A

Lab Manual of

Compiler Design Laboratory

Is Submitted to

NAVRACHANA
UNIVERSITY
a UGC recognized University

School of Engineering and Technology

Toward the fulfilment of the requirements of the Subject

Compiler Design Laboratory – (CSE606)

SUBMITTED BY

Raj Mistry - 22000995

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

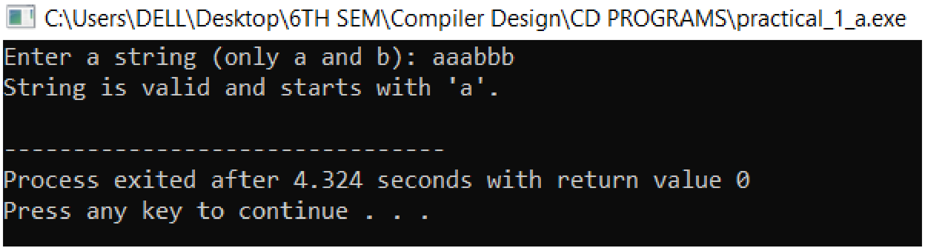
PRACTICAL – 1

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

PROGRAM CODE:

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

int isValidString(const char *str) {
   int i;
   if (str[0] != 'a') {
      return 0;
   }

   for (i = 1; str[i] != '\0'; i++) {
      if (str[i] != 'a' && str[i] != 'b') {
         return 0;
      }
   }

   return 1;
}

int main() {
   char input[100];

   printf("Enter a string (only a and b): ");
   scanf("%s", input);

   if (isValidString(input)) {
      printf("String is valid and starts with 'a'.\n");
   } else {
        printf("Invalid string. It must start with 'a' and contain only 'a' or 'b'.\n");
    }

    return 0;
}
```

OUTPUT:



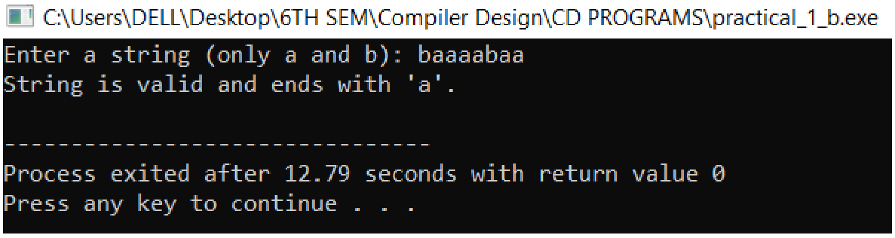
```
C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS\practical_1_a.exe
Enter a string (only a and b): aaabbb
String is valid and starts with 'a'.

--------------------------------
Process exited after 4.324 seconds with return value 0
Press any key to continue . . .
```

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

PROGRAM CODE:

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

int isValidString(const char *str) {
    int i, len = strlen(str);
    if (len == 0) {
        return 0;
    }

    for (i = 0; i < len; i++) {
        if (str[i] != 'a' && str[i] != 'b') {
            return 0;
        }
    }

    if (str[len - 1] != 'a') {
        return 0;
    }

    return 1;
}

int main() {
    char input[100];

    printf("Enter a string (only a and b): ");
    scanf("%s", input);

    if (isValidString(input)) {
        printf("String is valid and ends with 'a'.\n");
    } else {
        printf("Invalid string. It must end with 'a' and contain only 'a' or 'b'.\n");
    }
    return 0;
}
```

OUTPUT:



C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS\practical_1_b.exe

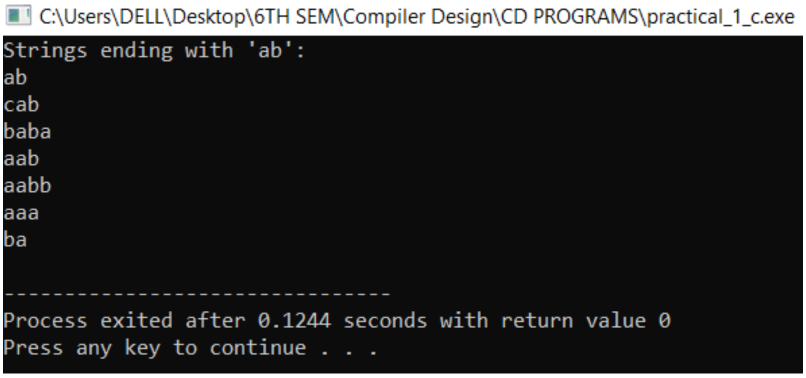
```
Enter a string (only a and b): baaaabaa
String is valid and ends with 'a'.

--------------------------------
Process exited after 12.79 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.

PROGRAM CODE:

```
#include <stdio.h>
#include <string.h>
int endsWithAB(const char *str) {
    int len = strlen(str);
    return (len >= 2 && str[len - 2] == 'a' && str[len - 1] == 'b');
}
int main() {
    FILE *file;
    char line[100];
    file = fopen("input.txt", "r");
    if (file == NULL) {
        printf("Error: Could not open input.txt\n");
        return 1;
    }

    printf("Strings ending with 'ab':\n");
    while (fgets(line, sizeof(line), file)) {
        line[strcspn(line, "\n")] = '\0';

