LAB MANUAL

of

**Compiler Design Laboratory**

**(CSE606)**

**Bachelor of Technology (CSE)**

By

Dhyan Patel (22000936)



Department of Computer Science and Engineering

School Engineering and Technology

Navrachana University, Vadodara

Autumn Semester

(2024-2025)

**TABLE OF CONTENT**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No** | **Experiment Title** | | |
| 1 |  | 1. Write a program to recognize strings starts with ‘a’ over {a, b}. 2. Write a program to recognize strings end with ‘a’. 3. Write a program to recognize strings end with ‘ab’. Take the input from text file. 4. Write a program to recognize strings contains ‘ab’. Take the input from text file. |
| 2 |  | 1. Write a program to recognize the valid identifiers and keywords. 2. Write a program to recognize the valid operators. 3. Write a program to recognize the valid number. 4. Write a program to recognize the valid comments. 5. Program to implement Lexical Analyzer. |
| 3 |  | To Study about Lexical Analyzer Generator (LEX) and Flex(Fast Lexical Analyzer) |
| 4 |  | Implement following programs using Lex.  a. Write a Lex program to take input from text file and count no of characters, no. of lines & no. of words.  b. Write a Lex program to take input from text file and count number of vowels and consonants.  c. Write a Lex program to print out all numbers from the given file.  d. Write a Lex program which adds line numbers to the given file and display the same into different file.  e. Write a Lex program to printout all markup tags and HTML comments in file. |
| 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.  b. Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program. |
| 6 |  | Program to implement Recursive Descent Parsing in C. |
| 7 |  | a. To Study about Yet Another Compiler-Compiler(YACC).  b. Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, \* and / .  c. Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments.  d. Create Yacc and Lex specification files are used to convert infix expression to postfix expression. |

**AIM 1:**

**a)**

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

**SOURCE CODE:**

#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");

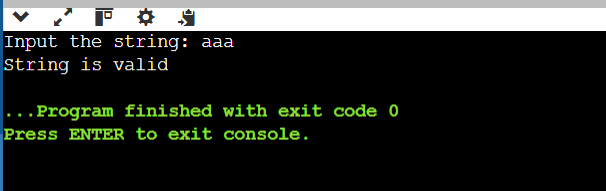
   else if (state==2 || state==0) printf("String is invalid");

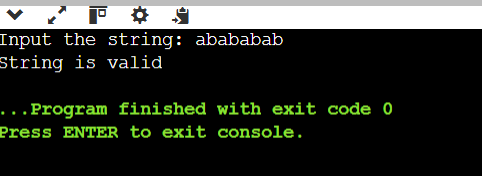
   else if (state==3) printf("String is not recognized");

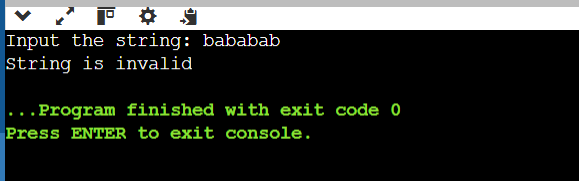
   return 0;

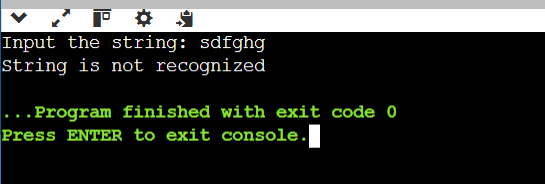
}

**OUTPUT:**









**b)**

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

**SOURCE CODE:**

#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++;

   }

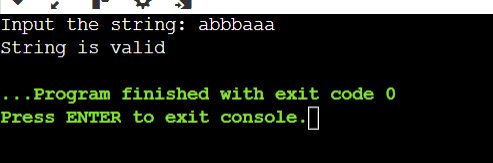
   if (state==1) printf("String is valid");

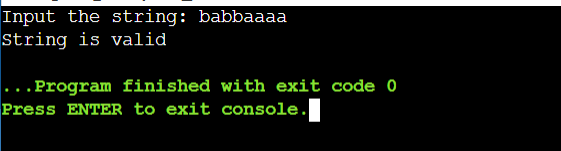
   else if (state==0) printf("String is invalid");

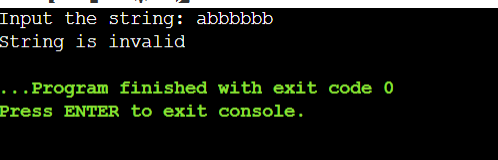
   return 0;

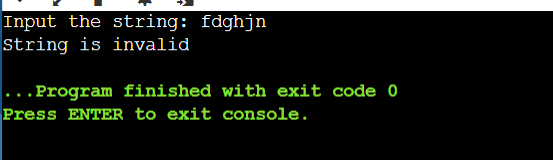
}

**OUTPUT:**

****

****

****

****

**c)**

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

**SOURCE CODE:**

#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 {

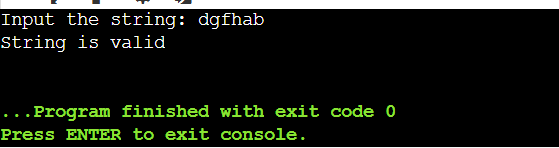
       printf("String is invalid\n");

