BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Empowered Autonomous Institute Affiliated to Mumbai University)
Department Of Computer Engineering

Name	Manish Shashikant Jadhav
UID	2023301005
Subject	System Programming and Compiler Construction Lab
Experiment No.	1

Aim

Experiment 1: Optimization of DFA

Input: Regular Expression

Output: **for each node** print nullable , firstpos , lastpos , followpos

```
            print state transition table of DFA
            State name |    Symbol 1  | Symbol 2   |
           ____________________________________
          |          |              |           |
          _____________________________________
          |          |              |            |
          ______________________________________
```

Code:

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

// Define initial capacities
#define INITIAL_CAPACITY 10
#define MAX_ALPHABET_SIZE 256

// Struct for dynamic integer set
typedef struct IntSet {
    int *elements;
    int size;
    int capacity;
} IntSet;

// Initialize IntSet
void initIntSet(IntSet *set) {
    set->elements = (int *)malloc(INITIAL_CAPACITY * sizeof(int));
    set->size = 0;
    set->capacity = INITIAL_CAPACITY;
}

// Add element to IntSet if not present
void addToIntSet(IntSet *set, int value) {
    for(int i = 0; i < set->size; i++) {
        if(set->elements[i] == value)
            return;
    }
    if(set->size == set->capacity) {
        set->capacity *= 2;
        set->elements = (int *)realloc(set->elements, set->capacity * sizeof(int));
    }
    set->elements[set->size++] = value;
}

// Check if IntSet contains a value
bool containsIntSet(IntSet *set, int value) {
    for(int i = 0; i < set->size; i++) {
        if(set->elements[i] == value)
            return true;
    }
    return false;
}

// Struct for dynamic list of integers
typedef struct IntList {
    int *items;
    int size;
    int capacity;
} IntList;

// Initialize IntList
void initIntList(IntList *list) {
    list->items = (int *)malloc(INITIAL_CAPACITY * sizeof(int));
    list->size = 0;
    list->capacity = INITIAL_CAPACITY;
}

// Add item to IntList
void addToIntList(IntList *list, int value) {
    if(list->size == list->capacity) {
        list->capacity *= 2;
        list->items = (int *)realloc(list->items, list->capacity * sizeof(int));
    }
    list->items[list->size++] = value;
}

// Struct for Node
typedef struct Node {
    char value;
    struct Node *leftc;
    struct Node *rightc;
    int posNumber;
    IntSet firstpos;
    IntSet lastpos;
    IntSet followpos;
    bool nullable;
} Node;

// Initialize Node
Node* createNode(char value) {
    Node *node = (Node *)malloc(sizeof(Node));
    node->value = value;
    node->leftc = NULL;
    node->rightc = NULL;
    node->posNumber = 0;
    initIntSet(&(node->firstpos));
    initIntSet(&(node->lastpos));
    initIntSet(&(node->followpos));
    node->nullable = false;
    return node;
```

```
}

// Struct for State
typedef struct State {
    IntList value;
    bool marked;
} State;

// Initialize State
State* createState() {
    State *state = (State *)malloc(sizeof(State));
    initIntList(&(state->value));
    state->marked = false;
    return state;
}

// Struct for Transition
typedef struct Transition {
    State *from;
    State *to;
    char value;
} Transition;

// Struct for dynamic list of Nodes
typedef struct NodeList {
    Node **items;
    int size;
    int capacity;
} NodeList;

// Initialize NodeList
void initNodeList(NodeList *list) {
    list->items = (Node **)malloc(INITIAL_CAPACITY * sizeof(Node *));
    list->size = 0;
    list->capacity = INITIAL_CAPACITY;
}
```

```
// Add Node to NodeList
void addToNodeList(NodeList *list, Node *node) {
    if(list->size == list->capacity) {
        list->capacity *= 2;
        list->items = (Node **)realloc(list->items, list->capacity * sizeof(Node *));
    }
    list->items[list->size++] = node;
}

// Struct for dynamic list of States
typedef struct StateList {
    State **items;
    int size;
    int capacity;
} StateList;

// Initialize StateList
void initStateList(StateList *list) {
    list->items = (State **)malloc(INITIAL_CAPACITY * sizeof(State *));
    list->size = 0;
    list->capacity = INITIAL_CAPACITY;
}

