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.
        4. If current character is white space:
            1. If buffer is empty, ignore the character and go to next iteration of the loop.

```
/* 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;
    }
    return true;
```

    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

```
}

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

```
    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;
```

9. Print the new NFA.
10. Stop

```
}

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.
8. Create DFA data structure with number of states equalling size of the state mapping linked list.

```
    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;
```

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

```
}

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.
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

```
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){
```

12. Stop

```
            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.