

Practical – 1

AIM: Implementation of Finite Automata and String Validation.

➤ PR-1.c

```
#include <stdio.h>

int main()
{
    char str[100];
    printf("-----STRING VALIDATON-----\n");
    printf("Enter string : ");
    scanf("%s", str);
    printf("Validating string %s for string ending with 10", str);
    printf("\nLoading.....");

    char f = 'a';
    int i;
    for (i = 0; str[i] != '\0'; i++)
    {
        switch (f)
        {
            case 'a':
                if (str[i] == '0')
                    f = 'a';
                else
                    f = 'b';
                break;

            case 'b':
                if (str[i] == '0')
                    f = 'c';
                else
                    f = 'b';
                break;

            case 'c':
                if (str[i] == '0')
                    f = 'a';
                else
                    f = 'b';
                break;
        }
    }

    if (f == 'c')
        printf("\n%s is valid ", str);
}
```

```
else
    printf("\n%s is not valid", str);
return 0;
}
```

➤ **OUTPUT**

```
PS C:\Users\Hp\OneDrive\Documents\implementation code> cd "c:\Users
\Hp\OneDrive\Documents\implementation code\" ; if ($?) { gcc PR-1.c
-o PR-1 } ; if ($?) { .\PR-1 }
-----STRING VALIDATON-----
Enter string : 1011
Validating string 1011 for string ending with 10
Loading.....
1011 is not valid
PS C:\Users\Hp\OneDrive\Documents\implementation code> cd "c:\Users
\Hp\OneDrive\Documents\implementation code\" ; if ($?) { gcc PR-1.c
-o PR-1 } ; if ($?) { .\PR-1 }
-----STRING VALIDATON-----
Enter string : 10110
Validating string 10110 for string ending with 10
Loading.....
10110 is valid
PS C:\Users\Hp\OneDrive\Documents\implementation code> █
```

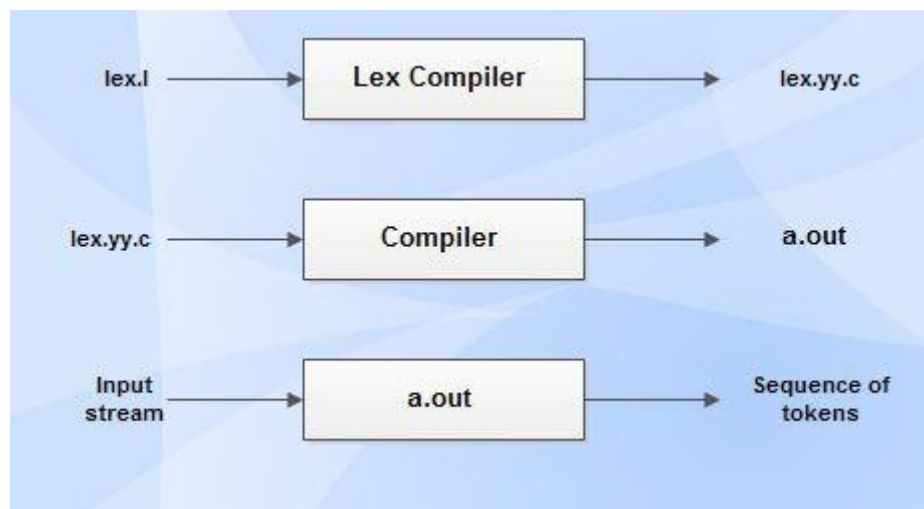
Practical – 2

AIM: Introduction to Lex Tool.

➤ LEX (Lexical Analyzer Generator):

- Lex is a tool in lexical analysis phase to recognize tokens using regular expression.
- Lex tool itself is a lex compiler.

➤ Use of Lex



- *lex.l* is an input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms *lex.l* to a C program known as *lex.yy.c*.
- *lex.yy.c* is compiled by the C compiler to a file called *a.out*.
- The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.
- *yylval* is a global variable which is shared by lexical analyzer and parser to return the name and an attribute value of token.
- The attribute value can be numeric code, pointer to symbol table or nothing.
- Another tool for lexical analyzer generation is Flex.

➤ Structure of Lex Programs

- Lex program will be in following form

declarations

%%

translation rules

%%

auxiliary functions

- **Declarations:** This section includes declaration of variables, constants and regular definitions.
- **Translation rules:** It contains regular expressions and code segments.
Form: Pattern {Action}
 - Pattern is a regular expression or regular definition.
 - Action refers to segments of code.
- **Auxiliary functions:** This section holds additional functions which are used in actions. These functions are compiled separately and loaded with lexical analyzer.
- Lexical analyzer produced by lex starts its process by reading one character at a time until a valid match for a pattern is found.
- Once a match is found, the associated action takes place to produce token.
- The token is then given to parser for further processing.

➤ **Design of Lexical Analyzer**

- Lexical analyzer can either be generated by NFA or by DFA.
- DFA is preferable in the implementation of lex.

Practical – 3

AIM: Implement following Programs Using Lex

a. Generate Histogram of words

b. Ceasor Cypher

c. Extract single and multiline comments from C Program.

➤ **PR-3A.1**

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

char word1 [] = "atul";
char word2 [] = "satish";
char word3 [] = "shubh";
char word4 [] = "sagar";

int count1 = 0, count2 = 0, count3 = 0, count4 = 0;
}%

%%
[a-zA-Z]+ {
    if(strcmp(yytext, word1)==0)
        count1++;
    if(strcmp(yytext, word2)==0)
        count2++;
    if(strcmp(yytext, word3)==0)
        count3++;
    if(strcmp(yytext, word4)==0)
        count4++;
}
.;
%%

int yywrap()
{
    return 1;
}

int main()
{
    extern FILE *yyin, *yyout;

