A
Assignment of
"Compiler Design Laboratory"
Is Submitted to

School of Engineering and Technology

Toward the fulfilment of the requirements of the Subject

Compiler Design Laboratory (CSE606)

SUBMITTED BY

VRAJ SHAH - 22000776

Subject In-Charge:- Prof. Vaibhavi Patel

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
SCHOOL OF ENGINEERING AND TECHNOLOGY
BHAYLI, VASNA-BHAYLI MAIN ROAD VADODARA

TABLE OF CONTENT

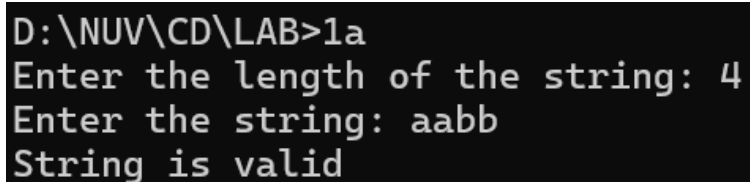| Sr. No	Experiment Title
1	a) Write a program to recognize strings starts with 'a' over {a, b}. b) Write a program to recognize strings end with 'a'. c) Write a program to recognize strings end with 'ab'. Take the input from text file. d) Write a program to recognize strings contains 'ab'. Take the input from text file.
2	a) Write a program to recognize the valid identifiers. b) Write a program to recognize the valid operators. c) Write a program to recognize the valid number. d) Write a program to recognize the valid comments. e) 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.

Practical – 1

AIM a: Write a program to recognize strings starts with 'a' over {a, b}.

PROGRAM CODE:

```c
#include  <stdio.h>
#include <stdlib.h>
void main()
{
int n, i = 0, state = 0;
printf("Enter the length of the string: ");
scanf("%d", &n);
char input[n];
printf("Enter 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 == 0 || state == 2)

{

printf("String is invalid");

}

else if (state == 1)

{

printf("String is valid");

}

else

printf("String is not recognized");

}
```

OUTPUT:

```
D:\NUV\CD\LAB>1a
Enter the length of the string: 4
Enter the string: aabb
String is valid
```

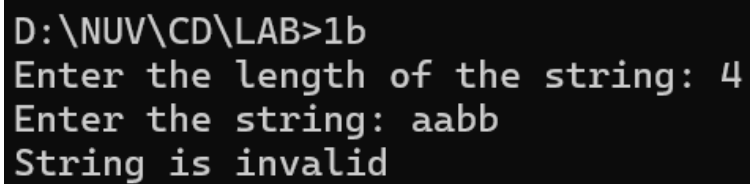**AIM b:** Write a program to recognize strings end with 'a'.

PROGRAM CODE:

```c
#include  <stdio.h>
#include <stdlib.h>
void main()
{
int n, i = 0, state = 0;
printf("Enter the length of the string: ");
scanf("%d", &n);
char input[n];
printf("Enter 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++;
}
```

Btech cse (sem-6)

if (state == 0)

{

printf("String is invalid");

}

else if (state == 1)

{

printf("String is valid");

}

}

OUTPUT:

```
D:\NUV\CD\LAB>1b
Enter the length of the string: 4
Enter the string: aabb
String is invalid
```

if (state == 0)

{

Btech cse (sem-6)

AIM c: Write a program to recognize strings end with 'ab'. Take the input from text file.
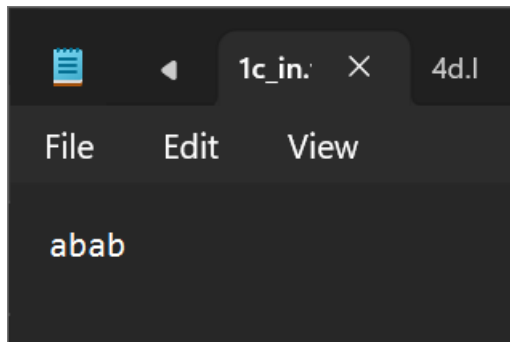
PROGRAM CODE:

```c
#include <stdio.h>
void main()
{
int state = 0, i = 0;
FILE *fptr;
fptr = fopen("1c_in.txt", "r");
char input[100];
fgets(input, 100, fptr);
printf("Input string: %s", input);
fclose(fptr);
while (input[i] != '\0')
{
switch (state)
{
case 0:
if (input[i] == 'a')
{
state = 1;
}
else
{
state = 0;
}
break;
case 1:
if (input[i] == 'b')
{
state = 2;
}
else if (input[i] == 'a')
```

```
{
state = 1;
}
else
{
state = 0;
}
break;

case 2:
if (input[i] == 'a')
{
state = 1;
}
else
{
state = 0;
}
break;
}
i++;
}
if (state == 2)
{
printf("\nString is valid");
}
else
{
printf("\nString is invalid");
}
}
```

INPUT File:

Btech cse (sem-6)

File   Edit   View

abab

OUTPUT:

```
D:\NUV\CD\LAB>1c
Input string: abab
String is valid
D:\NUV\CD\LAB>
```

Btech cse (sem-6)

AIM d: Write a program to recognize strings contains 'ab'. Take the input from text file.

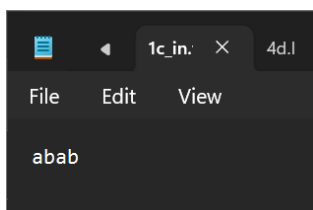**PROGRAM CODE:**

```
#include <stdio.h>
void main()
{
int state = 0, i = 0;
FILE *fptr;
fptr = fopen("1c_in.txt", "r");
char input[100];
fgets(input, 100, fptr);
printf("Input string: %s", input);
fclose(fptr);
while (input[i] != '\0')
{
switch (state)
{
case 0:
if (input[i] == 'a')
{
state = 1;
}
else
{
state = 0;
}
break;
case 1:
if (input[i] == 'b')
{
state = 2;
}
else if (input[i] == 'a')
```

```
{
state = 1;
}
else
{
state = 0;
}
break;

case 2:
state = 2;
break;
}
i++;
}
if (state == 2)
{
printf("\nString is valid");
}
else
{
printf("\nString is invalid");
}
}
```

INPUT File:

OUTPUT:

Btech cse (sem-6)

Practical – 2

AIM a: Write a program to recognize the valid identifiers.