        if (endsWithAB(line)) {
            printf("%s\n", line);
        } else {
            printf("%s\n", line);
        }
    }
    fclose(file);
    return 0;
}
```

OUTPUT:



C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS\practical_1_c.exe

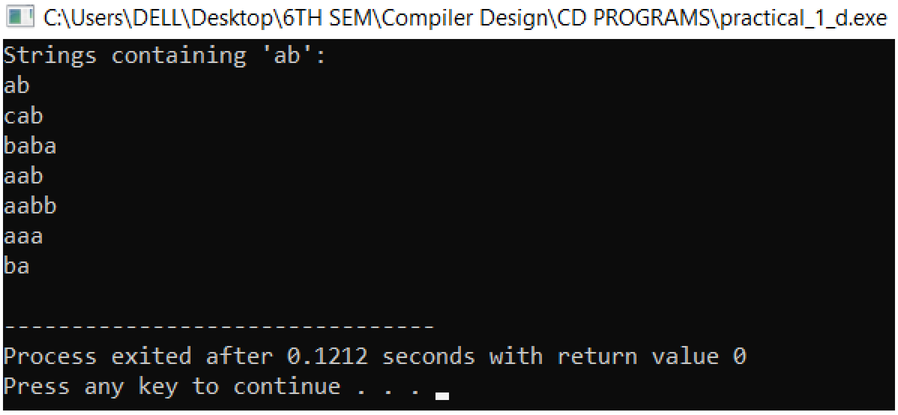
```
Strings ending with 'ab':
ab
cab
baba
aab
aabb
aaa
ba

--------------------------------
Process exited after 0.1244 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.

PROGRAM CODE:

```
#include <stdio.h>
#include <string.h>
int containsAB(const char *str) {
   return (strstr(str, "ab") != NULL);
}
int main() {
   FILE *file;
   char line[100];

   file = fopen("input.txt", "r");
   if (file == NULL) {
      printf("Error: Could not open input.txt\n");
      return 1;
   }
   printf("Strings containing 'ab':\n");
   while (fgets(line, sizeof(line), file)) {
      line[strcspn(line, "\n")] = '\0';
      if (containsAB(line)) {
         printf("%s\n", line);
      } else {
            printf("%s\n", line);
        }
    }
    fclose(file);
    return 0;
}
```

OUTPUT:



C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS\practical_1_d.exe

```
Strings containing 'ab':
ab
cab
baba
aab
aabb
aaa
ba

--------------------------------
Process exited after 0.1212 seconds with return value 0
Press any key to continue . . .
```

PRACTICAL – 2

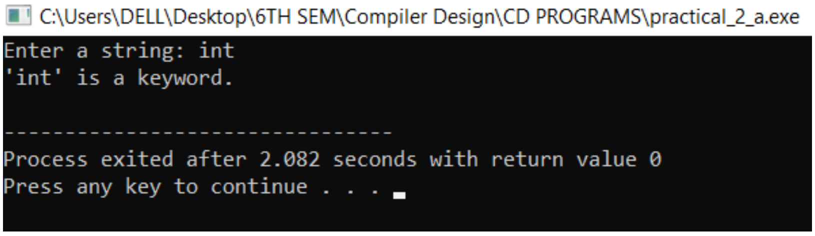
AIM: a) Write a program to recognize the valid identifiers and keywords.

PROGRAM CODE:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
int isKeyword(char *word) {
   const char *keywords[] = {
      "int", "float", "if", "else", "while", "for", "do", "char", "return", "void", "switch", "case"
   };
   int n = sizeof(keywords) / sizeof(keywords[0]);
   for (int i = 0; i < n; i++) {
      if (strcmp(word, keywords[i]) == 0)
         return 1;
   }
   return 0;
}
int isValidIdentifier(char *word) {
   if (!isalpha(word[0]) && word[0] != '_') return 0;
   for (int i = 1; word[i]; i++) {
      if (!isalnum(word[i]) && word[i] != '_') return 0;
   }
   return 1;
}
int main() {
   char word[100];
   printf("Enter a string: ");
   scanf("%s", word);
   if (isKeyword(word))
      printf("'%s' is a keyword.\n", word);
```

```
    else if (isValidIdentifier(word))
        printf("'%s' is a valid identifier.\n", word);
    else
        printf("'%s' is not a valid identifier.\n", word);
    return 0;
}
```

OUTPUT:


C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS\practical_2_a.exe

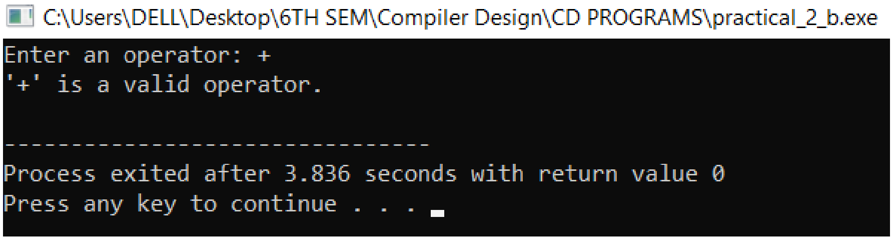
```
Enter a string: int
'int' is a keyword.

--------------------------------
Process exited after 2.082 seconds with return value 0
Press any key to continue . . .
```

b) Write a program to recognize the valid operators.

PROGRAM CODE:

```
#include <stdio.h>
#include <string.h>
int isOperator(char *op) {
    char *operators[] = {"+", "-", "*", "/", "=", "==", "!=", "<", "<=", ">", ">=", "&&", "||"};
    int n = sizeof(operators) / sizeof(operators[0]);
    int i;
    for (i = 0; i < n; i++) {
        if (strcmp(op, operators[i]) == 0)
            return 1;
    }
    return 0;
}
int main() {
    char op[5];
    printf("Enter an operator: ");
    scanf("%s", op);
    if (isOperator(op))
        printf("'%s' is a valid operator.\n", op);
    else
        printf("'%s' is not a valid operator.\n", op);
    return 0;
}
```

OUTPUT:



C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS\practical_2_b.exe

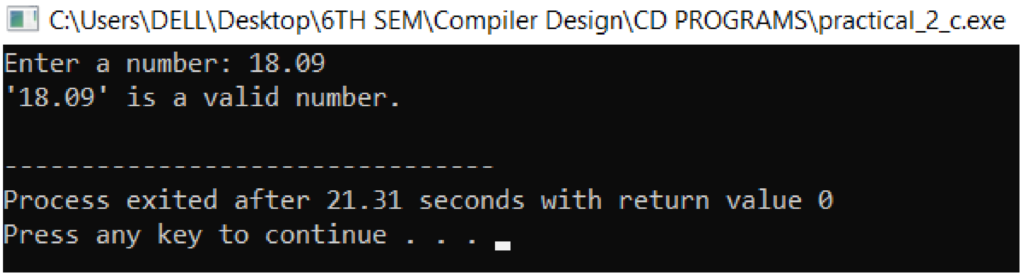
```
Enter an operator: +
'+' is a valid operator.

--------------------------------
Process exited after 3.836 seconds with return value 0
Press any key to continue . . .
```

c) Write a program to recognize the valid number.

PROGRAM CODE:

```
#include <stdio.h>
#include <ctype.h>
int isNumber(char *str) {
    int i = 0, dotCount = 0;
    if (str[i] == '-' || str[i] == '+')
        i++;
    for (; str[i]; i++) {
        if (str[i] == '.') {
            dotCount++;
            if (dotCount > 1) return 0;
        } else if (!isdigit(str[i])) {
            return 0;
        }
    }
    return (i > 0);
}
int main() {
    char num[50];
    printf("Enter a number: ");
    scanf("%s", num);
    if (isNumber(num))
        printf("'%s' is a valid number.\n", num);
    else
        printf("'%s' is not a valid number.\n", num);
    return 0;
}
```

OUTPUT:



C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS\practical_2_c.exe

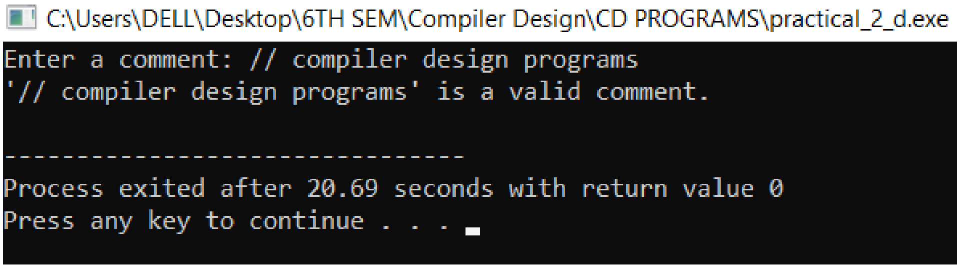
```
Enter a number: 18.09
'18.09' is a valid number.

--------------------------------
Process exited after 21.31 seconds with return value 0
Press any key to continue . . .
```

d) Write a program to recognize the valid comments.

PROGRAM CODE:

```
#include <stdio.h>
#include <string.h>
int isComment(const char *str) {
    int len = strlen(str);
    if (strncmp(str, "//", 2) == 0)
        return 1;
    if (strncmp(str, "/*", 2) == 0 && len >= 4 && str[len - 2] == '*' && str[len - 1] == '/')
        return 1;
    return 0;
}
int main() {
    char comment[200];
    printf("Enter a comment: ");
    fgets(comment, sizeof(comment), stdin);
    size_t len = strlen(comment);
    if (len > 0 && comment[len - 1] == '\n') {
        comment[len - 1] = '\0';
    }
    if (isComment(comment))
        printf("'%s' is a valid comment.\n", comment);
    else
        printf("'%s' is not a valid comment.\n", comment);
    return 0;
}
```

OUTPUT:



C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS\practical_2_d.exe

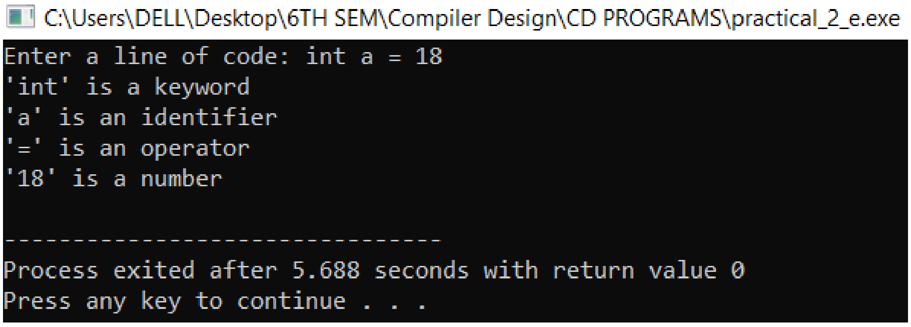
```
Enter a comment: // compiler design programs
'// compiler design programs' is a valid comment.