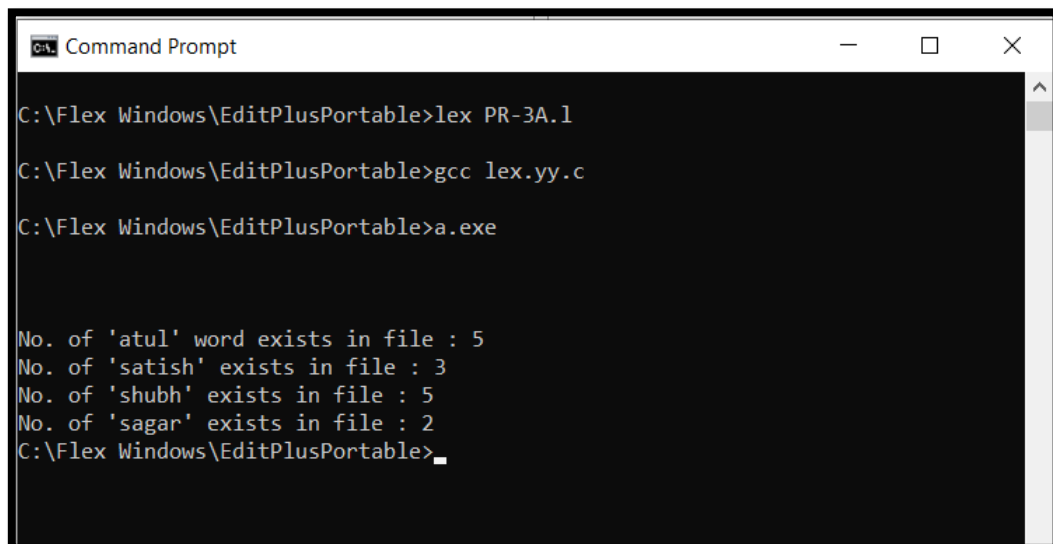
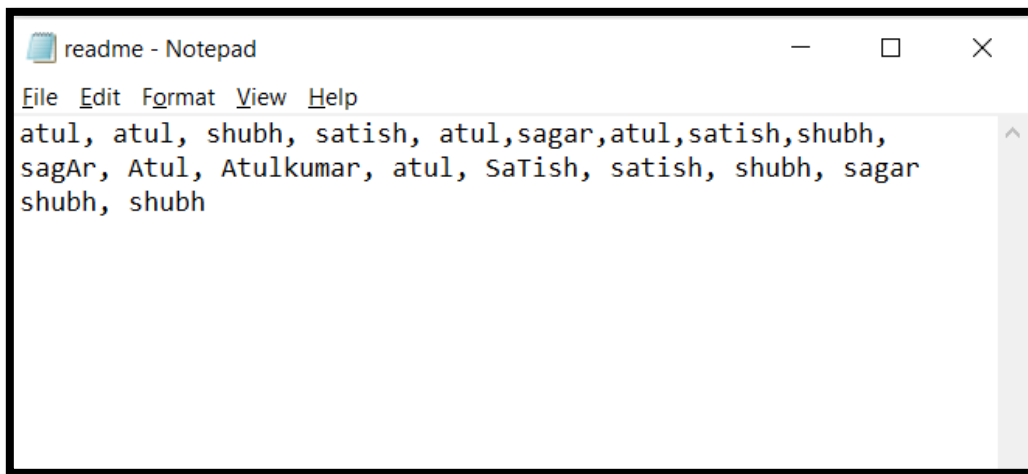
    /* open the input file
```

```
in read mode */
yyin=fopen("readme.txt", "r");
yylex();

printf("\nNo. of \'%s\' word exists in file : %d", word1,count1);
printf("\nNo. of \'%s\' exists in file : %d", word2,count2);
printf("\nNo. of \'%s\' exists in file : %d", word3,count3);
printf("\nNo. of \'%s\' exists in file : %d", word4,count4);

}
```

➤ OUTPUT



➤ **PR-3B.1**

```

%{
    #include<stdio.h>
    int shift;
}%

%%
[a-z] {
    char ch = yytext[0];
    ch += shift;
    if (ch> 'z') ch -= 26;
    printf ("%c" ,ch );
}
[A-Z] {
    char ch = yytext[0] ;
    ch += shift;
    if (ch> 'Z') ch -= 26;
    printf("%c",ch);
}
. {exit(0);}
%%

int main(){
    printf("Enter an no. of alphabet to shift: \n");
    scanf("%d", &shift);
    printf("Enter the string: \n");
    yylex();
    return 0;
}
int yywrap(){
    return 1;
}

```

➤ **OUTPUT**

```

C:\Flex Windows\EditPlusPortable>lex PR-3B.1

C:\Flex Windows\EditPlusPortable>gcc lex.yy.c

C:\Flex Windows\EditPlusPortable>a.exe
Enter an no. of alphabet to shift:
2
Enter the string:
Atul
Cvwn
ATUL
CVWN
Computer
Eqorwvgt

```

➤ **PR-3C.1**

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

char str[] = "";
%}

%%
(\\\/)(.*) {printf("%s\n", yytext);};
\\\/*(.*)*.*\\\/ {printf("%s\n", yytext);};
%%

int yywrap()
{
return 1;
}

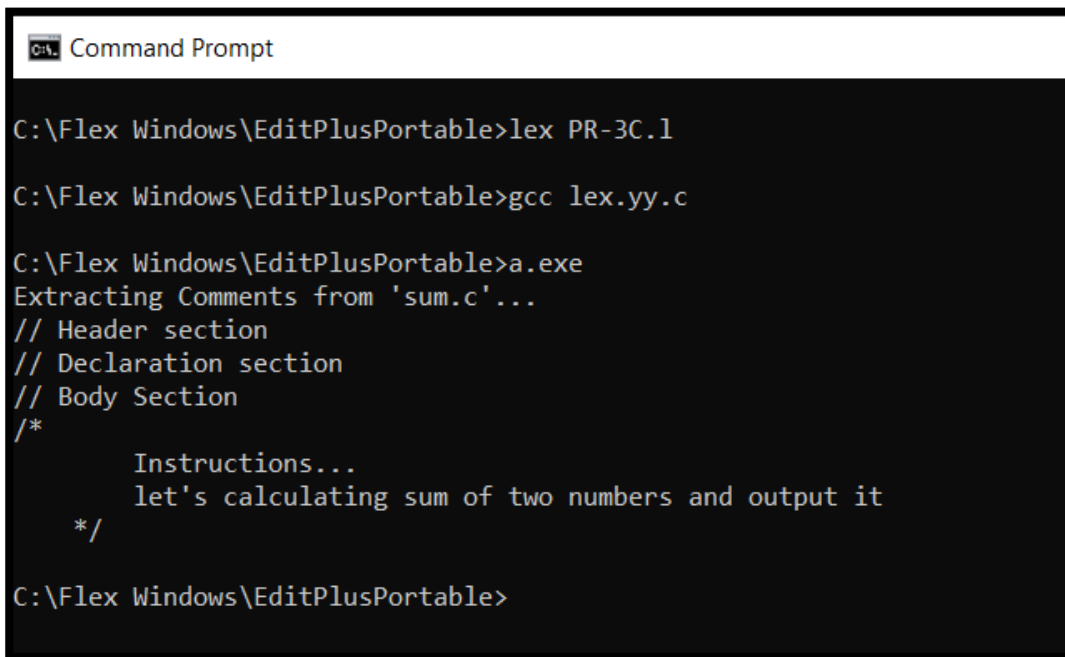
int main()
{
yyin=fopen("sum.c","r");
yyout = fopen("Output.txt", "w");
printf("Extracting Comments from 'sum.c'...\n");
yylex();
return 0;
}
```

➤ **OUTPUT:**• **Input file (sum.c):**

```
// Header section
#include <stdio.h>

// Declaration section
int a = 5, b = 10, c;

// Body Section
void main()
{
/*
Instructions...
let's calculating sum of two numbers and output it
*/
c = a + b;
printf("%d + %d = %d", a, b, c);
}
```

```
Command Prompt

C:\Flex Windows\EditPlusPortable>lex PR-3C.l

C:\Flex Windows\EditPlusPortable>gcc lex.yy.c

C:\Flex Windows\EditPlusPortable>a.exe
Extracting Comments from 'sum.c'...
// Header section
// Declaration section
// Body Section
/*
    Instructions...
    let's calculating sum of two numbers and output it
*/

C:\Flex Windows\EditPlusPortable>
```