PROGRAM CODE:

```c
#include  <stdio.h>
#include <stdlib.h>
#include <ctype.h>
void main()
{
int state = 0, i = 0;
FILE *fptr;
fptr = fopen("2a_in.txt", "r");
char input[100];
while (fgets(input, 100, fptr) != NULL)
{
printf("Input string: %s", input);
i = 0;
state = 0;
while (input[i] != '\0' && input[i] != '\n')
{
switch (state)
{
case 0:
if (input[i] == 'i')
{
state = 1;
}
else if (input[i] == '_' || isalpha(input[i]))
{
state = 4;
}
else
{
```
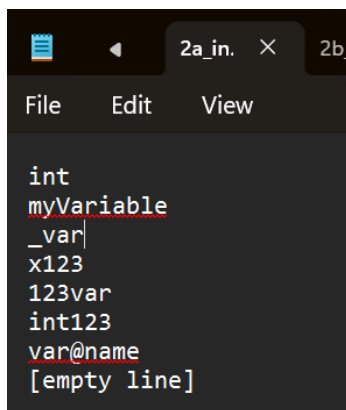
```
state = 5;
}
break;
case 1:
if (input[i] == 'n')
{
state = 2;
}
else if (input[i] == '_' || isalpha(input[i]))
{
state = 4;
}
else
{
state = 5;
}
break;
case 2:
if (input[i] == 't')
{
state = 3;
}
else if (input[i] == '_' || isalpha(input[i]))
{
state = 4;
}
else
{
state = 5;
}
break;
case 3:
if (isalpha(input[i]) || isdigit(input[i]) || input[i] == '_')
```

12

```
{
state = 4;
}
else
{
state = 5;
}
break;
case 4:
if (isalpha(input[i]) || isdigit(input[i]) || input[i] == '_')
{
state = 4;
}
else
{
state = 5;
}
break;
}
i++;
}
if (state == 4)
{
printf("\nString is Identifier\n");
}
else if (state == 3)
{
printf("\nString is a valid Keyword\n");
}
else if (state == 5)
{
printf("\nString is invalid string\n");
}
```

13

else

{

printf("\nString is empty\n");

}

}

fclose(fptr);

}

INPUT File:

OUTPUT:

Btech cse (sem-6)

AIM b: Write a program to recognize the valid operators.

PROGRAM CODE:

```
#include <stdio.h>

int main()

{

char input[100];

int state = 0, i = 0;

FILE *file = fopen("2b_in.txt", "r");

if (file == NULL)

{

printf("Error opening file.\n");

return 1;

}

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

fclose(file);

while (input[i] != '\0')

{

switch (state)

{

case 0:

if (input[i] == '+') state = 1;

else if (input[i] == '-') state = 5;

else if (input[i] == '*') state = 9;

else if (input[i] == '/') state = 12;

else if (input[i] == '%') state = 15;

else if (input[i] == '&') state = 18;

else if (input[i] == '|') state = 21;

else if (input[i] == '<') state = 24;

else if (input[i] == '>') state = 28;

else if (input[i] == '!') state = 32;

else if (input[i] == '~') state = 34;

else if (input[i] == '^') state = 35;

else if (input[i] == '=') state = 36;
```

```
break;
case 1:
if (input[i] == '+')
{
state = 2;
printf("++ unary operator");

}
else if (input[i] == '=')
{
state = 3;
printf("+= assignment operator");
}
else
{
state = 4;
printf("+ arithmetic operator");
}
break;
case 5:
if (input[i] == '-')
{
state = 6;
printf("-- unary operator");
}
else if (input[i] == '=')
{
state = 7;
printf("-= assignment operator");
}
else
{
state = 8;
printf("- arithmetic operator");
```

```
}
break;
case 9:
if (input[i] == '=')
{
state = 10;
printf("*= assignment operator");
}
else
{
state = 11;
printf("* arithmetic operator");
}
break;
case 12:

if (input[i] == '=')
{
state = 13;
printf("/= assignment operator");
}
else
{
state = 14;
printf("/ arithmetic operator");
}
break;
case 15:
if (input[i] == '=')
{
state = 16;
printf("%%= assignment operator");
}
else
```

Btech cse (sem-6)

```
{
state = 17;
printf("%% arithmetic operator");
}
break;
case 18:
if (input[i] == '&')
{
state = 19;
printf("&& Logical operator");
}
else
{
state = 20;
printf("& Bitwise operator");
}
break;
case 21:
if (input[i] == '|')
{
state = 22;
printf("|| Logical operator");
}
Else
{
state = 23;
printf("| Bitwise operator");
}
break;
case 24:
if (input[i] == '<')
{
state = 25;
```

```c
printf("<< Bitwise operator");
}
else if (input[i] == '=')
{
state = 27;
printf("<= Relational operator");
}
else
{
state = 26;
printf("< Relational operator");
}
break;
case 28:
if (input[i] == '>')
{
state = 29;
printf(">> Bitwise operator");
}
else if (input[i] == '=')
{
state = 30;
printf(">= Relational operator");
}
else
{
state = 31;
printf("> Relational operator");
}
break;
case 32:
if (input[i] == '=')
{
```

```
state = 33;
printf("!= Relational operator");
}
break;
case 36:
if (input[i] == '=')
{
state = 37;
printf("== Relational operator");
}
break;
default:
break;
}
i++;
}
printf("\nState is %d\n", state);
switch (state)
{
case 1: break;
case 5: printf("- arithmetic operator\n"); break;
case 9: printf("* arithmetic operator\n"); break;
case 12: printf("/ arithmetic operator\n"); break;
case 15: printf("%% arithmetic operator\n"); break;
case 18: printf("& Bitwise operator\n"); break;
case 21: printf("| Bitwise operator\n"); break;
case 24: printf("< Relational operator\n"); break;
case 28: printf("> Relational operator\n"); break;
case 32: printf("! Logical operator\n"); break;
case 34: printf("~ Bitwise operator\n"); break;
case 35: printf("^ Bitwise operator\n"); break;
case 36: printf("= Assignment operator\n"); break;
}
return 0;
```

}

INPUT File:

OUTPUT:

Btech cse (sem-6)
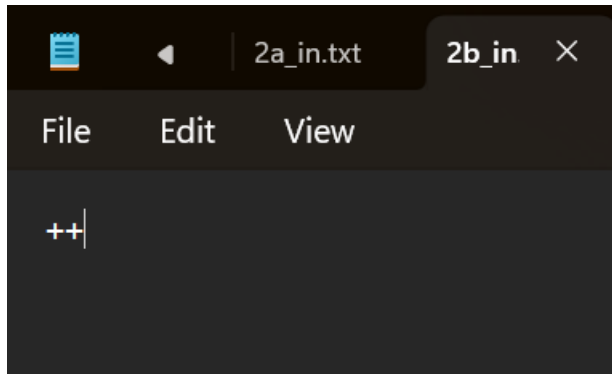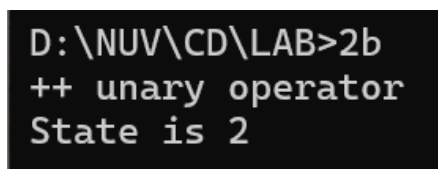
AIM c: Write a program to recognize the valid number.

