**Compiler Design Lab Work List**

1. Write a program to implement lexical analyser by using c program.
2. Write a program to convert an expression from Infix to Postfix.
3. Write a program to find the FIRST of a given program.
4. Write a program to find the FOLLOW of a given program.
5. Write a program to implement Recursive Descent Parser of a given grammar.
6. Write a program to implement Shift reduce parser.
7. Write a program to implement of quadruples of a given expression.
8. Construct a DFA from a given regular expression
9. Write a program for LL(1) parser of a given grammar

Write a program to implement lexical analyser by using c program.

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

bool isDelimiter(char ch){
   if (ch == '+' || ch == '-' || ch == '*' ||ch == '/' ||
        ch == ' ' ||ch == ',' || ch == ';' || ch == '>' ||
         ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
         ch == '[' || ch == ']' || ch == '{' || ch == '}')
         return (true);
   return (false);
}

bool isOperator(char ch){
   if (ch == '+' || ch == '-' || ch == '*' ||
         ch == '/' || ch == '>' || ch == '<' ||
         ch == '=')
```

```
        return (true);
    return (false);
}

bool validIdentifier(char* str){
    if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
        str[0] == '3' || str[0] == '4' || str[0] == '5' ||
        str[0] == '6' || str[0] == '7' || str[0] == '8' ||
        str[0] == '9' || isDelimiter(str[0]) == true)
        return (false);
    return (true);
}

bool isKeyword(char* str){
    if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
        !strcmp(str, "break") ||
        !strcmp(str, "continue") || !strcmp(str, "int")
        || !strcmp(str, "double") || !strcmp(str, "float")
        || !strcmp(str, "return") || !strcmp(str, "char")
        || !strcmp(str, "case") || !strcmp(str, "char")
        || !strcmp(str, "sizeof") || !strcmp(str, "long")
        || !strcmp(str, "short") || !strcmp(str, "typedef")
        || !strcmp(str, "switch") || !strcmp(str, "unsigned")
        || !strcmp(str, "void") || !strcmp(str, "static")
        || !strcmp(str, "struct") || !strcmp(str, "goto"))
        return (true);
    return (false);
}

bool isInteger(char* str){
    int i, len = strlen(str);
    if (len == 0)
```

```
        return (false);
    for (i = 0; i < len; i++){
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
                && str[i] != '3' && str[i] != '4' && str[i] != '5'
                && str[i] != '6' && str[i] != '7' && str[i] != '8'
                && str[i] != '9' || (str[i] == '-' && i > 0))
                return (false);
    }
    return (true);
}

bool isRealNumber(char* str){
    int i, len = strlen(str);
    bool hasDecimal = false;
    if (len == 0)
        return (false);
    for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
                && str[i] != '3' && str[i] != '4' && str[i] != '5'
                && str[i] != '6' && str[i] != '7' && str[i] != '8'
                && str[i] != '9' && str[i] != '.' ||
                (str[i] == '-' && i > 0))
                return (false);
        if (str[i] == '.')
                hasDecimal = true;
    }
    return (hasDecimal);
}


char* subString(char* str, int left, int right){
    int i;
    char* subStr = (char*)malloc(
```

```
                        sizeof(char) * (right - left + 2));
        //char subStr[right-left+2];
    for (i = left; i <= right; i++)
         subStr[i - left] = str[i];
        subStr[right - left + 1] = '\0';
    return (subStr);
}

void parse(char str[]){
    int left = 0, right = 0;
    int len = strlen(str);
    while (right <= len && left <= right) {
        if (isDelimiter(str[right]) == false)
            right++;

        if (isDelimiter(str[right]) == true && left == right) {
            if (isOperator(str[right]) == true)
                printf("%c is an operator.\n", str[right]);

            right++;
            left = right;
        } else if (isDelimiter(str[right]) == true && left != right
                || (right == len && left != right)) {
            char* subStr = subString(str, left, right - 1);

            if (isKeyword(subStr) == true)
                printf("%s is a keyword.\n", subStr);

            else if (isInteger(subStr) == true)
                printf("%s is an integer.\n", subStr);

            else if (isRealNumber(subStr) == true)
                printf("%s is a real number.\n", subStr);
```