Practical – 4

AIM: Implement following Programs Using Lex

a. Convert Roman to Decimal

b. Check weather given statement is compound or simple

c. Extract html tags from .html file

➤ **PR-4A.1**

```
%{
    #include<stdio.h>
    #include<string.h>
    #define I 1
    #define V 5
    #define X 10
    #define L 50
    #define C 100
    #define D 500
    #define M 1000

    char str[10];
    int value(char r);
}%

%%
^M{0,3}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$ strcpy(str, yytext);

.* { printf("##Wrong input##\n"); };

[\n] return 0;

%%

int yywrap()
{
    return 1;
}

int main()
{
    printf("----- Roman Number to Decimal Converter ----- \n");
    printf("\nEnter Roman Number : ");
    yylex();

    int res = 0,i;

    if(strlen(str)>0)
    {
```

```

    for (i = 0; i < strlen(str); i++)
    {
        int s1 = value(str[i]);

        if (i + 1 < strlen(str))
        {
            int s2 = value(str[i + 1]);

            if (s1 >= s2)
            {
                res = res + s1;
            }
            else
            {
                res = res + s2 - s1;
                i++;
            }
        }
        else {
            res = res + s1;
        }
    }
    printf("Result : %d\n",res);
}
return 0;
}
int value(char r)
{
    if (r == 'I')
        return 1;
    if (r == 'V')
        return 5;
    if (r == 'X')
        return 10;
    if (r == 'L')
        return 50;
    if (r == 'C')
        return 100;
    if (r == 'D')
        return 500;
    if (r == 'M')
        return 1000;

    return -1;
}

```

➤ OUTPUT

```
C:\Windows\System32\cmd.exe
C:\Flex Windows\EditPlusPortable>lex PR-4A.l.

C:\Flex Windows\EditPlusPortable>gcc lex.yy.c

C:\Flex Windows\EditPlusPortable>a.exe
----- Roman Number to Decimal Converter -----

Enter Roman Number : XLIX
Result : 49

C:\Flex Windows\EditPlusPortable>a.exe
----- Roman Number to Decimal Converter -----

Enter Roman Number : CD
Result : 400

C:\Flex Windows\EditPlusPortable>a.exe
----- Roman Number to Decimal Converter -----

Enter Roman Number : IB
##Wrong input##

C:\Flex Windows\EditPlusPortable>
```

➤ **PRA-4B.1**

```

%%
"and"|"or"|"but"|"because"|"nevertheless" {printf("COMPOUND SENTANCE");
exit(0); }
.;
\n return 0;
%%

int yywrap()
{
return 1;
}

main()
{
printf("\nENTER THE SENTANCE : ");
yylex();
printf("SIMPLE SENTANCE");

}

```

➤ **OUTPUT**

```

C:\Flex Windows\EditPlusPortable>lex PR-4B.1

C:\Flex Windows\EditPlusPortable>gcc lex.yy.c

C:\Flex Windows\EditPlusPortable>a.exe

ENTER THE SENTANCE : this is simple but elegant
COMPOUND SENTANCE
C:\Flex Windows\EditPlusPortable>a.exe

ENTER THE SENTANCE : this is simple
SIMPLE SENTANCE
C:\Flex Windows\EditPlusPortable>

```

➤ **PR-4C.1**

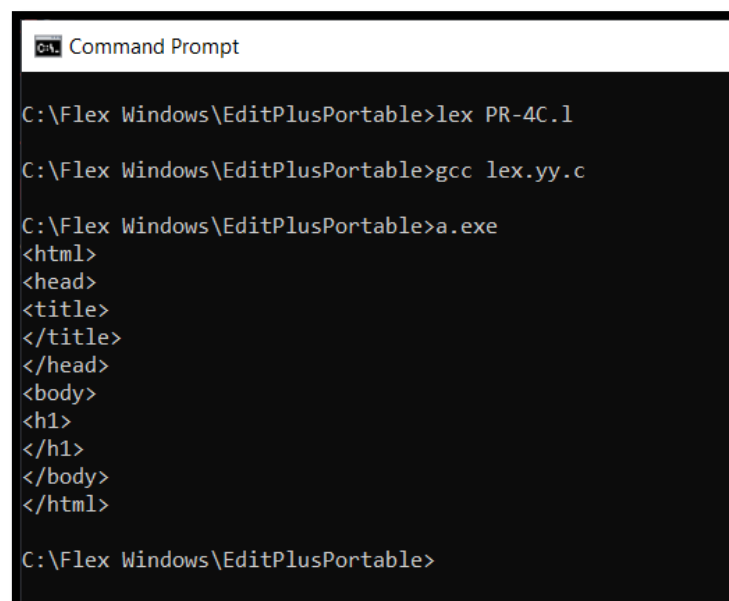
```
%%
\<[^>]*\> {printf("%s\n",yytext);};
.\n;
%%
```

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

```
int main()
{ yyin = fopen("hello.html","r");
  yyout = fopen("Output1.txt","w");
  yylex();
  return 0;
}
```

➤ **OUTPUT**• **Input file (hello.html)**

```
<html>
<head>
  <title>Hello Page</title>
</head>
<body>
  <h1>Hello, World</h1>
</body>
</html>
```



```
Command Prompt

C:\Flex Windows\EditPlusPortable>lex PR-4C.1

C:\Flex Windows\EditPlusPortable>gcc lex.yy.c

C:\Flex Windows\EditPlusPortable>a.exe
<html>
<head>
<title>
</title>
</head>
<body>
<h1>
</h1>
</body>
</html>

C:\Flex Windows\EditPlusPortable>
```

Practical – 5

AIM: Implementation of Recursive Descent Parser without backtracking.

Input: The string to be parsed.

Output: Whether string parsed successfully or not.

➤ **PR-5.c**

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
void Tprime();
void Eprime();
void E();
void check();
void T();
char inputString[10];
int count, flag;

void main()
{
    count = 0;
    flag = 0;
    printf("\nEnter an Algebraic Expression : \t");
    scanf("%s", inputString);
    E();
    if ((strlen(inputString) == count) && (flag == 0))
    {
        printf("The inputString %s is parsed succesfully.\n\n", inputString);
    }
    else
    {
        printf("The inputString %s is Not parsed successfully.\n\n", inputString);
    }
}

void E()
{
    T();
    Eprime();
}

void T()
{
    check();
    Tprime();
}
```

```
void Tprime()
{
    if (inputString[count] == '*')
    {
        count++;
        check();
        Tprime();
    }
}

void check()
{
    if (isalnum(inputString[count]))
    {
        count++;
    }
    else if (inputString[count] == '(')
    {
        count++;
        E();
        if (inputString[count] == ')')
        {
            count++;
        }
        else
        {
            flag = 1;
        }
    }
    else
    {
        flag = 1;
    }
}

void Eprime()
{
    if (inputString[count] == '+')
    {
        count++;
        T();
        Eprime();
    }
}
```


➤ OUTPUT

```
PS C:\Flex Windows\EditPlusPortable> cd "c:\Flex Windows\EditPlusPortable\" ; if ($?) { gcc PR-5.c -o PR-5 } ; if ($?) { .\PR-5 }

Enter an Algebraic Expression : (a+b)c
The inputString (a+b)c is Not parsed successfully.

PS C:\Flex Windows\EditPlusPortable> cd "c:\Flex Windows\EditPlusPortable\" ; if ($?) { gcc PR-5.c -o PR-5 } ; if ($?) { .\PR-5 }

Enter an Algebraic Expression : (a+b)/c
The inputString (a+b)/c is Not parsed successfully.

PS C:\Flex Windows\EditPlusPortable> cd "c:\Flex Windows\EditPlusPortable\" ; if ($?) { gcc PR-5.c -o PR-5 } ; if ($?) { .\PR-5 }

Enter an Algebraic Expression : (a+b)*c
The inputString (a+b)*c is parsed successfully.

