SCHOOL OF ENGINEERING & TECHNOLOGY

BACHELOR OF TECHNOLOGY

COMPILER DESIGN

6TH SEMESTER

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Laboratory Manual

By,

Name: Yuvraj Ahuja

Enrollment Id: 22000738

Batch: B1

TABLE OF CONTENT

Practical-1

Aim:

a) Write a program to recognize strings starts with 'a' over {a, b}.
b) Write a program to recognize strings end with 'a'.
c) Write a program to recognize strings end with 'ab'. Take the input from text file.
d) Write a program to recognize strings contains 'ab'. Take the input from text file.

Code:

a)

```
#include <stdio.h>
int main()
{
        printf("This is the first lab of compiler design..>>");
        int i=0,state=0;
        char input[10];
        printf("\nEnter the input string: ");
        scanf("%s",input);
        while(input[i]!='\0'){
                switch(state){
                case 0:
                        if(input[i]=='a'){
                                state=1;
                        }
                        else if(input[i]=='b') state=2;
                        else state=3;
                        break;
                case 1:
                        if(input[i]=='a'||input[i]=='b') state=1;
                        else state=3;
                        break;
                case 2:
                        if(input[i]=='a'||input[i]=='b') state=2;
                        else state=3;
                        break;
                case 3:
                        state=3;
                }
                i++;
        }
if(state==1) printf("Input string is valid.");
else if(state==2 || state==0) printf("Input string is not valid.");
else if(state==3) printf("Input string is not recognized");
return 0;
}
```

b)

```
#include <stdio.h>
int main()
{
        int i=0,state=0;
        char input[10];
        printf("\nEnter the input string: ");
        scanf("%s",input);
        while(input[i]!='\0'){
                switch(state){
                        case 0:
                                if(input[i]=='a'){
                                        state=1;
                                        }
                                else state=0;
                                break;
                        case 1:
                                if(input[i]=='a') state=1;
                                else state=0;
                                break;
                }
                i++;
        }
        if(state==1) printf("Input string is valid.");
        else if(state==0) printf("Input string is not valid.");
        return 0;
}
```

c)

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

#define MAX_LEN 100
typedef enum { START, A, AB } State;
int ends_with_ab(const char *str) {
  State state = START;
  char c;
```

```
    for (int i = 0; str[i] != '\0'; i++) {
        c = str[i];
        switch (state) {
            case START:
                if (c == 'a') state = A;
                else state = START;
                break;
            case A:
                if (c == 'b') state = AB;
                else if (c == 'a') state = A;
                else state = START;
                break;
            case AB:
                if (c == 'a') state = A;
                else state = START;
                break;
        }
    }
    return state == AB;
}

int main() {
    FILE *file = fopen("input.txt", "r");
    char line[MAX_LEN];
    if (!file) {
        printf("Error opening file.\n");
        return 1;
    }
    printf("Strings that end with 'ab':\n");
    while (fgets(line, sizeof(line), file)) {
        line[strcspn(line, "\n")] = 0;
        if (ends_with_ab(line)) {
            printf("%s\n", line);
        }
    }
    fclose(file);
    return 0;
}
```

d)

```
#include <stdio.h>
#include <string.h>
#define MAX_LEN 100
```

```
typedef enum { START, A, ACCEPT } State;
int contains_ab(const char *str) {
    State state = START;
    char c;
    for (int i = 0; str[i] != '\0'; i++) {
        c = str[i];
        switch (state) {
            case START:
                if (c == 'a') state = A;
                else state = START;
                break;
            case A:
                if (c == 'b') return 1; // Accept immediately
                else if (c == 'a') state = A;
                else state = START;
                break;
        }
    }
    return 0;
}
int main() {
    FILE *file = fopen("input.txt", "r");
    char line[MAX_LEN];

    if (!file) {
        printf("Error opening file.\n");
        return 1;
    }
    printf("Strings that contain 'ab':\n");