```
                else if (validIdentifier(subStr) == true
                        && isDelimiter(str[right - 1]) == false)
                    printf("%s is a valid identifier.\n", subStr);

                else if (validIdentifier(subStr) == false
                        && isDelimiter(str[right - 1]) == false)
                    printf("%s is not a valid identifier.\n", subStr);
                left = right;
            }
        }
        return;
}

int main(){
  FILE *file =freopen("file.txt","r+",stdin);
  if (file != NULL ){
        char str[200];
        while(fgets(str,sizeof str, file )!= NULL ){ /* read a line */
        parse(str);
        }
        fclose (file);
  }

}
```

**Write a program to convert an expression from Infix to Postfix.**

Examples of infix to postfix expressions

| Infix Expression | Postfix Expression |
|---|---|
| A + B | A B + |

| | |
|---|---|
| A + B * C | A B C * + |
| (A + B) * C | A B + C * |
| A + B * C + D | A B C * + D + |
| (A + B) * (C + D) | A B + C D + * |
| A * B + C * D | A B * C D * + |
| A + B + C + D | A B + C + D + |
| A+ (B*C-(D/E^F)*G)*H | ABC*DEF^/G*-H*+ |

**Algorithm**

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1. Push "(" onto Stack, and add ")" to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
    2. Add operator to Stack.
    [End of If]
6. If a right parenthesis is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
    2. Remove the left Parenthesis.
    [End of If]
    [End of If]
7. END.

Working process simulation from infix to postfix expression

| Symbol | Scanned | STACK | Postfix Expression | Description |
|---|---|---|---|---|
| 1. | | ( | | Start |
| 2. | A | ( | A | |
| 3. | + | (+ | A | |
| 4. | ( | (+( | A | |
| 5. | B | (+( | AB | |
| 6. | * | (+(* | AB | |
| 7. | C | (+(* | ABC | |
| 8. | - | (+(- | ABC* | '*' is at higher precedence than '-' |
| 9. | ( | (+(-( | ABC* | |
| 10. | D | (+(-( | ABC*D | |
| 11. | / | (+(-(/ | ABC*D | |
| 12. | E | (+(-(/ | ABC*DE | |
| 13. | ^ | (+(-(/^ | ABC*DE | |
| 14. | F | (+(-(/^ | ABC*DEF | |
| 15. | ) | (+(- | ABC*DEF^/ | Pop from top on Stack, that's why '^' Come first |
| 16. | * | (+(-* | ABC*DEF^/ | |
| 17. | G | (+(-* | ABC*DEF^/G | |
| 18. | ) | (+ | ABC*DEF^/G*- | Pop from top on Stack, that's why '^' Come first |
| 19. | * | (+* | ABC*DEF^/G*- | |
| 20. | H | (+* | ABC*DEF^/G*-H | |
| 21. | ) | Empty | ABC*DEF^/G*-H*+ | END |

**Program in C**

```
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<string.h>
#define SIZE 60

char stack[SIZE];
int top = -1;

void push(char item){
        top = top+1;
        stack[top] = item;
}
char pop(){
   char item ;
        item = stack[top];
        top = top-1;
        return item;
}
int is_operator(char symbol){
   if(symbol == '^' || symbol == '*' || symbol == '/' || symbol == '+' || symbol =='-'){
```

```
        return 1;
    }
    else{
        return 0;
    }
}

int precedence(char symbol){
    if(symbol == '^'){
        return 3;
    }
    else if(symbol == '*' || symbol == '/'){
        return 2;
    }
    else if(symbol == '+' || symbol == '-'){
        return 1;
    }
    else{
        return 0;
    }
}

void InfixToPostfix(char infix_exp[], char postfix_exp[]){
    int i, j;
    char item,x;
    push('(');
    strcat(infix_exp,")");

    i=0,j=0;
    item=infix_exp[i];
    while(item != '\0'){
        if(item == '('){
            push(item);
        }
        else if(isdigit(item)||isalpha(item)){
            postfix_exp[j] = item;
            j++;}
        else if(is_operator(item)==1){
            x=pop();
```

