**Министерство науки и высшего образования Российской Федерации**
**Федеральное государственное бюджетное образовательное учреждение**
**высшего образования**
**«Московский государственный технический университет**
**имени Н.Э. Баумана**
**(национальный исследовательский университет)»**
**(МГТУ им. Н.Э. Баумана)**

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

**Лабораторная работа № __1__**

**Дисциплина Конструирование компиляторов**

**Тема Распознование цепочек регулярного языка**

**Вариант №02**

**Студент _Кибамба Ж.Ж._____**

**Группа _ИУ7И-21М_____**

**Преподаватель _Ступников А.А.**

**23.03.2024**

Москва, 2024 г.

# Описание задания

Напишите программу, которая в качестве входа принимает произвольное выражение, и выполняет следующие преобразования:

1. По регулярному выраению строит НКА.
2. По НКА строит эквивалентный ему ДКА.
3. По ДКА строит эквивалентный ему КА, имеющий найменьшее возможное количество состояний. Указание. Возпользоваться алгоритмом, приведенным по адресу http://neerc.ifmo.ru/wiki/index.php?title=Минимизация_ДКА,_алгоритм_за_O(n^2)_с_построением_пар_различимых_состояний
4. Моделирует минимальный КА для входной цепочки из терминалов исходной грамматики.

# Текст программы

Листинг 1 – Класс, представляющий НКА

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
public class NFA {
    private final ArrayList<Integer> states = new ArrayList<>();
    private final ArrayList<Transition> transitions = new ArrayList<>();
    private int finalState;
    public int getStatesCount() {
        return this.states.size();
    }
    public void setStates(int totalStates) {
        for (int i = 0; i < totalStates; i++) {
            this.states.add(i);
        }
    }
    public void addTransition(int stateFrom, int stateTo, char symbol) {
        Transition trans = new Transition(stateFrom, stateTo, symbol);
        this.transitions.add(trans);
    }
    public void setFinalState(int finalState) {
        this.finalState = finalState;
    }
    public int getFinalState() {
        return this.finalState;
    }
    public List<Transition> getTransitions() {
        return this.transitions;
    }
    public void display() {
        for (Transition temp : transitions) {
            System.out.println("q" + temp.getFromState() + " " + temp.getSymbol() + "
--> q" + temp.getToState());
        }
```

```
        System.out.println("The final state is q" + getFinalState());
    }
    public ArrayList<Character> findPossibleInputSymbols(ArrayList<Integer> states) {
        ArrayList<Character> result = new ArrayList<>();
        for (int stateFrom : states) {
            for (Transition transition : transitions) {
                if (transition.getFromState() == stateFrom &&
transition.getSymbol() != 'e') {
                    result.add(transition.getSymbol());
                }
            }
        }
        return result;
    }
    public ArrayList<Integer> unique(ArrayList<Integer> list) {
        return IntStream
                .range(0, list.size())
                .filter(i -> ((i < list.size() - 1 && !list.get(i).equals(list.get(i
+ 1))) || i == list.size() - 1))

.mapToObj(list::get).collect(Collectors.toCollection(ArrayList::new));
    }
    public ArrayList<Integer> eclosure(ArrayList<Integer> states) {
        ArrayList<Integer> result = new ArrayList<>();
        boolean[] visited = new boolean[getStatesCount()];
        for (Integer integer : states) {
            eclosure(integer, result, visited);
        }
        Collections.sort(result);
        return unique(result);
    }
    public void eclosure(int x, ArrayList<Integer> result, boolean[] visited)
//Simple DFS
    {
        result.add(x);
        for (Transition transition : transitions) {
            if (transition.getFromState() == x && transition.getSymbol() == 'e') {
                int y = transition.getToState();
                if (!visited[y]) {
                    visited[y] = true;
                    eclosure(y, result, visited);
                }
            }
        }
    }
    public ArrayList<Integer> move(ArrayList<Integer> T, char symbol) {
        ArrayList<Integer> result = new ArrayList<>();
        for (int t : T) {
            for (Transition transition : transitions) {
                if (transition.getFromState() == t && transition.getSymbol() ==
symbol) {
                    result.add(transition.getToState());
                }
            }
        }
        Collections.sort(result);
        int l1 = result.size();
        unique(result);
        int l2 = result.size();
        if (l2 < l1) {
            System.out.println("move(T, a) returns non-unique ArrayList");
```