    while (fgets(line, sizeof(line), file)) {
        line[strcspn(line, "\n")] = 0;
        if (contains_ab(line)) {
            printf("%s\n", line);
        }
    }
    fclose(file);
    return 0;
}
```

Practical-2

Aim:

a) Write a program to recognize the valid identifiers and keywords.
b) Write a program to recognize the valid operators.
c) Write a program to recognize the valid number.
d) Write a program to recognize the valid comments.
e) Program to implement Lexical Analyzer.

Code:

a)

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

int main(){

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

        while(input[i]!='\0'){
                switch(state){
                        case 0:
                                if(input[i]=='i') state=1;
                                else if(isalpha(input[i])) {
                                state=4; }
                                break;
                        case 1:
                                if(input[i]=='n') state=2;
                                else state=3;
                                break;
                        case 2:
                                if(input[i]=='t') state=3;
                                else state=4;
                                break;
                        case 3:
                                if(input[i]=='\0') state=5;
                                else state=4;
                                break;
                        case 4:
                                if(input[i]=='\0' || input[i]=='_' || isdigit(input[i]) ) state=6;
```

```
                    break;
            }
            i++;
        }

        if(state==3) {
        printf("The string is Keyword");}
        else if(state==4) printf("The string is valid identifier");
        else printf("The string is neither a keyword and also not a valid identifier");


        return 0;
    }
```

b) Njdkv

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

int main()
{
   char string[1000];
   printf("Enter the string: ");
   scanf("%s", string);  // Prevents buffer overflow
   int i = 0, state = 0;
   int num=1;
   while (num>0)
   {
      switch(state)
      {
         case 0:
            if (string[i] == '+') state = 51;
            else if(string[i] == '*') state = 51;
            else if(string[i] == '/') state = 60;
            else if(string[i] == '=') state = 53;
            else if(string[i] == '-') state = 54;
            else if(string[i] == '?') state = 55;
            else if(string[i] == '<') state = 56;
            else if(string[i] == '>') state = 59;
            else if(string[i] == '!') state = 57;
            else if(string[i] == '&') state = 58;
            else if(string[i] == '|') state = 61;
            else if(string[i] == '~' || string[i] == '^') state = 62;
            break;
```

```
case 51: // Arithmetic
    if (string[i] == '\0') state = 51;
    else if (string[i] == '+') state = 52;
    else if (string[i] == '=') state = 53;
    else state = 0;
    num--;
    break;

case 52: // Unary
    break;

case 53: // Assignment
    if(string[i] == '=') state = 56;
    else state = 0;
    num--;
    break;

case 54: // -
    if (string[i] == '\0') state = 51;
    else if (string[i] == '-') state = 52;
    else if (string[i] == '=') state = 53;
    else state = 0;
    num--;
    break;

case 55: // Ternary
    if (string[i] == ':') state = 55;
    else if(string[i] == '\0') state = 0;
    else state = 0;
    num--;
    break;

case 56: //Relational
    if (string[i] == '=') state = 56;
    else if(string[i] == '<') state = 58;
    else state = 0;
    num--;
    break;

case 59: // >
    if (string[i] == '>') state = 58;
```

```
            else if (string[i] == '=') state = 56;
            else state = 0;
            num--;
            break;

        case 57: // Logical
            if (string[i] == '=') state = 56;
            else if(string[i] == '\0') state = 57;
            else state = 0;
            num--;
            break;

        case 58: // Bitwise
            if (string[i] == '&') state = 57;
            else if(string[i] == '\0') state = 58;
            else state = 0;
            num--;
            break;

        case 60: // "/"
            if (string[i] == '\0') state = 51;
            else state = 0;
            num--;
            break;

        case 61: // "|"
            if (string[i] == '|') state = 57;
            else if(string[i] == '\0') state = 58;
            else state = 0;
            num--;
            break;

        case 62: // ~ ^
            if (string[i] == '\0') state = 58;
            else state = 0;
            num--;
            break;
    }
    i++;
}

if (state == 51) printf("%s is an Arithmatic operator.", string);
```

```
    else if(state == 52) printf("%s is an Unary operator.", string);
    else if(state == 53) printf("%s is an Assignment operator.", string);
    else if(state == 55) printf("%s is ternary or conditional operator.", string);
    else if(state == 56) printf("%s is a Relational Operator.", string);
    else if(state == 57) printf("%s is a Logical operator.", string);
    else if(state == 58) printf("%s is a Bitwise operator.", string);
    else
        printf("Processing....");