```
                while(is_operator(x) == 1 && precedence(x)>= precedence(item)){
                        postfix_exp[j] = x;
                        j++;
                        x = pop();
                }
                push(x);
                push(item);
        }
        else if(item == ')'){
                x = pop();
                while(x != '('){
                        postfix_exp[j] = x;
                        j++;
                        x = pop();
                }
        }
        else{
                printf("\n Invalid infix Expression.\n");
                getchar();
                exit(1);
        }
        i++;
        item = infix_exp[i];
   }
   postfix_exp[j] = '\0';
}
int main(){
   char infix[SIZE],postfix[SIZE];
   while(1){
        printf("Enter an infix expression:\n");
        gets(infix);
        InfixToPostfix(infix,postfix);
        printf("Postfix Expression: ");
        puts(postfix);
   }
   return 0;
}
```

Write a program to find the FIRST of a given program.

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

void findfirst(char,int,int);
char calc_first[20][20],production[20][20],f[20],first[20],ck;
int count,n=0,k,e,ii;

int main(){
        int jm=0,km=0,i;
        char c;
        freopen("input2.txt","r+",stdin);
        scanf("%d",&count);
        getchar();
        for(ii=0;ii<count;ii++){
        gets(production[ii]);
        }
        char done[count];
        int key,ptr=-1;
        int point1=0,point2,xxx;
        for(k=0;k<count; k++){
        c=production[k][0];
        point2=0;
        xxx=0;
        // Checking if First of c has already been calculated
        for(key=0;key<=ptr;key++)
        if(c==done[key])
                xxx=1;

        if (xxx==1)
        continue;
```

```
        findfirst(c,0,0);
        ptr+=1;
        // Adding c to the calculated list
        done[ptr]=c;
        printf("\nFirst(%c) = { ",c);
        calc_first[point1][point2++]=c;
        // Printing the First Sets of the grammar
        for(i=0+jm; i<n; i++){
        int lark=0, chk=0;
        for(lark=0; lark<point2; lark++){
                if (first[i]==calc_first[point1][lark]){
                chk=1;
                break;
                }
        }
        if(chk==0){
                printf("%c ", first[i]);
                calc_first[point1][point2++] = first[i];
        }
        }
        printf("}\n");
        jm = n;
        point1++;
        }
}

void findfirst(char c,int q1,int q2){
        int j;
        if(!(isupper(c))){
        first[n++]=c;
        }
        for(j=0; j<count; j++){
        if(production[j][0]==c){
```

```
        if(production[j][2]=='#'){
                if(production[q1][q2]=='\0')
                first[n++]='#';
                else if(production[q1][q2]!='\0'
                && (q1!=0||q2!=0)){
                findfirst(production[q1][q2],q1,(q2+1));
                }
                else
                first[n++]='#';
        }
        else if(!isupper(production[j][2])){
                first[n++]=production[j][2];
        }
        else{
                findfirst(production[j][2],j,3);
        }
        }
        }
}
```

Write a program to find the FOLLOW of a given program.

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

void followfirst(char, int, int);
void follow(char c);
void findfirst(char, int, int);
// Stores the final result of the First Sets
char calc_first[100][100];
// Stores the final result of the Follow Sets
char calc_follow[100][100];
// Stores the production rules
```

```
char production[100][100];
char f[100],first[100];
char ck;

int count,n=0,m=0,k,e,ii;
int main(){
    int jm = 0,km = 0,i,choice;
    char c, ch;

    freopen("input9.txt","r+",stdin);
        scanf("%d",&count);
        getchar();
        for(ii=0;ii<count;ii++){
        gets(production[ii]);
        }

    int kay;
    char done[count];
    int ptr = -1;

    // Initializing the calc_first array
    for(k = 0; k < count; k++) {
        for(kay = 0; kay < 100; kay++) {
                calc_first[k][kay] = '!';
        }
    }

    int point1 = 0, point2, xxx;
    for(k = 0; k < count; k++){
        c = production[k][0];
        point2 = 0;
        xxx = 0;
// Checking if First of c has already been calculated
```

```
        for(kay = 0; kay <= ptr; kay++)
                if(c == done[kay])
                        xxx = 1;

        if (xxx == 1)
                continue;

        findfirst(c, 0, 0);
        ptr += 1;

