Experiment No:1

Aim:- Design and implement a lexical analyzer for a given language using C and the lexical analyzer should ignore redundant spaces, tabs and newlines.

Code:-

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>
#define MAX_TOKEN_SIZE 100
// Token types enumeration
typedef enum {
   IDENTIFIER,
   NUMBER,
   OPERATOR,
   PUNCTUATOR,
   KEYWORD,
   INVALID
} TokenType;
// Token structure
typedef struct {
   TokenType type;
   char lexeme[MAX_TOKEN_SIZE];
} Token;
// Function to check if a character is whitespace
int isWhitespace(char c) {
   return (c == ' ' || c == '\t' || c == '\n');
}
// Function to check if a character is a valid part of an identifier
int isValidIdentifierChar(char c) {
   return (isalnum(c) || c == '_');
}
// Function to recognize and print tokens
void lexicalAnalyzer(char *input) {
   int i = 0;
   while (input[i] != '\0') {
      // Ignore whitespace
      while (isWhitespace(input[i])) {
         i++;
      }
      // Check for identifiers or keywords
      if (isalpha(input[i]) || input[i] == '_') {
         char lexeme[MAX_TOKEN_SIZE];
         int j = 0;
```

```
        while (isValidIdentifierChar(input[i])) {
            lexeme[j++] = input[i++];
        }
        lexeme[j] = '\0';
        // Check if the lexeme is a keyword
        if (strcmp(lexeme, "if") == 0 || strcmp(lexeme, "else") == 0 || strcmp(lexeme, "int") == 0) {
            printf("Keyword: %s\n", lexeme);
        } else {
            printf("Identifier: %s\n", lexeme);
        }
    }// Check for numbers
    else if (isdigit(input[i])) {
        char lexeme[MAX_TOKEN_SIZE];
        int j = 0;
        while (isdigit(input[i])) {
            lexeme[j++] = input[i++];
        }
        lexeme[j] = '\0';
        printf("Number: %s\n", lexeme);
    }// Check for operators
    else if (input[i] == '+' || input[i] == '-' || input[i] == '*' || input[i] == '/') {
        printf("Operator: %c\n", input[i++]);
    }// Check for punctuators
    else if (input[i] == '(' || input[i] == ')' || input[i] == '{' || input[i] == '}' || input[i] == ';') {
        printf("Punctuator: %c\n", input[i++]);
    } else {
        i++;
    }
  }
}
int main() {
  // input
  char input[] = "int main() {\n\tint x = 10;\n\tif (x > 5) {\n\t\n printf(\"Hello, World!\");\n\t}\n\t return
0;\n}";
  lexicalAnalyzer(input);
  return 0;
}
```

Input:-

```
int main() {
        int x = 10;
        if (x > 5) {
                printf("Hello, World!");
        }
        return 0;
}
```

Output:-

```
Keyword: int
Identifier: main
Punctuator: (
Punctuator: )
Punctuator: {
Keyword: int
Identifier: x
Number: 10
Punctuator: ;
Keyword: if
Punctuator: (
Identifier: x
Number: 5
Punctuator: )
Punctuator: {
Identifier: printf
Punctuator: (
Identifier: Hello
Identifier: World
Punctuator: )
Punctuator: ;
Punctuator: }
Identifier: return
Number: 0
Punctuator: ;
Punctuator: }
```

Experiment No:2

Aim:-Implementation of Lexical Analyzer using Lex Tool.

Code:-

```
%{
#include <stdio.h>
%}
%option noyywrap
%%
"int"               { printf("Keyword: %s\n", yytext); }
[a-zA-Z_][a-zA-Z0-9_]*  { printf("Identifier: %s\n", yytext); }
[0-9]+              { printf("Number: %s\n", yytext); }
"="                 { printf("Operator: %s\n", yytext); }
[+*\/-]             { printf("Operator: %s\n", yytext); }
"printf"            { printf("Function: %s\n", yytext); }
"(" | ")" | "{" | "}" | ";"  { printf("Punctuator: %s\n", yytext); }
"return"            { printf("Keyword: %s\n", yytext); }
"%%"                { printf("Lex rule delimiter\n"); }
.                   { printf("Invalid token: %s\n", yytext); }
%%
int main() {
   yylex();
   return 0;
}
```

Input:-