PS C:\Flex Windows\EditPlusPortable> █
```

Practical – 6

AIM: Finding “First” set

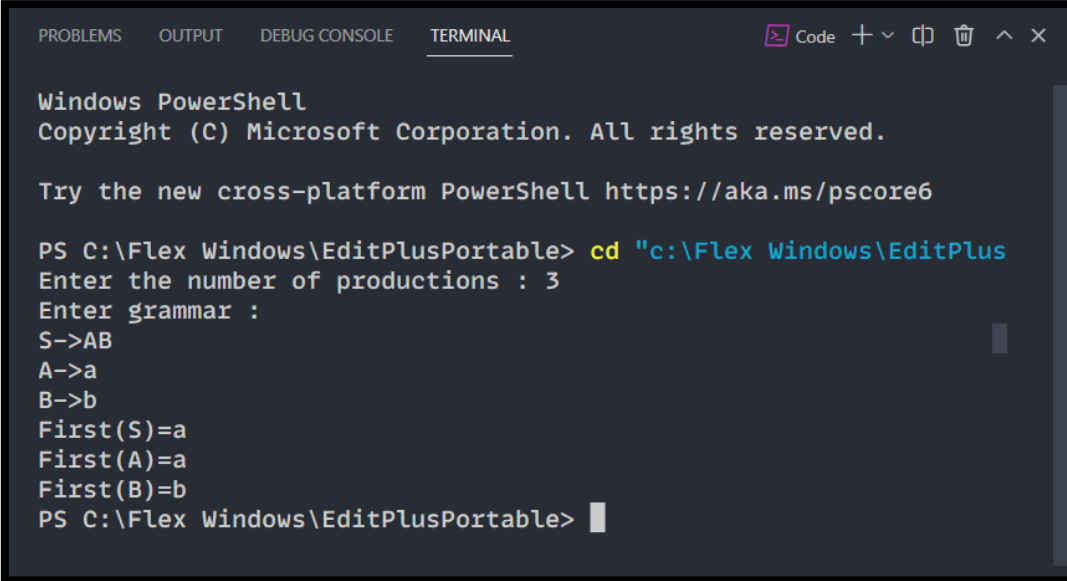
Input: The string consists of grammar symbols.

Output: The First set for a given string.

➤ **pr-6.c**

```
#include <stdio.h>
#include <ctype.h>
int main()
{
    int i, n, j, k;
    char str[10][10], f;
    printf("Enter the number of productions : ");
    scanf("%d", &n);
    printf("Enter grammar : \n");
    for (i = 0; i < n; i++)
        scanf("%s", &str[i]);
    for (i = 0; i < n; i++) {
        f = str[i][0];
        int temp = i;
        if (isupper(str[i][3]))
        {
            repeat:
            for (k = 0; k < n; k++)
            {
                if (str[k][0] == str[i][3])
                {
                    if (isupper(str[k][3]))
                    {
                        i = k;
                        goto repeat;
                    }
                    else
                    {
                        printf("First(%c)=%c\n", f, str[k][3]);
                    }
                }
            }
        }
        else
        {
            printf("First(%c)=%c\n", f, str[i][3]);
        }
        i = temp;
    }
}
```

➤ **OUTPUT**



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\Flex Windows\EditPlusPortable> cd "c:\Flex Windows\EditPlus
Enter the number of productions : 3
Enter grammar :
S->AB
A->a
B->b
First(S)=a
First(A)=a
First(B)=b
PS C:\Flex Windows\EditPlusPortable>
```

Practical – 7

AIM: Generate 3-tuple intermediate code for given infix expression.

➤ **PR-7.c:**

```
#include <stdio.h>
#include <string.h>
void pm();
void plus();
void div();
int i, ch, j, l, addr = 100;
char ex[10], expr[10], exp1[10], expr2[10], id1[5], op[5], id2[5];

void main()
{
    printf("\nEnter the Infix expression: ");
    scanf("%s", ex);
    strcpy(expr, ex);
    l = strlen(expr);
    exp1[0] = '\0';

    for (i = 0; i < l; i++)
    {
        if (expr[i] == '+' || expr[i] == '-')
        {
            if (expr[i + 2] == '/' || expr[i + 2] == '*')
            {
                pm();
                break;
            }
            else
            {
                plus();
                break;
            }
        }
        else if (expr[i] == '/' || expr[i] == '*')
        {
            div();
            break;
        }
    }
}

void pm()
{
    strrev(expr);
```

```

    j = l - i - 1;
    strncat(exp1, expr, j);
    strrev(exp1);
    printf("\nThree address code:\ntemp=%s\ntemp1=%c%cctemp\n", exp1, expr[j +
1], expr[j]);
}
void div()
{
    strncat(exp1, expr, i + 2);
    printf("\nThree address code:\ntemp=%s\ntemp1=temp%c%c\n", exp1, expr[i +
2], expr[i + 3]);
}
void plus()
{
    strncat(exp1, expr, i + 2);
    printf("\nThree address code:\ntemp=%s\ntemp1=temp%c%c\n", exp1, expr[i +
2], expr[i + 3]);
}

```

➤ OUTPUT

```

PS C:\Flex Windows\EditPlusPortable> cd "c:\Flex Windows\EditPlus
Portable\" ; if ($?) { gcc PR-7.c -o PR-7 } ; if ($?) { .\PR-7 }

Enter the Infix expression: A*B+C

Three address code:
temp=A*B
temp1=temp+C
PS C:\Flex Windows\EditPlusPortable> cd "c:\Flex Windows\EditPlus
Portable\" ; if ($?) { gcc PR-7.c -o PR-7 } ; if ($?) { .\PR-7 }

Enter the Infix expression: A/B*c

Three address code:
temp=A/B
temp1=temp*c
PS C:\Flex Windows\EditPlusPortable>

```

Practical – 8

AIM: Extract Predecessor and Successor from given Control Flow Graph.

```
package cd_p_8;
class CD_P_8{

    static class Node
    {
        int key;
        Node left, right;

        public Node()
        {}

        public Node(int key)
        {
            this.key = key;
            this.left = this.right = null;
        }
    };

    static Node pre = new Node(), suc = new Node();

    static void findPreSuc(Node root, int key)
    {
        if (root == null)
            return;

        if (root.key == key)
        {
            if (root.left != null)
            {
                Node tmp = root.left;
                while (tmp.right != null)
                    tmp = tmp.right;

                pre = tmp;
            }
            if (root.right != null)
            {
                Node tmp = root.right;
```

```

        while (tmp.left != null)
            tmp = tmp.left;

        suc = tmp;
    }
    return;
}
if (root.key > key)
{
    suc = root;
    findPreSuc(root.left, key);
}
else
{
    pre = root;
    findPreSuc(root.right, key);
}
}
static Node insert(Node node, int key)
{
    if (node == null)
        return new Node(key);
    if (key < node.key)
        node.left = insert(node.left, key);
    else
        node.right = insert(node.right, key);

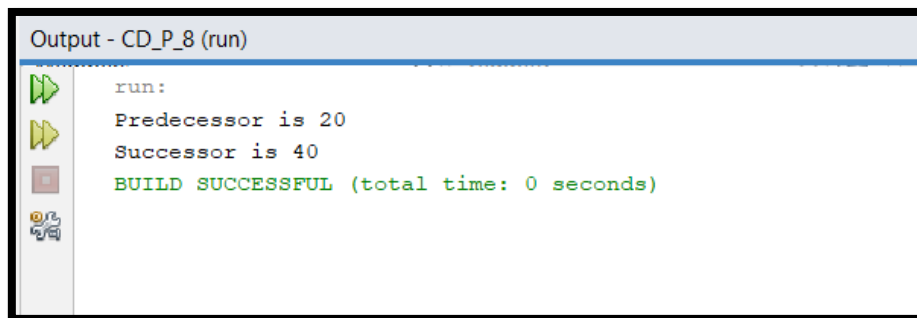
    return node;
}
public static void main(String[] args)
{
    int key = 32;
    Node root = new Node();
    root = insert(root, 10);
    insert(root, 70);
    insert(root, 40);
    insert(root, 60);
    insert(root, 80);
    insert(root, 90);
    insert(root, 20);

    findPreSuc(root, key);
    if (pre != null)
        System.out.println("Predecessor is " + pre.key);
    else
        System.out.println("No Predecessor");
}