        // Adding c to the calculated list
        done[ptr] = c;
        calc_first[point1][point2++] = c;

        // Printing the First Sets of the grammar
        for(i = 0 + jm; i < n; i++) {
                int lark = 0, chk = 0;
                for(lark = 0; lark < point2; lark++) {
                        if (first[i] == calc_first[point1][lark])
                        {
                                chk = 1;
                                break;
                        }
                }
                if(chk == 0)
                {
                        calc_first[point1][point2++] = first[i];
                }
        }
        jm = n;
        point1++;
}
```

```
    char donee[count];
    ptr = -1;
    // Initializing the calc_follow array
    for(k = 0; k < count; k++) {
        for(kay = 0; kay < 100; kay++) {
            calc_follow[k][kay] = '!';
        }
    }
    point1 = 0;
    int land = 0;
    for(e = 0; e < count; e++){
        ck = production[e][0];
        point2 = 0;
        xxx = 0;

//Checking if Follow of ck has already been calculated
        for(kay = 0; kay <= ptr; kay++)
            if(ck == donee[kay])
                xxx = 1;

        if (xxx == 1)
            continue;
        land += 1;

        // Function call
        follow(ck);
        ptr += 1;

        // Adding ck to the calculated list
        donee[ptr] = ck;
        printf(" Follow(%c) = { ", ck);
        calc_follow[point1][point2++] = ck;
```

```
        // Printing the Follow Sets of the grammar
        for(i = 0 + km; i < m; i++) {
                int lark = 0, chk = 0;
                for(lark = 0; lark < point2; lark++)
                {
                        if (f[i] == calc_follow[point1][lark])
                        {
                                chk = 1;
                                break;
                        }
                }
                if(chk == 0)
                {
                        printf("%c ", f[i]);
                        calc_follow[point1][point2++] = f[i];
                }
        }
        printf(" }\n\n");
        km = m;
        point1++;
    }
}

void follow(char c){
    int i, j;

//Adding "$" to the follow set of the start symbol
    if(production[0][0]==c) {
        f[m++]='$';
    }
    for(i = 0; i <100; i++){
        for(j = 2;j <100; j++){
                if(production[i][j] == c){
```

```
                        if(production[i][j+1] != '\0'){
//Calculate the first of the next Non-Terminal in the production
                                followfirst(production[i][j+1], i, (j+2));
                        }

                        if(production[i][j+1]=='\0' && c!=production[i][0]){
                                // Calculate the follow of the Non-Terminal
                                // in the L.H.S. of the production
                                follow(production[i][0]);
                        }
                }
        }
    }
}

void findfirst(char c, int q1, int q2){
    int j;
    if(!(isupper(c))) {
        first[n++] = c;
    }
    for(j = 0; j < count; j++){
        if(production[j][0] == c){
            if(production[j][2] == '#'){
                if(production[q1][q2] == '\0')
                    first[n++] = '#';
                else if(production[q1][q2] != '\0'
                            && (q1 != 0 || q2 != 0)){
                    findfirst(production[q1][q2], q1, (q2+1));
                }
                else
                    first[n++] = '#';
            }
            else if(!isupper(production[j][2])){
```

```
                        first[n++] = production[j][2];
                }
                else{
                        findfirst(production[j][2], j, 3);
                }
        }
    }
}

void followfirst(char c, int c1, int c2){
    int k;
    // The case where we encounter a Terminal
    if(!(isupper(c)))
        f[m++] = c;
    else{
        int i = 0, j = 1;
        for(i = 0; i < count; i++){
                if(calc_first[i][0] == c)
                        break;
        }
//Including the First set of the Non-Terminal in the Follow of
// the original query
        while(calc_first[i][j] != '!'){
                if(calc_first[i][j] != '#'){
                        f[m++] = calc_first[i][j];
                }
                else{
                        if(production[c1][c2] == '\0'){
//Case where we reach the end of a production
                                follow(production[c1][0]);
                        }
                        else{
//Recursion to the next symbol in case we encounter a "#"
```