--------------------------------
Process exited after 20.69 seconds with return value 0
Press any key to continue . . .
```

e) Program to implement Lexical Analyzer.

PROGRAM CODE:

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
char keywords[10][10] = {
    "int", "float", "char", "if", "else",
    "while", "for", "return", "void", "double"
};
int isKeyword(char *word) {
    for (int i = 0; i < 10; i++) {
        if (strcmp(word, keywords[i]) == 0)
            return 1;
    }
    return 0;
}
int isOperator(char ch) {
    return ch == '+' || ch == '-' || ch == '*' || ch == '/' || ch == '=' || ch == '<' || ch == '>';
}
int main() {
    char line[256], token[50];
    int i = 0, j = 0;
    printf("Enter a line of code: ");
    fgets(line, sizeof(line), stdin);
    while (line[i] != '\0') {
        if (isspace(line[i])) {
            i++;
        } else if (isalpha(line[i]) || line[i] == '_') {
            j = 0;
            while (isalnum(line[i]) || line[i] == '_') {
                token[j++] = line[i++];
            }
            token[j] = '\0';
            if (isKeyword(token))
                printf("'%s' is a keyword\n", token);
            else
                printf("'%s' is an identifier\n", token);
        } else if (isdigit(line[i])) {
            j = 0;
            while (isdigit(line[i]) || line[i] == '.') {
                token[j++] = line[i++];
            }
            token[j] = '\0';
            printf("'%s' is a number\n", token);
        } else if (line[i] == '/' && line[i + 1] == '/') {
            printf("'//%s' is a comment\n", &line[i + 2]);
            break;
        } else if (isOperator(line[i])) {
            printf("'%c' is an operator\n", line[i]);
            i++;
        } else {
            printf("'%c' is an unknown character\n", line[i]);
            i++;
        }
    }
    return 0;
}
```

OUTPUT:



C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS\practical_2_e.exe

```
Enter a line of code: int a = 18
'int' is a keyword
'a' is an identifier
'=' is an operator
'18' is a number

--------------------------------
Process exited after 5.688 seconds with return value 0
Press any key to continue . . .
```

PRACTICAL – 3

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

1. Lexical Analyzer Generator (LEX)

- **Overview:** Lex is a tool that generates a lexical analyzer (also called a scanner or tokenizer) from a set of regular expressions and associated actions.
- **Functionality:** It reads an input stream, applies the regular expressions to the input, and produces tokens that represent the different parts of the input.
- **Use Case:** Commonly used in the front-end of compilers to split source code into meaningful components (such as keywords, identifiers, operators, etc.).

Basic Structure:

- Definitions: Includes headers, definitions, and initialization code.
- Rules: Specifies patterns (regular expressions) for different tokens.
- Actions: Defines the code that will execute when a pattern is matched.

Example

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

- **Compilation:** After writing the Lex file (.l extension), it is processed by Lex to generate C code. The resulting C code is compiled to produce the lexical analyzer.
- **Lexical Analyzer Execution:** The lexer reads the input and matches the patterns defined in the rules. It then executes the corresponding actions.

2. Flex (Fast Lexical Analyzer Generator)

- **Overview:** Flex is a more efficient and feature-rich version of Lex. It is compatible with Lex but provides additional features, optimizations, and better performance.
- **Key Improvements:**
  1) It supports regular expressions, which allow you to define patterns more effectively.
  2) It can generate optimized C code for faster execution.
- **Structure of Flex Files:** Flex files are like Lex files but with improvements:
  1) Definitions Section: Include definitions for constants and macros.
  2) Rules Section: Contains patterns and corresponding actions.
  3) User Code Section: Contains custom code, initialization, and main function.

Example

```
%%
[0-9]+       { printf("Integer: %s\n", yytext); }
[a-zA-Z]+    { printf("Keyword: %s\n", yytext); }
%%
int main() {
   yylex();
   return 0;
}
```

- **Compilation:** Flex reads the .l file, generates a C file (usually lex.yy.c), which can then be compiled to produce the lexer.

PRACTICAL – 4

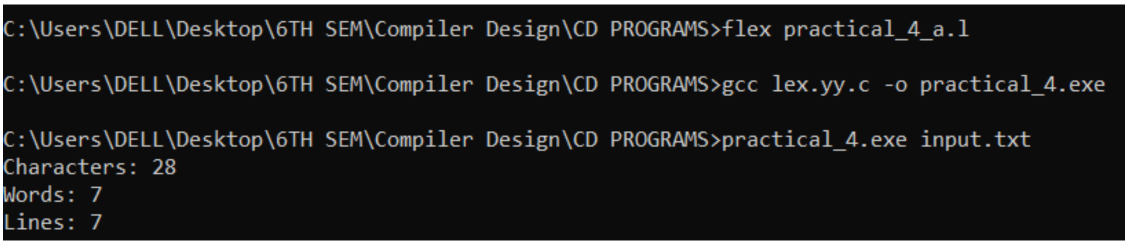
AIM: Implement following programs using Lex:

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

PROGRAM CODE:

```
%{
#include <stdio.h>
int char_count = 0;
int line_count = 0;
int word_count = 0;
%}

%%
\n          { line_count++; }
[ \t\n]+    { /* Ignore spaces, tabs, and newlines */ }
[A-Za-z0-9]+ { word_count++; }
.           { char_count++; }
%%
int main(int argc, char **argv) {
    if (argc < 2) {
        printf("Usage: %s <input_file>\n", argv[0]);
        return 1;
    }
    FILE *file = fopen(argv[1], "r");
    if (!file) {
        perror("Error opening file");
        return 1;
    }
    yyin = file;
    yylex();
    fclose(file);
    printf("Characters: %d\n", char_count);
    printf("Words: %d\n", word_count);
    printf("Lines: %d\n", line_count);
    return 0;
}
```

OUTPUT:



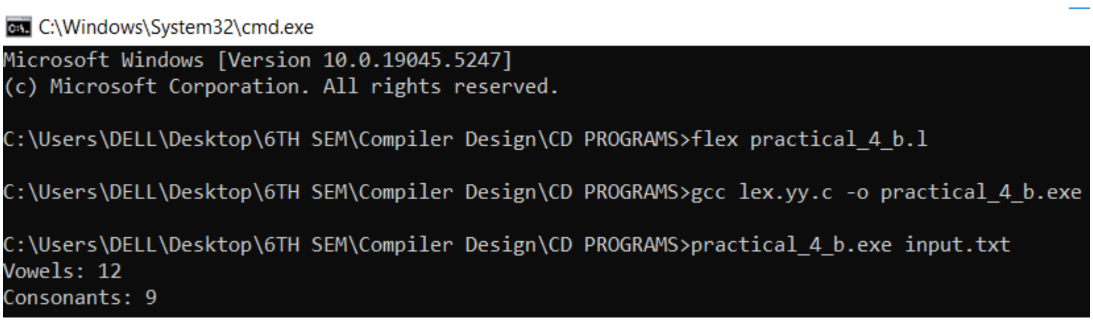
```
C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>flex practical_4_a.l

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>gcc lex.yy.c -o practical_4.exe

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>practical_4.exe input.txt
Characters: 28
Words: 7
Lines: 7
```

