SCHOOL OF ENGINEERING & TECHNOLOGY

BACHELOR OF TECHNOLOGY

COMPILER DESIGN

6TH SEMESTER

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Laboratory Manual

TABLE OF CONTENT

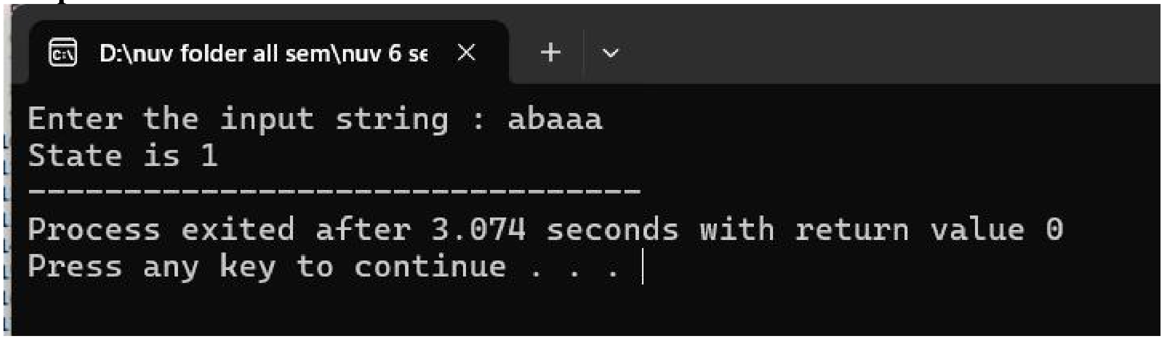
Practical-1

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

Input:

```
#include<stdio.h>
int main ()
{
    char input[10];
    int state=0,i=0;

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

    while(input[i]!='\0'){
            switch (state){
                    case 0:
                            if (input[i]=='a'){
                                    state = 1;
                                    i++;
                            }
                            else if (input[i]=='b'){
                                    state = 2;
                                    i++;
                            }
                            else {
                                    state = 3;
                                    i++;
                            }
                            break;
                    case 1:
                            if (input[i]=='a'||input[i]=='b'){
                                    state = 1;
                                    i++;
                            }
                            else {
                                    state = 3;
                                    i++;
                            }
                            break;
                    case 2:
                            if (input[i]=='a'||input[i]=='b'){
                                    state = 2;
                                    i++;
                            }
                            else {
                                    state = 3;
                                    i++;
                            }
                            break;
                    case 3:
                            state=3;
                            i++;
                            break;
            }
    }
    printf("State is %d",state);
    return 0;
}
```

Output:

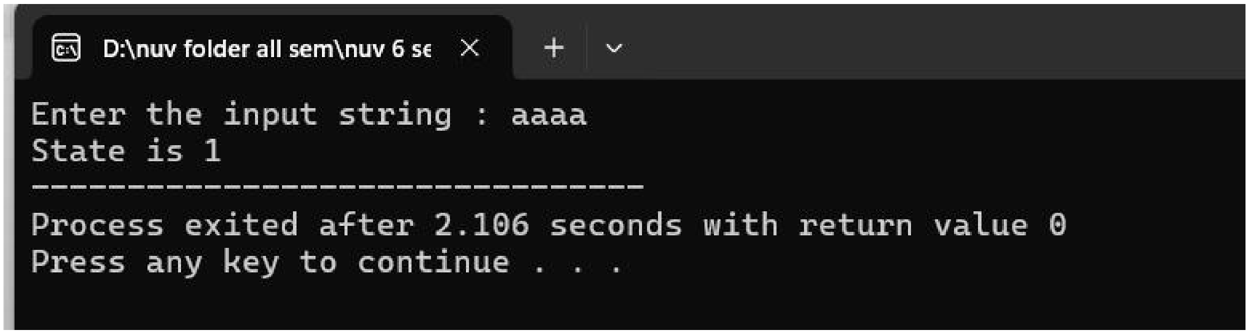


```
Enter the input string : abaaa
State is 1
--------------------------------
Process exited after 3.074 seconds with return value 0
Press any key to continue . . .
```

Input (With File) (extra practical):

```
#include<stdio.h>
 #include <stdlib.h>
int main ()
{
    char input[100];
    int state=0,i=0;
    //write a program for string starts with 'a'.
    FILE* ptr;

    ptr = fopen("Hello.txt", "r");

    if (ptr == NULL)
    {
    printf("Error While opening file");
    exit(1);
    }
    if(fgets(input, 80, ptr) != NULL)
    {
    puts(input);
    }
    fclose(ptr);
    printf("\nInput is  : %s ",input);

//  printf("Enter : ");
//  scanf("%s",input);

    while(input[i]!='\0'){
            switch (state){
                    case 0:
                            if (input[i]=='a'){
                                    state = 1;
                                    i++;
                            }
                            else {
                                    state = 2;
                                    i++;
                            }
                            break;
                    case 1:
                            state = 1;
                            i++;
                            break;
                    case 2:
                            state = 2;
                            i++;
                            break;
            }
    }
    if(state == 1){
            printf("\nThe String is valid.",input);
    }
    else if(state == 2){
            printf("\nThe String is invalid.",input);
    }
    printf("\nState is %d",state);
    return 0;
}
```

Output:



```
Enter the input string : aaaa
State is 1
--------------------------------
Process exited after 2.106 seconds with return value 0
Press any key to continue . . .
```

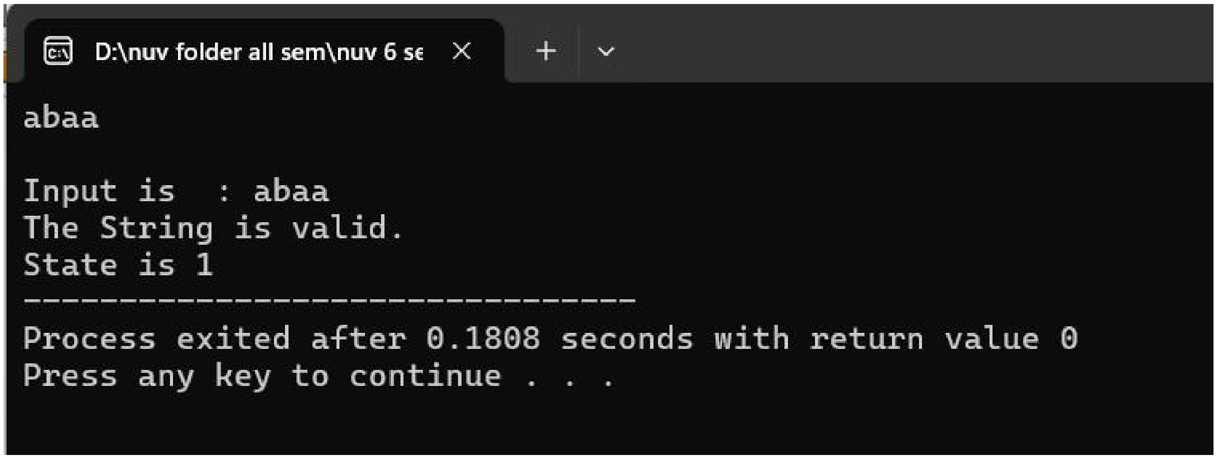
b) Write a program to recognize strings end with ‘a’.

Input:

```
#include<stdio.h>
 #include <stdlib.h>
int main ()
{
                    char input[100];
                    int state=0,i=0;
                    //write a program for string ends with 'a'.
                    FILE* ptr;

                    ptr = fopen("Hello.txt", "r");

                    if (ptr == NULL)
                    {
                    printf("Error While opening file");
                    exit(1);
                    }

                    if(fgets(input, 80, ptr) != NULL)
                    {
                    puts(input);
                    }
                    fclose(ptr);
                    printf("\nInput is  : %s ",input);

//                  printf("Enter : ");
//                  scanf("%s",input);

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

```
                        break;
                }
                }
                if(state == 1){
                 printf("\nThe String is valid.",input);
                }
                else if(state == 0){
                 printf("\nThe String is invalid.",input);
                }
                printf("\nState is %d",state);
                return 0;
}
```

Output:



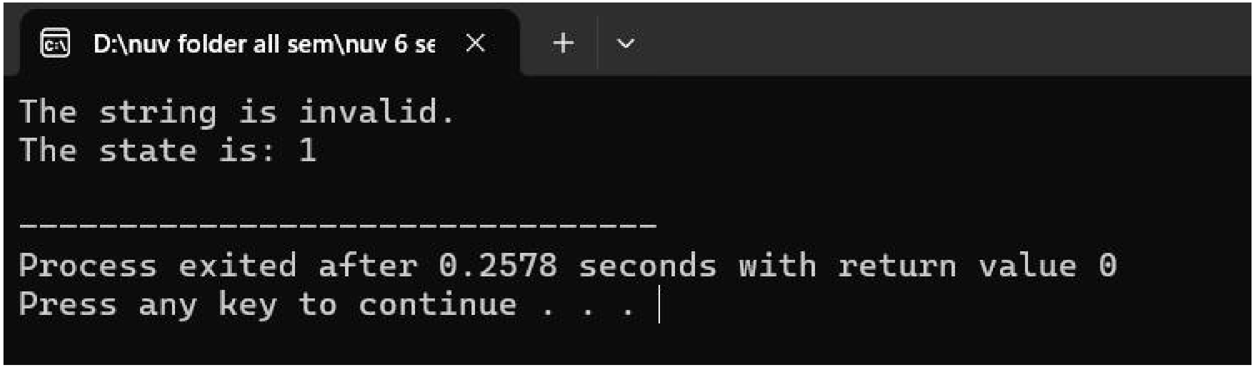
```
abaa

Input is  : abaa
The String is valid.
State is 1
--------------------------------
Process exited after 0.1808 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.

Input:

```
//string ends with ab and take input from a file.
#include <stdio.h>

int main() {
    char input[100];
    int state = 0, i = 0;
    FILE *file; // File pointer

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

    if (fgets(input, sizeof(input), file) == NULL) {
        printf("Error: Could not read from file or file is empty.\n");
        fclose(file);
        return 1;
    }

    fclose(file);

    // Removing newline character, if present
    for (i = 0; input[i] != '\0'; i++) {
        if (input[i] == '\n') {
            input[i] = '\0';
            break;
        }
    }

    i = 0; // Reset index for processing the string

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

    if (state == 0) {
        printf("String is invalid.\n");
        printf("The state is: %d\n", state);
    } else if (state == 1) {
        printf("The string is invalid.\n");
        printf("The state is: %d\n", state);
    } else if (state == 2) {
        printf("The string is valid.\n");
        printf("The state is: %d\n", state);
    }

    return 0;
}
```

Output:



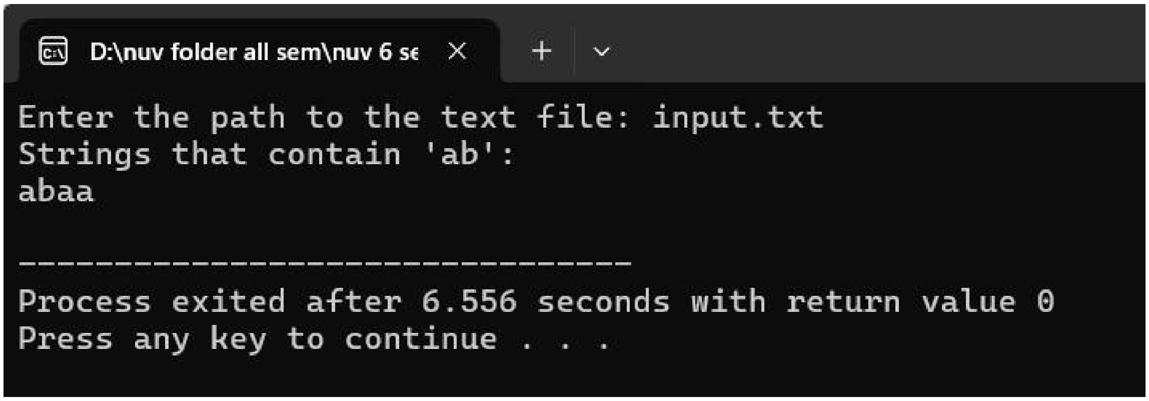
```
The string is invalid.
The state is: 1

--------------------------------
Process exited after 0.2578 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.

Input:

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

#define MAX_LINE_LENGTH 100 // Maximum length for each line

// Function to check strings containing "ab"
void checkStringsContainingAb(const char *filePath) {
    FILE *file = fopen(filePath, "r"); // Open the file in read mode
    if (file == NULL) {
        printf("Error: Could not open file '%s'\n", filePath);
        return; // Exit the function if the file cannot be opened
    }

    char line[MAX_LINE_LENGTH]; // Buffer to hold each line
    printf("Strings that contain 'ab':\n");

    // Read each line from the file
    while (fgets(line, sizeof(line), file) != NULL) {
        // Remove the trailing newline character if present
        size_t len = strlen(line);
        if (len > 0 && line[len - 1] == '\n') {
            line[len - 1] = '\0';
        }

        // Check if the string contains "ab"
        if (strstr(line, "ab") != NULL) {
            printf("%s\n", line);
        }
    }

    fclose(file); // Close the file
}

int main() {
    char filePath[100];

    // Prompt the user for the input file path
    printf("Enter the path to the text file: ");
    scanf("%s", filePath);

    // Call the function to check strings
    checkStringsContainingAb(filePath);

    return 0;
}
```