PROGRAM CODE:

```
#include  <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
int main() {
char c, buffer[1000], lexeme[1000];
int  i = 0, state = 0, f = 0, j = 0;
FILE *fp = fopen("2c_in.txt", "r");
if (fp == NULL) {
printf("Error opening file.\n");
return 1;
}
// Read file content into buffer
while ((c = fgetc(fp)) != EOF && j < 1000) {
buffer[j++] = c;
}
buffer[j] = '\0';
fclose(fp);
// DFA processing
while (buffer[i] != '\0') {
c = buffer[i];
switch (state) {
case 0:
if (isdigit(c)) {
state = 1;
lexeme[f++] = c;
} else if (c == '+' || c == '-') {
state = 0; // Allow leading + or -
lexeme[f++] = c;
} else if (isspace(c)) {
// Skip whitespace
```

```c
} else {
state = 99; // Invalid character
}
break;
case 1:

if (isdigit(c)) {
state = 1;
lexeme[f++] = c;
} else if (c == '.') {
state = 2;
lexeme[f++] = c;
} else if (c == 'e' || c == 'E') {
state = 4;
lexeme[f++] = c;
} else {
lexeme[f] = '\0';
printf("The input %s is a valid integer.\n", lexeme);
f = 0;
state = 0;
i--; // Re-check current character
}
break;
case 2:
if (isdigit(c)) {
state = 3;
lexeme[f++] = c;
} else {
lexeme[f] = '\0';
printf("%s is an invalid floating-point input.\n", lexeme);
f = 0;
state = 0;
i--;
}
```

```
break;
case 3:
if (isdigit(c)) {
state = 3;
lexeme[f++] = c;
} else if (c == 'e' || c == 'E') {
state = 4;
lexeme[f++] = c;
} else {
lexeme[f] = '\0';
printf("The input %s is a valid floating-point number.\n", lexeme);
f = 0;
state = 0;
i--;
}

break;
case 4:
if (isdigit(c)) {
state = 6;
lexeme[f++] = c;
} else if (c == '+' || c == '-') {
state = 5;
lexeme[f++] = c;
} else {
lexeme[f] = '\0';
printf("%s is an invalid scientific notation.\n", lexeme);
f = 0;
state = 0;
i--;
}
break;
case 5:
if (isdigit(c)) {
```
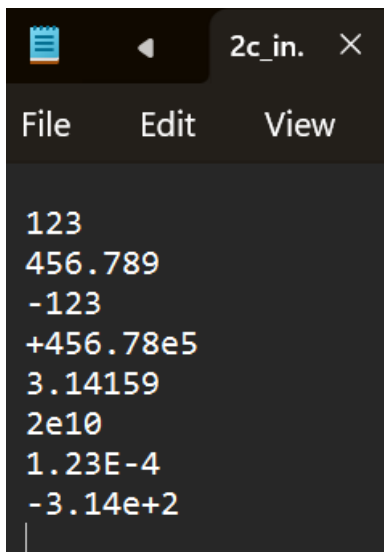
Btech cse (sem-6)

```
state = 6;
lexeme[f++] = c;
} else {
lexeme[f] = '\0';
printf("%s is an invalid scientific notation.\n", lexeme);
f = 0;
state = 0;
i--;
}
break;
case 6:
if (isdigit(c)) {
state = 6;
lexeme[f++] = c;
} else {
lexeme[f] = '\0';
printf("The input %s is a valid scientific notation number.\n", lexeme);
f = 0;
state = 0;
i--;
}
break;
case 99:

printf("Invalid character encountered: %c\n", c);
f = 0;
state = 0;
break;
}
i++;
}
// Final token check
if (f != 0) {
lexeme[f] = '\0';
```

if (state == 1)

printf("The input %s is a valid integer.\n", lexeme);

else if (state == 3)

printf("The input %s is a valid floating-point number.\n", lexeme);

else if (state == 6)

printf("The input %s is a valid scientific notation number.\n", lexeme);

}

return 0;

}

INPUT File:

OUTPUT:

AIM d: Write a program to recognize the valid comments.

PROGRAM CODE:
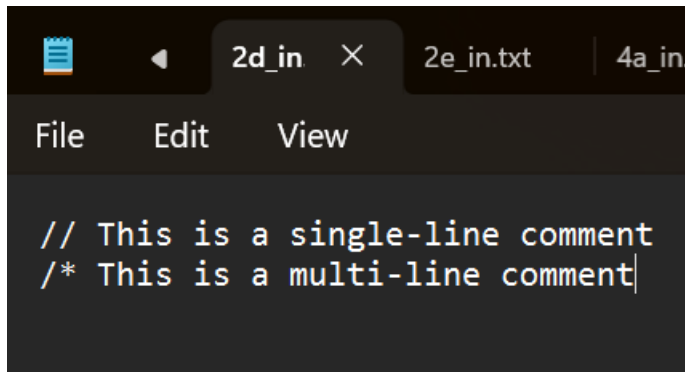
```c
#include <stdio.h>
void main() {
int state = 0, i = 0;
FILE *fptr;
fptr = fopen("2d_in.txt", "r");
// Read input from the file
char input[100];
fgets(input, 100, fptr);
printf("Input string: %s", input);
fclose(fptr);
// State machine to detect comments
while (input[i] != '\0') {
switch (state) {
case 0:
if (input[i] == '/') {
state = 1;
} else {
state = 3;
}
break;
case 1:
if (input[i] == '*') {
state = 4;
} else if (input[i] == '/') {
state = 2;
} else {
state = 3;
}
break;
case 4:
if (input[i] == '*') {
```

```
state = 5;
} else {
state = 4;
}
break;

case 5:
if (input[i] == '/') {
state = 6;
} else {
state = 4;
}
break;
case 6:
if (input[i] != '\0') {
state = 3;
}
break;
}
i++;
}
// Determine if the comment is valid or invalid
if (state == 2) {
printf("\nString is a valid single-line comment");
} else if (state == 6) {
printf("\nString is a valid multi-line comment");
} else {
printf("\nString is an invalid comment");
}
}
```
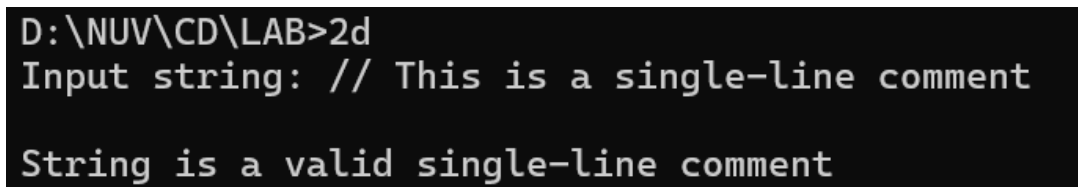
INPUT File:

OUTPUT:

AIM e: Program to implement Lexical Analyzer.

