SCHOOL OF ENGINEERING & TECHNOLOGY

BACHELOR OF TECHNOLOGY

COMPILER DESIGN

6TH SEMESTER

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Laboratory Manual

Name: Ravalji Maitryba

Enroll ID: 22000410

Batch: B – 1

TABLE OF CONTENT

Experiment – 1

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

Source Code:

```
#include<stdio.h>
int main(){
    char input[20],c;
    int state=0,i=0;
    printf("\n Enter a string:");
    scanf("%s",input);
    while(input[i]!='\0'){
        c=input[i++];
        switch(state){
            case 0:
                if(c=='a')
                    state=1;
                else if(c=='b')
                    state=2;
                else
                    state=3;
                break;
            case 1:
                if(c=='a' || c=='b' )
                    state=1;
                else
```

```
            state=3;
        break;
    case 2:
        if(c=='a' || c=='b' )
            state=2;
        break;
    case 3:
        printf("\n %s is not recognized.",input);
        exit(0);
    }
  }
if(state==1)
 printf("\n %s is valid",input);
else
   printf("\n %s is not valid",input);
return 0;
}
```

OUTPUT:

```
 Enter a string:abba

 abba is valid
--------------------------------
Process exited after 3.858 seconds with return value 0
Press any key to continue . . .
```

Aim - b: Write a program to recognize strings end with 'a'.

Source Code:

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

int main() {
    char input[100];
    int length, flag = 1;

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

    length = strlen(input);

                    int i;
    for (i = 0; i < length; i++) {
        switch (input[i]) {
            case 'a':
            case 'b':
                break;
            default:
                flag = 0;
                break;
        }
```

```
        if (!flag) {
            break;
        }
    }

    if (flag) {
        if (length > 0) {
            switch (input[length - 1]) {
                case 'a':
                    printf("The string ends with 'a'.\n");
                    break;
                default:
                    printf("The string does not end with 'a'.\n");
                    break;
            }
        } else {
            printf("The string is empty.\n");
        }
    } else {
        printf("Invalid input! The string should contain only 'a' and 'b'.\n");
    }

    return 0;
}
```

OUTPUT:

```
Enter a string: baaba
The string ends with 'a'.

--------------------------------
Process exited after 4.19 seconds with return value 0
Press any key to continue . . .
```

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

Source Code:

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

#define MAX_LEN 100

int main() {
    FILE *file = fopen("input.txt", "r");
    if (!file) {
        printf("Error opening file\n");
        return 1;
    }
```

```
    char line[MAX_LEN];
    while (fgets(line, MAX_LEN, file)) {
        line[strcspn(line, "\n")] = 0;
        int len = strlen(line);
        switch (len >= 2 && line[len - 2] == 'a' && line[len - 1] == 'b') {
            case 1:
                printf("%s - Valid\n", line);
                break;
            default:
                printf("%s - Not Valid\n", line);
                break;
        }
    }

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

OUTPUT:

```
abbabab - Valid

--------------------------------
Process exited after 0.08224 seconds with return value 0
Press any key to continue . . .
```

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

Source Code:

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

int main() {
    FILE *fp;
    char input[100], c;
    int state = 0, i = 0;

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

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

    while (input[i] != '\0') {
        c = input[i++];
        switch (state) {
            case 0:
```

```
            if (c == 'a')
                state = 1;
            else
                state = 3;
            break;
        case 1:
            if (c == 'a')
                state = 1;
            else if (c == 'b')
                state = 2;
            else
                state = 3;
            break;
        case 2:
            if (c == 'a' || c == 'b')
                state = 2;
            else
                state = 3;
            break;
        case 3: /
            printf("\n%s is not valid.\n", input);
            return 0;
    }
}

if (state == 2)
```

```
        printf("\n%s is valid.\n", input);
    else
        printf("\n%s is not valid.\n", input);

    return 0;
}
```

OUTPUT:

```
abbab is valid.

--------------------------------
Process exited after 0.1069 seconds with return value 0
Press any key to continue . . .
```

Experiment – 2

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

Source Code:

```
#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;

                }
                else
                printf("invalid identifier");



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

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

OUTPUT:

```
Enter an identifier:vk_18
Valid identifier
--------------------------------
Process exited after 48.83 seconds with return value 16
Press any key to continue . . .
```

Aim - b: Write a program to recognize the valid operators.

Source Code:

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

int main() {
    char input[50];
    const char *validOperators[] = {
        "+", "-", "*", "/", "%",
        "=", "+=", "-=", "*=", "/=", "%=",
        "==", "!=", ">", "<", ">=", "<=",
        "&&", "||", "!",
        "&", "|", "^", "~", "<<", ">>",
        "++", "--",
        ",", ".", "->",
```

```
    "(", ")", "[", "]", "{", "}",
    "?", ":",
    "sizeof",
    "->", "."
};

int numOperators = sizeof(validOperators) / sizeof(validOperators[0]);

printf("Enter a potential C operator (or 'exit' to quit): ");

while (1) {
    scanf("%49s", input);

    if (strcmp(input, "exit") == 0) {
        break;
    }

    bool found = false;
    int i = 0;
    while (i < numOperators) {
        switch (strcmp(input, validOperators[i])) {
            case 0:
                found = true;
                i = numOperators;
                break;
            default:
```

```
                i++;
                break;
            }
        }

        if (found) {
            printf("\"%s\" is a valid C operator.\n", input);
        } else {
            printf("\"%s\" is NOT a valid C operator.\n", input);
        }

        printf("Enter another operator (or 'exit' to quit): ");
    }

    printf("Exiting.\n");
    return 0;
}
```

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 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 36.52 seconds with return value 0
Press any key to continue . . .
```

Aim - c: Write a program to recognize the valid number.

Source Code:

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

void check_valid_number(char *input) {
    int state = 0, i = 0;
    char lexeme[100];
```

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

    switch (state) {
        case 0:
            if (isdigit(c)) {
                state = 1;
            } else if (c == '.') {
                state = 2;
            } else {
                printf("Invalid number: %s\n", input);
                return;
            }
            break;

        case 1:
            if (isdigit(c)) {
                state = 1;
            } else if (c == '.') {
                state = 3;
            } else if (c == 'E' || c == 'e') {
                state = 5;
            } else {
                printf("%s is a valid number\n", input);
                return;
            }
```

```
        break;

    case 2:
        if (isdigit(c)) {
            state = 3;
        } else {
            printf("Invalid number: %s\n", input);
            return;
        }
        break;

    case 3:
        if (isdigit(c)) {
            state = 3;
        } else if (c == 'E' || c == 'e') {
            state = 5;
        } else {
            printf("%s is a valid number\n", input);
            return;
        }
        break;

    case 5:
        if (c == '+' || c == '-') {
            state = 6;
        } else if (isdigit(c)) {
```

```
            state = 7;
        } else {
            printf("Invalid number: %s\n", input);
            return;
        }
        break;

    case 6:
        if (isdigit(c)) {
            state = 7;
        } else {
            printf("Invalid number: %s\n", input);
            return;
        }
        break;

    case 7:
        if (isdigit(c)) {
            state = 7;
        } else {
            printf("%s is a valid number\n", input);
            return;
        }
        break;
}
i++;
```

```
    }

    if (state == 1 || state == 3 || state == 7) {
        printf("%s is a valid number\n", input);
    } else {
        printf("Invalid number: %s\n", input);
    }
}

int main() {
    char input[100];

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

    check_valid_number(input);

    return 0;
}
```

OUTPUT:

```
Enter a number: 115.2
115.2 is a valid number

--------------------------------
Process exited after 4.671 seconds with return value 0
Press any key to continue . . .
```

Aim - d: Write a program to recognize the valid comments.

Source Code:

```
#include <stdio.h>

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

    file = fopen("input3.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 if (input[i] == '*')
            state = 5;
        else
            state = 4;
        break;

    case 6:
        state = 3;
        break;
```

```
        }
        i++;
    }

    if (state == 2)
        printf("This is a single line comment.\n");
    else if (state == 6)
        printf("This is a multiline comment.\n");
    else
        printf("This is not a comment.\n");

    return 0;
}
```

INPUT FILE:

```
//hello
```

OUTPUT:

```
This is a single line comment.

