l:

```
%{
#include<stdio.h>
int num = 0;
%}
%%
\<[a-zA-z0-9]+">" printf("%s is valid markup tag \n",yytext);
"<!--"(.|\n)*"-->" num++;
\n ;
. ;
%%
void main(){
yyin=fopen("input.txt","r");
yylex();
printf("%d comment", num);
return 0;
}
int yywrap(){ return(1);}
```

OUTPUT:


d5.1: In function 'main':
d5.1:15:8: warning: 'return' with a value, in function returning void
d5.1:11:6: note: declared here
<html> is valid markup tag
<head> is valid markup tag
<body> is valid markup tag
1 comment
...Program finished with exit code 9
Press ENTER to exit console.

Experiment 5

a)

Write a Lex program to count the number of C comment lines from a given C program. Also eliminate them and copy that program into separate file.

Sample code:

```c
#include <stdio.h>

// This is a single-line comment
int main() {
    int a = 5; /* Inline multi-line
                  comment */
    int b = 10; // Another comment
    printf("Sum = %d\n", a + b);
    /* Entire
       block
       comment */
    return 0;
}
```

Lex Code:

```
%{
#include <stdio.h>
int comment_count = 0;
FILE *out;
%}

%%
"//".*                          { comment_count++; /* skip single-line comment */ }
"/*"([^*]|\*+[^*/])*\*+"/"      { comment_count++; /* skip multi-line comment */ }
.|\n                           { fputc(yytext[0], out); }
%%

int main() {
    FILE *in = fopen("sample_input.c", "r");
    if (!in) {
        perror("Input file error");
        return 1;
    }
    out = fopen("cleaned_output.c", "w");
    if (!out) {
        perror("Output file error");
        return 1;
    }
    yyin = in;
    yylex();
    fclose(in);
    fclose(out);
    printf("Total comments removed: %d\n", comment_count);
    return 0;
}

int yywrap() {
    return 1;
}
```

Output:

```
#include <stdio.h>


int main() {
    int a = 5;
    int b = 10;
    printf("Sum = %d\n", a + b);

    return 0;
}
```

b)

Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program.

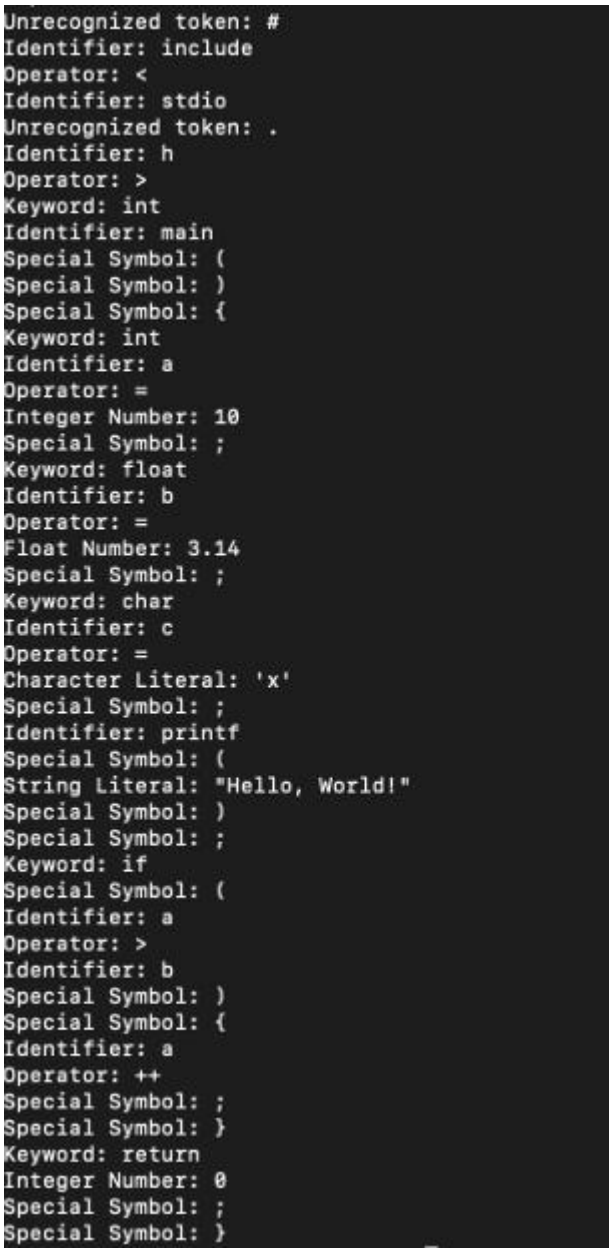
Sample Code:

```c
#include <stdio.h>

int main() {
    int a = 10;
    float b = 3.14;
    char c = 'x';
    printf("Hello, World!");
    if (a > b) {
        a++;
    }
    return 0;
}
```