// Add State to StateList
void addToStateList(StateList *list, State *state) {
    if(list->size == list->capacity) {
        list->capacity *= 2;
        list->items = (State **)realloc(list->items, list->capacity * sizeof(State *));
    }
    list->items[list->size++] = state;
}

// Struct for dynamic list of Transitions
typedef struct TransitionList {
    Transition **items;
    int size;
    int capacity;
```

```
} TransitionList;

// Initialize TransitionList
void initTransitionList(TransitionList *list) {
    list->items = (Transition **)malloc(INITIAL_CAPACITY * sizeof(Transition *));
    list->size = 0;
    list->capacity = INITIAL_CAPACITY;
}

// Add Transition to TransitionList
void addToTransitionList(TransitionList *list, Transition *trans) {
    if(list->size == list->capacity) {
        list->capacity *= 2;
        list->items = (Transition **)realloc(list->items, list->capacity * sizeof(Transition *));
    }
    list->items[list->size++] = trans;
}

// Struct for Queue of State pointers
typedef struct QueueNode {
    State *state;
    struct QueueNode *next;
} QueueNode;

typedef struct Queue {
    QueueNode *front;
    QueueNode *rear;
} Queue;

// Initialize Queue
void initQueue(Queue *q) {
    q->front = q->rear = NULL;
}

// Enqueue State
void enqueue(Queue *q, State *state) {
    QueueNode *temp = (QueueNode *)malloc(sizeof(QueueNode));
    temp->state = state;
    temp->next = NULL;
    if(q->rear == NULL) {
        q->front = q->rear = temp;
        return;
    }
    q->rear->next = temp;
    q->rear = temp;
}

// Dequeue State
State* dequeue(Queue *q) {
    if(q->front == NULL)
        return NULL;
    QueueNode *temp = q->front;
    State *state = temp->state;
    q->front = q->front->next;
    if(q->front == NULL)
        q->rear = NULL;
    free(temp);
    return state;
}

// Check if Queue is empty
bool isQueueEmpty(Queue *q) {
    return q->front == NULL;
}

// Struct for Tree
typedef struct Tree {
    Node *root;
    int count;
    bool alphabet[MAX_ALPHABET_SIZE];
    NodeList leaves;
    StateList Dstates;
    TransitionList Dtrans;
} Tree;
```

```
// Initialize NodeList, StateList, TransitionList in Tree
void initTree(Tree *tree) {
    tree->root = NULL;
    tree->count = 0;
    for(int i = 0; i < MAX_ALPHABET_SIZE; i++)
        tree->alphabet[i] = false;
    initNodeList(&(tree->leaves));
    initStateList(&(tree->Dstates));
    initTransitionList(&(tree->Dtrans));
}

// Function to check if character is a letter
bool isLetter(char c) {
    return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');
}

// Function to parse regex
void parseRegex(Tree *tree, char *regex) {
    // Implement a simple stack using dynamic array
    int stackCapacity = INITIAL_CAPACITY;
    char *stack = (char *)malloc(stackCapacity * sizeof(char));
    int top = -1;

    int i = 0;
    while(i < strlen(regex)) {
        char current = regex[i];
        if(current == '(') {
            i++;
            while(regex[i] != ')' && i < strlen(regex)) {
                if(top == stackCapacity -1) {
                    stackCapacity *=2;
                    stack = (char *)realloc(stack, stackCapacity * sizeof(char));
                }
                stack[++top] = regex[i];
                if(isLetter(regex[i])) {
                    tree->count++;
                    tree->alphabet[(int)regex[i]] = true;
                }
                i++;
            }
            // Pop three characters
            if(top >=2) {
                char c1 = stack[top--];
                char c2 = stack[top--];
                char c3 = stack[top--];
                Node *n1 = createNode(c1);
                Node *n2 = createNode(c2);
                Node *n3 = createNode(c3);
                n2->leftc = n3;
                n2->rightc = n1;
                tree->root = n2;
            }
            i++;
        }
        else if(current == '*') {
            Node *temp = createNode('*');
            temp->leftc = tree->root;
            tree->root = temp;
            i++;
        }
        else if(isLetter(current)) {
            tree->count++;
            tree->alphabet[(int)current] = true;
            if(tree->root != NULL) {
                if(tree->root->value != '.') {
                    Node *temp = createNode('.');
                    temp->leftc = tree->root;
                    temp->rightc = createNode(current);
                    tree->root = temp;
                }
                else {
                    if(tree->root->rightc != NULL) {
                        Node *temp = createNode('.');
```

