Practical 1

Aim: Implementation of Finite Automata and String Validation

Code:

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

int main() {
    char state = 'A'; // Start state
    char str[100];
    printf("Enter input string (consisting of a/b): ");
    scanf("%s", str);

    for (int i = 0; i < strlen(str); i++) {
        char ch = str[i];

        switch (state) {
            case 'A': // Start state
                if (ch == 'a')
                    state = 'B';
                else if (ch == 'b')
                    state = 'A';
                else {
                    printf("Invalid input symbol.\n");
                    return 0;
                }
                break;

            case 'B':
                if (ch == 'a')
                    state = 'B';
                else if (ch == 'b')
                    state = 'C';
                else {
                    printf("Invalid input symbol.\n");
                    return 0;
                }
                break;

            case 'C':
                if (ch == 'a')
                    state = 'B';
                else if (ch == 'b')
                    state = 'A';
                else {
                    printf("Invalid input symbol.\n");
                    return 0;
                }
                break;
        }
    }

    if (state == 'C')
        printf("String is Accepted (ends with 'ab')\n");
    else
        printf("String is Rejected\n");

    return 0;
}
```

Sample Output:

```
Enter input string (consisting of a/b): aabb
String is Accepted (ends with 'ab')
```

Result: Program successfully validates input string using finite automata.

Practical 2

Aim: Introduction to LEX Tool

Code:

```
%{
```

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

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

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

Sample Output:

```
Input: Hello 123
Output:
Word: Hello
Number: 123
```

Result: Program demonstrates the working of Lex tool.

Practical 3(a)

Aim: Generate Histogram of Words using LEX

Code:

```
%{
#include <stdio.h>
#include <string.h>
int count[100]; int idx = 0;
char words[100][50];
%}

%%
[a-zA-Z]+ {
    strcpy(words[idx], yytext);
    count[idx++] = 1;
}
\n ;
. ;
%%

int main() {
    yylex();
    printf("\nWord Histogram:\n");
    for(int i = 0; i < idx; i++) {
        printf("%s: ", words[i]);
        for(int j = 0; j < count[i]; j++) printf("*");
        printf("\n");
    }
    return 0;
}
```

Sample Output:

```
Input: hello world hello
Output:
Word Histogram:
hello: **
world: *
```

Result: Program generates histogram of words using Lex.

Practical 3(b)

Aim: Caesar Cipher using LEX

Code:

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

```
[a-zA-Z]+ {
    for(int i=0; yytext[i]!='\0'; i++) {
        char ch = yytext[i];
        if(ch >= 'a' && ch <= 'z')
            ch = ((ch-'a'+3)%26)+'a';
        else if(ch >= 'A' && ch <= 'Z')
            ch = ((ch-'A'+3)%26)+'A';
        printf("%c", ch);
    }
}
.|\n { printf("%s", yytext); }
%%

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

Sample Output:

```
Input: ABC
Output: DEF
```

Result: Program encrypts text using Caesar Cipher method.

Practical 3(c)

Aim: Extract Comments from C Program using LEX

Code:

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

%%
//.*               { printf("Single-line: %s\n", yytext); }
/\*([^*]|\*+[^*/])*\*+/  { printf("Multi-line: %s\n", yytext); }
.|\n                ;
%%

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

Sample Output:

```
Input: /* comment */
// line
Output:
Multi-line: /* comment */
Single-line: // line
```

Result: Program extracts comments from C source file.

Practical 4(a)

Aim: Convert Roman to Decimal using LEX

Code:

```
%{
#include <stdio.h>
int romanToDec(char *r){
    int val=0;
    for(int i=0; r[i]; i++){
        switch(r[i]){
            case 'I': val+=1; break;
            case 'V': val+=5; break;
            case 'X': val+=10; break;
            case 'L': val+=50; break;
            case 'C': val+=100; break;
            case 'D': val+=500; break;
            case 'M': val+=1000; break;
        }
```

```
    }
    return val;
}
%}

%%
[I|V|X|L|C|D|M]+  { printf("%s = %d\n", yytext, romanToDec(yytext)); }
.|\n              ;
%%

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

Sample Output:

```
Input: X
Output: X = 10
```

Result: Program converts Roman numerals to decimal values.

Practical 4(b)

Aim: Check Whether Statement is Compound or Simple using LEX

Code:

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

%%
"if"|"while"|"for" { printf("Compound Statement\n"); }
[a-zA-Z0-9_ ]+";"  { printf("Simple Statement\n"); }
.|\n               ;
%%

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

Sample Output:

```
Input: if(a>0){a=1;}
Output: Compound Statement
```

Result: Program identifies if a statement is simple or compound.

Practical 4(c)

Aim: Extract HTML Tags using LEX

Code:

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

%%
\<[a-zA-Z0-9/ ]+\> { printf("Tag: %s\n", yytext); }
.|\n               ;
%%

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

Sample Output:

```
Input: <html><body></body></html>
Output:
Tag: <html>
Tag: <body>
```

```
Tag: </body>
Tag: </html>
```

Result: Program extracts HTML tags from input file.

Practical 5

Aim: Implementation of Recursive Descent Parser without Backtracking

Code:

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

char input[100];
int i = 0;

// Function prototypes
void E();
void Eprime();
void T();
void Tprime();
void F();

void error() {
    printf("Error in parsing!\n");
    exit(0);
}

void match(char t) {
    if (input[i] == t)
        i++;
    else
        error();
}

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

void Eprime() {
    if (input[i] == '+') {
        match('+');
        T();
        Eprime();
    }
}

void T() {
    F();
    Tprime();
}

void Tprime() {
    if (input[i] == '*') {
        match('*');
        F();
        Tprime();
    }
}

void F() {
    if (input[i] == '(') {
        match('(');
        E();
        match(')');
    } else if (isalpha(input[i])) {
        match(input[i]);
    } else {
        error();
    }
}

int main() {
```

```
    printf("Enter input expression: ");
    scanf("%s", input);
    E();

    if (input[i] == '\0')
        printf("String is accepted.\n");
    else
        printf("String is not accepted.\n");

    return 0;
}
```

Sample Output:

```
Enter input expression: a+a*a
String is accepted.
```

Result: Program implements recursive descent parser for given grammar.

Practical 6

Aim: Finding FIRST Set

Code:

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

char prod[10][10];
char first[10];

void findFirst(char c) {
    for(int i=0; i<3; i++) {
        if(prod[i][0]==c) {
            if(!isupper(prod[i][2]))
                printf("%c ", prod[i][2]);
            else
                findFirst(prod[i][2]);
        }
    }
}

int main() {
    int n;
    printf("Enter number of productions: ");
    scanf("%d",&n);
    for(int i=0;i<n;i++)
        scanf("%s",prod[i]);
    printf("Enter symbol: ");
    char c; scanf(" %c",&c);
    printf("FIRST(%c) = { ", c);
    findFirst(c);
    printf("}\n");
    return 0;
}
```

Sample Output:

```
Enter number of productions: 3
S=AB
A=a
B=b
Enter symbol: S
FIRST(S) = { a }
```

Result: Program finds FIRST set for given grammar.

Practical 7

Aim: Finding FOLLOW Set

Code:

```
#include <stdio.h>
#include <string.h>
char prod[10][10];
```

```
void follow(char c){
    if(c=='S') printf("$ ");
    for(int i=0;i<3;i++){
        for(int j=0;j<strlen(prod[i]);j++){
            if(prod[i][j]==c && prod[i][j+1]!='\0')
                printf("%c ", prod[i][j+1]);
        }
    }
}

int main(){
    int n; printf("Enter number of productions: ");
    scanf("%d",&n);
    for(int i=0;i<n;i++) scanf("%s",prod[i]);
    char c; printf("Enter symbol: ");
    scanf(" %c",&c);
    printf("FOLLOW(%c) = { ", c);
    follow(c);
    printf("}\n");
    return 0;
}
```

Sample Output:

```
Enter number of productions: 2
S=AB
A=a
Enter symbol: A
FOLLOW(A) = { B }
```

Result: Program finds FOLLOW set for given grammar.

Practical 8

Aim: Generate 3-Tuple Intermediate Code

Code:

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

int main(){
    char exp[50];
    printf("Enter infix expression: ");
    scanf("%s", exp);
    int n=strlen(exp), t=1;
    for(int i=0;i<n;i++){
        if(exp[i]=='+'||exp[i]=='-'||exp[i]=='*'||exp[i]=='/'){
            printf("t%d = %c %c %c\n", t++, exp[i-1], exp[i], exp[i+1]);
        }
    }
    return 0;
}
```

Sample Output:

```
Enter infix expression: a+b*c
t1 = b * c
t2 = a + b
```

Result: Program generates 3-tuple intermediate code for given infix expression.

Practical 9

Aim: YACC Calculator Program

Code:

```
LEX FILE (lex.l)
%{
#include "y.tab.h"
%}

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

```
[+\-*/()]  { return yytext[0]; }
[ \t]   ;
.       ;
%%

YACC FILE (yacc.y)
%{
#include <stdio.h>
#include <stdlib.h>
%}

%token NUMBER
%left '+' '-'
%left '*' '/'

%%
S : E { printf("Result = %d\n", $1); }
  ;
E : E '+' E { $$ = $1 + $3; }
  | E '-' E { $$ = $1 - $3; }
  | E '*' E { $$ = $1 * $3; }
  | E '/' E { $$ = $1 / $3; }
  | '(' E ')' { $$ = $2; }
  | NUMBER { $$ = $1; }
  ;
%%

int main() { return yyparse(); }
int yyerror() { printf("Error\n"); }
```

Sample Output:

```
Input: 2+3*4
Result = 14
```

Result: Program demonstrates YACC-based calculator.

Practical 10

Aim: Operator Precedence Parsing

Code:

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

int main() {
    char exp[50];
    printf("Enter expression: ");
    scanf("%s", exp);
    printf("Parsing Steps:\n");
    for(int i=0; i<strlen(exp); i++) {
        if(exp[i]=='+'||exp[i]=='-'||exp[i]=='*'||exp[i]=='/')
            printf("Reduce %c%c%c\n", exp[i-1], exp[i], exp[i+1]);
    }
    printf("Expression parsed successfully.\n");
    return 0;
}
```

Sample Output:

```
Enter expression: a+b*c
Parsing Steps:
Reduce b*c
Reduce a+b
Expression parsed successfully.
```

Result: Program performs operator precedence parsing.