```

```
    if (suc != null)
        System.out.println("Successor is " + suc.key);
    else
        System.out.println("No Successor");
}
```

Output Screenshots:



Practical – 9

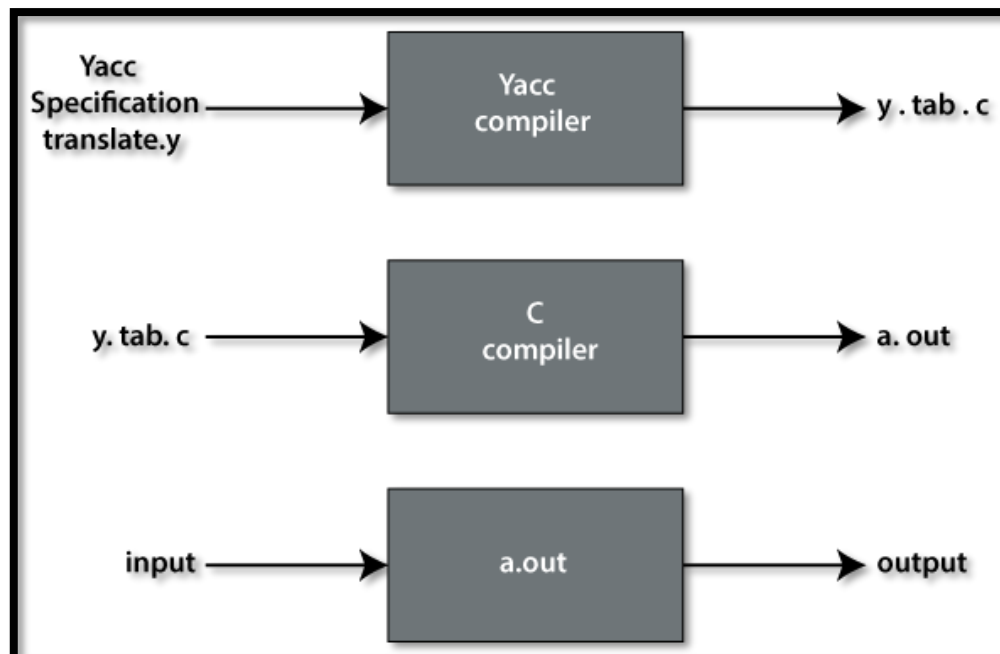
AIM: Introduction to YACC and generate Calculator Program.

➤ YACC:

- YACC is known as Yet Another Compiler Compiler. It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar, and the output is a C program. Stephen C. Johnson creates the first kind of YACC.
- If we have a file **translate.y** that consists of YACC specification, then the UNIX system command is:

YACC translate.y

- This command converts the file translate.y into a C file y.tab.c. It represents an LALR parser prepared in C with some other user's prepared C routines.
- By compiling y.tab.c along with the ly library, we will get the desired object program a.out that performs the operation defined by the original YACC program.
- The construction of translation using YACC is illustrated in the figure below:



- A YACC source program contains three parts:

- **Declarations**

%%

- **Translation rules**

%%

- **Supporting C rules**

- **Declarations Part**

- This part of YACC has two sections; both are optional. The first section has ordinary C declarations, which is delimited by %{ and %}. Any temporary variable used by the second and third sections will be kept in this part.
- This part defined the tokens that can be used in the later parts of a YACC specification.

- **Translation Rule Part**

- After the first %% pair in the YACC specification part, we place the translation rules. Every rule has a grammar production and the associated semantic action.
- A set of productions:

$$\langle \text{head} \rangle \Rightarrow \langle \text{body} \rangle_1 \mid \langle \text{body} \rangle_2 \mid \dots \mid \langle \text{body} \rangle_n$$
would be written in YACC as

$\langle \text{head} \rangle :$	$\langle \text{body} \rangle_1$	$\{ \langle \text{semantic action} \rangle_1 \}$
	$\mid \langle \text{body} \rangle_2$	$\{ \langle \text{semantic action} \rangle_2 \}$
	\dots	
	$\mid \langle \text{body} \rangle_n$	$\{ \langle \text{semantic action} \rangle_n \}$
	;	
- In a YACC production, an unquoted string of letters and digits that are not considered tokens is treated as non-terminals.
- The semantic action of YACC is a set of C statements. In a semantic action, the symbol \$\$ considered to be an attribute value associated with the head's non-terminal. While \$i considered as the value associated with the ith grammar production of the body.
- If we have left only with associated production, the semantic action will be performed. The value of \$\$ is computed in terms of \$i's by semantic action.

- **Supporting C-Rules**

- It is the last part of the YACC specification and should provide a lexical analyzer named **yylex()**. These produced tokens have the token's name and are associated with its attribute value. Whenever any token like DIGIT is returned, the returned token name should have been declared in the first part of the YACC specification.
- The attribute value which is associated with a token will communicate to the parser through a variable called yylval. This variable is defined by a YACC.
- Whenever YACC reports that there is a conflict in parsing-action, we should have to create and consult the file y.output to see why this conflict in the parsing-action has arisen and to see whether the conflict has been resolved smoothly or not.
- Instructed YACC can reduce all parsing action conflict with the help of two rules that are mentioned below:
- A reduce/reduce conflict can be removed by choosing the production which has conflict mentioned earlier in the YACC specification.
- A shift/reduce conflict is reduced in favor of shift. A shift/reduce conflict that arises from the dangling-else ambiguity can be solved correctly using this rule.

➤ **Calculator Program**

- **PR-9.1**

```
%{
#include<stdio.h>
#include "PR-9.tab.h"
extern int yylval;

}%

%%
[0-9]+ {
    yylval=atoi(yytext);
    return NUMBER;

}
[\t];

[\n] return 0;

. return yytext[0];
```

```
%%
```