```
                    followfirst(production[c1][c2],c1,c2+1);
                }
            }
            j++;
        }
    }
}
```

**Write a program to implement Recursive Descent Parser of a given grammar.**

A recursive descent parser is a top-down parser. It is used to build a parse tree from top to bottom and reads the input from left to right.The parser gets an input and reads it from left to right and checks it. If the source code fails to parse properly, then the parser exits by giving an error (flag) message. If it parses the source code properly then it exits without giving an error message.It needs no backtracking.It uses procedures for every non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking.

RD parser will verify whether the syntax of the input stream is correct by checking each character from left to right. A basic operation necessary is reading characters from the input stream and matching then with terminals from the grammar that describes the syntax of the input.

To implement Recursive Descent parsing preconditions are

- Grammar must be Unambiguous .
- Grammar must be Left recursive free.

The grammar on which we are going to do recursive descent parsing is:

**E -> E+T | T**

**T -> T*F | F**

**F -> (E) | id**

After elimination the left recursion we get

**E -> TE'**

**E' -> +TE' | ε**

**T -> FT'**

**T' -> *FT' | ε**

**F -> (E) | id**

**NB.we replace id with a**

The given grammar can accept all arithmetic equations involving +, * and ()

a+(a*a) a+a*a , (a), a , a+a+a*a+a.... etc are accepted

a++a, a***a, +a, a*, ((a . . . etc are rejected.

**Program in C**

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>
char input[100];
int i,error,length;
void E();
void T();
void E_desh();
void T_desh();
void F();

int main(){
        printf("Enter a string: ");
        while(1){
        i=0,error=0;
        scanf("%s",input);
        length=strlen(input);
        E();
        if(length==i&&error==0)
        printf("\nGiven string is accepted.\n");
        else
        printf("\nGiven string is rejected.\n");
        }
}

void E(){
        T();
        E_desh();
}
void E_desh(){
        if(input[i]=='+'){
        i++;
        T();
        E_desh();
        }
}
void T(){
```

```
        F();
        T_desh();
}
void T_desh(){
        if(input[i]=='*'){
        i++;
        F();
        T_desh();
        }
}



void F(){
        if((input[i])=='a')
        i++;
        else if(input[i]=='('){
        i++;
        E();
        if(input[i]==')')
        i++;
        else
        error=1;
  }

        else
        error=1;
}
```

## Write a program to implement Shift reduce parser.

```
//This program works only for same Non-terminal
#include<stdio.h>
#include<string.h>
int i=0,j=0,k=0,z=0,length=0;
char stk[15],input[16],act[10],ac[20];
void check();
int main(){
        printf("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id\n");
        printf("Enter input string:");
```

```
        scanf("%s",&input);
        length=strlen(input);

        strcpy(act,"SHIFT->");
        printf("Stack \t Input \t\t Action\n");
        printf("$ \t %s$ \t\t ...",input);
        for(k=0,i=0; j<length; k++,i++,j++){
        if(input[j]=='i' && input[j+1]=='d'){
        stk[i]=input[j];
        stk[i+1]=input[j+1];
        stk[i+2]='\0';
        input[j]=' ';
        input[j+1]=' ';
        printf("\n$%s\t%s$\t%sid",stk,input,act);
        check();
        }
        else{
        stk[i]=input[j];
        stk[i+1]='\0';
        input[j]=' ';
        printf("\n$%s\t%s$\t%ssymbols",stk,input,act);
        check();
        }
        }

}
void check(){
        strcpy(ac,"REDUCE TO E->");

        for(z=0;z<length;z++){
        if(stk[z]=='i' && stk[z+1]=='d'){
        stk[z]='E';
        stk[z+1]='\0';
        printf("\n$%s\t%s$\t%s",stk,input,ac);
        j++;
        }
        }

        for(z=0; z<length; z++){
```

```
        if(stk[z]=='E' && stk[z+1]=='-' && stk[z+2]=='E'){
        stk[z]='E';
        stk[z+1]='\0';
        stk[z+2]='\0';
        printf("\n$%s\t%s$\t%s",stk,input,ac);
        i=i-2;
        }
        }
        for(z=0; z<length; z++){
        if(stk[z]=='E' && stk[z+1]=='*' && stk[z+2]=='E'){
        stk[z]='E';
        stk[z+1]='\0';
        stk[z+1]='\0';
        printf("\n$%s\t%s$\t%s",stk,input,ac);
        i=i-2;
        }
        }
}
```