```
                        temp->leftc = tree->root;
                        temp->rightc = createNode(current);
                        tree->root = temp;
                    }
                    else {
                        tree->root->rightc = createNode(current);
                    }
                }
            }
            else {
                Node *temp = createNode('.');
                temp->leftc = createNode(current);
                tree->root = temp;
            }
            i++;
        }
        else {
            i++;
        }
    }
    // Append '.' and '#' to the tree
    Node *temp = createNode('.');
    temp->rightc = createNode('#');
    temp->leftc = tree->root;
    tree->root = temp;
    tree->count++;
    free(stack);
}

// Function to print Tree
void printTreeHelper(Node *n) {
    if(n == NULL)
        return;
    printf("%-6c | %-12s | %-12s | %-8s | ", n->value,
        n->leftc != NULL ? (char[]){n->leftc->value, '\0'} : "null",
        n->rightc != NULL ? (char[]){n->rightc->value, '\0'} : "null",
        n->nullable ? "true" : "false");
    printf("{");
    for(int i = 0; i < n->firstpos.size; i++) {
        printf("%d", n->firstpos.elements[i]);
        if(i < n->firstpos.size -1) printf(", ");
    }
    printf("} | {");
    for(int i = 0; i < n->lastpos.size; i++) {
        printf("%d", n->lastpos.elements[i]);
        if(i < n->lastpos.size -1) printf(", ");
    }
    printf("} | {");
    for(int i = 0; i < n->followpos.size; i++) {
        printf("%d", n->followpos.elements[i]);
        if(i < n->followpos.size -1) printf(", ");
    }
    printf("}\n");
    printTreeHelper(n->leftc);
    printTreeHelper(n->rightc);
}

void printTree(Tree *tree) {
    printf("%-6s | %-12s | %-12s | %-8s | %-10s | %-9s | %-11s\n",
        "Value", "Left Child", "Right Child", "Nullable", "Firstpos", "Lastpos", "Followpos");
    printTreeHelper(tree->root);
}

// Function to check if Node is leaf
bool isLeaf(Node *n) {
    return n->leftc == NULL && n->rightc == NULL;
}

// Function to add Node to leaves
void addToLeaves(Tree *tree, Node *n) {
    // Insert at beginning
    if(tree->leaves.size == tree->leaves.capacity) {
        tree->leaves.capacity *=2;
        tree->leaves.items = (Node **)realloc(tree->leaves.items, tree->leaves.capacity * sizeof(Node *));
    }
    for(int i = tree->leaves.size; i >0; i--) {
        tree->leaves.items[i] = tree->leaves.items[i-1];
    }
    tree->leaves.items[0] = n;
    tree->leaves.size++;
}

// Function to number leaves
void numberLeaves(Tree *tree, Node *n) {
    if(isLeaf(n)) {
        n->posNumber = tree->count;
        addToIntSet(&(n->firstpos), tree->count);
        addToIntSet(&(n->lastpos), tree->count);
        addToLeaves(tree, n);
        tree->count--;
        return;
    }
    if(n->value == '*') {
        numberLeaves(tree, n->leftc);
    }
    else {
        numberLeaves(tree, n->rightc);
        numberLeaves(tree, n->leftc);
    }
}

// Function to assign nullable
void assignNullable(Tree *tree, Node *n) {
    if(n == NULL)
        return;
    if(n->value == '|') {
        assignNullable(tree, n->leftc);
        assignNullable(tree, n->rightc);
        n->nullable = n->leftc->nullable || n->rightc->nullable;
    }
    else if(n->value == '.') {
        assignNullable(tree, n->leftc);
        assignNullable(tree, n->rightc);
        n->nullable = n->leftc->nullable && n->rightc->nullable;
    }
    else if(n->value == '*') {
        assignNullable(tree, n->leftc);
        n->nullable = true;
    }
    else {
        n->nullable = false;
    }
}