PROGRAM CODE:

```c
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#define BUFFER_SIZE 1000
// Function declarations
void check_keyword_or_identifier(char *lexeme);
void recognize_number(char *lexeme);
void recognize_operator(char *buffer, int *index);
void recognize_comment(char *buffer, int *index);
void main() {
FILE *f1;
char *buffer;
char lexeme[50];
char c;
int i = 0, f = 0, state = 0;
// Open file for reading
f1 = fopen("2e_in.txt", "r");
if (f1 == NULL) {
printf("Error: Could not open input.txt\n");
return;
}
// Read the file into memory
fseek(f1, 0, SEEK_END);
long file_size = ftell(f1);
rewind(f1);
buffer = (char *)malloc(file_size + 1);
fread(buffer, 1, file_size, f1);
buffer[file_size] = '\0';
fclose(f1);
// Start lexical analysis
```

```
while (buffer[f] != '\0') {
c = buffer[f];
switch (state) {
case 0:
if (isalpha(c) || c == '_') {

state = 1;
lexeme[i++] = c;
} else if (isdigit(c)) {
state = 2;
lexeme[i++] = c;
} else if (c == '/' && (buffer[f + 1] == '/' || buffer[f + 1] == '*')) {
recognize_comment(buffer, &f);
state = 0;
} else if (strchr("+-*/%=<>!", c)) {
recognize_operator(buffer, &f);
state = 0;
} else if (strchr(";,{}()", c)) {
printf("%c is a symbol\n", c);
state = 0;
} else if (isspace(c)) {
state = 0;
}
break;
case 1:
if (isalnum(c) || c == '_') {
lexeme[i++] = c;
} else {
lexeme[i] = '\0';
check_keyword_or_identifier(lexeme);
lexeme[i] = '\0';
i = 0;
state = 0;
f--; // Go back to recheck the current character
```

```
}
break;
case 2:
if (isdigit(c)) {
lexeme[i++] = c;
} else if (c == '.') {
state = 3;
lexeme[i++] = c;
} else if (c == 'E' || c == 'e') {
state = 4;
lexeme[i++] = c;
} else {
lexeme[i] = '\0';
recognize_number(lexeme);
i = 0;

state = 0;
f--; // Go back to recheck the current character
}
break;
case 3:
if (isdigit(c)) {
lexeme[i++] = c;
} else {
lexeme[i] = '\0';
recognize_number(lexeme);
i = 0;
state = 0;
f--; // Go back to recheck the current character
}
break;
case 4:
if (isdigit(c) || c == '+' || c == '-') {
state = 5;
```

```c
lexeme[i++] = c;
} else {
lexeme[i] = '\0';
recognize_number(lexeme);
i = 0;
state = 0;
f--; // Go back to recheck the current character
}
break;
case 5:
if (isdigit(c)) {
lexeme[i++] = c;
} else {
lexeme[i] = '\0';
recognize_number(lexeme);
i = 0;
state = 0;
f--; // Go back to recheck the current character
}
break;
}
f++; // Move forward in the buffer
}

// Free dynamically allocated memory
free(buffer);
}
// Function to check if the lexeme is a keyword or identifier
void check_keyword_or_identifier(char *lexeme) {
int i = 0;
char *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",
```

33

```c
"sizeof", "static", "struct", "switch", "typedef", "union", "unsigned",
"void", "volatile", "while"
};
for (i = 0; i < 32; i++) {
if (strcmp(lexeme, keywords[i]) == 0) {
printf("%s is a keyword\n", lexeme);
return;
}
}
printf("%s is an identifier\n", lexeme);
}
// Function to recognize a number (integer, floating-point, scientific notation)
void recognize_number(char *lexeme) {
printf("%s is a valid number\n", lexeme);
}
// Function to recognize an operator
void recognize_operator(char *buffer, int *index) {
char operators[][3] = {"+", "-", "*", "/", "%", "=", "==", "!=", "<", ">", "<=", ">="};
char op[3] = {buffer[*index], buffer[*index + 1], '\0'};
int i = 0;
// Handle two-character operators (e.g., "==" or ">=")
for (i = 0; i < 12; i++) {
if (strcmp(op, operators[i]) == 0) {
printf("%s is an operator\n", op);
(*index)++; // Skip the next character since we used two chars
return;
}
}
// Handle single-character operators

printf("%c is an operator\n", buffer[*index]);
}
// Function to recognize comments
void recognize_comment(char *buffer, int *index) {
```

```c
if (buffer[*index] == '/' && buffer[*index + 1] == '/') {
printf("// is a single-line comment\n");
while (buffer[*index] != '\n' && buffer[*index] != '\0')
(*index)++;
} else if (buffer[*index] == '/' && buffer[*index + 1] == '*') {
printf("/* is the start of a multi-line comment\n");
(*index) += 2;
while (!(buffer[*index] == '*' && buffer[*index + 1] == '/') && buffer[*index] != '\0')
(*index)++;
if (buffer[*index] == '*' && buffer[*index + 1] == '/') {
printf("*/ is the end of a multi-line comment\n");
(*index) += 2;
}
}
}
```

INPUT File:

OUTPUT:

```
D:\NUV\CD\LAB>2e
int is a keyword
main is an identifier
( is a symbol
) is a symbol
{ is a symbol
int is a keyword
x is an identifier
= is an operator
5 is a valid number
; is a symbol
// is a single-line comment
/* is the start of a multi-line comment
*/ is the end of a multi-line comment
x is an identifier
= is an operator
x is an identifier
+ is an operator
1 is a valid number
; is a symbol
return is a keyword
0 is a valid number
; is a symbol
} is a symbol
A is an identifier
= is an operator
C is an identifier
```

Btech cse (sem-6)

Practical – 3

AIM: To Study about Lexical Analyser Generator (LEX) and Flex(Fast Lexical Analyser)

What is a Lexical Analyzer Generator?

A Lexical Analyzer Generator is a tool that automates the creation of a Lexical Analyzer (lexer), which is the first phase of a compiler. Its job is to scan the source code and convert it into a stream of tokens—the basic building blocks like keywords, operators, identifiers, etc.

What is LEX?

LEX (short for Lexical Analyzer Generator) was one of the earliest tools developed to help create lexical analyzers. It uses regular expressions to specify token patterns and generates C code that recognizes these patterns.

- Developed in the 1970s as part of the Unix toolchain.
- Usually used with Yacc (Yet Another Compiler Compiler) for syntax analysis.

Structure of a LEX program:

```
%{
// C declarations
%}
%%
[0-9]+ { printf("NUMBER\n"); }
[a-zA-Z]+ { printf("WORD\n"); }
. { printf("UNKNOWN\n"); }
%%
int main() {
yylex();
return 0;
}
```

What is Flex?

Flex (Fast Lexical Analyzer Generator) is a free and faster alternative to LEX, and it is more commonly used today.

•   Flex is open-source and generates more efficient code.

•   It is backward-compatible with LEX but offers extra features and optimizations.

•   It produces a C source file (e.g., lex.yy.c) which can be compiled with GCC.

Advantages of Flex over LEX:

Feature	LEX	Flex
Speed	Slower	Faster
Portability	Unix-only	Cross-platform (POSIX)
Extensions	Limited	More modern extensions
Output Code	Less optimized	Highly optimized

How It Works

1. Input: Regular expressions and actions (C code) for each pattern.

2. Processing: Generates a C program (lex.yy.c) that uses a finite state machine (DFA).

3. Output: The compiled lexer reads input, matches patterns, and executes associated actions.

Use Cases

•   Programming language compilers (e.g., C, Python).

•   Interpreters.

•   Code analyzers or format checkers.

•   Custom parsers in domain-specific languages.

Practical – 4

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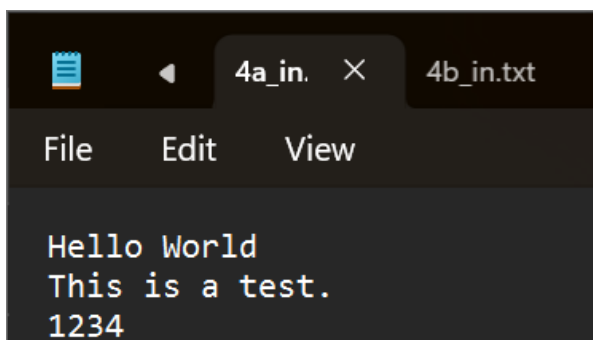
PROGRAM CODE:

```
%{
#include<stdio.h>
int l=0, w=0, c=0;
int in_word=0;
%}
%%
[a-zA-Z] { c++; in_word = 1; }
\n { l++; if (in_word) { w++; in_word=0; } }
[ \t]+ { if (in_word) { w++; in_word=0; } }
. { if (in_word) { w++; in_word = 0; } }
%%
void main() {
yyin = fopen("4a_in.txt", "r");
yylex();
printf("Number of characters: %d \nNumber of lines: %d \nNumber of words: %d", c, l, w);
}
int yywrap() {
return (1);
}
```
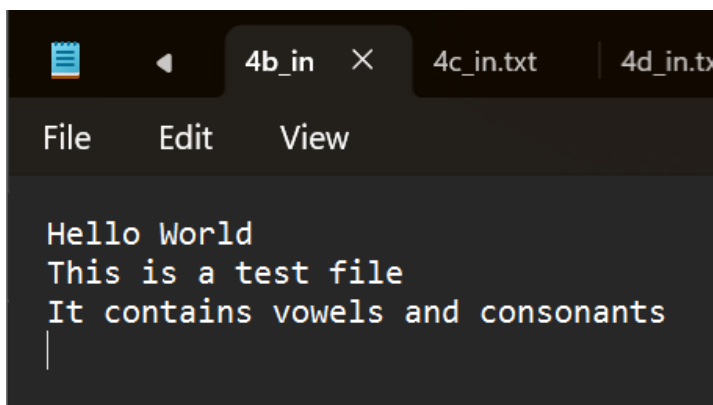
INPUT File:

OUTPUT:

```
D:\NUV\CD\LAB>4a
Number of characters: 21
Number of lines: 3
Number of words: 6
```

Btech cse (sem-6)

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

PROGRAM CODE:

```
%{
#include<stdio.h>
int consonants = 0, vowels = 0;
%}
%%
[aeiouAEIOU] { vowels++; }
[a-zA-Z] { consonants++; }
\n
.
%%
int main() {
yyin = fopen("4b_in.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);
}
```

INPUT File:
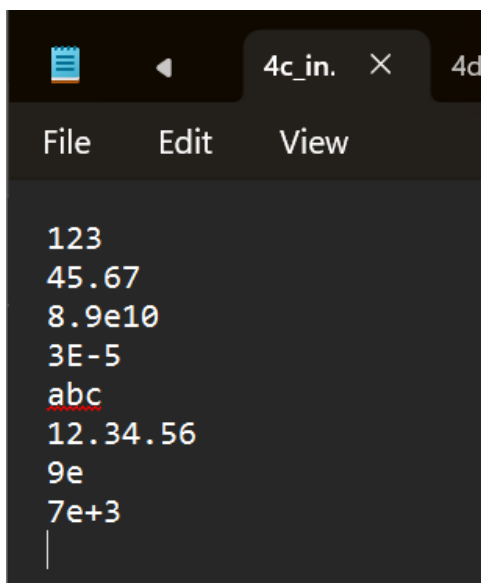
OUTPUT:

```
D:\NUV\CD\LAB>4b
 This File contains...
        19 vowels
        35 consonants
D:\NUV\CD\LAB>
```

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

PROGRAM CODE:

```
%{
#include<stdio.h>
%}
%%
[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?  {
printf("%s is a valid number \n", yytext);
}
\n ;
. ;
%%
int main() {
yyin = fopen("4c_in.txt", "r");
yylex();
return 0;
}
int yywrap() {
return (1);
}
```

INPUT File:

OUTPUT:

```
D:\NUV\CD\LAB>4c
123 is a valid number
45.67 is a valid number
8.9e10 is a valid number
3E-5 is a valid number
12.34 is a valid number
56 is a valid number
9 is a valid number
7e+3 is a valid number
```

Btech cse (sem-6)

AIM d: Write a Lex program which adds line numbers to the given file and display the same into different file.
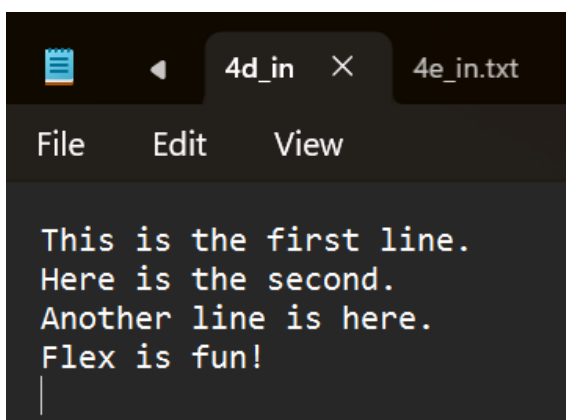
PROGRAM CODE:

```
%{
#include<stdio.h>
int line_number = 1;
%}
%%
.+ {
fprintf(yyout, "%d: %s", line_number, yytext);
line_number++;
}
%%
int main() {
yyin = fopen("4d_in.txt", "r");
yyout = fopen("fourth_output.txt", "w");
yylex();
printf("done");
return 0;
}
int yywrap() {
return (1);
}
```
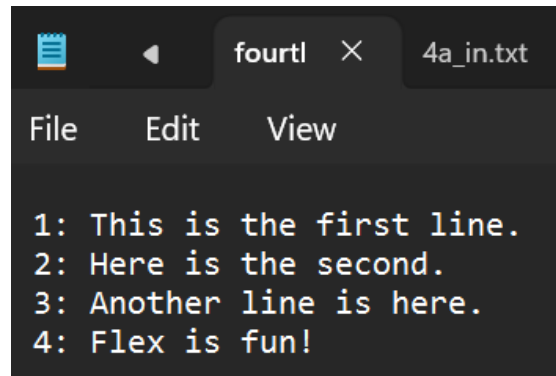
INPUT File:

OUTPUT:

```
D:\NUV\CD\LAB>4d
done
```



```
1: This is the first line.
2: Here is the second.
3: Another line is here.
4: Flex is fun!
```
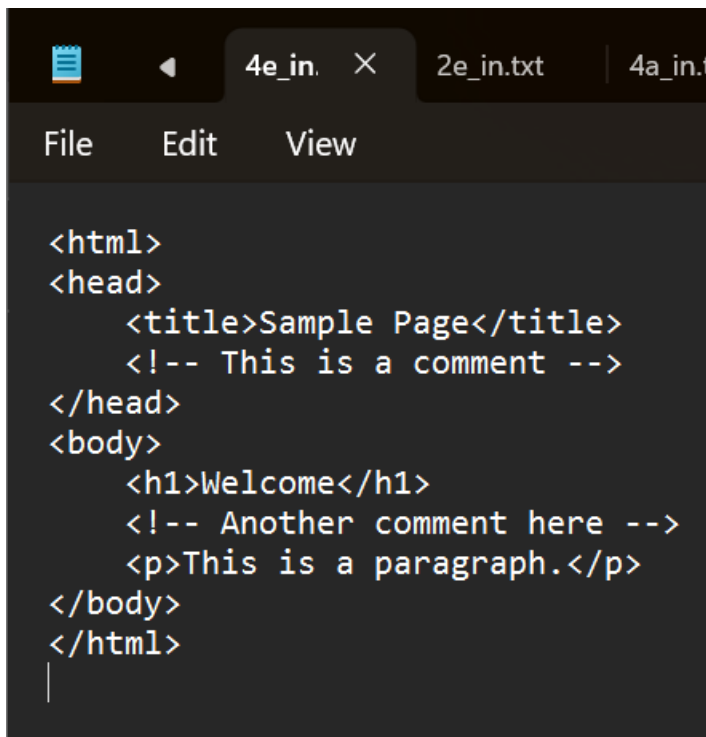
Btech cse (sem-6)

AIM e: Write a Lex program to printout all markup tags and HTML comments in file.
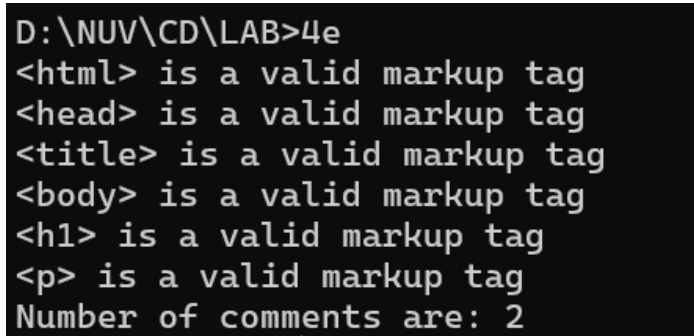
PROGRAM CODE:

```
%{
#include<stdio.h>
int num = 0;
%}
%%
"<"[A-Za-z0-9]+">" {
printf("%s is a valid markup tag\n", yytext);
}
"<!--"[^-->]*"-->" {
num++;
}
\n ;
. ;
%%
int main() {
yyin = fopen("4e_in.txt", "r");
yylex();
printf("Number of comments are: %d", num);
return 0;
}
int yywrap() {
return (1);
}
```

INPUT File:

```
<html>
<head>
    <title>Sample Page</title>
    <!-- This is a comment -->
</head>
<body>
    <h1>Welcome</h1>
    <!-- Another comment here -->
    <p>This is a paragraph.</p>
</body>
</html>
```

OUTPUT:

```
D:\NUV\CD\LAB>4e
<html> is a valid markup tag
<head> is a valid markup tag
<title> is a valid markup tag
<body> is a valid markup tag
<h1> is a valid markup tag
<p> is a valid markup tag
Number of comments are: 2
```
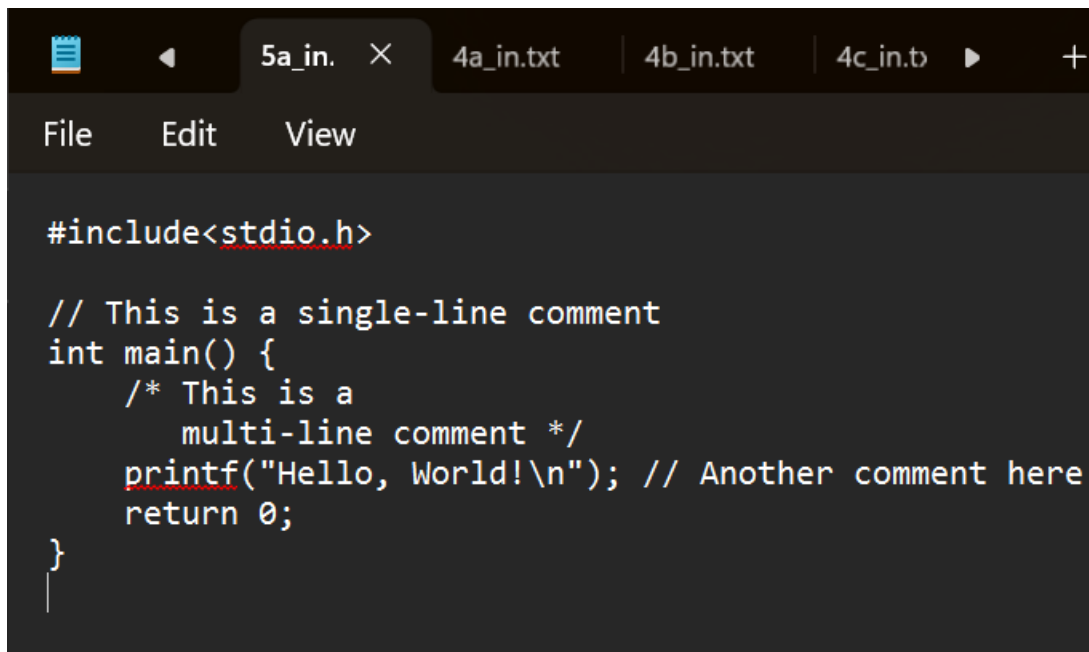
Practical – 5

AIM 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.

PROGRAM CODE:

```
%{
#include<stdio.h>
int cmt = 0;
%}
%%
"//".* {
fprintf(yyout, "\n");
cmt++;
}
"/*"([^*]|\*+[^*/])"*"+"/" {
fprintf(yyout, "\n");
cmt++;
}
.|\n {
fprintf(yyout, "%s", yytext);
}
%%
void main(){
yyin = fopen("5a_in.txt", "r");
yyout = fopen("5a_out.txt", "w");
yylex();
printf("%d Comment(s)\n", cmt);
}
int yywrap(){
return(1);
}
```

INPUT File:

OUTPUT:

```
D:\NUV\CD\LAB>5a
2 Comment(s)
```
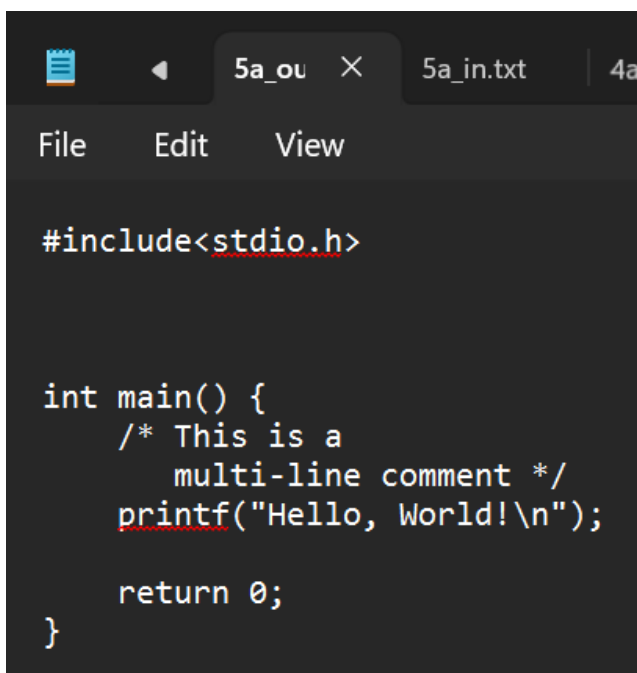
AIM b: Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program.

PROGRAM CODE:

```
%{
#include <stdio.h>
#include <string.h>
int is_keyword(const char *str);
%}
%option noyywrap
%%
"auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"|
"else"|"enum"|"extern"|"float"|"for"|"goto"|"if"|"inline"|"int"|"long"|
"register"|"return"|"short"|"signed"|"sizeof"|"static"|"struct"|"switch"|
"typedef"|"union"|"unsigned"|"void"|"volatile"|"while" {
printf("Keyword: %s\n", yytext);
}
[ \t\n]+ ; // Skip whitespace characters
"=="|"!="|"<="|">="|"="|"+"|"-"|"*"|"/"|"%"|"&&"|"||"|"!"|"<"|">" {
printf("Operator: %s\n", yytext);
}
[0-9]+(\.[0-9]+)? {
printf("Number: %s\n", yytext);
}
\"([^\\\"]|\\.)*\" {
printf("String Literal: %s\n", yytext);
}
\'.\' {
printf("Character Literal: %s\n", yytext);
}
[{}()\[\],;.] {
printf("Special Symbol: %s\n", yytext);
}
[a-zA-Z_][a-zA-Z0-9_]* {
```
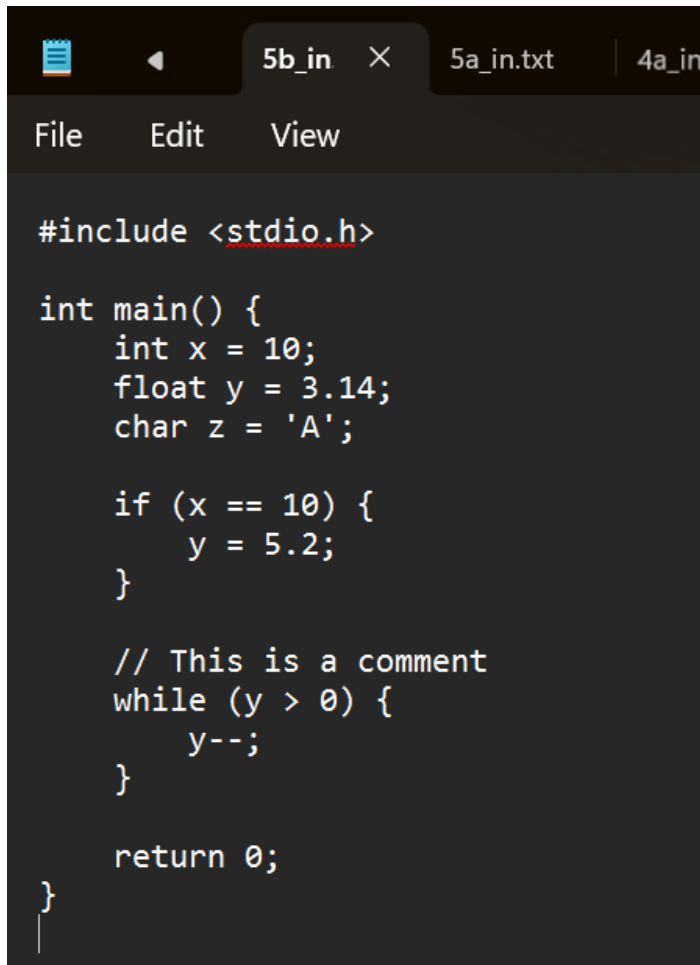
```
if (is_keyword(yytext))
printf("Keyword: %s\n", yytext);
else
printf("Identifier: %s\n", yytext);
}
. {
printf("Unrecognized Character: %s\n", yytext);
}
%%
int is_keyword(const char *str) {
const char *keywords[] = {
"auto", "break", "case", "char", "const", "continue", "default", "do", "double",
"else", "enum", "extern", "float", "for", "goto", "if", "inline", "int", "long",
"register", "return", "short", "signed", "sizeof", "static", "struct", "switch",
"typedef", "union", "unsigned", "void", "volatile", "while", NULL
};
for (int i = 0; keywords[i] != NULL; i++) {
if (strcmp(keywords[i], str) == 0)
return 1;
}
return 0;
}
int main() {
yylex(); // Start lexical analysis
return 0;
}
```

INPUT File:

```c
#include <stdio.h>

int main() {
    int x = 10;
    float y = 3.14;
    char z = 'A';

    if (x == 10) {
        y = 5.2;
    }

    // This is a comment
    while (y > 0) {
        y--;
    }

    return 0;
}
```

Practical – 6

AIM: Program to implement Recursive Descent Parsing in C.

PROGRAM CODE:

```c
#include  <stdio.h>
#include <stdlib.h>
char s[20];
int i = 1;
char l;
int match(char l);
int E1();
int E()
{
if (l == 'i')
{
match('i');
E1();
}
else
{
printf("Error parsing string");
exit(1);
}
return 0;
}
int E1()
{
if (l == '+')
{
match('+');
match('i');
E1();
}
else
```

```c
{
return 0;
}
}
int match(char t)
{
if (l == t)
{
l = s[i];
i++;
}
else
{
printf("Syntax Error");
exit(1);
}
return 0;
}
void main()
{
printf("Enter the string: ");
scanf("%s", &s);
l = s[0];
E();
if (l == '$')
{
printf("parsing successful");
}
else
{
printf("Error while parsing the string\n");
}
}
```

OUTPUT:

```
D:\NUV\CD\LAB>6a
Enter the string: i+i+i$
parsing successful
D:\NUV\CD\LAB>
```

Practical – 7

AIM a: To Study about Yet Another Compiler-Compiler(YACC).

What is YACC?

YACC (Yet Another Compiler-Compiler) is a tool used to generate parsers, which are part of the syntax analysis phase of a compiler. It takes a formal grammar (usually written in BNF-like syntax) and produces C code that can parse input sequences according to that grammar.

•   Developed by Stephen C. Johnson in the 1970s at Bell Labs.

•   Works closely with LEX/Flex, which handles lexical analysis (tokenization).

•   YACC focuses on syntax parsing (checking structure of token sequences).

Components of YACC

A YACC program consists of three sections, just like LEX:

%{

// C declarations

%}

%%

// Grammar rules with actions

%%

// Supporting C code (like main)

How YACC Works

1. Input: A context-free grammar (CFG) with actions (usually in C).

2. Output: A parser in C that uses LALR(1) parsing (Look-Ahead LR).

3. Integration: Uses token definitions from LEX (via yylex()).

Executes specific C actions when grammar rules match.

Key Features

Feature	Description
Grammar Type	Context-Free Grammar
Parsing Method	LALR(1) Parser (Bottom-Up)
Integration	Works with LEX/Flex
Output	C code (y.tab.c)
Action Language	C (executed when rules match)

Use Cases

•   Building compilers/interpreters

•   Scripting languages

•   Code validators or analyzers

•   Structured data parsers (e.g., config files, DSLs)

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

PROGRAM CODE:

.l file

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "7b.tab.h"
%}
%%
[0-9]+ {yylval = atoi(yytext); return NUM;}
[a-zA-Z_][a-zA-Z_0-9]* {return id;}
[-+*\n] {return *yytext;}
[ \t] { }
. yyerror("invalid character");
%%
int yywrap() {
return 0;
}
```

.y file

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(char *);
%}
%token NUM
%token id
%%
S: E '\n' {
printf("valid syntax");
return(0);
}
E: E '+' T { }
```
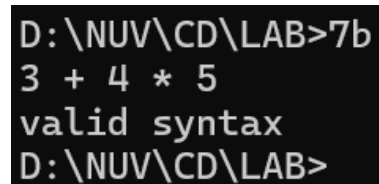
| E '-' T { }

| T { }

T: T '*' F { }

| F { }

F: NUM { }

| id { }

%%

void yyerror(char *s) {

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

}

int main() {

yyparse();

return 0;

}

OUTPUT:

```
D:\NUV\CD\LAB>7b
3 + 4 * 5
valid syntax
D:\NUV\CD\LAB>
```

AIM c: Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments.
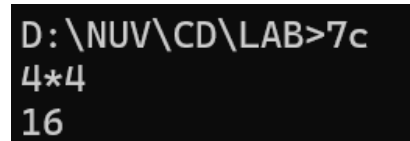
PROGRAM CODE:

.l file

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "7c.tab.h"
%}
%%
[0-9]+ { yylval = atoi(yytext); return NUM; }
[-+*\n] { return *yytext; }
[ \t] { }
. { yyerror("invalid character"); }
%%
int yywrap() {
return 0;
}
```

.y file

```
%{
#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) {

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

}

int main() {

yyparse();

return 0;

}

OUTPUT:

```
D:\NUV\CD\LAB>7c
4*4
16
```

AIM d: Create Yacc and Lex specification files are used to convert infix expression.

PROGRAM CODE:

.l file

```
%{
#include<stdio.h>
#include "7d.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;
}
```
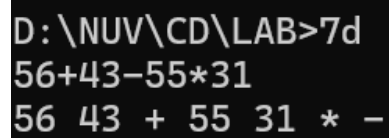
.y file

```
%{
#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:

```
D:\NUV\CD\LAB>7d
56+43-55*31
56 43 + 55 31 * -
```

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

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