LAB MANUAL
of
Compiler Design Laboratory (CSE606)

Bachelor of Technology (CSE)

By

Ronit Girglani (22001010 - A2)

Department of Computer Science and Engineering

School Engineering and Technology

Navrachana University, Vadodara

6th Semester

(2025)

List of Experiments

Lab – 1:

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

Code:

```c
#include<stdio.h>

void main()
{
    char input[100];
    int state = 0, i = 0;
    FILE *fp = fopen("input_1_a.txt", "r");
    if (fp == NULL) {
        printf("Error opening file.\n");
        return;
    }
    fgets(input, sizeof(input), fp);
    fclose(fp);
    while(input[i] != '\0') {
        switch (state)
        {
        case 0:
            if (input[i] =='a'){
                state = 1;
            }
            else if (input[i]=='b'){
                state = 2;
            }
            else{
                state = 2;
            }
            break;

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

```
        case 2:

            break;
        }
        i++;
    }
    if (state == 1) {
        printf("input string '%s' is Accepted !\n", input);
    } else {
        printf("Not Accepted\n");
    }
}
```

C:\Users\Ronit\NUV\SEM6_NU   ✕   +   ∨

```
input string 'ab' is Accepted !

--------------------------------
Process exited after 0.08366 seconds with return value 32
Press any key to continue . . .
```
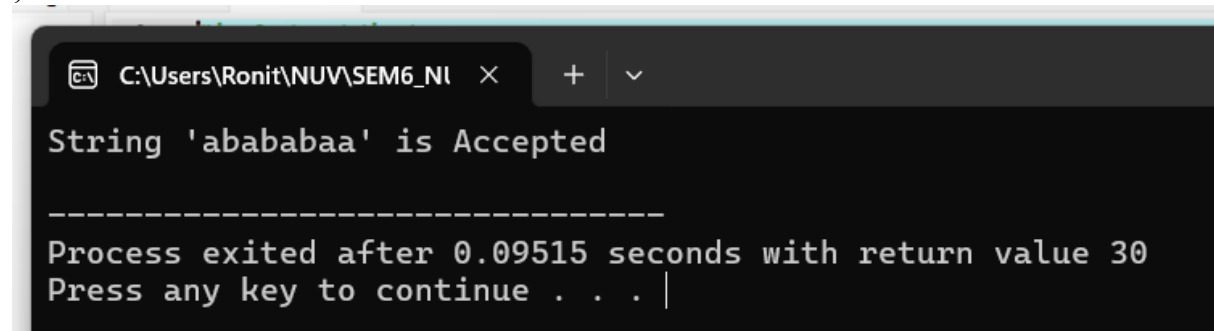
B: Write a program to recognize strings end with 'a'.
Code:

```c
#include<stdio.h>

void main(){
    // program ends with 'a'.
    char input[100];
    int i = 0,state=0;
    FILE *fp = fopen("input_1_b.txt","r");
    if(fp == NULL){
        printf("Error opening file\n");
        return;
    }
    fscanf(fp,"%s",input);
    while(input[i] != '\0'){
        switch(state){
            case 0:
                if(input[i]== 'a'){
                    state = 1;
                }else{
                    state = 0;
                }
                break;
            case 1:
                if(input[i] == 'a'){
                    state = 1;
                }else{
                    state = 0;
                }
                break;
        }
        i++;
    }
    if(state == 1){
        printf("String '%s' is Accepted\n",input);
    }else{
        printf("Not Accepted\n");
    }
}
```

}

String 'abababaa' is Accepted

_____

Process exited after 0.09515 seconds with return value 30
Press any key to continue . . .
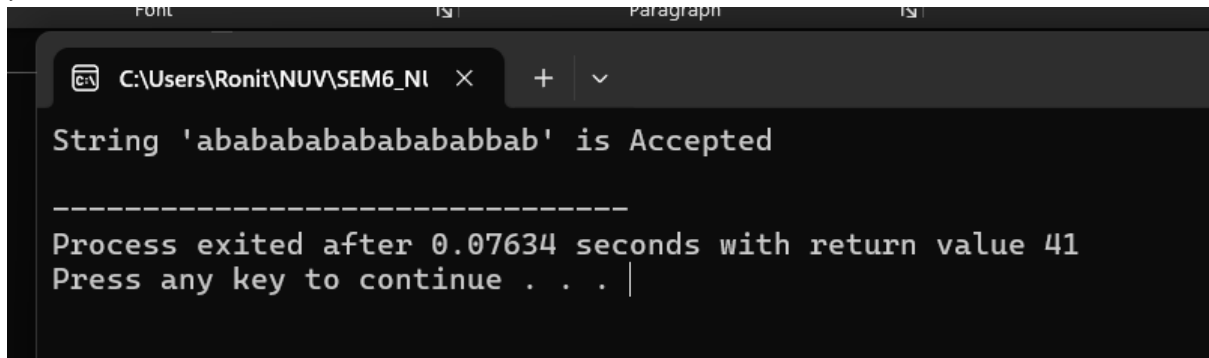
C: Write a program to recognize strings end with 'ab'. Take the input from text file.

Code:

```c
#include<stdio.h>

void main(){
    // program ends with 'ab'.
    char input[100];
    int i = 0,state=0;
    FILE *fp = fopen("input_1_c.txt","r");
    fscanf(fp,"%s",input);
    fclose(fp);
    while(input[i] != '\0'){
        switch(state){
            case 0:
                if(input[i]=='a'){
                    state = 1;
                }
                else{
                    state = 0;
                }
                break;
            case 1:
                if(input[i] == 'b'){
                    state = 2;
                }
                else if(input[i] == 'a'){
                    state = 1;
                }
                else{
                    state = 0;
                }
                break;
            case 2:
                if(input[i] == 'a'){
                    state = 1;
                }
                else{
                    state = 0;
                }
                break;
```

```
    }
    i++;
  }
  if(state == 2){
    printf("String '%s' is Accepted\n",input);
  }else{
    printf("Not Accepted\n");
  }
}
```
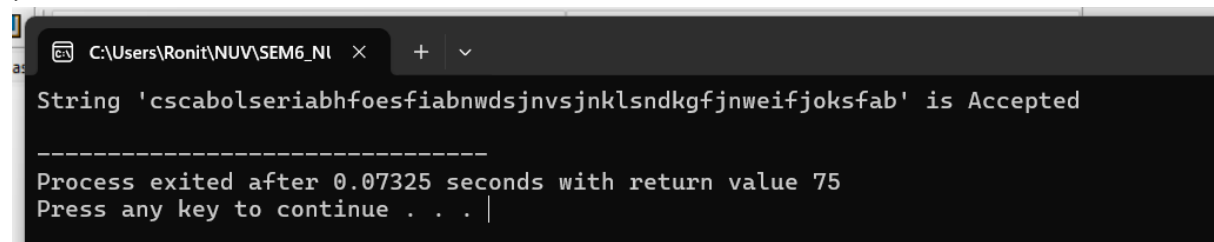
D: Write a program to recognize strings contains 'ab'. Take the input from text file.

Code:

```c
#include<stdio.h>

void main(){
    char input[100];
    int i=0,state=0;
    FILE *fp = fopen("input_1_d.txt","r");
    fscanf(fp,"%s",input);
    fclose(fp);
    while(input[i] != '\0'){
        switch(state){
            case 0:
                if(input[i]=='a'){
                    state=1;
                }
                else{
                    state=0;
                }
                break;
            case 1:
                if(input[i]=='b'){
                    state=2;
                }
                else if(input[i]=='a'){
                    state=1;
                }
                else{
                    state=0;
                }
                break;
            case 2:
                state=2;
                break;

        }
        i++;
    }
    if(state==2){
        printf("String '%s' is Accepted\n",input);
```

```
    }else{
       printf("Not Accepted\n");
    }
}
```

String 'cscabolseriabhfoesfiabnwdsjnvsjnklsndkgfjnweifjoksfab' is Accepted

_____
Process exited after 0.07325 seconds with return value 75
Press any key to continue . . .
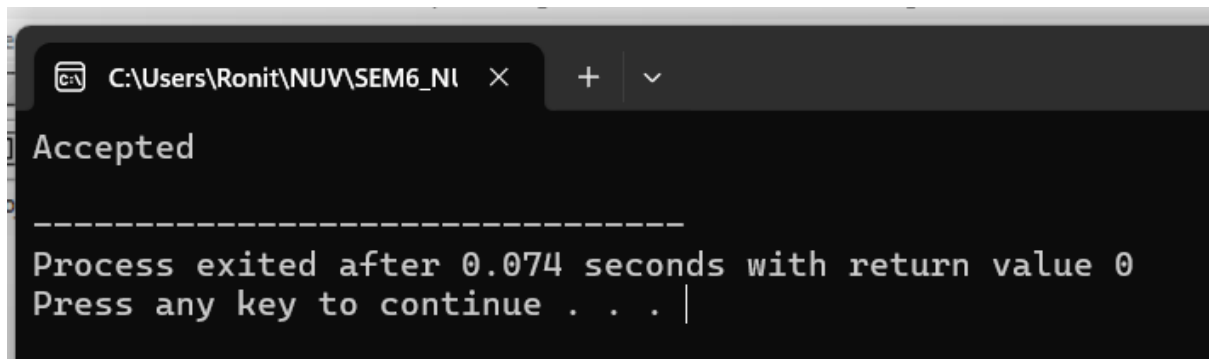
Lab – 2:

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

Code:

```
#include<stdio.h>
#include<conio.h>
// Q2a. Write a program to recognize the valid identifiers.(and keyword but only int.)
int main()
{
    char input[10];
    int state = 0 , i=0;
    printf("Enter Input:");
    scanf("%s",input);
    while (i<=10)
    {
        switch (state)
        {
        case 0:
            if(input[i]=='i'){
                state = 1;
            }
            else if((input[i] >= '0' && input[i] <= '9') || (input[i] >= 'A' && input[i] <= 'Z') ||
(input[i] >= 'a' && input[i] <= 'z') ||( input[i] == '_'))
            {
                state = 5;
            }
            break;
        case 1:
            if(input[i]=='n'){
                state = 2;
            }
            else if((input[i] >= '0' && input[i] <= '9') || (input[i] >= 'A' && input[i] <= 'Z') ||
(input[i] >= 'a' && input[i] <= 'z') ||( input[i] == '_'))
            {
                state = 5;
            }
            break;
        case 2:
            if(input[i]=='t'){
                state = 3;
            }
```

```c
        else if((input[i] >= '0' && input[i] <= '9') || (input[i] >= 'A' && input[i] <= 'Z') ||
(input[i] >= 'a' && input[i] <= 'z') ||( input[i] == '_'))
            {
                state = 5;
            }
            else{
                state = 0;
            }
            break;
        case 3:
            if(input[i]=='\0'){
                state = 0;
            }
            else if((input[i] >= '0' && input[i] <= '9') || (input[i] >= 'A' && input[i] <= 'Z') ||
(input[i] >= 'a' && input[i] <= 'z') ||( input[i] == '_'))
            {
                state = 5;
            }
            break;
        case 5:
            if((input[i] >= '0' && input[i] <= '9') || (input[i] >= 'A' && input[i] <= 'Z') || (input[i]
>= 'a' && input[i] <= 'z') ||( input[i] == '_'))
            {
                state = 5;
            }
            else if(input[i]=='\0'){
                state=0;
            }
            break;
        }
        i++;
    }
    printf("\n");
    if(state == 0){
        printf("Accepted\n");
    }
    else {
        printf("The input is not recognized.(%d) \n",state);}
    return 0;
}
```

B: Write a program to recognize the valid operators.
Code:

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

void main() {
    char c, buffer[1000], lexeme[10];
    int i = 0, j = 0, f = 0, state = 0;
    FILE *fp = fopen("input_operators.txt", "r");

    if (!fp) {
        printf("Error opening file.\n");
        return;
    }

    // Read entire file into buffer
    while ((c = fgetc(fp)) != EOF && j < 1000) {
        buffer[j++] = c;
    }
    buffer[j] = '\0';
    fclose(fp);

    printf("File read.\n");

    i = 0;
    while (buffer[i] != '\0') {
        c = buffer[i];
        switch (state) {
            case 0:
                if (isspace(c)) {
                    // Skip spaces
                } else if (c == '=') {
                    lexeme[f++] = c;
                    state = 1;
                } else if (c == '!') {
                    lexeme[f++] = c;
                    state = 2;
                } else if (c == '<') {
                    lexeme[f++] = c;
                    state = 3;
```

```c
      } else if (c == '>') {
         lexeme[f++] = c;
         state = 4;
      } else if (c == '&') {
         lexeme[f++] = c;
         state = 5;
      } else if (c == '|') {
         lexeme[f++] = c;
         state = 6;
      } else if (c == '+' || c == '-' || c == '*' || c == '/' || c == '%' || c == '^' || c == '~') {
         lexeme[0] = c;
         lexeme[1] = '\0';
         printf("Valid Operator: %s\n", lexeme);
         f = 0;
      } else {
         printf("Invalid character encountered: %c\n", c);
      }
      break;

   case 1: // after '='
      if (c == '=') {
         lexeme[f++] = c;
         lexeme[f] = '\0';
         printf("Valid Relational Operator: %s\n", lexeme); // ==
      } else {
         lexeme[f] = '\0';
         printf("Valid Assignment Operator: %s\n", lexeme); // =
         i--; // reprocess current char
      }
      f = 0;
      state = 0;
      break;

   case 2: // after '!'
      if (c == '=') {
         lexeme[f++] = c;
         lexeme[f] = '\0';
         printf("Valid Relational Operator: %s\n", lexeme); // !=
      } else {
         lexeme[f] = '\0';
         printf("Invalid Operator: %s\n", lexeme); // only ! is invalid here
         i--;
```

```c
        }
        f = 0;
        state = 0;
        break;

    case 3: // after '<'
        if (c == '=') {
            lexeme[f++] = c;
            lexeme[f] = '\0';
            printf("Valid Relational Operator: %s\n", lexeme); // <=
        } else {
            lexeme[f] = '\0';
            printf("Valid Relational Operator: %s\n", lexeme); // <
            i--;
        }
        f = 0;
        state = 0;
        break;

    case 4: // after '>'
        if (c == '=') {
            lexeme[f++] = c;
            lexeme[f] = '\0';
            printf("Valid Relational Operator: %s\n", lexeme); // >=
        } else {
            lexeme[f] = '\0';
            printf("Valid Relational Operator: %s\n", lexeme); // >
            i--;
        }
        f = 0;
        state = 0;
        break;

    case 5: // after '&'
        if (c == '&') {
            lexeme[f++] = c;
            lexeme[f] = '\0';
            printf("Valid Logical Operator: %s\n", lexeme); // &&
        } else {
            lexeme[f] = '\0';
            printf("Valid Bitwise Operator: %s\n", lexeme); // &
            i--;
```

```c
            }
            f = 0;
            state = 0;
            break;

        case 6: // after '|'
            if (c == '|') {
                lexeme[f++] = c;
                lexeme[f] = '\0';
                printf("Valid Logical Operator: %s\n", lexeme); // ||
            } else {
                lexeme[f] = '\0';
                printf("Valid Bitwise Operator: %s\n", lexeme); // |
                i--;
            }
            f = 0;
            state = 0;
            break;

        default:
            printf("Unknown error.\n");
            f = 0;
            state = 0;
            break;
        }
        i++;
    }
}
```

```
File read.
Valid Operator: +
Valid Operator: -
Valid Operator: *
Valid Operator: /
Valid Assignment Operator: =
Valid Relational Operator: ==
Valid Relational Operator: !=
Valid Relational Operator: <
Valid Relational Operator: <=
Valid Relational Operator: >
Valid Relational Operator: >=
Valid Bitwise Operator: &
Valid Logical Operator: &&
Valid Bitwise Operator: |
Valid Logical Operator: ||
Valid Operator: ^
Valid Operator: ~

--------------------------------
```

