# GOVT. MODEL ENGINEERING COLLEGE

**THRIKKAKKARA, ERNAKULAM**


**CSL411 COMPILER LAB**

**NAME : PRADYUMN R PAI**

**BRANCH : COMPUTER SCIENCE AND ENGINEERING**

**SEMESTER : 7th** **ROLL NO : 22CSA50**

***Certified that this is the bonafide work done by***

. PRADYUMN R PAI

***Staff- in Charge*** ***Head of the Department***

Register No: MDL22CS154 Date:

Year and month: August 2025 Thrikkakkara

**Internal Examiner** **External Examiner**

# INDEX

| SL.NO. | DATE | NAME OF THE PROGRAM | REMARKS |
|---|---|---|---|
| 1.1 | 07/08/2025 | Lexical Analyser | |
| 1.2 | 07/08/2025 | Epsilon Closure | |
| 1.3 | 07/08/2025 | Epsilon NFA to NFA conversion | |
| 1.4 | 07/08/2025 | NFA to DFA conversion | |
| 1.5 | 07/08/2025 | DFA minimization | |
| 2.1 | 07/08/2025 | Name matching | |
| 2.2 | 07/08/2025 | Valid Identifier | |
| 2.3 | 07/08/2025 | Calculator | |
| 2.4 | 07/08/2025 | Abstract Syntax Tree | |
| 2.5 | 07/08/2025 | Valid for loop | |

**Name:** Pradyumn R Pai
**Roll No:** 50
**Class:** CS7A

# PROGRAM CODE

```
#include <stdio.h>

#include <string.h>

#include <stdbool.h>

#include <ctype.h> //For isalnum, isalpha, isdigit, isspace


typedef enum {
    TOKEN_KEYWORD,
    TOKEN_IDENTIFIER,
    TOKEN_INT_CONST,
    TOKEN_STRING_LITERAL,
    TOKEN_OPERATOR,
    TOKEN_PUNCTUATOR,
    TOKEN_UNKNOWN,
    TOKEN_DIRECTIVE,
    TOKEN_END
} TokenType;


const char* TokenTypeNames[] = {
    "Keyword",
    "Identifier",
    "Integer Constant",
    "String Literal",
    "Operator",
    "Punctuator",
    "Unknown",
    "Compiler Directive"
};
```

**Date:** 07/08/2024

# Experiment 1.1

## AIM

To develop lexical analyzer using C

## ALGORITHM

1. Start
2. Define token type and names. For C, we are considering Keywords, Identifiers, Integer constants, String literals, Operators, Punctuators, Compiler directives and token representing end of input.
3. Define arrays containing list of operators, keywords, and punctuators.
4. Define helper function isInteger to verify whether a string is a valid integer.
5. Define helper function isPunctuator to verify whether a character is a punctuator.
6. Define helper function isKeyword to verify whether a string is a keyword.
7. Define helper function isOperator to verify whether a string is an operator.
8. Use the above functions to define helper function identifierParse to determine if a given buffer contains a keyword, integer constant, a valid identifier, or is unknown.
9. Define function getToken to extract token from stdin as follows:
    1. Reset the buffer.
    2. Read characters one by one using getchar and do the following:
        1. If current caracter is EOF:
            1. If buffer is empty, return END token.
            2. Else, parse buffer and return token using identifierParse function.
        2. If the input is at a newline starting from #, this indicates a compiler directive.
            1. Read the whole line and store in buffer.
            2. Return DIRECTIVE token.
        3. If the current character is a punctuator:
            1. If buffer is empty, store punctuator in buffer and return PUNCTUATOR token.
            2. Else, move input pointer backward and parse and return current contents of the buffer using identifierParse.

```
/* operators match the regex: (>>=)|(<<=)|(\/=)|(->)|(%=)|(\*=)|(&=)|(--)|(-=)|(\+\+)|(\^=)|
(&&)|(>=)|(<<)|(<=)|(\|\|)|(>>)|(!=)|(\+=)|(==)|(\|=)|[>|+/=%&^*!~.\-<] */

char operators[][10] = {"+","-","*","/","%","<",">","!","&","|","^","~","=",".","+
+","--","<=",">=","==","!=","&&","||","<<",">>","+=","-=","*=","/=","%=","&=","|=","^=","-
>",">>=","<<="};

char punctuators[] = "{}[](),;:.?";


char keywords[][10] = {"auto", "break", "case", "char", "const", "continue", "default", "do",
"double", "else", "enum", "extern", "float", "for", "goto", "if", "int", "long", "register",
"return", "short", "signed", "sizeof", "static", "struct", "switch", "typedef", "union",
"unsigned", "void", "volatile", "while"};


char buffer[1024] = "";

int bufferLength = 0;

bool atNewLine = true;


bool isSubset(char* buffer, char charSet[][10], int len) {
   for (int i=0;i<len;++i){
      if (strcmp(buffer,charSet[i])==0){
         return true;
      }
   }
   return false;
}


bool isInteger(char* buffer){
   if (buffer[0]=='0' && buffer[1]=='\0') return true;
   if (buffer[0]=='0') return false;


   int n = strlen(buffer);
   for (int i=0;i<n;++i){
      if (!isdigit(buffer[i])) return false;
   }
```

4. If current character is white space:
    1. If buffer is empty, ignore the character and go to next iteration of the loop.
    2. Else, parse and return current contents of the buffer using identifierParse.
5. If the current character is a forward slash:
    1. Look ahead one character to see if this is beginning of a comment.
    2. If next character is a ‘/’, then read till newline and skip the characters.
    3. If next character is a ‘*’, read till “*/” is encountered and skip the characters.
    4. If no comment is found, move input pointer backward so the next character can be processed as usual. Otherwise, go to next iteration of the loop.
6. Add current character to buffer.
7. If buffer contains a valid operator:
    1. Read into buffer toll it contains at least 3 characters (The maximum length of an operator)
    2. Remove last character and move input pointer backward until the buffer has a valid operator.
    3. This ensures that larger operators are considered first.
8. If the current character is a double quote:
    1. Read characters and store in buffer till another double quote character is encountered.
    2. Return token STRING_LITERAL.
10. Create main function to continuously get tokens until TOKEN_END is returned and print the type and value of each token.
11. Stop

```
    return true;
}

bool isKeyword(char* buffer){
    return isSubset(buffer,keywords,sizeof(keywords)/sizeof(keywords[0]));
}

bool isPunctuator(char c){
    int n = sizeof(punctuators);
    for (int i=0;i<n;++i){
        if (c==punctuators[i]) return true;
    }
    return false;
}

bool isOperator(char* buffer){
    return isSubset(buffer,operators,sizeof(operators)/sizeof(operators[0]));
}

TokenType identifierParse(char* buffer){
    int n = strlen(buffer);
    bool validFlag = false;
    for (int i=0;i<n;++i){
        if (!isalnum(buffer[i]) && buffer[i]!='_'){
            if (i==0) return TOKEN_UNKNOWN;
            for (int j=n-1;j>=i;--j){
                ungetc(buffer[j],stdin);
            }
            buffer[i] = '\0';
            validFlag = true;
            break;
```

```
        }
    }
    if (isInteger(buffer)) return TOKEN_INT_CONST;
    if (isKeyword(buffer)) return TOKEN_KEYWORD;
    if (isdigit(buffer[0])) return TOKEN_UNKNOWN; //Ensure first digit is alphabet or _
    return TOKEN_IDENTIFIER;
}

TokenType getToken(){
    //Reset buffer
    buffer[0] = '\0';
    bufferLength = 0;

    while (true){
        char c = getchar();
        if (c=='\n'){
            atNewLine = true;
        }

        if (c==EOF){
            if (bufferLength==0) return TOKEN_END;
            ungetc(c,stdin);
            return identifierParse(buffer);
        }

        //Handle compiler directives
        if (atNewLine && c=='#'){
            if (bufferLength!=0){
                ungetc(c,stdin);
                return identifierParse(buffer);
            }
```

```
    buffer[bufferLength++] = c;
    while (true) {
        c = getchar();
        if (c==EOF) {
            if (bufferLength==0) return TOKEN_END;
            ungetc(c,stdin);
            return identifierParse(buffer);
        }
        if (c=='\n') break;
        buffer[bufferLength++] = c;
    }
    buffer[bufferLength] = '\0';
    return TOKEN_DIRECTIVE;
}

// Handle punctuators
if (isPunctuator(c)){
    if (bufferLength==0){
        buffer[0] = c;
        buffer[1] = '\0';
        return TOKEN_PUNCTUATOR;
    }
    ungetc(c,stdin);
    return identifierParse(buffer);
}

// Handle white space
if (isspace(c)){
    if (bufferLength!=0) {
        return identifierParse(buffer);
```

```
    }
    continue;
}

// Handle comments
if (c=='/'){
    char n = getchar();
    bool commentFlag = true;
    if (n=='/'){
        while (commentFlag){
            c = getchar();
            if (c==EOF) break;
            if (c=='\n') commentFlag = false;
        }
        continue;
    } else if (n=='*'){
        while (commentFlag) {
            c = n;
            n = getchar();
            if (n==EOF) break;
            if (c=='*' && n=='/') commentFlag = false;
        }
        continue;
    } else {
        ungetc(n,stdin);
    }
    if (n==EOF){
        return TOKEN_END;
    }
    if (!commentFlag) continue;
}
```

```
        buffer[bufferLength++] = c;
        buffer[bufferLength] = '\0';

        if (isOperator(buffer)){
            while (bufferLength<3){
                buffer[bufferLength++] = getchar();
                buffer[bufferLength] = '\0';
            }
            while (bufferLength>1 && !isOperator(buffer)){
                ungetc(buffer[--bufferLength],stdin);
                buffer[bufferLength] = '\0';
            }
            return TOKEN_OPERATOR;
        }

        if (c==""){
            while ((c=getchar())!=""){
                if (c==EOF){
                    ungetc(c,stdin);
                    return identifierParse(buffer);
                }
                buffer[bufferLength++] = c;
            }
            buffer[bufferLength] = "";
            buffer[++bufferLength] = '\0';
            return TOKEN_STRING_LITERAL;
        }
    }

}
```

```
int main(){
    TokenType currentToken;
    while ((currentToken=getToken())!=TOKEN_END){
        printf("<%s,%s>\n",TokenTypeNames[currentToken],buffer);
    }
}
```