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

PROGRAM CODE:

```
%{
#include <stdio.h>
int vowels = 0, consonants = 0;
%}


%%
[aAeEiIoOuU]    { vowels++; }
[b-df-hj-np-tv-zB-DF-HJ-NP-TV-Z] { consonants++; }
.|\n            ;
%%
int yywrap()
{
return 1;
}
int main() {
    FILE *file = fopen("input.txt", "r");
    if (!file) { perror("input.txt"); return 1; }
    yyin = file;
    yylex();
    fclose(file);
    printf("Vowels: %d\nConsonants: %d\n", vowels, consonants);
    return 0;
}
```

OUTPUT:



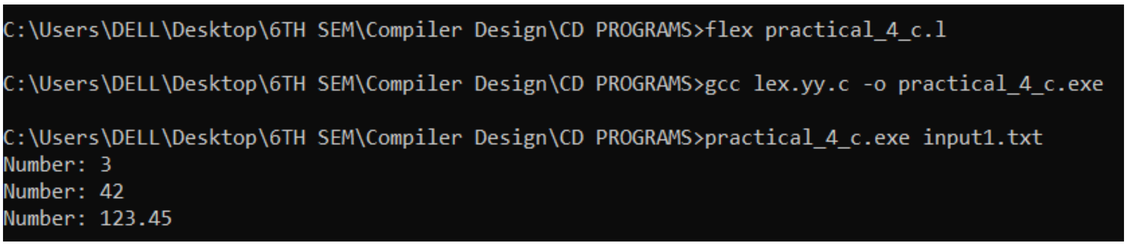
```
Microsoft Windows [Version 10.0.19045.5247]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>flex practical_4_b.l

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>gcc lex.yy.c -o practical_4_b.exe

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>practical_4_b.exe input.txt
Vowels: 12
Consonants: 9
```

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

PROGRAM CODE:

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


%%
[0-9]+(\.[0-9]+)?   { printf("Number: %s\n", yytext); }
.|\n                ;
%%
int yywrap()
{
return 1;
}
int main() {
    FILE *file = fopen("input1.txt", "r");
    if (!file) { perror("input1.txt"); return 1; }
    yyin = file;
    yylex();
    fclose(file);
    return 0;
}
```

OUTPUT:



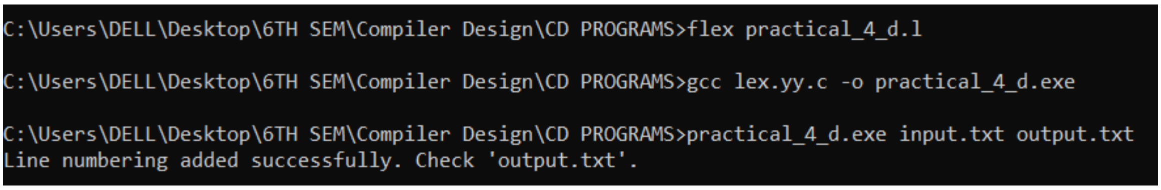
```
C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>flex practical_4_c.l

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>gcc lex.yy.c -o practical_4_c.exe

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>practical_4_c.exe input1.txt
Number: 3
Number: 42
Number: 123.45
```

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

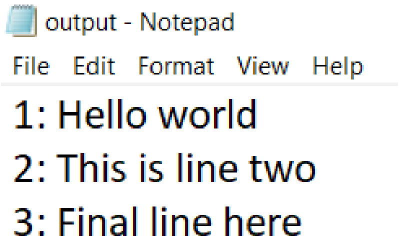
PROGRAM CODE:

```
%{
#include <stdio.h>
int lineno = 1;
FILE *outfile;
%}


%%
^.*\n     { fprintf(outfile, "%d: %s", lineno++, yytext); }
^[^\n]+   { fprintf(outfile, "%d: %s\n", lineno++, yytext); }
%%
int yywrap()
{
return 1;
}
int main() {
    FILE *file = fopen("input.txt", "r");
    outfile = fopen("output.txt", "w");
    if (!file || !outfile) {
        perror("File error");
        return 1;
    }
    yyin = file;
    yylex();
    fclose(file);
    fclose(outfile);
    printf("Line numbering added successfully. Check 'output.txt'.\n");
    return 0;
}
```

OUTPUT:



```
C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>flex practical_4_d.l

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>gcc lex.yy.c -o practical_4_d.exe

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>practical_4_d.exe input.txt output.txt
Line numbering added successfully. Check 'output.txt'.
```