```
            System.exit(1);
        }
        return result;
    }
}
```

Листинг 2 – Класс, представляющий ДКА

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Set;
public class DFA {
    private final ArrayList<Transition> transitions = new ArrayList<>();
    private final ArrayList<ArrayList<Integer>> entries = new ArrayList<>();
    private final ArrayList<Boolean> marked = new ArrayList<>();
    private final ArrayList<Integer> finalStates = new ArrayList<>();
    /**
     * Add newly_created entry into DFA
     */
    public int addEntry(ArrayList<Integer> entry) {
        entries.add(entry);
        marked.add(false);
        return entries.size() - 1;
    }
    /**
     * Return the array position of the next unmarked entry
     */
    public int nextUnmarkedEntryIdx() {
        for (int i = 0; i < marked.size(); i++) {
            if (!marked.get(i)) {
                return i;
            }
        }
        return -1;
    }
    /**
     * mark the entry specified by index as marked (marked = true)
     */
    public void markEntry(int idx) {
        marked.set(idx, true);
    }
    public ArrayList<Integer> entryAt(int i) {
        return entries.get(i);
    }
    public int findEntry(ArrayList<Integer> entry) {
        for (int i = 0; i < entries.size(); i++) {
            ArrayList<Integer> it = entries.get(i);
            if (it.equals(entry)) {
                return i;
            }
        }
        return -1;
    }
    public void setFinalState(int nfaFinalState) {
        for (int i = 0; i < entries.size(); i++) {
            ArrayList<Integer> entry = entries.get(i);
            for (int state : entry) {
                if (state == nfaFinalState) {
                    finalStates.add(i);
                }
```

```
            }
        }
    }
    public void setMinDfaFinalState(int minDfaFinalState) {
        this.finalStates.add(minDfaFinalState);
    }
    public void setTransition(int fromState, int toState, char symbol) {
        Transition newTransition = new Transition(fromState, toState, symbol);
        transitions.add(newTransition);
    }
    public ArrayList<Integer> getFinalStates() {
        return finalStates;
    }
    public void display() {
        System.out.println();
        for (Transition transition : transitions) {
            System.out.println("q" + transition.getFromState() +
                    " {" + FAUtils.join(entries.get(transition.getFromState()), ",")
                    + "} " + transition.getSymbol() + " --> q" +
transition.getToState() +
                    " {" + FAUtils.join(entries.get(transition.getToState()), ",")
                    + "}");
        }
        System.out.println("The final states are q : " + FAUtils.join(finalStates,
","));
    }
    public void displayMinDFA() {
        System.out.println();
        for (Transition transition : transitions) {
            var printStr = "q" + transition.getFromState() + " " +
transition.getSymbol() +
                    " --> q" + transition.getToState();
            if (finalStates.contains(transition.getToState())) {
                printStr = printStr.concat(" final state ");
            }
            System.out.println(printStr);
        }
    }
    public boolean evaluate(String x) {
        int state = 0;
        for (var i = 0; i < x.length(); i++) {
            char ch = x.charAt(i);
            for (Transition transition : transitions) {
                if (transition.getFromState() == state && transition.getSymbol() ==
ch) {
                    state = transition.getToState();
                    break;
                }
            }
        }
        return finalStates.contains(state);
    }
    public ArrayList<Transition> getTransitions() {
        return transitions;
    }
    public int countStates() {
        var stateCount = new HashSet<Integer>();
        for (Transition transition : this.transitions) {
            stateCount.add(transition.getFromState());
            stateCount.add(transition.getToState());
        }
```

```
        return stateCount.size();
    }
    public Set<Integer> getAllStatesToBySymbol(int to, char symbol) {
        Set<Integer> result = new HashSet<>();
        for (Transition transition : this.transitions) {
            if (transition.getToState() == to && transition.getSymbol() == symbol) {
                result.add(transition.getFromState());
            }
        }
        return result;
    }
    private Set<Integer> getAllStatesFrom(int from) {
        Set<Integer> result = new HashSet<>();
        for (Transition transition : this.transitions) {
            if (transition.getFromState() == from) {
                result.add(transition.getToState());
            }
        }
        return result;
    }
    private Set<Integer> getStatesFromStartSet(Set<Integer> fromStart) {
        var result = new HashSet<Integer>();
        for (Integer i : fromStart) {
            result.addAll(getAllStatesFrom(i));
        }
        return result;
    }
    public Set<Integer> getReachableStatesFromStart() {
        var start = 0;
        var fromStart = getAllStatesFrom(start);
        Set<Integer> result = new HashSet<>(fromStart);
        Set<Integer> temp = new HashSet<>(result);
        var containAll = false;
        do {
            temp = getStatesFromStartSet(temp);
            containAll = result.containsAll(temp);
            result.addAll(temp);
        } while (!containAll);
        return result;
    }
}
```