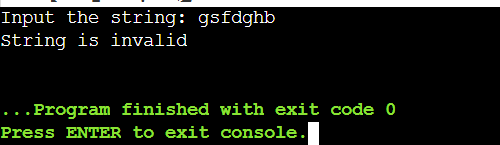
   }

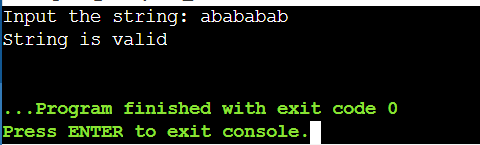
   return 0;

}

**OUTPUT:**







**d)**

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

**SOURCE CODE:**

#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 {

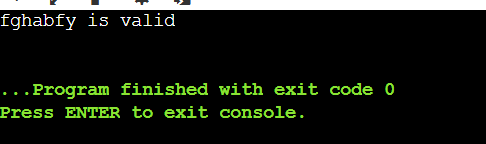
        printf("%s is invalid\n",input);

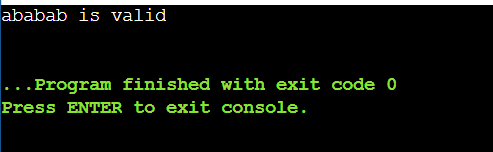
    }

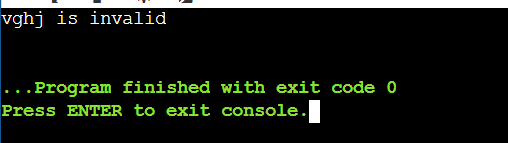
    return 0;

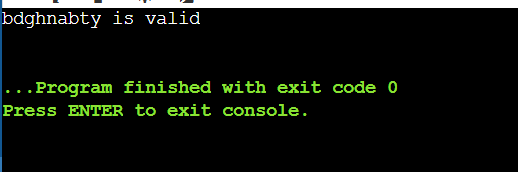
}

**OUTPUT:**









**AIM: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:

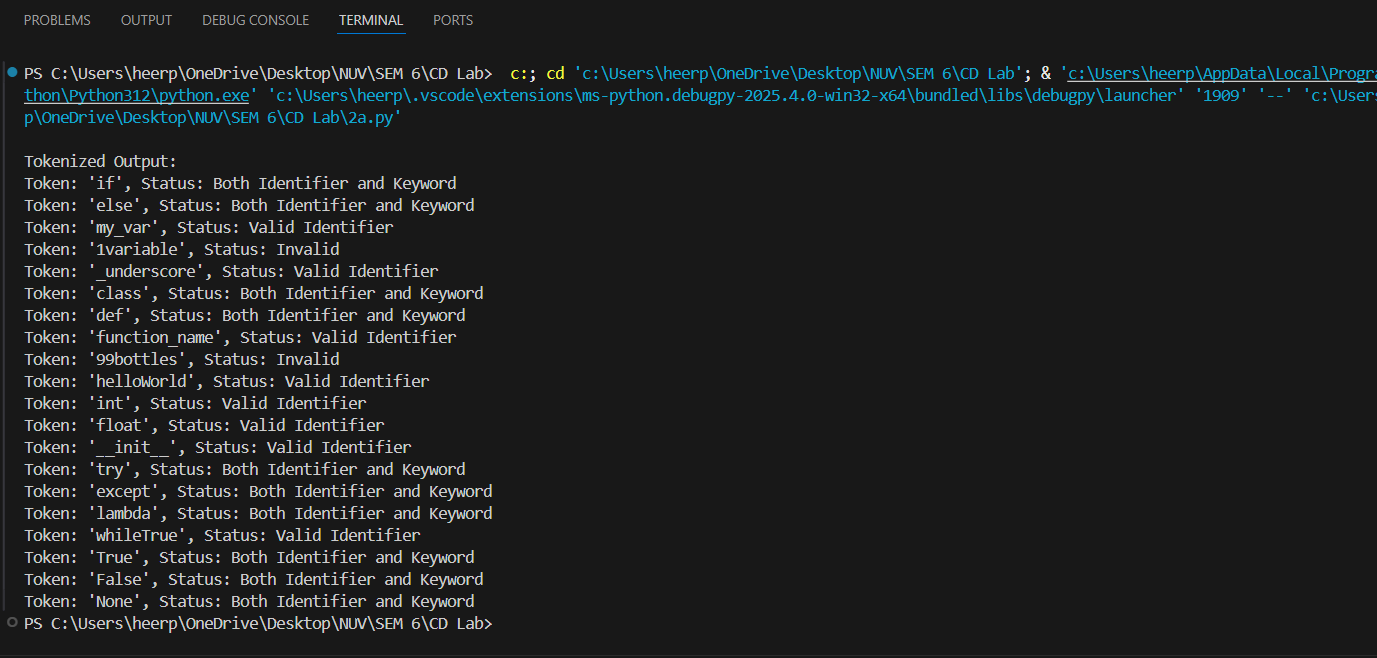
            file.write(f"Token: '{token}', Status: {status}\n")

    print("\nTokenized Output:")

    for token, status in results:

        print(f"Token: '{token}', Status: {status}")