```
int main() {
   int a=5,b=10;
   int sum=5+10;
   printf("%d",sum);
   return 0;
}
```

Output:-

```
Keyword: int
Identifier: main
Punctuator: (
Punctuator: )
Punctuator: {
Keyword: int
Identifier: a
Operator: =
Number: 5
Punctuator: ,
Identifier: b
Operator: =
Number: 10
```

```
Punctuator: ;
Keyword: int
Identifier: sum
Operator: =
Number: 5
Operator: +
Number: 10
Punctuator: ;
Function: printf
Punctuator: (
String: "%d"
Punctuator: ,
Identifier: sum
Punctuator: )
Punctuator: ;
Keyword: return
Number: 0
Punctuator: ;
Punctuator: }
```

Experiment No:3(a)

Aim:-Program to recognize a valid arithmetic expression that uses operator +, – , * and /.

Code:-

```
%{
#include <stdio.h>
#include <stdlib.h>
void push(char c);
char pop();
int is_empty();
int error_flag = 0;
%}
%option noyywrap
%union {
   char operator;
}

%token <operator> ADD_OP SUB_OP MUL_OP DIV_OP
%token NUMBER LPAREN RPAREN
%%
expr    : term
       | expr ADD_OP term
       | expr SUB_OP term
       ;
term    : factor
       | term MUL_OP factor
       | term DIV_OP factor
       ;
factor  : NUMBER
       | LPAREN expr RPAREN
       ;
%%
// Stack to keep track of parentheses
char stack[100];
int top = -1;
void push(char c) {
   if (top == 99) {
      fprintf(stderr, "Stack overflow\n");
      exit(EXIT_FAILURE);
   }
   stack[++top] = c;
}
char pop() {
   if (top == -1) {
      fprintf(stderr, "Stack underflow\n");
```

```
        exit(EXIT_FAILURE);
    }
    return stack[top--];
}
int is_empty() {
    return top == -1;
}
int main() {
    yyparse();
    if (is_empty() && !error_flag) {
        printf("The given arithmetic expression is valid.\n");
    } else {
        printf("The given arithmetic expression is not valid.\n");
    }
    return 0;
}
int yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
    error_flag = 1;
    return 0;
}
```

Input:-

(3+2)*5

Output:-

The given arithmetic expression is valid.

Experiment No:3(b)

Aim:- Program to recognize a valid variable which starts with a letter followed by any number of letters or digits.

CODE:-

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

%option noyywrap

%%
[A-Za-z][A-Za-z0-9]* {
   printf("Valid Variable: %s\n", yytext);
}

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

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

Input:-

```
a123
123a
```

Output:-

```
Valid Variable: a123
Invalid Token: 123a
```

Experiment No:3(c)

Aim:-Implementation of Calculator using LEX and YACC.

Code:-

```
//in calc.l for lex
%{
#include "y.tab.h"
%}
%%
[0-9]+        { yylval = atoi(yytext); return NUMBER; }
[-+*/()]      { return yytext[0]; }
\n            { return EOL; }
[ \t]         ; /* Ignore whitespace */
.             { printf("Invalid character: %s\n", yytext); }
%%
int yywrap(void) {
   return 1;
}

//In calc.y for yacc
%{
#include <stdio.h>
%}
%token NUMBER
%token EOL
%%
stmt_list:
        | stmt_list statement EOL { printf("Result: %d\n", $2); }
        ;
statement: expr { $$ = $1; }
        ;
expr: expr '+' term { $$ = $1 + $3; }
   | expr '-' term { $$ = $1 - $3; }
   | term          { $$ = $1; }
   ;
term: term '*' factor { $$ = $1 * $3; }
   | term '/' factor {
      if ($3 != 0)
         $$ = $1 / $3;
      else {
         yyerror("Division by zero");
         $$ = 0;
      }
   }
   | factor         { $$ = $1; }
```

```
   ;
factor: NUMBER { $$ = $1; }
     | '(' expr ')' { $$ = $2; }
     ;
%%
int main() {
   yyparse();
   return 0;
}

void yyerror(const char* msg) {
   fprintf(stderr, "Error: %s\n", msg);
}
```

Input:-

2 + 5*(3-1)

Output:- 12

Experiment No:3(d)

Aim:-Convert the BNF rules into YACC form and write code to generate abstract syntax tree.

Code:-