## OUTPUT:

**Input.txt:**

```
#include <stdio.h>

/* Multi
Line Comment*/
int main() {
    //Single line comment
    int arr[2] = {1, 2};
    int a,b,c;
    int *p = &a;
    c = a + b;
    a++;
    c += a;
    c = (a && b);
    return 0;
}
```

**output:**

```
<Compiler Directive,#include <stdio.h>>
<Keyword,int>
<Identifier,main>
<Punctuator,(>
```

<Punctuator,)>

<Punctuator,{>

<Keyword,int>

<Identifier,arr>

<Punctuator,[>

<Integer Constant,2>

<Punctuator,]>

<Operator,=>

<Punctuator,{>

<Integer Constant,1>

<Punctuator,,>

<Integer Constant,2>

<Punctuator,}>

<Punctuator,;>

<Keyword,int>

<Identifier,a>

<Punctuator,,>

<Identifier,b>

<Punctuator,,>

<Identifier,c>

<Punctuator,;>

<Keyword,int>

<Operator,*>

<Identifier,p>

<Operator,=>

<Operator,&>

<Identifier,a>

<Punctuator,;>

<Identifier,c>

<Operator,=>

<Identifier,a>

<Operator,+>

<Identifier,b>

<Punctuator,;>

<Identifier,a>

<Operator,++>

<Punctuator,;>

<Identifier,c>

<Operator,+=>

<Identifier,a>

<Punctuator,;>

<Identifier,c>

<Operator,=>

<Punctuator,(>

<Identifier,a>

<Operator,&&>

<Identifier,b>

<Punctuator,)>

<Punctuator,;>

<Keyword,return>

<Integer Constant,0>

<Punctuator,;>

<Punctuator,}>

## RESULT

Successfully performed lexical analysis of given C program.

**Name:** Pradyumn R Pai
**Roll No:** 50
**Class:** CS7A

# PROGRAM CODE

**nfa_ds.c:**

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

struct TransitionNode {
    int target_state;
    char input;
    struct TransitionNode* next;
};

struct State {
    int id;
    struct TransitionNode* transitionListHead;
    bool finalState;
};

struct NFA {
    int stateNum;
    char * inputAlphabet;
    struct State* stateList;
};

struct NFA* init_NFA(int n, char* inputAlphabet){
    struct NFA* out = malloc(sizeof(struct NFA));
    if (!out){
        return NULL; //failed allocation
    }
```

**Date:** 07/08/2024

# Experiment 1.2

## AIM

To find ε – closure of all states of any given NFA with ε transition

## ALGORITHM

1. Start
2. Create utility functions for NFA data structure to read and write transitions.
3. Read NFA input as follows:
    1. The first line contains the number of states (n) , number of final states (f) , number of input alphabets(m), and number of transitions(t).
    2. The next line contains f space separated integers denoting the final states.
    3. The next line contains the m input alphabets as a single string.
    4. The next t lines contain transitions as “qi qj c” representing a transition from qi to qj on input alphabet c. Here, the alphabet ‘e’ denotes epsilon.
4. Create a DFS function that finds epsilon closure of a state and stores in a boolean array as follows:
    1. If state is visited, terminate function call.
    2. Mark state as visited.
    3. For each transition from the given state via input alphabet epsilon:
        1. Recursively call the DFS function for the target state.
5. For each state in the epsilon NFA, find the epsilon closure using DFS.
6. Print the set of states in the epsilon closure of each state.
7. Stop

```
    out->stateNum = n;
    out->inputAlphabet = inputAlphabet;
    out->stateList = malloc(sizeof(struct State)*n);
    if (!out->stateList){
        free(out);
        return NULL;
    }

    for (int i=0;i<n;++i){
        out->stateList[i].id = i;
        out->stateList[i].transitionListHead = NULL;
        out->stateList[i].finalState = false;
    }

    return out;
}

void addTransitionNFA(struct NFA* n, int s, int t, char c){
    struct TransitionNode** head = &(n->stateList[s].transitionListHead);
    while (*head){
        if ((*head)->input==c && (*head)->target_state==t){
            return ; //avoid duplicates
        }
        head = &((*head)->next);
    }
    *head = malloc(sizeof(struct TransitionNode));
    if (!*head){
        return; // allocation failed
    }
    (*head)->target_state = t;
```

```
    (*head)->input = c;
    (*head)->next = NULL;
}

void freeStateNFA(struct State s){
    struct TransitionNode* head = s.transitionListHead;
    while (head){
        struct TransitionNode* next = head->next;
        free(head);
        head = next;
    }
}

void freeNFA(struct NFA* n){
    if (!n) return;
    for (int i=0;i<(n->stateNum);++i){
        freeStateNFA(n->stateList[i]);
    }

    free(n->inputAlphabet);
    free(n->stateList);
    free(n);
}

void printNFA(struct NFA* nfa){
    printf("The transition table is as follows:\n");
    int n = nfa->stateNum;
    int m = strlen(nfa->inputAlphabet);
    printf("\t");
    for (int i=0;i<m;++i){
        printf("%c\t",nfa->inputAlphabet[i]);
```

```
    }
    printf("epsilon\n");
    for (int i=0;i<n;++i){
        if (i==0){
            printf("->");
        }
        if (nfa->stateList[i].finalState){
            printf("*");
        }
        printf("q%d\t",i);
        struct State s = nfa->stateList[i];
        for (int j=0;j<=m;++j){
            char c = nfa->inputAlphabet[j];
            if (j==m){
                c = 'e';
            }
            for (struct TransitionNode *current=s.transitionListHead;current;current=current->next){
                if (current->input==c){
                    printf("q%d",current->target_state);
                }
            }
            printf("\t");
        }
        printf("\n");
    }
}



struct NFA* readNFA() {
    // read input
```

```
int n, m, t, f;
scanf("%d%d%d%d", &n, &f, &m, &t);
if (f<0 || f>n){
    printf("Invalid number of final states\n");
    return NULL;
}

int finalStates[f];
for (int i=0;i<f;++i){
    scanf("%d",finalStates+i);
    if (finalStates[i]<0 || finalStates[i]>=n){
        printf("Invalid final state %d\n",finalStates[i]);
        return NULL;
    }
}



char* inputChars = malloc(sizeof(char)*(m+1));
if (!inputChars) {
    printf("Failed to allocate memory for input characters\n");
    return NULL;
}

scanf("%s\n", inputChars);
if (strlen(inputChars) != m) {
    free(inputChars);
    printf("Input characters length mismatch\n");
    return NULL;
}

struct NFA *nfa = init_NFA(n,inputChars);
```

```
if (!nfa) {
    free(inputChars);
    printf("Failed to initialize NFA\n");
    return NULL;
}

for (int i=0;i<f;++i){
    nfa->stateList[finalStates[i]].finalState = true;
}

for (int i = 0; i < t; ++i) {
    int a, b;
    char c;
    scanf("q%d q%d %c\n", &a, &b, &c);
    if (a < 0 || a >= n || b < 0 || b >= n) {
        printf("Invalid transition from %d to %d\n", a, b);
        freeNFA(nfa);
        return NULL;
    }
    bool validChar = false;
    for (int j = 0; j < m; ++j) {
        if (inputChars[j] == c) {
            validChar = true;
            break;
        }
    }
    if (!validChar && c != 'e') { // 'e' for epsilon transition
        printf("Invalid input character '%c' for transition from %d to %d\n", c, a, b);
        freeNFA(nfa);
        return NULL;
    }
```

```
        addTransitionNFA(nfa, a, b, c);
    }
    return nfa;
}
```

**enfa_functions.c:**

```
#include "nfa_ds.c"
void dfs_closure(struct NFA* nfa,int state, bool visited[]){
    if (visited[state]) return;
    visited[state] = true;
    for (struct TransitionNode* current = (nfa-
>stateList[state]).transitionListHead;current;current = current->next){
        if (current->input=='e'){
            dfs_closure(nfa,current->target_state,visited);
        }
    }
}

bool* find_epsilon_closure(struct NFA* nfa, int state){
    int n = nfa->stateNum;
    int m = strlen(nfa->inputAlphabet);
    bool* closure = malloc(sizeof(bool)*n);
    for (int i=0;i<n;++i){
        closure[i] = false;
    }
    dfs_closure(nfa,state,closure);
    return closure;
}
```

**epsilon_closure.c:**

```
#include <stdio.h>
#include "enfa_functions.c"
```

```
void print_epsilon_closure(struct NFA* nfa,int state){
    bool* closure = find_epsilon_closure(nfa,state);

    printf("The epsilon closure of state %d is: {",state);
    bool flag = false;
    for (int i=0;i<nfa->stateNum;++i){
        if (!closure[i]) continue;
        if (flag){
            printf(",");
        }
        flag = true;
        printf("q%d",i);
    }
    printf("}\n");

    free(closure);
}

int main(){
    struct NFA* nfa = readNFA();
    if (!nfa) return 1;
    printNFA(nfa);