C: Write a program to recognize the valid number.
Code:

```c
#include<stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
void main(){
        char c,buffer[1000],lexeme[1000];
        int i = 0, state =0, f = 0,j = 0;
        FILE *fp = fopen("input_2_Number.txt","r");
        while((c = fgetc(fp)) != EOF && j < 1000){
    buffer[j++] = c;
  }
  buffer[j] = '\0';
        printf("File read.");
        fclose(fp);
        while(buffer[i] != '\0'){
            c = buffer[i];
            switch(state){
              case 0:
                if(isdigit(c)){
                   state = 1;
                   lexeme[f++] = c;
                }
                else if(c == '+' || c == '-'){
                   state = 0;
                   lexeme[f++] = c;
                }
                else if(isspace(c)){

                }
                else{
                   state = 99;
                }
                break;

              case 1:
                if(isdigit(c)){
                   state = 1;
                   lexeme[f++] = c;
                }
                else if(c == '.'){
```

```
            state = 2;
            lexeme[f++] = c;
        }
        else if(c == 'e' || c == 'E'){
            state = 4;
            lexeme[f++] = c;
        }
        else{
            lexeme[f] = '\0';
            printf("The input %s is a valid integer.\n", lexeme);
            f = 0;
            state = 0;
            i--;
        }
        break;

    case 2:
        if(isdigit(c)){
            state = 3;
            lexeme[f++] = c;
        }
        else{
            lexeme[f] = '\0';
            printf("%s is an invalid floating point input.\n", lexeme);
            f = 0;
            state = 0;
            i--;
        }
        break;

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

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

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

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

    case 6:
        if(isdigit(c)){
            state = 6;
            lexeme[f++] = c;
        }
        else{
```

```
                    lexeme[f] = '\0';
                    printf("The input %s is a valid scientific notation number.\n", lexeme);
                    f = 0;
                    state = 0;
                    i--;
                }
                break;

            default:
                printf("Invalid character encountered: %c\n", c);
                f = 0;
                state = 0;
                break;
        }
        i++;
    }

    // Final Token Check
    if(f != 0){
        lexeme[f] = '\0';
        if(state == 1)
            printf("The input %s is a valid integer.\n", lexeme);
        else if(state == 3)
            printf("The input %s is a valid floating-point number.\n", lexeme);
        else if(state == 6)
            printf("The input %s is a valid scientific notation number.\n", lexeme);
    }
```

```
C:\Users\Ronit\NUV\SEM6_NL  ×    +   ∨

File read.The input 100 is a valid integer.
The input 100.1 is a valid floating-point number.
The input 10.2e10 is a valid scientific notation number.
The input -1000 is a valid integer.
The input 10.2E10 is a valid scientific notation number.
The input 1.2e+10 is a valid scientific notation number.

--------------------------------
Process exited after 0.09651 seconds with return value 57
Press any key to continue . . .
```