```
//code to convert in ast
%{
#include <stdio.h>
#include <stdlib.h>
struct Node {
   char* value;
   struct Node* left;
   struct Node* right;
};
struct Node* createNode(char* value, struct Node* left, struct Node* right) {
   struct Node* node = (struct Node*)malloc(sizeof(struct Node));
   node->value = value;
   node->left = left;
   node->right = right;
   return node;
}
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%%
expr    : expr '+' term   { $$ = createNode("+", $1, $3); }
       | expr '-' term   { $$ = createNode("-", $1, $3); }
       | term          { $$ = $1; }
       ;
term    : term '*' factor { $$ = createNode("*", $1, $3); }
       | term '/' factor { $$ = createNode("/", $1, $3); }
       | factor        { $$ = $1; }
       ;
factor  : '(' expr ')'    { $$ = $2; }
       | NUMBER        { $$ = createNode($1, NULL, NULL); }
       ;

%%
int main() {
   yyparse();
   return 0;
}
void yyerror(const char* msg) {
   fprintf(stderr, "Error: %s\n", msg);
}
```

Input:-

(2 + 3) * 5

Output:-

```
*
| \
+  5
| \
2  3
```

Experiment No:4

Aim :- Write a program to find ε – closure of all states of any given NFA with ε transition.

Code:-

```
 #include <stdio.h>
#include <stdbool.h>
#define MAX_STATES 50
#define MAX_TRANSITIONS 10
struct State {
   char stateId;
   char epsilonTransitions[MAX_TRANSITIONS];
};
void computeEpsilonClosure(struct State nfa[], int numStates, char stateId, bool visited[]) {
   visited[stateId - 'A'] = true;
   for (int i = 0; nfa[stateId - 'A'].epsilonTransitions[i] != 'X'; i++) {
      char nextState = nfa[stateId - 'A'].epsilonTransitions[i];
      if (!visited[nextState - 'A']) {
         computeEpsilonClosure(nfa, numStates, nextState, visited);
      }
   }
}
void printEpsilonClosure(struct State nfa[], int numStates) {
   for (int i = 0; i < numStates; i++) {
      printf("ε-closure(%c): ", nfa[i].stateId);
      bool visited[MAX_STATES] = {false};
      computeEpsilonClosure(nfa, numStates, nfa[i].stateId, visited);
      // Print the ε-closure for the current state
      for (int j = 0; j < numStates; j++) {
         if (visited[j]) {
            printf("%c ", nfa[j].stateId);
         }
      }
      printf("\n");
   }
}
int main() {
   int numStates;
   printf("Enter the number of states: ");
   scanf("%d", &numStates);
   struct State nfa[MAX_STATES];
   for (int i = 0; i < numStates; i++) {
      printf("Enter transitions for state %c (use 'ε' for epsilon, type 'X' to end): ", 'A' + i);
      nfa[i].stateId = 'A' + i;
      int j = 0;
      while (true) {
```

```
            scanf(" %c", &nfa[i].epsilonTransitions[j]);
            if (nfa[i].epsilonTransitions[j] == 'X') {
                break;
            }
            j++;
        }
    }
    // Print the ε-closure of all states
    printEpsilonClosure(nfa, numStates);
    return 0;
}
```

Input:-

```
Enter the number of states: 3
Enter transitions for state A (use 'ε' for epsilon, type 'X' to end): ε B C X
Enter transitions for state B (use 'ε' for epsilon, type 'X' to end): a C X
Enter transitions for state C (use 'ε' for epsilon, type 'X' to end): b A X
```

Output:-