    // epsilon closure
    for (int i=0;i<nfa->stateNum;++i){
        print_epsilon_closure(nfa,i);
    }

    freeNFA(nfa);
    return 0;
```

```
}
```

## OUTPUT:

**input.txt:**

5 1 2 7

2

01

q0 q1 1

q1 q0 1

q0 q2 e

q2 q3 0

q3 q2 0

q2 q4 1

q4 q2 0

**output:**

The transition table is as follows:

| | 0 | 1 | epsilon |
|---|---|---|---|
| ->q0 | | q1 | q2 |
| q1 | | q0 | |
| *q2 | q3 | | q4 |
| q3 | q2 | | |
| q4 | q2 | | |

The epsilon closure of state 0 is: {q0,q2}

The epsilon closure of state 1 is: {q1}

The epsilon closure of state 2 is: {q2}

The epsilon closure of state 3 is: {q3}

The epsilon closure of state 4 is: {q4}

## RESULT

Successfully calculated epsilon closure of all states.

**Name:** Pradyumn R Pai
**Roll No:** 50
**Class:** CS7A

# PROGRAM CODE

**nfa_ds.c:**

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

struct TransitionNode {
    int target_state;
    char input;
    struct TransitionNode* next;
};

struct State {
    int id;
    struct TransitionNode* transitionListHead;
    bool finalState;
};

struct NFA {
    int stateNum;
    char * inputAlphabet;
    struct State* stateList;
};

struct NFA* init_NFA(int n, char* inputAlphabet){
    struct NFA* out = malloc(sizeof(struct NFA));
    if (!out){
        return NULL; //failed allocation
    }
```

**Date:** 07/08/2024

# Experiment 1.3

## AIM

To convert NFA with ε transition to NFA without ε transition.

## ALGORITHM

1. Start
2. Create utility functions for NFA data structure to read and write transitions.
3. Create a DFS function that finds epsilon closure of a state and stores in a boolean array as follows:
    1. If state is visited, terminate function call.
    2. Mark state as visited.
    3. For each transition from the given state via input alphabet ε:
        1. Recursively call the DFS function for the target state.
4. Read NFA input as follows:
    1. The first line contains the number of states (n) , number of final states (f) , number of input alphabets(m), and number of transitions(t).
    2. The next line contains f space separated integers denoting the final states.
    3. The next line contains the m input alphabets as a single string.
    4. The next t lines contain transitions as “qi qj c” representing a transition from qi to qj on input alphabet c. Here, the alphabet ‘e’ denotes epsilon.
5. For each state in the ε- NFA, find the ε- closure using DFS.
6. Create new NFA with same number of states as input NFA.
7. For each state s in the original NFA:
    1. For each state s’ in the ε- closure of s:
        1. For each transition from s to a state t via input symbol c such that c!= ε :
            1. For each state t’ in the ε- closure of t:
                1. Add transition from s to t’ in the new NFA.
8. For each state s in the original NFA:
    1. For each state s’ in the ε- closure of s:

```
    out->stateNum = n;
    out->inputAlphabet = inputAlphabet;
    out->stateList = malloc(sizeof(struct State)*n);
    if (!out->stateList){
        free(out);
        return NULL;
    }

    for (int i=0;i<n;++i){
        out->stateList[i].id = i;
        out->stateList[i].transitionListHead = NULL;
        out->stateList[i].finalState = false;
    }

    return out;
}

void addTransitionNFA(struct NFA* n, int s, int t, char c){
    struct TransitionNode** head = &(n->stateList[s].transitionListHead);
    while (*head){
        if ((*head)->input==c && (*head)->target_state==t){
            return ; //avoid duplicates
        }
        head = &((*head)->next);
    }
    *head = malloc(sizeof(struct TransitionNode));
    if (!*head){
        return; // allocation failed
    }
    (*head)->target_state = t;
```

    1. If s’ is a final state in original NFA, mark s as a final state in the output NFA.

9. Print the new NFA.
10. Stop

```
    (*head)->input = c;
    (*head)->next = NULL;
}

void freeStateNFA(struct State s){
    struct TransitionNode* head = s.transitionListHead;
    while (head){
        struct TransitionNode* next = head->next;
        free(head);
        head = next;
    }
}

void freeNFA(struct NFA* n){
    if (!n) return;
    for (int i=0;i<(n->stateNum);++i){
        freeStateNFA(n->stateList[i]);
    }

    free(n->inputAlphabet);
    free(n->stateList);
    free(n);
}

void printNFA(struct NFA* nfa){
    printf("The transition table is as follows:\n");
    int n = nfa->stateNum;
    int m = strlen(nfa->inputAlphabet);
    printf("\t");
    for (int i=0;i<m;++i){
        printf("%c\t",nfa->inputAlphabet[i]);
```

```
    }
    printf("epsilon\n");
    for (int i=0;i<n;++i){
        if (i==0){
            printf("->");
        }
        if (nfa->stateList[i].finalState){
            printf("*");
        }
        printf("q%d\t",i);
        struct State s = nfa->stateList[i];
        for (int j=0;j<=m;++j){
            char c = nfa->inputAlphabet[j];
            if (j==m){
                c = 'e';
            }
            for (struct TransitionNode *current=s.transitionListHead;current;current=current->next){
                if (current->input==c){
                    printf("q%d",current->target_state);
                }
            }
            printf("\t");
        }
        printf("\n");
    }
}



struct NFA* readNFA() {
    // read input
```

```
int n, m, t, f;
scanf("%d%d%d%d", &n, &f, &m, &t);
if (f<0 || f>n){
    printf("Invalid number of final states\n");
    return NULL;
}

int finalStates[f];
for (int i=0;i<f;++i){
    scanf("%d",finalStates+i);
    if (finalStates[i]<0 || finalStates[i]>=n){
        printf("Invalid final state %d\n",finalStates[i]);
        return NULL;
    }
}



char* inputChars = malloc(sizeof(char)*(m+1));
if (!inputChars) {
    printf("Failed to allocate memory for input characters\n");
    return NULL;
}

scanf("%s\n", inputChars);
if (strlen(inputChars) != m) {
    free(inputChars);
    printf("Input characters length mismatch\n");
    return NULL;
}

struct NFA *nfa = init_NFA(n,inputChars);
```

```
if (!nfa) {
    free(inputChars);
    printf("Failed to initialize NFA\n");
    return NULL;
}

for (int i=0;i<f;++i){
    nfa->stateList[finalStates[i]].finalState = true;
}

for (int i = 0; i < t; ++i) {
    int a, b;
    char c;
    scanf("q%d q%d %c\n", &a, &b, &c);
    if (a < 0 || a >= n || b < 0 || b >= n) {
        printf("Invalid transition from %d to %d\n", a, b);
        freeNFA(nfa);
        return NULL;
    }
    bool validChar = false;
    for (int j = 0; j < m; ++j) {
        if (inputChars[j] == c) {
            validChar = true;
            break;
        }
    }
    if (!validChar && c != 'e') { // 'e' for epsilon transition
        printf("Invalid input character '%c' for transition from %d to %d\n", c, a, b);
        freeNFA(nfa);
        return NULL;
    }
```

```
        addTransitionNFA(nfa, a, b, c);
    }
    return nfa;
}
```

**enfa_functions.c:**

```
#include "nfa_ds.c"
void dfs_closure(struct NFA* nfa,int state, bool visited[]){
    if (visited[state]) return;
    visited[state] = true;
    for (struct TransitionNode* current = (nfa->stateList[state]).transitionListHead;current;current = current->next){
        if (current->input=='e'){
            dfs_closure(nfa,current->target_state,visited);
        }
    }
}

bool* find_epsilon_closure(struct NFA* nfa, int state){
    int n = nfa->stateNum;
    int m = strlen(nfa->inputAlphabet);
    bool* closure = malloc(sizeof(bool)*n);
    for (int i=0;i<n;++i){
        closure[i] = false;
    }
    dfs_closure(nfa,state,closure);
    return closure;
}

struct NFA* epsilon_removal(struct NFA* enfa){
    int n = enfa->stateNum;
    int m = strlen(enfa->inputAlphabet);
```

```
    char* inputAlphabet = malloc(sizeof(char)*(m+1));
    if (!inputAlphabet){
        return NULL;
    }

    strcpy(inputAlphabet,enfa->inputAlphabet);
    struct NFA* outNFA = init_NFA(n,inputAlphabet);
    if (!outNFA){
        return NULL;
    }

    bool* closure_matrix[n]; //matrix[a][b] means that b is part of epsilon closure of a
    for (int i=0;i<n;++i){
        closure_matrix[i] = find_epsilon_closure(enfa,i);
    }

    for (int s=0;s<n;++s){
        for (int s1=0;s1<n;++s1){
            if (!closure_matrix[s][s1]) continue;
            for (struct TransitionNode* current = (enfa->stateList[s1]).transitionListHead;current;current = current->next){
                if (current->input=='e') continue;
                // addTransitionNFA(outNFA,s,current->target_state,current->input);
                int t = current->target_state;
                char c = current->input;
                for (int t1=0;t1<n;++t1){
                    if (!closure_matrix[t][t1]) continue;
                    //t1 is a state part of epsilon closure of t
                    addTransitionNFA(outNFA,s,t1,c);
                }
            }
```

```
        }
    }

    for (int i=0;i<n;++i){
        for (int j=0;j<n;++j){
            if (!enfa->stateList[j].finalState) continue;
            if (!closure_matrix[i][j]) continue;
            outNFA->stateList[i].finalState = true;
        }
    }

    for (int i=0;i<n;++i){
        free(closure_matrix[i]);
    }

    return outNFA;
}
```