D: Write a program to recognize the valid comments.
Code:

```c
#include <stdio.h>
#include <string.h>
// Write a program to recognize the valid comment.(Both single and multi-line)
int main() {
    char str[100];

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

    if (fgets(str, sizeof(str), file) == NULL) {
        printf("Error reading from file.\n");
        return 1;
    }

    int state = 0, i = 0;
    while (i < strlen(str)) {
        switch (state) {
            case 0:
                if (str[i] == '/') {
                    state = 1;
                } else {
                    state = 3;
                }
                break;

            case 1:
                if (str[i] == '/') {
                    state = 2;
                } else if (str[i] == '*') {
                    state = 4;
                } else {
                    state = 3;
                }
                break;

            case 2:
```

```c
            state = 2;
            break;

        case 4:
            if (str[i] == '*') {
                state = 5;
            } else {
                state = 4;
            }
            break;

        case 5:
            if (str[i] == '/') {
                state = 6;
            } else {
                state = 4;
            }
            break;

        case 6:
            state = 6;
            break;
    }
    i++;
}

if (state == 2) {
    printf("Input (%s) is a single-line comment.\n", str);
} else if (state == 6) {
    printf("Input (%s) is a multi-line comment.\n", str);
} else {
    printf("Input (%s) is not a comment.\n", str);
}

fclose(file);
return 0;
```
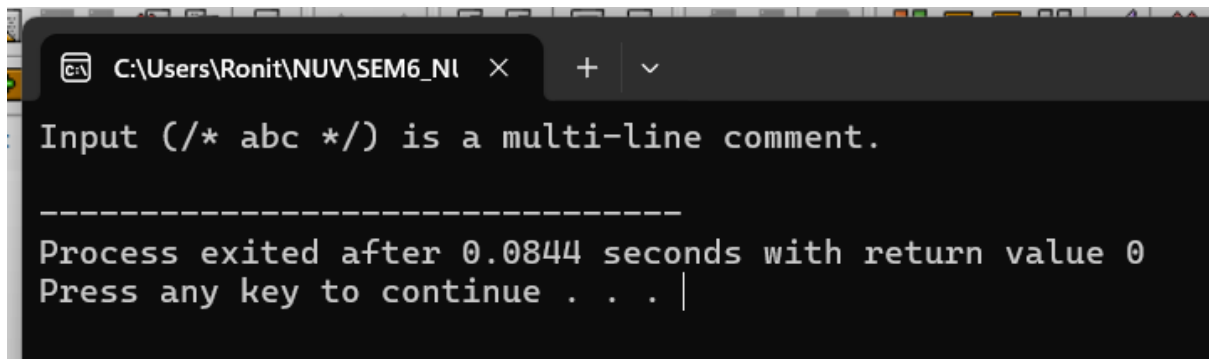
```
Input (/* abc */) is a multi-line comment.

--------------------------------
Process exited after 0.0844 seconds with return value 0
Press any key to continue . . .
```

E: Program to implement Lexical Analyzer.