--------------------------------
Process exited after 0.1063 seconds with return value 0
Press any key to continue . . .
```

INPUT FILE:

```
/* hello */
```

OUTPUT:

```
This is a multiline comment.

--------------------------------
Process exited after 0.1302 seconds with return value 0
Press any key to continue . . .
```

Aim - e: Program to implement Lexical Analyzer.

Source Code:

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

const char *keywords[] = {"int", "float", "if", "else", "while", "return", "for",
"do", "switch", "case"};
#define NUM_KEYWORDS (sizeof(keywords) / sizeof(keywords[0]))

int isKeyword(char *str) {
                        int i;
    for (i = 0; i < NUM_KEYWORDS; i++) {
        if (strcmp(str, keywords[i]) == 0)
```

```
            return 1;
        }
        return 0;
    }

    int isOperator(char ch) {
        char operators[] = "+-*/=<>!&|";
        int i;
        for (i = 0; operators[i] != '\0'; i++) {
            if (ch == operators[i])
                return 1;
        }
        return 0;
    }

    void lexicalAnalyzer(char *input) {
        int i = 0;
        char token[50];
        int tokenIndex = 0;

        while (input[i] != '\0') {
            if (isspace(input[i])) {
                i++;
                continue;
            }
```

```
if (isalpha(input[i])) {
    tokenIndex = 0;
    while (isalnum(input[i])) {
        token[tokenIndex++] = input[i++];
    }
    token[tokenIndex] = '\0';
    if (isKeyword(token)) {
        printf("Keyword: %s\n", token);
    } else {
        printf("Identifier: %s\n", token);
    }
}
else if (isdigit(input[i])) {
    tokenIndex = 0;
    while (isdigit(input[i])) {
        token[tokenIndex++] = input[i++];
    }
    token[tokenIndex] = '\0';
    printf("Number: %s\n", token);
}
else if (isOperator(input[i])) {
    printf("Operator: %c\n", input[i]);
    i++;
}
else {
    printf("Special Symbol: %c\n", input[i]);
```

```
            i++;
        }
    }
}

int main() {
    char input[100];
    printf("Enter a string for lexical analysis: ");
    fgets(input, sizeof(input), stdin);
    lexicalAnalyzer(input);
    return 0;
}
```

OUTPUT:

```
Enter a string for lexical analysis: if (x < 100) return x;
Keyword: if
Special Symbol: (
Identifier: x
Operator: <
Number: 100
Special Symbol: )
Keyword: return
Identifier: x
Special Symbol: ;

--------------------------------
Process exited after 17.99 seconds with return value 0
Press any key to continue . . .
```

Experiment – *3*

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

The Lexical Analyzer Generator (LEX) is a powerful tool used in the field of compiler design to generate lexical analyzers, also known as tokenizers or scanners. These analyzers process input text and break it into meaningful tokens, which are then passed to the parser for further syntactic analysis. LEX allows programmers to define patterns using regular expressions, and for each pattern, an action (typically written in C) is specified. When LEX processes the input, it matches the patterns and executes the associated actions. It is often used alongside YACC (Yet Another Compiler Compiler) to create full programming language compilers.

Flex, short for Fast Lexical Analyzer, is an enhanced version of LEX. It is widely used in modern systems because it is faster, more efficient, and open-source. The syntax and structure used in Flex are very similar to LEX, which makes it easy to switch from one to the other. The typical Flex program has three sections: definitions, rules, and user code. When a Flex program is run, it reads a .l file (e.g., example.l) containing regular expressions and C code, compiles it into a C file lex.yy.c, and then this file is compiled using a C compiler to produce an executable scanner.