**epsilon_removal.c:**

```
#include <stdio.h>
#include "enfa_functions.c"

int main(){
    struct NFA* enfa = readNFA();
    if (!enfa) return 1;
    printNFA(enfa);

    // epsilon removal
    struct NFA* nfa = epsilon_removal(enfa);
    if (!nfa){
        printf("Epsilon removal failes\n");
        freeNFA(enfa);
```

```
        return 1;
    }
    printf("\n\nAfter epsilon removal...\n");
    printNFA(nfa);

    freeNFA(enfa);
    freeNFA(nfa);
    return 0;
}
```

## OUTPUT:

**input.txt:**

5 1 2 7

2

01

q0 q1 1

q1 q0 1

q0 q2 e

q2 q3 0

q3 q2 0

q2 q4 1

q4 q2 0

**output:**

The transition table is as follows:

| | 0 | 1 | epsilon |
|---|---|---|---|
| ->q0 | | q1 | q2 |
| q1 | | q0 | |
| *q2 | q3 | q4 | |
| q3 | q2 | | |
| q4 | q2 | | |

After epsilon removal...

The transition table is as follows:

| | 0 | 1 | epsilon |
|---|---|---|---|
| ->*q0 | q3 | q1q4 | |
| q1 | | q0q2 | |
| *q2 | q3 | q4 | |
| q3 | q2 | | |
| q4 | q2 | | |

## RESULT

Successfully converted NFA with ε transition to NFA without  ε transition.

**Name:** Pradyumn R Pai
**Roll No:** 50
**Class:** CS7A

# PROGRAM CODE

**nfa_ds.c:**

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

struct TransitionNode {
    int target_state;
    char input;
    struct TransitionNode* next;
};

struct State {
    int id;
    struct TransitionNode* transitionListHead;
    bool finalState;
};

struct NFA {
    int stateNum;
    char * inputAlphabet;
    struct State* stateList;
};

struct NFA* init_NFA(int n, char* inputAlphabet){
    struct NFA* out = malloc(sizeof(struct NFA));
    if (!out){
        return NULL; //failed allocation
    }
```

**Date:** 07/08/2024

# Experiment 1.4

## AIM

To convert a given NFA to DFA

## ALGORITHM

1. Start
2. Create utility functions for NFA data structure to read and write transitions.
3. Read NFA input as follows:
    1. The first line contains the number of states (n) , number of final states (f) , number of input alphabets(m), and number of transitions(t).
    2. The next line contains f space separated integers denoting the final states.
    3. The next line contains the m input alphabets as a single string.
    4. The next t lines contain transitions as “qi qj c” representing a transition from qi to qj on input alphabet c. Here, the alphabet ‘e’ denotes epsilon.
4. Remove epsilon transitions from the NFA.
5. Initialize state mapping as linked list to map NFA state set to DFA states.
    1. NFA states are represented as bit mask.
    2. Transitions are stored as a vector of size m containing the transition from this state to the next
6. Define a recursive function to create state mapping from NFA starting with a given state set:
    1. Check if the mapping for the given state set exists in the linked list.
    2. If found, terminate function call.
    3. Otherwise, add node n mapping the state set to new DFA state to the linked list.
    4. For each input symbol i:
        1. Compute set of reachable states for given state set using input symbol i.
        2. Recursively process the new state set.
        3. Add transition from current DFA state to the state corresponding to the given state set.
7. Recursively convert NFA starting with state set {q0} represented as bit mask of 1.

```
    out->stateNum = n;
    out->inputAlphabet = inputAlphabet;
    out->stateList = malloc(sizeof(struct State)*n);
    if (!out->stateList){
        free(out);
        return NULL;
    }

    for (int i=0;i<n;++i){
        out->stateList[i].id = i;
        out->stateList[i].transitionListHead = NULL;
        out->stateList[i].finalState = false;
    }

    return out;
}

void addTransitionNFA(struct NFA* n, int s, int t, char c){
    struct TransitionNode** head = &(n->stateList[s].transitionListHead);
    while (*head){
        if ((*head)->input==c && (*head)->target_state==t){
            return ; //avoid duplicates
        }
        head = &((*head)->next);
    }
    *head = malloc(sizeof(struct TransitionNode));
    if (!*head){
        return; // allocation failed
    }
    (*head)->target_state = t;
```

8. Create DFA data structure with number of states equalling size of the state mapping linked list.
    1. Copy input alphabets from the NFA.
    2. For each state in state mapping:
        1. Mark DFA state as final if any NFA state in the set is final.
        2. Populate DFA transition using transitions in the state mapping.
9. Priint the DFA.
10. Stop

```
    (*head)->input = c;
    (*head)->next = NULL;
}

void freeStateNFA(struct State s){
    struct TransitionNode* head = s.transitionListHead;
    while (head){
        struct TransitionNode* next = head->next;
        free(head);
        head = next;
    }
}

void freeNFA(struct NFA* n){
    if (!n) return;
    for (int i=0;i<(n->stateNum);++i){
        freeStateNFA(n->stateList[i]);
    }

    free(n->inputAlphabet);
    free(n->stateList);
    free(n);
}

void printNFA(struct NFA* nfa){
    printf("The transition table is as follows:\n");
    int n = nfa->stateNum;
    int m = strlen(nfa->inputAlphabet);
    printf("\t");
    for (int i=0;i<m;++i){
        printf("%c\t",nfa->inputAlphabet[i]);
```

```
    }
    printf("epsilon\n");
    for (int i=0;i<n;++i){
        if (i==0){
            printf("->");
        }
        if (nfa->stateList[i].finalState){
            printf("*");
        }
        printf("q%d\t",i);
        struct State s = nfa->stateList[i];
        for (int j=0;j<=m;++j){
            char c = nfa->inputAlphabet[j];
            if (j==m){
                c = 'e';
            }
            for (struct TransitionNode *current=s.transitionListHead;current;current=current->next){
                if (current->input==c){
                    printf("q%d",current->target_state);
                }
            }
            printf("\t");
        }
        printf("\n");
    }
}



struct NFA* readNFA() {
    // read input
```

```
int n, m, t, f;
scanf("%d%d%d%d", &n, &f, &m, &t);
if (f<0 || f>n){
    printf("Invalid number of final states\n");
    return NULL;
}

int finalStates[f];
for (int i=0;i<f;++i){
    scanf("%d",finalStates+i);
    if (finalStates[i]<0 || finalStates[i]>=n){
        printf("Invalid final state %d\n",finalStates[i]);
        return NULL;
    }
}



char* inputChars = malloc(sizeof(char)*(m+1));
if (!inputChars) {
    printf("Failed to allocate memory for input characters\n");
    return NULL;
}

scanf("%s\n", inputChars);
if (strlen(inputChars) != m) {
    free(inputChars);
    printf("Input characters length mismatch\n");
    return NULL;
}

struct NFA *nfa = init_NFA(n,inputChars);
```

```
if (!nfa) {
    free(inputChars);
    printf("Failed to initialize NFA\n");
    return NULL;
}

for (int i=0;i<f;++i){
    nfa->stateList[finalStates[i]].finalState = true;
}

for (int i = 0; i < t; ++i) {
    int a, b;
    char c;
    scanf("q%d q%d %c\n", &a, &b, &c);
    if (a < 0 || a >= n || b < 0 || b >= n) {
        printf("Invalid transition from %d to %d\n", a, b);
        freeNFA(nfa);
        return NULL;
    }
    bool validChar = false;
    for (int j = 0; j < m; ++j) {
        if (inputChars[j] == c) {
            validChar = true;
            break;
        }
    }
    if (!validChar && c != 'e') { // 'e' for epsilon transition
        printf("Invalid input character '%c' for transition from %d to %d\n", c, a, b);
        freeNFA(nfa);
        return NULL;
    }
```

```
        addTransitionNFA(nfa, a, b, c);
    }
    return nfa;
}
```

**dfa_ds.c:**

```
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include "dsu.c"

struct DFA {
    int stateNum;
    bool* finalState;
    char * inputAlphabet;
    int ** transitionTable;
};

void freeDFA(struct DFA* dfa){
    if (!dfa) return;
    if (dfa->finalState){
        free(dfa->finalState);
    }
    if (dfa->transitionTable){
        int n = dfa->stateNum;
        for (int i=0;i<n;++i){
            if (dfa->transitionTable[i]){
                free(dfa->transitionTable[i]);
            }
        }
        free(dfa->transitionTable);
    }
```

```
    free(dfa);
}

struct DFA* init_DFA(int n, char* inputAlphabet){
    struct DFA* out = malloc(sizeof(struct DFA));
    if (!out){
        return NULL; //failed allocation
    }

    out->stateNum = n;
    out->inputAlphabet = inputAlphabet;
    int m = strlen(inputAlphabet);

    out->transitionTable = malloc(sizeof(int*)*n);
    if (!out->transitionTable){
        freeDFA(out);
        return NULL;
    }

    out->finalState = malloc(sizeof(bool)*n);
    if (!out->finalState){
        freeDFA(out);
    }
    for (int i=0;i<n;++i){
        out->finalState[i] = false;
    }

    for (int i=0;i<n;++i){
        out->transitionTable[i] = malloc(sizeof(int)*m);
        if (!out->transitionTable[i]){
            freeDFA(out);
```