**OUTPUT:**



**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:

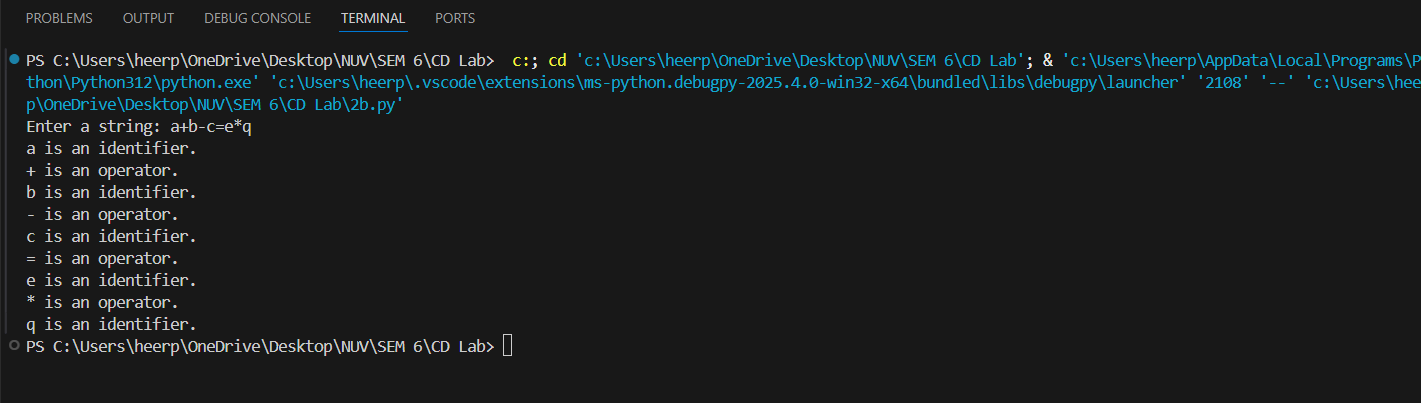
    if char in operators:

        print(f"{char} is an operator.")

    elif char.isalnum():

        print(f"{char} is an identifier.")

**OUTPUT:**



**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:

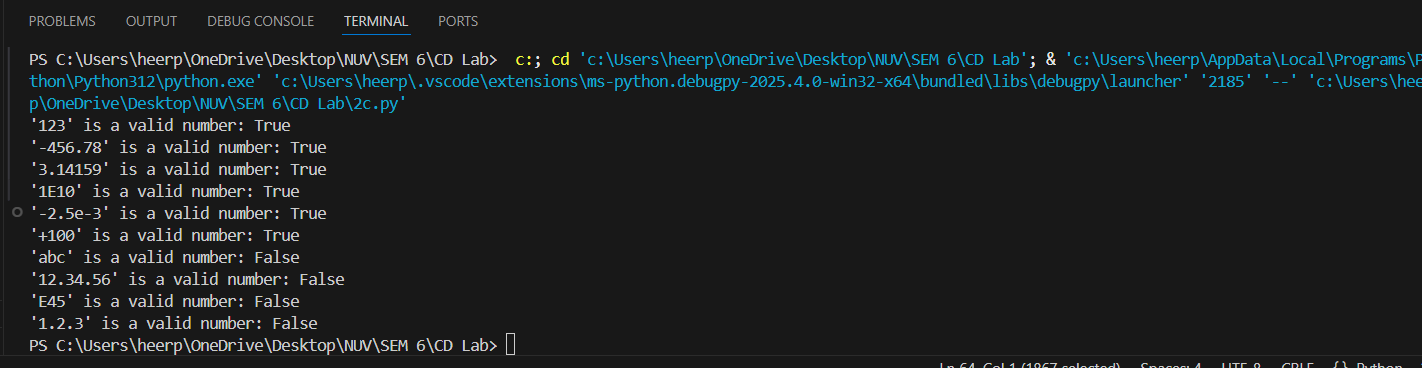
                number = line.strip()

                print(f"'{number}' is a valid number: {is\_valid\_number\_fsm(number)}")

    except FileNotFoundError:

        print("Error: 'numbers.txt' file not found.")

**OUTPUT:**



**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:

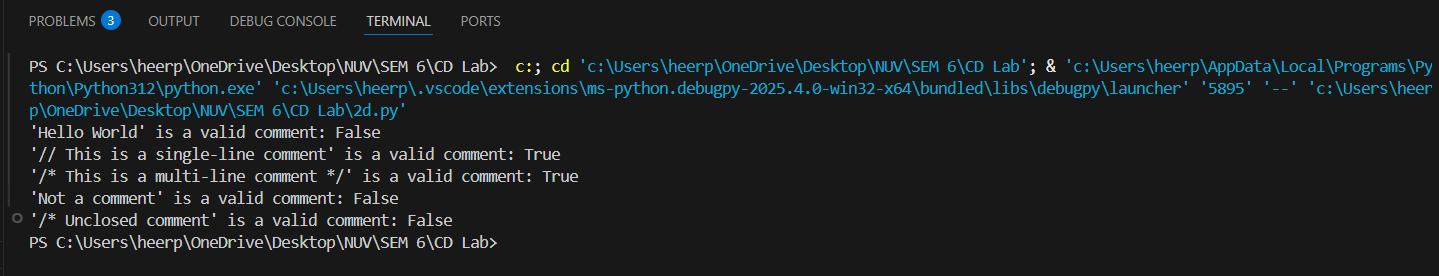
                line = line.strip()

                print(f"'{line}' is a valid comment: {is\_valid\_comment(line)}")

    except FileNotFoundError:

        print("Error: 'comments.txt' file not found.")