Code:

```c
#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("input.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];
                printf("DEBUG: Processing lexeme = [%s]\n", lexeme);
        switch (state) {
```

```c
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);
        lexeme[i] = '\0';
        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 {
```

```c
                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();
```

```c
    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;
       }
    }
}
```

```
int is a keyword
main is an identifier
( is a symbol
) is a symbol
{ is a symbol
int is a keyword
a is an identifier

= is an operator
10 is a valid number
; is a symbol
b is an identifier
= is an operator
25 is a valid number
; is a symbol
int is a keyword
d is an identifier
= is an operator
a is an identifier
+ is an operator
b is an identifier
; is a symbol
return is a keyword
0 is a valid number
; is a symbol
} is a symbol


--------------------------------
Process exited after 1.211 seconds with return value 1
Press any key to continue . . .
```

Lab – 3:

Aim: 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.

Lab – 4:

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

Code:

```
%{
#include <stdio.h>
int letters = 0, words = 0, Lines = 0, chars = 0;
int inWord = 0;
%}

%%

[a-zA-Z] {letters++;chars++;if (!inWord) {words++;inWord = 1;}}

[\t \n] {chars++;Lines++;inWord = 0;}

. { chars++; }

%%

int main() {
   yyin = fopen("input.txt", "r");
   if (!yyin) {
      printf("Cannot open the file.\n");
      return 1;
   }
   printf("File has opened.\n");

   yylex();

   printf("This file has %d letters.\n", letters);
   printf("This file has %d words.\n", words);
   printf("This file has %d lines.\n", Lines + 1);
   printf("This file has %d characters.\n", chars);

   return 0;
}

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

```
C:\Users\Ronit\NUV\SEM6_NUV\Compiler Design\Practice>Lab-4_1 input.txt
File has opened.
This file has 61 letters.
This file has 7 words.
This file has 13 lines.
This file has 95 characters.
```

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

Code:

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

```
C:\Users\Ronit\NUV\SEM6_NUV\Compiler Design\Practice>Lab-4_2 input.txt
file opened.




This file has 48 consonants.
This file has 5 vowels.
```

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

Code:

```
%{
#include<stdio.h>
char i=0;
%}
%%
[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)? {i++;printf("%s",yytext);}
.   {i++;}
%%
void main(){
    yyin = fopen("input.txt","r");
    yylex();

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

```
C:\Users\Ronit\NUV\SEM6_NUV\Compiler Design\Practice>Lab-4_3 input.txt


1
1.1
10
```

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

Code:

```
%{
#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;
}
```
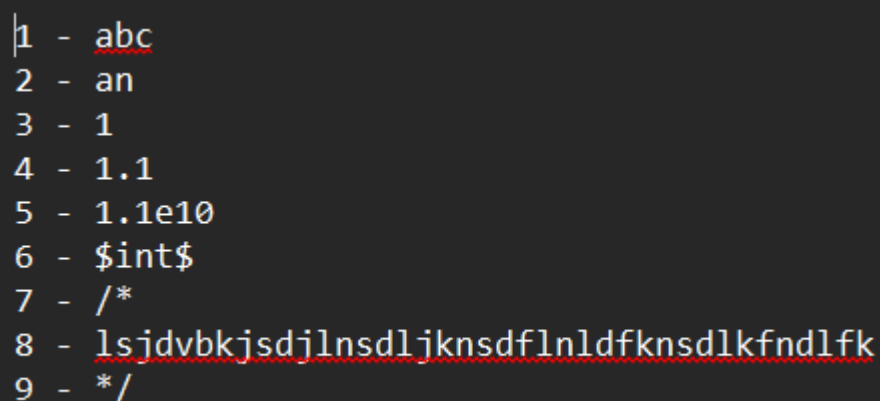
OUTPUT:

```
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.

Code:

```
%{
#include <stdio.h>
%}
%%
"<!--"(.|\n)*"-->"     { printf("Comment: %s\n", yytext); }
"<"[^>]*">"         { printf("Tag: %s\n", yytext); }
.                 ;
%%
void main() {
   yyin=fopen("Test.txt","r");
   printf("File has opened.");
   yylex();
}

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

OUTPUT:

```
C:\Users\Ronit\NUV\SEM6_NUV\Compiler Design\Practice>Lab-4_5
File has opened.Tag: <html>
Tag: </html>

Comment: <!-- hello -->
```

Lab – 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.