## Write a program to implement of quadruples of a given expression.

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

int i=1,j=0,no=0,tmpch=90;
char str[100],left[15],right[15];
void findopr();
void explore();
void fleft(int);
void fright(int);
struct exp{
   int pos;
   char op;
}k[15];
struct quard{
        char op;
        char arg1;
        char arg2;
```

```
        char result;
};
void main(){
   printf("Enter the Expression :");
   gets(str);
   findopr();
   explore();
   getch();
}
void findopr(){
   for(i=0;str[i]!='\0';i++)
         if(str[i]=='/'){
        k[j].pos=i;
        k[j++].op='/';
         }
   for(i=0;str[i]!='\0';i++)
         if(str[i]=='*'){
        k[j].pos=i;
        k[j++].op='*';
         }
   for(i=0;str[i]!='\0';i++)
         if(str[i]=='+'){
        k[j].pos=i;
        k[j++].op='+';
         }
   for(i=0;str[i]!='\0';i++)
         if(str[i]=='-'){
        k[j].pos=i;
        k[j++].op='-';
         }
}

void explore(){
   i=0;
   int xx=0;
   printf(" OP\t\arg1\targ2\tresult\n");
   while(k[i].op!='\0'){
         fleft(k[i].pos);
         fright(k[i].pos);
```

```
            str[k[i].pos]=tmpch--;
            printf("%c\t%s\t%s\t%c",k[i].op,left,right,str[k[i].pos]);
            printf("\n");
            i++;
    }
    fright(-1);
    if(no==0)
    {
            fleft(strlen(str));
            printf("=\t%s\t\t%s\n",left,right);
            getch();
            exit(0);
    }
    printf("=\t%s\t\t%s\n",left,right);
    getch();
}
void fleft(int x){
    int w=0,flag=0,pp=x;
    x--;
                                        while(x!=  -1  &&str[x]!=  '+'
&&str[x]!='*'&&str[x]!='='&&str[x]!='\0'&&str[x]!='-'&&str[x]!='/'&&str[x]!=':'){
            if(str[x]!='$'&& flag==0){
           left[w++]=str[x];
            left[w]='\0';
            str[x]='$';
            flag=1;
             }
             x--;
    }
}
void fright(int x){
    int w=0,flag=0;
    x++;
                                        while(str[x]!=  '\0'  &&  str[x]!=
'+'&&str[x]!='*'&&str[x]!='\0'&&str[x]!='='&&str[x]!=':'&&str[x]!='-'&&str[x]!='/'){
            if(str[x]!='$'&& flag==0){
            right[w++]=str[x];
            right[w]='\0';
            str[x]='$';
```

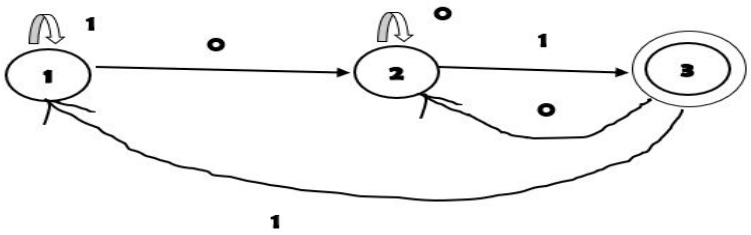
```
        flag=1;
        }
        x++;
    }
}
```

## Construct a DFA from a given regular expression

Procedure for solving--
Given: Regular Expression
Step 1: Write language, valid and invalid strings.
Step 2: Draw Dfa of given regular expression
Step 3: Construct transition table of Dfa
Step 4: C Program implementation

**Given : Regular Expression ((0 +1)* 01)**

Step 1
Language
{01,001,101,0001,0101,1001,1101,00001,00101,01001,01101,10001,10101,11001,11101,..... }
Minimum strings = { 01,001,101}
Few examples of invalid strings ={ 000,11100,01010}
Step 2
Draw Dfa of given regular expression




**Step 3**
Construct transition table of Dfa

| delta | 0 | 1 |
|---|---|---|
| 1 [initial] | 2 | 1 |
| 2 | 2 | 3 |
| 3 [final] | 2 | 1 |

**Step 4**

C program implementation