**OUTPUT:**



**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 = ""

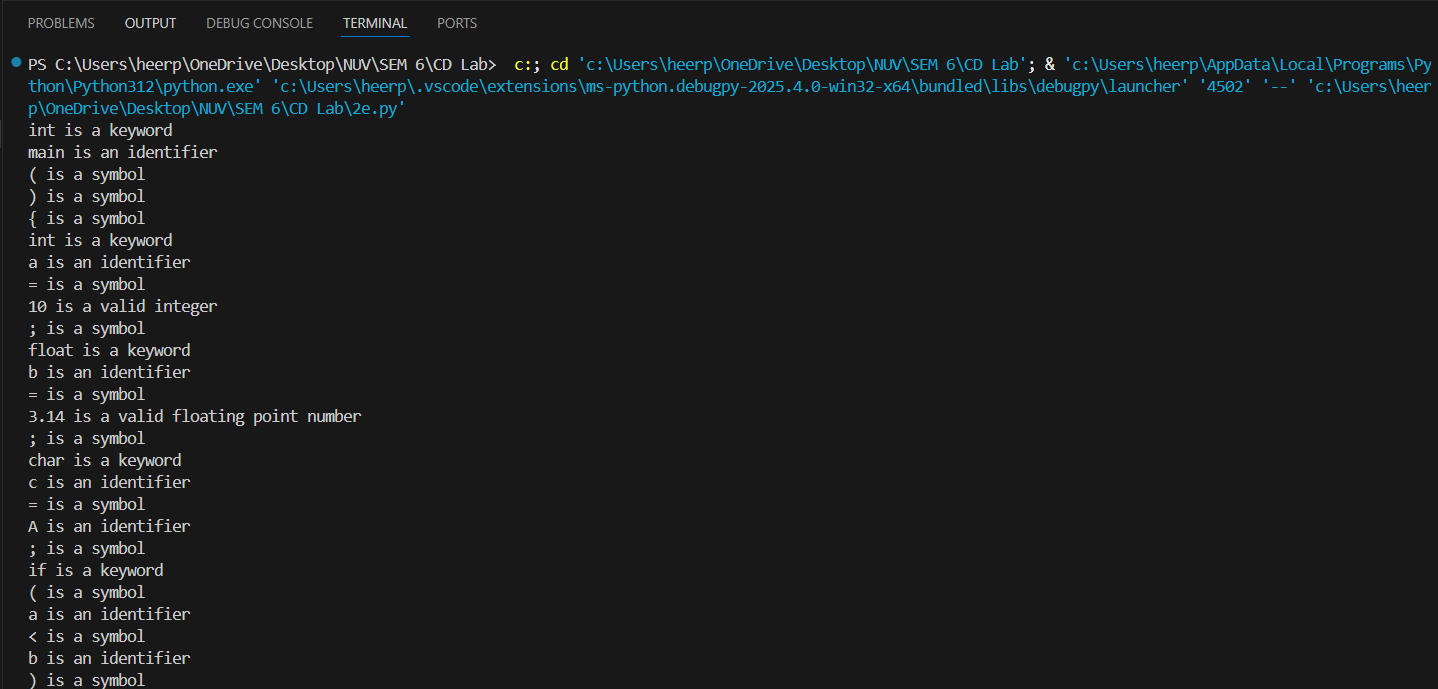
                state = 0

                f -= 1

        f += 1

lexer("input2.txt")

**OUTPUT:**





**AIM 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.

**AIM 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);

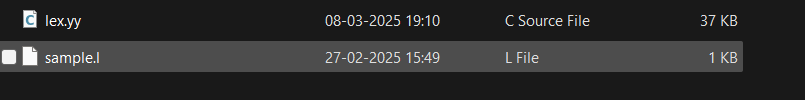
printf("This file contains %d words\n", word\_count);

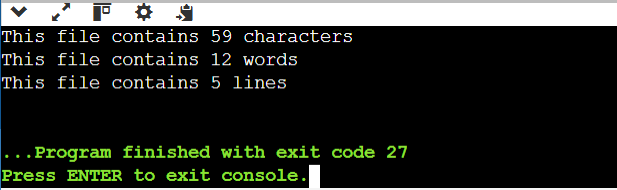
printf("This file contains %d lines\n", line\_count);

}

int yywrap(){ return(1);}

**OUTPUT:**





**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);

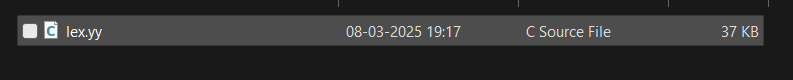
printf("\n\t%d consonants ",consonants);

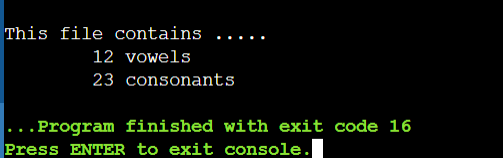
return 0;

}

int yywrap(){ return(1);}

**OUTPUT:**





**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(){

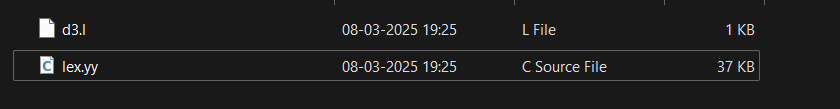
yyin=fopen("input.txt","r");