Output:



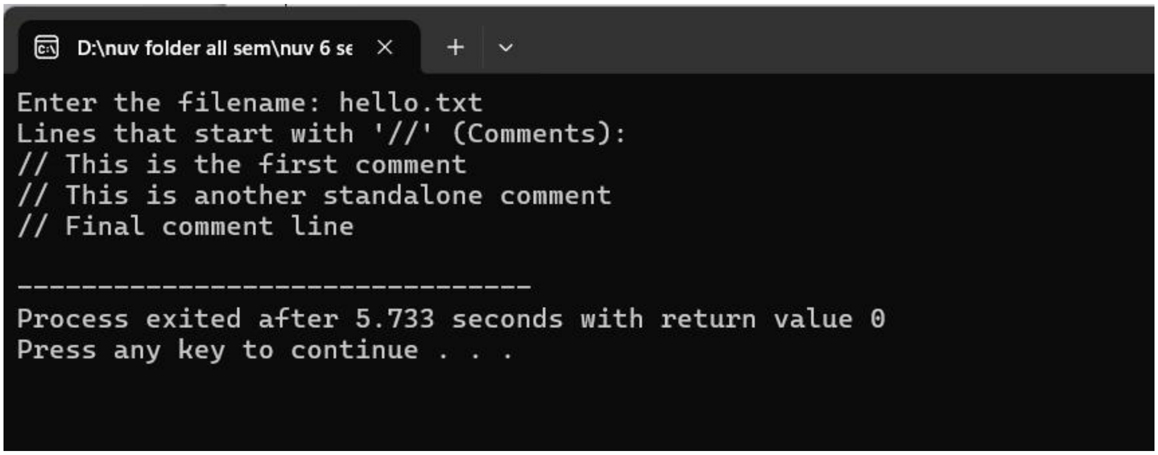
```
Enter the path to the text file: input.txt
Strings that contain 'ab':
abaa

--------------------------------
Process exited after 6.556 seconds with return value 0
Press any key to continue . . .
```

e) Write a program to recognize strings that starts with ‘//’ (Comments). Take the input from text file (extra practical).

Input: (Single line code)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LINE_LENGTH 256

// Function to check if a line starts with "//"
int isComment(const char *line) {
  // Check if the line starts with '//' and is not empty
  if (line[0] == '/' && line[1] == '/') {
    return 1;
  }
  return 0;
}

int main() {
  char filename[100];
  char line[MAX_LINE_LENGTH];
  printf("Enter the filename: ");
  scanf("%s", filename);

  // Open the file in read mode
  FILE *file = fopen(filename, "r");
  if (file == NULL) {
    printf("Error: Could not open file %s\n", filename);
    return 1;
    }
    printf("Lines that start with '//' (Comments):\n");

    // Read the file line by line
    while (fgets(line, sizeof(line), file)) {
        // Remove the newline character from the line if it exists
        line[strcspn(line, "\n")] = 0;

        // Check if the line starts with "//"
        if (isComment(line)) {
            printf("%s\n", line);
        }
    }

    // Close the file
    fclose(file);
    return 0;
}
```

Output:



```
Enter the filename: hello.txt
Lines that start with '//' (Comments):
// This is the first comment
// This is another standalone comment
// Final comment line

--------------------------------
Process exited after 5.733 seconds with return value 0
Press any key to continue . . .
```

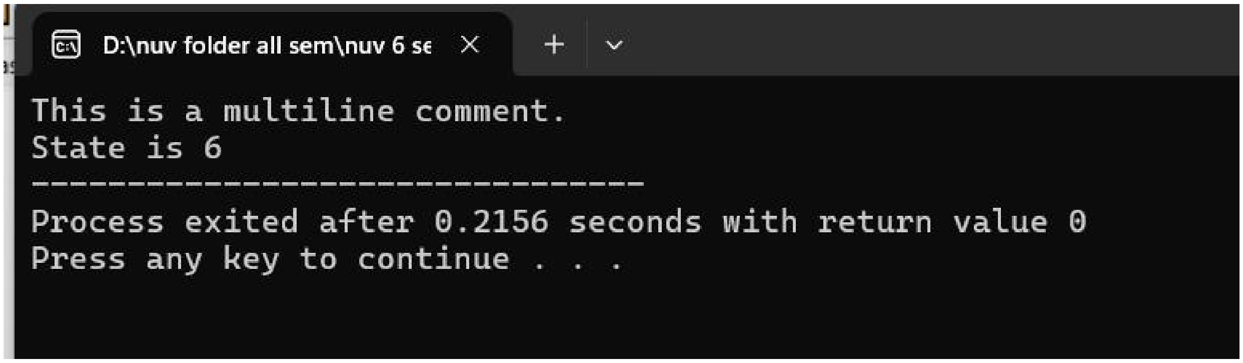
Input: (Multi line code)

```
//accept only comments single line and multiline both.
#include<stdio.h>
int main(){

                char input[100];
                int state =0, i=0;
                FILE *file;

                file = fopen("p1(e)with multiple line.txt","r");
                if(file==NULL){
                 printf("Error: Couldn't open the file.\n");
                 return 1;
                }

                if(fgets(input,sizeof(input),file)==NULL){
                 printf("Error: Couldn't read the file or file is empty.");
                 fclose(file);
                 return 1;
                }
                fclose(file);

                for (i = 0; input[i] != '\0'; i++) {
        if (input[i] == '\n') {
            input[i] = '\0';
            break;
        }
```

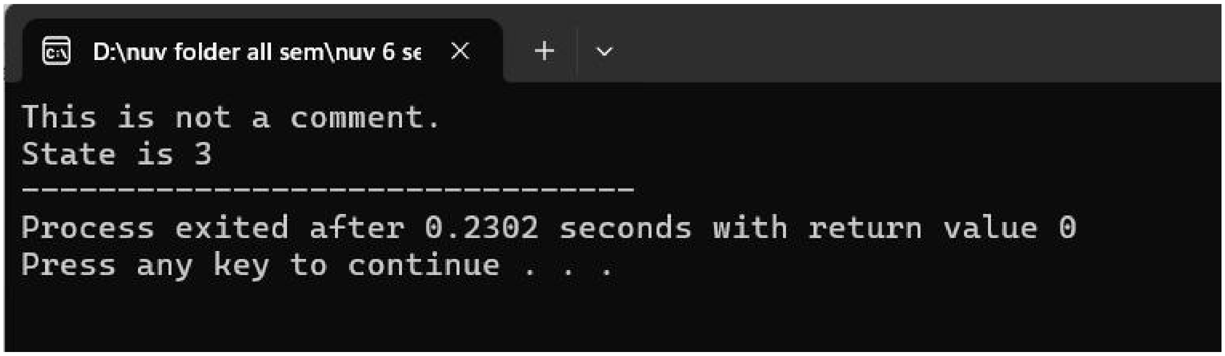
```
}
i = 0;
        while(input[i]!='\0'){
        switch(state){
            case 0:
                if(input[i]=='/')state = 1;
                else state =3;
                break;
            case 1:
                if(input[i]=='/') state=2;
                else if(input[i]=='*') state =4;
                else state=3;
                break;
            case 2:
                state = 2;
                break;
            case 3:
                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:
                state = 3;
                break;
 }
 i++;
}
if(state==0){
 printf("This is not a comment.");
 printf("\nState is %d",state);
 }
else if(state==1){
 printf("This is not a comment.");
 printf("\nState is %d",state);
 }
else if(state==2){
 printf("This is a single line comment.");
 printf("\nState is %d",state);
 }
else if(state==3){
 printf("This is not a comment.");
 printf("\nState is %d",state);
 }
else if(state==4){
 printf("This is not a comment.");
 printf("\nState is %d",state);
 }
else if(state==5){
```

```
            printf("This is not a comment.");
            printf("\nState is %d",state);
            }
            else if(state==6){
            printf("This is a multiline comment.");
            printf("\nState is %d",state);
            }
            return 0;
}
```

Output:



```
This is a multiline comment.
State is 6
--------------------------------
Process exited after 0.2156 seconds with return value 0
Press any key to continue . . .
```



```
This is not a comment.
State is 3
--------------------------------
Process exited after 0.2302 seconds with return value 0
Press any key to continue . . .
```

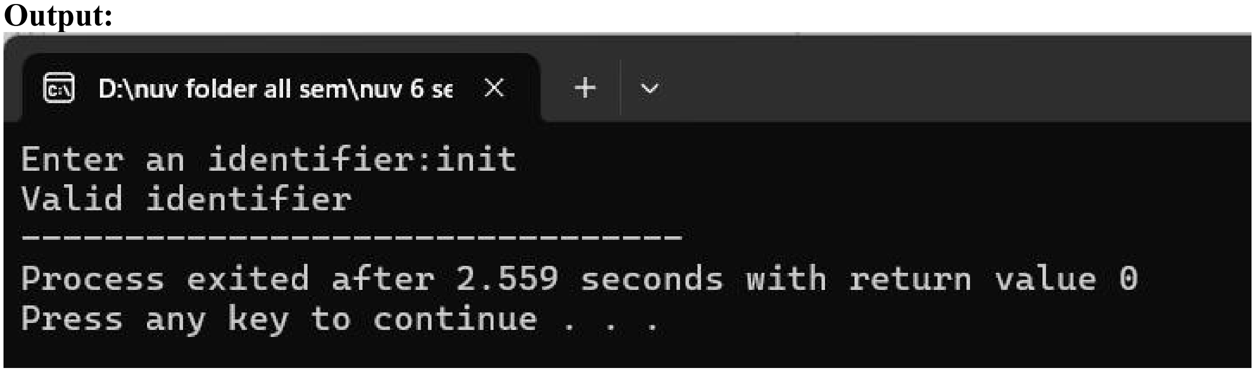
Practical-2

a) **Write a program to recognize the valid identifiers.**

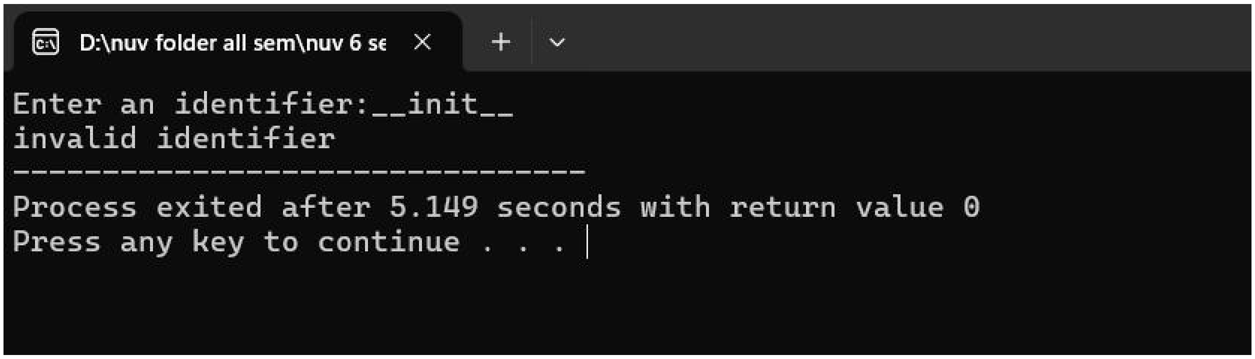
Input:

```
//Write a program to recognize the valid identifiers.
#include <stdio.h>
#include <ctype.h>
int main()
{
      char a[10];
      int flag, i=1;

      printf("Enter an identifier:");
      scanf("%s",&a);


      if(isalpha(a[0])){

            flag = 1; // If the first character is an alphabet, set flag = 1 (indicating a
valid start).