```
1: Hello world
2: This is line two
3: Final line here
```

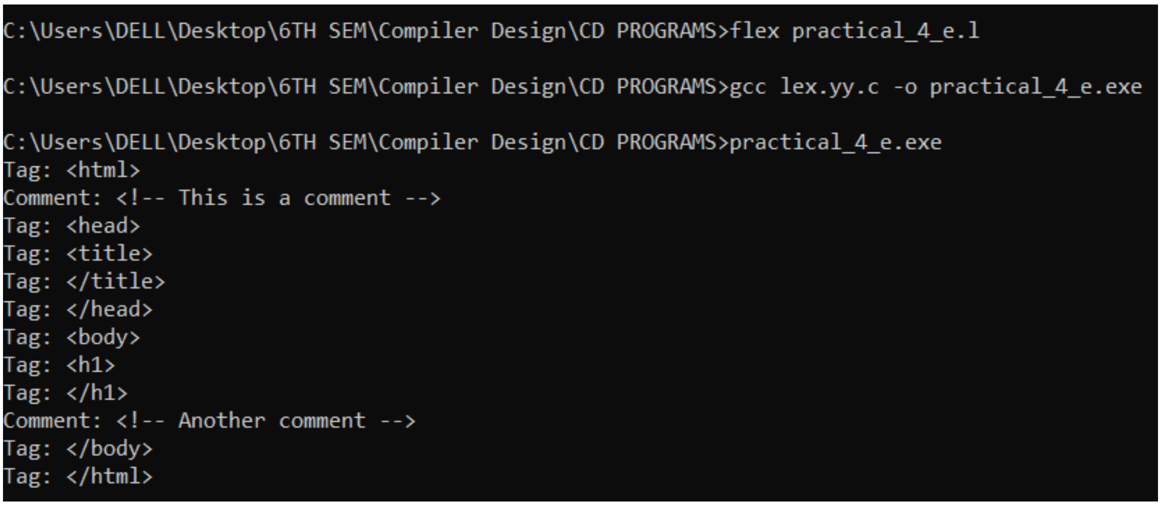
e) Write a Lex program to printout all markup tags and HTML comments in file.

PROGRAM CODE:

```
%{
#include <stdio.h>
FILE *yyin;
%}

%%
"<!--"([^>-]|"-"[^>])*"-->"     { printf("Comment: %s\n", yytext); }
"<"[^>]+">"                     { printf("Tag: %s\n", yytext); }
```

```
.|\n                    { /* Skip all other characters */ }


%%
int yywrap() {
   return 1;
}
int main() {
   yyin = fopen("input.txt", "r");
   if (!yyin) {
      perror("File not found");
      return 1;
   }
   yylex();
   fclose(yyin);
   return 0;
}
```

OUTPUT:



```
C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>flex practical_4_e.l

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>gcc lex.yy.c -o practical_4_e.exe

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>practical_4_e.exe
Tag: <html>
Comment: <!-- This is a comment -->
Tag: <head>
Tag: <title>
Tag: </title>
Tag: </head>
Tag: <body>
Tag: <h1>
Tag: </h1>
Comment: <!-- Another comment -->
Tag: </body>
Tag: </html>
```

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.

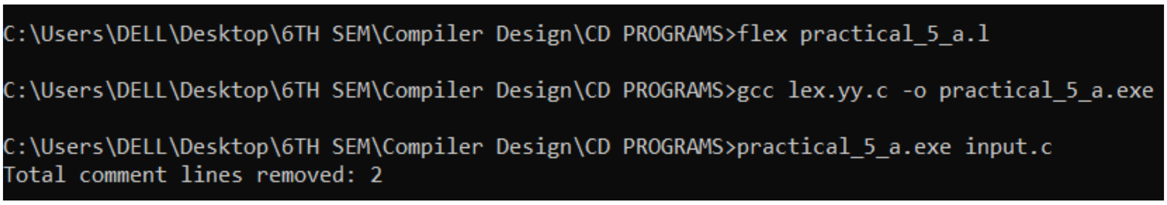
PROGRAM CODE:

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

%%
"/*"([^*]|\*+[^*/])*"*"+"/"    { comment_count++; /* block comment */ }
"//".*                         { comment_count++; /* line comment */ }
.|\n                          { fputc(yytext[0], out); }
%%
int yywrap() { return 1; }
int main() {
   FILE *in = fopen("input.c", "r");
   out = fopen("cleaned_output.c", "w");
   if (!in || !out) {
      printf("File error!\n");
      return 1;
   }
   yyin = in;
   yylex();
   fclose(in);
   fclose(out);
   printf("Total comment lines removed: %d\n", comment_count);
   return 0;
}
```

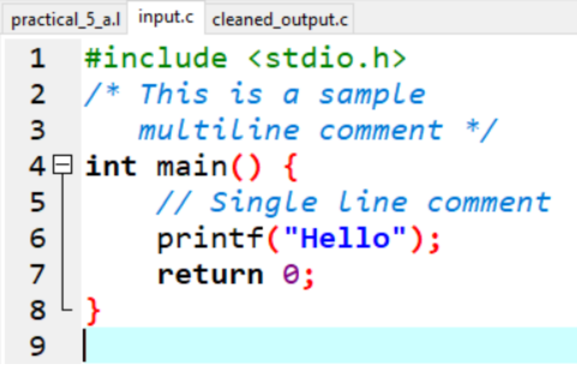
input.c

```
#include <stdio.h>
/* This is a sample
  multiline comment */
int main() {
   // Single line comment
   printf("Hello");
   return 0;
}
```

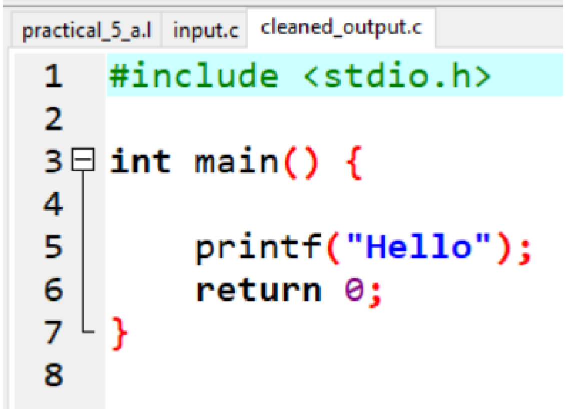
OUTPUT:



```
C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>flex practical_5_a.l

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>gcc lex.yy.c -o practical_5_a.exe

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>practical_5_a.exe input.c
Total comment lines removed: 2
```



```
#include <stdio.h>
/* This is a sample
   multiline comment */
int main() {
    // Single line comment
    printf("Hello");
    return 0;
}

```



```
#include <stdio.h>

int main() {

    printf("Hello");
    return 0;
}

```

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

  "=="|"!="|"<="|">="|"="|"+"|"-"|"*"|"/"|"%"|"&&"|"||"|"!"|"<"|">" {
    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);
}

%%  // DO NOT MISS THIS!