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

- **PR-9.y**

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

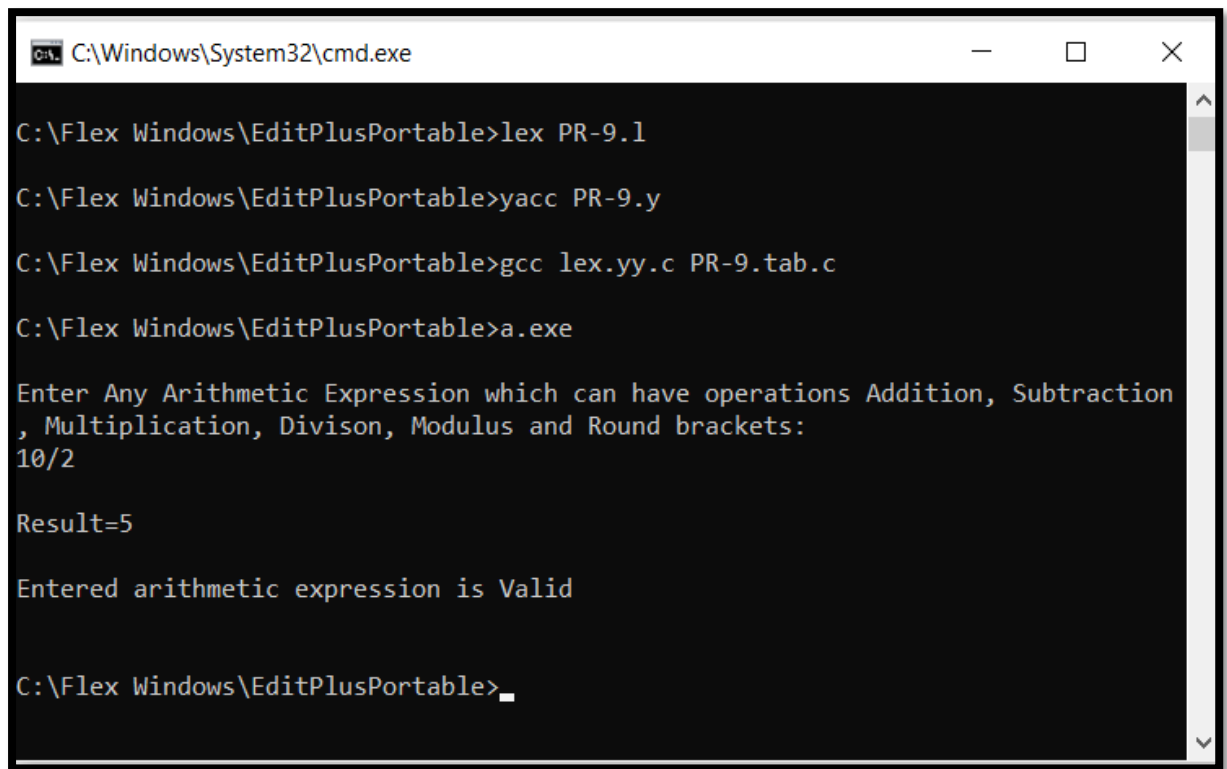
```
%token NUMBER
%left '+' '-'
%left '*' '/' '%'
%left '(' ')'
```

```
%%
ArithmeticExpression: E{
printf("\nResult=%d\n", $$);
return 0;
};
E:E+'E' {$$=$1+$3;}| E-'E' {$$=$1-$3;}| E'*'E {$$=$1*$3;}| E/'E' {$$=$1/$3;}|
E%'E' {$$=$1%$3;}| '('E')' {$$=$2;}| NUMBER {$$=$1;}
%%
```

```
int main()
{
printf("\nEnter Any Arithmetic Expression which can have operations Addition,
Subtraction, Multiplication, Divison, Modulus and Round brackets:\n");
yyvsparse();
if(flag==0)
printf("\nEntered arithmetic expression is Valid\n\n");
return 0;
}
```

```
int yyerror()
{
printf("\nEntered arithmetic expression is Invalid\n\n");
flag=1;
return 0;
}
```

- **OUTPUT**



```
C:\Windows\System32\cmd.exe

C:\Flex Windows\EditPlusPortable>lex PR-9.l
C:\Flex Windows\EditPlusPortable>yacc PR-9.y
C:\Flex Windows\EditPlusPortable>gcc lex.yy.c PR-9.tab.c
C:\Flex Windows\EditPlusPortable>a.exe

Enter Any Arithmetic Expression which can have operations Addition, Subtraction
, Multiplication, Divison, Modulus and Round brackets:
10/2

Result=5

Entered arithmetic expression is Valid

C:\Flex Windows\EditPlusPortable>_
```

Practical – 10

AIM: Finding “Follow” set.

Input: The string consists of grammar symbols.

Output: The Follow set for a given string.

➤ **PR-10.c**

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

int n, m = 0, p, i = 0, j = 0;
char a[10][10], f[10];
void follow(char c);
void first(char c);

int main()
{
    int i, z;
    char c, ch;
    printf("Enter the no of prooductions: ");
    scanf("%d", &n);
    printf("Enter the productions:\n");
    for (i = 0; i < n; i++)
        scanf("%s%c", a[i], &ch);
    do
    {
        printf("Enter the elemets whose follow is to be found:");
        scanf("%c", &c);
        m = 0;
        follow(c);
        printf("Follow(%c)={", c);
        for (i = 0; i < m; i++)
            printf("%c", f[i]);
        printf("}\n");
        printf("Continue(0/1)?");
        scanf("%d%c", &z, &ch);
    } while (z == 1);
    return (0);
}

void first(char c)
{
    int k;
    if (!isupper(c))
        f[m++] = c;
    for (k = 0; k < n; k++)
```

```

    {
        if (a[k][0] == c)
        {
            if (a[k][2] == '$')
                follow(a[k][0]);
            else if (islower(a[k][2]))
                f[m++] = a[k][2];
            else
                first(a[k][2]);
        }
    }
}

void follow(char c)
{
    if (a[0][0] == c)
        f[m++] = '$';
    for (i = 0; i < n; i++)
    {
        for (j = 2; j < strlen(a[i]); j++)
        {
            if (a[i][j] == c)
            {
                if (a[i][j + 1] != '\0')
                    first(a[i][j + 1]);
                if (a[i][j + 1] == '\0' && c != a[i][0])
                    follow(a[i][0]);
            }
        }
    }
}

```

➤ **OUTPUT**

```
PS C:\Flex Windows\EditPlusPortable> cd "c:\Flex Windows\Edit
-10 }
Enter the no of prooductions: 8
Enter the productions:
S=ABCD
A=a
A=$
B=b
C=c
C=$
D=d
D=e
Enter the elemets whose follow is to be found:S
Follow(S)={$}
Continue(0/1)?1
Enter the elemets whose follow is to be found:A
Follow(A)={b}
Continue(0/1)?B
Enter the elemets whose follow is to be found:Follow(B)={cde}

Continue(0/1)?C
Enter the elemets whose follow is to be found:Follow(C)={de}
Continue(0/1)?D
Enter the elemets whose follow is to be found:Follow(D)={$}
Continue(0/1)?0
PS C:\Flex Windows\EditPlusPortable> █
```


Practical – 11

AIM: Implement a C program for constructing LL (1) parsing.

➤ **PR-11.c**

```
#include <stdio.h>
#include <string.h>
char s[20], stack[20];

void main()
{
    char m[5][6][3] = {"tb", " ", " ", "tb", " ", " ", " ", "+tb", " ", " ", "n", "n", "fc", " ", " ", "fc",
    " ", " ", " ", "n", "*fc", "a", "n", "n", "i", " ", " ", "(e)", " ", " "};
    int size[5][6] = {2, 0, 0, 2, 0, 0, 0, 3, 0, 0, 1, 1, 2, 0, 0, 2, 0, 0, 0, 1, 3, 0, 1, 1, 1, 0, 0, 3, 0,
    0};
    int i, j, k, n, str1, str2;
    printf("\n Enter the input string: ");
    scanf("%s", s);
    strcat(s, "$");
    n = strlen(s);
    stack[0] = '$';
    stack[1] = 'e';
    i = 1;
    j = 0;
    printf("\nStack Input\n");
    printf("_____ \n");
    while ((stack[i] != '$') && (s[j] != '$'))
    {
        if (stack[i] == s[j])
        {
            i--;
            j++;
        }
        switch (stack[i])
        {
            case 'e':
                str1 = 0;
                break;
            case 'b':
                str1 = 1;
                break;
            case 't':
                str1 = 2;
                break;
            case 'c':
                str1 = 3;
                break;
```