Листинг 3 – Класс, создающий НКА по регулярному выражению

```
import java.util.*;

public class RegexRecognizer {
    //a.b
    private static NFA concat(NFA a, NFA b) {
        NFA result = new NFA();
        //No new state added in concatenation
        result.setStates(a.getStatesCount() + b.getStatesCount());
        //Copy all old transitions of a
        for (Transition transition : a.getTransitions()) {
            result.addTransition(transition.getFromState(), transition.getToState(),
transition.getSymbol());
        }
        //Creating the link; final state of a will link to initial state of b
        result.addTransition(a.getFinalState(), a.getStatesCount(), 'e');
        //Copy all old transitions of b with offset as a's states have already been
added
```

```
        var offset = a.getStatesCount();
        for (Transition transition : b.getTransitions()) {
            result.addTransition(transition.getFromState() + offset,
transition.getToState() + offset, transition.getSymbol());
        }
        //b is the final state of this created NFA
        result.setFinalState(offset + b.getStatesCount() - 1);
        return result;
    }
    //a*
    private static NFA kleene(NFA a) {
        NFA result = addStateBefore(a);
        var oldFinalState = a.getStatesCount();
        var oldInitialState = 1;
        var newInitialState = 0;
        var newFinalState = oldFinalState + 1;
        //Epsilon transition to new final state
        result.addTransition(oldFinalState, newFinalState, 'e');
        //Reverse epsilon transition
        result.addTransition(oldFinalState, oldInitialState, 'e');
        //Forward total epsilon transition
        result.addTransition(newInitialState, newFinalState, 'e');
        //Mark final state
        result.setFinalState(newFinalState);
        return result;
    }
    //a+
    private static NFA plus(NFA a) {
        NFA result = addStateBefore(a);
        var oldFinalState = a.getStatesCount();
        var oldInitialState = 1;
        var newFinalState = oldFinalState + 1;
        //Epsilon transition to new final state
        result.addTransition(oldFinalState, newFinalState, 'e');
        //Reverse epsilon transition
        result.addTransition(oldFinalState, oldInitialState, 'e');
        //Mark final state
        result.setFinalState(newFinalState);
        return result;
    }
    //s0->s1 as result s0->s1->s2 , where s0->s1 epsilon's transition
    private static NFA addStateBefore(NFA a) {
        NFA result = new NFA();
        /*
         * +2 because we will have one new initial state with epsilon transition to
a's initial
         * and one new final state with epsilon transition from a's final state and
from the new initial created
         */
        result.setStates(a.getStatesCount() + 2);
        result.addTransition(0, 1, 'e');
        for (Transition transition : a.getTransitions()) {
            result.addTransition(transition.getFromState() + 1,
transition.getToState() + 1, transition.getSymbol());
        }
        return result;
    }
    //a|b
    private static NFA orSelection(ArrayList<NFA> selections, int noOfSelections) {
        NFA result = new NFA();
        var stateCount = 2;
```

```
        //Find total states by summing all NFAs
        for (var i = 0; i < noOfSelections; i++) {
            stateCount += selections.get(i).getStatesCount();
        }
        result.setStates(stateCount);
        var adderTrack = 1;
        for (var i = 0; i < noOfSelections; i++) {
            //Initial epsilon transition to the first block of 'OR'
            result.addTransition(0, adderTrack, 'e');
            NFA selectedNFA = selections.get(i);
            for (Transition transition : selectedNFA.getTransitions()) {
                result.addTransition(transition.getFromState() + adderTrack,
transition.getToState() + adderTrack, transition.getSymbol());
            }
            adderTrack += selectedNFA.getStatesCount();
            //Add epsilon transition to final state
            result.addTransition(adderTrack - 1, stateCount - 1, 'e');
        }
        result.setFinalState(stateCount - 1);
        return result;
    }
    private static boolean isNotOperator(char currentSymbol) {
        return currentSymbol != '(' && currentSymbol != ')' && currentSymbol != '*'
                && currentSymbol != '|' && currentSymbol != '.' && currentSymbol !=
'+';
    }
    public static NFA regexToNfa(String regex) {
        regex = FAUtils.normalizeInputRegex(regex);
        Stack<Character> operators = new Stack<>();
        Stack<NFA> operands = new Stack<>();
        char operatorSymbol;
        int operatorCount;
        char currentSymbol;
        NFA newSym;
        char[] x = regex.toCharArray();
        for (char value : x) {
            currentSymbol = value;
            if (isNotOperator(currentSymbol)) //Must be a character, so build the
simplest NFA
            {
                newSym = new NFA();
                newSym.setStates(2);
                newSym.addTransition(0, 1, currentSymbol);
                newSym.setFinalState(1);
                operands.push(newSym);//push it back
            } else {
                switch (currentSymbol) {
                    case '*':
                        NFA starSym = operands.pop();
                        operands.push(kleene(starSym));
                        break;
                    case '+':
                        NFA plusSym = operands.pop();
                        operands.push(plus(plusSym));
                        break;
                    case '.':
                    case '|':
                    case '(':
                        operators.push(currentSymbol);
                        break;
                    default:
```

```
                        operatorCount = 0;
                        operatorSymbol = operators.peek();
                        //Keep searching operands
                        if (operatorSymbol == '(') {
                            continue;
                        }
                        //Collect operands
                        do {
                            operators.pop();
                            operatorCount++;
                        } while (operators.peek() != '(');
                        operators.pop();
                        NFA firstOperand;
                        NFA secondOperand;
                        ArrayList<NFA> selections = new ArrayList<>();
                        if (operatorSymbol == '.') {
                            for (int ii = 0; ii < operatorCount; ii++) {
                                secondOperand = operands.pop();
                                firstOperand = operands.pop();
                                operands.push(concat(firstOperand, secondOperand));
                            }
                        } else if (operatorSymbol == '|') {
                            for (int j = 0; j < operatorCount + 1; j++) {
                                selections.add(new NFA());
                            }
                            int tracker = operatorCount;
                            for (int k = 0; k < operatorCount + 1; k++) {
                                selections.set(tracker, operands.pop());
                                tracker--;
                            }
                            operands.push(orSelection(selections, operatorCount +
1));
                        }
                        break;
                    }
                }
            }
            return operands.peek();//Return the single entity. operands.poll() is also
fine
        }

}
```