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.txt

```
#include <stdio.h>
int main() {
    int a = 10;
    float b = 20.5;
    char c = 'A';
    if (a < b) {
        printf("a is less than b\n");
    }
    return 0;
}
```

PRACTICAL – 6

AIM: Program to implement Recursive Descent Parsing in C.

PROGRAM CODE:

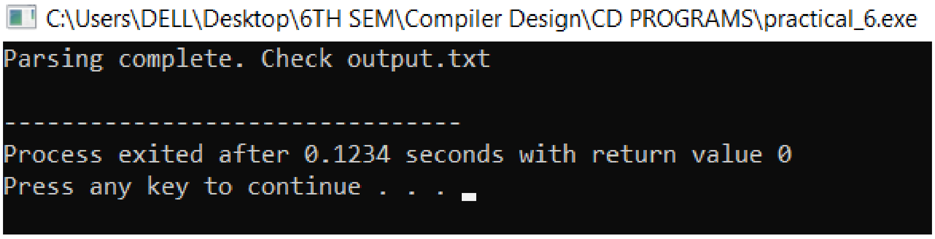
practical_6.c

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

char input[100];
int pos = 0;
FILE *output;

void error() {
    fprintf(output, "Error in parsing\n");
    exit(1);
}

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

void E();
void E_();
void T();
void T_();
void F();
void E() {
    T();
    E_();
}

void E_() {
    if (input[pos] == '+') {
        match('+');
        T();
        E_();
    }
}

void T() {
    F();
    T_();
}

void T_() {
    if (input[pos] == '*') {
        match('*');
        F();
        T_();
    }
}

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

int main() {
    FILE *fp = fopen("input.txt", "r");
    output = fopen("output.txt", "w");

    if (fp == NULL || output == NULL) {
        printf("Error opening file.\n");
        return 1;
    }

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

    E();

    if (input[pos] == '\0') {
        fprintf(output, "String is accepted.\n");
    } else {
        fprintf(output, "String is rejected.\n");
    }

    fclose(output);
    printf("Parsing complete. Check output.txt\n");
    return 0;
}
```

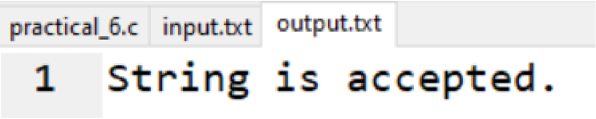
input.txt

```
(a+b)*c
a+*(b)
```

OUTPUT:


C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS\practical_6.exe

```
Parsing complete. Check output.txt

--------------------------------
Process exited after 0.1234 seconds with return value 0
Press any key to continue . . .
```



```
String is accepted.
```

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)

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

PROGRAM CODE:

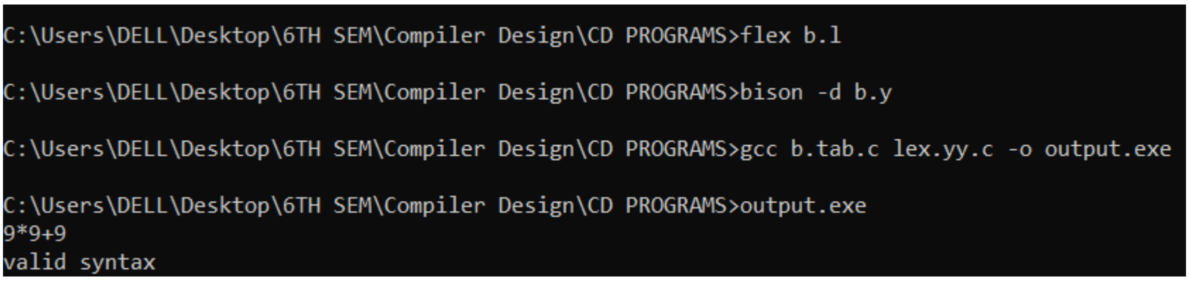
b.l

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "b.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;
}
```

b.y

```
%{
 #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:



```
C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>flex b.l

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>bison -d b.y

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>gcc b.tab.c lex.yy.c -o output.exe

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\CD PROGRAMS>output.exe
9*9+9
valid syntax
```

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

PROGRAM CODE:

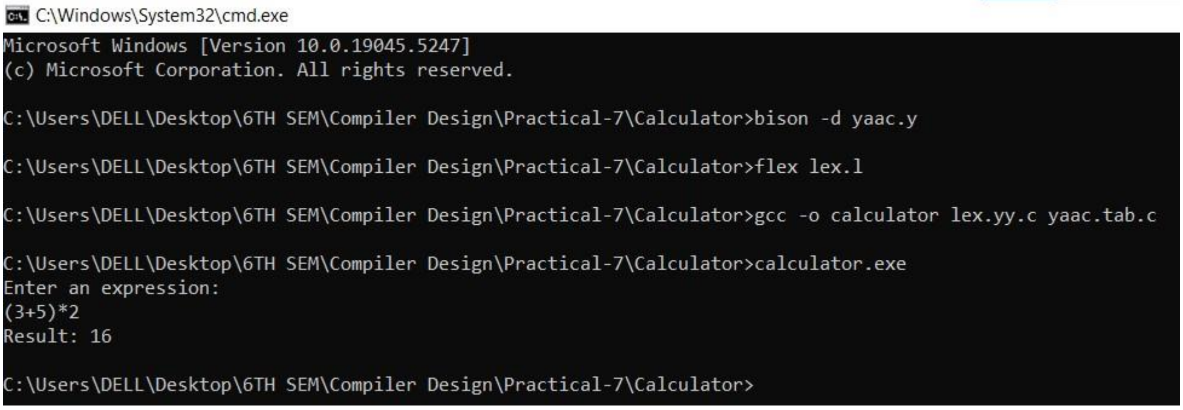
lex.l

