EX NO: 1 **Implementation of Lexical Analyzer**
Date:

Aim:

To develop a lexical analyzer capable of tokenizing a subset of a programming language, identifying tokens such as identifiers, keywords, literals, and punctuation symbols.

Algorithm:

1. **Start**
2. **Input**: Take the source code as input.
3. **Initialize**: Initialize an empty list/array to store tokens.
4. **Tokenization Loop**: Iterate through each character in the source code.
    - If the character is a whitespace, skip it.
    - If the character is a digit, start collecting characters until a non-digit or non-decimal point character is encountered. Add the collected characters as a token of type "NUMBER" to the token list.
    - If the character is an alphabet or underscore, start collecting characters until a non-alphanumeric or underscore character is encountered. Add the collected characters as a token of type "IDENTIFIER" to the token list.
    - If the character is an operator or delimiter (e.g., +, -, *, /, =, (,), etc.), add it as a token to the token list.
5. **Output**: Return the list of tokens.
6. **End**

Source Code:

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

#define MAX_CODE_LENGTH 1000
#define MAX_TOKEN_LENGTH 100

// Token types
typedef enum {
   NUMBER,
   IDENTIFIER,
```

```
    ASSIGN,
    PLUS,
    MINUS,
    MULTIPLY,
    DIVIDE,
    LPAREN,
    RPAREN,
    NEWLINE,
    WHITESPACE,
    END_OF_FILE
} TokenType;

// Token structure
typedef struct {
    TokenType type;
    char value[MAX_TOKEN_LENGTH];
} Token;

// Function to tokenize the input code
Token* tokenize(const char* code) {
    Token* tokens = malloc(strlen(code) * sizeof(Token));
    if (!tokens) {
        fprintf(stderr, "Memory allocation error\n");
        exit(EXIT_FAILURE);
    }

    int i = 0;
    while (*code) {
        if (isspace(*code)) {
            code++;
            continue;  // Skip whitespace
        }
        if (isdigit(*code)) {
            char* start = code;
            while (isdigit(*code) || *code == '.') {
                code++;
            }
            int len = code - start;
            tokens[i].type = NUMBER;
```

```
            strncpy(tokens[i].value, start, len);
            tokens[i].value[len] = '\0';  // Null-terminate string
        } else if (isalpha(*code) || *code == '_') {
            char* start = code;
            while (isalnum(*code) || *code == '_') {
                code++;
            }
            int len = code - start;
            tokens[i].type = IDENTIFIER;
            strncpy(tokens[i].value, start, len);
            tokens[i].value[len] = '\0';  // Null-terminate string
        } else {
            switch (*code) {
                case '=':
                    tokens[i].type = ASSIGN;
                    strncpy(tokens[i].value, "=", 1);
                    break;
                case '+':
                    tokens[i].type = PLUS;
                    strncpy(tokens[i].value, "+", 1);
                    break;
                case '-':
                    tokens[i].type = MINUS;
                    strncpy(tokens[i].value, "-", 1);
                    break;
                case '*':
                    tokens[i].type = MULTIPLY;
                    strncpy(tokens[i].value, "*", 1);
                    break;
                case '/':
                    tokens[i].type = DIVIDE;
                    strncpy(tokens[i].value, "/", 1);
                    break;
                case '(':
                    tokens[i].type = LPAREN;
                    strncpy(tokens[i].value, "(", 1);
                    break;
                case ')':
                    tokens[i].type = RPAREN;
```

```
                strncpy(tokens[i].value, ")", 1);
                break;
            case '\n':
                tokens[i].type = NEWLINE;
                strncpy(tokens[i].value, "\n", 1);
                break;
            default:
                fprintf(stderr, "Illegal character: %c\n", *code);
                exit(EXIT_FAILURE);
        }
        code++;
        tokens[i].value[1] = '\0';  // Null-terminate string
    }
    i++;
  }

  tokens[i].type = END_OF_FILE;
  tokens[i].value[0] = '\0';  // Null-terminate string

  return tokens;
}

// Function to free memory allocated for tokens
void free_tokens(Token* tokens) {
  free(tokens);
}

// Function to print token type
const char* token_type_to_string(TokenType type) {
  switch(type) {
    case NUMBER: return "NUMBER";
    case IDENTIFIER: return "IDENTIFIER";
    case ASSIGN: return "ASSIGN";
    case PLUS: return "PLUS";
    case MINUS: return "MINUS";
    case MULTIPLY: return "MULTIPLY";
    case DIVIDE: return "DIVIDE";
    case LPAREN: return "LPAREN";
    case RPAREN: return "RPAREN";
```

```
        case NEWLINE: return "NEWLINE";
        case WHITESPACE: return "WHITESPACE";
        case END_OF_FILE: return "END_OF_FILE";
        default: return "UNKNOWN";
    }
}

// Test the lexer
int main() {
    char code[MAX_CODE_LENGTH];
    printf("Enter code (max %d characters):\n", MAX_CODE_LENGTH - 1);
    if (fgets(code, MAX_CODE_LENGTH, stdin) == NULL) {
        fprintf(stderr, "Failed to read input\n");
        return EXIT_FAILURE;
    }

    Token* tokens = tokenize(code);
    for (int i = 0; tokens[i].type != END_OF_FILE; i++) {
        printf("Token: Type=%s, Value=%s\n", token_type_to_string(tokens[i].type),
tokens[i].value);
    }
    free_tokens(tokens);

    return EXIT_SUCCESS;
}
```

Output:

```
Enter code (max 999 characters):
x = 10 + 20 y = x * 2
Token: Type=IDENTIFIER, Value=x
Token: Type=ASSIGN, Value==
Token: Type=NUMBER, Value=10
Token: Type=PLUS, Value=+
Token: Type=NUMBER, Value=20
Token: Type=IDENTIFIER, Value=y
Token: Type=ASSIGN, Value==
Token: Type=IDENTIFIER, Value=x
Token: Type=MULTIPLY, Value=*
Token: Type=NUMBER, Value=2
```

Result:

Thus, a program a lexical analyzer capable of tokenizing a subset of a programming language, identifying tokens such as identifiers, keywords, literals, and punctuation symbols was implemented

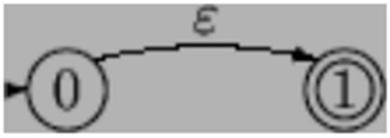
EX NO: 2 **Conversion from Regular Expression to NFA**
Date:

Aim:

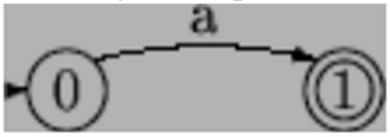
Program to convert Regular Expression (R.E.) to Non-Deterministic Finite Automata (N.F.A.)

Algorithm:

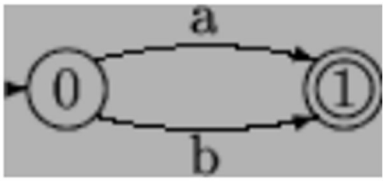
1. The NFA representing the empty string is:



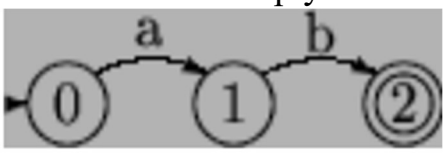
2. If the regular expression is just a character, eg. a, then the corresponding NFA is :



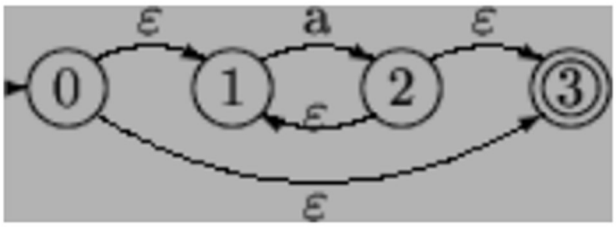
3. The union operator is represented by a choice of transitions from a node; thus a|b can be represented as:



4. Concatenation simply involves connecting one NFA to the other; eg. ab is:



5. The Kleene closure must allow for taking zero or more instances of the letter from the input; thus a* looks like:



Source Code:

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

int ret[100];
static int pos = 0;
static int sc = 0;

void nfa(int st, int p, char *s) {
    int i,sp,fs[15],fsc=0;
  sp=st;pos=p;sc=st;
  while(*s!=NULL)
  {
  if(isalpha(*s))
  {
  ret[pos++]=sp;
  ret[pos++]=*s;
  ret[pos++]=++sc;}
  if(*s=='.')
  {
  sp=sc;
  ret[pos++]=sc;
  ret[pos++]=238;
  ret[pos++]=++sc;
  sp=sc;}
 if(*s=='|')
  {
  sp=st;
  fs[fsc++]=sc;}
 if(*s=='*')
  {
  ret[pos++]=sc;
  ret[pos++]=238;
  ret[pos++]=sp;
ret[pos++]=sp;
   ret[pos++]=238;
   ret[pos++]=sc;
```

```
   }
  if (*s=='(')
  {char ps[50];
   int i=0,flag=1;
   s++;
     while(flag!=0)
   {ps[i++]=*s;
    if (*s=='(')
    flag++;
    if (*s==')')
    flag--;
    s++;}
    ps[--i]='\0';
    nfa(sc,pos,ps);
    s--;
  }
  s++;
 }
 sc++;
   for(i=0;i<fsc;i++)
   {ret[pos++]=fs[i];
    ret[pos++]=238;
    ret[pos++]=sc;
    }
    ret[pos++]=sc-1;
    ret[pos++]=238;
    ret[pos++]=sc;
}

int main() {
    int i;
    char inp[50]; // Changed from pointer to array

    printf("Enter the regular expression: ");
    fgets(inp, sizeof(inp), stdin);
    inp[strcspn(inp, "\n")] = '\0';  // Remove newline character if present

    nfa(1, 0, inp);
```

```
    printf("\nState  Input  State\n");
    for (i = 0; i < pos; i = i + 3)
        printf("%d     --%c-->      %d\n", ret[i], ret[i + 1], ret[i + 2]);

    return 0;
}
```

Output:

```
Enter the regular expression: a+b*

State  Input  State
1      --a-->      2
1      --b-->      3
3      --ε-->      1
1      --ε-->      3
3      --ε-->      4

Process returned 0 (0x0)   execution time : 16.745 s
Press any key to continue.
```

Result:

Thus, a program to convert Regular Expression (R.E.) to Non-Deterministic Finite Automata (N.F.A.) was implemented

EX NO: 3 **Conversion from NFA to DFA**
Date:

Aim:

Program to convert NFA to Deterministic Finite Automata(D.F.A.)

Algorithm:

1. Convert into NFA using above rules for operators (union, concatenation and closure) and precedence.
2. Find Ɛ -closure of all states.
3. Start with epsilon closure of start state of NFA.
4. Apply the input symbols and find its epsilon closure. Dtran[state, input symbol] = Ɛ -closure(move(state, input symbol)) where Dtran transition function of DFA
5. Analyze the output state to find whether it is a new state.
6. If new state is found, repeat step 4 and step 5 until no more new states are found.
7. Construct the transition table for Dtran function.
8. Draw the transition diagram with start state as the Ɛ -closure (start state of NFA) and final state is the state that contains final state of NFA drawn.

Source Code:

```
#include<stdio.h>
#include<conio.h>
#define MAX 20

//============================================================
struct nfa_state
{
 int a, b, eps1, eps2;
}NFA[20];

struct dfa_state
{int state[20],a[20],b[20];
}DFA[20];

int cur, initial_state, final_state;

int stack[MAX];
int top;
```

```
//=============================================================
void push(int val)
{
 stack[++top]=val;
}

int pop()
{
 return stack[top--];}
//=============================================================
int priority(char op)
{
 switch(op)
 {
 case '+': return 1;
 case '.': return 2;
 case '*':   return 3;
 }return 0;
}
//=============================================================
void init_nfa_table()
{
 int i;
 for(i=0; i<20; i++)
 {
  NFA[i].a = NFA[i].b = -1;
  NFA[i].eps1 = NFA[i].eps2 = -1;
 }
}
//=============================================================
void symbol(char c)
{
 if(c=='a')
  NFA[cur].a = cur+1;

 if(c=='b')
  NFA[cur].b = cur+1;push(cur);
 push(cur+1);
```

```
 cur += 2;
}//============================================================
void concat()
{
 int first1, first2, last1, last2;

 last2 = pop();
 first2 = pop();
 last1 = pop();
 first1 = pop();

 NFA[last1].eps1 = first2;

 push(first1);
 push(last2);
}
//============================================================
void parallel()
{
 int first1, first2, last1, last2;last2 = pop();
 first2 = pop();
 last1 = pop();
 first1 = pop();

 NFA[cur].eps1 = first1;
NFA[cur].eps2 = first2;
 NFA[last1].eps1 = cur+1;
 NFA[last2].eps2 = cur+1;

 push(cur);
 push(cur+1);
 cur += 2;
}
//============================================================
void closure()
{
 int first,last;last = pop();
 first = pop();
```

```
 NFA[cur].eps1 = first;
 NFA[cur].eps2 = cur+1;
 NFA[last].eps1 = first;
 NFA[last].eps2 = cur+1;

 push(cur);
 push(cur+1);

 cur += 2;
}//=============================================================
void construct_nfa(char *postfix)
{
 int i=0;

 top=-1;

 for(i=0; postfix[i]!='\0'; i++)
 {
  switch(postfix[i])
  {
  case 'a':
  case 'b': symbol(postfix[i]);
     break;
  case '.': concat();
     break;
  case '+':  parallel();
     break;
  case '*': closure(); }
 }
 final_state = pop();
 initial_state = pop();
}
//=============================================================
void disp_NFA(){
 int i;
 printf("\nstate\ta\tb\t?");
 for(i=0;i<cur;i++)
 {
  if(i==initial_state)
```

```
  printf("\n->%d",i);
 else
  if(i==final_state)
  printf("\n* %d",i);
 else
  printf("\n  %d",i);

 if(NFA[i].a==-1)
  printf("\t-");
 else
  printf("\t{%d}",NFA[i].a);

 if(NFA[i].b==-1)printf("\t-");
 else
  printf("\t{%d}",NFA[i].b);

 if(NFA[i].eps1!=-1)
  {
  printf("\t{%d",NFA[i].eps1);if(NFA[i].eps2!=-1)
   {
    printf(",%d",NFA[i].eps2);
   }
  printf("}");
  }
 else
  printf("\t-");
 }
}
//=============================================================
void init_dfa_table()
{
 int i,j;
 for(i=0;i<20;i++)
 {
  for(j=0;j<20;j++)
  {
   DFA[i].state[j]=-1;
   DFA[i].a[j]=-1;
   DFA[i].b[j]=-1;
```

```
  }
 }
}
//=================================================================
void print_state(int t[])
{
 int i=0;
 printf("[");
 for(i=0;t[i]!=-1;i++)
  printf("%d,",t[i]);
 printf("\b]");
}
//=================================================================
int isPresent(int T[], int v)
{
 int i;
 for(i=0;T[i]!=-1;i++)
  if(T[i]==v)
   return 1;
 return 0;
} //=================================================================
void disp_DFA(int n)
{
 int i;
 printf("\nstate\t\t\ta\t\t\tb");
 for(i=0;i<=n;i++)
 {
  printf("\n");
  if(i==0)
   printf("->");
if(isPresent(DFA[i].state,final_state))
   printf("*");

   print_state(DFA[i].state);
   printf("\t\t");

  if(DFA[i].a[0]!=-1)
   print_state(DFA[i].a);
  else
```

```
   printf("\t-");
  printf("\t\t");
  if(DFA[i].b[0]!=-1)
   print_state(DFA[i].b);
  else
   printf("\t-");
 }
}
//=============================================================
void epsilon_closure(int T[], int t[])
{
 int i,v;
 top=-1;

 for(i=0;t[i]!=-1;i++)
  push(t[i]);
i=0;

 while(top!=-1)
 {
  v = pop();

  if(isPresent(T,v)==0)
  {
   T[i++]=v;
  }

  if(NFA[v].eps1!=-1)
  {
   push(NFA[v].eps1);
  }

  if(NFA[v].eps2!=-1)
  {
   push(NFA[v].eps2);
  }
 }
}
//=============================================================
```

```
void init_t(int t[])
{int i;
 for(i=0;i<20;i++)
  t[i]=-1;
}
//=============================================================
int search(int n,int t2[])
{
 int i,j;
 for(i=0;i<=n;i++)
 {
  for(j=0;t2[j]!=-1;j++)
   if(isPresent(DFA[i].state,t2[j])==0)
    break;
  if(t2[j]==-1)
   return 1;
 }
 return 0;
}
//=============================================================
void copy(int t1[], int t2[])
{
 int i;
 for(i=0;t2[i]!=-1;i++)
  t1[i]=t2[i];
}
//=============================================================
void main()
{
 char postfix[20];
 int t[20],v;
 int n=0,i=0,j,k;

 system("cls");

 printf("\nEnter Regular Expression: ");
 scanf("%s",postfix);
```

```
printf("\nPostfix Expression: %s",postfix);
getch();

init_nfa_table();
construct_nfa(postfix);
system("cls");

disp_NFA();
getch();

init_dfa_table();
init_t(t);t[0]=initial_state;
epsilon_closure(DFA[0].state,t);


init_t(t);

for(j=0,k=0; DFA[0].state[j]!=-1 ; j++)
{
 v = DFA[0].state[j];

 if(NFA[v].a!=-1)
 {
  if(isPresent(t,NFA[v].a)==0)
   t[k++]=NFA[v].a;
 }
}

epsilon_closure(DFA[0].a,t);

init_t(t);

for(j=0,k=0;DFA[0].state[j]!=-1;j++)
{
 v = DFA[0].state[j];if(NFA[v].b!=-1)
 {if(isPresent(t,NFA[v].b)==0)
   t[k++]=NFA[v].b;
 }
}
```

```
epsilon_closure(DFA[0].b,t);

for(i=0;i<=n;i++)
{
 if( search( n , DFA[i].a)==0 )
 {
  n++;
  copy(DFA[n].state,DFA[i].a);

  init_t(t);

  for( j=0,k=0; DFA[n].state[j]!=-1 ; j++)
  {
   v = DFA[n].state[j];

   if(NFA[v].a!=-1)
   {
    if(isPresent(t,NFA[v].a)==0)
     t[k++]=NFA[v].a;
   }
  } epsilon_closure(DFA[n].a,t);

  init_t(t);

  for(j=0,k=0;DFA[n].state[j]!=-1;j++)
  {
   v = DFA[n].state[j];
   if(NFA[v].b!=-1)
   {
    if(isPresent(t,NFA[v].b)==0)
     t[k++]=NFA[v].b;
   }
  }
  epsilon_closure(DFA[n].b,t);

 }

 if( search( n , DFA[i].b ) ==0)
 {
```

```
   n++;
   copy(DFA[n].state,DFA[i].b);

   init_t(t);
   for( j=0,k=0; DFA[n].state[j]!=-1 ; j++)
   {
    v = DFA[n].state[j];
if( NFA[v].a!=-1)
    {
     if(isPresent(t,NFA[v].a)==0)
      t[k++]=NFA[v].a;
    }
   }
   epsilon_closure(DFA[n].a,t);

   init_t(t);
   for(j=0,k=0;DFA[n].state[j]!=-1;j++)
   {
    v = DFA[n].state[j];
    if(NFA[v].b!=-1)

    {
     if(isPresent(t,NFA[v].b)==0)
      t[k++]=NFA[v].b;
    }
   }
   epsilon_closure(DFA[n].b,t);}
 }
 disp_DFA(n);
   getch();
}
```

OUTPUT:

```
Enter Regular Expression: ab+*

Postfix Expression: ab+*
```

```
state   a       b       ?
  0     {1}     -       -
  1     -       -       {5}
  2     -       {3}     -
  3     -       -       -
  4     -       -       {0,2}
  5     -       -       {4,7}
->6     -       -       {4,7}
* 7     -       -       -
state                   a                       b
->*[6,7,4,2,0]          [1,5,7,4,2,0]           [3,5,7,4,2,0]
*[1,5,7,4,2,0]          [1,5,7,4,2,0]           [3,5,7,4,2,0]
*[3,5,7,4,2,0]          [1,5,7,4,2,0]           [3,5,7,4,2,0]
Process returned 13 (0xD)   execution time : 22.481 s
Press any key to continue.
```

Result:

Thus, a Program to convert NFA to Deterministic Finite Automata(D.F.A.) was implemented.

EX NO: 4 Elimination of Ambiguity, Left Recursion and Left Factoring
Date:

Aim:

A program for Elimination of Left Recursion and implementation Of Left Factoring

Algorithm:

Left Recursion Elimination:

1. Start the program.
2. Initialize the arrays for taking input from the user.
3. Prompt the user to input the no. of non-terminals having left recursion and no. of productions for these non-terminals.
4. Prompt the user to input the production for non-terminals.
5. Eliminate left recursion using the following rules:-

   A->Aα1| Aα2 | . . . . . |Aαm
   A->β1| β2| . . . . .| βn
   Then replace it by
   A-> βi A' i=1,2,3,…..m
   A'-> αj A' j=1,2,3,…..n
   A'-> Ɛ
6. After eliminating the left recursion by applying these rules, display the productions without left recursion.
7. Stop.

Left Factoring:

1. Start
2. Ask the user to enter the set of productions
3. Check for common symbols in the given set of productions by comparing with:
   1. A->aB1|aB2
4. If found, replace the particular productions with:
   a. A->aA'
   b. A'->B1 | B2|ε
5. Display the output
6. Exit

Source Code:

Left Recursion Elimination: Left Recursion Elimination.cpp

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;
int main()
{
 int n;
 cout<<"\nEnter number of non terminals: ";
 cin>>n;
 cout<<"\nEnter non terminals one by one: ";
 int i;
 vector<string> nonter(n);
 vector<int> leftrecr(n,0);
 for(i=0;i<n;++i) {
 cout<<"\nNon terminal "<<i+1<<" : ";
 cin>>nonter[i];
 }
 vector<vector<string> > prod;
 cout<<"\nEnter '^' for null";
 for(i=0;i<n;++i) {
 cout<<"\nNumber of "<<nonter[i]<<" productions: ";
 int k;
 cin>>k;
 int j;
 cout<<"\nOne by one enter all "<<nonter[i]<<" productions";
 vector<string> temp(k);
 for(j=0;j<k;++j) {
 cout<<"\nRHS of production "<<j+1<<": ";
 string abc;
 cin>>abc;
 temp[j]=abc;
if(nonter[i].length()<=abc.length()&&nonter[i].compare(abc.substr(0,nonter[i].length()))==0)
 leftrecr[i]=1;
 }
 prod.push_back(temp);
 }
 for(i=0;i<n;++i) {
```

```
 cout<<leftrecr[i];
 }
 for(i=0;i<n;++i) {
 if(leftrecr[i]==0)
 continue;
 int j;
 nonter.push_back(nonter[i]+"'");
 vector<string> temp;
 for(j=0;j<prod[i].size();++j) {
if(nonter[i].length()<=prod[i][j].length()&&nonter[i].compare(prod[i][j].substr(0,nonter[i].lengt
h
()))==0) {
 string
abc=prod[i][j].substr(nonter[i].length(),prod[i][j].length()-nonter[i].length())+nonter[i]+"'";
 temp.push_back(abc);
 prod[i].erase(prod[i].begin()+j);
 --j;
 }
 else {
 prod[i][j]+=nonter[i]+"'";
 }
 }
 temp.push_back("^");
 prod.push_back(temp);
 }
 cout<<"\n\n";
 cout<<"\nNew set of non-terminals: ";
 for(i=0;i<nonter.size();++i)
 cout<<nonter[i]<<" ";
 cout<<"\n\nNew set of productions: ";
 for(i=0;i<nonter.size();++i) {
 int j;
 for(j=0;j<prod[i].size();++j) {
 cout<<"\n"<<nonter[i]<<" -> "<<prod[i][j];
 }
 }
 return 0;
}
```

Left Factoring: Left Factoring.cpp

```
#include <iostream>
#include <math.h>
#include <vector>
#include <string>
#include <stdlib.h>
using namespace std;
int main()
{
 cout<<"\nEnter number of productions: ";
 int p;
 cin>>p;
 vector<string> prodleft(p),prodright(p);
 cout<<"\nEnter productions one by one: ";
 int i;
 for(i=0;i<p;++i) {
 cout<<"\nLeft of production "<<i+1<<": ";
 cin>>prodleft[i];
 cout<<"\nRight of production "<<i+1<<": ";
 cin>>prodright[i];
 }
 int j;
 int e=1;
 for(i=0;i<p;++i) {
 for(j=i+1;j<p;++j) {
 if(prodleft[j]==prodleft[i]) {
 int k=0;
 string com="";
while(k<prodright[i].length()&&k<prodright[j].length()&&prodright[i][k]==prodright[j][k]) {
 com+=prodright[i][k];
 ++k;
 }
 if(k==0)
 continue;
 char* buffer;
 string comleft=prodleft[i];
 if(k==prodright[i].length()) {
 prodleft[i]+=string(itoa(e,buffer,10));
 prodleft[j]+=string(itoa(e,buffer,10));
```

```
 prodright[i]="^";
 prodright[j]=prodright[j].substr(k,prodright[j].length()-k);
 }
 else if(k==prodright[j].length()) {
 prodleft[i]+=string(itoa(e,buffer,10));
 prodleft[j]+=string(itoa(e,buffer,10));
 prodright[j]="^";
 prodright[i]=prodright[i].substr(k,prodright[i].length()-k);
 }
 else {
 prodleft[i]+=string(itoa(e,buffer,10));
 prodleft[j]+=string(itoa(e,buffer,10));
 prodright[j]=prodright[j].substr(k,prodright[j].length()-k);
 prodright[i]=prodright[i].substr(k,prodright[i].length()-k);
 }
 int l;
 for(l=j+1;l<p;++l) {
if(comleft==prodleft[l]&&com==prodright[l].substr(0,fmin(k,prodright[l].length()))) {
 prodleft[l]+=string(itoa(e,buffer,10));
 prodright[l]=prodright[l].substr(k,prodright[l].length()-k);
 }
 }
 prodleft.push_back(comleft);
 prodright.push_back(com+prodleft[i]);
 ++p;
 ++e;
 }
 }
 }
 cout<<"\n\nNew productions";
 for(i=0;i<p;++i) {
 cout<<"\n"<<prodleft[i]<<"->"<<prodright[i];
 }
 return 0;
}
```

SAMPLE OUTPUT:

Left Recursion Elimination:

```
Enter number of non terminals: 3

Enter non terminals one by one:
Non terminal 1 : E

Non terminal 2 : T

Non terminal 3 : F

Enter '^' for null
Number of E productions: 2

One by one enter all E productions
RHS of production 1: E+T

RHS of production 2: T

Number of T productions: 2

One by one enter all T productions
RHS of production 1: T*F

RHS of production 2: F

Number of F productions: 2

One by one enter all F productions
RHS of production 1: (E)

RHS of production 2: I
110


New set of non-terminals: E T F E' T'

New set of productions:
E -> TE'
T -> FT'
F -> (E)
F -> I
E' -> +TE'
E' -> ^
T' -> *FT'
T' -> ^
Process returned 0 (0x0)   execution time : 40.299 s
Press any key to continue.
```

Left Factoring:

```
Enter number of productions: 3

Enter productions one by one:
Left of production 1: A

Right of production 1: abcd

Left of production 2: B

Right of production 2: abef

Left of production 3: C

Right of production 3: abgh


New productions
A->abcd
B->abef
C->abgh
Process returned 0 (0x0)   execution time : 197.964 s
Press any key to continue.
```

Result:

Thus, program for Elimination of Left Recursion and implementation Of Left Factoring was implemented.

EX NO: 5 **FIRST AND FOLLOW**
Date:

Aim:

To write a program to perform first and follow using any language.

Algorithm :

FIRST(X) for all grammar symbols X:

1. If X is terminal, FIRST(X) = {X}.
2. If X → ε is a production, then add ε to FIRST(X).
3. If X is a non-terminal, and X → Y1 Y2 … Yk is a production, and ε is in all of FIRST(Y1), …, FIRST(Yk), then add ε to FIRST(X).
4. If X is a non-terminal, and X → Y1 Y2 … Yk is a production, then add a to FIRST(X) if for some i, a is in FIRST(Yi), and ε is in all of FIRST(Y1), …, FIRST(Yi-1).

FOLLOW(A) for all non-terminals A:

1. If $ is the input end-marker, and S is the start symbol, $ ∈ FOLLOW(S).
2. If there is a production, A → αBβ, then (FIRST(β) – ε) ⊆ FOLLOW(B).
3. If there is a production, A → αB, or a production A → αBβ, where ε ∈ FIRST(β), then FOLLOW(A) ⊆ FOLLOW(B).

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<String.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 productions:\n");
 scanf("%d",&n);
 printf("Enter the productions:\n");
 for(i=0;i<n;i++)
 scanf("%s%c",a[i],&ch);
 do{
 m=0;
```

```
 printf("Enter the elements whose first & follow is to be found:");
 scanf("%c",&c);
 first(c);
 printf("First(%c)={",c);
 for(i=0;i<m;i++)
 printf("%c",f[i]);
 printf("}\n");
 strcpy(f," ");
 fflush(stdin);
 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]);
}
}
}
}
```

SAMPLE OUTPUT:

```
Enter the no of productions:
5
Enter the productions:
S=AbCd
A=Cf
A=a
C=gE
E=h
Enter the elements whose first & follow is to be found:S
First(S)={ga}
Follow(S)={$}
Continue(0/1)?1
Enter the elements whose first & follow is to be found:A
First(A)={ga}
Follow(A)={b}
Continue(0/1)?1
Enter the elements whose first & follow is to be found:C
First(C)={g}
Follow(C)={df}
Continue(0/1)?1
Enter the elements whose first & follow is to be found:E
First(E)={h}
Follow(E)={df}
Continue(0/1)?0
```

Result:

Thus, program to perform first and follow using any language was implemented

EX NO: 6 **PREDICTIVE PARSING**
Date:

Aim:

Program for the construction of predictive parsing table.

Algorithm:

1. Start the program.
2. Initialize the required variables.
3. Get the number of coordinates and productions from the user.
4. Perform the following
   for (each production A → α in G) {
   for (each terminal a in FIRST(α))
   add A → α to M[A, a];
   if (ε is in FIRST(α))
   for (each symbol b in FOLLOW(A))
   add A → α to M[A, b];
5. Print the resulting stack.
6. Print if the grammar is accepted or not.
7. Exit the program.

Code:

PREDICTIVE PARSING: predictive_pasing.c

```
#include<stdio.h>
#include<string.h>
#include<ctype.h> // Include ctype.h for isupper()

void main() {
   char st[10][20], ft[20][20], fol[20][20];
   int n, i, j, k, l;

   printf("Enter the number of non-terminals: ");
   scanf("%d", &n);
   printf("Enter the productions in the grammar:\n");
   for (i = 0; i < n; i++)
      scanf("%s", st[i]);

   for (i = 0; i < n; i++)
```

```
    fol[i][0] = '\0';

// Compute FIRST sets
for (i = 0; i < n; i++) {
    j = 3;
    l = 0;
    while (st[i][j] != '\0') {
        if (!isupper(st[i][j])) {
            ft[i][l++] = st[i][j];
            ft[i][l] = '\0'; // Terminate the string
            break; // Break after encountering a terminal symbol
        } else {
            k = 0;
            while (st[i][j] != st[k][0])
                k++;
            strcat(ft[i], ft[k]);
            if (strchr(ft[k], '@') == NULL) // If '@' (epsilon) is not in the FIRST set
                break; // Break if epsilon is not in the FIRST set of current non-terminal
        }
        j++;
    }
}

// Compute FOLLOW sets
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        l = strlen(st[j]);
        for (k = 3; k < l; k++) {
            if (st[j][k] == st[i][0]) {
                if (st[j][k + 1] != '\0') {
                    if (!isupper(st[j][k + 1])) {
                        fol[i][strlen(fol[i])] = st[j][k + 1];
                        fol[i][strlen(fol[i])] = '\0'; // Terminate the string
                    } else {
                        int x = 0;
                        while (st[j][k + 1] != st[x][0])
                            x++;
                        strcat(fol[i], ft[x]);
                    }
```

```
                } else {
                    int x = 0;
                    while (st[j][0] != st[x][0])
                        x++;
                    strcat(fol[i], fol[x]);
                }
            }
        }
    }
}

// Print FIRST and FOLLOW sets
printf("\nFIRST:\n");
for (i = 0; i < n; i++)
    printf("FIRST[%c] = %s\n", st[i][0], ft[i]);

printf("\nFOLLOW:\n");
for (i = 0; i < n; i++)
    printf("FOLLOW[%c] = %s\n", st[i][0], fol[i]);
}
```

Output :

```
Enter the number of non-terminals: 2
Enter the productions in the grammar:
S->aAb
A->b

FIRST:
FIRST[S] = a
FIRST[A] = b

FOLLOW:
FOLLOW[S] =
FOLLOW[A] = b

Process returned 2 (0x2)   execution time : 23.452 s
Press any key to continue.
```

Result:

Thus, the program was successfully compiled and run.

EX NO: 7 **Implementation of Shift Reduce Parsing**
Date:

Aim:

To write a program to Implementation of Shift Reduce Parsing

Algorithm:

Repeat: For each production A ∈ ∈of the grammar do
    For each terminal in FIRST(∈)
        add A ∈ ∈ to M[A, a]
    if FIRST(∈) contains ∈
        add A ∈ ∈ to M[A, b] for each bin FOLLOW(A)
    if ∈is in FIRST(∈) and $ is in FOLLOW(A)
        add A ∈ ∈ to M[A,$]
    make each undefined entry of M be error

Code:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct prodn
{
    char p1[10];
    char p2[10];
};
void main()
{
    char input[20],stack[50],temp[50],ch[2],*t1,*t2,*t;
    int i,j,s1,s2,s,count=0;
    struct prodn p[10];
    FILE *fp=fopen("sr_input.txt","r");
    stack[0]='\0';
    clrscr();
    printf("\n Enter the input string\n");
    scanf("%s",&input);
    while(!feof(fp))
    {
        fscanf(fp,"%s\n",temp);
        t1=strtok(temp,"->");
        t2=strtok(NULL,"->");
```

```
        strcpy(p[count].p1,t1);
        strcpy(p[count].p2,t2);
        count++;
}
i=0;
while(1)
{
        if(i<strlen(input))
        {
                ch[0]=input[i];
                ch[1]='\0';
                i++;
                strcat(stack,ch);
                printf("%s\n",stack);
        }
        for(j=0;j<count;j++)
        {
                t=strstr(stack,p[j].p2);
                if(t!=NULL)
                {
                        s1=strlen(stack);
                        s2=strlen(t);
                        s=s1-s2;
                        stack[s]='\0';
                        strcat(stack,p[j].p1);
                        printf("%s\n",stack);
                        j=-1;
                }
        }
        if(strcmp(stack,"E")==0&&i==strlen(input))
        {
                printf("\n Accepted");
                break;
        }
        if(i==strlen(input))
        {
                printf("\n Not Accepted");
                break;
        }
```

```
    }
    getch();
}
```

Input File: sr_input.txt

```
E->E+E
E->E*E
E->i
```

Output:

Result:

Thus, the program was successfully compiled and run.

EX NO: 8 **Computation of Leading and Trailing**
Date:

Aim:

To write a program to Computation of Leading and Trailing

Algorithm:

Leading sets:

Repeat:
For each production A → αβ in the grammar:
Compute leading set for non-terminal A
if α is a terminal or α is ε:
Add α to leading[A]
else if α is a non-terminal:
Add leading[α] to leading[A] // Add leading of non-terminal α to leading of A

If α derives ε, compute leading set for β and add it to A's leading set
if FIRST(α) contains ε:
Add FIRST(β) - {ε} to leading[A]

Trailing sets

Repeat:
For each production A → αβ in the grammar:
Compute trailing set for non-terminal A
if β is a terminal or β is ε:
Add β to trailing[A]
else if β is a non-terminal:
Add trailing[β] to trailing[A] // Add trailing of non-terminal β to trailing of A

If β derives ε, compute trailing set for α and add it to A's trailing set
if FIRST(β) contains ε:
Add trailing[α] to trailing[A]

Code:

Leading sets:

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

#define MAX_NONTERMINALS 10
#define MAX_TERMINALS 10

typedef struct {
    char symbol;
    int numProductions;
    char productions[10][10];
} NonTerminal;

void computeLeadingSet(NonTerminal nonTerminals[], char
leadingSet[][MAX_TERMINALS], int numNonTerminals, int numTerminals);
bool isTerminal(char symbol);
bool isNonTerminal(char symbol);
bool addToLeadingSet(char leadingSet[][MAX_TERMINALS], char symbol, char terminal, int
nonTerminalIndex);

int main() {
    int numNonTerminals, numTerminals;

    printf("Enter the number of non-terminals: ");
    scanf("%d", &numNonTerminals);

    printf("Enter the number of terminals: ");
    scanf("%d", &numTerminals);

    NonTerminal nonTerminals[MAX_NONTERMINALS];
    for (int i = 0; i < numNonTerminals; i++) {
        printf("Enter non-terminal %d and its productions separated by space: ", i + 1);
        scanf(" %c", &nonTerminals[i].symbol);
        scanf("%d", &nonTerminals[i].numProductions);
        for (int j = 0; j < nonTerminals[i].numProductions; j++) {
            scanf("%s", nonTerminals[i].productions[j]);
```

```
        }
    }

    char leadingSet[MAX_NONTERMINALS][MAX_TERMINALS];
    computeLeadingSet(nonTerminals, leadingSet, numNonTerminals, numTerminals);
    printf("\nLeading Sets:\n");
    for (int i = 0; i < numNonTerminals; i++) {
        printf("Leading(%c): { ", nonTerminals[i].symbol);
        for (int j = 0; j < numTerminals; j++) {
            if (leadingSet[i][j] != '\0') {
                printf("%c ", leadingSet[i][j]);
            }
        }
        printf("}\n");
    }

    return 0;
}

void computeLeadingSet(NonTerminal nonTerminals[], char
leadingSet[][MAX_TERMINALS], int numNonTerminals, int numTerminals) {
    for (int i = 0; i < numNonTerminals; i++) {
        memset(leadingSet[i], '\0', sizeof(leadingSet[i]));
    }

    bool changed = true;
    while (changed) {
        changed = false;

        for (int i = 0; i < numNonTerminals; i++) {
            for (int j = 0; j < nonTerminals[i].numProductions; j++) {
                char firstSymbol = nonTerminals[i].productions[j][0];
                if (isTerminal(firstSymbol)) {
                    changed |= addToLeadingSet(leadingSet, nonTerminals[i].symbol, firstSymbol, i);
                } else if (isNonTerminal(firstSymbol)) {
                    for (int k = 0; k < numNonTerminals; k++) {
                        if (nonTerminals[k].symbol == firstSymbol) {
                            for (int l = 0; l < numTerminals; l++) {
                                if (leadingSet[k][l] != '\0') {
```

```
                        changed |= addToLeadingSet(leadingSet, nonTerminals[i].symbol,
leadingSet[k][l], i);
                    }
                }
                if (!nonTerminals[k].productions[0][0]) {
                    changed |= addToLeadingSet(leadingSet, nonTerminals[i].symbol, 'e', i);
                }
            }
        }
    }
  }
}

bool isTerminal(char symbol) {
  return symbol >= 'a' && symbol <= 'z';
}

bool isNonTerminal(char symbol) {
  return symbol >= 'A' && symbol <= 'Z';
}

bool addToLeadingSet(char leadingSet[][MAX_TERMINALS], char symbol, char terminal, int
nonTerminalIndex) {
  for (int i = 0; i < MAX_TERMINALS; i++) {
    if (leadingSet[nonTerminalIndex][i] == terminal) {
      return false;
    }
    if (leadingSet[nonTerminalIndex][i] == '\0') {
      leadingSet[nonTerminalIndex][i] = terminal;
      return true;
    }
  }
  return false;
}
```

Trailing sets:

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

#define MAX_NONTERMINALS 10
#define MAX_TERMINALS 10

typedef struct {
    char symbol;
    int numProductions;
    char productions[10][10];
} NonTerminal;

void computeTrailingSet(NonTerminal nonTerminals[], char
trailingSet[][MAX_TERMINALS], int numNonTerminals, int numTerminals);
bool isTerminal(char symbol);
bool isNonTerminal(char symbol);
bool addToTrailingSet(char trailingSet[][MAX_TERMINALS], char symbol, char
terminal, int nonTerminalIndex);

int main() {
    int numNonTerminals, numTerminals;

    printf("Enter the number of non-terminals: ");
    scanf("%d", &numNonTerminals);

    printf("Enter the number of terminals: ");
    scanf("%d", &numTerminals);

    NonTerminal nonTerminals[MAX_NONTERMINALS];
    for (int i = 0; i < numNonTerminals; i++) {
        printf("Enter non-terminal %d and its productions separated by space: ", i + 1);
```

```
        scanf(" %c", &nonTerminals[i].symbol);
        scanf("%d", &nonTerminals[i].numProductions);
        for (int j = 0; j < nonTerminals[i].numProductions; j++) {
            scanf("%s", nonTerminals[i].productions[j]);
        }
    }

    char trailingSet[MAX_NONTERMINALS][MAX_TERMINALS];
    computeTrailingSet(nonTerminals, trailingSet, numNonTerminals,
numTerminals);
    printf("\nTrailing Sets:\n");
    for (int i = 0; i < numNonTerminals; i++) {
        printf("Trailing(%c): { ", nonTerminals[i].symbol);
        for (int j = 0; j < numTerminals; j++) {
            if (trailingSet[i][j] != '\0') {
                printf("%c ", trailingSet[i][j]);
            }
        }
        printf("}\n");
    }

    return 0;
}

void computeTrailingSet(NonTerminal nonTerminals[], char
trailingSet[][MAX_TERMINALS], int numNonTerminals, int numTerminals) {
    for (int i = 0; i < numNonTerminals; i++) {
        memset(trailingSet[i], '\0', sizeof(trailingSet[i]));
    }

    bool changed = true;
    while (changed) {
        changed = false;
```

```
        for (int i = 0; i < numNonTerminals; i++) {
            for (int j = 0; j < nonTerminals[i].numProductions; j++) {
                int len = strlen(nonTerminals[i].productions[j]);
                char lastSymbol = nonTerminals[i].productions[j][len - 1];

                if (isTerminal(lastSymbol)) {
                    changed |= addToTrailingSet(trailingSet, nonTerminals[i].symbol,
lastSymbol, i);
                } else if (isNonTerminal(lastSymbol)) {
                    for (int k = 0; k < numNonTerminals; k++) {
                        if (nonTerminals[k].symbol == lastSymbol) {
                            for (int l = 0; l < numTerminals; l++) {
                                if (trailingSet[k][l] != '\0') {
                                    changed |= addToTrailingSet(trailingSet,
nonTerminals[i].symbol, trailingSet[k][l], i);
                                }
                            }
                            if (!nonTerminals[k].productions[0][0]) {
                                changed |= addToTrailingSet(trailingSet,
nonTerminals[i].symbol, 'e', i);
                            }
                        }
                    }
                }
            }
        }
    }
}

bool isTerminal(char symbol) {
    return symbol >= 'a' && symbol <= 'z';
}

bool isNonTerminal(char symbol) {
```

```
    return symbol >= 'A' && symbol <= 'Z';
}

bool addToTrailingSet(char trailingSet[][MAX_TERMINALS], char symbol, char
terminal, int nonTerminalIndex) {
    for (int i = 0; i < MAX_TERMINALS; i++) {
        if (trailingSet[nonTerminalIndex][i] == terminal) {
            return false;
        }
        if (trailingSet[nonTerminalIndex][i] == '\0') {
            trailingSet[nonTerminalIndex][i] = terminal;
            return true;
        }
    }
    return false;
}
```

Output:

Leading set:

```
Enter the number of non-terminals: 2
Enter the number of terminals: 3
Enter non-terminal 1 and its productions separated by space: S 2 Ab c
Enter non-terminal 2 and its productions separated by space: A 2 d ε

Leading Sets:
Leading(S): { c d }
Leading(A): { d }
```

Trailing Set:

```
Enter the number of non-terminals: 2
Enter the number of terminals: 3
Enter non-terminal 1 and its productions separated by space: S 1 Ab
Enter non-terminal 2 and its productions separated by space: A 3 aA bA c

Trailing Sets:
Trailing(S): { b }
Trailing(A): { c }
```

Result:

Thus, the program was successfully compiled and run.

EX NO: 9 **Computation of LR (0) items**
Date:

Aim:

To write a program to Computation of LR (0) items

Algorithm:

```
ComputeLR0ItemsAndPrint():
   Input:
      numRules <- Prompt user to enter the number of rules
      for i from 0 to numRules - 1:
         rules[i] <- Prompt user to enter the rule in the format 'S -> AB'

   LR0ItemComputationPhase(rules, numRules)
   PrintLR0Items()

Algorithm LR0ItemComputationPhase(rules[], numRules):
   numLR0Items <- 0
   for i from 0 to numRules - 1:
      lr0Items[numLR0Items].ruleIndex <- i
      lr0Items[numLR0Items].dotPosition <- 0
      numLR0Items++

Algorithm PrintLR0Items():
   for each item in lr0Items:
      Print item with dot before the first symbol of the right-hand side
```

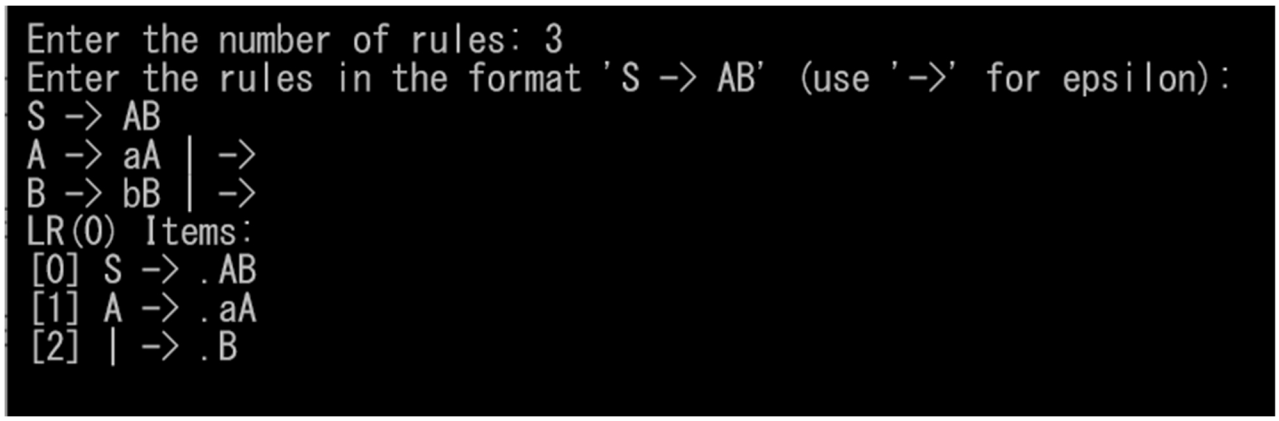
Code:

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

#define MAX_RULES 10
#define MAX_SYMBOLS 10

typedef struct {
   char left;
   char right[MAX_SYMBOLS];
} ProductionRule;
```

```
typedef struct {
    int ruleIndex;
    int dotPosition;
} LR0Item;

void printLR0Items(ProductionRule rules[], int numRules) {
    int i, j;
    printf("LR(0) Items:\n");
    for (i = 0; i < numRules; i++) {
        printf("[%d] %c -> ", i, rules[i].left);
        for (j = 0; rules[i].right[j] != '\0'; j++) {
            if (j == 0) printf(".");
            printf("%c", rules[i].right[j]);
        }
        if (j == 0) printf(".");
        printf("\n");
    }
}

void computeLR0Items(ProductionRule rules[], int numRules, LR0Item lr0Items[], int
*numLR0Items) {
    int i;
    *numLR0Items = 0;
    for (i = 0; i < numRules; i++) {
        lr0Items[*numLR0Items].ruleIndex = i;
        lr0Items[*numLR0Items].dotPosition = 0;
        (*numLR0Items)++;
    }
}

int main() {
    ProductionRule rules[MAX_RULES];
    LR0Item lr0Items[MAX_RULES];
    int numRules, numLR0Items;

    printf("Enter the number of rules: ");
    scanf("%d", &numRules);
    printf("Enter the rules in the format 'S -> AB' (use '->' for epsilon):\n");
```

```
    for (int i = 0; i < numRules; i++) {
        scanf(" %c -> %s", &rules[i].left, rules[i].right);
    }

    computeLR0Items(rules, numRules, lr0Items, &numLR0Items);
    printLR0Items(rules, numRules);

    return 0;
}
```

Output:



Result:

Thus, the program was successfully compiled and run.

EX NO: 10a Intermediate core generation - Postfix
Date:

Aim:

To write a program to generate Intermediate core generation – Postfix

Algorithm:

infixToPostfix(infix, postfix):
Initialize an empty stack
Initialize an empty string for the postfix expression

For each character 'ch' in the infix expression:
If 'ch' is an operand:
Append 'ch' to the postfix expression
Else If 'ch' is '(':
Push 'ch' onto the stack
Else If 'ch' is ')':
While the stack is not empty and the top of the stack is not '(':
Pop an operator from the stack and append it to the postfix expression
Pop '(' from the stack
Else If 'ch' is an operator:
While the stack is not empty and precedence of 'ch' is less than or equal to precedence of the top of the stack:
Pop an operator from the stack and append it to the postfix expression
Push 'ch' onto the stack

While the stack is not empty:
Pop an operator from the stack and append it to the postfix expression

Return the postfix expression

Main Function:
Input infix expression from the user
Convert infix expression to postfix using infixToPostfix function
Output the resulting postfix expression

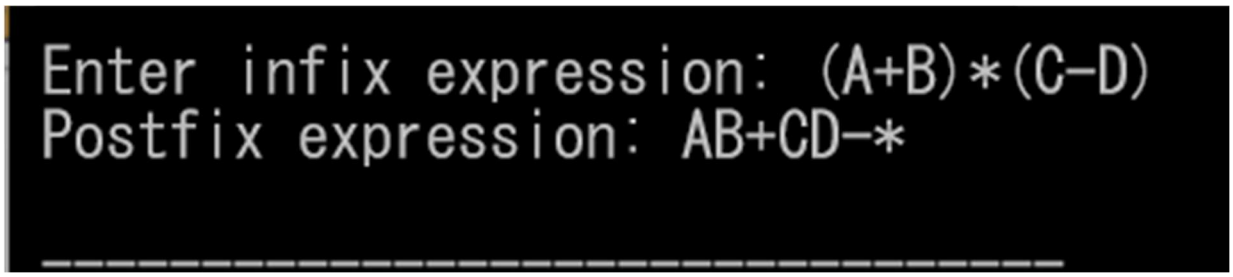
Code:

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

#define MAX_EXPR_SIZE 100

typedef struct {
    int top;
    char items[MAX_EXPR_SIZE];
} Stack;

void initialize(Stack *s) {
    s->top = -1;
}

int isEmpty(Stack *s) {
    return s->top == -1;
}

void push(Stack *s, char c) {
    s->items[++(s->top)] = c;
}

char pop(Stack *s) {
    return s->items[(s->top)--];
}

int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

int precedence(char op) {
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return 0;
```

```
}

void infixToPostfix(char *infix, char *postfix) {
    Stack stack;
    initialize(&stack);

    int i, j = 0;
    for (i = 0; infix[i]; i++) {
        char ch = infix[i];
        if (isalnum(ch)) {
            postfix[j++] = ch;
        } else if (ch == '(') {
            push(&stack, ch);
        } else if (ch == ')') {
            while (!isEmpty(&stack) && stack.items[stack.top] != '(') {
                postfix[j++] = pop(&stack);
            }
            if (!isEmpty(&stack) && stack.items[stack.top] != '(') {
                printf("Invalid expression\n");
                exit(1);
            } else {
                pop(&stack);
            }
        } else {
            while (!isEmpty(&stack) && precedence(ch) <= precedence(stack.items[stack.top])) {
                postfix[j++] = pop(&stack);
            }
            push(&stack, ch);
        }
    }

    while (!isEmpty(&stack)) {
        postfix[j++] = pop(&stack);
    }
    postfix[j] = '\0';
}

int main() {
    char infix[MAX_EXPR_SIZE], postfix[MAX_EXPR_SIZE];
```

```
    printf("Enter infix expression: ");
    fgets(infix, sizeof(infix), stdin);
    infix[strcspn(infix, "\n")] = '\0';

    infixToPostfix(infix, postfix);

    printf("Postfix expression: %s\n", postfix);

    return 0;
}
```

Output:



Result:

Thus, the program was successfully compiled and run.

EX NO:10b Intermediate Code Generation - Prefix
Date:

Aim:

To write a program to generate Intermediate Code Generation - Prefix

Algorithm:

infixToPrefix(infix, prefix):
Initialize an empty stack
Initialize an empty string for the postfix expression
For each character 'ch' in the infix expression:
If 'ch' is an operand:
Append 'ch' to the postfix expression
Else If 'ch' is '(':
Push 'ch' onto the stack
Else If 'ch' is ')':
While the stack is not empty and the top of the stack is not '(':
Pop an operator from the stack and append it to the postfix expression
Pop '(' from the stack
Else If 'ch' is an operator:
While the stack is not empty and precedence of 'ch' is less than or equal to precedence of the top of the stack:
Pop an operator from the stack and append it to the postfix expression
Push 'ch' onto the stack

While the stack is not empty:
Pop an operator from the stack and append it to the postfix expression

Reverse the postfix expression to obtain the prefix expression

Main Function:
Input infix expression from the user
Convert infix expression to prefix using infixToPrefix function
Output the resulting prefix expression

Code:

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

#define MAX_EXPR_SIZE 100

typedef struct {
    int top;
    char items[MAX_EXPR_SIZE];
} Stack;

void initialize(Stack *s) {
    s->top = -1;
}

int isEmpty(Stack *s) {
    return s->top == -1;
}

void push(Stack *s, char c) {
    s->items[++(s->top)] = c;
}

char pop(Stack *s) {
    return s->items[(s->top)--];
}

int isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

int precedence(char op) {
    if (op == '+' || op == '-')
        return 1;
    if (op == '*' || op == '/')
        return 2;
    return 0;
}
```

```
void infixToPostfix(char *infix, char *postfix) {
   Stack stack;
   initialize(&stack);

   int i, j = 0;
   for (i = 0; infix[i]; i++) {
      char ch = infix[i];
      if (isalnum(ch)) {
         postfix[j++] = ch;
      } else if (ch == '(') {
         push(&stack, ch);
      } else if (ch == ')') {

         while (!isEmpty(&stack) && stack.items[stack.top] != '(') {
            postfix[j++] = pop(&stack);
         }
         if (!isEmpty(&stack) && stack.items[stack.top] != '(') {
            printf("Invalid expression\n");
            exit(1);
         } else {
            pop(&stack);
         }
      } else {

         while (!isEmpty(&stack) && precedence(ch) <= precedence(stack.items[stack.top])) {
            postfix[j++] = pop(&stack);
         }
         push(&stack, ch);
      }
   }


   while (!isEmpty(&stack)) {
      postfix[j++] = pop(&stack);
   }
   postfix[j] = '\0';
}
```

```
void reverse(char *str) {
    int length = strlen(str);
    for (int i = 0; i < length / 2; i++) {
        char temp = str[i];
        str[i] = str[length - i - 1];
        str[length - i - 1] = temp;
    }
}

int main() {
    char infix[MAX_EXPR_SIZE], postfix[MAX_EXPR_SIZE], prefix[MAX_EXPR_SIZE];

    printf("Enter infix expression: ");
    fgets(infix, sizeof(infix), stdin);
    infix[strcspn(infix, "\n")] = '\0';

    infixToPostfix(infix, postfix);

    reverse(postfix);

    printf("Prefix expression: %s\n", postfix);

    return 0;
}
```

Output:

```
Enter infix expression: (A+B) * (C-D)
Prefix expression:  -DC *+BA
```

Result:

Thus, the program was successfully compiled and run**.**

EX NO:11 **Intermediate Code Generation - Quadruple, Triple,**
Date: **Indirect triple**

Aim:

To write a program to generate Intermediate Code Generation - Quadruple

Algorithm:

The algorithm takes a sequence of three-address statements as input. For each three address statements of the form a: = b or c performs the various actions. These are as follows:

1. Invoke a function getreg to find out the location L where the result of computation b op c should be stored.
2. Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y'. If the value of y is not already in L then generate the instruction MOV y' , L to place a copy of y in L.
3. Generate the instruction OP z' , L where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptors.
4. If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of x : = y op z those register will no longer contain y or z.

Code:

```
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<string.h>
void small();
void dove(int i);
int p[5]={0,1,2,3,4},c=1,i,k,l,m,pi;
char sw[5]={'=','-','+','/','*'},j[20],a[5],b[5],ch[2];
void main()
{
printf("Enter the expression:");
scanf("%s",j);
printf("\tThe Intermediate code is:\n");
small();
}
void dove(int i)
```

```
{
a[0]=b[0]='\0';
if(!isdigit(j[i+2])&&!isdigit(j[i-2]))
{
a[0]=j[i-1];
b[0]=j[i+1];
}
if(isdigit(j[i+2])){
a[0]=j[i-1];
b[0]='t';
b[1]=j[i+2];
}
if(isdigit(j[i-2]))
{
b[0]=j[i+1];
a[0]='t';
a[1]=j[i-2];
b[1]='\0';
}
if(isdigit(j[i+2]) &&isdigit(j[i-2]))
{
a[0]='t';
b[0]='t';
a[1]=j[i-2];
b[1]=j[i+2];
sprintf(ch,"%d",c);
j[i+2]=j[i-2]=ch[0];
}
if(j[i]=='*')
printf("\tt%d=%s*%s\n",c,a,b);
if(j[i]=='/')
printf("\tt%d=%s/%s\n",c,a,b);
if(j[i]=='+')
printf("\tt%d=%s+%s\n",c,a,b);if(j[i]=='-')
printf("\tt%d=%s-%s\n",c,a,b);
if(j[i]=='=')
printf("\t%c=t%d",j[i-1],--c);
sprintf(ch,"%d",c);
j[i]=ch[0];
```

```
c++;
small();
}
void small()
{
pi=0;l=0;
for(i=0;i<strlen(j);i++)
{
for(m=0;m<5;m++)
if(j[i]==sw[m])
if(pi<=p[m])
{
pi=p[m];
l=1;
k=i;
}
}
if(l==1)
dove(k);
else
exit(0);}
```

Output:

```
Enter the expression:a=b+c-d
        The Intermediate code is:
        t1=b+c
        t2=t1-d
        a=t2
Process returned 0 (0x0)   execution time : 17.410 s
Press any key to continue.
```

Result:

Thus, the program was successfully compiled and run.

EX NO:12 **A Simple Code Generator**
Date:

Aim:

To write a program to create a Simple Code Generator

Algorithm:

1. Start
2. Define a function generate_code(expr) that takes an arithmetic expression as input.
3. Within generate_code(expr):
    a. Tokenize the expression using strtok() function.
    b. Initialize a register counter reg to 1.
    c. Loop through the tokens:
    i. Print "MOV R%d, token" where token is the current token and %d represents the current register.
    ii. Get the next token.
    iii. If there's a next token:
        i. Determine the operation (+, -, *, /) based on the operator preceding the token.
        ii. Print the corresponding assembly-like operation (ADD, SUB, MUL, DIV) using the current register and the token.
    iv. Increment the register counter.
4. End generate_code function.
5. In the main function:
    a. Declare a character array expr to store the input expression.
    b. Prompt the user to input an arithmetic expression and store it in expr using scanf().
    c. Call generate_code(expr) to generate the assembly-like code for the input expression.
6. End

Code: (Python)

```
class CodeGenerator:
    def __init__(self):
        self.code = []

    def emit(self, instruction):
        self.code.append(instruction)

    def generate_code(self, ast):
```

```
        if isinstance(ast, int):
            self.emit(f"MOV R1, {ast}")
        elif isinstance(ast, str):
            self.emit(f"MOV R1, {ast}")
        elif isinstance(ast, tuple):
            op, left, right = ast
            self.generate_code(left)
            self.emit(f"MOV R2, R1")
            self.generate_code(right)
            if op == '+':
                self.emit("ADD R1, R2, R1")
            elif op == '-':
                self.emit("SUB R1, R2, R1")
            elif op == '*':
                self.emit("MUL R1, R2, R1")
            elif op == '/':
                self.emit("DIV R1, R2, R1")

# Function to parse the arithmetic expression
def parse_expression(expression):
    # Implementation of your parsing logic goes here
    # For simplicity, let's assume the expression is already parsed into an AST
    return ('+', 3, ('*', 4, 5))  # Example AST for "3 + 4 * 5"

expression = input("Enter an arithmetic expression: ")
ast = parse_expression(expression)

generator = CodeGenerator()
generator.generate_code(ast)

print("Generated Code:")
for instruction in generator.code:
    print(instruction)
```

Output:

```
Enter an arithmetic expression: 3 + 4 * 5
Generated Code:
MOV R1, 3
MOV R2, R1
MOV R1, 4
MOV R2, R1
MOV R1, 5
MUL R1, R2, R1
ADD R1, R2, R1
```

Result:

Thus, the program was successfully compiled and run.

EX NO:13 **Implementation of DAG**
Date:

Aim:

To write a program to create a Simple Code Generator

Algorithm:

1. The leaves of a graph are labeled by a unique identifier and that identifier can be variable names or constants.
2. Interior nodes of the graph are labeled by an operator symbol.
3. Nodes are also given a sequence of identifiers for labels to store the computed value.
4. If y operand is undefined then create node(y).
5. If z operand is undefined then for case(i) create node(z).
6. For case(i), create node(OP) whose right child is node(z) and left child is node(y).
7. For case(ii), check whether there is node(OP) with one child node(y).
8. For case(iii), node n will be node(y).
9. For node(x) delete x from the list of identifiers. Append x to attached identifiers list for the node n found in step 2. Finally set node(x) to n.

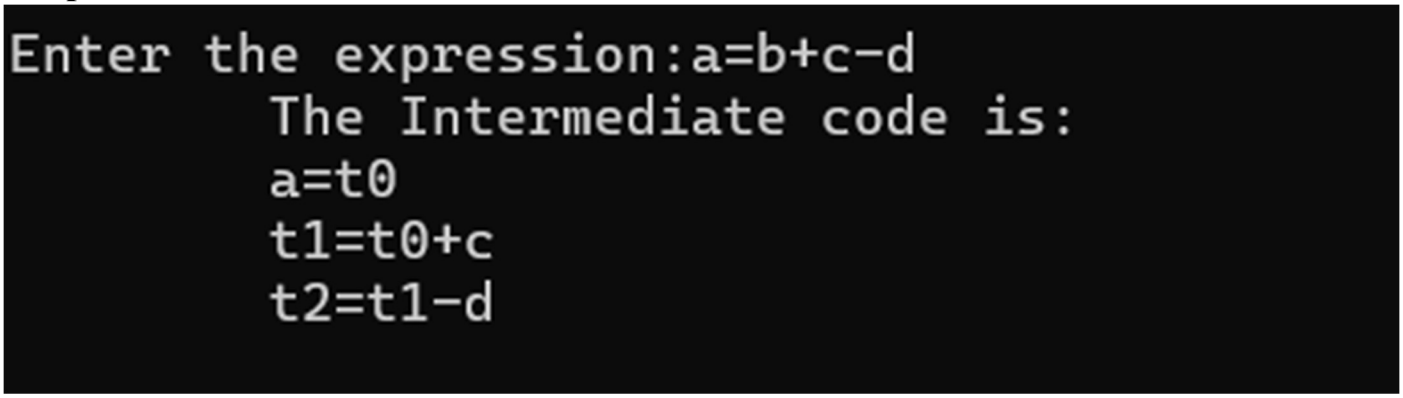
Code:

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

void small();
void dove(int i);

int p[5] = {0, 1, 2, 3, 4}, c = 1, i, k, l, m, pi;
char sw[5] = {'=', '-', '+', '/', '*'}, j[20], a[5], b[5], ch[2];

int main() {
   printf("Enter the expression:");
   scanf("%s", j);
   printf("\tThe Intermediate code is:\n");
   small();
   return 0;
}

void dove(int i) {
```

```
    a[0] = b[0] = '\0';
    if (!isdigit(j[i + 2]) && !isdigit(j[i - 2])) {
        a[0] = j[i - 1];
        b[0] = j[i + 1];
    }
    if (isdigit(j[i + 2])) {
        a[0] = j[i - 1];
        b[0] = 't';
        b[1] = j[i + 2];
    }
    if (isdigit(j[i - 2])) {
        b[0] = j[i + 1];
        a[0] = 't';
        a[1] = j[i - 2];
        b[1] = '\0';
    }
    if (isdigit(j[i + 2]) && isdigit(j[i - 2])) {
        a[0] = 't';
        b[0] = 't';
        a[1] = j[i - 2];
        b[1] = j[i + 2];
        sprintf(ch, "%d", c);
        j[i + 2] = j[i - 2] = ch[0];
    }
    if (j[i] == '*')
        printf("\tt%d=%s*%s\n", c, a, b);
    if (j[i] == '/')
        printf("\tt%d=%s/%s\n", c, a, b);
    if (j[i] == '+')
        printf("\tt%d=%s+%s\n", c, a, b);
    if (j[i] == '-')
        printf("\tt%d=%s-%s\n", c, a, b);
    if (j[i] == '=')
        printf("\t%c=t%d\n", j[i - 1], --c);
    sprintf(ch, "%d", c);
    j[i] = ch[0];
    c++;
    small();
}
```

```
void small() {
    pi = 0;
    l = 0;
    for (i = 0; i < strlen(j); i++) {
        for (m = 0; m < 5; m++) {
            if (j[i] == sw[m]) {
                if (pi <= p[m]) {
                    pi = p[m];
                    l = 1;
                    k = i;
                }
                if (l == 1)
                    dove(k);
                else
                    exit(0);
            }
        }
    }
}
```

Output:



Result:

Thus, the program was successfully compiled and run.

EX NO:14 Implementation of global data flow analysis
Date:

Aim:

To write a C program to implement Data Flow Analysis

Algorithm:

1. Start the Program Execution.
2. Read the total Numbers of Expression
3. Read the Left and Right side of Each Expressions
4. Display the Expressions with Line No
5. Display the Data flow movement with Particular Expressions
6. Stop the Program Execution.

Code: (c programming)

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

struct op {
   char l[20];
   char r[20];
} op[10];

int main() {
   int i, j, n, lineno = 1;
   char *match;

   printf("Enter the number of statements: ");
   scanf("%d", &n);

   for (i = 0; i < n; i++) {
      printf("Statement %d (left): ", i + 1);
      scanf("%s", op[i].l);
      printf("Statement %d (right): ", i + 1);
      scanf("%s", op[i].r);
   }

   printf("\nIntermediate Code:\n");
   for (i = 0; i < n; i++) {
      printf("Line No=%d\n", lineno++);
```

```
        printf("\t\t\t%s = %s\n", op[i].l, op[i].r);
    }

    printf("\n*** Data Flow Analysis for the Above Code ***\n");
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            match = strstr(op[j].r, op[i].l);
            if (match != NULL) {
                printf("\n%s is live at %s\n", op[i].l, op[j].r);
            }
        }
    }

    return 0;
}
```

Output:

```
Enter the number of statements: 3
Statement 1 (left): x
Statement 1 (right): y+z
Statement 2 (left): y
Statement 2 (right): a*b
Statement 3 (left): z
Statement 3 (right): p-q

Intermediate Code:
Line No=1
                        x = y+z
Line No=2
                        y = a*b
Line No=3
                        z = p-q

*** Data Flow Analysis for the Above Code ***

y is live at y+z

z is live at y+z
```

Result:

Thus, the program was successfully compiled and run.

EX NO:15 **Implement any one storage allocation strategies**
Date: **(heap, stack, static)**

Aim:

To implement Stack storage allocation strategies using C program.

Algorithm:

1. Initially check whether the stack is empty
2. Insert an element into the stack using push operation
3. Insert more elements onto the stack until stack becomes full
4. Delete an element from the stack using pop operation
5. Display the elements in the stack
6. Top the stack element will be displayed

Code: (C programming)

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

#define TRUE 1
#define FALSE 0

typedef struct Heap
{
   int data;
   struct Heap *next;
} node;

node *create();
void display(node *);
node *search(node *,int);
node *insert(node *);
void dele(node **);

int main()
{
   int choice,val;
   char ans;
   node *head = NULL;

   do
```

```
    {
        printf("\nprogram to perform various operations on heap using dynamic memory
management");
        printf("\n1.create");
        printf("\n2.display");
        printf("\n3.insert an element in a list");
        printf("\n4.delete an element from list");
        printf("\n5.quit");
        printf("\nenter your choice(1-5): ");
        scanf("%d", &choice);

        switch(choice)
        {
            case 1:
                head = create();
                break;
            case 2:
                display(head);
                break;
            case 3:
                head = insert(head);
                break;
            case 4:
                dele(&head);
                break;
            case 5:
                exit(0);
                break;
            default:
                printf("invalid choice, try again\n");
                break;
        }
    }
    while(choice != 5);

    return 0;
}

node* create()
```

```
{
   node *temp, *New, *head;
   int val;
   char ans='y';

   temp = NULL;
   head = NULL;
   do
   {
      printf("\n enter the element: ");
      scanf("%d", &val);
      New = (node*)malloc(sizeof(node));
      if(New == NULL)
      {
         printf("\nMemory is not allocated\n");
         return NULL;
      }
      New->data = val;
      New->next = NULL;

      if(head == NULL)
      {
         head = New;
         temp = head;
      }
      else
      {
         temp->next = New;
         temp = New;
      }

      printf("\nDo you want to enter more elements? (y/n): ");
      scanf(" %c", &ans);
   }
   while(ans == 'y' || ans == 'Y');

   printf("\nThe list is created\n");
   return head;
}
```

```
void display(node *head)
{
   node *temp;
   temp = head;

   if(temp == NULL)
   {
      printf("\nThe list is empty\n");
      return;
   }

   while(temp != NULL)
   {
      printf("%d -> ", temp->data);
      temp = temp->next;
   }
   printf("NULL\n");
}

node *search(node *head, int key)
{
   node *temp;
   temp = head;

   if(temp == NULL)
   {
      printf("The linked list is empty\n");
      return NULL;
   }

   while(temp != NULL)
   {
      if(temp->data == key)
      {
         printf("\nThe element is present in the list\n");
         return temp;
      }
      temp = temp->next;
```

```
    }

    printf("The element is not present in the list\n");
    return NULL;
}

node *insert(node *head)
{
    int choice;
    node *insert_head(node *);
    void insert_last(node *);
    void insert_after(node *);

    printf("\n1. Insert a node as a head node");
    printf("\n2. Insert a node as the last node");
    printf("\n3. Insert a node at an intermediate position in the list");
    printf("\nEnter your choice for insertion of node: ");
    scanf("%d", &choice);

    switch(choice)
    {
        case 1:
            head = insert_head(head);
            break;
        case 2:
            insert_last(head);
            break;
        case 3:
            insert_after(head);
            break;
        default:
            printf("Invalid choice\n");
            break;
    }
    return head;
}

node *insert_head(node *head)
{
```

```
    node *New, *temp;
    int val;

    New = (node*)malloc(sizeof(node));
    if(New == NULL)
    {
        printf("Memory allocation failed\n");
        return head;
    }

    printf("\nEnter the element which you want to insert: ");
    scanf("%d", &val);

    New->data = val;
    New->next = head;
    head = New;

    return head;
}

void insert_last(node *head)
{
    node *New, *temp;
    int val;

    New = (node*)malloc(sizeof(node));
    if(New == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }

    printf("\nEnter the element which you want to insert: ");
    scanf("%d", &val);

    New->data = val;
    New->next = NULL;

    temp = head;
```

```
    while(temp->next != NULL)
    {
        temp = temp->next;
    }

    temp->next = New;
}

void insert_after(node *head)
{
    int key;
    node *New, *temp;
    New = (node*)malloc(sizeof(node));

    if(New == NULL)
    {
        printf("Memory allocation failed\n");
        return;
    }

    printf("\nEnter the element which you want to insert: ");
    scanf("%d", &New->data);

    printf("\nEnter the element after which you want to insert the node: ");
    scanf("%d", &key);

    temp = head;
    while(temp != NULL)
    {
        if(temp->data == key)
        {
            New->next = temp->next;
            temp->next = New;
            return;
        }
        temp = temp->next;
    }

    printf("Element %d not found in the list\n", key);
```

```
}

void dele(node **head)
{
   node *temp, *prev;
   int key;

   temp = *head;

   if(temp == NULL)
   {
      printf("\nThe list is empty\n");
      return;
   }

   printf("\nEnter the element you want to delete: ");
   scanf("%d", &key);

   temp = search(*head, key);

   if(temp != NULL)
   {
      prev = *head;
      if(prev == temp)
      {
         *head = temp->next;
         free(temp);
         printf("\nThe element is deleted\n");
         return;
      }

      while(prev->next != temp)
      {
         prev = prev->next;
      }

      prev->next = temp->next;
      free(temp);
      printf("\nThe element is deleted\n");
```

```
    }
}
```

Output:

```
program to perform various operations on heap using dynamic memory management
1.create
2.display
3.insert an element in a list
4.delete an element from list
5.quit
enter your choice(1-5): 1

 enter the element: 10

Do you want to enter more elements? (y/n): y

 enter the element: 20

Do you want to enter more elements? (y/n): y

 enter the element: 30

Do you want to enter more elements? (y/n): n

The list is created

program to perform various operations on heap using dynamic memory management
1.create
2.display
3.insert an element in a list
4.delete an element from list
5.quit
enter your choice(1-5): 2
10 -> 20 -> 30 -> NULL

program to perform various operations on heap using dynamic memory management
1.create
2.display
3.insert an element in a list
4.delete an element from list
5.quit
enter your choice(1-5): 3

1. Insert a node as a head node
2. Insert a node as the last node
3. Insert a node at an intermediate position in the list
Enter your choice for insertion of node: 2

Enter the element which you want to insert: 40
```

```
program to perform various operations on heap using dynamic memory management
1.create
2.display
3.insert an element in a list
4.delete an element from list
5.quit
enter your choice(1-5): 2
10 -> 20 -> 30 -> 40 -> NULL

program to perform various operations on heap using dynamic memory management
1.create
2.display
3.insert an element in a list
4.delete an element from list
5.quit
enter your choice(1-5): 5

Process returned 0 (0x0)   execution time : 113.236 s
Press any key to continue.
```

Result:

Thus, the program was successfully compiled and run.