// Function to assign firstpos and lastpos
void assignFirstLastPos(Tree *tree, Node *n) {
    if(n == NULL)
        return;
    if(n->value == '|') {
        assignFirstLastPos(tree, n->leftc);
        assignFirstLastPos(tree, n->rightc);
        for(int i =0; i < n->leftc->firstpos.size; i++)
            addToIntSet(&(n->firstpos), n->leftc->firstpos.elements[i]);
        for(int i =0; i < n->rightc->firstpos.size; i++)
            addToIntSet(&(n->firstpos), n->rightc->firstpos.elements[i]);
        for(int i =0; i < n->leftc->lastpos.size; i++)
            addToIntSet(&(n->lastpos), n->leftc->lastpos.elements[i]);
        for(int i =0; i < n->rightc->lastpos.size; i++)
            addToIntSet(&(n->lastpos), n->rightc->lastpos.elements[i]);
    }
    else if(n->value == '.') {
        assignFirstLastPos(tree, n->leftc);
        assignFirstLastPos(tree, n->rightc);
        if(n->leftc->nullable) {
            for(int i =0; i < n->leftc->firstpos.size; i++)
                addToIntSet(&(n->firstpos), n->leftc->firstpos.elements[i]);
            for(int i =0; i < n->rightc->firstpos.size; i++)
                addToIntSet(&(n->firstpos), n->rightc->firstpos.elements[i]);
        }
        else {
            for(int i =0; i < n->leftc->firstpos.size; i++)
                addToIntSet(&(n->firstpos), n->leftc->firstpos.elements[i]);
        }
        if(n->rightc->nullable) {
            for(int i =0; i < n->leftc->lastpos.size; i++)
                addToIntSet(&(n->lastpos), n->leftc->lastpos.elements[i]);
            for(int i =0; i < n->rightc->lastpos.size; i++)
                addToIntSet(&(n->lastpos), n->rightc->lastpos.elements[i]);
        }
        else {
            for(int i =0; i < n->rightc->lastpos.size; i++)
                addToIntSet(&(n->lastpos), n->rightc->lastpos.elements[i]);
        }
    }
    else if(n->value == '*') {
        assignFirstLastPos(tree, n->leftc);
        for(int i =0; i < n->leftc->firstpos.size; i++)
            addToIntSet(&(n->firstpos), n->leftc->firstpos.elements[i]);
        for(int i =0; i < n->leftc->lastpos.size; i++)
            addToIntSet(&(n->lastpos), n->leftc->lastpos.elements[i]);
    }
    else {
        return;
    }
}

// Function to calculate followpos
void calculateFollowPos(Tree *tree, Node *n) {
    if (n == NULL)
        return;

    if (n->value == '.') {
        // For each position i in lastpos(c1), all positions in firstpos(c2) are in followpos(i)
        for (int i = 0; i < n->leftc->lastpos.size; i++) {
            int pos = n->leftc->lastpos.elements[i];
            for (int j = 0; j < n->rightc->firstpos.size; j++) {
                addToIntSet(&(tree->leaves.items[pos-1]->followpos), n->rightc->firstpos.elements[j]);
            }
        }
    }
    else if (n->value == '*') {
        // For each position i in lastpos(n), all positions in firstpos(n) are in followpos(i)
        for (int i = 0; i < n->lastpos.size; i++) {
            int pos = n->lastpos.elements[i];
            for (int j = 0; j < n->firstpos.size; j++) {
                addToIntSet(&(tree->leaves.items[pos-1]->followpos), n->firstpos.elements[j]);
            }
        }
    }

    // Recursively process left and right children
    calculateFollowPos(tree, n->leftc);
    calculateFollowPos(tree, n->rightc);
}

// Function to assign followpos
void assignFollowPos(Tree *tree, Node *n) {
    calculateFollowPos(tree, n);
}

// Function to check if two IntSets are equal
bool areIntSetsEqual(IntSet *a, IntSet *b) {
    if(a->size != b->size)
        return false;
    for(int i =0; i < a->size; i++) {
        bool found = false;
        for(int j =0; j < b->size; j++) {
            if(a->elements[i] == b->elements[j]) {
                found = true;
                break;
            }
        }
        if(!found)
            return false;
    }
    return true;
}