    return 0;
}
```

c)

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

int main() {
    FILE *file;
    char buffer[100];
    char lexeme[100];
    char c;
    int f, i, state;

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

    while (fgets(buffer, 100, file)) {
        buffer[strcspn(buffer, "\n")] = 0; // Remove newline character
        f = 0;
        i = 0;
        state = 0;

        while (buffer[f] != '\0') // Loop through each character in the buffer
        {
            switch (state) {
                case 0:
                    c = buffer[f];
                    if (isdigit(c)) { state = 40; lexeme[i++] = c; }
    else { state = 0; }
    break;

case 40:
    c = buffer[f];
    if (isdigit(c)) { state = 40; lexeme[i++] = c; }
    else if (c == '.') { state = 41; lexeme[i++] = c; }
    else if (c == 'E' || c == 'e') { state = 43; lexeme[i++] = c; }
    else {
        lexeme[i] = '\0';
        printf("%s is a valid number\n", lexeme);
        i = 0; // Reset lexeme index for the next number
        state = 0; // Reset state for the next number
        f--;
    }
    break;

case 41:
    c = buffer[f];
    if (isdigit(c)) { state = 42; lexeme[i++] = c; }
    else {
        lexeme[i] = '\0';
        printf("%s is an invalid number (expected digit after decimal)\n", lexeme);
        i = 0; // Reset lexeme index for the next number
        state = 0; // Reset state for the next number
        f--;
    }
    break;

case 42:
    c = buffer[f];
    if (isdigit(c)) { state = 42; lexeme[i++] = c; }
    else if (c == 'E' || c == 'e') { state = 43; lexeme[i++] = c; }
    else {
        lexeme[i] = '\0';
        printf("%s is a valid number\n", lexeme);
        i = 0; // Reset lexeme index for the next number
        state = 0; // Reset state for the next number
        f--;
    }
    break;
```

```
            case 43:
              c = buffer[f];
              if (c == '+' || c == '-') { state = 44; lexeme[i++] = c; }
              else if (isdigit(c)) { state = 45; lexeme[i++] = c; }
              else {
                lexeme[i] = '\0';
                printf("%s is an invalid number (expected digit or sign after 'E'/'e')\n",
lexeme);
                i = 0; // Reset lexeme index for the next number
                state = 0; // Reset state for the next number
                f--;
              }
              break;

            case 44:
              c = buffer[f];
              if (isdigit(c)) { state = 45; lexeme[i++] = c; }
              else {
                lexeme[i] = '\0';
                printf("%s is an invalid number (expected digit after sign in exponent)\n",
lexeme);
                i = 0; // Reset lexeme index for the next number
                state = 0; // Reset state for the next number
                f--;
              }
              break;

            case 45:
              c = buffer[f];
              if (isdigit(c)) { state = 45; lexeme[i++] = c; }
              else {
                lexeme[i] = '\0';
                printf("%s is a valid number\n", lexeme);
                i = 0; // Reset lexeme index for the next number
                state = 0; // Reset state for the next number
                f--;
              }
              break;
          }
          f++;
        }
        if (state == 40 || state == 41 || state == 42 || state == 45) {
            lexeme[i] = '\0';
            printf("%s is a valid number\n", lexeme);
        } else {
            printf("%s is an invalid number\n", buffer);
        }
    }

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

d)

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

int main()
{
    char string[1000];
    printf("Enter the string: ");
    scanf("%s", string);

    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 (int i = 0; i < 32; i++)
    {
        if (strcmp(string, keywords[i]) == 0)
        {
            printf("%s is a keyword\n", string);
        }
    }
        printf("%s is an identifier\n", string);

return 0;
}
```

e)