```
ε-closure(A): A B C
ε-closure(B): B C
ε-closure(C): C A B
```

Experiment No: 5

Aim:- Write a program to convert NFA with ε transition to NFA without ε transition.

Code:-

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_STATES 50
#define MAX_SYMBOLS 10
struct State {
   char stateId;
   char transitions[MAX_SYMBOLS][MAX_STATES];
   int numTransitions[MAX_SYMBOLS];
};
void computeEpsilonClosure(struct State nfa[], int numStates, char stateId, bool visited[], bool
epsilonClosure[]) {
   visited[stateId - 'A'] = true;
   epsilonClosure[stateId - 'A'] = true;
   for (int i = 0; i < nfa[stateId - 'A'].numTransitions['e' - 'a']; i++) {
      char nextState = nfa[stateId - 'A'].transitions['e' - 'a'][i];
      if (!visited[nextState - 'A']) {
         computeEpsilonClosure(nfa, numStates, nextState, visited, epsilonClosure);
      }
   }
}
void removeEpsilonTransitions(struct State nfa[], int numStates) {
   for (int i = 0; i < numStates; i++) {
      printf("State %c: ", nfa[i].stateId);
      bool visited[MAX_STATES] = {false};
      bool epsilonClosure[MAX_STATES] = {false};
      computeEpsilonClosure(nfa, numStates, nfa[i].stateId, visited, epsilonClosure);
      for (int j = 0; j < numStates; j++) {
         if (epsilonClosure[j]) {
            printf("%c ", nfa[j].stateId);
         }
      }
      printf("\n");
   }
}
int main() {
   int numStates;
   printf("Enter the number of states: ");
   scanf("%d", &numStates);

   struct State nfa[MAX_STATES];
   for (int i = 0; i < numStates; i++) {
```

```
        nfa[i].stateId = 'A' + i;
        printf("Enter transitions for state %c (use 'e' for epsilon, type 'X' to end): ", nfa[i].stateId);
        // Initialize numTransitions array for each state
        for (int j = 0; j < MAX_SYMBOLS; j++) {
            nfa[i].numTransitions[j] = 0;
        }
        int j = 0;
        while (true) {
            scanf(" %c", &nfa[i].transitions[j / numStates][nfa[i].numTransitions[j / numStates]]);
            if (nfa[i].transitions[j / numStates][nfa[i].numTransitions[j / numStates]] == 'X') {
                break;
            }
            nfa[i].numTransitions[j / numStates]++;
            j++;
        }
    }
    removeEpsilonTransitions(nfa, numStates);
    return 0;
}
```

Input:-

```
Enter the number of states: 3
Enter transitions for state A (use 'e' for epsilon, type 'X' to end): e B C X
Enter transitions for state B (use 'e' for epsilon, type 'X' to end): a C X
Enter transitions for state C (use 'e' for epsilon, type 'X' to end): b A X
```

Output:-

```
State A: A B C
State B: B C
State C: C A B
```

Experiment No:6

Aim:- Write a program to convert NFA to DFA.

Code:-

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_STATES 50
#define MAX_SYMBOLS 10
struct NFAState {
   char stateId;
   char transitions[MAX_SYMBOLS][MAX_STATES];
   int numTransitions[MAX_SYMBOLS];
};
struct DFAState {
   char stateId;
   char transitions[MAX_SYMBOLS];
};
bool isStatePresent(struct DFAState dfa[], int numDFAStates, char stateId) {
   for (int i = 0; i < numDFAStates; i++) {
      if (dfa[i].stateId == stateId) {
         return true;
      }
   }
   return false;
}
void computeEpsilonClosure(struct NFAState nfa[], int numStates, char stateId, bool visited[], bool
epsilonClosure[]) {
   visited[stateId - 'A'] = true;
   epsilonClosure[stateId - 'A'] = true;
   for (int i = 0; i < nfa[stateId - 'A'].numTransitions['e' - 'a']; i++) {
      char nextState = nfa[stateId - 'A'].transitions['e' - 'a'][i];
      if (!visited[nextState - 'A']) {
         computeEpsilonClosure(nfa, numStates, nextState, visited, epsilonClosure);
      }
   }
}
void computeNextStates(struct NFAState nfa[], int numStates, char currentStates[], char symbol, bool
visited[], bool epsilonClosure[]) {
      char nextStates[MAX_STATES];
   int numNextStates = 0;
   for (int i = 0; currentStates[i] != '\0'; i++) {
      char currentState = currentStates[i];
      // Compute ε-closure for the current state
      computeEpsilonClosure(nfa, numStates, currentState, visited, epsilonClosure);
      for (int j = 0; j < numStates; j++) {
```