      }
      else
            printf("invalid identifier");


      while (a[i] != '\0') {
    if (!isalnum(a[i]) && a[i] != '_') {
      flag = 0;
      break;

    }
    i++;
  }


      if(flag == 1){
            printf("Valid identifier");
      }
      //getch();
}
```

Output:



```
Enter an identifier:init
Valid identifier
--------------------------------
Process exited after 2.559 seconds with return value 0
Press any key to continue . . .
```



```
Enter an identifier:__init__
invalid identifier
--------------------------------
Process exited after 5.149 seconds with return value 0
Press any key to continue . . .
```

b) **Write a program to recognize the valid operators.**

Input:

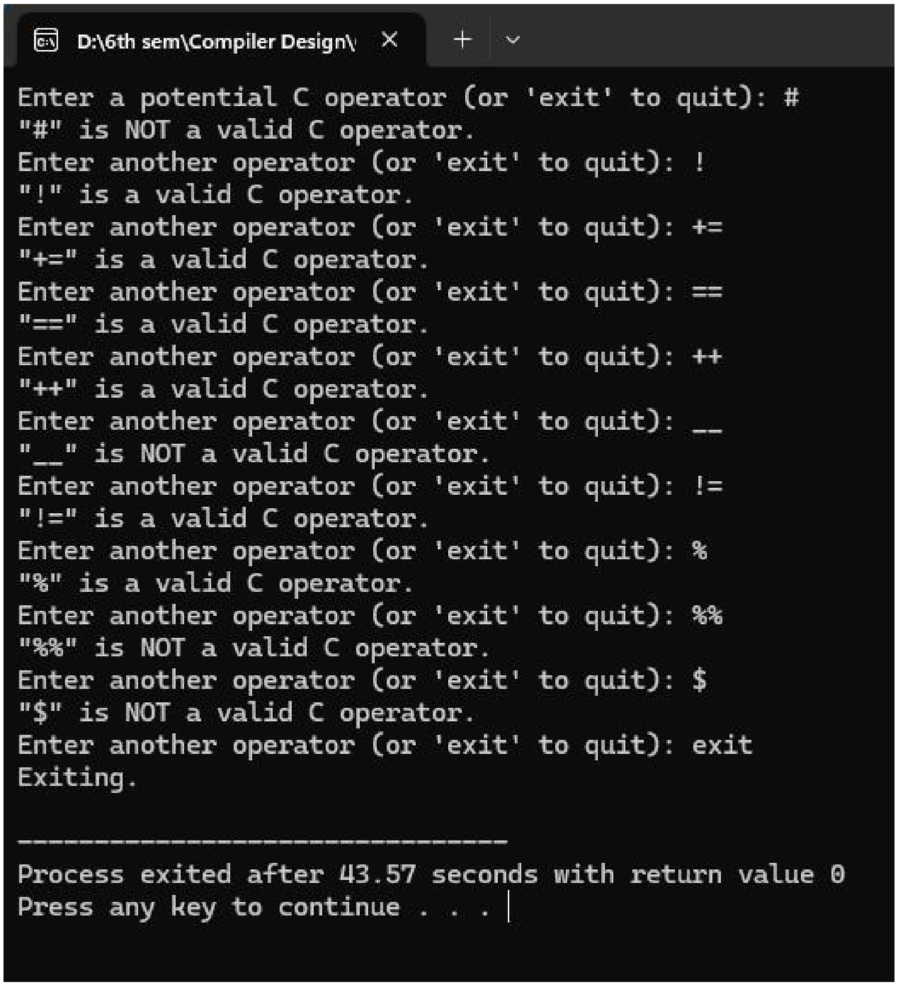
```
//Lexical Analyzer
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#define BUFFER_SIZE 1000

void check_keyword_or_identifier(char *lexeme);
void recognize_number(char *lexeme);
void recognize_operator(char c);
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;


   f1 = fopen("hello.txt", "r");
   if (f1 == NULL) {
      printf("Error: Could not open input.txt\n");
      return;
   }

   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);

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(c);
                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);
                i = 0;
                state = 0;
                f--;
            }
            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--;
    }
    break;

case 3:
    if (isdigit(c)) {
        lexeme[i++] = c;
    } else {
        lexeme[i] = '\0';
        recognize_number(lexeme);
        i = 0;
        state = 0;
        f--;
    }
    break;

case 4:
    if (isdigit(c) || c == '+' || c == '-') {
        state = 5;
        lexeme[i++] = c;
    } else {
        lexeme[i] = '\0';
        recognize_number(lexeme);
```

```
                    i = 0;
                    state = 0;
                    f--;
                }
                break;

            case 5:
                if (isdigit(c)) {
                    lexeme[i++] = c;
                } else {
                    lexeme[i] = '\0';
                    recognize_number(lexeme);
                    i = 0;
                    state = 0;
                    f--;
                }
                break;
        }
        f++;
    }

    free(buffer);
}


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",
        "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);
}


void recognize_number(char *lexeme) {
    printf("%s is a valid number\n", lexeme);
}

void recognize_operator(char c) {

    char operators[][3] = {"+", "-", "*", "/", "%", "=", "==", "!=", "<", ">", "<=", ">="};
    char next = getchar();
    char op[3] = {c, next, '\0'};
        int i = 0;
    for (i = 0; i < 12; i++) {
        if (strcmp(op, operators[i]) == 0) {
            printf("%s is an operator\n", op);
            return;
        }
    }

    printf("%c is an operator\n", c);
    ungetc(next, stdin);
}



void recognize_comment(char *buffer, int *index) {
    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;
        }
    }
}
```

Output:



```
Enter a potential C operator (or 'exit' to quit): #
"#" is NOT a valid C operator.
Enter another operator (or 'exit' to quit): !
"!" is a valid C operator.
Enter another operator (or 'exit' to quit): +=
"+=" is a valid C operator.
Enter another operator (or 'exit' to quit): ==
"==" is a valid C operator.
Enter another operator (or 'exit' to quit): ++
"++" is a valid C operator.
Enter another operator (or 'exit' to quit): __
"__" is NOT a valid C operator.
Enter another operator (or 'exit' to quit): !=
"!=" is a valid C operator.
Enter another operator (or 'exit' to quit): %
"%" is a valid C operator.
Enter another operator (or 'exit' to quit): %%
"%%" is NOT a valid C operator.
Enter another operator (or 'exit' to quit): $
"$" is NOT a valid C operator.
Enter another operator (or 'exit' to quit): exit
Exiting.

--------------------------------
Process exited after 43.57 seconds with return value 0
Press any key to continue . . .
```

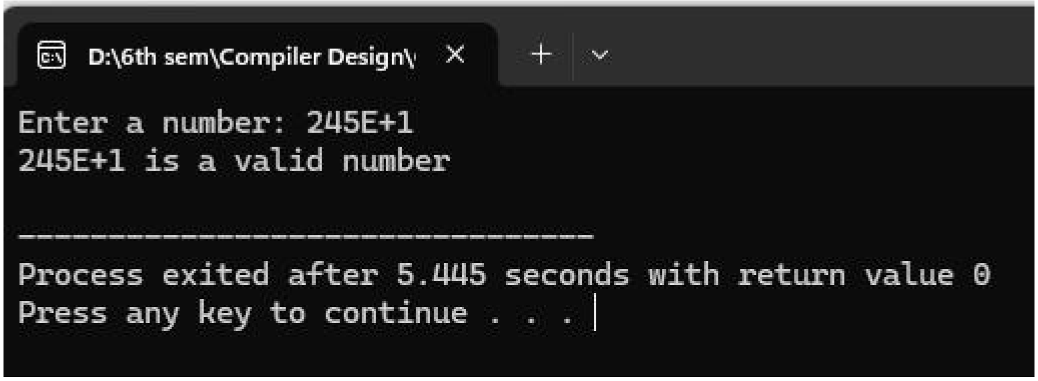
c) **Write a program to recognize the valid number.**

Input:

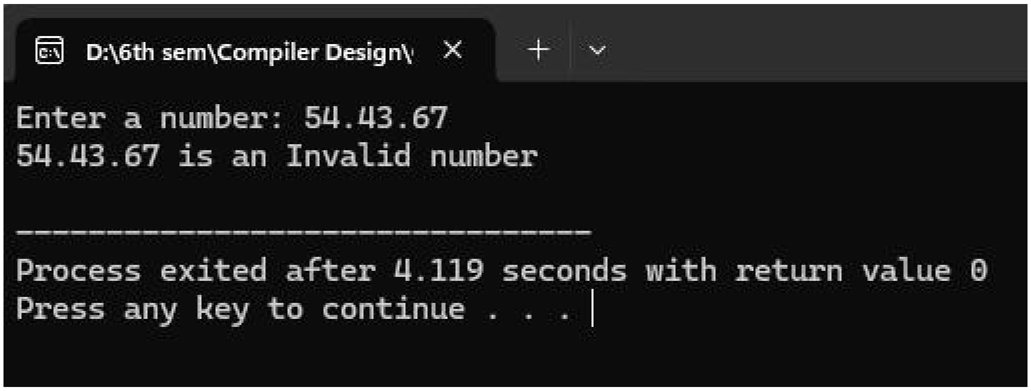
```
//Lexical Analyzer
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#define BUFFER_SIZE 1000

void check_keyword_or_identifier(char *lexeme);
void recognize_number(char *lexeme);
void recognize_operator(char c);
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;


   f1 = fopen("hello.txt", "r");
   if (f1 == NULL) {
      printf("Error: Could not open input.txt\n");
      return;
   }

   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);

   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(c);
        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);
        i = 0;
        state = 0;
        f--;
    }
    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--;
        }
        break;

    case 3:
        if (isdigit(c)) {
            lexeme[i++] = c;
        } else {
            lexeme[i] = '\0';
            recognize_number(lexeme);
            i = 0;
            state = 0;
            f--;
        }
        break;

    case 4:
        if (isdigit(c) || c == '+' || c == '-') {
            state = 5;
            lexeme[i++] = c;
        } else {
            lexeme[i] = '\0';
            recognize_number(lexeme);
            i = 0;
            state = 0;
            f--;
        }
        break;

    case 5:
        if (isdigit(c)) {
            lexeme[i++] = c;
        } else {
            lexeme[i] = '\0';
            recognize_number(lexeme);
            i = 0;
            state = 0;
            f--;
        }
        break;
        }
        f++;
    }

    free(buffer);
}


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",
        "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);
}


void recognize_number(char *lexeme) {
    printf("%s is a valid number\n", lexeme);
}

void recognize_operator(char c) {

    char operators[][3] = {"+", "-", "*", "/", "%", "=", "==", "!=", "<", ">", "<=", ">="};
    char next = getchar();
    char op[3] = {c, next, '\0'};
            int i = 0;
    for (i = 0; i < 12; i++) {
        if (strcmp(op, operators[i]) == 0) {
            printf("%s is an operator\n", op);
            return;
        }
    }
    printf("%c is an operator\n", c);
    ungetc(next, stdin);
}


void recognize_comment(char *buffer, int *index) {
    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;
        }
    }
}
```

Output:



```
Enter a number: 245E+1
245E+1 is a valid number

--------------------------------
Process exited after 5.445 seconds with return value 0
Press any key to continue . . .
```