// Function to get State by value
State* getStateByValue(StateList *states, IntSet *value) {
    for(int i =0; i < states->size; i++) {
        if(areIntSetsEqual(&(states->items[i]->value), value))
            return states->items[i];
    }
    return NULL;
}

// Function to construct Dstates
void constructDstates(Tree *tree) {
    State *s0 = createState();
    for(int i =0; i < tree->root->firstpos.size; i++)
        addToIntList(&(s0->value), tree->root->firstpos.elements[i]);
    addToStateList(&(tree->Dstates), s0);
    Queue queue;
    initQueue(&queue);
    enqueue(&queue, s0);
    // Implement a simple processedStates as list of IntSets
    int processedCapacity = INITIAL_CAPACITY;
    int processedSize =0;
    IntSet *processedStates = (IntSet *)malloc(processedCapacity * sizeof(IntSet));
    initIntSet(&processedStates[processedSize]);
    for(int i =0; i < s0->value.size; i++)
        addToIntSet(&processedStates[processedSize], s0->value.items[i]);
    processedSize++;

    while(!isQueueEmpty(&queue)) {
        State *currentState = dequeue(&queue);
        // Iterate over alphabet
        for(int a =0; a < MAX_ALPHABET_SIZE; a++) {
```

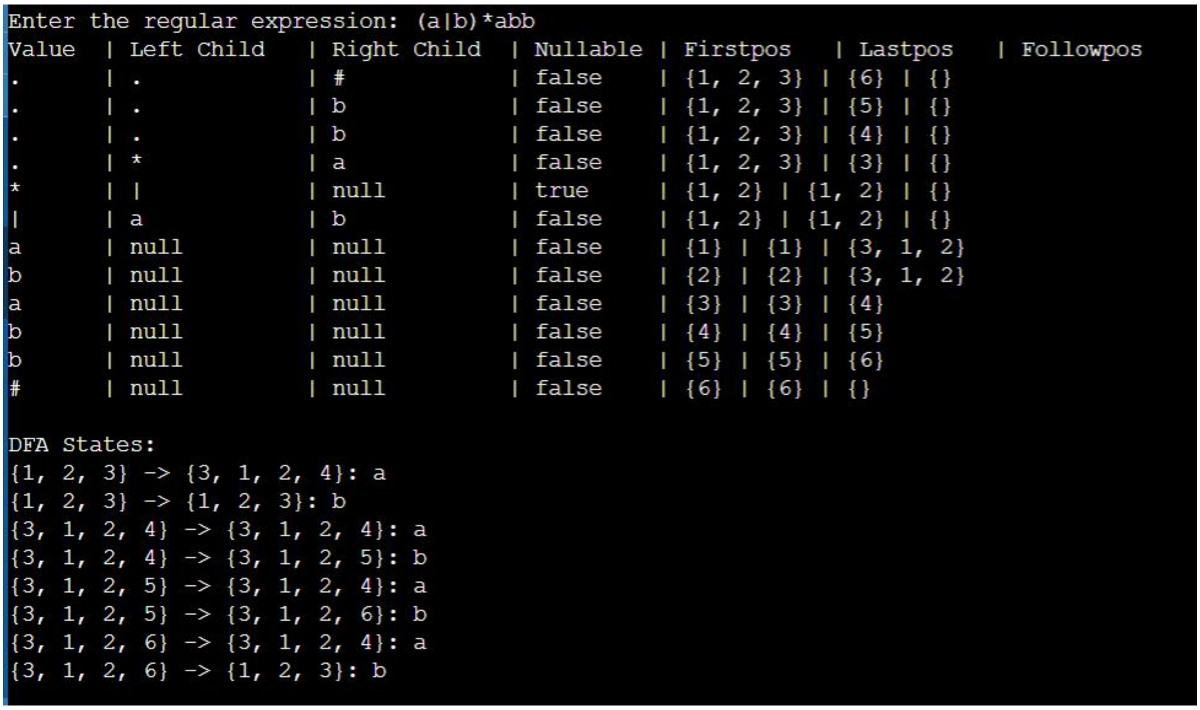
```
if(!tree->alphabet[a])
    continue;
IntSet U;
initIntSet(&U);
for(int p =0; p < currentState->value.size; p++) {
    int pos = currentState->value.items[p];
    Node *node = tree->leaves.items[pos-1];
    if(node->value == (char)a) {
        for(int f =0; f < node->followpos.size; f++)
            addToIntSet(&U, node->followpos.elements[f]);
    }
}
if(U.size ==0)
    continue;
// Check if U is already processed
bool found = false;
State *existingState = NULL;
for(int s =0; s < tree->Dstates.size; s++) {
    if(areIntSetsEqual(&U, &(tree->Dstates.items[s]->value))) {
        found = true;
        existingState = tree->Dstates.items[s];
        break;
    }
}
if(!found) {
    State *newState = createState();
    for(int u =0; u < U.size; u++)
        addToIntList(&(newState->value), U.elements[u]);
    addToStateList(&(tree->Dstates), newState);
    enqueue(&queue, newState);
    existingState = newState;
}
// Create Transition
Transition *trans = (Transition *)malloc(sizeof(Transition));
trans->from = currentState;
trans->to = existingState;
trans->value = (char)a;
```