```
            return NULL;
        }
        for (int j=0;j<m;++j){
            out->transitionTable[i][j] = i;
        }
    }

    return out;
}

int inputIndexDFA(struct DFA* dfa, char c){
    int m = strlen(dfa->inputAlphabet);
    for (int i=0;i<m;++i){
        if (c==dfa->inputAlphabet[i]){
            return i;
        }
    }
    return -1;
}

void addTransitionDFA(struct DFA* dfa, int s, int t, char c){
    int i = inputIndexDFA(dfa,c);
    if (i!=-1){
        dfa->transitionTable[s][i] = t;
    }
}



void printDFA(struct DFA* dfa){
    printf("The transition table is as follows:\n");
    int n = dfa->stateNum;
```

```
    int m = strlen(dfa->inputAlphabet);
    printf("\t");
    for (int i=0;i<m;++i){
        printf("%c\t",dfa->inputAlphabet[i]);
    }
    printf("\n");
    for (int i=0;i<n;++i){
        if (i==0){
            printf("->");
        }
        if (dfa->finalState[i]){
            printf("*");
        }
        printf("q%d\t",i);
        for (int j=0;j<m;++j){
            printf("q%d\t",dfa->transitionTable[i][j]);
        }
        printf("\n");
    }
}

struct DFA* readDFA() {
    // read input
    int n, f, m;
    scanf("%d%d%d", &n, &f, &m);
    if (f<0 || f>n){
        printf("Invalid number of final states\n");
        return NULL;
    }

    int finalStates[f];
```

```
for (int i=0;i<f;++i){
    scanf("%d",finalStates+i);
    if (finalStates[i]<0 || finalStates[i]>=n){
        printf("Invalid final state %d\n",finalStates[i]);
        return NULL;
    }
}


char* inputChars = malloc(sizeof(char)*(m+1));
if (!inputChars) {
    printf("Failed to allocate memory for input characters\n");
    return NULL;
}

scanf("%s\n", inputChars);
if (strlen(inputChars) != m) {
    free(inputChars);
    printf("Input characters length mismatch\n");
    return NULL;
}

struct DFA *dfa = init_DFA(n,inputChars);
if (!dfa) {
    free(inputChars);
    printf("Failed to initialize DFA\n");
    return NULL;
}

for (int i=0;i<f;++i){
    dfa->finalState[finalStates[i]] = true;
```

```
    }

    for (int i=0;i<n;++i){
        for (int j=0;j<m;++j) {
            scanf("%d",dfa->transitionTable[i]+j);
        }
    }

    return dfa;
}
```

**dfa_conversion.c:**

```
#include <stdio.h>
#include "enfa_functions.c"
#include "dfa_ds.c"

struct StateMappingNode {
    int nfa_value, dfa_value;
    int* transitions;
    struct StateMappingNode* next;
};

void freeStateMappingList(struct StateMappingNode* head){
    if (!head) return;
    freeStateMappingList(head->next);
    free(head->transitions);
    free(head);
}

void printStateMapping(struct StateMappingNode* head){
    for (struct StateMappingNode* current = head;current;current = current->next){
        printf("%d: ", current->dfa_value);
```

```
        for (int i=0,bm=current->nfa_value;bm>0;bm>>=1,++i){
            if (bm&1){
                printf("q%d",i);
            }
        }
        printf("\n");
    }
}

int stateCount = 0;

int transition(struct NFA* nfa, int state, int input){
    int out = 0;
    int copy = state;
    for (int bitCounter=0;state>0;++bitCounter,state >>= 1){
        if ((state&1)==0) continue;
        for (struct TransitionNode* current = nfa-
>stateList[bitCounter].transitionListHead;current;current = current->next){
            if (current->input!=nfa->inputAlphabet[input]) continue;
            out = out | (1<<(current->target_state));
        }
    }
    return out;
}

void recursiveConvert(struct NFA* nfa,struct StateMappingNode** head,int state){
    int m = strlen(nfa->inputAlphabet);

    struct StateMappingNode** indirect = head;

    while (*indirect){
```

```
        if ((*indirect)->nfa_value==state) return;
        indirect = &((*indirect)->next);
    }
    *indirect = malloc(sizeof(struct StateMappingNode));
    (*indirect)->dfa_value = stateCount++;
    (*indirect)->nfa_value = state;
    (*indirect)->transitions = malloc(sizeof(int)*m);
    (*indirect)->next = NULL;

    for (int i=0;i<m;++i){
        int t = transition(nfa,state,i);
        recursiveConvert(nfa,head,t);
        (*indirect)->transitions[i] = t;
    }
}

int stateMapping(struct StateMappingNode* head,int nfa_state){
    while(head){
        if (head->nfa_value==nfa_state){
            return head->dfa_value;
        }
        head = head->next;
    }
    return -1;
}

struct DFA* dfa_conversion(struct NFA* enfa){
    struct NFA* nfa = epsilon_removal(enfa);

    int n = nfa->stateNum;
    int m = strlen(nfa->inputAlphabet);
```

```
if (n>=32){
    printf("NFA with 32 or more states cannot be converted\n");
}

struct StateMappingNode* head = NULL;

recursiveConvert(nfa,&head,1);

char* inputAlphabet = malloc(strlen(nfa->inputAlphabet)*sizeof(char));
strcpy(inputAlphabet,nfa->inputAlphabet);
struct DFA* dfa = init_DFA(stateCount,inputAlphabet);
printf("Mapping:\n");
if (dfa){
    struct StateMappingNode* current = head;
    while (current){
        int s = current->dfa_value;
        for (int i=0;i<n;++i){
            if (nfa->stateList[i].finalState && ((current->nfa_value) & (1 << i))) {
                dfa->finalState[i] = true;
            }
        }
        for (int i=0;i<m;++i){
            int t = stateMapping(head,current->transitions[i]);
            dfa->transitionTable[s][i] = t;
        }
        current = current->next;
    }
}

freeStateMappingList(head);
freeNFA(nfa);
```

```
    return dfa;
}

int main(){
    struct NFA* nfa = readNFA();
    if (!nfa) {
        printf("NFA creation failed\n");
    }
    printf("For the NFA:\n");
    printNFA(nfa);

    struct DFA* dfa = dfa_conversion(nfa);
    if (!dfa){
        printf("DFA conversion failed\n");
    }

    printf("\n\nFor the DFA:\n");
    printDFA(dfa);

    freeDFA(dfa);
    freeNFA(nfa);
}
```

## OUTPUT:

**input.txt:**

```
5 1 2 7
2
01
q0 q1 1
q1 q0 1
q0 q2 e
```

q2 q3 0

q3 q2 0

q2 q4 1

q4 q2 0

**output:**

For the NFA:

The transition table is as follows:

| | 0 | 1 | epsilon |
|---|---|---|---|
| ->q0 | | q1 | q2 |
| q1 | | q0 | |
| *q2 | q3 | q4 | |
| q3 | q2 | | |
| q4 | q2 | | |

Mapping:

For the DFA:

The transition table is as follows:

| | 0 | 1 |
|---|---|---|
| ->*q0 | q1 | q5 |
| q1 | q2 | q4 |
| *q2 | q1 | q3 |
| q3 | q2 | q4 |
| q4 | q4 | q4 |
| q5 | q2 | q6 |
| q6 | q1 | q5 |

## RESULT

Successfully converted the given NFA to DFA.

**Name:** Pradyumn R Pai
**Roll No:** 50
**Class:** CS7A

# PROGRAM CODE

**dfa_ds.c:**

```
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include "dsu.c"

struct DFA {
    int stateNum;
    bool* finalState;
    char * inputAlphabet;
    int ** transitionTable;
};

void freeDFA(struct DFA* dfa){
    if (!dfa) return;
    if (dfa->finalState){
        free(dfa->finalState);
    }
    if (dfa->transitionTable){
        int n = dfa->stateNum;
        for (int i=0;i<n;++i){
            if (dfa->transitionTable[i]){
                free(dfa->transitionTable[i]);
            }
        }
        free(dfa->transitionTable);
    }
    free(dfa);
```

**Date:** 07/08/2024

# Experiment 1.5

## AIM

To minimize a given DFA

## ALGORITHM

1. Start
2. Create a DFA Data Structure that represents the input characters as numbers between 0 to m-1 and the states as 0 to n-1 where n and m are the number of states and input characters respectively.
3. Read the DFA as follows:
    1. The first line contains number of states n, number of final states f, and number of input characters m.
    2. The next line contains f space separated numbers representing the final states.
    3. Next line contains a single string denoting the input characters.
    4. Next n lines contain m space separated integers representing the n*m transition table of the DFA.
4. Create a 2D boolean n*n grid to mark distinguishable state pairs.
5. For each pair of states (i,j) set grid[i][j] = true if and only if exactly one of the two states is a final state.
6. Mark all distinguishable pairs by repeating the following till no changes are made:
    1. For each pair (i,j) where i>j and grid[i][j] is false (i.e they haven't been marked as distinguishable), do the following:
        1. For each input symbol c:
            1. Let x, y be the transition state for i and j respectively for the given character c.
            2. If grid[x][y] is true, set grid[i][j] = grid[j][i] =true and and mark that a change has been made.
7. Initialize a Disjoint Set Union structure d for all states.
8. For each pair (i,j) where i>j and grid[i][j] is false, merge the states i and j in d
9. Based on the DSU, create a DFA where each state represents a set in the DFA.