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#define BUFFER_SIZE 1000
void check(char *lexeme);

int main() {
  FILE *f1;
  char buffer[BUFFER_SIZE], lexeme[50]; // Static buffer for input and lexeme storage
  char c;
  int f = 0, state = 0, i = 0;
  f1 = fopen("Input.txt", "r");
  fread(buffer, sizeof(char), BUFFER_SIZE - 1, f1);
  buffer[BUFFER_SIZE - 1] = '\0'; // Null termination
  fclose(f1);

  while (buffer[f] != '\0') {
    c = buffer[f];
    switch (state) {
      case 0:
        if (isalpha(c) || c == '_') {
          state = 1;
          lexeme[i++] = c;
        }
        else if (c == ' ' || c == '\t' || c == '\n') {
          state = 0;
        }
        else if(isdigit(c)) {
          state = 13;
          lexeme[i++] = c;
        }
        else if (c == '/') {
          state = 11;   // For comment
        }
        else if (c == ';' || c == ',' || c == '{' || c == '}') {
          printf(" %c is a symbol\n", c);
          state = 0;
        }
        else if (strchr("+-*/=%?<>!&|~^", c)) {
          state = 50;
          lexeme[i++] = c;
        }
        else {
          state = 0;
        }
        break;

      case 1:
        if (isalpha(c) || isdigit(c) || c == '_') {
          state = 1;
          lexeme[i++] = c;
        } else {
          lexeme[i] = '\0'; // Null-terminate the lexeme
          check(lexeme);  // Check if it's a keyword or identifier
          state = 0;
          i = 0;
          f--; // Step back to reprocess the current non-alphanumeric character
        }
        break;

      case 13:
        if(isdigit(c)) {
          state = 13;
          lexeme[i++] = c;
        }
        else if(c=='.') {
          state=14;
          lexeme[i++]=c;
        }
        else if(c=='E'||c=='e') {
          state=16;
          lexeme[i++]=c;
        }
        else {
          lexeme[i]='\0';
          printf("%s is a valid number\n", lexeme);
          i=0;
          state=0;
          f--;
        }
        break;

      case 50: // Operator Handling
```

```
switch (lexeme[0]) {
  case '+':
    if (c == '+') {
      printf("%s is a Unary operator\n", lexeme);
      state = 0;
    }
    else if (c == '=') {
      printf("%s is an Assignment operator\n", lexeme);
      state = 0;
    }
    else {
      printf("%s is an Arithmetic operator\n", lexeme);
      state = 0;
      f--;
    }
    break;

  case '-':
    if (c == '-') {
      printf("%s is a Unary operator\n", lexeme);
      state = 0;
    }
    else if (c == '=') {
      printf("%s is an Assignment operator\n", lexeme);
      state = 0;
    }
    else {
      printf("%s is an Arithmetic operator\n", lexeme);
      state = 0;
      f--;
    }
    break;

  case '*':
  case '/':
  case '%':
    if (c == '=') {
      printf("%s is an Assignment operator\n", lexeme);
      state = 0;
    }
    else {
      printf("%s is an Arithmetic operator\n", lexeme);
```

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

case '=':
    if (c == '=') {
        printf("%s is a Relational operator\n", lexeme);
        state = 0;
    }
    else {
        printf("%s is an Assignment operator\n", lexeme);
        state = 0;
        f--;
    }
    break;

case '<':
case '>':
    if (c == '=' || c == lexeme[0]) {
        printf("%s is a Relational operator\n", lexeme);
        state = 0;
    }
    else {
        printf("%s is a Relational operator\n", lexeme);
        state = 0;
        f--;
    }
    break;

case '!':
case '&':
case '|':
    if (c == '=') {
        printf("%s is a Logical operator\n", lexeme);
        state = 0;
    }
    else if (c == lexeme[0]) {
        printf("%s is a Logical operator\n", lexeme);
        state = 0;
    }
    else {
```

```
                printf("%s is a Logical operator\n", lexeme);
                state = 0;
                f--;
            }
            break;

        case '~':
        case '^':
            printf("%s is a Bitwise operator\n", lexeme);
            state = 0;
            f--;
            break;

        case '?':
            if (c == ':') {
                printf("%s is a Ternary or conditional operator\n", lexeme);
                state = 0;
            }
            else {
                state = 0;
                f--;
            }
            break;

        default:
            state = 0;
            break;
    }
    lexeme[0] = '\0';
    i = 0;
    break;

  default:
    state = 0;
    break;
  }
  f++;
 }
}

void check(char *lexeme) {
 char *keywords[] = {
    "auto", "break", "case", "char", "const", "continue", "default", "do",
    "double", "else", "ef", "extern", "float", "for", "goto", "if",
    "inline", "int", "long", "register", "restrict", "return", "short", "signed",
    "sizeof", "static", "struct", "switch", "typedef", "union", "unsigned", "void", "volatile",
"while"
};
    for (int 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);
}
```