Example:

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

%%
[0-9]+      { printf("Number: %s\n", yytext); }
[a-zA-Z]+   { printf("Word: %s\n", yytext); }
.|\n        { ; } // Ignore other characters
%%
```

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


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

In this example, the lexer recognizes sequences of digits as numbers and alphabetic strings as words. When you run this program and provide an input like 123 hello, the output will be:

Number: 123

Word: hello

Experiment – 4

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

Source Code:

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


int char_count = 0, word_count = 0, line_count = 0;
FILE *yyin;
%}


%%
```

```
\n          { line_count++; char_count++; }
[^\n\t ]+     { word_count++; char_count += yyleng; }
.           { char_count++; }

%%

int main() {
    FILE *file = fopen("first.txt", "r");
    if (!file) {
        perror("Error opening file");
        return 1;
    }

    yyin = file;
    yylex();
    fclose(file);

    printf("\nNumber of Characters: %d", char_count);
    printf("\nNumber of Words: %d", word_count);
    printf("\nNumber of Lines: %d\n", line_count);

    return 0;
}

int yywrap() {
    return 1;
```

}

INPUT FILE:

```
Flex (Fast Lexical Analyzer Generator, or simply Flex, is
a tool for generating lexical analyzers scanners or
lexers. \n
Written by Vern Paxson in C, circa 1987, Flex is designed
to produce lexical analyzers that is faster than the
original Lex program. \n
Today it is often used along with Berkeley Yacc or GNU
Bison parser generators.
```

OUTPUT:

```
Number of Characters: 336
Number of Words: 57
Number of Lines: 2

--------------------------------
Process exited after 0.1181 seconds with return value 0
Press any key to continue . . .
```

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

Source Code:

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

```
%%
[aeiouAEIOU] {vowels++;}
[a-zA-Z]  {consonants++; }
\n
.
%%
int main() {
yyin = fopen("myfile.txt", "r");
yylex();
 printf(" This File contains...");
 printf("\n\t%d vowels",vowels);
 printf ("\n\t%d consonants",consonants);
return 0;
}
int yywrap() { return 1; }
```

Input File:

```
Flex (Fast Lexical Analyzer Generator, or simply Flex, is
a tool for generating lexical analyzers scanners or
lexers. \n
Written by Vern Paxson in C, circa 1987, Flex is designed
to produce lexical analyzers that is faster than the
original Lex program. \n
Today it is often used along with Berkeley Yacc or GNU
Bison parser generators.
```

OUTPUT:

```
 This File contains...
        96 vowels
        170 consonants
--------------------------------
Process exited after 0.06981 seconds with return value 0
Press any key to continue . . .
```

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

Source Code:

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

%%

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

%%

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

```
int main() {
    FILE *file = fopen("myfilenum.txt", "r");
    if (!file) {
        perror("Error opening input.txt");
        return 1;
    }

    yyin = file;
    yylex();

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

INPUT FILE:

```
1 2 3 4 5 6
```

OUTPUT:

```
Number found: 1
Number found: 2
Number found: 3
Number found: 4
Number found: 5
Number found: 6

--------------------------------
Process exited after 0.1402 seconds with return value 0
Press any key to continue . . .
```

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

Source Code:

```
%{
#include<stdio.h>
int line_number = 1;
%}
%%
.+ {fprintf(yyout, "%d: %s", line_number, yytext);line_number++;}
%%
int main()
{
   yyin = fopen("fourth.txt", "r");
   yyout= fopen("fourth_output.txt", "w");
   yylex();
   printf("done");
   return 0;
}
int yywrap(){return (1);}
```

INPUT FILE:

```
Flex Fast Lexical Analyzer Generator, or simply Flex, is a
tool for generating lexical analyzers scanners or lexers.
\n
Written by Vern Paxson in C, circa 1987, Flex is designed
to produce lexical analyzers that is faster than the
original Lex program. \n
Today it is often used along with Berkeley Yacc or GNU
Bison parser generators.
```

OUTPUT:

```
done
--------------------------------
Process exited after 0.1506 seconds with return value 0
Press any key to continue . . .
```

```
1: Flex Fast Lexical Analyzer Generator, or simply Flex,
is a tool for generating lexical analyzers scanners or
lexers. \n
2: Written by Vern Paxson in C, circa 1987, Flex is
designed to produce lexical analyzers that is faster than
the original Lex program. \n
3: Today it is often used along with Berkeley Yacc or GNU
Bison parser generators.
```

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

Source Code:

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