Code:

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

int comment_lines = 0;
%}

%%
"/*"[^*/]*"*/"   {fprintf(yyout, "");comment_lines++;}
"//".*           {fprintf(yyout, "");comment_lines++; }
.*           { fprintf(yyout, "%s",yytext); }
%%

void main() {
    yyin  = fopen("input.txt", "r");
    yyout = fopen("output.txt", "w");
    yylex();
    printf("Number of comment lines removed: %d\n", comment_lines);
    fclose(yyin);
    fclose(yyout);
}
int yywrap() {
    return 1;
}
```

OUTPUT:

```
C:\Users\Ronit\NUV\SEM6_NUV\Compiler Design\Practice>Lab-5_1 input.txt
Number of comment lines removed: 2
```

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

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

```
"#include"[ \t]*["<"]?[a-zA-Z0-9\.]+[">"]?      {printf("include statement: %s \n",yytext);}
auto|int|main|if|else|float|char|printf|return   {printf("Keyword: %s \n",yytext);}
[a-zA-Z_]([0-9|a-zA-Z])*   {printf("Identifier: %s \n ",yytext);}
"+"|"-"|"++"|"--"    {printf("Operator: %s \n",yytext);}
[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?    {printf("Number: %s \n",yytext);}
";"|"."|"["|"]"|"{"|"}"|"("|")"     {printf("Special Symbol: %s \n",yytext);}
\"[^"\n]*\"    {printf("String Literals: %s \n",yytext);}
\'[^'\n]*\'    {printf("Char Literals: %s \n",yytext);}
.   {printf("");}
%%
int main() {
   yyin = fopen("input.c", "r");

   yylex();

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

OUTPUT:

```
C:\Users\Ronit\NUV\SEM6_NUV\Compiler Design\Practice>a.exe input.txt
include statement: #include <stdio.h>

Keyword: int
Keyword: main
Special Symbol: (
Special Symbol: )
Special Symbol: {

Keyword: int
Identifier: x
 Number: 10
Special Symbol: ;

Keyword: float
Identifier: y
 Number: 3.14e-2
Special Symbol: ;

Keyword: char
Identifier: c
 Char Literals: 'a'
Special Symbol: ;

Keyword: printf
Special Symbol: (
String Literals: "Hello World"
Special Symbol: )
Special Symbol: ;

Keyword: if
Special Symbol: (
Identifier: x
 Number: 0
Special Symbol: )
Special Symbol: {

Identifier: x
 Operator: ++
Special Symbol: ;

Identifier: x
 Operator: --
Special Symbol: ;

Special Symbol: }

Keyword: return
Number: 0
Special Symbol: ;

Special Symbol: }
```

Lab – 6:

Aim: Program to implement Recursive Descent Parsing in C.

Code:

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

char s[20];
int i = 1;
char l;
int match(char l);
int E1();
int E()
{
    if (l == 'i')
    {
        match('i');
        E1();
    }
    else
    {
        printf("Error parsing string");
        exit(1);
    }
    return 0;
}

int E1()
{
    if (l == '+')
    {
        match('+');
        match('i');
        E1();
    }
    else
    {
        return 0;
    }
}

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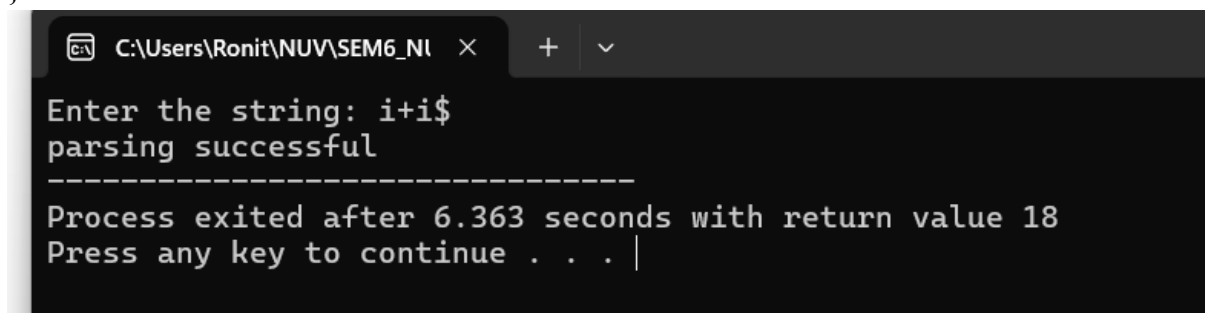
```c
    if (l == t)
    {
        l = s[i];
        i++;
    }
    else
    {
        printf("Syntax Error");
        exit(1);
    }
    return 0;
}
void main()
{
    printf("Enter the string: ");
    scanf("%s", &s);
    l = s[0];
    E();
    if (l == '$')
    {
        printf("parsing successful");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