Lex code:

```
 %{
#include <stdio.h>
%}
digit       [0-9]
alpha       [a-zA-Z]
id
          {alpha}({alpha}|{digit})*
int_const   {digit}+
float_const {digit}+"."{digit}+
string_lit  \"([^\\\"]|\\.)*\"
char_lit    \'([^\\\']|\\.)\'
%%
"auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"|"else"
|"enum"|"
 extern"|"float"|"for"|"goto"|"if"|"inline"|"int"|"long"|"register"|"restrict"
|"return"|"short
 "|"signed"|"sizeof"|"static"|"struct"|"switch"|"typedef"|"union"|"unsigned"|"
void"|"volatil
 e"|"while"    { printf("Keyword: %s\n", yytext); }
{id}                { printf("Identifier: %s\n", yytext); }
"=="|"!="|"<="|">="|"&&"|"||"|"++"|"--"|"+"|"-"|"*"|"/"|"="|"<"|">"
{ printf("Operator: %s\n", yytext); }
{float_const}       { printf("Float Number: %s\n", yytext); }
{int_const}         { printf("Integer Number: %s\n", yytext); }
{string_lit}        { printf("String Literal: %s\n", yytext); }
{char_lit}          { printf("Character Literal: %s\n", yytext); }
[{}();,]            { printf("Special Symbol: %s\n", yytext); }
[ \t\n]+
            { /* skip whitespace */ }
.                   { printf("Unrecognized token: %s\n", yytext); }
%%
int main() {
    yyin = fopen("sample_input.c", "r");
    if (!yyin) {
        perror("Could not open file");
        return 1;
    }
    yylex();
    fclose(yyin);
    return 0;
}
int yywrap() {
    return 1;
}
```

Output:


Unrecognized token: #
Identifier: include
Operator: <
Identifier: stdio
Unrecognized token: .
Identifier: h
Operator: >
Keyword: int
Identifier: main
Special Symbol: (
Special Symbol:)
Special Symbol: {
Keyword: int
Identifier: a
Operator: =
Integer Number: 10
Special Symbol: ;
Keyword: float
Identifier: b
Operator: =
Float Number: 3.14
Special Symbol: ;
Keyword: char
Identifier: c
Operator: =
Character Literal: 'x'
Special Symbol: ;
Identifier: printf
Special Symbol: (
String Literal: "Hello, World!"
Special Symbol:)
Special Symbol: ;
Keyword: if
Special Symbol: (
Identifier: a
Operator: >
Identifier: b
Special Symbol:)
Special Symbol: {
Identifier: a
Operator: ++
Special Symbol: ;
Special Symbol: }
Keyword: return
Integer Number: 0
Special Symbol: ;
Special Symbol: }

Experiment 6

Aim: Program to implement Recursive Descent Parsing in C.