```
Enter a number: 54.43.67
54.43.67 is an Invalid number

--------------------------------
Process exited after 4.119 seconds with return value 0
Press any key to continue . . .
```

d) **Write a program to recognize the valid comments.**

Input:

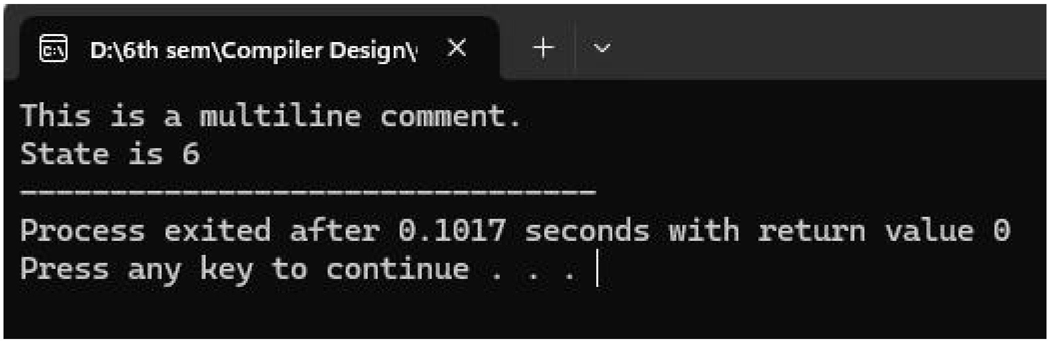
```
//Lexical Analyzer
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#define BUFFER_SIZE 1000

void check_keyword_or_identifier(char *lexeme);
void recognize_number(char *lexeme);
void recognize_operator(char c);
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;


    f1 = fopen("hello.txt", "r");
    if (f1 == NULL) {
        printf("Error: Could not open input.txt\n");
        return;
    }

    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);

    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(c);
                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);
                i = 0;
                state = 0;
                f--;
            }
            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--;
    }
    break;

case 3:
    if (isdigit(c)) {
        lexeme[i++] = c;
    } else {
        lexeme[i] = '\0';
        recognize_number(lexeme);
        i = 0;
        state = 0;
        f--;
    }
    break;

case 4:
    if (isdigit(c) || c == '+' || c == '-') {
        state = 5;
        lexeme[i++] = c;
    } else {
        lexeme[i] = '\0';
        recognize_number(lexeme);
        i = 0;
        state = 0;
        f--;
    }
    break;

case 5:
    if (isdigit(c)) {
        lexeme[i++] = c;
    } else {
        lexeme[i] = '\0';
        recognize_number(lexeme);
        i = 0;
        state = 0;
        f--;
    }
    break;
```

```
        }
        f++;
    }

    free(buffer);
}


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",
        "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);
}


void recognize_number(char *lexeme) {
    printf("%s is a valid number\n", lexeme);
}

void recognize_operator(char c) {

    char operators[][3] = {"+", "-", "*", "/", "%", "=", "==", "!=", "<", ">", "<=", ">="};
    char next = getchar();
    char op[3] = {c, next, '\0'};
            int i = 0;
    for (i = 0; i < 12; i++) {
        if (strcmp(op, operators[i]) == 0) {
            printf("%s is an operator\n", op);
            return;
        }
    }
    printf("%c is an operator\n", c);
    ungetc(next, stdin);
}


void recognize_comment(char *buffer, int *index) {
    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;
        }
    }
}
```

Output:



```
This is a multiline comment.
State is 6
--------------------------------
Process exited after 0.1017 seconds with return value 0
Press any key to continue . . .
```

e) Program to implement Lexical Analyzer.

Input:

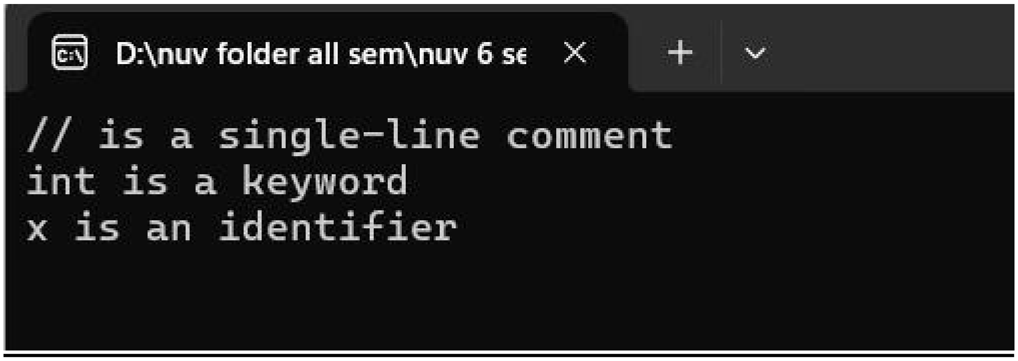
```
//Lexical Analyzer
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#define BUFFER_SIZE 1000

void check_keyword_or_identifier(char *lexeme);
void recognize_number(char *lexeme);
void recognize_operator(char c);
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;


   f1 = fopen("hello.txt", "r");
   if (f1 == NULL) {
      printf("Error: Could not open input.txt\n");
      return;
   }

   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);

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(c);
                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);
            i = 0;
            state = 0;
            f--;
        }
        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--;
    }
    break;

case 3:
    if (isdigit(c)) {
        lexeme[i++] = c;
    } else {
        lexeme[i] = '\0';
        recognize_number(lexeme);
        i = 0;
        state = 0;
        f--;
    }
    break;

case 4:
    if (isdigit(c) || c == '+' || c == '-') {
        state = 5;
        lexeme[i++] = c;
    } else {
        lexeme[i] = '\0';
        recognize_number(lexeme);
        i = 0;
        state = 0;
```

```
                    f--;
                }
                break;

            case 5:
                if (isdigit(c)) {
                    lexeme[i++] = c;
                } else {
                    lexeme[i] = '\0';
                    recognize_number(lexeme);
                    i = 0;
                    state = 0;
                    f--;
                }
                break;
        }
        f++;
    }

    free(buffer);
}


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",
        "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);
}


void recognize_number(char *lexeme) {
    printf("%s is a valid number\n", lexeme);
}

void recognize_operator(char c) {

    char operators[][3] = {"+", "-", "*", "/", "%", "=", "==", "!=", "<", ">", "<=", ">="};
    char next = getchar();
    char op[3] = {c, next, '\0'};
            int i = 0;
    for (i = 0; i < 12; i++) {
        if (strcmp(op, operators[i]) == 0) {
            printf("%s is an operator\n", op);
            return;
        }
    }
```

```
        printf("%c is an operator\n", c);
        ungetc(next, stdin);
    }



    void recognize_comment(char *buffer, int *index) {
        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;
            }
        }
    }
```

Output:



```
// is a single-line comment
int is a keyword
x is an identifier
```

Practical - 3

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

Introduction:

Lexical analysis is the first phase of a compiler, where the source code is converted into a sequence of tokens. Tokens are the basic building blocks of programming languages such as keywords, identifiers, constants, operators, and symbols. Writing a lexical analyzer manually is both time-consuming and error-prone. To address this, tools like **LEX** and **Flex** are used to automatically generate efficient lexical analyzers.

LEX and Flex:

LEX is a tool developed for generating lexical analyzers based on patterns described using regular expressions. It reads a given set of rules and produces a C program that can identify the corresponding lexical elements in the input stream.
Flex (Fast Lexical Analyzer) is a free and open-source alternative to LEX. It is compatible with LEX specifications but provides improved performance and additional features. Flex scans the source code using the rules defined in a ".l" file and outputs a C source file that can be compiled to perform token recognition.

Both LEX and Flex are typically used in conjunction with parser generators like **YACC** or **Bison**, enabling the seamless integration of lexical and syntax analysis in compiler construction.

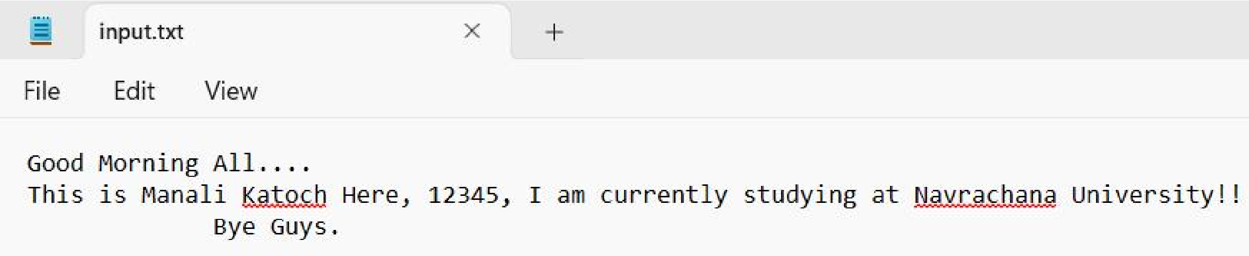
Practical - 4

Sample Program

Input:

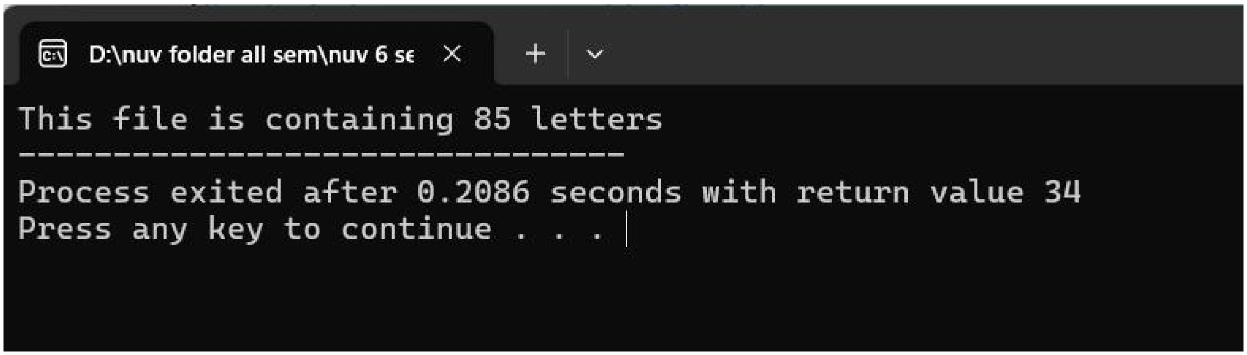
```
%{
   #include<stdio.h>
   int letters=0;
%}
%%
[a-zA-Z]  {letters++;}
\n  ;
.  ;
%%
void main(){
   yyin=fopen("input.txt","r");
   yylex();
   printf("This file is containing %d letters",letters);
}
int yywrap(){  return(1);}
```

Text file:



```
Good Morning All....
This is Manali Katoch Here, 12345, I am currently studying at Navrachana University!!
              Bye Guys.
```

Output:



```
This file is containing 85 letters
--------------------------------
Process exited after 0.2086 seconds with return value 34
Press any key to continue . . .
```

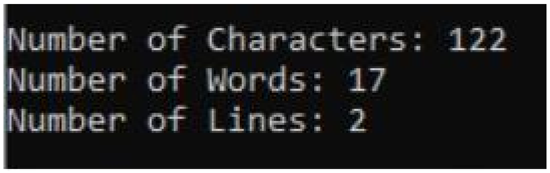
Implement following programs using Lex.

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

Input:

```
%{
#include <stdio.h>
int char_count = 0, word_count = 0, line_count = 0;
%}
%%
\n         { line_count++; char_count++; }
[^\n\t ]+   { word_count++; char_count += yyleng; }
.          { char_count++; }
%%
int main() {
   FILE *file = fopen("input.txt", "r");  // Open the file
   if (!file) {
      printf("Error: Could not open file 'input.txt'\n");
      return 1;
   }
   yyin = file;  // Set Lex input to the file
   yylex();      // Process the file
   printf("\nNumber of Characters: %d", char_count);
   printf("\nNumber of Words: %d", word_count);
   printf("\nNumber of Lines: %d\n", line_count);
   fclose(file);  // Close the file
   return 0;
}
int yywrap() {
   return 1;
}
```

Output:



```
Number of Characters: 122
Number of Words: 17
Number of Lines: 2
```

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

Input:

Lex Code [count1.l]

```
%{
   int vowels = 0;
   int consonants = 0;
   FILE *yyin;
%}

%%

[aeiouAEIOU]    { vowels++; }
[a-zA-Z]      { consonants++; }
.|\n         { /* Ignore other characters */ }

%%
int yywrap() {

   return 1;

}

int main(int argc, char *argv[]) {

   if (argc < 2) {

      printf("Usage: %s input2.txt\n", argv[0]);

      return 1;

   }


   FILE *file = fopen(argv[1], "r");

   if (!file) {

      printf("Cannot open file %s\n", argv[1]);

      return 1;

   }
```

```
    yyin = file;
    yylex();

    printf("Number of vowels: %d\n", vowels);
    printf("Number of consonants: %d\n", consonants);

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

Input2.txt Code:

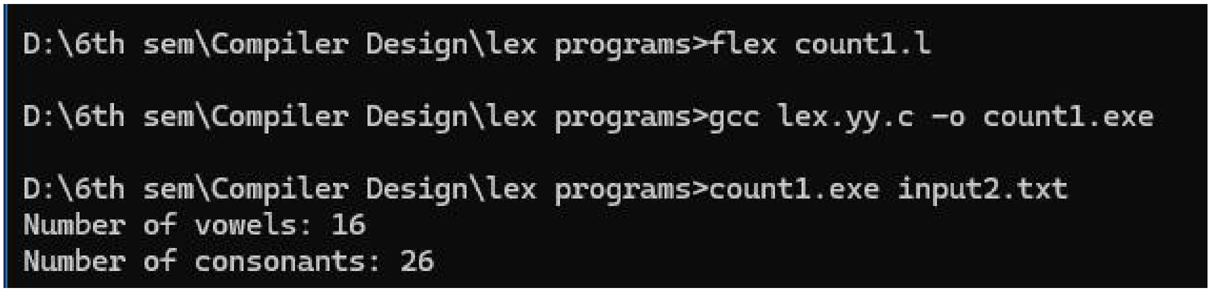
Hello World!