```
%{
#include <stdlib.h>
#include "yaac.tab.h"
void yyerror(char *);
%}
%%
[0-9]+     { yylval = atoi(yytext); return NUM; }
[+\-*/\n]  { return *yytext; }
"("        { return '('; }
")"        { return ')'; }
[ \t]      { /* ignore whitespace */ }
.          { yyerror("Invalid character"); }
%%
int yywrap() {
    return 0;
}
```

yaac.y

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(char *);
%}
%token NUM
%%
S: E '\n'           { printf("Result: %d\n", $1); return 0; }

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

T: T '*' F           { $$ = $1 * $3; }
 | T '/' F          {
                    if ($3 == 0) {
                        yyerror("Error: Division by zero");
                        YYABORT;
                    }
                    $$ = $1 / $3;
                   }
 | F                { $$ = $1; }

F: '(' E ')'        { $$ = $2; }
 | NUM               { $$ = $1; }
%%
void yyerror(char *s) {
    fprintf(stderr, "Syntax Error: %s\n", s);
}
int main() {
    printf("Enter an expression:\n");
    yyparse();
    return 0;
}
```

OUTPUT:



```
Microsoft Windows [Version 10.0.19045.5247]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\Practical-7\Calculator>bison -d yaac.y

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\Practical-7\Calculator>flex lex.l

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\Practical-7\Calculator>gcc -o calculator lex.yy.c yaac.tab.c

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\Practical-7\Calculator>calculator.exe
Enter an expression:
(3+5)*2
Result: 16

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\Practical-7\Calculator>
```

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

PROGRAM CODE:

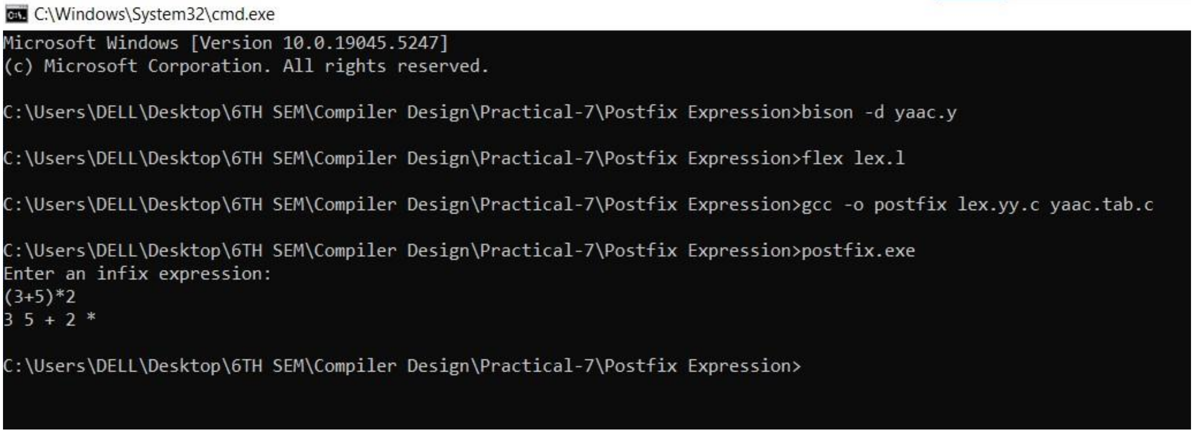
lex.l

```
%{
#include <stdlib.h>
#include "yaac.tab.h"
void yyerror(char *);
%}


%%
[0-9]+     { yylval = atoi(yytext); return NUM; }
[+\-*/\n]  { return *yytext; }
"("        { return '('; }
")"        { return ')'; }
[ \t]      { /* ignore whitespace */ }
.          { yyerror("Invalid character"); }
%%
int yywrap() {
   return 0;
}
```

yaac.y

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

%token NUM
%%
S: E '\n'      { printf("\n"); return 0; }
E: E '+' T      { printf("+ "); }
 | E '-' T      { printf("- "); }
 | T
T: T '*' F      { printf("* "); }
 | T '/' F      { printf("/ "); }
 | F
F: '(' E ')'
 | NUM          { printf("%d ", $1); }
%%
void yyerror(char *s) {
   fprintf(stderr, "Syntax Error: %s\n", s);
}
int main() {
   printf("Enter an infix expression:\n");
   yyparse();
   return 0;
}
```

OUTPUT:



```
Microsoft Windows [Version 10.0.19045.5247]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\Practical-7\Postfix Expression>bison -d yaac.y

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\Practical-7\Postfix Expression>flex lex.l

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\Practical-7\Postfix Expression>gcc -o postfix lex.yy.c yaac.tab.c

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\Practical-7\Postfix Expression>postfix.exe
Enter an infix expression:
(3+5)*2
3 5 + 2 *

C:\Users\DELL\Desktop\6TH SEM\Compiler Design\Practical-7\Postfix Expression>
```