```
C:\Users\Ronit\NUV\SEM6_NU   ✕    +   ⌄

Enter the string: i+i$
parsing successful
--------------------------------
Process exited after 6.363 seconds with return value 18
Press any key to continue . . . |
```

Lab – 7:
A: To Study about Yet Another Compiler-Compiler(YACC).
Code:

Introduction:

YACC (Yet Another Compiler-Compiler) is a parser generator developed to automate the process of creating the syntax analysis phase of a compiler. It reads a grammar specification written in a format similar to Backus-Naur Form (BNF) and generates a parser in the C programming language. This parser can recognize the syntax of a programming language and build syntax trees or perform semantic actions.

YACC is typically used in combination with **LEX/Flex**, where LEX handles lexical analysis and YACC handles syntax analysis. Together, they allow for the development of efficient and structured compilers. The grammar rules in YACC are written in the form of production rules, and associated C code (semantic actions) can be embedded within these rules to define what happens when a rule is recognized.

YACC Features:

- Supports LALR(1) parsing.
- Allows easy embedding of semantic actions using C code.
- Automatically handles parsing tables and parser generation.
- Provides error-handling mechanisms for syntax errors.
- Easily integrates with Flex for complete compiler front-end development.

B: Create YACC and Lex specification files to recognizes arithmetic expressions involving +, -, * and / .

Code:

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:

```
C:\Users\Ronit\NUV\SEM6_NUV\Compiler Design\Lab\End Sem\Prep\IdentifyArithmeticExpressions>compiler input.txt
The string is valid.
```

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

Code:

YACC:

```
%{
    #include <stdio.h>
    int yylex(void);
    void yyerror(char *);
    extern FILE *yyin;
%}
%token NUM
%%
S:E {printf("%d\n",$1); return 0;}
E:E'+'T  {$$ = $1 + $3;}
 | E'-'T {$$ = $1 - $3;}
 | T    {$$ = $1}
T: T '*' F {$$ = $1 * $3;}
 | T '/' F {$$ = $1 / $3;}
 | F {$$ = $1}
F: NUM {$$ = $1}
%%
void yyerror(char* s){
    fprintf(stderr,"%s\n",s);
}
int main( int argc,char *argv[] ) {
    if (argc < 1) {
        fprintf(stderr, "Usage: %s <input_file>\n", argv[0]);
        return 1;
    }

    FILE *input_file = fopen(argv[1], "r");
    if (!input_file) {
        perror("Error opening file");
        return 1;
    }
    yyin = input_file;
    yyparse();
    fclose(input_file);
    return 0;
}
```

LEX:

```
%{
```

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

INPUT.txt:

5*8+9*6+8

OUTPUT:

```
C:\Users\Ronit\NUV\SEM6_NUV\Compiler Design\Lab\End Sem\Prep\Calc>compiler input.txt
102
```

D: Create YACC and Lex specification files are used to convert infix expression to postfix expression.

Code:

YACC:

```
%{
#include<stdio.h>
#include<stdlib.h>
int yylex(void);
void yyerror(char *);
extern FILE *yyin;
%}
%union{
    char *str;
    int num;
}
%token <str> ID
%token <num> INT
%left '+' '-'
%left '*' '/'
%%
S:E '\n' {return 0;}
 | E {return 0;}
E:E '+' T {printf("+ ");}
 | E '-' T {printf("- ");}
 | T { }
T: T '*' F {printf("* ");}
 | T '/' F {printf("/ ");}
 | F { }
F:INT {printf("%d ",$1);}
| ID  {printf("%s ",$1);}
%%
void yyerror(char *s){
    fprintf(stderr,"%s\n",s);
}
int main(int argc, char *argv[]){
    if(argc<1){
        printf("Usage: %s <input>",argv[0]);
    }
    FILE *input = fopen(argv[1], "r");
    if(!input){
        printf("Error Opening File.\n");
        return 1;
```

```
    }
    yyin=input;
    yyparse();
    fclose(input);
    return 0;
}
```

LEX:

```
%{
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include "parser.tab.h"
void yyerror(char *);
%}
%%
[0-9]+  {yylval.num=atoi(yytext);return INT;}
[A-Za-z_]+ {yylval.str=strdup(yytext);return ID;}
[-+/*]   {return *yytext;}
\n  {return '\n';}
<<EOF>>    { return 0; }
[ \t]   { }
.  {yyerror("invalid input.");}
%%
int yywrap(){
    return 0;
}
```

INPUT.txt:

a+1+2

OUTPUT:

```
C:\Users\Ronit\NUV\SEM6_NUV\Compiler Design\Lab\End Sem\Prep\InfixPostfix>compiler input.txt
a 1 + 2 +
```