Lex is fun.

123

Manali Katoch born on 16 Dec 2003

Output:



```
D:\6th sem\Compiler Design\lex programs>flex count1.l

D:\6th sem\Compiler Design\lex programs>gcc lex.yy.c -o count1.exe

D:\6th sem\Compiler Design\lex programs>count1.exe input2.txt
Number of vowels: 16
Number of consonants: 26
```

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

Input:

Lex Code [numbers.l]

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

%%

[0-9]+(\.[0-9]+)?    { printf("Number found: %s\n", yytext); }
.|\n                 { /* Ignore all other characters */ }

%%

int yywrap() {
   return 1;
}

int main() {
   yylex();  // Start the lexical analysis
   return 0;
}
```

Input2.txt Code:

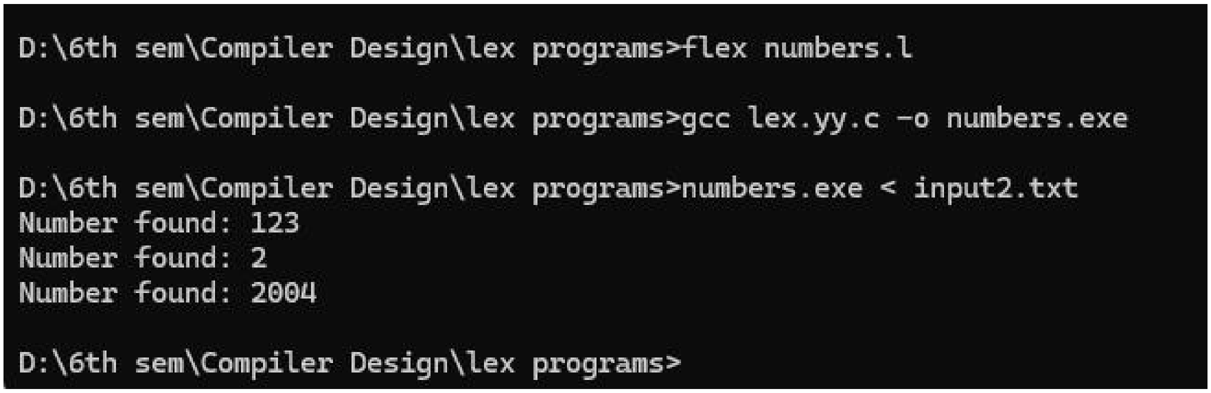
Hello World!

Lex is fun.

123

Manali Katoch born on 16 Dec 2004

Output:



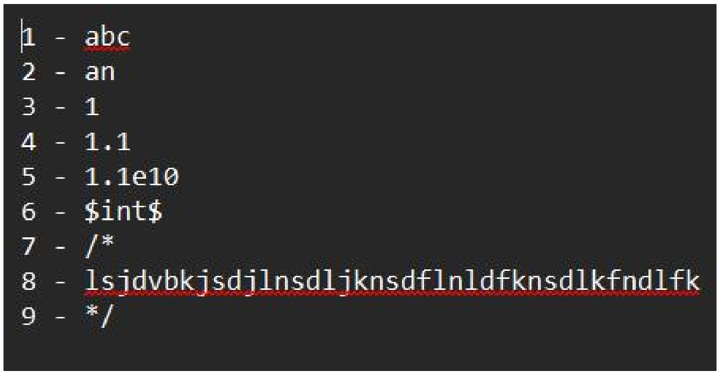
```
D:\6th sem\Compiler Design\lex programs>flex numbers.l

D:\6th sem\Compiler Design\lex programs>gcc lex.yy.c -o numbers.exe

D:\6th sem\Compiler Design\lex programs>numbers.exe < input2.txt
Number found: 123
Number found: 2
Number found: 2004

D:\6th sem\Compiler Design\lex programs>
```

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

Input:

```
%{
#include<stdio.h>
int i = 0;
char line[1000];  // Buffer to store line content
int line_pos = 0;  // Position in line buffer
%}
%%
[^\n]   {line[line_pos++] = yytext[0];}
[\n]    {i++; line[line_pos] = '\0';fprintf(yyout, "%d - %s\n", i, line);line_pos = 0;}
%%
int main() {
   yyin = fopen("input.txt", "r");

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

   yylex();

   fclose(yyin);
   fclose(yyout);
   return 0;
}
int yywrap() {
   return 1;
}
```

Output:



```
1 - abc
2 - an
3 - 1
4 - 1.1
5 - 1.1e10
6 - $int$
7 - /*
8 - lsjdvbkjsdjlnsdljknsdflnldfknsdlkfndlfk
9 - */
```

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

Input:

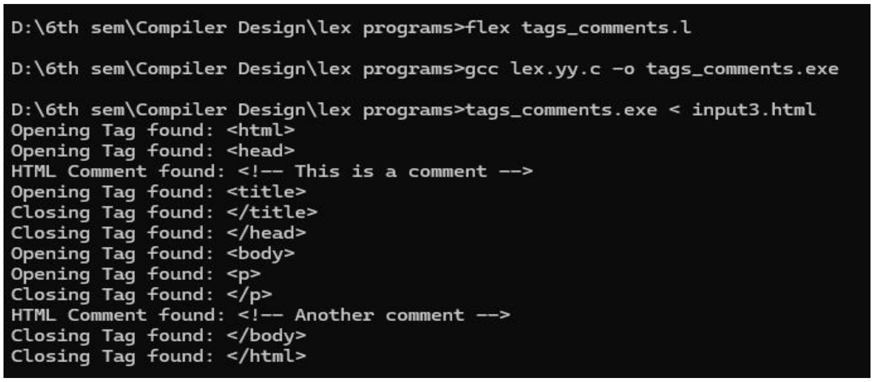
Lex Code [tags_comments.l]

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

%%

"<!--"([^>]|[\n])*"-->"            { printf("HTML Comment found: %s\n", yytext); }
"<"[a-zA-Z][a-zA-Z0-9]*">"         { printf("Opening Tag found: %s\n", yytext); }
"</"[a-zA-Z][a-zA-Z0-9]*">"        { printf("Closing Tag found: %s\n", yytext); }
"<"[a-zA-Z][^>]*"/>"               { printf("Self-closing Tag found: %s\n", yytext); }

.|\n                               { /* Ignore other content */ }

%%

int yywrap() { return 1; }

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

input3.html code:

```
<html>
<head>
<!-- This is a comment -->
<title>Page Title</title>
</head>
<body>
<p>Welcome to the page!</p>
<!-- Another comment -->
</body>
</html>
```

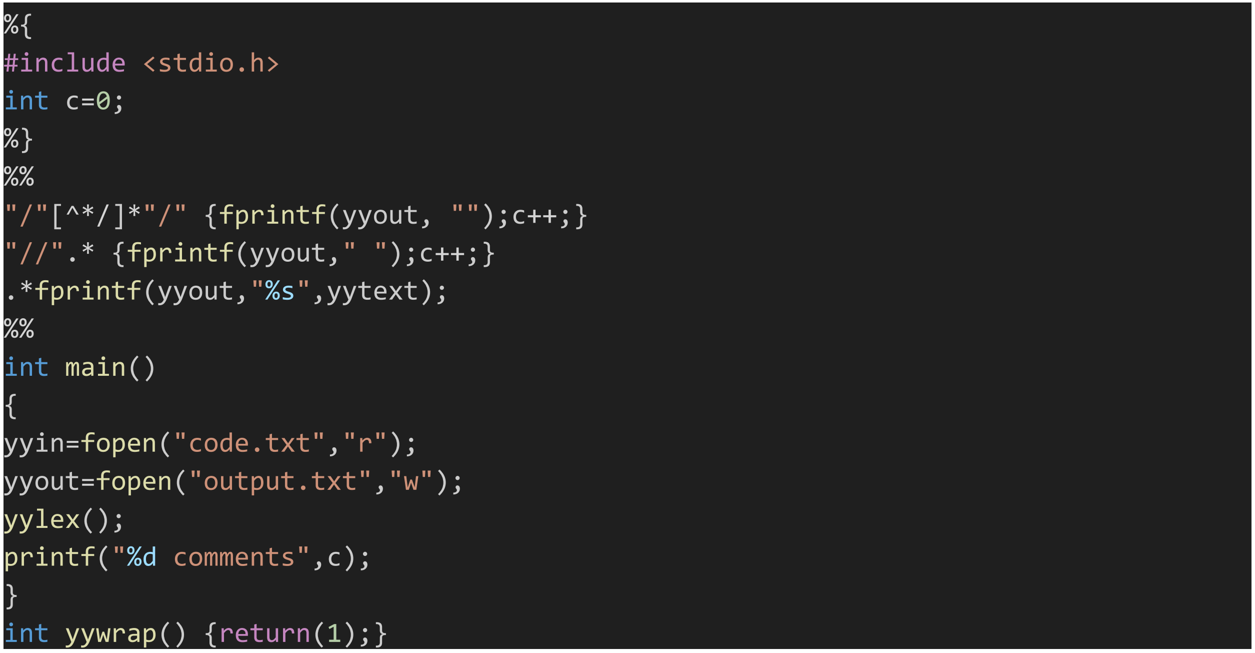
Output:



```
D:\6th sem\Compiler Design\lex programs>flex tags_comments.l

D:\6th sem\Compiler Design\lex programs>gcc lex.yy.c -o tags_comments.exe

D:\6th sem\Compiler Design\lex programs>tags_comments.exe < input3.html
Opening Tag found: <html>
Opening Tag found: <head>
HTML Comment found: <!-- This is a comment -->
Opening Tag found: <title>
Closing Tag found: </title>
Closing Tag found: </head>
Opening Tag found: <body>
Opening Tag found: <p>
Closing Tag found: </p>
HTML Comment found: <!-- Another comment -->
Closing Tag found: </body>
Closing Tag found: </html>
```

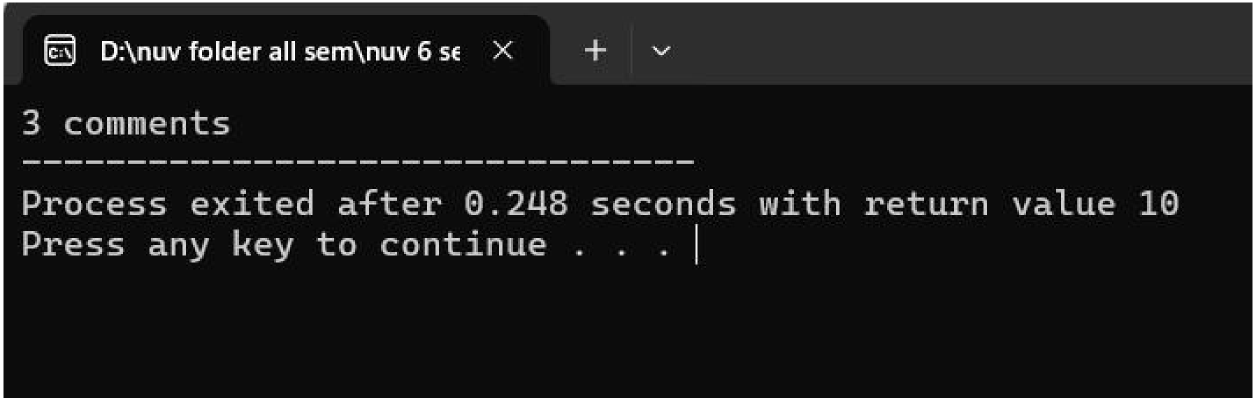
Practical - 5

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.**
Input:

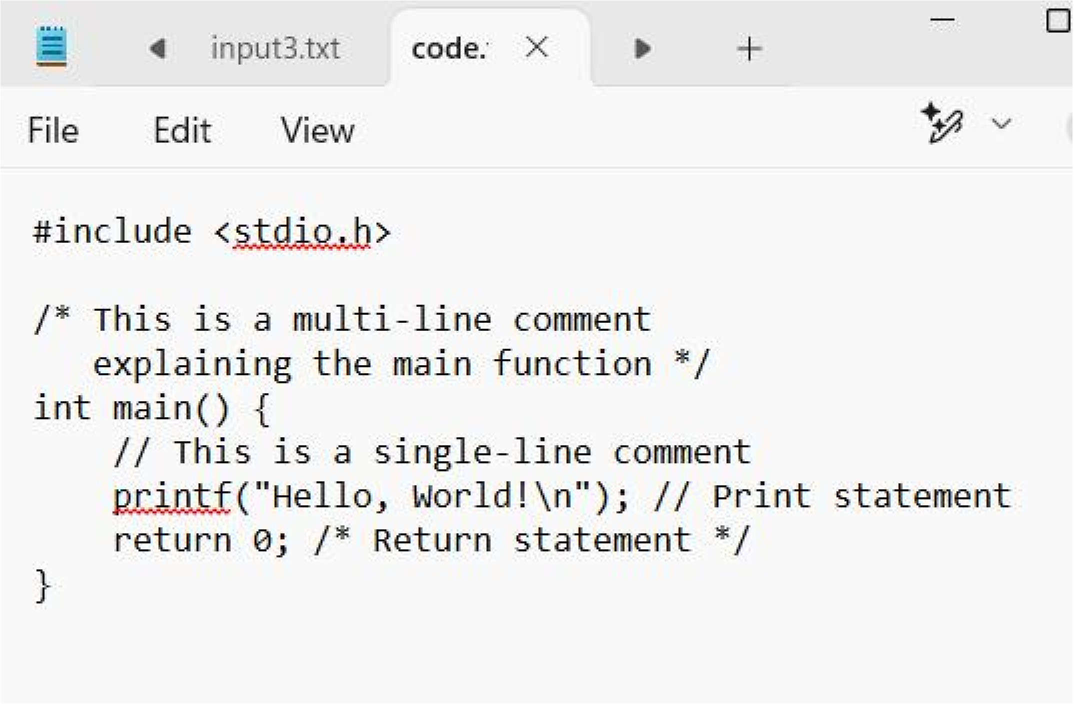


```
%{
#include <stdio.h>
int c=0;
%}
%%
"/"[^*/]*"/" {fprintf(yyout, "");c++;}
"//".* {fprintf(yyout," ");c++;}
.*fprintf(yyout,"%s",yytext);
%%
int main()
{
yyin=fopen("code.txt","r");
yyout=fopen("output.txt","w");
yylex();
printf("%d comments",c);
}
int yywrap() {return(1);}
```

Output:



```
3 comments
--------------------------------
Process exited after 0.248 seconds with return value 10
Press any key to continue . . .
```



```
#include <stdio.h>

/* This is a multi-line comment
   explaining the main function */
int main() {
    // This is a single-line comment
    printf("Hello, World!\n"); // Print statement
    return 0; /* Return statement */
}
```

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

Input:

tokenizer.l

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

DIGIT       [0-9]
LETTER      [a-zA-Z]
IDENTIFIER  {LETTER}({LETTER}|{DIGIT})*
NUMBER      {DIGIT}+(\.{DIGIT}+)?
OPERATOR    [+\-*/%=><|&!]
SPECIAL     [(){}[\];,]
LITERAL     \"(\\.|[^"\\])*\"

%%

"auto"      { printf("Keyword: %s\n", yytext); }
"break"     { printf("Keyword: %s\n", yytext); }
"case"      { printf("Keyword: %s\n", yytext); }
"char"      { printf("Keyword: %s\n", yytext); }
"const"     { printf("Keyword: %s\n", yytext); }
"continue"  { printf("Keyword: %s\n", yytext); }
"default"   { printf("Keyword: %s\n", yytext); }
```

```
"do"        { printf("Keyword: %s\n", yytext); }
"double"    { printf("Keyword: %s\n", yytext); }
"else"      { printf("Keyword: %s\n", yytext); }
"enum"      { printf("Keyword: %s\n", yytext); }
"extern"    { printf("Keyword: %s\n", yytext); }
"float"     { printf("Keyword: %s\n", yytext); }
"for"       { printf("Keyword: %s\n", yytext); }
"goto"      { printf("Keyword: %s\n", yytext); }
"if"        { printf("Keyword: %s\n", yytext); }
"int"       { printf("Keyword: %s\n", yytext); }
"long"      { printf("Keyword: %s\n", yytext); }
"register"  { printf("Keyword: %s\n", yytext); }
"return"    { printf("Keyword: %s\n", yytext); }
"short"     { printf("Keyword: %s\n", yytext); }
"signed"    { printf("Keyword: %s\n", yytext); }
"sizeof"    { printf("Keyword: %s\n", yytext); }
"static"    { printf("Keyword: %s\n", yytext); }
"struct"    { printf("Keyword: %s\n", yytext); }
"switch"    { printf("Keyword: %s\n", yytext); }
"typedef"   { printf("Keyword: %s\n", yytext); }
"union"     { printf("Keyword: %s\n", yytext); }
"unsigned"  { printf("Keyword: %s\n", yytext); }
"void"      { printf("Keyword: %s\n", yytext); }
"volatile"  { printf("Keyword: %s\n", yytext); }
"while"     { printf("Keyword: %s\n", yytext); }

{IDENTIFIER}    { printf("Identifier: %s\n", yytext); }
```

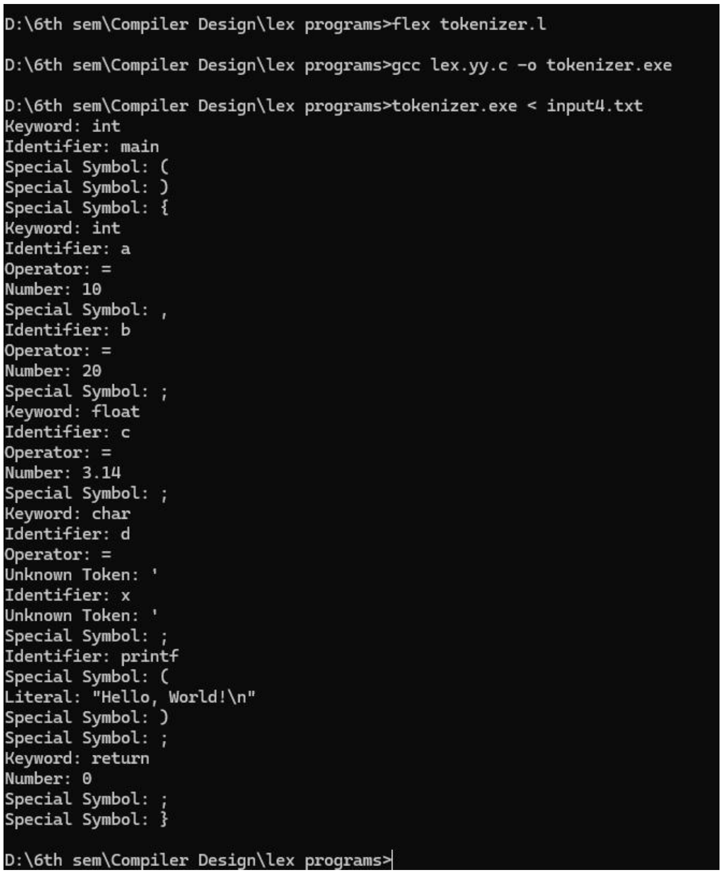
```
{NUMBER}       { printf("Number: %s\n", yytext); }
{OPERATOR}     { printf("Operator: %s\n", yytext); }
{SPECIAL}      { printf("Special Symbol: %s\n", yytext); }
{LITERAL}      { printf("Literal: %s\n", yytext); }

[ \t\n]        { /* Ignore whitespace and newlines */ }

.              { printf("Unknown Token: %s\n", yytext); }

%%

int yywrap() {
   return 1;
}

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

input4.txt

```
int main() {
   int a = 10, b = 20;
   float c = 3.14;
   char d = 'x';
   printf("Hello, World!\n");
   return 0;
```

}

Output:



```
D:\6th sem\Compiler Design\lex programs>flex tokenizer.l

D:\6th sem\Compiler Design\lex programs>gcc lex.yy.c -o tokenizer.exe

D:\6th sem\Compiler Design\lex programs>tokenizer.exe < input4.txt
Keyword: int
Identifier: main
Special Symbol: (
Special Symbol: )
Special Symbol: {
Keyword: int
Identifier: a
Operator: =
Number: 10
Special Symbol: ,
Identifier: b
Operator: =
Number: 20
Special Symbol: ;
Keyword: float
Identifier: c
Operator: =
Number: 3.14
Special Symbol: ;
Keyword: char
Identifier: d
Operator: =
Unknown Token: '
Identifier: x
Unknown Token: '
Special Symbol: ;
Identifier: printf
Special Symbol: (
Literal: "Hello, World!\n"
Special Symbol: )
Special Symbol: ;
Keyword: return
Number: 0
Special Symbol: ;
Special Symbol: }

D:\6th sem\Compiler Design\lex programs>
```

Practical - 6

Program to implement Recursive Descent Parsing in C.

Code:

```
#include <stdio.h>

#include <string.h>

#define SUCCESS 1

#define FAILED 0

// Function prototypes

int E(), Edash(), T(), Tdash(), F();

const char *cursor;

char string[64];

int main()

{

   puts("Enter the string");

   scanf("%s", string); // Read input from the user

   cursor = string;

   puts("");

   puts("Input          Action");

   puts("--------------------------------");