```
            addToTransitionList(&(tree->Dtrans), trans);
        }
    }
    free(processedStates);
}

// Function to print DFA
void printDFA(Tree *tree) {
    printf("\nDFA States:\n");
    for(int i =0; i < tree->Dtrans.size; i++) {
        Transition *t = tree->Dtrans.items[i];
        printf("{");
        for(int j =0; j < t->from->value.size; j++) {
            printf("%d", t->from->value.items[j]);
            if(j < t->from->value.size -1) printf(", ");
        }
        printf("} -> {");
        for(int j =0; j < t->to->value.size; j++) {
            printf("%d", t->to->value.items[j]);
            if(j < t->to->value.size -1) printf(", ");
        }
        printf("}: %c\n", t->value);
    }
}

// Main function
int main() {
    Tree t;
    initTree(&t);
    char regex[100];
    printf("Enter the regular expression: ");
    fgets(regex, sizeof(regex), stdin);
    // Remove newline character
    regex[strcspn(regex, "\n")] = 0;
    parseRegex(&t, regex);
    numberLeaves(&t, t.root);
    assignNullable(&t, t.root);
    assignFirstLastPos(&t, t.root);
    assignFollowPos(&t, t.root);
    constructDstates(&t);
    printTree(&t);
    printDFA(&t);
    // Free allocated memory (not implemented for brevity)
    return 0;
}
```

Output:



```
Enter the regular expression: (a|b)*abb
Value   | Left Child    | Right Child   | Nullable | Firstpos    | Lastpos    | Followpos
.       | .             | #             | false    | {1, 2, 3} | {6} | {}
.       | .             | b             | false    | {1, 2, 3} | {5} | {}
.       | .             | b             | false    | {1, 2, 3} | {4} | {}
.       | *             | a             | false    | {1, 2, 3} | {3} | {}
*       | |             | null          | true     | {1, 2} | {1, 2} | {}
|       | a             | b             | false    | {1, 2} | {1, 2} | {}
a       | null          | null          | false    | {1} | {1} | {3, 1, 2}
b       | null          | null          | false    | {2} | {2} | {3, 1, 2}
a       | null          | null          | false    | {3} | {3} | {4}
b       | null          | null          | false    | {4} | {4} | {5}
b       | null          | null          | false    | {5} | {5} | {6}
#       | null          | null          | false    | {6} | {6} | {}

DFA States:
{1, 2, 3} -> {3, 1, 2, 4}: a
{1, 2, 3} -> {1, 2, 3}: b
{3, 1, 2, 4} -> {3, 1, 2, 4}: a
{3, 1, 2, 4} -> {3, 1, 2, 5}: b
{3, 1, 2, 5} -> {3, 1, 2, 4}: a
{3, 1, 2, 5} -> {3, 1, 2, 6}: b
{3, 1, 2, 6} -> {3, 1, 2, 4}: a
{3, 1, 2, 6} -> {1, 2, 3}: b
```

Conclusion

Hence by completing this experiment we came to know how to optimize DFA from RE.