```
}

struct DFA* init_DFA(int n, char* inputAlphabet){
    struct DFA* out = malloc(sizeof(struct DFA));
    if (!out){
        return NULL; //failed allocation
    }

    out->stateNum = n;
    out->inputAlphabet = inputAlphabet;
    int m = strlen(inputAlphabet);

    out->transitionTable = malloc(sizeof(int*)*n);
    if (!out->transitionTable){
        freeDFA(out);
        return NULL;
    }

    out->finalState = malloc(sizeof(bool)*n);
    if (!out->finalState){
        freeDFA(out);
    }
    for (int i=0;i<n;++i){
        out->finalState[i] = false;
    }

    for (int i=0;i<n;++i){
        out->transitionTable[i] = malloc(sizeof(int)*m);
        if (!out->transitionTable[i]){
            freeDFA(out);
            return NULL;
```

10. For each transition from x to y with character c in the original DFA, add transition from the state corresponding to the sets containing x and y in the new DFA
11. Display the new DFA
12. Stop

```
        }
        for (int j=0;j<m;++j){
            out->transitionTable[i][j] = i;
        }
    }

    return out;
}

int inputIndexDFA(struct DFA* dfa, char c){
    int m = strlen(dfa->inputAlphabet);
    for (int i=0;i<m;++i){
        if (c==dfa->inputAlphabet[i]){
            return i;
        }
    }
    return -1;
}

void addTransitionDFA(struct DFA* dfa, int s, int t, char c){
    int i = inputIndexDFA(dfa,c);
    if (i!=-1){
        dfa->transitionTable[s][i] = t;
    }
}



void printDFA(struct DFA* dfa){
    printf("The transition table is as follows:\n");
    int n = dfa->stateNum;
    int m = strlen(dfa->inputAlphabet);
```

```
    printf("\t");
    for (int i=0;i<m;++i){
        printf("%c\t",dfa->inputAlphabet[i]);
    }
    printf("\n");
    for (int i=0;i<n;++i){
        if (i==0){
            printf("->");
        }
        if (dfa->finalState[i]){
            printf("*");
        }
        printf("q%d\t",i);
        for (int j=0;j<m;++j){
            printf("q%d\t",dfa->transitionTable[i][j]);
        }
        printf("\n");
    }
}

struct DFA* readDFA() {
    // read input
    int n, f, m;
    scanf("%d%d%d", &n, &f, &m);
    if (f<0 || f>n){
        printf("Invalid number of final states\n");
        return NULL;
    }

    int finalStates[f];
    for (int i=0;i<f;++i){
```

```
    scanf("%d",finalStates+i);
    if (finalStates[i]<0 || finalStates[i]>=n){
        printf("Invalid final state %d\n",finalStates[i]);
        return NULL;
    }
}


char* inputChars = malloc(sizeof(char)*(m+1));
if (!inputChars) {
    printf("Failed to allocate memory for input characters\n");
    return NULL;
}

scanf("%s\n", inputChars);
if (strlen(inputChars) != m) {
    free(inputChars);
    printf("Input characters length mismatch\n");
    return NULL;
}

struct DFA *dfa = init_DFA(n,inputChars);
if (!dfa) {
    free(inputChars);
    printf("Failed to initialize DFA\n");
    return NULL;
}

for (int i=0;i<f;++i){
    dfa->finalState[finalStates[i]] = true;
}
```

```
    for (int i=0;i<n;++i){
        for (int j=0;j<m;++j) {
            scanf("%d",dfa->transitionTable[i]+j);
        }
    }

    return dfa;
}
```

**dfa_minimization.c:**

```
#include <stdio.h>
#include "dfa_ds.c"

int main(){
    struct DFA* dfa = readDFA();
    if (!dfa){
        printf("DFA initialization failed\n");
        return 1;
    }
    printDFA(dfa);

    struct DFA* minimizeddfa = dfsMinimization(dfa);
    if (!minimizeddfa){
        printf("DFA minimization failed\n");
        return 1;
    }
    printf("\n\nThe minimized dfa is:\n");
    printDFA(minimizeddfa);

    freeDFA(dfa);
    freeDFA(minimizeddfa);
```

```
}
```

## OUTPUT:

**input.txt:**

6 3 2

1 2 4

01

3 1

2 5

2 5

0 4

2 5

5 5

**output:**

The transition table is as follows:

| | 0 | 1 |
|---|---|---|
| ->q0 | q3 | q1 |
| *q1 | q2 | q5 |
| *q2 | q2 | q5 |
| q3 | q0 | q4 |
| *q4 | q2 | q5 |
| q5 | q5 | q5 |

The minimized dfa is:

The transition table is as follows:

| | 0 | 1 |
|---|---|---|
| ->q0 | q0 | q1 |
| *q1 | q1 | q5 |
| q2 | q5 | q5 |

## RESULT

Successfully minimized given DFA.

**Date:** 07/08/2024

# Experiment 2.1

## AIM

To understand LEX and YACC tools.

## THEORY

LEX is a computer program that generates lexical analysers ("scanner" or "lexers"). It is commonly used with the YACC parser generator and is the standard lexical analyser generator on many Unix and Unix-like systems. LEX reads an input stream specifying the lexical analyser and writes source code which implements the lexical analyser in the C programming language.

**Structure of Lex Programs:**

A LEX program is organized into three main sections: Declarations, Rules, and Auxiliary Functions. Each section serves a distinct purpose in defining the behaviour of the lexical analyser.

- **Declarations**

  The Declarations section comprises two parts:

  - Regular definitions: Define macros and shorthand notations for regular expressions to simplify rule definitions.
  - Auxiliary Declarations: Contains C code such as header file inclusions, function prototypes, and global variable declarations. This code is enclosed within %{ and %} and is copied verbatim into the generated lex.yy.c file.

- **Rules**

  The Rules section consists of pattern-action pairs, where each pair defines a regular expression pattern to match and the corresponding C code to execute upon a successful match.

- **Auxiliary Functions**

  The Auxiliary Functions section includes additional C code that is not part of the rules. This typically contains the main function and any helper functions required by the lexer. This code is placed after the second %% and is directly copied into the lex.yy.c file.

**yylex()**

The 'yylex()' function, defined by Lex in the 'lex.yy.c' file, reads the input stream, matches it against regular expressions, and executes the corresponding actions for each match. It also

generates tokens for further processing by parsers or other components, though the programmer must explicitly invoke 'yylex()' within the auxiliary functions of the LEX program.

**yywrap()**

The function yywrap() is called by yylex() when the end of an input file is reached. If yywrap() returns 0, scanning continues; if it returns a non-zero value, yylex() terminates and returns 0.

**Compilation Steps:**

Step-1: Start

Step-2: Create a file with a .l and write your LEX.

Step-3: Give the command ‘lex simple_calc.l’. This command generates a C source file named lex.yy.c.

Step-4: ‘gcc -o simple_calc lex.yy.c -lfl’ to compile the generated C code.

Step-5: Running the Executable Execute the compiled program using: ‘./simple_calc’

YACC (Yet Another Compiler Compiler) generates LALR parsers from formal grammar specifications. It is often used with Lex for building compilers and interpreters, handling the syntactic analysis phase. YACC reads a grammar file and produces a C source parser that processes token sequences (typically from LEX) to check if they follow the grammar. The modern counterpart of YACC is Bison.

Step 1: Create a file with a .y and write your YACC.

Step 2: Give the command ‘yacc -d simple_calc.y’. This command generates a C source file named y.tab.c.

Step 3: Use ‘gcc -o simple_calc lex.yy.c -lfl’ to compile the generated C code.

Step 4: Running the Executable Execute the compiled program using: ‘./simple_calc’

**Structure of YACC Program**

%{

C Declarations

%}

Yacc Declarations

%%

Grammar Rules

%%

Additional Code (Comments enclosed in /*....*/ may appear in any section)

## RESULT

Familiarized ourselves with the working of YACC and LEX tools for compiler design.

**Name:** Pradyumn R Pai
**Roll No:** 50
**Class:** CS7A

# PROGRAM CODE

**name.lex:**

```
name .*[pP][rR][aA][dD].*
%%
{name} printf("Invalid\n");
exit exit(0);
.* printf("Valid\n");
%%
int main() {
   yylex();
}

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

# OUTPUT:

```
qwerty
Valid

Pradyumn
Invalid

asdfkjlfbpradkaslfsd
Invalid

lfsadnh_2132/.
Valid
```

**Date:** 07/08/2024

# Experiment 2.2

## AIM

To write a lex program to recognize all strings which do not contain the first four characters of our name as a substring.

## ALGORITHM

1. Start
2. Create a lex file with following lexical rules:
    1. Define regex name as .*[pP][rR][aA][dD].* to recognize strings containing the first four letters of my name as a substring
    2. If input matches the above substring, print “Invalid”
    3. Otherwise, if input matches the string”exit”, terminate the program
    4. Otherwise, print “Valid”
    5. Define main function in suer code section to call yylex()
3. Use lex command to generate C program
4. Run the  C program
5. Stop

## RESULT

Successfully compiled and ran the lex program.