   // Call the starting non-terminal E

   if (E() && *cursor == '\0')
```

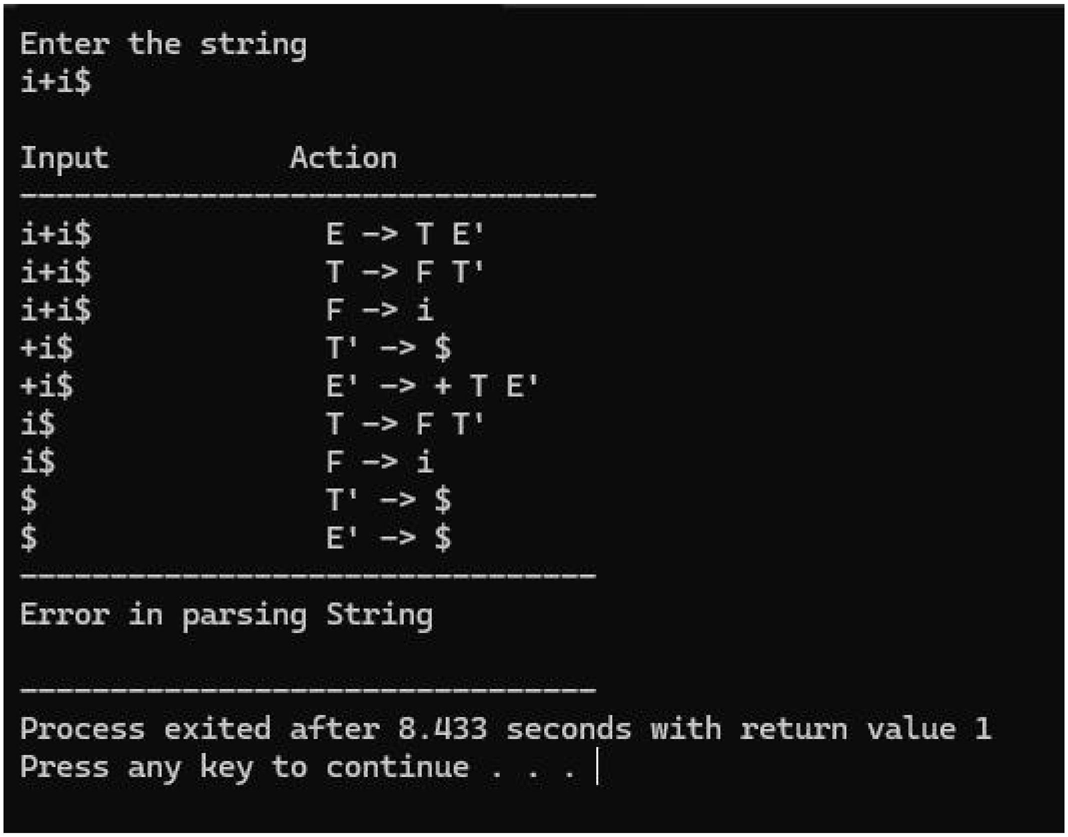
```
    { // If parsing is successful and the cursor has reached the end
        puts("--------------------------------");
        puts("String is successfully parsed");
        return 0;
    }
    else
    {
        puts("--------------------------------");
        puts("Error in parsing String");
        return 1;
    }
}
// Grammar rule: E -> T E'
int E()
{
    printf("%-16s E -> T E'\n", cursor);
    if (T())
    { // Call non-terminal T
        if (Edash())
        { // Call non-terminal E'
            return SUCCESS;
        }
        else
        {
            return FAILED;
        }
    }
    else
    {
        return FAILED;
    }
}
// Grammar rule: E' -> + T E' | $
int Edash()
{
    if (*cursor == '+')
    {
        printf("%-16s E' -> + T E'\n", cursor);
        cursor++;

        if (T())
        { // Call non-terminal T
            if (Edash())
            { // Call non-terminal E'
```

```
                return SUCCESS;

            }

            else

            {

                return FAILED;

            }

        }

        else

        {

            return FAILED;

        }

    }

    else

    {

        printf("%-16s E' -> $\n", cursor);

        return SUCCESS;

    }

}

// Grammar rule: T -> F T'

int T()

{

    printf("%-16s T -> F T'\n", cursor);
    if (F())
    { // Call non-terminal F
        if (Tdash())
        { // Call non-terminal T'
            return SUCCESS;
        }
        else
        {
            return FAILED;
        }
    }
    else
    {
        return FAILED;
    }
}

// Grammar rule: T' -> * F T' | $
int Tdash()
{
    if (*cursor == '*')
    {
```

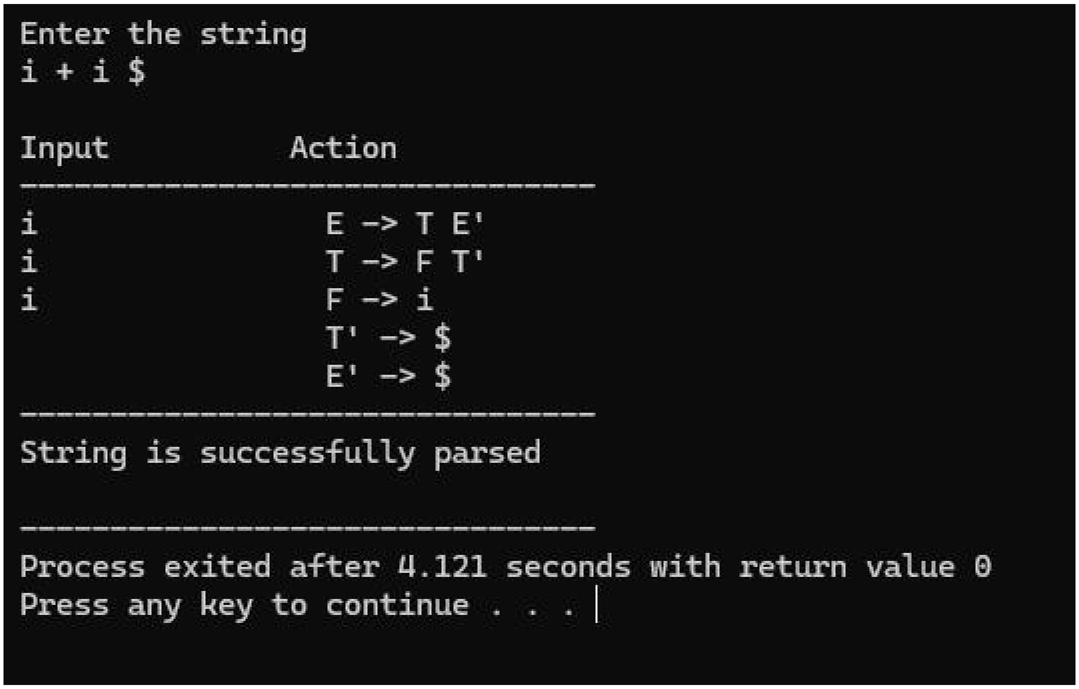
```
    printf("%-16s T' -> * F T'\n", cursor);

    cursor++;


    if (F())

    { // Call non-terminal F

        if (Tdash())

        { // Call non-terminal T'

            return SUCCESS;

        }

        else

        {

            return FAILED;

        }

    }

    else

    {

        return FAILED;

    }

}

else

{

    printf("%-16s T' -> $\n", cursor);
```

```
        return SUCCESS;

    }

}

// Grammar rule: F -> ( E ) | i

int F()

{

    if (*cursor == '(')

    {

        printf("%-16s F -> ( E )\n", cursor);

        cursor++;

        if (E())

        { // Call non-terminal E

            if (*cursor == ')')

            {

                cursor++;

                return SUCCESS;

            }

            else

            {

                return FAILED;

            }

        }
        else
        {
            return FAILED;
        }
    }
    else if (*cursor == 'i')
    {
        printf("%-16s F -> i\n", cursor);
        cursor++;
        return SUCCESS;
    }
    else
    {
        return FAILED;
    }
}
```

Output:



```
Enter the string
i+i$

Input          Action
--------------------------------
i+i$             E -> T E'
i+i$             T -> F T'
i+i$             F -> i
+i$              T' -> $
+i$              E' -> + T E'
i$               T -> F T'
i$               F -> i
$                T' -> $
$                E' -> $
--------------------------------
Error in parsing String

--------------------------------
Process exited after 8.433 seconds with return value 1
Press any key to continue . . .
```



```
Enter the string
i + i $

Input          Action
--------------------------------
i                E -> T E'
i                T -> F T'
i                F -> i
                 T' -> $
                 E' -> $
--------------------------------
String is successfully parsed

--------------------------------
Process exited after 4.121 seconds with return value 0
Press any key to continue . . .
```

Practical-7

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

Input:

What is YACC ?

- YACC (Yet Another Compiler-Compiler) is a tool used in compiler design to generate parsers. It helps you build the syntax analysis part of a compiler.
- It was developed by Stephen C. Johnson at AT&T Bell Labs.

Why is YACC used ?

- Writing a parser manually (like recursive descent) is complex and error-prone.
- YACC automates this by generating C code for the parser from a grammar specification.
- It works well with lex, the lexical analyzer generator.

How does YACC work ?

- You write a grammar using BNF (Backus-Naur Form) or similar syntax.
- You assign semantic actions to grammar rules (using C code).
- YACC generates a parser in C that uses a bottom-up parsing algorithm (usually LALR(1)).
- The parser works with lex to analyze tokens.

Structure of a YACC file

A YACC source file has three sections, separated by %%:

```
%{
   // Declarations (C code, headers)
%}

%token ID NUM // Token definitions
```

```
%%

E : E '+' T   { printf("Adding\n"); }
  | T         { /* do nothing */ }
  ;

T : T '*' F   { printf("Multiplying\n"); }
  | F         { /* do nothing */ }
  ;

F : '(' E ')'
  | ID
  | NUM
  ;

%%

// Additional C code (main function etc.)
```

YACC and LEX Integration

- LEX handles scanning/tokenizing (splits input into tokens).
- YACC handles parsing (checks if token sequence is valid as per grammar).
- They work together to build front ends for compilers.

Advantages of YACC

- Speeds up parser development.
- Helps build robust parsers for programming languages.
- Well-suited for formal language processing tasks.

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

expr.l code:

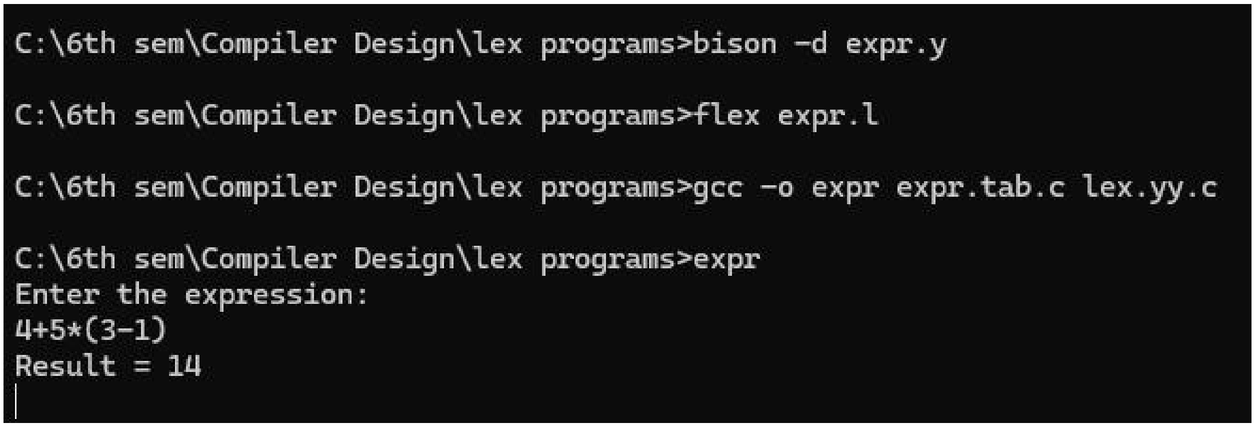
```
%{
   #include "expr.tab.h"
   #include <stdlib.h>
%}

%%

[0-9]+        { yylval.ival = atoi(yytext); return NUMBER; }
[a-zA-Z]+     { yylval.ival = 0; return ID; }
[ \t]+        ; // skip whitespace
\n            { return '\n'; }
.             { return yytext[0]; }

%%

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

expr.y code:

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

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

%union {
   int ival;
}

%token <ival> NUMBER
%token <ival> ID
%type <ival> E

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

%%

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

E:
   E '+' E    { $$ = $1 + $3; }
 | E '-' E    { $$ = $1 - $3; }
 | E '*' E    { $$ = $1 * $3; }
 | E '/' E    { $$ = $1 / $3; }
 | '-' E      { $$ = -$2; }
 | '(' E ')'  { $$ = $2; }
 | NUMBER     { $$ = $1; }
 | ID         { $$ = $1; }
 ;
```

```
%%

int main(void) {
    printf("Enter the expression:\n");
    yyparse();
    return 0;
}

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}
```

Output:



```
C:\6th sem\Compiler Design\lex programs>bison -d expr.y

C:\6th sem\Compiler Design\lex programs>flex expr.l

C:\6th sem\Compiler Design\lex programs>gcc -o expr expr.tab.c lex.yy.c

C:\6th sem\Compiler Design\lex programs>expr
Enter the expression:
4+5*(3-1)
Result = 14
```

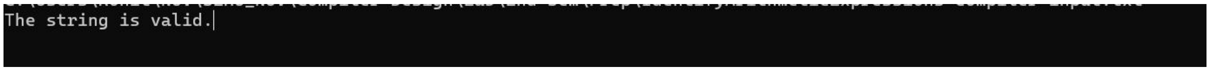
b.Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, * and / .

Input:

```
YACC:
%{
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
void yyerror(char *);
int yylex(void);
extern FILE *yyin;
%}
%union{
   char *str;
   int num;
}
%token <str> ID
%token <num> INT
%left '+' '-'
%left '*' '/'
%%
S:E '\n' {printf("The string is valid.");}
E:E '+' T { }
 | E '-' T { }
 | T { }
T:T '*' F { }
 | T '/' F { }
 | F { }
F:INT { }
 | ID { }
%%
void yyerror(char *s){
   fprintf(stderr,"%s\n",s);
}
int main(int argc, char **argv){
   if(argc<2){
      printf("Usage: %s <input>",argv[0]);
```

```
        exit(1);
    }

    FILE *input = fopen(argv[1],"r");
    if(!input){
        printf("Error Opening File.");
        exit(1);
    }
    yyin=input;
    yyparse();
    fclose(input);
    return 0;
}
LEX:
%{
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
void yyerror(char *);
#include "yacc.tab.h"
%}
%%
[0-9]+ {yylval.num=atoi(yytext); return INT;}
[A-Za-z][A-Za-z0-9_]* {yylval.str=strdup(yytext); return ID;}
[-+*/]  {return *yytext;}
\n  {return '\n';}
[ \t] { }
.   {yyerror("Invalid input.");}
%%
int yywrap(){return 0;}
INPUT.txt:
1+2+3+5+8
```

Output:



```
The string is valid.
```

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

Input:

calc.l code:

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

%%

[0-9]+        { yylval.ival = atoi(yytext); return NUMBER; }
[ \t]+        ; // skip whitespace
\n            { return '\n'; }
.             { return yytext[0]; }

%%

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

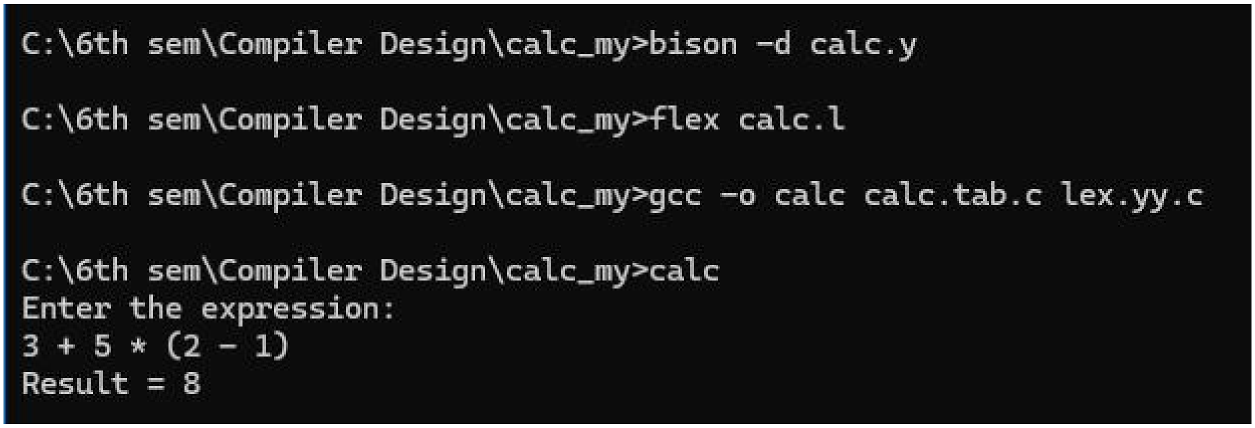
calc.y code:

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

void yyerror(const char *s);
int yylex(void);
```

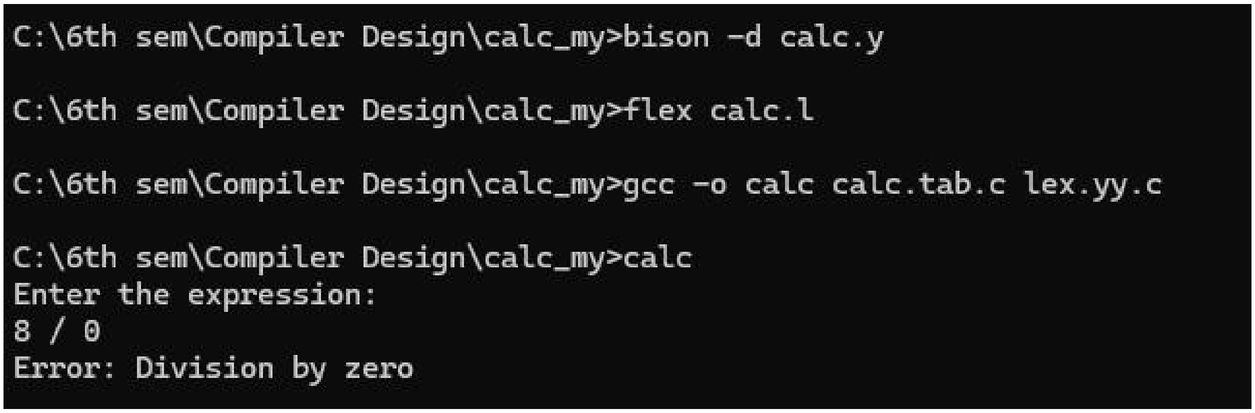
```
%}

%union {
    int ival;
}

%token <ival> NUMBER
%type <ival> expr

%left '+' '-'
%left '*' '/'
%start input

%%

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

expr:
    expr '+' expr   { $$ = $1 + $3; }
  | expr '-' expr   { $$ = $1 - $3; }
  | expr '*' expr   { $$ = $1 * $3; }
  | expr '/' expr   {
                      if ($3 == 0) {
                        yyerror("Division by zero");
                        YYABORT; // Exit the parsing process immediately
                      } else {
                        $$ = $1 / $3;
                      }
```

```
                  }
  | '(' expr ')'    { $$ = $2; }
  | NUMBER          { $$ = $1; }
  ;

%%

int main() {
    printf("Enter the expression:\n");
    return yyparse();
}

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}
```

Output:



```
C:\6th sem\Compiler Design\calc_my>bison -d calc.y

C:\6th sem\Compiler Design\calc_my>flex calc.l

C:\6th sem\Compiler Design\calc_my>gcc -o calc calc.tab.c lex.yy.c

C:\6th sem\Compiler Design\calc_my>calc
Enter the expression:
3 + 5 * (2 - 1)
Result = 8
```



```
C:\6th sem\Compiler Design\calc_my>bison -d calc.y

C:\6th sem\Compiler Design\calc_my>flex calc.l

C:\6th sem\Compiler Design\calc_my>gcc -o calc calc.tab.c lex.yy.c

C:\6th sem\Compiler Design\calc_my>calc
Enter the expression:
8 / 0
Error: Division by zero
```

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

Input:

infix_to_postfix.l code:

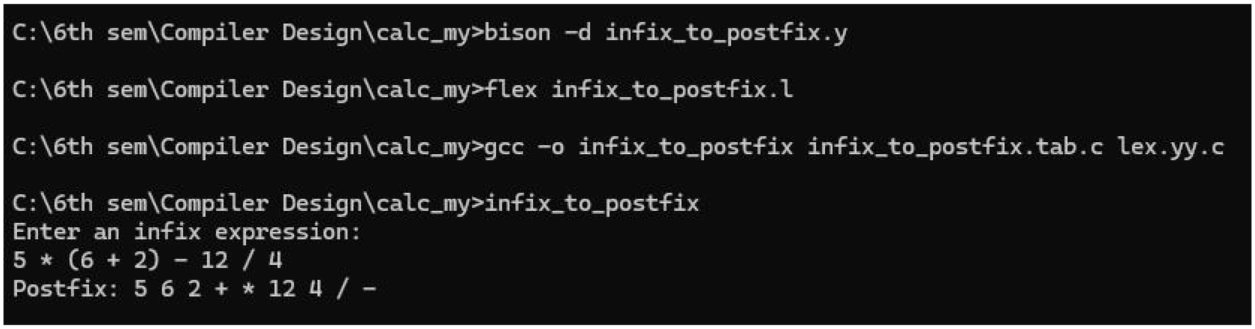
```
%{
#include "infix_to_postfix.tab.h"
#include <stdlib.h>
#include <string.h>
%}

DIGIT   [0-9]
WS      [ \t\r]+

%%

{DIGIT}+    {
              yylval.str = strdup(yytext);
              return NUMBER;
            }
"("         { return '('; }
")"         { return ')'; }
"+"         { return '+'; }
"-"         { return '-'; }
"*"         { return '*'; }
"/"         { return '/'; }
{WS}        { /* skip whitespace */ }
\n          { return '\n'; }
.           { return yytext[0]; }
```

```
%%

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

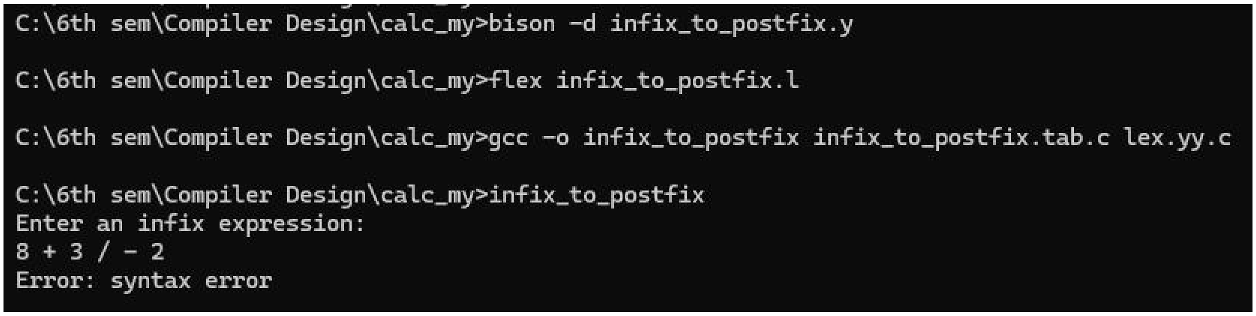
infix_to_postfix.y code:

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

// custom asprintf implementation for Windows
int asprintf(char **strp, const char *fmt, ...) {
    va_list args;
    va_start(args, fmt);
    int size = vsnprintf(NULL, 0, fmt, args);
    va_end(args);

    if (size < 0) return -1;

    *strp = (char *)malloc(size + 1);
    if (!*strp) return -1;

    va_start(args, fmt);
    vsnprintf(*strp, size + 1, fmt, args);
    va_end(args);
    return size;
}

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

%union {
    char *str;
}

%token <str> NUMBER
%left '+' '-'
%left '*' '/'
%token '(' ')'

%type <str> expr

%%

input:
    /* empty */
    | input expr '\n' {
        printf("Postfix: %s\n", $2);
        free($2);
    }
    ;

expr:
      NUMBER              { $$ = strdup($1); free($1); }
    | expr '+' expr       { asprintf(&$$, "%s %s +", $1, $3); free($1); free($3); }
    | expr '-' expr       { asprintf(&$$, "%s %s -", $1, $3); free($1); free($3); }
    | expr '*' expr       { asprintf(&$$, "%s %s *", $1, $3); free($1); free($3); }
    | expr '/' expr       { asprintf(&$$, "%s %s /", $1, $3); free($1); free($3); }
    | '(' expr ')'        { $$ = $2; }
    ;

%%

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}

int main() {
    printf("Enter an infix expression:\n");
    yyparse();
    return 0;
}
```

Output:



```
C:\6th sem\Compiler Design\calc_my>bison -d infix_to_postfix.y

C:\6th sem\Compiler Design\calc_my>flex infix_to_postfix.l

C:\6th sem\Compiler Design\calc_my>gcc -o infix_to_postfix infix_to_postfix.tab.c lex.yy.c

C:\6th sem\Compiler Design\calc_my>infix_to_postfix
Enter an infix expression:
5 * (6 + 2) - 12 / 4
Postfix: 5 6 2 + * 12 4 / -
```



```
C:\6th sem\Compiler Design\calc_my>bison -d infix_to_postfix.y

C:\6th sem\Compiler Design\calc_my>flex infix_to_postfix.l

C:\6th sem\Compiler Design\calc_my>gcc -o infix_to_postfix infix_to_postfix.tab.c lex.yy.c

C:\6th sem\Compiler Design\calc_my>infix_to_postfix
Enter an infix expression:
8 + 3 / - 2
Error: syntax error
```