Grammar:

E -> i E'

E' -> + i E' | ε

Code:

```c
#include <stdio.h>
#include <string.h>

const char *input;
int pos = 0;

void E();
void Eprime();

void error() {
    printf("❌ Syntax Error at position %d\n", pos);
    exit(1);
}

void match(char expected) {
    if (input[pos] == expected) {
        pos++;
    } else {
        error();
    }
}

void E() {
    if (input[pos] == 'i') {
        match('i');
        Eprime();
    } else {
        error();
    }
}

void Eprime() {
    if (input[pos] == '+') {
```

```
        match('+');
        if (input[pos] == 'i') {
            match('i');
            Eprime();
        } else {
            error();
        }
    }
    // else epsilon production: do nothing
}

int main() {
    char str[100];
    printf("Enter the input string (end with $): ");
    scanf("%s", str);
    input = str;

    E();

    if (input[pos] == '$') {
        printf("☑ Parsing successful.\n");
    } else {
        error();
    }

    return 0;
}
```

Experiment 7

A. Create Yacc and Lex specification files to recognises arithmetic expressions involving +, -, * and / .

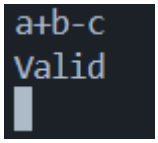
Lex.l

```
%{
#include "yacc.tab.h"
#include <stdlib.h>
#include <string.h>
%}

%%
[0-9]+                  { yylval = atoi(yytext); return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]*  { return IDENTIFIER; }
[+\-*/]                 { return yytext[0]; }
\n                      { return '\n'; }
[ \t]                   ;
.                       { return 0; }
%%

int yywrap() { return 1; }
```

yacc.y

```
%{
#include <stdio.h>
#include <stdlib.h>

int yylex(void);
int yyerror(char *s);
%}

%token NUMBER IDENTIFIER

%left '+' '-'
%left '*' '/'

%%
stmt: expr '\n' { printf("Valid\n"); }
    ;

expr: expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | NUMBER
    | IDENTIFIER
    ;
%%

int main() {
    return yyparse();
}

int yyerror(char *s) {
    printf("Invalid\n");
    return 0;
}
```

Output:



B. Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments.

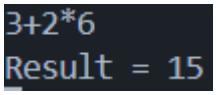
Lex.l

```
%{
#include "yacc.tab.h"
%}

%%
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
[+\-*/\n]   { return yytext[0]; }
[ \t]       ;
%%

int yywrap() { return 1; }

```

yacc.y

```
%{
#include <stdio.h>
#include <stdlib.h>

int yylex(void);
int yyerror(char *s);
%}

%token NUMBER

%left '+' '-'
%left '*' '/'

%%
stmt: expr '\n' { printf("Result = %d\n", $1); }
    ;

expr: expr '+' expr { $$ = $1 + $3; }
    | expr '-' expr { $$ = $1 - $3; }
    | expr '*' expr { $$ = $1 * $3; }
    | expr '/' expr {
        if ($3 == 0) {
            printf("Error: Divide by zero\n");
            exit(1);
        }
        $$ = $1 / $3;
      }
    | NUMBER
    ;
%%

int main() {
    return yyparse();
}

int yyerror(char *s) {
    printf("Error: %s\n", s);
    return 0;
}
```

Output:


3+2*6
Result = 15

C. Create Yacc and Lex specification files are used to convert infix expression to postfix expression.

Lex.l

```
%{
#include "yacc.tab.h"
#include <stdlib.h>
#include <string.h>
%}

%%
[0-9]+                  { yylval.num = atoi(yytext); return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]*  { yylval.id = strdup(yytext); return IDENTIFIER; }
[+\-*/()\n]             { return yytext[0]; }
[ \t]+                  ;
.                       { printf("Invalid character: %s\n", yytext); return -1; }
%%

int yywrap() { return 1; }

```

yacc.y

```
%{
#include <stdio.h>
#include <stdlib.h>

int yylex(void);
int yyerror(char *s);
%}

%union {
    int num;
    char* id;
}

%token <num> NUMBER
%token <id> IDENTIFIER

%%
stmt: expr '\n' { printf("\n"); }
    ;

expr: expr '+' term   { printf("+ "); }
    | expr '-' term   { printf("- "); }
    | term
    ;

term: term '*' factor { printf("* "); }
    | term '/' factor { printf("/ "); }
    | factor
    ;

factor: '(' expr ')'
      | NUMBER        { printf("%d ", $1); }
      | IDENTIFIER    { printf("%s ", $1); free($1); }
      ;
%%

int main() {
    return yyparse();
}

int yyerror(char *s) {
    fprintf(stderr, "Error: %s\n", s);
    return 0;
}

```

Output: