

LAB Manual
Of
Compiler Design Laboratory
(CSE606)

Bachelor of Technology (CSE)

By

Nirav Lad (22000743)

Third Year, Semester 6

Course In-charge: Prof. Vaibhavi Patel



**NAVRACHANA
UNIVERSITY**

Accredited with
Grade 'A' by NAAC

Department of Computer Science and Engineering

School Engineering and Technology

Navrachana University, Vadodara

Spring Semester

(2024-2025)

SCHOOL OF ENGINEERING & TECHNOLOGY

BACHELOR OF TECHNOLOGY

COMPILER DESIGN

6TH SEMESTER

DEPARTMENT OF COMPUTER SCIENCE &
ENGINEERING

Laboratory Manual

By:

Nirav Lad

22000743

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.

Program – 1

a) Write a program to recognize strings starts with an 'a' over {a, b}.

Code-

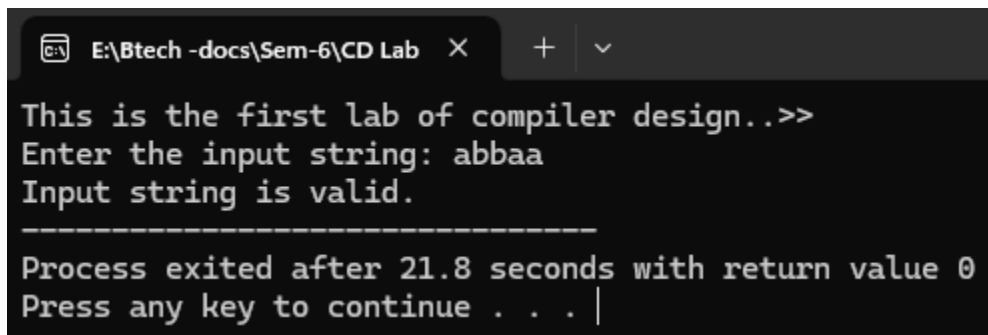
```
#include <stdio.h>

int main()
{
    printf("This is the first lab of compiler design..>>");
    nti =0,state=0;
    char input[10];
    printf("\nEnter the input 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;
```

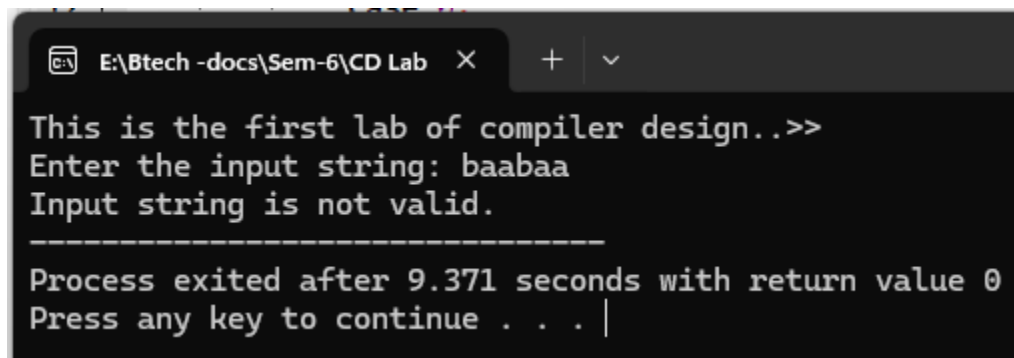
```
        else state=3;
        break;
    case 3:
        state=3;
    }
    i++;
}
if(state==1) printf("Input string is valid.");
else if(state==2 || state==0) printf("Input string is not
valid.");
else if(state==3) printf("Input string is not recognized");

return 0;
}
```

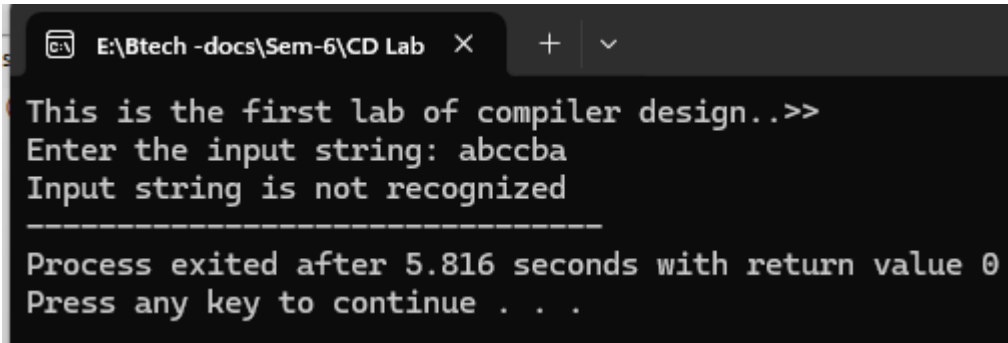
Output-



```
E:\Btech -docs\Sem-6\CD Lab  X  +  v
This is the first lab of compiler design..>>
Enter the input string: abbaa
Input string is valid.
-----
Process exited after 21.8 seconds with return value 0
Press any key to continue . . . |
```



```
E:\Btech -docs\Sem-6\CD Lab  X  +  v
This is the first lab of compiler design..>>
Enter the input string: baabaa
Input string is not valid.
-----
Process exited after 9.371 seconds with return value 0
Press any key to continue . . . |
```



```
E:\Btech -docs\Sem-6\CD Lab
This is the first lab of compiler design..>>
Enter the input string: abccba
Input string is not recognized
-----
Process exited after 5.816 seconds with return value 0
Press any key to continue . . .
```

b) Write a program to recognize strings end with 'a'.

Code-

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

```
int main()
```

```
{
```

```
    int i=0,state=0;
```

```
    char input[10];
```

```
    printf("\nEnter the input string: ");
```

```
    scanf("%s",input);
```

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

```
        switch(state){
```

```
            case 0:
```

```
                if(input[i]=='a'){
```

```
                    state=1;
```

```
                }
```

```
                else state=0;
```

```
                break;
```

```
            case 1:
```

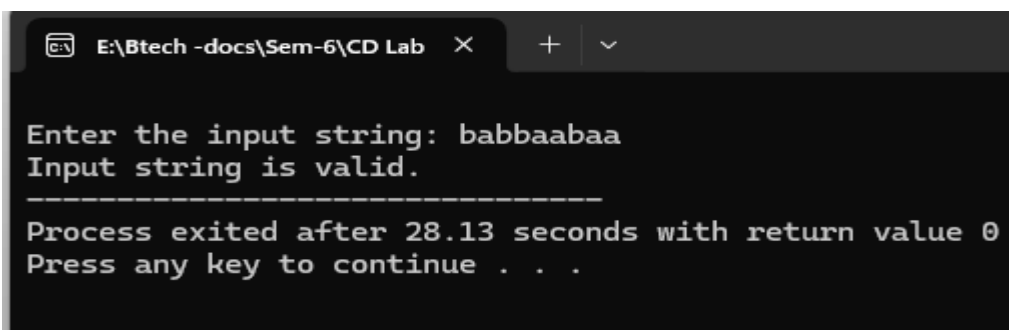
```
                if(input[i]=='a') state=1;
```

```
                else state=0;
```

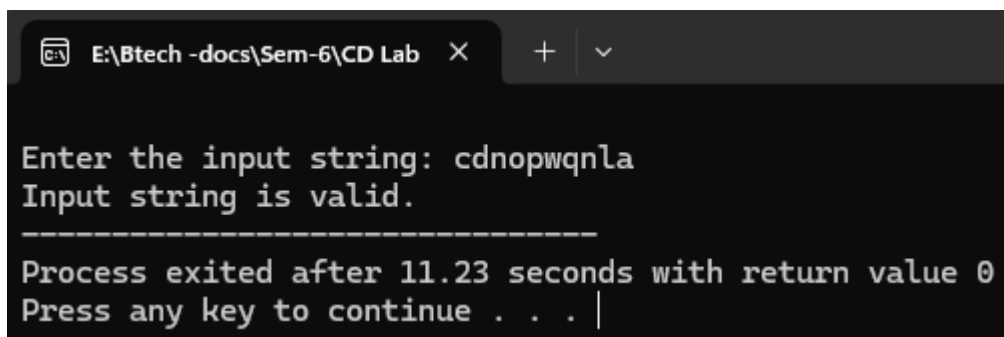
```
                break;
```

```
    }  
    i++;  
    }  
    if(state==1) printf("Input string is valid.");  
    else if(state==0) printf("Input string is not valid.");  
  
    return 0;  
}
```

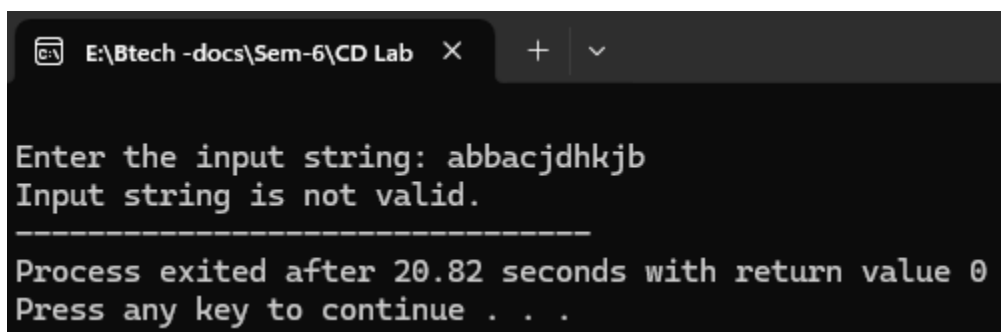
Output-



```
E:\Btech -docs\Sem-6\CD Lab X + v  
Enter the input string: babbaabaa  
Input string is valid.  
-----  
Process exited after 28.13 seconds with return value 0  
Press any key to continue . . .
```



```
E:\Btech -docs\Sem-6\CD Lab X + v  
Enter the input string: cdnopwqnla  
Input string is valid.  
-----  
Process exited after 11.23 seconds with return value 0  
Press any key to continue . . . |
```



```
E:\Btech -docs\Sem-6\CD Lab X + v  
Enter the input string: abbacjdjhkb  
Input string is not valid.  
-----  
Process exited after 20.82 seconds with return value 0  
Press any key to continue . . .
```

c) Write a program to recognize strings end with 'ab'. Take the input from text file.

Code-

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

int main() {
    FILE *file;
    char input[100];

    file = fopen("lab-1-test.txt", "r");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    while (fscanf(file, "%s", input) != EOF) {
        nti = 0, state = 0;

        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;
```

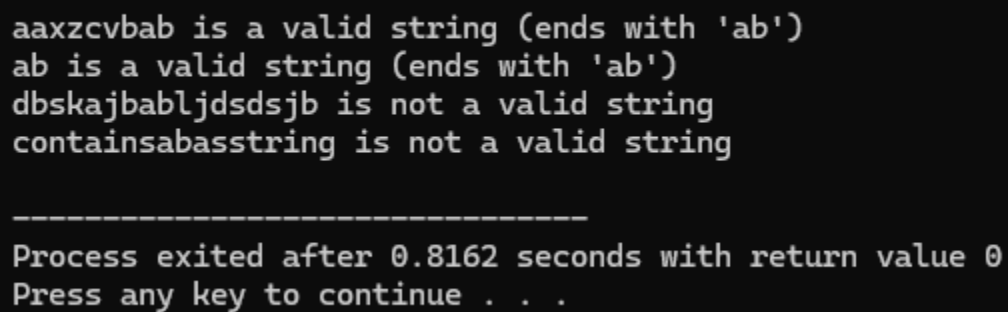


```
        else if (input[i] == 'a') state = 1;
        else state = 0;
        break;
    case 2:
        if (input[i] == 'a') state = 1;
        else state = 0;
        break;
    }
    i++;
}

if (state == 2) printf("%s is a valid string (ends with 'ab')\n", input);
else printf("%s is not a valid string\n", input);
}

fclose(file);
return 0;
}
```

Output-



```
aaxzcvbab is a valid string (ends with 'ab')
ab is a valid string (ends with 'ab')
dbskajbabljdsdsjb is not a valid string
containsabasstring is not a valid string

-----
Process exited after 0.8162 seconds with return value 0
Press any key to continue . . .
```

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

Code-

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

int main() {
    FILE *file;
    char input[100];

    file = fopen("lab-1-test.txt", "r");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    while (fscanf(file, "%s", input) != EOF) {
        nti = 0, state = 0;

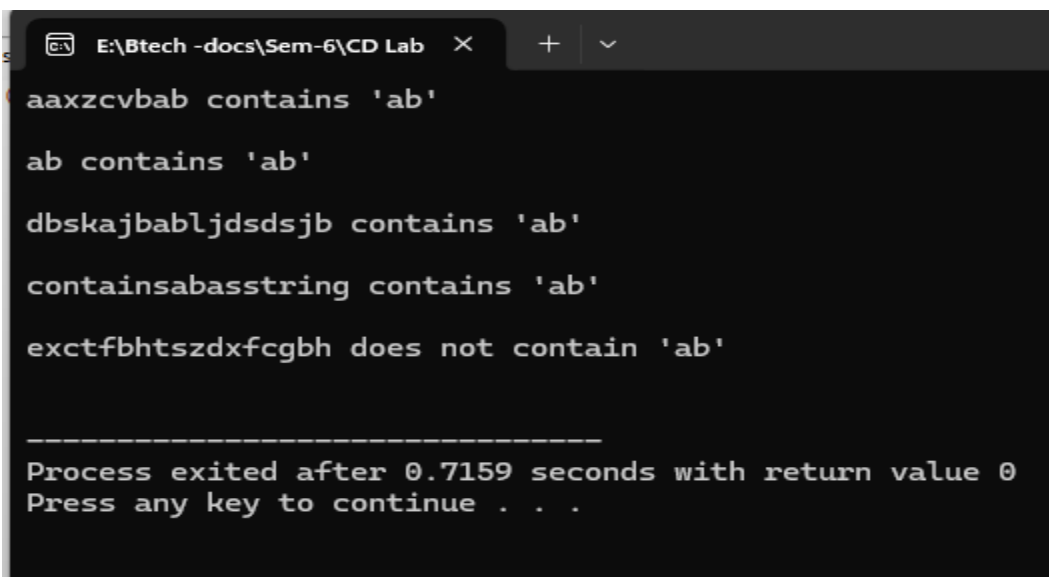
        while (input[i] != '\0') {
            switch (state) {
                case 0:
                    if (input[i] == 'a') state = 1;
                    break;
                case 1:
                    if (input[i] == 'b') state = 2; // Found 'ab'
```

```
        else if (input[i] != 'a') state = 0; // Reset if not 'a'
        break;
    case 2:
        break; // Remain in final state once 'ab' is found
    }
    i++;
}

if (state == 2) printf("%s contains 'ab'\n\n", input);
else printf("%s does not contain 'ab'\n\n", input);
}

fclose(file);
return 0;
}
```

Output-



```
E:\Btech -docs\Sem-6\CD Lab
aaxzcvbab contains 'ab'
ab contains 'ab'
dbskajbabljdsdsjb contains 'ab'
containsabasstring contains 'ab'
exctfbhtszdxfdbh does not contain 'ab'

-----
Process exited after 0.7159 seconds with return value 0
Press any key to continue . . .
```

Program – 2

a) Write a program to recognize the valid identifiers.

Code –

```
#include <stdio.h>
#include <ctype.h>

int main() {
    int i = 0, state = 0;
    char input[10];

    printf("Enter the input string: ");
    scanf("%9s", input);

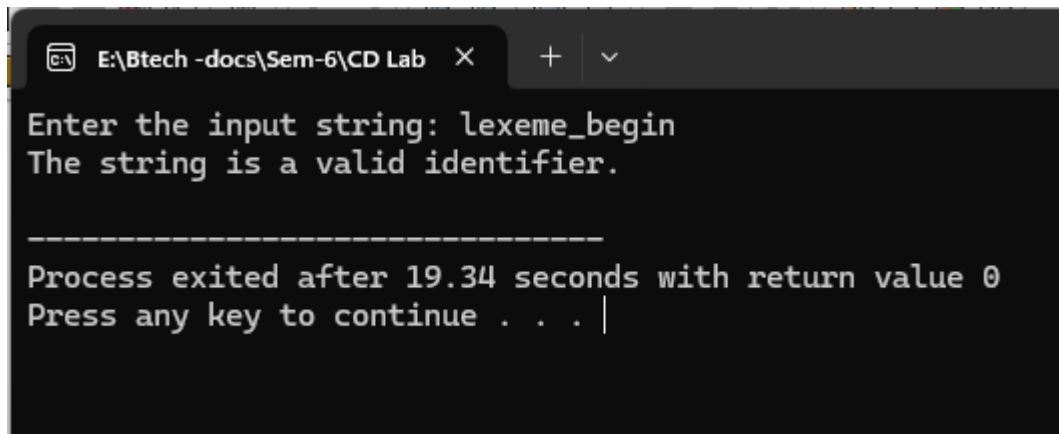
    while (input[i] != '\0') {
        switch (state) {
            case 0:
                if (input[i] == 'i') state = 1;
                else if (isalpha(input[i]) || input[i] == '_') state = 4;
                break;
            case 1:
                if (input[i] == 'n') state = 2;
                else state = 4;
                break;
            case 2:
                if (input[i] == 't') state = 3;
                else state = 4;
                break;
            case 3:
                if (input[i] == 'n') state = 2;
                else state = 4;
                break;
            case 4:
                break;
        }
        i++;
    }
}
```

```
        break;
    case 3:
        if (input[i] == '\0') state = 5;
        else state=4;
        break;
    case 4:
        if (isalpha(input[i]) || input[i] == '_' || isdigit(input[i])) state =
4;
        else if (input[i] == '\0') state = 6;
        else state = 6;
        break;
    case 5:
        state = 5;
        break;
    case 6:
        state = 6;
        break;
    }
    i++;
}

if (state == 5 || state==3) printf("The string is a valid keyword.\n");
else if (state==4) printf("The string is a valid identifier.\n");
else printf("The string is neither a keyword nor an identifier.\n");

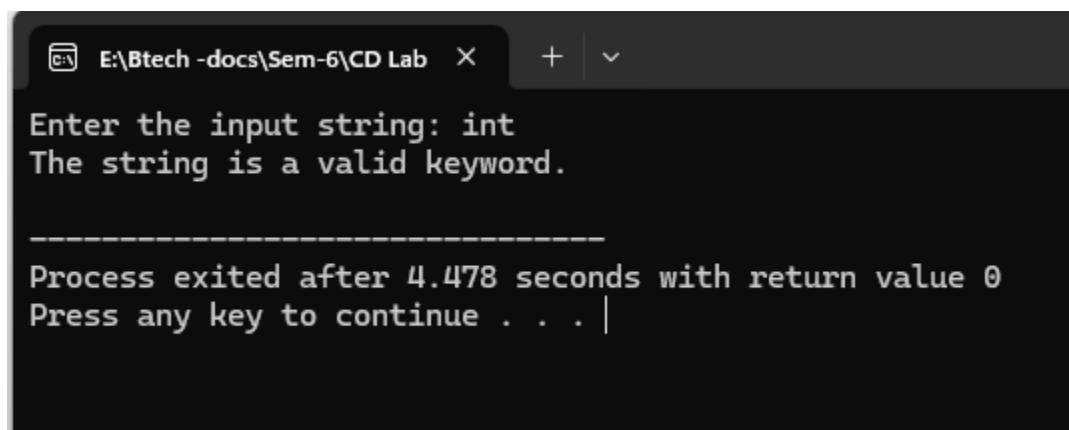
return 0;
}
```

Output –



```
E:\Btech -docs\Sem-6\CD Lab  X  +  v
Enter the input string: lexeme_begin
The string is a valid identifier.

-----
Process exited after 19.34 seconds with return value 0
Press any key to continue . . . |
```



```
E:\Btech -docs\Sem-6\CD Lab  X  +  v
Enter the input string: int
The string is a valid keyword.

-----
Process exited after 4.478 seconds with return value 0
Press any key to continue . . . |
```

b) Write a program to recognize the valid operators.

Code –

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

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

```
while (input[i] != '\0') {
    switch (state) {
        case 0:
            if (input[i] == '+') state = 51;
            else if (input[i] == '-') state = 54;
            else if (input[i] == '*') state = 57;
            else if (input[i] == '/') state = 59;
            else if (input[i] == '%') state = 61;
            else if (input[i] == '=') state = 63;
            else if (input[i] == '<') state = 65;
            else if (input[i] == '>') state = 68;
            else if (input[i] == '!') state = 71;
            else if (input[i] == '&') state = 73;
            else if (input[i] == '|') state = 75;
            else if (input[i] == '~') {
                printf("Bitwise operator [~].\n");
            }
            else if (input[i] == '^') {
                printf("Bitwise operator [^].\n");
            }
            else {
                printf("Invalid character: %c\n", input[i]);
            }
            break;
```

case 51:

```
    if (input[i] == '+') {  
        printf("Unary operator: ++\n");  
    }  
    else if (input[i] == '=') {  
        printf("Assignment operator: +=\n");  
    }  
    else {  
        printf("Arithmetic operator [+].\n");  
        i--; // Go back one step  
    }  
    state = 0;  
    break;
```

case 54:

```
    if (input[i] == '-') {  
        printf("Unary operator: --\n");  
    }  
    else if (input[i] == '=') {  
        printf("Assignment operator: -=\n");  
    }  
    else {  
        printf("Arithmetic operator [-].\n");  
        i--;  
    }  
    state = 0;
```



```
break;
```

```
case 57:
```

```
    if (input[i] == '=') {  
        printf("Assignment operator: *=\n");  
    }  
    else {  
        printf("Arithmetic operator [*].\n");  
        i--;  
    }  
    state = 0;  
    break;
```

```
case 59:
```

```
    if (input[i] == '=') {  
        printf("Assignment operator: /=\n");  
    }  
    else {  
        printf("Arithmetic operator [/].\n");  
        i--;  
    }  
    state = 0;  
    break;
```

```
case 61:
```

```
    if (input[i] == '=') {
```

```
        printf("Assignment operator: %%=\\n");
    }
    else {
        printf("Arithmetic operator [%%].\\n");
        i--;
    }
    state = 0;
    break;

case 63:
    if (input[i] == '=') {
        printf("Relational operator: ==\\n");
    }
    else {
        printf("Assignment operator [=].\\n");
        i--;
    }
    state = 0;
    break;

case 65:
    if (input[i] == '<') {
        printf("Relational operator: <=\\n");
    }
    else if (input[i] == '<') {
        printf("Bitwise operator: <<\\n");
```

```
}  
else {  
    printf("Relational operator [<].\n");  
    i--;  
}  
state = 0;  
break;
```

case 68:

```
if (input[i] == '=') {  
    printf("Relational operator: >=\n");  
}  
else if (input[i] == '>') {  
    printf("Bitwise operator: >>\n");  
}  
else {  
    printf("Relational operator [>].\n");  
    i--;  
}  
state = 0;  
break;
```

case 71:

```
if (input[i] == '=') {  
    printf("Relational operator: !=\n");  
}
```

```
else {  
    printf("Logical operator [!].\n");  
    i--;  
}  
state = 0;  
break;
```

case 73:

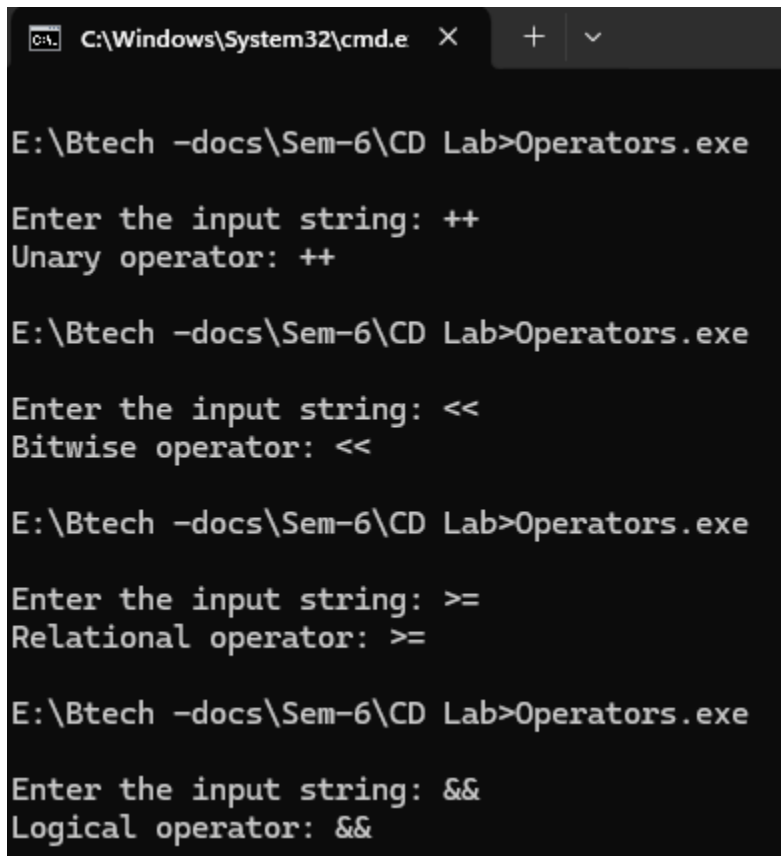
```
if (input[i] == '&') {  
    printf("Logical operator: &&\n");  
}  
else {  
    printf("Bitwise operator [&].\n");  
    i--;  
}  
state = 0;  
break;
```

case 75:

```
if (input[i] == '|') {  
    printf("Logical operator: ||\n");  
}  
else {  
    printf("Bitwise operator [||].\n");  
    i--;  
}
```

```
        state = 0;  
        break;  
    }  
    i++;  
}  
  
return 0;  
}
```

Output –



```
C:\Windows\System32\cmd.e X + v  
  
E:\Btech -docs\Sem-6\CD Lab>Operators.exe  
Enter the input string: ++  
Unary operator: ++  
  
E:\Btech -docs\Sem-6\CD Lab>Operators.exe  
Enter the input string: <<  
Bitwise operator: <<  
  
E:\Btech -docs\Sem-6\CD Lab>Operators.exe  
Enter the input string: >=  
Relational operator: >=  
  
E:\Btech -docs\Sem-6\CD Lab>Operators.exe  
Enter the input string: &&  
Logical operator: &&
```

c) Write a program to recognize the valid number.

Code –

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

```
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#define MAX_BUFFER_SIZE 1024

int main() {
    FILE *file;
    char buffer[MAX_BUFFER_SIZE];
    int i = 0, state = 0, lexIndex = 0;
    char ch, lexeme[100];

    file = fopen("input.txt", "r");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    while ((ch = fgetc(file)) != EOF && i < MAX_BUFFER_SIZE - 1) {
        buffer[i++] = ch;
    }
    buffer[i] = '\0';
    fclose(file);

    printf("File content:\n%s\n", buffer);
}
```

```
i = 0;
while (buffer[i] != '\0') {
    ch = buffer[i];
    switch (state) {
        case 0:
            if (isdigit(ch)) {
                state = 13;
                lexeme[lexIndex++] = ch;
            }
            break;
        case 13:
            if (isdigit(ch)) {
                lexeme[lexIndex++] = ch;
            } else if (ch == '.') {
                state = 14;
                lexeme[lexIndex++] = ch;
            } else if (ch == 'E' || ch == 'e') {
                state = 16;
                lexeme[lexIndex++] = ch;
            } else {
                state = 0;
                lexeme[lexIndex] = '\0';
                printf("%s is a valid number\n", lexeme);
                lexIndex = 0;
            }
            break;
```

case 14:

```
    if (isdigit(ch)) {  
        state = 15;  
        lexeme[lexIndex++] = ch;  
    } else {  
        state = 0;  
        lexIndex = 0;  
    }  
    break;
```

case 15:

```
    if (isdigit(ch)) {  
        lexeme[lexIndex++] = ch;  
    } else if (ch == 'E' || ch == 'e') {  
        state = 16;  
        lexeme[lexIndex++] = ch;  
    } else {  
        state = 0;  
        lexeme[lexIndex] = '\0';  
        printf("%s is a valid number\n", lexeme);  
        lexIndex = 0;  
    }  
    break;
```

case 16:

```
    if (ch == '+' || ch == '-') {  
        state = 17;  
        lexeme[lexIndex++] = ch;
```

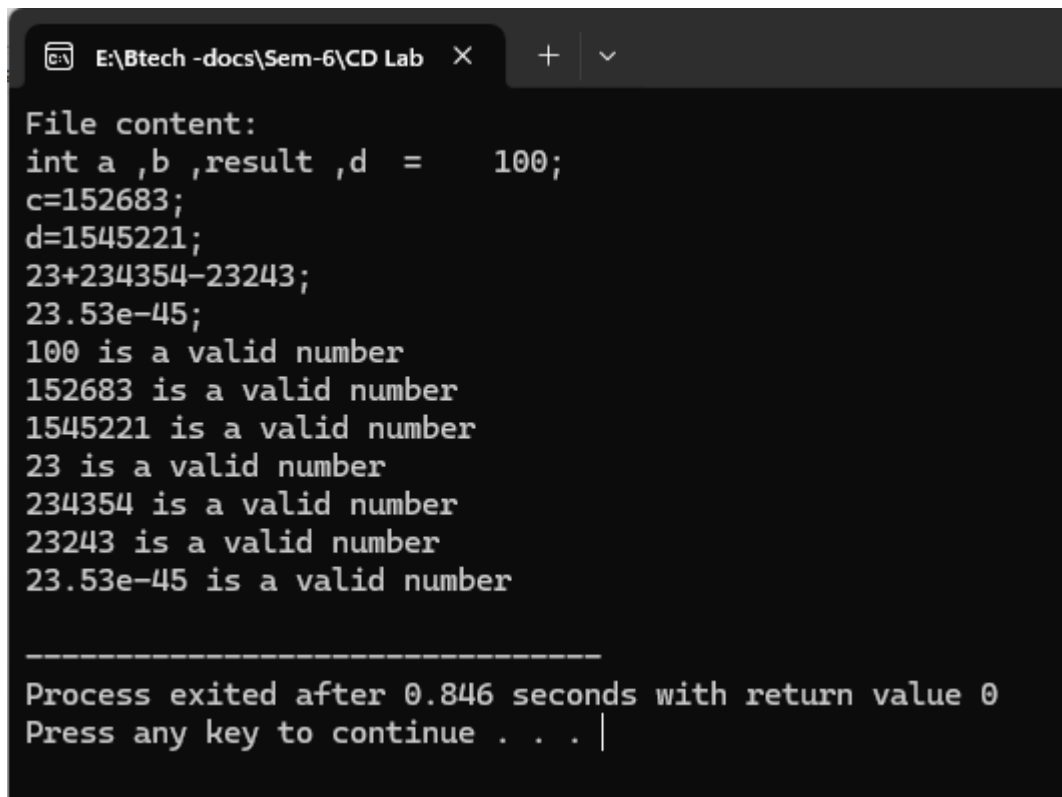


```
    } else if (isdigit(ch)) {
        state = 18;
        lexeme[lexIndex++] = ch;
    } else {
        state = 0;
        lexIndex = 0;
    }
    break;
case 17:
    if (isdigit(ch)) {
        state = 18;
        lexeme[lexIndex++] = ch;
    } else {
        state = 0;
        lexIndex = 0;
    }
    break;
case 18:
    if (isdigit(ch)) {
        lexeme[lexIndex++] = ch;
    } else {
        state = 0;
        lexeme[lexIndex] = '\0';
        printf("%s is a valid number\n", lexeme);
        lexIndex = 0;
    }
}
```

```
        break;
    }
    i++;
}

return 0;
}
```

Output –



```
File content:
int a ,b ,result ,d = 100;
c=152683;
d=1545221;
23+234354-23243;
23.53e-45;
100 is a valid number
152683 is a valid number
1545221 is a valid number
23 is a valid number
234354 is a valid number
23243 is a valid number
23.53e-45 is a valid number

-----
Process exited after 0.846 seconds with return value 0
Press any key to continue . . . |
```

d) Write a program to recognize the valid comments.

Code –

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

```
int main() {
    int state = 0;
    char input[200];
    FILE *file;

    file = fopen("comments.txt", "r");
    if (file == NULL) {
        printf("Error: Could not open the file.\n");
        return 1;
    }

    printf("Reading from the file:\n");

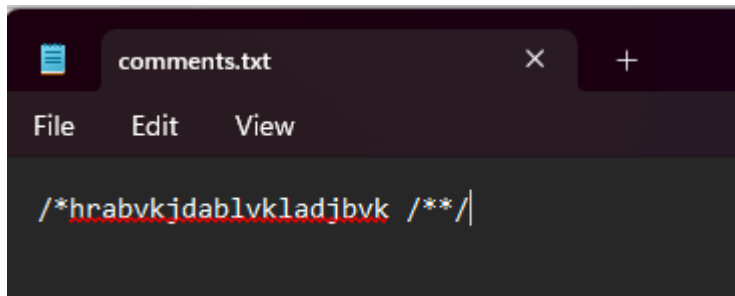
    while (fgets(input, sizeof(input), file) != NULL) {
        int i = 0;
        printf("%s", input); // Print each line for reference

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

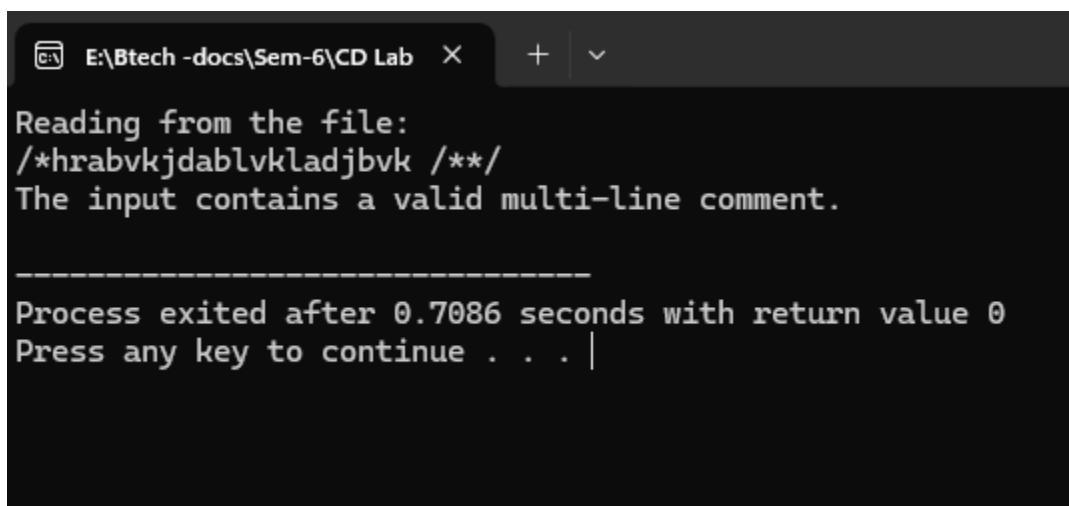
```
        if (input[i] == '/')
            state = 2;
        else if (input[i] == '*')
            state = 4;
        else
            state = 3;
        break;
    case 2:
        if (input[i] == '\n')
            state = 3;
        break;
    case 4:
        if (input[i] == '*')
            state = 5;
        break;
    case 5:
        if (input[i] == '/')
            state = 6;
        else if (input[i] != '*')
            state = 4;
        break;
    }
    i++;
}
}
fclose(file);
```

```
if (state == 2) {  
    printf("\nThe input contains a valid single-line comment.\n");  
} else if (state == 6) {  
    printf("\nThe input contains a valid multi-line comment.\n");  
} else {  
    printf("\nThe input does not contain a valid comment.\n");  
}  
return 0;  
}
```

The text file I am using is as following:



Output –



e) Program to implement Lexical Analyzer.

Code –

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

#define MAX_BUFFER_SIZE 1024

int cmp_keyword(const void *key, const void *element) {
    return strcmp((const char *)key, (const char *)element);
}

int main() {
    int state = 0;
    char input[MAX_BUFFER_SIZE];

    char keywords[][15] = {
        "auto", "break", "case", "char", "const", "continue", "default", "do",
        "double", "else", "enum", "extern", "float", "for", "goto", "if",
        "int", "long", "register", "return", "short", "signed", "sizeof",
        "static", "struct", "switch", "typedef", "union", "unsigned", "void",
        "volatile", "while"
    };

    int numKeywords = sizeof(keywords) / sizeof(keywords[0]);
```

```
FILE *file = fopen("lextest.txt", "r");  
if (file == NULL) {  
    printf("Error: Cannot open file.\n");  
    return 1;  
}
```

```
size_t bytesRead = fread(input, sizeof(char), sizeof(input) - 1, file);  
input[bytesRead] = '\0';  
fclose(file);
```

```
char token[50];  
int tokenIndex = 0;  
int lexIndex = 0;  
int i = 0;  
char lexeme[100];  
char ch;
```

```
while (input[i] != '\0') {  
    ch = input[i];  
    switch (state) {  
        case 0:  
            tokenIndex = 0;  
            lexIndex = 0;  
            if (isalpha(ch) || ch == '_') {  
                state = 1;  
                token[tokenIndex++] = ch;
```

```
} else if (ch == ' ' || ch == '\t' || ch == '\n') {  
    /* Skip whitespace */  
} else if (ch == '/') {  
    if (input[i + 1] == '/' || input[i + 1] == '*') {  
        state = 7;  
    } else {  
        state = 59;  
        i--;  
    }  
} else if (isdigit(ch)) {  
    state = 13;  
    lexeme[lexIndex++] = ch;  
} else if (ch == '+') state = 51;  
else if (ch == '-') state = 54;  
else if (ch == '*') state = 57;  
else if (ch == '%') state = 61;  
else if (ch == '=') state = 63;  
else if (ch == '<') state = 65;  
else if (ch == '>') state = 68;  
else if (ch == '!') state = 71;  
else if (ch == '&') state = 73;  
else if (ch == '|') state = 75;  
else if (ch == '~') printf("Bitwise operator [~].\n");  
else if (ch == '^') printf("Bitwise operator [^].\n");  
else printf("Character: %c\n", ch);  
break;
```


case 1:

```
    if (ch == 'n') {  
        state = 2;  
        token[tokenIndex++] = ch;  
    } else if (isalnum(ch) || ch == '_') {  
        state = 4;  
        token[tokenIndex++] = ch;  
    } else {  
        state = 0;  
        i--;  
    }  
    break;
```

case 2:

```
    if (ch == 't') {  
        state = 3;  
        token[tokenIndex++] = ch;  
    } else if (isalnum(ch) || ch == '_') {  
        state = 4;  
        token[tokenIndex++] = ch;  
    } else {  
        state = 0;  
        i--;  
    }  
    break;
```

case 3:

```
    if (!isalnum(ch) && ch != '_') {  
        printf("\nToken: int -> Keyword (Valid)\n");  
        state = 0;  
        i--;  
    } else {  
        state = 4;  
        token[tokenIndex++] = ch;  
    }  
    break;
```

case 4:

```
    if (isalnum(ch) || ch == '_') {  
        token[tokenIndex++] = ch;  
    } else {  
        token[tokenIndex] = '\0';  
        char *found = (char *)bsearch(token, keywords,  
numKeywords,  
                                sizeof(keywords[0]), cmp_keyword);  
        if (found) printf("Token: %s -> Keyword (Valid)\n", token);  
        else printf("Token: %s -> Identifier (Valid)\n", token);  
        state = 0;  
        i--;  
    }  
    break;
```

case 7:

```
    if (ch == '/') {  
        state = 8;  
        printf("Single-line comment: //");  
    } else if (ch == '*') {  
        state = 9;  
        printf("Multi-line comment: /*");  
    } else {  
        state = 59;  
        i--;  
    }  
    break;
```

case 8:

```
    if (ch == '\n' || ch == '\0') {  
        printf("\n");  
        state = 0;  
    } else printf("%c", ch);  
    break;
```

case 9:

```
    if (ch == '\0') {  
        printf("\nError: Unterminated multi-line comment\n");  
        state = 0;  
        break;  
    }
```

```
printf("%c", ch);  
if (ch == '*') state = 10;  
break;
```

case 10:

```
if (ch == '\0') {  
    printf("\nError: Unterminated multi-line comment\n");  
    state = 0;  
    break;  
}  
printf("%c", ch);  
if (ch == '/') state = 0;  
else if (ch != '*') state = 9;  
break;
```

case 13:

```
if (isdigit(ch)) {  
    lexeme[lexIndex++] = ch;  
} else if (ch == '.') {  
    state = 14;  
    lexeme[lexIndex++] = ch;  
} else if (ch == 'E' || ch == 'e') {  
    state = 16;  
    lexeme[lexIndex++] = ch;  
} else {  
    lexeme[lexIndex] = '\0';
```

```
    printf("%s is a valid number\n", lexeme);  
    state = 0;  
    lexIndex = 0;  
    i--;  
}  
break;
```

case 14:

```
    if (isdigit(ch)) {  
        state = 15;  
        lexeme[lexIndex++] = ch;  
    } else {  
        lexeme[lexIndex] = '\0';  
        printf("Error: Invalid number format: %s\n", lexeme);  
        state = 0;  
        lexIndex = 0;  
        i--;  
    }  
break;
```

case 15:

```
    if (isdigit(ch)) {  
        lexeme[lexIndex++] = ch;  
    } else if (ch == 'E' || ch == 'e') {  
        state = 16;  
        lexeme[lexIndex++] = ch;
```

```
} else {  
    lexeme[lexIndex] = '\\0';  
    printf("%s is a valid number\\n", lexeme);  
    state = 0;  
    lexIndex = 0;  
    i--;  
}  
break;
```

case 16:

```
if (ch == '+' || ch == '-') {  
    state = 17;  
    lexeme[lexIndex++] = ch;  
} else if (isdigit(ch)) {  
    state = 18;  
    lexeme[lexIndex++] = ch;  
} else {  
    lexeme[lexIndex] = '\\0';  
    printf("Error: Invalid scientific notation: %s\\n", lexeme);  
    state = 0;  
    lexIndex = 0;  
    i--;  
}  
break;
```

case 17:

```
if (isdigit(ch)) {  
    state = 18;  
    lexeme[lexIndex++] = ch;  
} else {  
    lexeme[lexIndex] = '\\0';  
    printf("Error: Invalid scientific notation: %s\\n", lexeme);  
    state = 0;  
    lexIndex = 0;  
    i--;  
}  
break;
```

case 18:

```
if (isdigit(ch)) {  
    lexeme[lexIndex++] = ch;  
} else {  
    lexeme[lexIndex] = '\\0';  
    printf("%s is a valid number\\n", lexeme);  
    state = 0;  
    lexIndex = 0;  
    i--;  
}  
break;
```

case 51:

```
if (input[i] == '+') printf("Unary operator: ++\\n");
```

```
else if (input[i] == '=') printf("Assignment operator: +=\n");
else {
    printf("Arithmetic operator [ + ].\n");
    i--;
}
state = 0;
break;
```

case 54:

```
if (input[i] == '-') printf("Unary operator: --\n");
else if (input[i] == '=') printf("Assignment operator: -=\n");
else {
    printf("Arithmetic operator [ - ].\n");
    i--;
}
state = 0;
break;
```

case 57:

```
if (input[i] == '=') printf("Assignment operator: *=\n");
else {
    printf("Arithmetic operator [ * ].\n");
    i--;
}
state = 0;
break;
```


case 59:

```
if (input[i] == '=') printf("Assignment operator: /=\n");
else {
    printf("Arithmetic operator [/.]\n");
    i--;
}
state = 0;
break;
```

case 61:

```
if (input[i] == '=') printf("Assignment operator: %%=\n");
else {
    printf("Arithmetic operator [%%].\n");
    i--;
}
state = 0;
break;
```

case 63:

```
if (input[i] == '=') printf("Relational operator: ==\n");
else {
    printf("Assignment operator [=].\n");
    i--;
}
state = 0;
```

```
break;
```

```
case 65:
```

```
    if (input[i] == '=') printf("Relational operator: <=\n");
```

```
    else {
```

```
        printf("Relational operator [<].\n");
```

```
        i--;
```

```
    }
```

```
    state = 0;
```

```
    break;
```

```
case 68:
```

```
    if (input[i] == '=') printf("Relational operator: >=\n");
```

```
    else {
```

```
        printf("Relational operator [>].\n");
```

```
        i--;
```

```
    }
```

```
    state = 0;
```

```
    break;
```

```
case 71:
```

```
    if (input[i] == '=') printf("Relational operator: !=\n");
```

```
    else {
```

```
        printf("Logical NOT operator [!].\n");
```

```
        i--;
```

```
    }
```

```
state = 0;
```

```
break;
```

```
case 73:
```

```
if (input[i] == '&') printf("Logical operator: &\n");
```

```
else if (input[i] == '=') printf("Assignment operator: &=\n");
```

```
else {
```

```
    printf("Bitwise operator [&].\n");
```

```
    i--;
```

```
}
```

```
state = 0;
```

```
break;
```

```
case 75:
```

```
if (input[i] == '|') printf("Logical operator: ||\n");
```

```
else if (input[i] == '=') printf("Assignment operator: |=\n");
```

```
else {
```

```
    printf("Bitwise operator [||].\n");
```

```
    i--;
```

```
}
```

```
state = 0;
```

```
break;
```

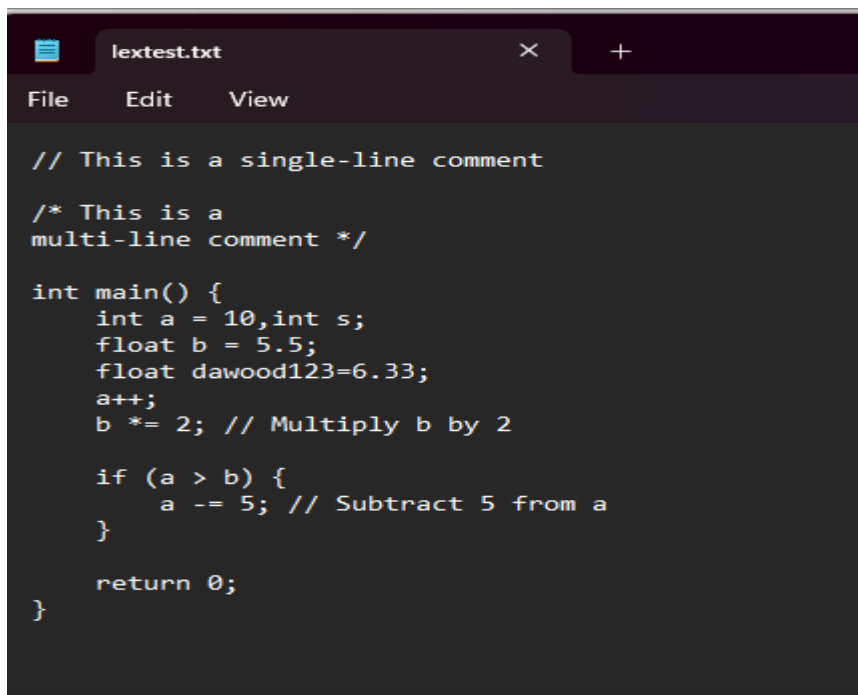
```
default:
```

```
state = 0;
```

```
break;
```

```
    }  
    i++;  
}  
  
// Handle remaining token at end of input  
if (tokenIndex > 0) {  
    token[tokenIndex] = '\0';  
    char *found = (char *)bsearch(token, keywords, numKeywords,  
                                   sizeof(keywords[0]), cmp_keyword);  
    if (found) printf("Token: %s -> Keyword (Valid)\n", token);  
    else printf("Token: %s -> Identifier (Valid)\n", token);  
}  
return 0;  
}
```

The input file I am using is named as lextest.txt which contains content as:



```
lextest.txt  
File Edit View  
  
// This is a single-line comment  
  
/* This is a  
multi-line comment */  
  
int main() {  
    int a = 10, int s;  
    float b = 5.5;  
    float dawood123=6.33;  
    a++;  
    b *= 2; // Multiply b by 2  
  
    if (a > b) {  
        a -= 5; // Subtract 5 from a  
    }  
  
    return 0;  
}
```

Output –

```
E:\Btech -docs\Sem-6\CD Lab  X  +  v
Single-line comment: // This is a single-line comment
Multi-line comment: /* This is a
multi-line comment */
Token: int -> Keyword (Valid)
Token: main -> Identifier (Valid)
Character: (
Character: )
Character: {

Token: int -> Keyword (Valid)
Assignment operator [=].
10 is a valid number
Character: ,

Token: int -> Keyword (Valid)
Character: ;
Token: float -> Keyword (Valid)
Assignment operator [=].
5.5 is a valid number
Character: ;
Token: float -> Keyword (Valid)
Token: dawood123 -> Identifier (Valid)
Assignment operator [=].
6.33 is a valid number
Character: ;
Unary operator: ++
Character: ;
Assignment operator: *=
2 is a valid number
Character: ;
Single-line comment: // Multiply b by 2
Token: if -> Keyword (Valid)
Character: (
Relational operator [>].
Character: )
```

```
Character: {  
Assignment operator: -=  
5 is a valid number  
Character: ;  
Single-line comment: // Subtract 5 from a  
Character: }  
Token: return -> Keyword (Valid)  
0 is a valid number  
Character: ;  
Character: }  
  
-----  
Process exited after 2.174 seconds with return value 0  
Press any key to continue . . . |
```

Program – 3

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

Introduction

Lex and Flex are tools used in compiler design for **automating the creation of lexical analyzers**, also known as *scanners* or *tokenizers*. These tools help in breaking down source code into meaningful tokens such as identifiers, keywords, operators, and symbols.

LEX

- **Lex** is a classic Unix tool designed to generate C code for lexical analyzers.
- It takes a set of regular expressions and actions written by the programmer and produces a C program that can scan and identify tokens.
- It works closely with **Yacc**, a parser generator.

Flex

- **Flex** is an improved, faster, and more modern version of Lex.
- It is backward-compatible with Lex, but faster and more portable.
- Flex generates a scanner in C, which reads an input stream and performs pattern matching.

How It Works

A Lex/Flex program has three sections:

1. **Definitions** – Declare variables or regular expressions.
2. **Rules** – List of regular expressions and corresponding C code to execute.
3. **User Code** – Additional C functions (like main()).

Example

```
%%
```

```
[0-9]+ { printf("Number: %s\n", yytext); }
```

```
[a-zA-Z]+ { printf("Word: %s\n", yytext); }  
%%  
int main() {  
    yylex();  
    return 0;  
}
```

This program identifies numbers and words from input.

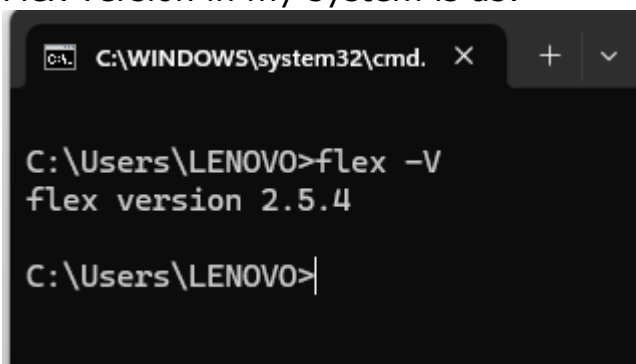
Why Use Lex/Flex

- Reduces manual coding effort.
- Handles complex pattern matching.
- Ensures better error handling and performance.

Conclusion

Lex and Flex are foundational tools in compiler construction and text processing. They allow developers to build robust scanners quickly and efficiently by using regular expressions to define token patterns.

Flex version in my system is as:



```
C:\WINDOWS\system32\cmd. X + v  
C:\Users\LENOVO>flex -V  
flex version 2.5.4  
C:\Users\LENOVO>
```


Program – 4

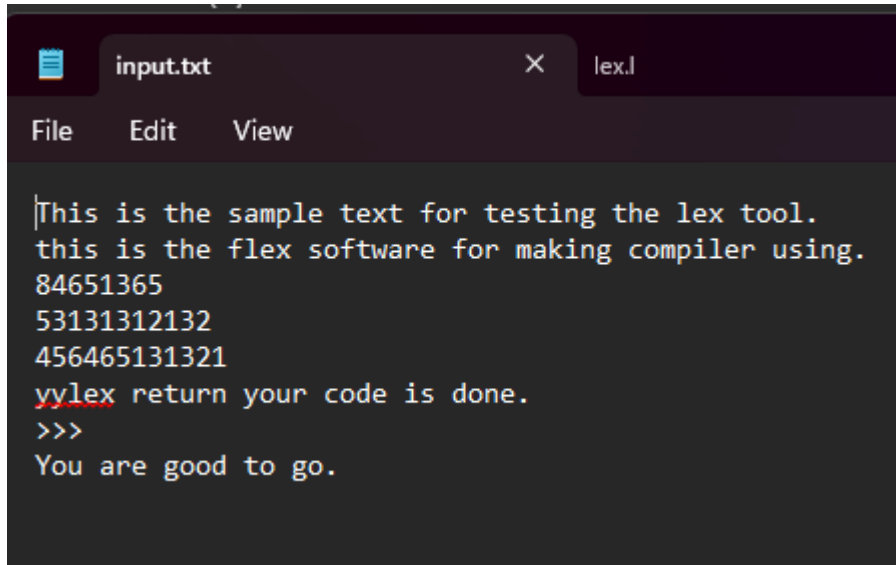
Implement following programs using Lex.

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

Lex code –

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

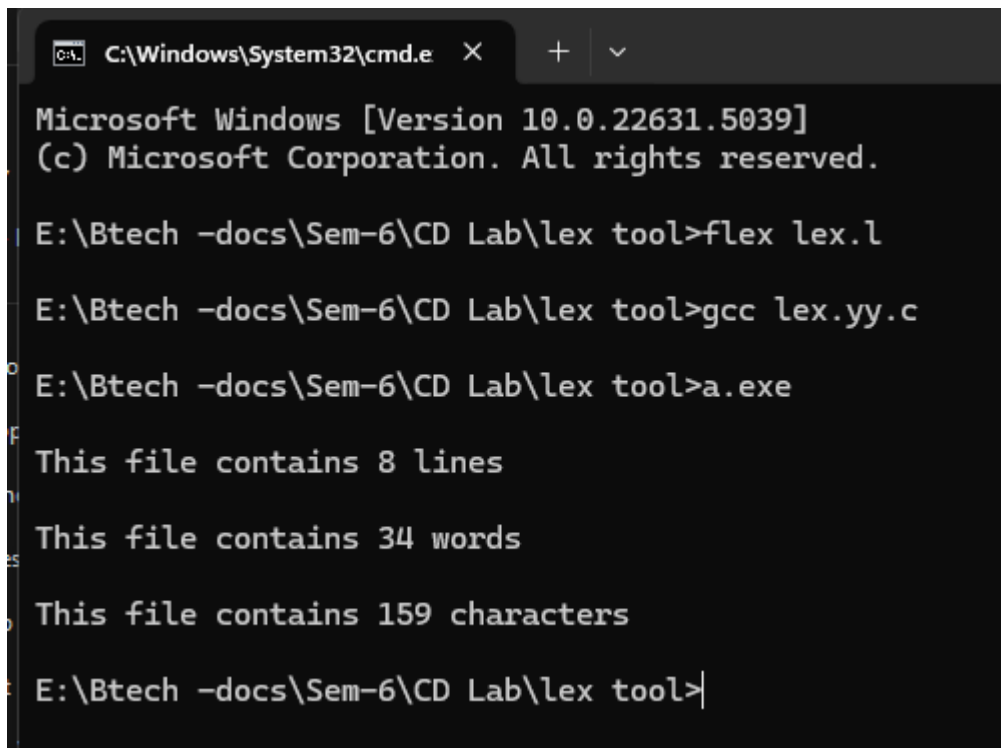
Input file used –



The screenshot shows a text editor window with a dark theme. The title bar at the top indicates the file is 'input.txt'. The menu bar includes 'File', 'Edit', and 'View'. The text content of the file is as follows:

```
|This is the sample text for testing the lex tool.  
this is the flex software for making compiler using.  
84651365  
53131312132  
456465131321  
vylex return your code is done.  
>>>  
You are good to go.
```

Output –



The screenshot shows a Windows command prompt window. The title bar indicates the path 'C:\Windows\System32\cmd.e'. The text content of the window is as follows:

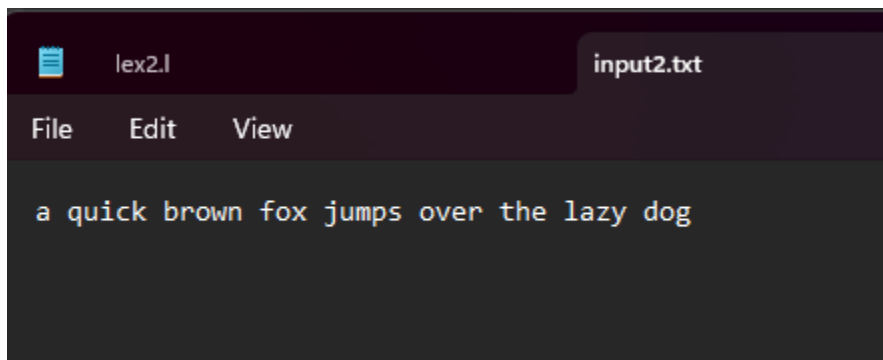
```
Microsoft Windows [Version 10.0.22631.5039]  
(c) Microsoft Corporation. All rights reserved.  
  
E:\Btech -docs\Sem-6\CD Lab\lex tool>flex lex.l  
  
E:\Btech -docs\Sem-6\CD Lab\lex tool>gcc lex.yy.c  
  
E:\Btech -docs\Sem-6\CD Lab\lex tool>a.exe  
  
This file contains 8 lines  
  
This file contains 34 words  
  
This file contains 159 characters  
  
E:\Btech -docs\Sem-6\CD Lab\lex tool>|
```

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

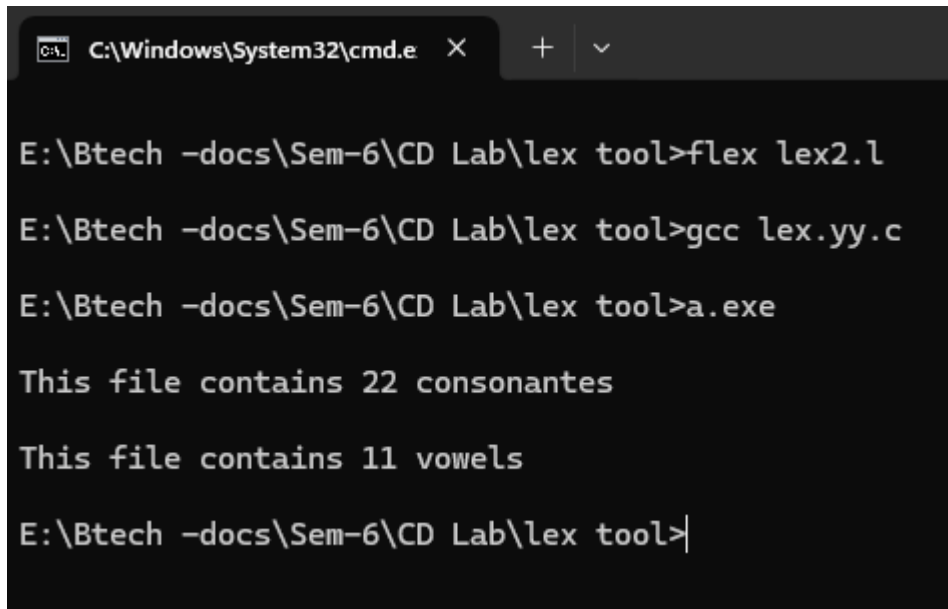
Lex code –

```
%{
#include<stdio.h>
int v=0,c=0;
}%
%%
[aeiouAEIOU] v++;
[a-zA-Z] c++;
. ;
%%
void main(){
    yyin=fopen("input2.txt","r");
    yylex();
    printf("\nThis file contains %d consonantes\n",c);
    printf("\nThis file contains %d vowels\n",v);
}
int yywrap(){ return(1);}
```

Input file used –



Output –



```
C:\Windows\System32\cmd.e X + v
E:\Btech -docs\Sem-6\CD Lab\lex tool>flex lex2.l
E:\Btech -docs\Sem-6\CD Lab\lex tool>gcc lex.yy.c
E:\Btech -docs\Sem-6\CD Lab\lex tool>a.exe
This file contains 22 consonantes
This file contains 11 vowels
E:\Btech -docs\Sem-6\CD Lab\lex tool>
```

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

Lex code –

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

%%

[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)? printf("This %s is valid number
\n",yytext);

\n ;

. ;

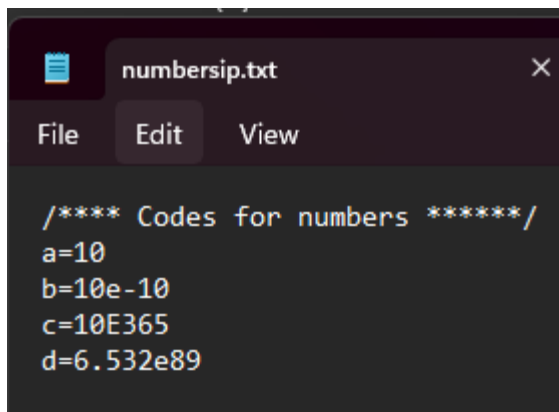
%%

int main()
{
```

```
yyin=fopen("numbersip.txt","r");  
yylex();  
return 0;  
}
```

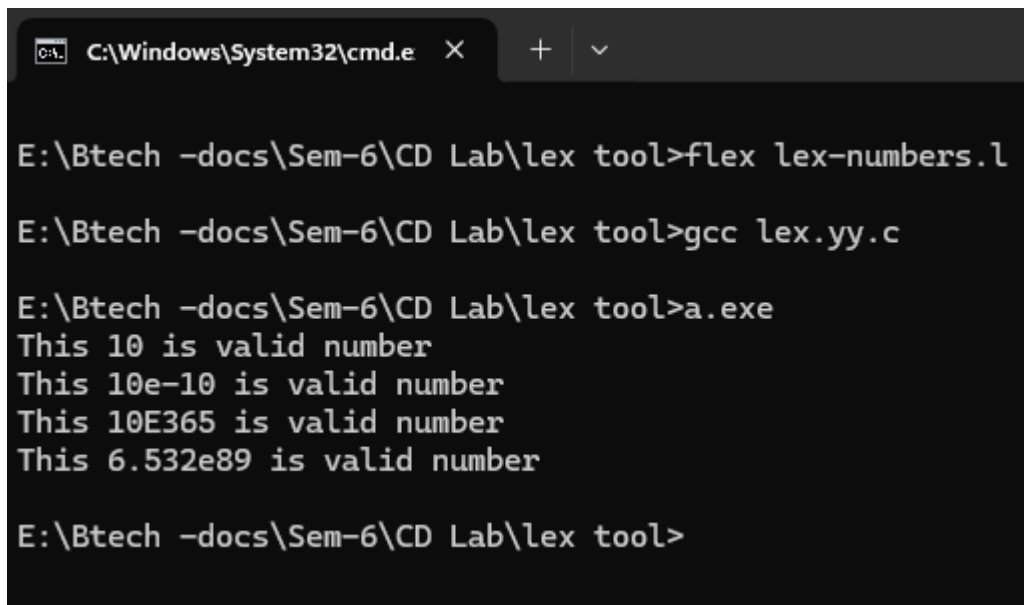
```
int yywrap()  
{return(1);}
```

Input file used –



```
File Edit View  
/**** Codes for numbers *****/  
a=10  
b=10e-10  
c=10E365  
d=6.532e89
```

Output –



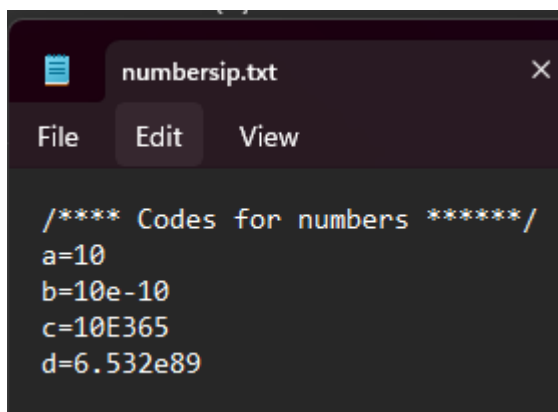
```
C:\Windows\System32\cmd.e X + v  
E:\Btech -docs\Sem-6\CD Lab\lex tool>flex lex-numbers.l  
E:\Btech -docs\Sem-6\CD Lab\lex tool>gcc lex.yy.c  
E:\Btech -docs\Sem-6\CD Lab\lex tool>a.exe  
This 10 is valid number  
This 10e-10 is valid number  
This 10E365 is valid number  
This 6.532e89 is valid number  
E:\Btech -docs\Sem-6\CD Lab\lex tool>
```

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

Lex code –

```
%{  
int line_nums=1;  
%}  
%%  
.+ {fprintf(yyout,"%d: %s",line_nums,yytext);line_nums++;}  
%%  
  
int main()  
{  
yyin=fopen("numbersip.txt","r");  
yyout=fopen("opwithlines.txt","w");  
yylex();  
printf("File writing with line numbers is done.");  
return 0;  
}  
  
int yywrap(){return(1);}
```

Input file used –



The screenshot shows a text editor window with the title 'numbersip.txt'. The menu bar includes 'File', 'Edit', and 'View'. The text content of the file is as follows:

```
/* Codes for numbers */  
a=10  
b=10e-10  
c=10E365  
d=6.532e89
```

Output –

```
C:\Windows\System32\cmd.e X + v
Microsoft Windows [Version 10.0.22631.5039]
(c) Microsoft Corporation. All rights reserved.

E:\Btech -docs\Sem-6\CD Lab\lex tool>flex addlinenums.l

E:\Btech -docs\Sem-6\CD Lab\lex tool>gcc lex.yy.c

E:\Btech -docs\Sem-6\CD Lab\lex tool>a.exe
File writing with line numbers is done.
E:\Btech -docs\Sem-6\CD Lab\lex tool>
```

```
opwithlines.txt X ad
File Edit View

1: /**** Codes for numbers *****/
2: a=10
3: b=10e-10
4: c=10E365
5: d=6.532e89
```

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

Lex code –

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

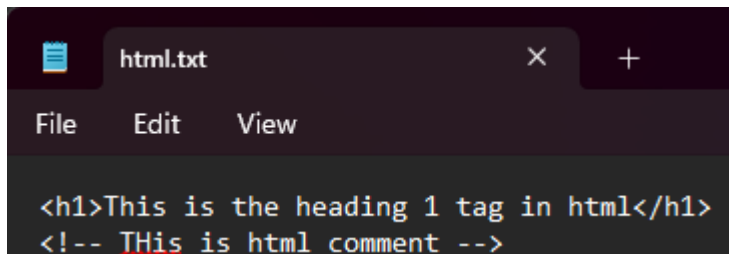
int num=0;
}%

%%

"<"[A-Za-z0-9]+>" printf("%s is valid mamrkup tag.\n",yytext);
```

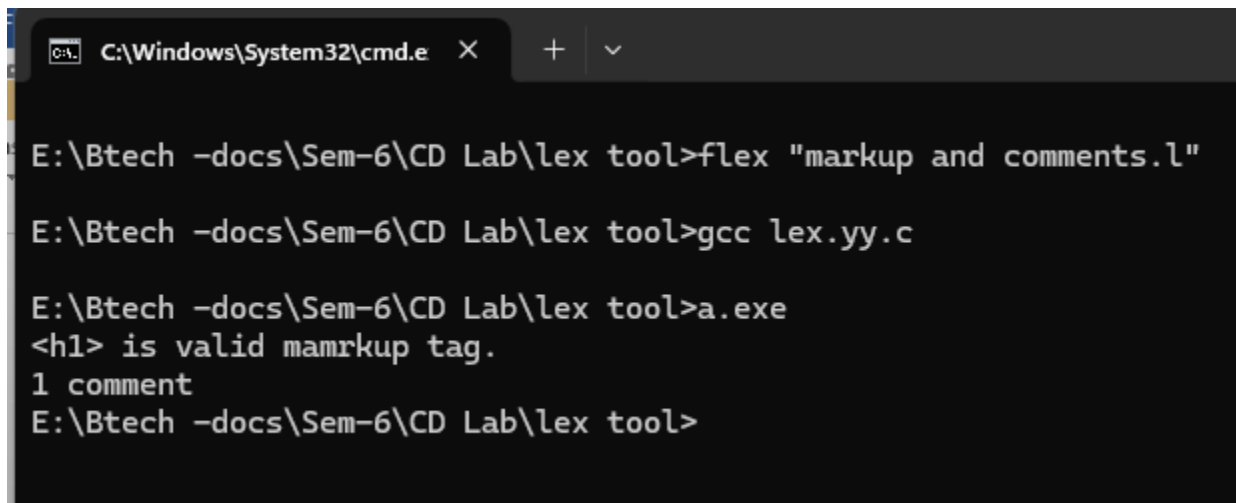
```
"<!--"(.|\n)*"-->" num++;  
\n ;  
.  
%;  
int main()  
{  
yyin=fopen("html.txt","r");  
yylex();  
printf("%d comment",num);  
return 0;  
}  
int yywrap() {return(1);}
```

Input file used –



A screenshot of a text editor window with a dark theme. The title bar shows 'html.txt'. The menu bar has 'File', 'Edit', and 'View'. The text content is: `<h1>This is the heading 1 tag in html</h1>` followed by `<!-- This is html comment -->` on the next line. The word 'This' in the comment is underlined in red.

Output –



A screenshot of a Windows command prompt window. The title bar shows 'C:\Windows\System32\cmd.e'. The command history shows the following commands and their outputs:
1. `E:\Btech -docs\Sem-6\CD Lab\lex tool>flex "markup and comments.l"`
2. `E:\Btech -docs\Sem-6\CD Lab\lex tool>gcc lex.yy.c`
3. `E:\Btech -docs\Sem-6\CD Lab\lex tool>a.exe`
The output of the third command is:
`<h1> is valid mamrkup tag.
1 comment`
The prompt then returns to `E:\Btech -docs\Sem-6\CD Lab\lex tool>`.

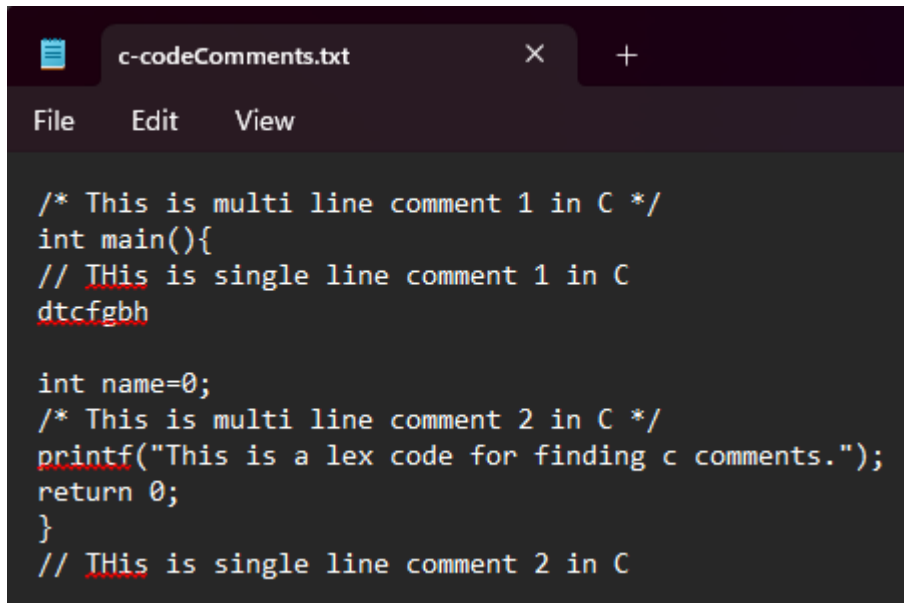
Program – 5

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.

Lex Code –

```
%{  
# include<stdio.h>  
int c=0;  
%}  
  
%%  
"/"^[^*/]*"/" {fprintf(yyout,"");c++;}  
"/"/* {fprintf(yyout," ");c++;}  
.* fprintf(yyout,"%s",yytext);  
%%  
  
int main()  
{  
yyin=fopen("c-codeComments.txt","r");  
yyout=fopen("c-out.txt","w");  
yylex();  
printf("%d comments are there in file.",c);  
}  
int yywrap()  
{return(1);}
```

Input file used –

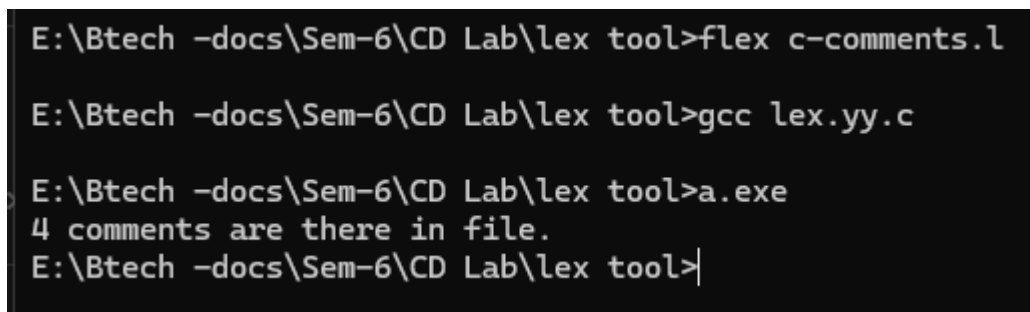


```
c-codeComments.txt
File Edit View

/* This is multi line comment 1 in C */
int main(){
// This is single line comment 1 in C
dttcfgh

int name=0;
/* This is multi line comment 2 in C */
printf("This is a lex code for finding c comments.");
return 0;
}
// This is single line comment 2 in C
```

Output –



```
E:\Btech -docs\Sem-6\CD Lab\lex tool>flex c-comments.l
E:\Btech -docs\Sem-6\CD Lab\lex tool>gcc lex.yy.c
E:\Btech -docs\Sem-6\CD Lab\lex tool>a.exe
4 comments are there in file.
E:\Btech -docs\Sem-6\CD Lab\lex tool>
```

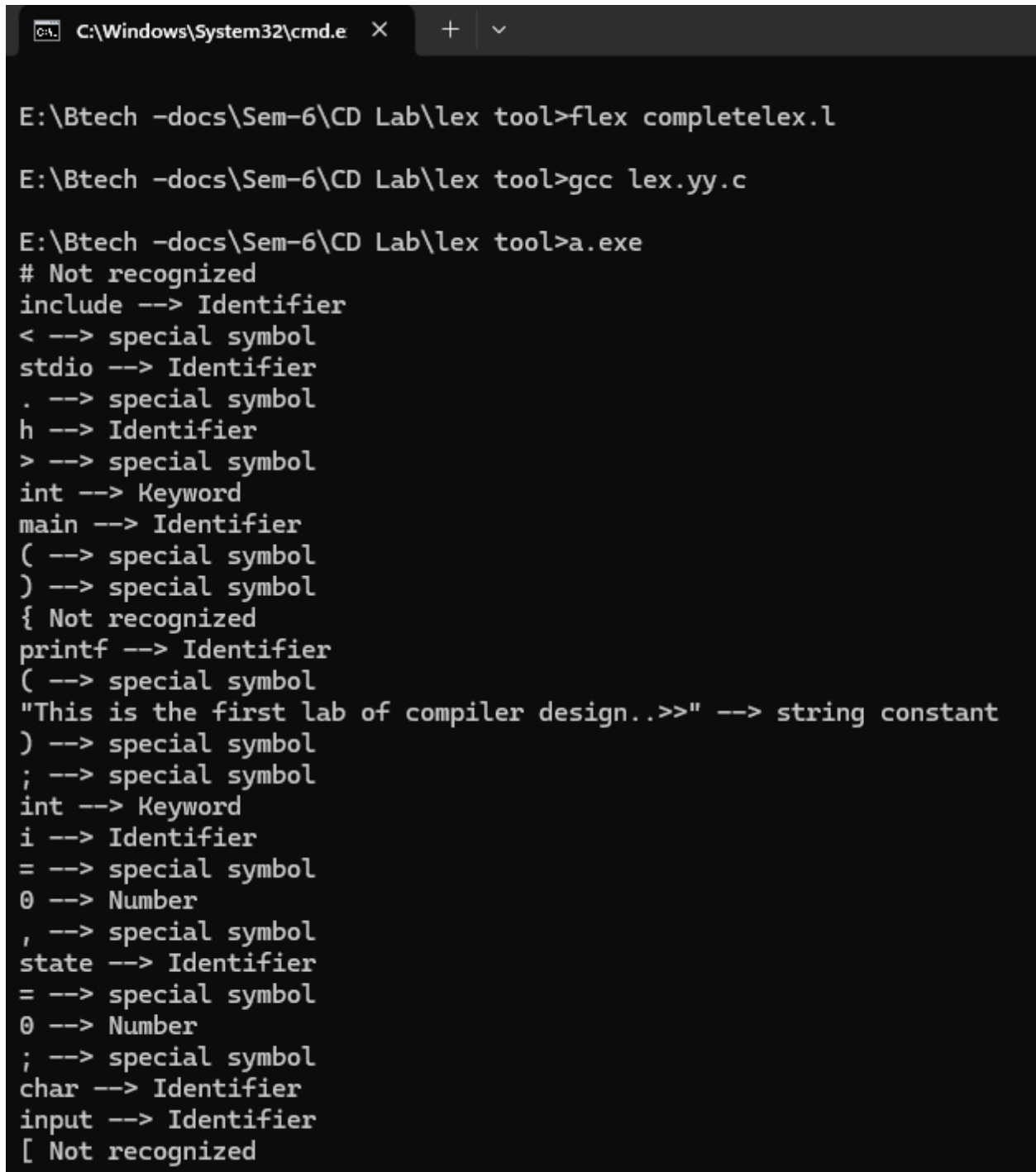
b. Write a Lex program to recognize keywords, identifiers, operators, numbers, special symbols, literals from a given C program.

Lex code –

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

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


Output –



```
C:\Windows\System32\cmd.e X + v

E:\Btech -docs\Sem-6\CD Lab\lex tool>flex completelex.l

E:\Btech -docs\Sem-6\CD Lab\lex tool>gcc lex.yy.c

E:\Btech -docs\Sem-6\CD Lab\lex tool>a.exe
# Not recognized
include --> Identifier
< --> special symbol
stdio --> Identifier
. --> special symbol
h --> Identifier
> --> special symbol
int --> Keyword
main --> Identifier
( --> special symbol
) --> special symbol
{ Not recognized
printf --> Identifier
( --> special symbol
"This is the first lab of compiler design..>" --> string constant
) --> special symbol
; --> special symbol
int --> Keyword
i --> Identifier
= --> special symbol
0 --> Number
, --> special symbol
state --> Identifier
= --> special symbol
0 --> Number
; --> special symbol
char --> Identifier
input --> Identifier
[ Not recognized
```

```
; --> special symbol
printf --> Identifier
( --> special symbol
"\nEnter the input string: " --> string constant
) --> special symbol
; --> special symbol
scanf --> Identifier
( --> special symbol
"%s" --> string constant
, --> special symbol
input --> Identifier
) --> special symbol
; --> special symbol
while --> Keyword
( --> special symbol
input --> Identifier
[ Not recognized
i --> Identifier
] Not recognized
! --> special symbol
= --> special symbol
' Not recognized
\ Not recognized
0 --> Number
' Not recognized
) --> special symbol
{ Not recognized
switch --> Keyword
( --> special symbol
state --> Identifier
) --> special symbol
{ Not recognized
case --> Keyword
0 --> Number
```

```
state --> Identifier
= --> special symbol
= --> special symbol
2 --> Number
| --> special symbol
| --> special symbol
state --> Identifier
= --> special symbol
= --> special symbol
0 --> Number
) --> special symbol
printf --> Identifier
( --> special symbol
"Input string is not valid." --> string constant
) --> special symbol
; --> special symbol
else --> Keyword
if --> Keyword
( --> special symbol
state --> Identifier
= --> special symbol
= --> special symbol
3 --> Number
) --> special symbol
printf --> Identifier
( --> special symbol
"Input string is not recognized" --> string constant
) --> special symbol
; --> special symbol
return --> Keyword
0 --> Number
; --> special symbol
} Not recognized
```

Program – 6

Program to implement Recursive Descent Parsing in C.

Code –

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

char ip[100];
int l = 0;

void match(char t);
void E();
void T();
void F();
void T_dash();
void E_dash();

void match(char t) {
    if (ip[l] == t) {
        l++;
    } else {
        printf("Error\n", ip[l]);
        exit(1);
    }
}

void E() {
```

```
T();  
E_dash();  
}  
  
void T() {  
    F();  
    T_dash();  
}  
  
void F() {  
    if (ip[l] == '(') {  
        match('(');  
        E();  
        match(')');  
    } else if (ip[l] == 'i') {  
        match('i');  
    } else if (ip[l] == 'n') {  
        match('n');  
    } else {  
        printf("Error\n", ip[l]);  
        exit(1);  
    }  
}  
  
void E_dash() {  
    if (ip[l] == '+') {
```



```
    match('+');  
    T();  
    E_dash();  
} else if (ip[l] == '-') {  
    match('-');  
    T();  
    E_dash();  
}  
}
```

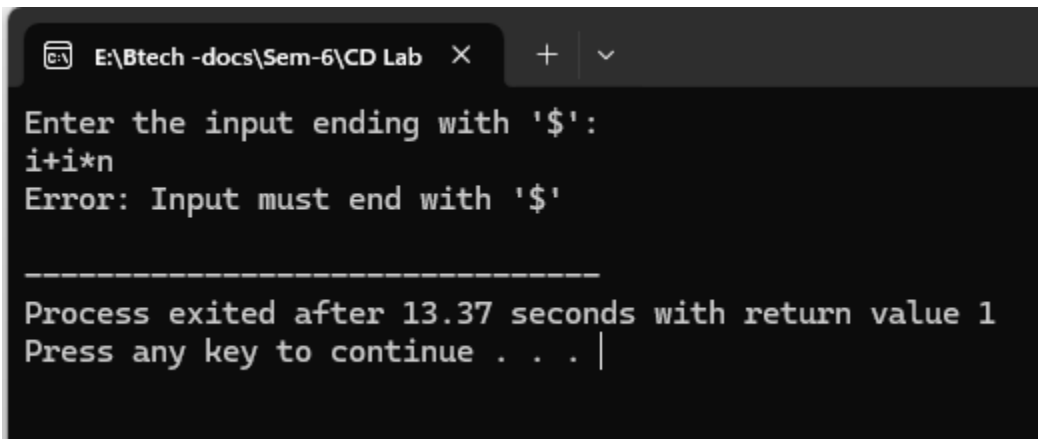
```
void T_dash() {  
    if (ip[l] == '*') {  
        match('*');  
        F();  
        T_dash();  
    } else if (ip[l] == '/') {  
        match('/');  
        F();  
        T_dash();  
    }  
}
```

```
int main() {  
    printf("Enter the input ending with '$':\n");  
    scanf("%s", ip);
```

```
if (ip[strlen(ip) - 1] != '$') {  
    printf("Error: Input must end with '$'\n");  
    return 1;  
}  
  
E();  
  
if (ip[l] == '$') {  
    printf("Parsing successful\n");  
} else {  
    printf("Error: Parsing incomplete at '%c'\n", ip[l]);  
}  
  
return 0;  
}
```

Output –

Error occurs if string does not ends with '\$'.



```
E:\Btech -docs\Sem-6\CD Lab X + v  
Enter the input ending with '$':  
i+i*n  
Error: Input must end with '$'  
-----  
Process exited after 13.37 seconds with return value 1  
Press any key to continue . . . |
```

Parsing supports 'i' and 'n' as identifiers and also supports precedence and '()' based expressions.

```
E:\Btech -docs\Sem-6\CD Lab  X  +  v
Enter the input ending with '$':
i+i*i/n-n+i$
Parsing successful

-----
Process exited after 16.18 seconds with return value 0
Press any key to continue . . . |
```

```
E:\Btech -docs\Sem-6\CD Lab  X  +  v
Enter the input ending with '$':
i+(n*n/n+i-i)*n/i$
Parsing successful

-----
Process exited after 5.766 seconds with return value 0
Press any key to continue . . . |
```

Error occurs when there is no identifier between 2 operators.

```
E:\Btech -docs\Sem-6\CD Lab  X  +  v
Enter the input ending with '$':
i*n-i++n$
Error

-----
Process exited after 8.23 seconds with return value 1
Press any key to continue . . . |
```

Program – 7

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

Yet Another Compiler-Compiler (YACC)

Introduction:

YACC stands for **Yet Another Compiler-Compiler**, a tool used to generate **parsers** – the part of a compiler that checks syntax and builds a syntax tree. Developed by **Stephen C. Johnson** at Bell Labs in the 1970s, YACC takes a formal description of a language's grammar (in **Backus-Naur Form**, BNF) and generates source code in C to parse that language.

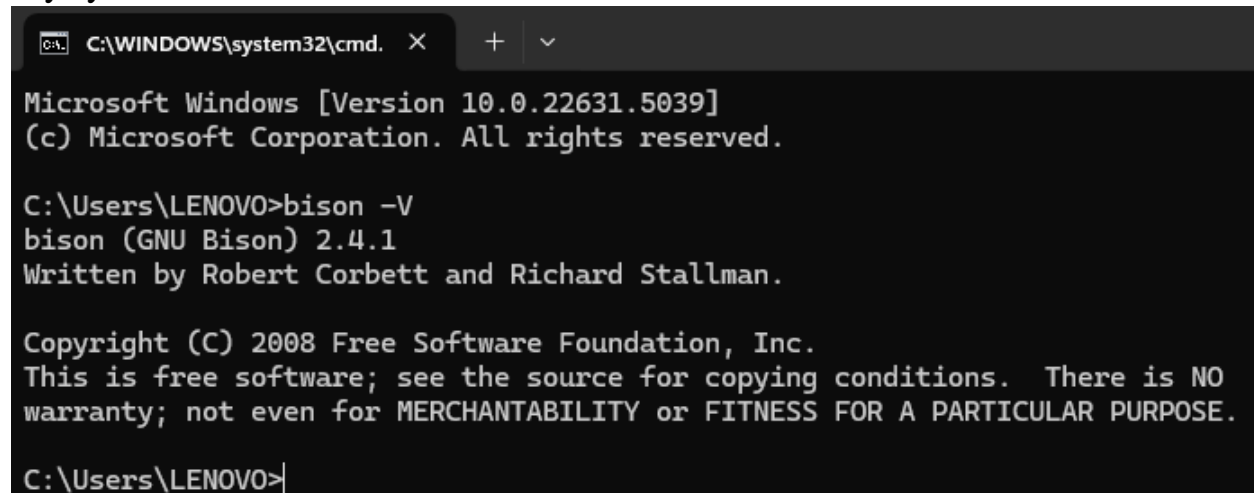
Purpose of YACC:

YACC is primarily used to automate the construction of syntax analyzers for compilers and interpreters. It simplifies the creation of parsers by allowing programmers to define grammar rules and corresponding actions.

How YACC Works:

1. **Input:** A file with grammar rules and C code snippets.
2. **Processing:** YACC translates these rules into a parser (typically in C).
3. **Output:** A C file (often y.tab.c) containing the parser logic.
4. **Integration:** This file is compiled with a lexer (commonly created using **Lex**) to produce the final parser program.

My system has version of YACC as :



```
C:\WINDOWS\system32\cmd. X + v
Microsoft Windows [Version 10.0.22631.5039]
(c) Microsoft Corporation. All rights reserved.

C:\Users\LENOVO>bison -V
bison (GNU Bison) 2.4.1
Written by Robert Corbett and Richard Stallman.

Copyright (C) 2008 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

C:\Users\LENOVO>
```

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

Lex code –

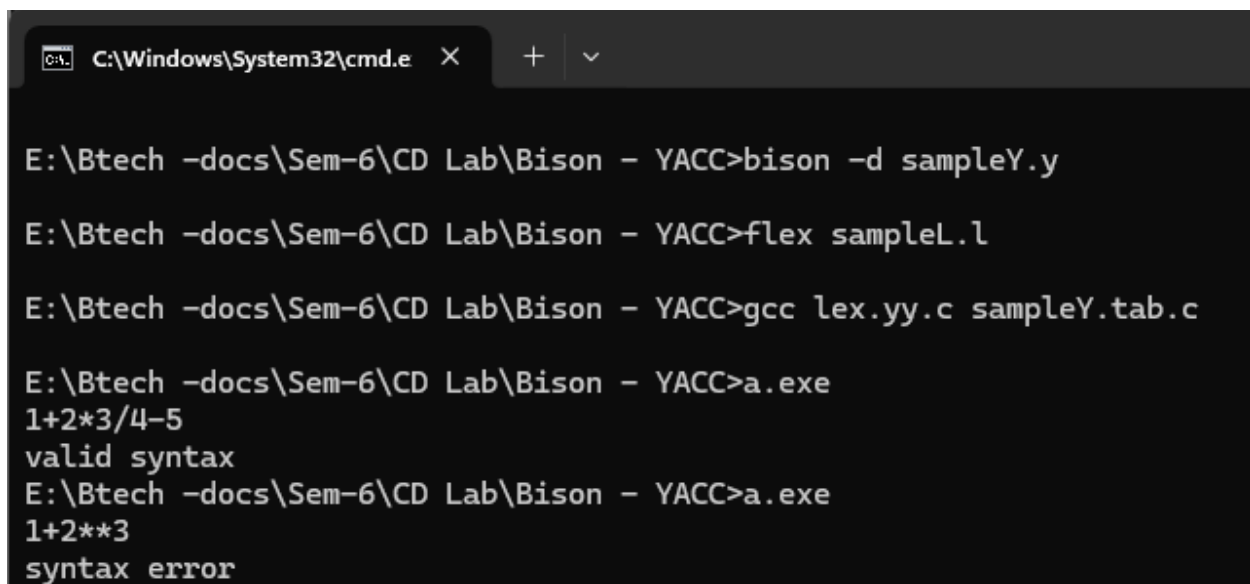
```
%{  
#include <stdlib.h>  
void yyerror(char *);  
#include "sampleY.tab.h"  
%}  
%%  
[0-9]+ return num;  
[-+*/\n] return *yytext;  
[ \t] ;  
. yyerror("invalid character");  
%%  
int yywrap() {  
    return 1;  
}
```

Yacc code –

```
%{  
#include<stdio.h>  
int yylex(void);  
void yyerror(char *);  
%}  
%token num  
%%  
S : E '\n' { printf("valid syntax");return 0; }
```

```
E : E '+' T { }  
    | E '-' T { }  
    | T { }  
T : T '*' F { }  
    | T '/' F { }  
    | F { }  
F : num { }  
%%  
void yyerror(char *s) {  
    printf("%s\n", s);  
}  
int main() {  
    yyparse();return 0;  
}
```

Output –



```
C:\Windows\System32\cmd.e X + v  
  
E:\Btech -docs\Sem-6\CD Lab\Bison - YACC>bison -d sampleY.y  
E:\Btech -docs\Sem-6\CD Lab\Bison - YACC>flex sampleL.l  
E:\Btech -docs\Sem-6\CD Lab\Bison - YACC>gcc lex.yy.c sampleY.tab.c  
E:\Btech -docs\Sem-6\CD Lab\Bison - YACC>a.exe  
1+2*3/4-5  
valid syntax  
E:\Btech -docs\Sem-6\CD Lab\Bison - YACC>a.exe  
1+2**3  
syntax error
```

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

Lex code –

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

Yacc code –

```
%{  
#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; }

 | T '/' F { \$\$ = \$1 / \$3; }

 | F { \$\$ = \$1; }

;

F: '(' E ')' { \$\$ = \$2; }

 | NUM { \$\$ = \$1; }

;

%%

void yyerror(char *s) {

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

}

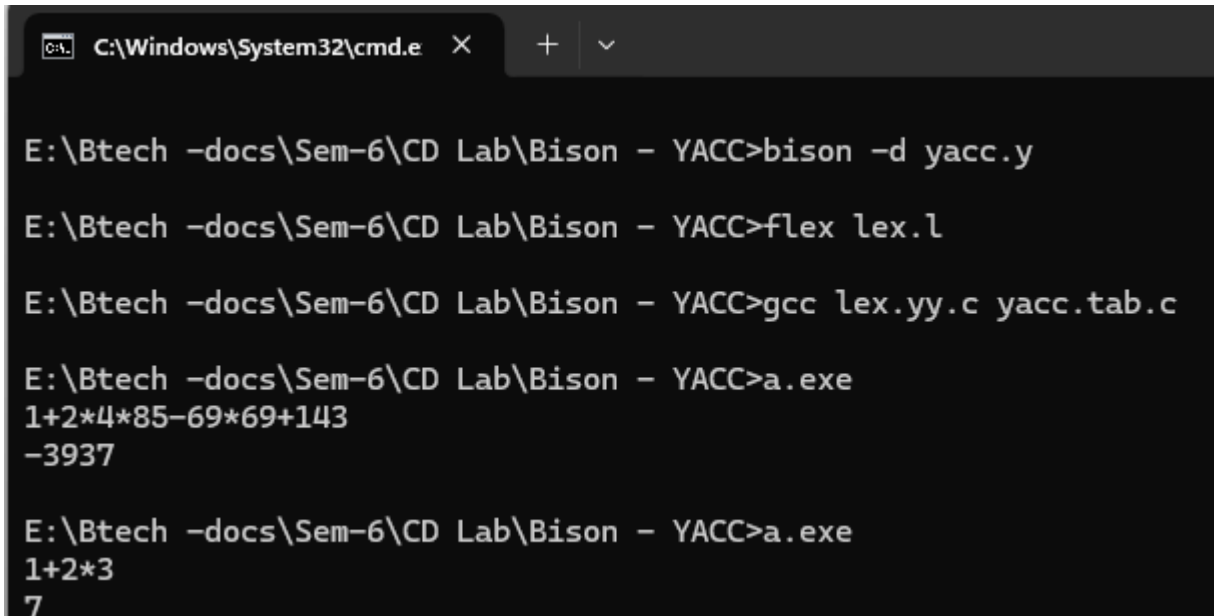
int main() {

 yyparse();

 return 0;

}

Output –



```
C:\Windows\System32\cmd.e X + v

E:\Btech -docs\Sem-6\CD Lab\Bison - YACC>bison -d yacc.y

E:\Btech -docs\Sem-6\CD Lab\Bison - YACC>flex lex.l

E:\Btech -docs\Sem-6\CD Lab\Bison - YACC>gcc lex.yy.c yacc.tab.c

E:\Btech -docs\Sem-6\CD Lab\Bison - YACC>a.exe
1+2*4*85-69*69+143
-3937

E:\Btech -docs\Sem-6\CD Lab\Bison - YACC>a.exe
1+2*3
7
```

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

Lex code –

```
%{
#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;  
}
```

Yacc code –

```
%{  
    #include<stdio.h>  
    int yylex(void);  
    void yyerror(char *);  
}%
```

```
%union {  
    char *str;  
    int num;  
}
```

```
%token <num> INTEGER  
%token <str> ID
```

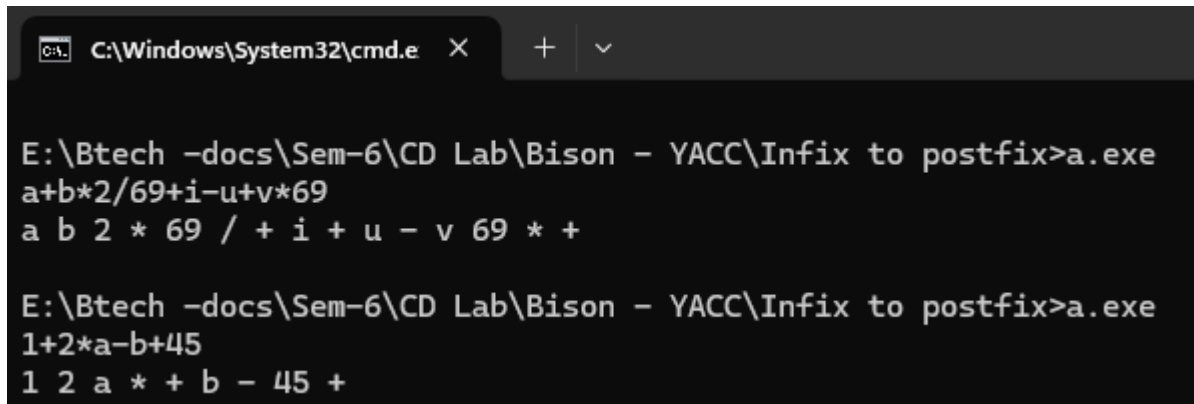
%%

```
S : E '\n' {printf("\n");}  
E : E '+' T {printf("+ ");}  
    | E '-' T {printf("- ");}  
    | T {}  
T : T '*' F {printf("* ");}
```

```
| 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 –



```
C:\Windows\System32\cmd.e  X  +  v  
  
E:\Btech -docs\Sem-6\CD Lab\Bison - YACC\Infix to postfix>a.exe  
a+b*2/69+i-u+v*69  
a b 2 * 69 / + i + u - v 69 * +  
  
E:\Btech -docs\Sem-6\CD Lab\Bison - YACC\Infix to postfix>a.exe  
1+2*a-b+45  
1 2 a * + b - 45 +
```