Листинг 4 – Метод создания ДКА по НКА

```
public static DFA nfaToDfa(NFA nfa) {
    DFA dfa = new DFA();
    ArrayList<Integer> start = new ArrayList<>();
    start.add(0);
    ArrayList<Integer> s0 = nfa.eclosure(start);
    int stateFrom = dfa.addEntry(s0);
    while (stateFrom != -1) {
        ArrayList<Integer> T = dfa.entryAt(stateFrom);
        dfa.markEntry(stateFrom);
        ArrayList<Character> symbols = nfa.findPossibleInputSymbols(T);
        for (char a : symbols) {
            ArrayList<Integer> U = nfa.eclosure(nfa.move(T, a));
            int stateTo = dfa.findEntry(U);
            if (stateTo == -1) { // U not already in S'
```

```
                stateTo = dfa.addEntry(U);
            }
            dfa.setTransition(stateFrom, stateTo, a);
        }
        stateFrom = dfa.nextUnmarkedEntryIdx();
    }
    // The finish states of the DFA are those which contain any
    // of the finish states of the NFA.
    dfa.setFinalState(nfa.getFinalState());
    return dfa;
}
```

Листинг 5 – Класс, который минимизирует ДКА по алгоритму O(n^ 2) с построением пар различимых состояний

```
import java.util.*;
/**
 * DFAMinimizer minimizes DFA by O(n^2) algorithm with the
 * construction of pairs of distinguishable states
 */
public class DFAMinimizer {
    /**
     * Step 1
     * build a table of lists of inverse edges of size n×|Σ|
     * n - source DFA's states number
     */
    private static Map<FAEdge, Set<Integer>> buildInverseTransition(DFA dfa) {
        var n = dfa.countStates();
        Map<FAEdge, Set<Integer>> dfaInverseEdges = new HashMap<>();
        for (int i = 0; i < n; i++) {
            for (Character c : getLiterate(dfa)) {
                dfaInverseEdges.put(buildFAEdge(i, c), dfa.getAllStatesToBySymbol(i,
c));
            }
        }
        return dfaInverseEdges;
    }
    /**
     * Step 2
     * build an array of reachability of states from the starting — reachable of
size n
     */
    private static Map<Integer, Boolean> buildReachableStateFromStart(DFA dfa) {
        Map<Integer, Boolean> result = new HashMap<>();
        var reachableStatesFromStart = dfa.getReachableStatesFromStart();
        var dfaStateCount = dfa.countStates();
        for (int i = 0; i < dfaStateCount; i++) {
            result.put(i, reachableStatesFromStart.contains(i));
        }
        return result;
    }
    /**
     * get all terminal states
     */
    private static boolean[] getTerminalStateArray(DFA dfa) {
        var n = dfa.countStates();
        var finalStates = dfa.getFinalStates();
```

```
        boolean[] result = new boolean[n];
        for (var i = 0; i < n; i++) {
            result[i] = finalStates.contains(i);
        }
        return result;
    }
    /**
     * Step 3 and 4
     */
    private static boolean[][] buildTable(DFA dfa) {
        int n = dfa.countStates();
        boolean[] isTerminal = getTerminalStateArray(dfa);
        Map<FAEdge, Set<Integer>> dfaInverseEdges = buildInverseTransition(dfa);
        Stack<StatePair> statePairs = new Stack<>();
        Set<Character> literate = getLiterate(dfa);
        boolean[][] marked = new boolean[n][n];
        //Step 3
        for (var i = 0; i < n; i++) {
            for (var j = 0; j < n; j++) {
                if (!marked[i][j] && isTerminal[i] != isTerminal[j]) {
                    marked[i][j] = marked[j][i] = true;
                    statePairs.push(new StatePair(i, j));
                }
            }
        }
        //Step 4
        while (!statePairs.isEmpty()) {
            var headStatePair = statePairs.pop();
            for (Character c : literate) {
                var rList = dfaInverseEdges.get(buildFAEdge(headStatePair.getI(),
c));
                for (Integer r : rList) {
                    var sList = dfaInverseEdges.get(buildFAEdge(headStatePair.getJ(),