**Name:** Pradyumn R Pai
**Roll No:** 50
**Class:** CS7A

# PROGRAM CODE

**program_2.lex:**

```
%{
#include <stdlib.h>
#include "y.tab.h"
#include <string.h>
%}
%option noyywrap
%%
[a-zA-Z][a-zA-Z0-9]* { yylval.str = strdup(yytext); return IDENTIFIER; }
\n { return '\n'; }
.* { yylval.str = strdup(yytext); return INVALID; }
%%
```

**program_2.y:**

```
%{
#include <stdio.h>
#define YYSTYPE char*
%}
%union {
   char* str;
}
%token IDENTIFIER
%token INVALID
%%
program: line
   | line program;

line: IDENTIFIER "\n" {printf("Valid\n");}
   | INVALID "\n" {printf("Invalid\n");}
```

**Date:** 07/08/2024

# Experiment 2.3

## AIM

To write a YACC program to identify valid variable names

## ALGORITHM

1. Start
2. Create a lex file with following lexical rules:
    1. Include y.tab.h generated by YACC program.
    2. If input starts with a letter followed by a combination of letters and digits:
        1. Copy value to yylval.
        2. Return token IDENTIFIER.
    3. If input is a newline character, return newline character.
    4. For any other string:
        1. Copy string to yylval.
        2. Return token INVALID.
3. Create YACC to parse input as follows:
    1. The input consists of multiple lines.
    2. Each line can be either:
    3. An identifier followed by a newline character. In this case, display “Valid”.
    4. An invalid token followed by a newline character. In this case, display “Invalid”.
    5. In user code section:
    6. Define error handling.
    7. Define main function to call yyparse().
4. Use lex command to generate C program.
5. Use yacc to create y.tab,h and y.tab.c
6. Compile and run y.tab.c along with lex program
7. Stop

```
%%
void yyerror(char *s){
   printf("Error: %s\n",s);
}
int main() {
   yyparse();
   return 0;
}
int yywrap() {
   return 1;
}
```

## OUTPUT:

```
abcd123
Valid
456gef
Invalid
a_b_1
Invalid
a123@
Invalid
pqr987
Valid
```

## RESULT

Successfully compiled and ran the YACC and LEX program.

**Name:** Pradyumn R Pai
**Roll No:** 50
**Class:** CS7A

# PROGRAM CODE

**calc.lex:**

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "y.tab.h"
#include <string.h>
extern int yylval;
%}
%option noyywrap
%%
[0-9]+ { yylval = atoi(yytext); return IDENTIFIER; }
[+\-*/()\n] { return yytext[0]; }
. { printf("Invalid character: %s\n", yytext); return INVALID; }
%%
```

**calc.y:**

```
%{
#include <stdio.h>
#define YYSTYPE int
extern YYSTYPE yylval;
%}
%token IDENTIFIER
%token INVALID
%%
program: line
    | program line;

line: exp "\n" {printf("Result: %d\n",$1);}
```

**Date:** 07/08/2024

# Experiment 2.4

## AIM

To implement a calculator using LEX and YACC

## ALGORITHM

1. Start
2. Create a lex file with following lexical rules:
    1. Include y.tab.h generated by YACC program.
    2. If input is a sequence of digits:
        1. Copy value to yylval.
        2. Return token IDENTIFIER.
    3. If input is an arithmetic operator or a newline character, return the first character in the input.
3. Create YACC to parse input as follows:
    1. The input consists of multiple lines.
    2. Each line contains an expression followed by a newline character.
    3. Each expression can be expanded as one of the following:
        1. expression + term

            In this case, the value of resultant expression is the sum of values of the first expression and the term.

        2. expression – term

            In this case, the value of resultant expression is the difference of values of the first expression and the term.

        3. term

            In this case, the value of the expression is that of the term.
    4. Each term can be expanded as one of the following:
        1. term * factor

            In this case, the value of resultant term is the product of values of the first term and the factor.

```
exp: exp "+" term {$$ = $1+$3;}
   | exp "-" term {$$ = $1-$3;}
   | term {$$ = $1;};

term: term "*" factor {$$ = $1*$3;};
   | term "/" factor {$$ = $1/$3;};
   | factor {$$ = $1;};

factor: "(" exp ")" {$$ = $2;}
   | IDENTIFIER {$$ = $1;};
%%
void yyerror(char *s){
   printf("Error: %s\n",s);
}
int main() {
   yyparse();
   return 0;
}
int yywrap() {
   return 1;
}
```

## OUTPUT:

15-3*4

Result: 3

20/4+7

Result: 12

1-1-1

Result: -1

10+20/(5-3)

Result: 20

2. term / factor

   In this case, the value of resultant term is the difference of values of the first term and the factor.

3. factor

   In this case, the value of the term is that of the factor.

5. A factor can be expanded as follows:
   1. A factor can be an IDENTIFIER token. In this case, the value of the factor is the numerical value of the identifier.
   2. ( expression )

      In this case, the value of the factor is that of the expression within the brackets.
6. In user code section:
   1. Define error handling.
   2. Define main function to call yyparse().

4. Use lex command to generate C program.
5. Use yacc to create y.tab,h and y.tab.c
6. Compile and run y.tab.c along with lex program
7. Stop

## RESULT

Successfully implemented a calculator with YACC and LEX.

**Name:** Pradyumn R Pai
**Roll No:** 50
**Class:** CS7A

# PROGRAM CODE

**ast.lex:**

```
%{
#include "y.tab.h"
#include <string.h>
%}
%option noyywrap
%%
[0-9]+ { yylval.intval = atoi(yytext); return INTEGER; }
[a-zA-Z_][a-zA-Z0-9_]* {yylval.strval = strdup(yytext); return IDENTIFIER;}
[+\-*/();=] { return yytext[0]; }
. { /*Skip remaining characters */ }
%%
```

**ast.y:**

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

%type <nodeType> program
%type <nodeType> stmt
%type <nodeType> exp
%type <nodeType> term
%type <nodeType> factor
%token <strval> IDENTIFIER
%token <intval> INTEGER

%{
```

**Date:** 14/08/2024

# Experiment 2.5

## AIM

To generate abstract syntax tree of an expression

## ALGORITHM

1. Start
2. Create a lex file with following lexical rules:
    1. Include y.tab.h generated by YACC program.
    2. If input is a sequence of digits:
        1. Copy value to yylval
        2. Return token INTEGER
    3. If input is a sequence of alphabets, digits, or '_' character sucha that it doesn't start with a digit:
        1. Copy value to yylval
        2. Return token IDENTIFIER
    4. If input is an arithmetic operator, parenthesis, semicolon, '=' symbol, or a newline character, return the first character in the input.
    5. For any other symbol, do nothing.
3. Create YACC to parse input as follows:
    1. Create structure AST to represent node of abstract syntax tree with following attributes:
        1. Union val storing value of the node. This can be an integer value, the name of the identifier, or a binary operator.
        2. Variable kind to identify the type of value stored.
        3. Left and right subtree pointers.
    2. Declare following functions:
        1. make_int takes a numerical value as input and returns an AST leaf node with that value.
        2. make_id takes string identifier as input and returns an AST leaf node with that value.

```
enum ASTKind {
   AST_INT,
   AST_VAR,
   AST_BINOP,
   AST_ASSIGN
};
struct AST {
   union ASTType{
      int intval;
      char *idname;
      char binop;
   } val;
   enum ASTKind kind;
   struct AST* left;
   struct AST* right;
};
%}

%union {
   int intval;
   char *strval;
   struct AST* nodeType;
}

%{
void addASTList(struct AST* n);
struct AST *make_int(int val);
struct AST *make_id(const char *s);
struct AST *make_binop(char op, struct AST *l, struct AST *r);
struct AST *make_assign(const char *name, struct AST *r);
%}
```

3. make_binop takes an operator and two AST nodes as input and returns a tree with a node representing the operator as root node with the input nodes as children.
4. make_assign takes a string value and an AST node as input and makes node assigning the string as value and the AST node as rightr subtree.
5. print_ast recursively prints a given tree with appropriate indentation.

3. Define parser grammar as follows:
    1. The input consists of multiple statements. Each statement returns an AST which should be printed and deleted after parsing.
    2. Each statement contains one of the following:
        1. IDENTIFIER = expression ;

            In this case, value of the statement can be obtained as make_assign(identifier,expression).
        2. expression ;

            In this case, the value of statement is that of the expression.
    3. Each expression can be expanded as one of the following:
        1. expression + term

            In this case, the value of resultant expression can be obtained as make_binop(‘+’,expression, term).
        2. expression – term

            In this case, the value of resultant expression can be obtained as make_binop(‘-’,expression, term).
        3. term

            In this case, the value of the expression is that of the term.
    4. Each term can be expanded as one of the following:
        1. term * factor

            In this case, the value of resultant term can be obtained as make_binop(‘*’,term, factor).
        2. term / factor

            In this case, the value of resultant term can be obtained as make_binop(‘/’,term, factor).
        3. factor

```
%%
program: /* empty*/ { $$ = NULL; }
   | program stmt; { $$ = NULL; print_ast($2,0);free_ast($2);}

stmt: exp ";" {$$ = $1;}
   | IDENTIFIER "=" exp ";" { $$ = make_assign($1, $3); };

exp: exp "+" term {$$ = make_binop('+',$1,$3);};
   | exp "-" term {$$ = make_binop('-',$1,$3);};
   | term {$$ = $1;};

term: term "*" factor {$$ = make_binop('*',$1,$3);};
   | term "/" factor {$$ = make_binop('/',$1,$3);};
   | factor {$$ = $1;};

factor: "(" exp ")" {$$ = $2;}
   | INTEGER {$$ = make_int($1);}
   | IDENTIFIER { $$ = make_id($1); };
%%
struct AST *make_int(int val){
   struct AST* n = malloc(sizeof(struct AST));
   n->left = NULL;
   n->right = NULL;
   n->val.intval = val;
   n->kind = AST_INT;
   return n;
}
struct AST *make_id(const char *s){
   struct AST* n = malloc(sizeof(struct AST));
   n->left = NULL;
```

4. In this case, the value of the term is that of the factor.

5. A factor can be expanded as follows:

    1. A factor can be an IDENTIFIER token. In this case, the value of the factor is obtained by passing the value of the identifier to make_id function.