```
            if (epsilonClosure[j] && !isStatePresent(nextStates, numNextStates, nfa[j].stateId)) {
                nextStates[numNextStates++] = nfa[j].stateId;
            }
        }
     for (int j = 0; j < nfa[currentState - 'A'].numTransitions[symbol - 'a']; j++) {
            char nextState = nfa[currentState - 'A'].transitions[symbol - 'a'][j];
            if (!isStatePresent(nextStates, numNextStates, nextState)) {
                nextStates[numNextStates++] = nextState;
            }
        }
    }
    for (int i = 0; i < numNextStates - 1; i++) {
        for (int j = 0; j < numNextStates - i - 1; j++) {
            if (nextStates[j] > nextStates[j + 1]) {
                char temp = nextStates[j];
                nextStates[j] = nextStates[j + 1];
                nextStates[j + 1] = temp;
            }
        }
    }
    printf("{");
    for (int i = 0; i < numNextStates; i++) {
        printf("%c", nextStates[i]);
    }
    printf("} ");
}
void convertNFAEToDFA(struct NFAState nfa[], int numStates, char alphabet[], int numAlphabet, struct
DFAState dfa[], int *numDFAStates) {
    // Initialize the queue for BFS
    char queue[MAX_STATES][MAX_STATES];
    int front = 0, rear = 0;
    bool visited[MAX_STATES] = {false};
    bool epsilonClosure[MAX_STATES] = {false};
    queue[rear][0] = nfa[0].stateId;
    rear++;
    visited[0] = true;
    while (front < rear) {
        char currentStates[MAX_STATES];
        int numCurrentStates = 0;
        for (int i = 0; queue[front][i] != '\0'; i++) {
            currentStates[numCurrentStates++] = queue[front][i];
        }
        front++;
        for (int i = 0; i < numAlphabet; i++) {
            for (int j = 0; j < numStates; j++) {
```

```
            visited[j] = false;
            epsilonClosure[j] = false;
        }
computeNextStates(nfa, numStates, currentStates, alphabet[i], visited, epsilonClosure);
        bool isPresent = false;
        for (int j = 0; j < *numDFAStates; j++) {
            bool match = true;
            for (int k = 0; dfa[j].transitions[i][k] != '\0'; k++) {
                if (dfa[j].transitions[i][k] != currentStates[k]) {
                    match = false;
                    break;
                }
            }
            if (match) {
                isPresent = true;
                break;
            }
        }
        if (!isPresent) {
            for (int j = 0; j < numCurrentStates; j++) {
                queue[rear][j] = currentStates[j];
            }
            for (int j = 0; epsilonClosure[j] != '\0'; j++) {
                queue[rear][numCurrentStates++] = epsilonClosure[j];
            }
            queue[rear][numCurrentStates] = '\0';
            rear++;
            dfa[*numDFAStates].stateId = 'A' + *numDFAStates;
            for (int j = 0; j < numCurrentStates; j++) {
                dfa[*numDFAStates].transitions[i][j] = currentStates[j];
            }
            dfa[*numDFAStates].transitions[i][numCurrentStates] = '\0';
            (*numDFAStates)++;
        }
    }
  }
}
void printDFATransitions(struct DFAState dfa[], int numDFAStates, char alphabet[], int numAlphabet) {
  printf("\nDFA Transitions:\n");
  for (int i = 0; i < numDFAStates; i++) {
    printf("State %c:\n", dfa[i].stateId);
    for (int j = 0; j < numAlphabet; j++) {
      printf("  %c -> {%s}\n", alphabet[j], dfa[i].transitions[j]);
    }
  }
```