c));
                    for (Integer s : sList) {
                        if (!marked[r][s]) {
                            marked[r][s] = marked[s][r] = true;
                            statePairs.push(new StatePair(r, s));
                        }
                    }
                }
            }
        }
        return marked;
    }
    /**
     * Step 6
     * Build the minimized DFA
     */
    private static DFA buildDFA(int[] component, DFA sourceDFA) {
        DFA result = new DFA();
        var oldFinalsState = sourceDFA.getFinalStates();
        var n = sourceDFA.countStates();
        var equivalentStates = getEquivalentState(component, sourceDFA);
        for (var state = 0; state < n; state++) {
            var currentNewState = component[state];
            for (Transition transition : sourceDFA.getTransitions()) {
                if (transition.getFromState() == state) {
                    var toNewState = component[transition.getToState()];
                    result.setTransition(currentNewState, toNewState,
transition.getSymbol());
```

```
                }
            }
            if (oldFinalsState.contains(state)) {
                result.setMinDfaFinalState(currentNewState);
            }
        }
        return result;
    }
    private static Map<Integer, List<Integer>> getEquivalentState(int[] component,
DFA dfa) {
        var n = dfa.countStates();
        Map<Integer, List<Integer>> result = new HashMap<>();
        for (var i = 0; i < n; i++) {
            var index = new ArrayList<Integer>();
            for (var j = i; j < n; j++) {
                if (component[j] == i) {
                    index.add(j);
                }
            }
            if (index.size() >= 2) {
                result.put(i, index);
            }
        }
        return result;
    }
    private static boolean areEquivalentState(int firstState, int secondState,
Map<Integer, List<Integer>> mapEquivalentStates) {
        return (mapEquivalentStates.containsKey(firstState) &&
mapEquivalentStates.get(firstState).contains(secondState))
                || (mapEquivalentStates.containsKey(secondState) &&
mapEquivalentStates.get(secondState).contains(firstState));
    }
    private static FAEdge buildFAEdge(int state, char symbol) {
        return new FAEdge(state, symbol);
    }
    private static Set<Character> getLiterate(DFA dfa) {
        ArrayList<Transition> dfaTransitions = dfa.getTransitions();
        Set<Character> literate = new HashSet<>();
        for (Transition transition : dfaTransitions) {
            if (FAUtils.isInputCharacter(transition.getSymbol())) {
                literate.add(transition.getSymbol());
            }
        }
        return literate;
    }
    /**
     * Main's method
     * Minimizes the source DFA
     */
    public static DFA minimization(DFA dfa) {
        boolean[][] marked = buildTable(dfa);
        var reachable = buildReachableStateFromStart(dfa);
        var n = dfa.countStates();
        //Step 5
        int[] component = new int[n];
        Arrays.fill(component, -1);
        for (var i = 0; i < n; i++) {
            if (!marked[0][i]) {
                component[i] = 0;
            }
        }
```

```
        int componentsCount = 0;
        for (var i = 0; i < n; i++) {
            if (!reachable.get(i)) {
                continue;
            }
            if (component[i] == -1) {
                componentsCount++;
                component[i] = componentsCount;
                for (var j = i + 1; j < n; j++) {
                    if (!marked[i][j]) {
                        component[j] = componentsCount;
                    }
                }
            }
        }
        //Step 6
        return buildDFA(component, dfa);
    }
}
```