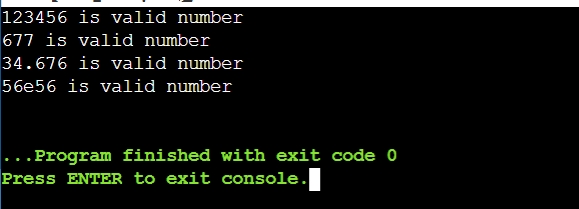
yylex();

}

int yywrap(){ return(1);}

**OUTPUT:**





**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();

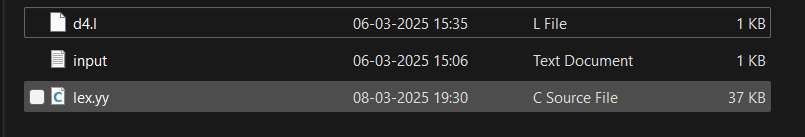
printf("done");

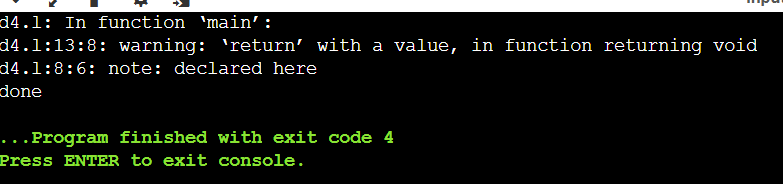
return 0;

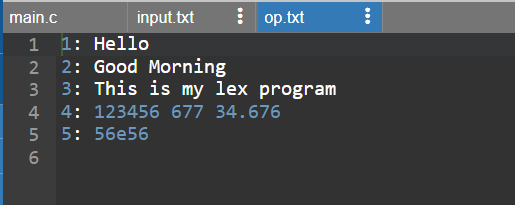
}

int yywrap(){ return(1);}

**OUTPUT:**







**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();

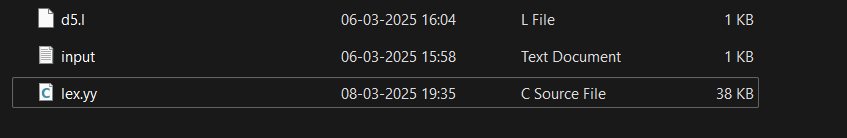
printf("%d comment", num);

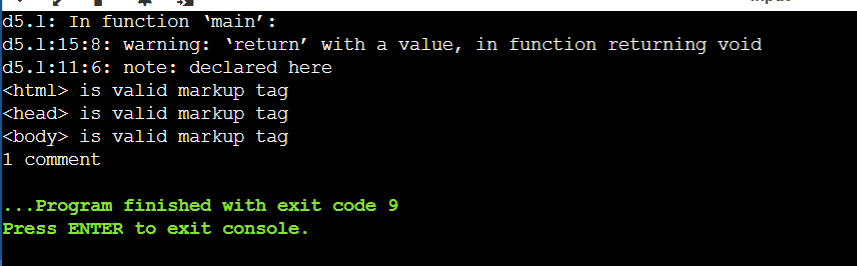
return 0;

}

int yywrap(){ return(1);}

**OUTPUT:**





**AIM-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.

**SOURCE CODE:**

**Code.txt**

#include <stdio.h>

int main() {

// This is a single-line comment

int a = 5; /\* This is an inline block comment \*/

/\*

This is a

multi-line block comment

\*/

printf("Hello, World!\n"); // Print message

return 0;

}

**aim6.l**

%{

#include <stdio.h>

int c = 0;

extern FILE \*yyin, \*yyout;

%}

%%

"//".\* { c++; /\* skip line comment \*/ }

"/\*"([^\*]|\\*+[^\*/])\*"\*"+"/" { c++; /\* skip block comment \*/ }

.|\n { fprintf(yyout, "%s", yytext); }

%%

int main() {

yyin = fopen("code.txt", "r");

yyout = fopen("output.txt", "w");

yylex();

printf("%d comments\n", c);

return 0;

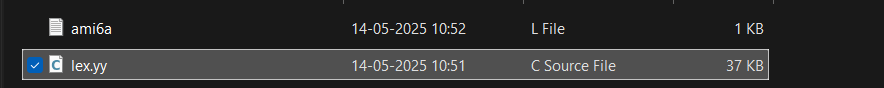
}

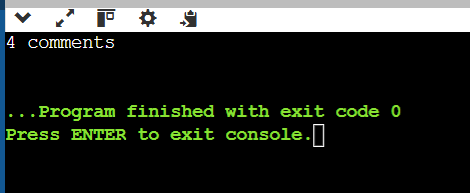
int yywrap() {

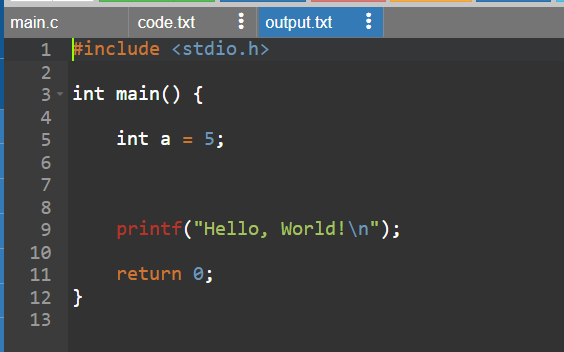
return 1;