```
}
int main() {
   int numStates, numAlphabet;
   printf("Enter the number of states: ");
   scanf("%d", &numStates);
   struct NFAState nfa[MAX_STATES];
   for (int i = 0; i < numStates; i++) {
      nfa[i].stateId = 'A' + i;
      printf("Enter transitions for state %c (use 'e' for epsilon, type 'X' to end): ", nfa[i].stateId);
      for (int j = 0; j < MAX_SYMBOLS; j++) {
         nfa[i].numTransitions[j] = 0;
      }
      int j = 0;
      while (true) {
         scanf(" %c", &nfa[i].transitions[j / numStates][nfa[i].numTransitions[j / numStates]]);
         if (nfa[i].transitions[j / numStates][nfa[i].numTransitions[j / numStates]] == 'X') {
            break;
         }
         nfa[i].numTransitions[j / numStates]++;
         j++;
      }
   }
   printf("Enter the number of symbols in the alphabet: ");
   scanf("%d", &numAlphabet);
   char alphabet[MAX_SYMBOLS];
   printf("Enter the alphabet symbols: ");
   for (int i = 0; i < numAlphabet; i++) {
      scanf(" %c", &alphabet[i]);
   }
   struct DFAState dfa[MAX_STATES];
   int numDFAStates = 0;
   convertNFAEToDFA(nfa, numStates, alphabet, numAlphabet, dfa, &numDFAStates);
   printDFATransitions(dfa, numDFAStates, alphabet, numAlphabet);
   return 0;
}
```

Input:-

```
Enter the number of states: 3
Enter transitions for state A (use e for epsilon, type 'X' to end): e B C X
Enter transitions for state B (use e for epsilon, type 'X' to end): a C X
Enter transitions for state C (use e' for epsilon, type 'X' to end): b A X
Enter the number of symbols in alphabet: 2
Enter the alphabet symbols:a b
```

Output:-

```
DFA Transitions:
State A:
  a -> {A B C}
  b -> {A C}
State B:
  a -> {C}
  b -> {A B}
State C:
  a -> {A B C}
  b -> {A C}
```

Experiment No:7

Aim:- Write a program to minimize any given DFA.

Code:-

```
 #include <stdio.h>
#include <stdbool.h>
#define MAX_STATES 50
#define MAX_SYMBOLS 10
// Structure to represent a state in the DFA
struct DFAState {
   char stateId;
   char transitions[MAX_SYMBOLS];
   bool isFinal;
};
// Function to check if two states are distinguishable
bool areStatesDistinguishable(struct DFAState dfa[], char state1, char state2, char alphabet[], int
numAlphabet) {
   for (int i = 0; i < numAlphabet; i++) {
      char symbol = alphabet[i];
      char nextState1 = dfa[state1 - 'A'].transitions[i];
      char nextState2 = dfa[state2 - 'A'].transitions[i];
      if (nextState1 != nextState2) {
         return true;
      }
   }
   return false;
}
void minimizeDFA(struct DFAState dfa[], char alphabet[], int numAlphabet, int numStates) {
      bool distinguishable[MAX_STATES][MAX_STATES];
   for (int i = 0; i < numStates; i++) {
      for (int j = 0; j < numStates; j++) {
         distinguishable[i][j] = areStatesDistinguishable(dfa, i + 'A', j + 'A', alphabet, numAlphabet);
      }
   }
   bool changed = true;
   while (changed) {
      changed = false;
      for (int i = 0; i < numStates; i++) {
         for (int j = 0; j < numStates; j++) {
            if (!distinguishable[i][j]) {
               for (int k = 0; k < numStates; k++) {
                  if (distinguishable[k][i] != distinguishable[k][j]) {
                     distinguishable[i][j] = true;
                     changed = true;
                     break;
```

```
                }
            }
        }
    }
  }
  printf("\nMinimized DFA Transitions:\n");
  for (int i = 0; i < numStates; i++) {
    printf("State %c:\n", dfa[i].stateId);
    for (int j = 0; j < numAlphabet; j++) {
      char symbol = alphabet[j];
      char nextState = dfa[i].transitions[j];
      if (!distinguishable[i][nextState - 'A']) {
        printf("  %c -> %c\n", symbol, nextState);
      }
    }
    printf("  Final state: %s\n", dfa[i].isFinal ? "Yes" : "No");
  }
}
int main() {
  int numStates, numAlphabet;
  printf("Enter the number of states: ");
  scanf("%d", &numStates);
  struct DFAState dfa[MAX_STATES];
  for (int i = 0; i < numStates; i++) {
    dfa[i].stateId = 'A' + i;
    printf("Enter transitions for state %c:\n", dfa[i].stateId);
    for (int j = 0; j < numAlphabet; j++) {
      printf("  Transition on symbol %c: ", 'a' + j);
      scanf(" %c", &dfa[i].transitions[j]);
    }
    printf("Is state %c a final state? (1 for Yes, 0 for No): ", dfa[i].stateId);
    scanf("%d", &dfa[i].isFinal);
  }
  printf("Enter the number of symbols in the alphabet: ");
  scanf("%d", &numAlphabet);
  char alphabet[MAX_SYMBOLS];
  printf("Enter the alphabet symbols: ");
  for (int i = 0; i < numAlphabet; i++) {
    scanf(" %c", &alphabet[i]);
  }
  // Minimize the DFA and print the result
  minimizeDFA(dfa, alphabet, numAlphabet, numStates);
  return 0;
}
```

Input:-