Practical-3

Aim:

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

Code:

LEX and Flex are tools used to generate lexical analyzers, which are essential components of a compiler. A lexical analyzer, or scanner, reads the input source code and converts it into a sequence of tokens, which are then passed to the parser for syntax analysis. LEX (Lexical Analyzer Generator) is the original tool developed for this purpose, while Flex (Fast Lexical Analyzer) is an enhanced, faster version compatible with LEX. These tools allow users to define regular expressions and actions in a specification file, which is then compiled into C code. This C code can identify and process patterns in input text efficiently, making LEX and Flex valuable in building compilers, interpreters, and text-processing tools.

Practical-4

Aim:

Implement following programs using Lex.

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

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

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

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

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

Code:

a)

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

b)

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

```
[aeiouAEIOU] vowels++;
[a-zA-z] consonants++;
. ;
%%
void main(){
yyin=fopen("input.txt","r");
yylex();
printf("This file is containing \n%d vowels , \n%d consonants",vowels,consonants);
}
int yywrap(){ return(1);}
```

c)

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

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

int main(){
yyin=fopen("inputfile.txt","r");
yylex();
return 0;
}
int yywrap(){ return(1);}
```

d)

```
%{
int line_number= 1;
%}

%%
.+ {fprintf(yyout,"%d: %s",line_number,yytext);line_number++;}
%%

int main(){
yyin=fopen("inputfile.txt","r");
yyout = fopen("op.txt","w");
yylex();
printf("Done");
return 0;
}
int yywrap(){ return(1);}
```

e)

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

%%
"<"[A-Za-z0-9]+">" printf("%s is valid markup tag\n",yytext);
"<!--"(.|\n)*?--> {printf("%s is a comment",yytext); comment_count++;}
\n;
. ;
%%

int main(){
yyin=fopen("inputfile1.txt","r");
yylex();
printf("There are total %d  html comments in the file",comment_count);
fclose(yyin);
return 0;
}
int yywrap(){ return(1);}
```

Practical-5

Aim:

a. Write a Lex program to count the number of C comment lines from a given C program. Also eliminate them and copy that program into separate file.

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

Code:

a)

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

%%
"//".* {printf("%s is a single line comment\n",yytext); comment_count++;
fprintf(yyout,"%s","");}
"/*"[^*/]*?"*/" {printf("%s is a multiline comment",yytext); comment_count++;
fprintf(yyout,"%s","");}
.+ {fprintf(yyout,"%s",yytext);};
%%

int main(){
yyin=fopen("inputfile2.txt","r");
yyout = fopen("op1.txt","w");
yylex();
printf("There are total %d comments in the code",comment_count);
fclose(yyin);
return 0;
}
int yywrap(){ return(1);}
```

b)

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

int keyword_count = 0;
int identifier_count = 0;
int operator_count = 0;
int number_count = 0;
int special_symbol_count = 0;
int literal_count = 0;

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

%%

"int"|"float"|"char"|"double"|"if"|"else"|"for"|"while"|"return"|"void"|"break"|"cont
inue"|"switch"|"case"|"default"|"struct"|"union"|"typedef"|"long"|"short"|"goto"|"si
zeof"|"volatile"|"const"|"static"|"extern"|"register"|"signed"|"unsigned" {
    printf("%s is a keyword\n", yytext);
    keyword_count++;
}


[A-Za-z_][A-Za-z0-9_]* {
    printf("%s is an identifier\n", yytext);
    identifier_count++;
}

"+"|"-"|"*"|"/"|"%"|"="|"=="|"<"|"<="|">"|">="|"!"|"!="|"&"|"|"|"&&"|"||"|"++"|"-
-" {
    printf("%s is an operator\n", yytext);
    operator_count++;
}


[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)? {
    printf("%s is a number\n", yytext);
    number_count++;
}


"%"|"&"|"!"|"(" | ")" | "{" | "}" | "[" | "]" | ";" | ","  {
    printf("%s is a special symbol\n", yytext);
    special_symbol_count++;
```

```
}


\"([^\n\"]|\\\")*\" {
    printf("%s is a string literal\n", yytext);
    literal_count++;
}


\'[^\']\' {
    printf("%s is a character literal\n", yytext);
    literal_count++;
}

[ \t\n]+

.

%%

int main() {
    yyin = fopen("inputfile3.txt","r");
    yylex();

    // Output the counts
    printf("\nSummary:\n");
    printf("Keywords: %d\n", keyword_count);
    printf("Identifiers: %d\n", identifier_count);
    printf("Operators: %d\n", operator_count);
    printf("Numbers: %d\n", number_count);
    printf("Special symbols: %d\n", special_symbol_count);
    printf("Literals: %d\n", literal_count);

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

int yywrap() {
    return 1;  // End of file
}
```