Листинг 6 – Основной метод выполнения программы

```
public static void recognise() {
    System.out.println("\nThe Thompson's Construction Algorithm takes a regular
expression as " +
            "an input and returns its corresponding Non-Deterministic Finite
Automaton \n");
    System.out.println("The current recognizer supports characters 'a' and 'b',
operations '.', '|', '+' and '*'");
    System.out.println("Metadata '(' and ')' \n\n");
    System.out.println("Enter the regular expression. Ex: (a|b)*aab");
    String regex = new Scanner(System.in).next();
    System.out.println("\nThe required NFA has the transitions: ");
    NFA requiredNfa;
    requiredNfa = regexToNfa(regex);
    requiredNfa.display();
    System.out.println("\nDFA :");
    DFA requiredDfa = nfaToDfa(requiredNfa);
    requiredDfa.display();
    DFA requiredMinDfa = DFAMinimizer.minimization(requiredDfa);
    System.out.println("\nmin DFA :");
    requiredMinDfa.displayMinDFA();
    String eval;
    do {
        System.out.println("Enter string to evaluate " + regex + " or tap 0 to
stop");
        eval = new Scanner(System.in).next();
        if (!eval.equals("0")) {
            if (requiredMinDfa.evaluate(eval)) {
                System.out.println(eval + " is accepted by regex " + regex);
            } else {
                System.out.println(eval + " is rejected by regex " + regex);
            }
        }
    } while (!eval.equals("0"));
}
```