```

    case 'f':
        str1 = 4;
        break;
    }
    switch (s[j])
    {
    case 'i':
        str2 = 0;
        break;
    case '+':
        str2 = 1;
        break;
    case '*':
        str2 = 2;
        break;
    case '(':
        str2 = 3;
        break;
    case ')':
        str2 = 4;
        break;
    case '$':
        str2 = 5;
        break;
    }
    if (m[str1][str2][0] == '\0')
    {
        printf("\nERROR");
        exit(0);
    }
    else if (m[str1][str2][0] == 'n')
        i--;
    else if (m[str1][str2][0] == 'i')
        stack[i] = 'i';
    else
    {
        for (k = size[str1][str2] - 1; k >= 0; k--)
        {
            stack[i] = m[str1][str2][k];
            i++;
        }
        i--;
    }
    for (k = 0; k <= i; k++)
        printf(" %c", stack[k]);
    printf(" ");
    for (k = j; k <= n; k++)
        printf("%c", s[k]);

```

```

    printf(" \n ");
}
printf("\n SUCCESS");
}

```

➤ **OUTPUT:**

Enter the input string:i*i+i

Stack	INPUT
\$bt	i*i+i\$
\$bcf	i*i+i\$
\$bci	i*i+i\$
\$bc	*i+i\$
\$bcf*	*i+i\$
\$bcf	i+i\$
\$bci	i+i\$
\$bc	+i\$
\$b	+i\$
\$bt+	+i\$
\$bt	i\$
\$bcf	i\$
\$ bci	i\$
\$bc	\$
\$b	\$
\$	\$

success

```
PS C:\Flex Windows\EditPlusPortable> cd "c:\
-11 }
PR-11.c: In function 'main':
PR-11.c:68:13: warning: incompatible implici

Enter the input string: i*i+i

Stack Input
-----
$ b t i*i+i$
$ b c f i*i+i$
$ b c i i*i+i$
$ b c f * *i+i$
$ b c i i+i$
$ b +i$
$ b t + +i$
$ b c f i$
$ b c i i$
$ b $

SUCCESS
PS C:\Flex Windows\EditPlusPortable> 
```

Practical – 12

AIM: Implement a C program to implement LALR parsing.

➤ **PR-12.c**

```
/*LALR PARSER
E->E+T
E->T
T->T*F
T->F
F->(E)
F->i
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void push(char *, int *, char);
char stacktop(char *);
void isproduct(char, char);
int ister(char);
int isnter(char);
int isstate(char);
void error();
void isreduce(char, char);
char pop(char *, int *);
void printt(char *, int *, char[], int);
void rep(char[], int);
struct action
{
    char row[6][5];
};
const struct action A[12] = {
    {"sf", "emp", "emp", "se", "emp", "emp"},
    {"emp", "sg", "emp", "emp", "emp", "acc"},
    {"emp", "rc", "sh", "emp", "rc", "rc"},
    {"emp", "re", "re", "emp", "re", "re"},
    {"sf", "emp", "emp", "se", "emp", "emp"},
    {"emp", "rg", "rg", "emp", "rg", "rg"},
    {"sf", "emp", "emp", "se", "emp", "emp"},
    {"sf", "emp", "emp", "se", "emp", "emp"},
    {"emp", "sg", "emp", "emp", "sl", "emp"},
    {"emp", "rb", "sh", "emp", "rb", "rb"},
    {"emp", "rb", "rd", "emp", "rd", "rd"},
    {"emp", "rf", "rf", "emp", "rf", "rf"}};
struct gotol
{
    char r[3][4];
```

```

};
const struct gotol G[12] = {
    {"b", "c", "d"},
    {"emp", "emp", "emp"},
    {"emp", "emp", "emp"},
    {"emp", "emp", "emp"},
    {"i", "c", "d"},
    {"emp", "emp", "emp"},
    {"emp", "j", "d"},
    {"emp", "emp", "k"},
    {"emp", "emp", "emp"},
    {"emp", "emp", "emp"},
};
char ter[6] = {'i', '+', '*', ')', '(', '$'};
char nter[3] = {'E', 'T', 'F'};
char states[12] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'm', 'j', 'k', 'l'};
char stack[100];
int top = -1;
char temp[10];
struct grammar
{
    char left;
    char right[5];
};
const struct grammar rl[6] = {
    {'E', "e+T"},
    {'E', "T"},
    {'T', "T*F"},
    {'T', "F"},
    {'F', "(E)"},
    {'F', "i"},
};
void main()
{
    char inp[80], x, p, dl[80], y, bl = 'a';
    int i = 0, j, k, l, n, m, c, len;
    printf(" Enter the input :");
    scanf("%s", inp);
    len = strlen(inp);
    inp[len] = '$';
    inp[len + 1] = '\0';
    push(stack, &top, bl);
    printf("\n stack \t\t\t input");
    printt(stack, &top, inp, i);
    do
    {
        x = inp[i];
        p = stacktop(stack);
    }

```

```

isproduct(x, p);
if (strcmp(temp, "emp") == 0)
    error();
if (strcmp(temp, "acc") == 0)
    break;
else
{
    if (temp[0] == 's')
    {
        push(stack, &top, inp[i]);
        push(stack, &top, temp[1]);
        i++;
    }
    else
    {
        if (temp[0] == 'r')
        {
            j = isstate(temp[1]);
            strcpy(temp, rl[j - 2].right);
            dl[0] = rl[j - 2].left;
            dl[1] = '\0';
            n = strlen(temp);
            for (k = 0; k < 2 * n; k++)
                pop(stack, &top);
            for (m = 0; dl[m] != '\0'; m++)
                push(stack, &top, dl[m]);
            l = top;
            y = stack[l - 1];
            isreduce(y, dl[0]);
            for (m = 0; temp[m] != '\0'; m++)
                push(stack, &top, temp[m]);
        }
    }
}
printt(stack, &top, inp, i);
} while (inp[i] != '\0');
if (strcmp(temp, "acc") == 0)
    printf("\n accept the input ");
else
    printf("\n do not accept the input ");
getch();
}

void push(char *s, int *sp, char item)
{
    if (*sp == 100)
        printf(" stack is full ");
    else
    {

```

```

        *sp = *sp + 1;
        s[*sp] = item;
    }
}
char stacktop(char *s)
{
    char i;
    i = s[top];
    return i;
}
void isproduct(char x, char p)
{
    int k, l;
    k = ister(x);
    l = isstate(p);
    strcpy(temp, A[l - 1].row[k - 1]);
}
int ister(char x)
{
    int i;
    for (i = 0; i < 6; i++)
        if (x == ter[i])
            return i + 1;
    return 0;
}
int isnter(char x)
{
    int i;
    for (i = 0; i < 3; i++)
        if (x == nter[i])
            return i + 1;
    return 0;
}
int isstate(char p)
{
    int i;
    for (i = 0; i < 12; i++)
        if (p == states[i])
            return i + 1;
    return 0;
}
void error()
{
    printf(" error in the input ");
    exit(0);
}
void isreduce(char x, char p)
{

```