```
%}
%%
"<"[A-Za-z0-9]+">"  printf("%s is a valid markup tag\n", yytext);
"<!--"[^-->]*"-->"  num++;
\n  ;
.  ;
%%
int main()
{
   yyin = fopen("fifth.txt", "r");
   yylex();
   printf("Number of comments are: %d", num);
   return 0;
}
int yywrap(){return (1);}
```

INPUT FILE:

```
<html>
<head>
<title>My File</title>
</head>
</html>
```

OUTPUT:

```
<html> is a valid markup tag
<head> is a valid markup tag
<title> is a valid markup tag
Number of comments are: 0
------------------------------
Process exited after 0.1288 seconds with return value 0
Press any key to continue . . .
```

Experiment – 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.

Source Code:

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

FILE *output;
int comment_count = 0;
%}

%%

\/\/.*                          { comment_count++; }
\/\*([^*]|\*+[^/])*\*+\/        { comment_count++; }
```

```
.|\n                    { fputc(yytext[0], output); }

%%

int yywrap() {
    return 1;
}

int main() {
    FILE *input = fopen("input3.txt", "r");
    output = fopen("output.txt", "w");

    if (!input || !output) {
        perror("File error");
        return 1;
    }

    yyin = input;
    yylex();

    fclose(input);
    fclose(output);

    printf("Number of comments removed: %d\n", comment_count);
    return 0;
}
```

INPUT FILE:

```
#include <stdio.h>

// This is a single-line comment

int main() {
    printf("Hello World"); /* Inline comment */
    return 0;
}

/*
Multiline
Comment
*/
```

OUTPUT:

```
Number of comments removed: 3

--------------------------------
Process exited after 0.1299 seconds with return value 0
Press any key to continue . . .
```

```
#include <stdio.h>



int main() {
    printf("Hello World");
    return 0;
}
```

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

Source Code:

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

FILE *yyin;
%}

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 */ }

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

%%

int yywrap() {
    return 1;
}

int main() {
    yyin = fopen("input.txt", "r");
    if (!yyin) {
        perror("Failed to open file");
        return 1;
    }
    yylex();
```

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

Input File:

```
int main() {
    float num = 3.14;
    char ch = 'A';
    printf("Hello, World!");
}
```

OUTPUT:

```
Keyword: int
Identifier: main
Special Symbol: (
Special Symbol: )
Special Symbol: {
Keyword: float
Identifier: num
Operator: =
Number: 3.14
Special Symbol: ;
Keyword: char
Identifier: ch
Operator: =
Unknown Token: '
Identifier: A
Unknown Token: '
Special Symbol: ;
Identifier: printf
Special Symbol: (
Literal: "Hello, World!"
Special Symbol: )
Special Symbol: ;
Special Symbol: }

--------------------------------
Process exited after 0.2053 seconds with return value 0
```

Experiment – 6

Aim: Program to implement Recursive Descent Parsing in C.

Source Code:

```
#include<stdio.h>
#include<stdlib.h>
/*
E-> iE_
E_-> +iE_ / -iE_ / epsilon
*/
char s[20];
int i=1;
char l;
int match(char t)
{
   if(l==t){
      l=s[i];
      i++; }
   else{
      printf("Sytax error");
      exit(1);}
}
int E_()
{
   if(l=='+'){
```

```
        match('+');
        match('i');
        E_(); }
    else  if(l=='-'){
        match('-');
        match('i');
        E_(); }
    else
        return(1);
}
int E()
{
    if(l=='i'){
        match('i');
        E_(); }
}

int main()
{
    printf("\n Enter the set of characters to be checked :");
    scanf("%s",&s);
    l=s[0];
    E();
    if(l=='$')
    {
        printf("Success \n");
```

```
    }
    else{
        printf("syntax error");
    }
    return 0;
}
```

OUTPUT:

```
 Enter the set of characters to be checked :i+i-i$
Success

--------------------------------
Process exited after 2.035 seconds with return value 0
Press any key to continue . . .
```

Experiment – 7

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

YACC, which stands for **Yet Another Compiler Compiler**, is a powerful tool used in compiler construction to automate the creation of syntax analyzers or parsers. It works by taking a formal grammar as input and generating the corresponding C code capable of parsing that grammar. YACC is typically used alongside **Lex**, which performs lexical analysis (tokenizing the input), while YACC focuses on parsing those tokens based on syntactic rules.

A YACC file consists of three main sections: the declaration section, the grammar rules section, and the C code section. In the declaration section, you define tokens (usually provided by Lex) and include any necessary headers. The grammar rules section uses a notation similar to Backus-Naur Form (BNF) to define how tokens

combine to form valid constructs, and you can include C code within actions {} to execute when a rule is matched. Finally, the user-defined C code section often includes the main() function and the error handling function yyerror(). The Lex file (expr.l) tokenizes the input, while the YACC file (expr.y) checks the syntax.

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

Lex Source Code:

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

Yacc Source Code:

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

E : E '+' T  { }
  | E '-' T  { }
  | T        { }
  ;

T : T '*' F  { }
  | T '/' F  { }
  | F        { }
  ;

F : num      { }
  ;
%%
void yyerror(char *s) {
```

```
 printf("Syntax Error: %s\n", s);
}
int main() {
 return yyparse();
}
```

OUTPUT:

```
D:\NUV\3rd_Year\6_sem\CD\lab\yacc_tool_tasks\Lab 7 que 2 syntax>flex sampleL.l

D:\NUV\3rd_Year\6_sem\CD\lab\yacc_tool_tasks\Lab 7 que 2 syntax>bison -d sampleY.y

D:\NUV\3rd_Year\6_sem\CD\lab\yacc_tool_tasks\Lab 7 que 2 syntax>gcc lex.yy.c sampleY.tab.c

D:\NUV\3rd_Year\6_sem\CD\lab\yacc_tool_tasks\Lab 7 que 2 syntax>a.exe
3+4-5*8/10
valid syntax

D:\NUV\3rd_Year\6_sem\CD\lab\yacc_tool_tasks\Lab 7 que 2 syntax>a.exe
7-+32*1
Syntax Error: syntax error
```

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

Lex Source Code:

```
%{
#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;
}
```

Yacc Source Code:

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

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

OUTPUT:

```
D:\NUV\3rd_Year\6_sem\CD\lab\yacc_tool_tasks\calc_my_1>flex lex.l

D:\NUV\3rd_Year\6_sem\CD\lab\yacc_tool_tasks\calc_my_1>bison -d yacc.y

D:\NUV\3rd_Year\6_sem\CD\lab\yacc_tool_tasks\calc_my_1>gcc lex.yy.c yacc.tab.c

D:\NUV\3rd_Year\6_sem\CD\lab\yacc_tool_tasks\calc_my_1>a.exe
2*3+4
10

D:\NUV\3rd_Year\6_sem\CD\lab\yacc_tool_tasks\calc_my_1>
```

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

Lex Source Code:

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

void yyerror(char *);
%}
```

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

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

Yacc Source Code:

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

%union {
   char *str;
   int num;
}

%token <num> INTEGER
```

```
%token <str> ID

%%
S: E '\n'      { printf("\n"); }
 ;

E: E '+' T     { printf("+ "); }
 | E '-' T     { printf("- "); }
 | T
 ;

T: T '*' F     { printf("* "); }
 | F
 ;

F: INTEGER     { printf("%d ", $1); }
 | ID          { printf("%s ", $1); }
 ;

%%

void yyerror(char *s) {
   printf("%s\n", s);
}

int main() {
```

```
    yyparse();
    return 0;
}
```

OUTPUT:

```
D:\NUV\3rd_Year\6_sem\CD\lab\yacc_tool_tasks\infix_postfix>flex lex.l

D:\NUV\3rd_Year\6_sem\CD\lab\yacc_tool_tasks\infix_postfix>bison -d bison.y
bison.y: conflicts: 3 shift/reduce

D:\NUV\3rd_Year\6_sem\CD\lab\yacc_tool_tasks\infix_postfix>gcc lex.yy.c bison.tab.c

D:\NUV\3rd_Year\6_sem\CD\lab\yacc_tool_tasks\infix_postfix>a.exe
3 * 4 - 2
3 4 * 2 -
```