## Набор тестов и ожидаемые результаты для проверки правильности программы

*Регулярное выражение (a|b)*abb*

1. Первый тест

   входная цепочка **abb**

   выход конечного автомата **да**
2. Второй тест

   входная цепочка **abbab**

   выход конечного автомата **нет**
3. Третий Тест

   входная цепочка **bbbbabb**

   выход конечного автомата **да**

*Регулярное выражение b(a|b)+*

1. Первый тест

   входная цепочка **ab**

   выход конечного автомата **нет**
2. Второй тест

   входная цепочка **abab**

   выход конечного автомата **да**
3. Третий Тест

   входная цепочка **bbabb**

   выход конечного автомата **да**

*Регулярное выражение b**

1. Первый тест

   входная цепочка **пустая цепочка (строка)**

   выход конечного автомата **да**
2. Второй тест

   входная цепочка **abab**

   выход конечного автомата **да**
3. Третий Тест

   входная цепочка **jacques**

   выход конечного автомата **да**

## Результаты выполнения программы

```
Enter the regular expression. Ex: (a|b)*aab
(a|b)*abb

The required NFA has the transitions:
q0 e --> q1
q1 e --> q2
q2 a --> q3
q3 e --> q6
q1 e --> q4
q4 b --> q5
q5 e --> q6
q6 e --> q7
q6 e --> q1
q0 e --> q7
q7 e --> q8
q8 a --> q9
q9 e --> q10
q10 b --> q11
q11 e --> q12
q12 b --> q13
The final state is q13
```

```
DFA :

q0 {0,1,2,4,7,8} a --> q1 {1,2,3,4,6,7,8,9,10}
q0 {0,1,2,4,7,8} b --> q2 {1,2,4,5,6,7,8}
q0 {0,1,2,4,7,8} a --> q1 {1,2,3,4,6,7,8,9,10}
q1 {1,2,3,4,6,7,8,9,10} a --> q1 {1,2,3,4,6,7,8,9,10}
q1 {1,2,3,4,6,7,8,9,10} b --> q3 {1,2,4,5,6,7,8,11,12}
q1 {1,2,3,4,6,7,8,9,10} a --> q1 {1,2,3,4,6,7,8,9,10}
q1 {1,2,3,4,6,7,8,9,10} b --> q3 {1,2,4,5,6,7,8,11,12}
q2 {1,2,4,5,6,7,8} a --> q1 {1,2,3,4,6,7,8,9,10}
q2 {1,2,4,5,6,7,8} b --> q2 {1,2,4,5,6,7,8}
q2 {1,2,4,5,6,7,8} a --> q1 {1,2,3,4,6,7,8,9,10}
q3 {1,2,4,5,6,7,8,11,12} a --> q1 {1,2,3,4,6,7,8,9,10}
q3 {1,2,4,5,6,7,8,11,12} b --> q4 {1,2,4,5,6,7,8,13}
q3 {1,2,4,5,6,7,8,11,12} a --> q1 {1,2,3,4,6,7,8,9,10}
q3 {1,2,4,5,6,7,8,11,12} b --> q4 {1,2,4,5,6,7,8,13}
q4 {1,2,4,5,6,7,8,13} a --> q1 {1,2,3,4,6,7,8,9,10}
q4 {1,2,4,5,6,7,8,13} b --> q2 {1,2,4,5,6,7,8}
q4 {1,2,4,5,6,7,8,13} a --> q1 {1,2,3,4,6,7,8,9,10}
The final states are q : 4
```