    2. A factor can be an INTEGER token. In this case, the value of the factor is obtained by passing the value of the identifier to make_int function.

    3. ( expression )

        In this case, the value of the factor is that of the expression within the brackets.

4. In user code section:

    1. Define error handling.

    2. Define main function to call yyparse().

4. Use lex command to generate C program.

5. Use yacc to create y.tab,h and y.tab.c

6. Compile and run y.tab.c along with lex program

7. Stop

```
    n->right = NULL;
    n->val.idname = strdup(s);
    n->kind = AST_VAR;
    return n;
}
struct AST *make_binop(char op, struct AST *l, struct AST *r){
    struct AST* n = malloc(sizeof(struct AST));
    n->left = l;
    n->right = r;
    n->val.binop = op;
    n->kind = AST_BINOP;
    return n;
}
struct AST *make_assign(const char *name, struct AST *r){
    struct AST* n = malloc(sizeof(struct AST));
    n->left = NULL;
    n->right = r;
    n->val.idname = name;
    n->kind = AST_ASSIGN;
    return n;
}

void print_ast(struct AST *a, int indent){
    if (!a) return;
    for (int i=0;i<indent;++i){
        printf("-");
    }
    switch(a->kind){
        case AST_BINOP:
            printf("Binop(%c)\n",a->val.binop);
            break;
```

```
        case AST_INT:
            printf("Int(%d)\n",a->val.intval);
            break;
        case AST_VAR:
            printf("Id(%s)\n",a->val.idname);
            break;
        case AST_ASSIGN:
            printf("Assign(%s)\n",a->val.idname);
    }
    print_ast(a->left,indent+2);
    print_ast(a->right,indent+2);
}

void free_ast(struct AST* a){
    if (!a) return;
    free_ast(a->left);
    free_ast(a->right);
    free(a);
}

void yyerror(char *s){
    printf("Error: %s\n",s);
}
int main() {
    yyparse();
    return 0;
}
int yywrap() {
    return 1;
}
```

## OUTPUT:

x=5;y=10;z=x+y*3/5;

Assign(x)

--Int(5)

Assign(y)

--Int(10)

Assign(z)

--Binop(+)

----Id(x)

----Binop(/)

------Binop(*)

--------Id(y)

--------Int(3)

------Int(5)

a = 1+(2*3+(6/2)-1)*9;

Assign(a)

--Binop(+)

----Int(1)

----Binop(*)

------Binop(-)

--------Binop(+)

----------Binop(*)

----------Int(2)

----------Int(3)

----------Binop(/)

----------Int(6)

----------Int(2)

--------Int(1)

------Int(9)

## RESULT

Successfully generated abstract syntax tree with YACC and LEX.

**Name:** Pradyumn R Pai
**Roll No:** 50
**Class:** CS7A

# PROGRAM CODE

**for.lex:**

```
%{

#include "y.tab.h"

#include <string.h>

%}

%option noyywrap

%%

[0-9]+ { return INTEGER; }

for { return FOR; }

int|float|char { return TYPE; }

[a-zA-Z_][a-zA-Z0-9_]* {return IDENTIFIER;}

(==) { return RELATIONAL_OPERATOR; }

=|(\+=)|(-=)|(\*=)|(\/=)|(\|\|=)|(&&=)|(<<=)|(>>=)|(\^=)|(%=) { return ASSIGN; }

[+/%*\-] { return ARITHMETIC_OPERATOR; }

(<=)|(>=)|(!=)|[<>] { return RELATIONAL_OPERATOR; }

(\|\|)|(&&) { return LOGICAL_OPERATOR; }

(<<)|(>>)|[&|^] { return BITWISE_OPERATOR;}

\( { return LPAREN; }

\) { return RPAREN; }

\{ { return LCURLY; }

\} { return RCURLY; }

; { return SEMICOLON; }

, { return COMMA; }

! { return NOT; }

. { /*Skip remaining characters */ }

%%
```

**Date:** 14/08/2024

# Experiment 2.6

## AIM

To check syntax of for loop in C

## ALGORITHM

1. Start
2. Create a lex file with following lexical rules:
    1. Include y.tab.h generated by the YACC program.
    2. If input is a sequence of digits:
        1. Return token INTEGER.
    3. If input is the keyword for:
        1. Return token FOR.
    4. If input is a type keyword (int, float, char):
        1. Return token TYPE.
    5. If input is a valid identifier (starts with a letter or _, followed by letters, digits, or _):
        1. Return token IDENTIFIER.
    6. If input matches relational operators (==, <=, >=, !=, <, >):
        1. Return token RELATIONAL_OPERATOR.
    7. If input matches assignment operators (=, +=, -=, *=, /=, etc.):
        1. Return token ASSIGN.
    8. If input matches arithmetic operators (+, -, *, /, %):
        1. Return token ARITHMETIC_OPERATOR.
    9. If input matches logical operators (||, &&):
        1. Return token LOGICAL_OPERATOR.
    10. If input matches bitwise operators (<<, >>, &, |, ^):
        1. Return token BITWISE_OPERATOR.
    11. If input matches parentheses, curly braces, semicolon, comma, or !:

**for.y:**

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

%token IDENTIFIER
%token INTEGER
%token FOR
%token TYPE
%token LOGICAL_OPERATOR
%token RELATIONAL_OPERATOR
%token ARITHMETIC_OPERATOR
%token BITWISE_OPERATOR
%token LPAREN
%token RPAREN
%token LCURLY
%token RCURLY
%token SEMICOLON
%token COMMA
%token NOT
%token ASSIGN

%left LOGICAL_OPERATOR
%left RELATIONAL_OPERATOR
%left ARITHMETIC_OPERATOR
%left BITWISE_OPERATOR
%right NOT
%left LPAREN RPAREN
```

    1. Return the corresponding token (LPAREN, RPAREN, LCURLY, RCURLY, SEMICOLON, COMMA, NOT).
    12. For any other symbol, skip.
3. Create YACC to parse input as follows:
    1. Define tokens for all operators, keywords, and symbols as per the LEX file.
    2. Define grammar rules to recognize a valid C-style for loop:
        1. The input consists of zero or more for loops.
        2. Each for loop must match the pattern:

            for ( initialization ; condition ; update ) statement_block

            On successful recognition of a for loop, print Valid.
        3. Initialization can be a comma seperated series of variable declarations or assignments. Initialization can be NULL as well.
        4. Condition can be any logical or relational expression.
        5. Update can be assignments or arithmetic expressions.
        6. Statement block can be a single statement or a block enclosed in {}.
    3. In the user code section:
        1. Define error handling function.
        2. Define main function to call yyparse().
4. Use lex command to generate C program.
5. Use yacc to create y.tab,h and y.tab.c
6. Compile and run y.tab.c along with lex program using gcc.
7. Stop

```
%%
//Grammar
program: /* empty */
   | program for_loop {printf("Valid\n");}
for_loop: FOR LPAREN init_statements SEMICOLON logic_opt SEMICOLON
update_stmts RPAREN stmt_block;
init_statements:  /* empty */
   | init_stmt_list;
logic_opt: /* empty */
   | logic_exp;
init_stmt_list: init_stmt
   | init_stmt_list COMMA init_stmt;
init_stmt: assignment_stmt
   | TYPE id_def;
update_stmts:  /* empty */
   | update_stmt_list;
update_stmt_list: update_stmt
   | update_stmt_list COMMA update_stmt;
update_stmt: assignment_stmt
   | arithmetic_exp;
stmt_block:LCURLY stmt_list RCURLY
   | stmt;
stmt_list: stmt
   | stmt_list stmt;
stmt: assignment_stmt SEMICOLON
   | for_loop
   | declaration_stmt SEMICOLON
   | arithmetic_exp SEMICOLON;
assignment_stmt: IDENTIFIER assignment_operator arithmetic_exp;
assignment_operator: ASSIGN
```

```
    ;
declaration_stmt : TYPE id_list
id_list : id_list COMMA id_def
    | id_def;
id_def: IDENTIFIER
    | assignment_stmt;
arithmetic_exp: arithmetic_exp ARITHMETIC_OPERATOR arithmetic_exp
    | arithmetic_exp BITWISE_OPERATOR arithmetic_exp
    | LPAREN arithmetic_exp RPAREN
    | IDENTIFIER
    | INTEGER;
logic_exp: arithmetic_exp RELATIONAL_OPERATOR arithmetic_exp
    | logic_exp LOGICAL_OPERATOR logic_exp
    | NOT logic_exp
    | LPAREN logic_exp RPAREN;

%%
void yyerror(char *s){
    printf("Error: %s\n",s);
}
int main() {
    yyparse();
    return 0;
}
int yywrap() {
    return 1;
}
```

## OUTPUT:

```
for (int i=0;i<n;i+=1) {
   a = i*3;
   b = 6;
}
for (a=0,b=1,c=2;c<10;a=a+3,b=b-2,c=c+5) x = a*b*c;
for (int i=0;i<n;i=i+1){
   for (int j=0;j<m;j=j+2){
      x = x + i-j;
   }
}
for (;;) ;
for (;;;)
```

**output:**

Valid

Valid

Valid

Error: syntax error

## RESULT

Successfully verified for loop syntax