Practical-6

Aim:

Program to implement Recursive Descent Parsing in C

Code:

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

char string[200];
int i = 0;
int Flag = 0;

void match(char t) {
   if (t == string[i]) {
      i++;
   } else {
      printf("Error");
      Flag = 1;
   }
}

void E_() {
   if (string[i] == '+' || string[i] == '-') {
      match(string[i]);
      match('i');
      E_();
   }
}

void E() {
   if (string[i] == 'i') {
      match('i');
      E_();
   } else {
      Flag = 1;
   }
}

int main() {
   printf("Enter the Sequence of Character ending the sequence with '$': ");
   scanf("%s", string);
```

```
    i = 0;
    Flag = 0;

    E();

    if (!Flag && string[i] == '$') {
        printf("Success");
    } else {
        if (!Flag) {
            printf("Error");
        }
    }

    return 0;
}
```

Practical-7

Aim:

- **a.** To Study about Yet Another Compiler-Compiler(YACC).
- **b.** Create Yacc and Lex specification files to recognizes arithmetic expressions involving +, -, * and /.
- **c.** Create Yacc and Lex specification files are used to generate a calculator which accepts integer type arguments.
- **d.** Create Yacc and Lex specification files are used to convert infix expression to postfix expression.

Code:

a)

YACC (Yet Another Compiler-Compiler) is a tool used to generate parsers for interpreting the syntax of programming languages. It works in conjunction with a lexical analyzer (like LEX or Flex) to form the front-end of a compiler. In this experiment, we study how YACC uses grammar rules written in a notation similar to BNF (Backus-Naur Form) to construct parsers that process tokens generated by a lexical analyzer. YACC helps in building syntax analyzers that check the structure of input data and generate parse trees. This experiment provides a foundational understanding of syntax analysis and the role of parser generators in compiler design.

b)

.l file:

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

}
```

.y file:

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

c)

.l file:

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

.y file:

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(char *);
%}
%token NUM
%%
S: E '\n' { printf("%d\n", $1); return(0); }
;
E: E '+' T { $$ = $1 + $3; }
 | E '-' T { $$ = $1 - $3; }
 | T      { $$ = $1; }
;
T: T '*' F { $$ = $1 * $3; }
 | T '/' F { $$ = $1 / $3; }
 | F      { $$ = $1; }
;
F: '(' E ')' { $$ = $2; }
 | NUM      { $$ = $1; }
;
%%
void yyerror(char *s) {
   printf("%s\n", s);
}
int main() {
   yyparse();
   return 0;
}
```

d)

.l file:

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

%%
[0-9]+ { yylval.num = atoi(yytext); return INTEGER; }
[A-Za-z_][A-Za-z0-9_]* { yylval.str = yytext; return ID; }
[-+;\n*] { return *yytext; }
[ \t] ;
.tterror("invalid character");
%%
```

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

.y file:

```
%{
#include<stdio.h>
int yylex(void);
void yyerror(char *);
%}
%union {
    char *str;
    int num;
}
%token <num> INTEGER
%token <str> ID
%%
S: E '\n' {printf("\n");}
E: E '+' T { printf("+ ");}
 | E '-' T {printf("- "); }
 | T { }
T: T '' F { printf(" "); }
 | F { }
F: INTEGER {printf("%d ",$1);}
 | ID { printf("%s ",$1); }
%%
void yyerror(char *s) {
printf("%s\n",s);
}
int main() {yyparse();return 0;}
```