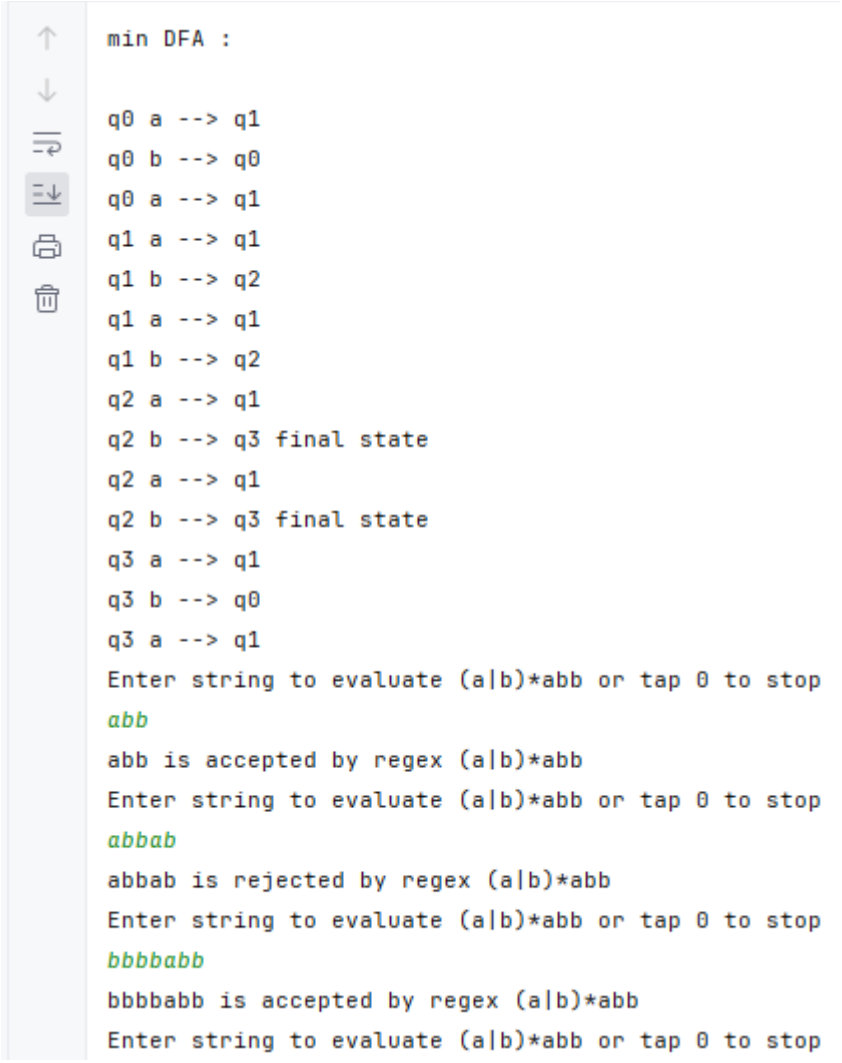


*Рис. 1: Результат выполения программы - регулярнное выражение (a|b)*abb*


*Рис. 2: Результат выполения программы - регулярнное выражение b(a|b)+*

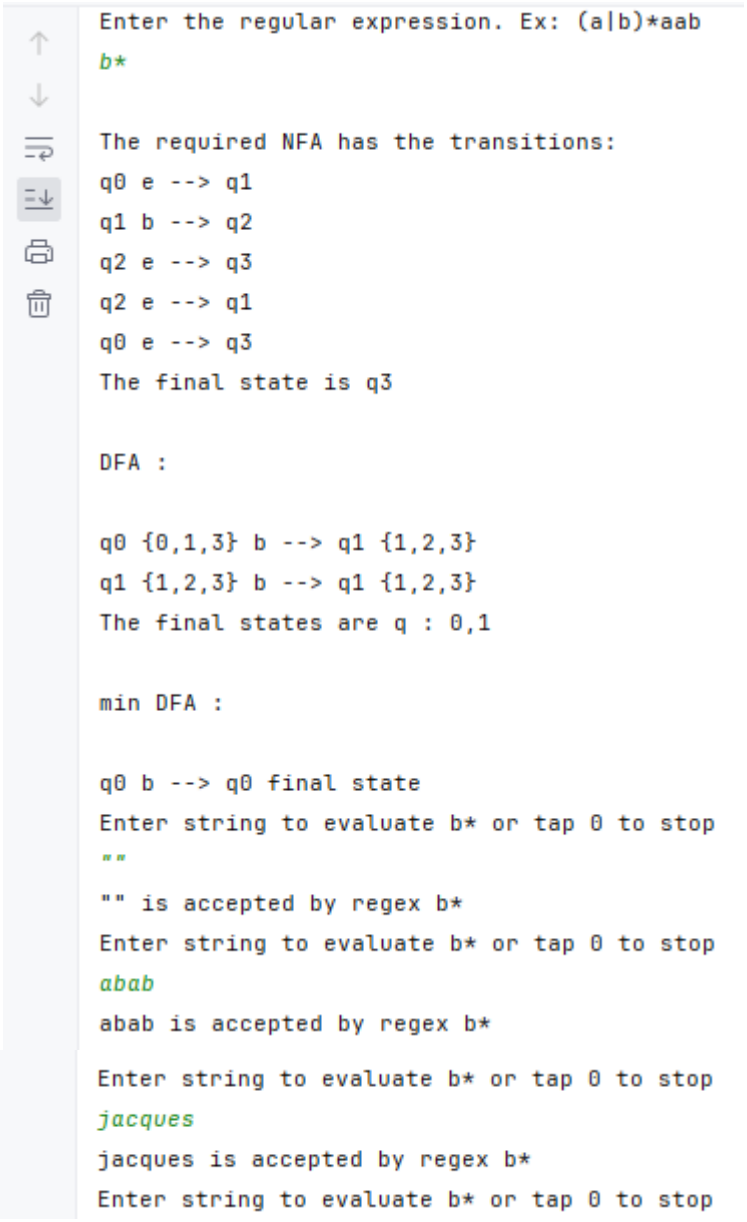


*Рис. 3: Результат выполения программы - регулярнное выражение b**