}

**OUTPUT:**







**b)**

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

**SOURCE CODE:**

**code.txt:**

#include <stdio.h>

int main() {

// This is a single-line comment

printf("Hello, World!\n");

/\* This is

a multi-line

comment \*/

int x = 10; // Another comment

return 0;

}

**5b.l:**

%{

#include<stdio.h>

%}

%%

if|else|while|do|switch|case|return|int {printf("<%s, Keyword>\n", yytext);}

[a-zA-Z\_][a-zA-Z0-9]\* {printf("<%s, Identifier>\n", yytext);}

[0-9]+(\.[0-9]+)?([Ee][+-]?[0-9]+)? {printf("<%s, Number>\n", yytext);}

"!"|"@"|"8"|"&"|"^"|"%"|")"|"("|"<"|">"|";"|","|"{"|"}"|"="|"." {printf("<%s, special symbol>\n", yytext);}

[ \t\n]+ ;

"/\*"[^\*/]\*"\*/"  ;

"//"[^\n]+ ;

\"[^\"]+\" {printf("<%s, string constant>\n", yytext);}

. printf("%s Not Recognised\n", yytext);

%%

int main()

{

yyin = fopen("code.txt", "r");

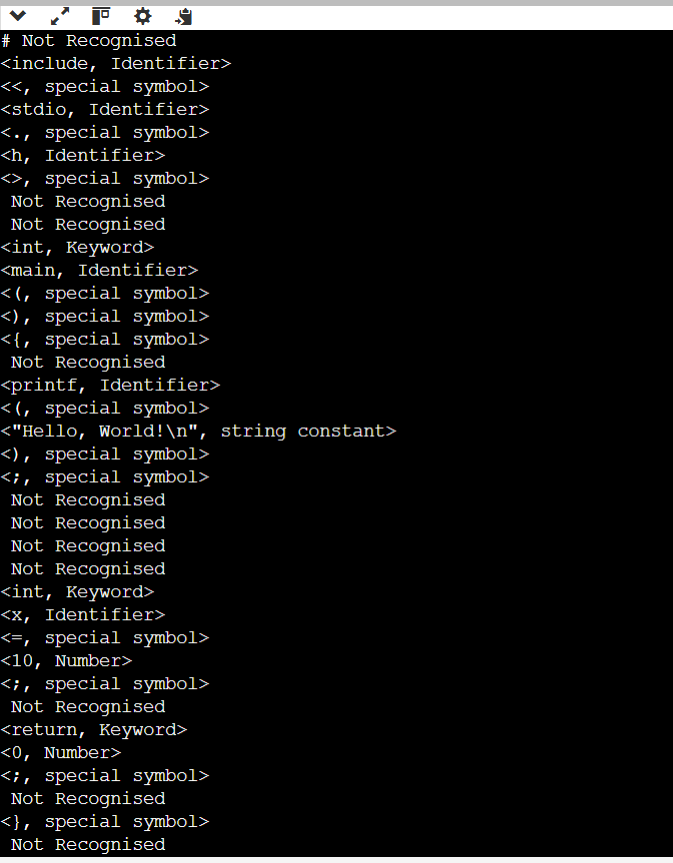
yylex();

return 0;

}

int yywrap() {return(1);}

**OUTPUT:**



**AIM – 6**

Program to implement Recursive Descent Parsing in C.

**SOURCE CODE:**

#include<stdio.h>

#include<stdlib.h>

/\*

E-> iE\_

E\_-> +iE\_ / -iE\_ / epsilon

\*/

char s[20];

int i=1;

char l;

int match(char t)

{

if(l==t){

l=s[i];

i++; }

else{

printf("Sytax error");

exit(1);}

}

int E\_()

{

if(l=='+'){

match('+');

match('i');

E\_(); }

else if(l=='-'){

match('-');

match('i');

E\_(); }

else

return(1);

}

int E()

{

if(l=='i'){

match('i');

E\_(); }

}

int main()

{

printf("\n Enter the set of characters to be checked :");

scanf("%s",&s);

l=s[0];

E();

if(l=='$')

{

printf("Success \n");

}

else{

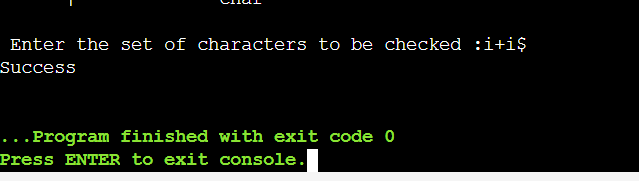
printf("syntax error");

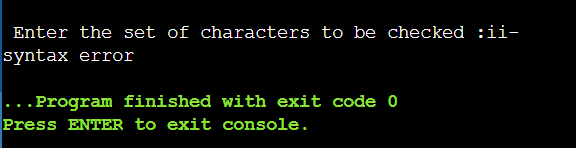
}

return 0;

}

**OUTPUT:**





**AIM – 7**

**a)**

To Study about Yet Another Compiler-Compiler(YACC).

**YACC – Yet Another Compiler-Compiler**

**Introduction**

* **YACC** is a **parser generator** developed by **Stephen C. Johnson** at AT&T Bell Labs.
* It automates the creation of a **syntax analyzer** (parser) based on a **context-free grammar (CFG)**.
* Typically used with **Lex**, which handles lexical analysis (tokenization).