```
Enter the number of states: 3
Enter transitions for state A:
  Transition on symbol a: B
  Transition on symbol b: A
  Final state? (1 for Yes, 0 for No): 0
Enter transitions for state B:
  Transition on symbol a: C
  Transition on symbol b: B
  Final state? (1 for Yes, 0 for No): 0
Enter transitions for state C:
  Transition on symbol a: C
  Transition on symbol b: B
  Final state? (1 for Yes, 0 for No): 1
Enter the number of symbols in the alphabet: 2
Enter the alphabet symbols: a b
```

Output-

```
Minimized DFA Transitions:
State A:
  a -> B
  Final state: No
  b -> A
  Final state: No
State B:
  a -> C
  Final state: No
  b -> B
  Final state: No
State C:
  a -> C
  Final state: Yes
  b -> B
  Final state: No
```

Experiment No:8

Aim:-Write a program to find first and follow of any given grammar.

Code:-

```
#include <ctype.h>
#include <stdio.h>
#include <string.h>
void followfirst(char, int, int);
void follow(char c);
void findfirst(char, int, int);
int count, n = 0;
char calc_first[10][100];
char calc_follow[10][100];
int m = 0;
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;
int main(int argc, char** argv){
        int jm = 0;
        int km = 0;
        int i, choice;
        char c, ch;
        count = 8;
        strcpy(production[0], "X=TnS");
        strcpy(production[1], "X=Rm");
        strcpy(production[2], "T=q");
        strcpy(production[3], "T=#");
        strcpy(production[4], "S=p");
        strcpy(production[5], "S=#");
        strcpy(production[6], "R=om");
        strcpy(production[7], "R=ST");
        int kay;
        char done[count];
        int ptr = -1;

        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;
		for (kay = 0; kay <= ptr; kay++)
			if (c == done[kay])
				xxx = 1;
		if (xxx == 1)
			continue;
		findfirst(c, 0, 0);
		ptr += 1;
		done[ptr] = c;
		printf("\n First(%c) = { ", c);
		calc_first[point1][point2++] = c;
		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++;
	}
	printf("\n");
	char donee[count];
	ptr = -1;
	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;
		for (kay = 0; kay <= ptr; kay++)
```

```
                        if (ck == donee[kay])
                                xxx = 1;

                if (xxx == 1)
                        continue;
                land += 1;
                follow(ck);
                ptr += 1;
                donee[ptr] = ck;
                printf(" Follow(%c) = { ", ck);
                calc_follow[point1][point2++] = ck;
                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;
        if (production[0][0] == c) {
                f[m++] = '$';
        }
        for (i = 0; i < 10; i++) {
                for (j = 2; j < 10; j++) {
                        if (production[i][j] == c) {
                                if (production[i][j + 1] != '\0') {
                                        followfirst(production[i][j + 1], i,(j + 2));
                                }
                                if (production[i][j + 1] == '\0'&& c != production[i][0]) {
                                        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;
	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;
		}
		while (calc_first[i][j] != '!') {
			if (calc_first[i][j] != '#') {
				f[m++] = calc_first[i][j];
			}
			else {
				if (production[c1][c2] == '\0') {
					follow(production[c1][0]);
				}
```

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

Output:-

```
First(X) = { q, n, o, p, #, }
First(T) = { q, #, }
First(S) = { p, #, }
First(R) = { o, p, q, #, }
Follow(X) = { $,  }
Follow(T) = { n, m,  }
Follow(S) = { $, q, m,  }
Follow(R) = { m,  }
```