```

    int k, l;
    k = isstate(x);
    l = isnter(p);
    strcpy(temp, G[k - 1].r[l - 1]);
}
char pop(char *s, int *sp)
{
    char item;
    if (*sp == -1)
        printf(" stack is empty ");
    else
    {
        item = s[*sp];
        *sp = *sp - 1;
    }
    return item;
}
void printt(char *t, int *p, char inp[], int i)
{
    int r;
    printf("\n");
    for (r = 0; r <= *p; r++)
        rep(t, r);
    printf("\t\t\t");
    for (r = i; inp[r] != '\0'; r++)
        printf("%c", inp[r]);
}
void rep(char t[], int r)
{
    char c;
    c = t[r];
    switch (c)
    {
        case 'a':
            printf("0");
            break;
        case 'b':
            printf("1");
            break;
        case 'c':
            printf("2");
            break;
        case 'd':
            printf("3");
            break;
        case 'e':
            printf("4");
            break;
    }
}

```

```

case 'f':
    printf("5");
    break;
case 'g':
    printf("6");
    break;
case 'h':
    printf("7");
    break;
case 'm':
    printf("8");
    break;
case 'j':
    printf("9");
    break;
case 'k':
    printf("10");
    break;
case 'l':
    printf("11");
    break;
default:
    printf("%c", t[r]);
    break;
}
}

```

➤ OUTPUT

Enter the input: i*i+1

Output

Stack	input
0	i*i+i\$
0i5	*i+i\$
0F3	*i+i\$
0T2	*i+i\$
0T2*7	i+i\$
0T2*7i5	+i\$
0T2*7i5F10	+i\$
0T2	+i\$
0E1	+i\$
0E1+6	i\$
0E1+6i5	\$
0E1+6F3	\$
0E1+6T9	\$
0E1	\$

accept the input*/

```

PS C:\Flex Windows\EditPlusPortable> cd "c:\F
lex Windows\EditPlusPortable\" ; if ($?) { gc
c PR-12.c -o PR-12 } ; if ($?) { .\PR-12 }
Enter the input :i*i+i

      stack                input
0      i*i+i$
0i5    *i+i$
0F3    *i+i$
0T2    *i+i$
0T2*7  i+i$
0T2*7i5 +i$
0T2*7F10      +i$
0E1      +i$
0E1+6     i$
0E1+6i5    $
0E1+6F3    $
0E1+6T9    $
0E1        $
accept the input █

```

Practical – 13

AIM: Implement a C program to implement operator precedence parsing.

➤ **PR-13.c**

```
#include<stdio.h>

#include<string.h>

#include<stdlib.h>

char *input;

int i=0;

char lasthandle[6],stack[50],handles[][5]={"E(","E*E","E+E","i","E^E"};

//(E) becomes )E( when pushed to stack

int top=0,l;

char prec[9][9]={

    /*input*/

    /*stack  +  -  *  /  ^  i  (  )  $ */

    /* + */ '>','>','<','<','<','<','<','>','>',

    /* - */ '>','>','<','<','<','<','<','>','>',

    /* * */ '>','>','>','>','<','<','<','>','>',

    /* / */ '>','>','>','>','<','<','<','>','>',

    /* ^ */ '>','>','>','>','<','<','<','>','>',

    /* i */ '>','>','>','>','>','>','e','e','>','>',

    /* ( */ '<','<','<','<','<','<','<','>','e',

    /* ) */ '>','>','>','>','>','>','e','e','>','>',

    /* $ */ '<','<','<','<','<','<','<','<','>',

};

int getindex(char c)

{switch(c)

{   case '+':return 0;
```

```

    case '-':return 1;
    case '*':return 2;
    case '/':return 3;
    case '^':return 4;
    case 'i':return 5;
    case '(':return 6;
    case ')':return 7;
    case '$':return 8;   }
}

int shift()
{
    stack[++top]=*(input+i++);
    stack[top+1]='\0';    }

int reduce()
{
    int i,len,found,t;
    for(i=0;i<5;i++)//selecting handles
    {
        len=strlen(handles[i]);
        if(stack[top]==handles[i][0]&&top+1>=len)
        {
            found=1;
            for(t=0;t<len;t++)
            {
                if(stack[top-t]!=handles[i][t])
                {
                    found=0;
                    break;                }
            }

            if(found==1)
            {
                stack[top-t+1]='E';
                top=top-t+1;
                strcpy(lasthandle,handles[i]);
                stack[top+1]='\0';
            }
        }
    }
}

```

```

        return 1;//successful reduction    }
    }
}

    return 0;    }

void dispstack()
{
    int j;
    for(j=0;j<=top;j++)
        printf("%c",stack[j]);    }

void dispinput()
{
    int j;
    for(j=i;j<l;j++)
        printf("%c",*(input+j));    }

void main()
{
    int j;
    input=(char*)malloc(50*sizeof(char));
    printf("\nEnter the string\n");
    scanf("%s",input);
    input=strcat(input,"$");
    l=strlen(input);
    strcpy(stack,"$");
    printf("\nSTACK\tINPUT\tACTION");
    while(i<=l)
    {
        shift();
        printf("\n");
        dispstack();
        printf("\t");
        dispinput();
        printf("\tShift");

```

```

if(prec[getindex(stack[top])][getindex(input[i])]=='>')
{
    while(reduce())
    {
        printf("\n");
        dispstack();
        printf("\t");
        dispinput();
        printf("\tReduced: E->%s",lasthandle);
    }
}
}

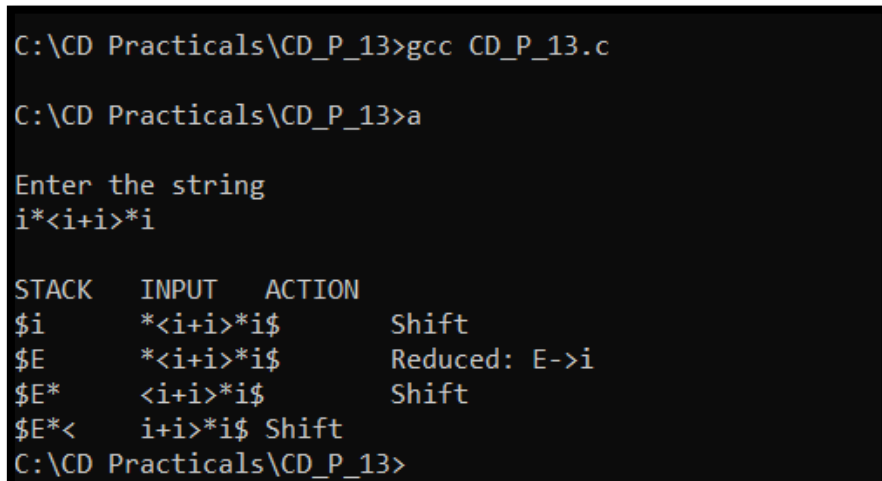
```

```

if(strcmp(stack,"$E$")==0)
    printf("\nAccepted;");
else
    printf("\nNot Accepted;");
}

```

Output Screenshots:



```

C:\CD Practicals\CD_P_13>gcc CD_P_13.c
C:\CD Practicals\CD_P_13>a
Enter the string
i*<i+i>*i
STACK  INPUT  ACTION
$i      *<i+i>*i$  Shift
$E      *<i+i>*i$  Reduced: E->i
$E*     <i+i>*i$  Shift
$E*<   i+i>*i$ Shift
C:\CD Practicals\CD_P_13>

```