```
#include<stdio.h>
#include<string.h>
int main(){
        while(1){
        char str[20],temp;
        int len,state,c;
        printf("Enter the string to check\n");
        gets(str);
        len=strlen(str);

        c=0;
        state=1; // initial state
        while(c<len){
        temp=str[c];
        switch(state){
        case 1:
                if(temp=='0')
                state=2;

                if(temp=='1')
                state=1;
                break;
```

```
        case 2:
                if(temp=='0')
                state=2;
                if(temp=='1')
                state=3;
                break;

        case 3:
                if(temp=='0')
                state=2;
                if(temp=='1')
                state=1;
                break;
        }
        c++;
        }

        if(state==3)
        printf("Valid string.\n");
        else
        printf("InValid string.\n");
        }
}
```

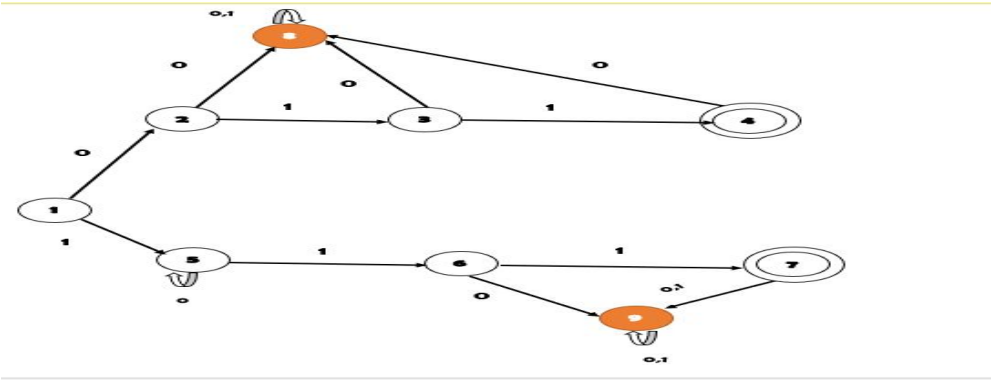
**Given : Regular Expression (01* +10*) 11**

**Step 1**

Language L = { 011,111,0111111111,10000011,1011 }
Minimum strings = { 011,111}
Few examples of invalid strings ={ 000,11100,01010}

**Step 2**

Draw Dfa of given regular expression



**Step 3**

construct transition table of Dfa

| delta | 0 | 1 |
|---|---|---|
| 1 [initial] | 2 | 5 |
| 2 | 8 | 3 |
| 3 | 8 | 4 |
| 4 [final] | 8 | 4 |
| 5 | 5 | 6 |
| 6 | 9 | 7 |
| 7 [final] | 9 | 9 |
| 8 | 8 | 8 |
| 9 | 9 | 9 |

**Step 4**

**C program implementation**

```
#include<stdio.h>
#include<string.h>
int main(){
        while(1){
        char str[20],temp;
        int len,state,c;
        printf("Enter the string to check\n");
        gets(str);
```

```
len=strlen(str);

c=0;
state=1; // initial state
while(c<len){
temp=str[c];
switch(state){
case 1:
        if(temp=='0')
        state=2;

        if(temp=='1')
        state=5;
        break;

case 2:
        if(temp=='0')
        state=8;
        if(temp=='1')
        state=3;
        break;

case 3:
        if(temp=='0')
        state=8;
        if(temp=='1')
        state=4;
        break;
case 4:
        if(temp=='0')
        state=8;
        if(temp=='1')
        state=4;
        break;

case 5:
        if(temp=='0')
        state=5;
        if(temp=='1')
```

```
            state=6;
            break;

        case 6:
            if(temp=='0')
            state=9;
            if(temp=='1')
            state=7;
            break;

        case 7:
            if(temp=='0')
            state=9;
            if(temp=='1')
            state=9;
            break;
        case 8:
            if(temp=='0')
            state=8;
            if(temp=='1')
            state=8;
            break;
        case 9:
            if(temp=='0')
            state=9;
            if(temp=='1')
            state=9;
            break;
        }
        c++;
        }

        if(state==4||state==7)
        printf("Valid string.\n");
        else
        printf("InValid string.\n");
        }
}
```