**Purpose of YACC**

* Translates **grammar rules** into a **C program** that parses the input.
* Useful for building **compilers**, **interpreters**, and **language translators**.
* Converts a sequence of tokens (from Lex) into meaningful structures by checking the syntax.

**Working of YACC**

1. **Input**: Grammar written in YACC syntax in a .y file.
2. **YACC processes this file** and generates y.tab.c (a C source file for parsing).
3. The file is compiled along with Lex output (lex.yy.c) to create an executable.
4. When the program runs, it calls yyparse(), which processes input based on grammar rules.

**Key Components**

| **Component** | **Description** |
| --- | --- |
| %token | Declares tokens (terminal symbols) |
| %% | Separates sections |
| **Grammar rules** | Rules in BNF form (e.g., expr : expr '+' term) |
| **Semantic actions** | C code within {} executed when a rule matches |
| yyparse() | Parser function generated by YACC |
| yyerror() | Error-handling function |
| yylval | Variable for passing values between Lex and YACC |

**Advantages of YACC**

* Automatically handles syntax checking.
* Simplifies writing parsers for new languages.
* Flexible and extensible.
* Well-integrated with Lex.

**Limitations**

* Cannot parse all grammars (e.g., ambiguous or context-sensitive grammars).
* Requires understanding of grammar and parsing concepts.
* Original YACC only supports **LALR(1)** parsing.

**Conclusion**

YACC is a powerful tool that automates the generation of parsers. When used with Lex, it enables efficient and organized compiler development. Understanding YACC is essential for building any system that needs to interpret or validate complex language input.

**b)**

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

**SOURCE CODE:**

**sampleL.l**

%{

#include <stdlib.h>

void yyerror(char \*);

#include "sampleY.tab.h"

%}

%%

[0-9]+ return NUM;

[a-zA-Z\_][a-zA-Z0-9\_]\* return id;

[-+\*\n] return \*yytext;

[ \t] ;

. yyerror("invalid character");

%%

int yywrap() {

return 1;

}

**sampleY.y**

%{

#include<stdio.h>

int yylex(void);

void yyerror(char \*);

%}

%token NUM

%token id

%%

S : E '\n' { printf("valid syntax");return 0; }

E : E '+' E { }

| E '-' E { }

| NUM { }

| id { }

%%

void yyerror(char \*s) {

fprintf(stderr, "%s\n", s);

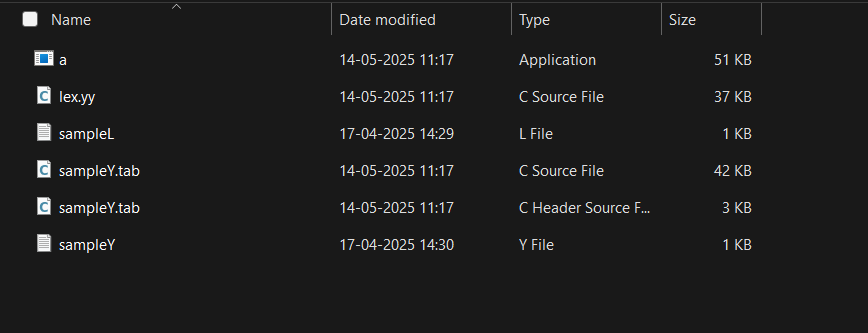
}

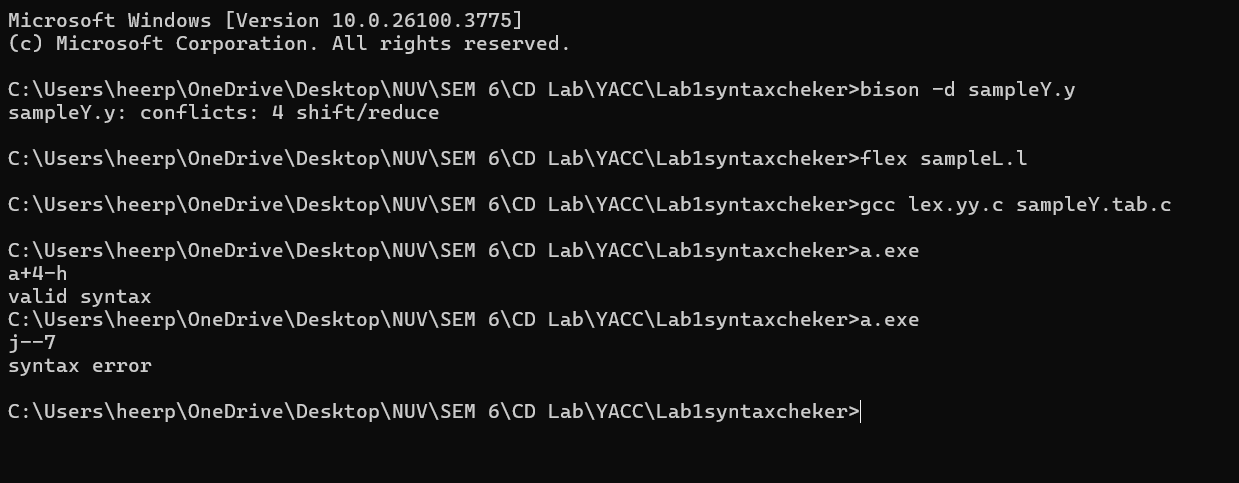
int main() {

yyparse();return 0;

}

**OUTPUT:**





**c)**

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

**SOURCE CODE:**

**sampleL.l**

%{

#include <stdlib.h>

void yyerror(char \*);

#include "sampleY.tab.h"

%}

%%

[0-9]+ {yylval = atoi(yytext); return NUM;}

[-+\*\n] {return \*yytext;}

[ \t] ; { }

. yyerror("invalid character");

%%

int yywrap() {

return 0;

}

**sampleY.y**

%{

#include<stdio.h>

int yylex(void);

void yyerror(char \*);

%}

%token NUM

%%

S : E '\n' { printf("%d\n", $1); return(0); }

E : E '+' T {$$ =$1 + $3; }

| E '-' T {$$ = $1 - $3; }

| T {$$ = $1; }

T : T '\*' F {$$ = $1 \* $3; }

| F {$$ = $1; }

F: NUM {$$ = $1; }

%%

void yyerror(char \*s) {

printf("%s\n", s);

}

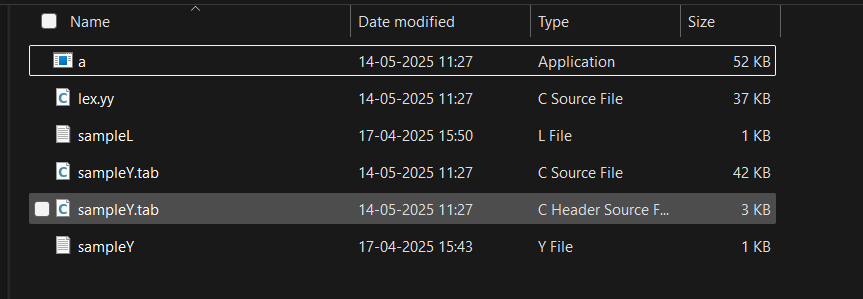
int main() {

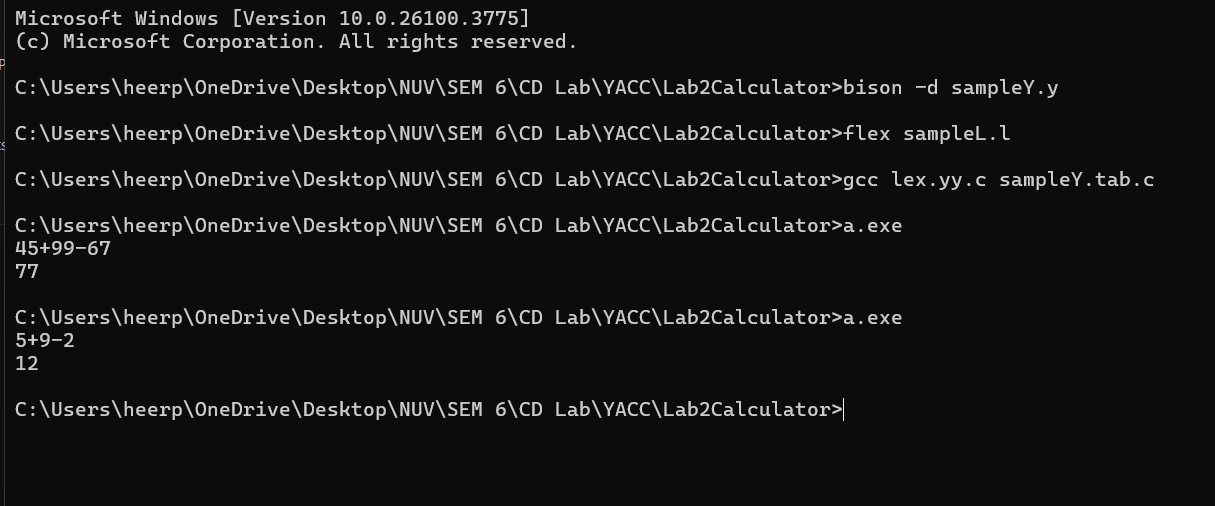
yyparse();

return 0;

}

**OUTPUT:**





**d)**

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

**SOURCE CODE:**

**b.l**

%{

#include <stdlib.h>

#include "b.tab.h"

void yyerror(char \*);

%}

%%

[0-9]+ { yylval.num = atoi(yytext); return INTEGER; }

[A-Za-z\_][A-Za-z0-9\_]\* { yylval.str = yytext; return ID; }

[-+;\n\*] { return \*yytext; }

[ \t] ;

. yyerror("invalid character");

%%

int yywrap() {

return 1;

}

**b.y**

%{

#include <stdio.h>

int yylex(void);

void yyerror(char \*);

%}

%union {

char \*str;

int num;

}

%token <num> INTEGER

%token <str> ID

%%

S: E '\n' {printf("\n");}

E: E '+' T { printf("+ "); }

| E '-' T { printf("- "); }

| T { }

T: T '\*' F { printf("\* "); }

| F { }

F: INTEGER { printf("%d ",$1);}

| ID { printf("%s ",$1 );}

%%

void yyerror(char \*s) {

printf("%s\n", s);

}

int main() {yyparse();return 0;}

**OUTPUT:**

