Государственное образовательное учреждение
высшего профессионального образования
“Московский государственный технический университет
имени Н.Э. Баумана”

Отчет

по лабораторной работе №1
по курсу “Конструирование компиляторов”
по теме
“Распознавание цепочек регулярного языка”

Студент:	Адамова И.О.
Группа:	ИУ7-22м
Вариант:	2
Преподаватель:	Ступников А.

Москва, 2020

Цель работы

Приобретение практических навыков реализации важнейших элементов лексических анализаторов на примере распознавания цепочек регулярного языка.

Задачи работы

1) Ознакомиться с основными понятиями и определениями, лежащими в основе построения лексических анализаторов.
2) Прояснить связь между регулярным множеством, регулярным выражением, праволинейным языком, конечно- автоматным языком и недетерминированным конечно-автоматным языком.
3) Разработать, тестировать и отладить программу распознавания цепочек регулярного или праволинейного языка в соответствии с предложенным вариантом грамматики.

Описание работы программы

На вход получаем регулярное выражение, в котором могут быть использованы буквы от ‘a’ до ‘z’ и от ‘A’ до ‘Z’, цифры от 0 до 9. ‘*’ означает замыкание Клини, ‘+’ замыкание, ‘|’ или, ‘(‘, ‘)’ скобки для приоритета операций.
В первую очередь к регулярному выражению добавляются точки, обозначающие конкатенацию, после чего регулярное выражение переводится в постфиксную форму. С помощью алгоритма Томпсона строится НКА, далее строится ДКА и минимизируется с помощью алгоритма Хопкрофта. После этого вводится слово для проверки, входит ли оно в язык, представляемый полученным ДКА, выводится цепочка состояний ДКА, через которые был произведен разбор слова, и ответ.
Для ДКА предусмотрен вывод в виде таблицы переходов и в виде списка состояний с переходами, выходящими из них. Для НКА - только вывод списка состояний с переходами из них.

Тесты

Для проверки проверки правильности работы программы, проведем тестирование на следующих тестах.

1) a*

Вывод:

```
NFA: start 2 end 3
state 0
trasitions: from: 0 to: 1 symbol: a
---------------
state 1
trasitions: from: 1 to: 3 symbol: eps

from: 1 to: 0 symbol: eps

---------------
state 2
trasitions: from: 2 to: 0 symbol: eps

from: 2 to: 3 symbol: eps

---------------
state 3 is final
---------------
DFA: state 0 is start  is final
from states: 0 2 3
trasitions: to: 1 symbol: a

state 1 is final
from states: 0 1 3
trasitions: to: 1 symbol: a

number of transitions 2
DFA: state 0 is start  is final
from states: 0 2 3
trasitions: to: 1 symbol: a

state 1 is final
from states: 0 1 3
trasitions: to: 1 symbol: a

Trasitions table
  |a |
0 |1 |S F
1 |1 |F

Back trasitions table
  |a |
0 |  |S F
1 |0 1 |F
min DFA: state 0 is start  is final
from states: 0 1
trasitions: to: 0 symbol: a

Trasitions table
  |a|
0 |0 |S F
```

Введем слово, соответствующее регулярному выражению - aaaab.

Вывод:

```
-0-> aaaab -0-> aaab -0-> aab -0-> ab -0-> | fail.
Letter b from your word is not in the alfabet.
Given word doesn't match your regular expression
```

Введем слово, соответствующее регулярному выражению - аааа.
Вывод:
```
-0-> aaaa -0-> aaa -0-> aa -0-> a -0-> | success
Given word matches your regular expression
```
2) ababa|ababb

Вывод:
```
NFA: start 20 end 21
state 0
trasitions: from: 0 to: 1 symbol: a
---------------
state 1
trasitions: from: 1 to: 2 symbol: eps

---------------
state 2
trasitions: from: 2 to: 3 symbol: b
---------------
state 3
trasitions: from: 3 to: 4 symbol: eps

---------------
state 4
trasitions: from: 4 to: 5 symbol: a
---------------
state 5
trasitions: from: 5 to: 6 symbol: eps

---------------
state 6
trasitions: from: 6 to: 7 symbol: b
---------------
state 7
trasitions: from: 7 to: 8 symbol: eps

---------------
state 8
trasitions: from: 8 to: 9 symbol: a
---------------
state 9
trasitions: from: 9 to: 21 symbol: eps

---------------
state 10
trasitions: from: 10 to: 11 symbol: a
---------------
state 11
trasitions: from: 11 to: 12 symbol: eps

---------------
state 12
trasitions: from: 12 to: 13 symbol: b
---------------
state 13
trasitions: from: 13 to: 14 symbol: eps

---------------
state 14
trasitions: from: 14 to: 15 symbol: a
---------------
state 15
```

```
trasitions: from: 15 to: 16 symbol: eps

---------------
state 16
trasitions: from: 16 to: 17 symbol: b
---------------
state 17
trasitions: from: 17 to: 18 symbol: eps

---------------
state 18
trasitions: from: 18 to: 19 symbol: b
---------------
state 19
trasitions: from: 19 to: 21 symbol: eps

---------------
state 20
trasitions: from: 20 to: 0 symbol: eps

from: 20 to: 10 symbol: eps

---------------
state 21 is final
---------------
number of transitions 6
DFA: state 0 is start
from states: 0 10 20
trasitions: to: 1 symbol: a
to: 7 symbol: b

state 1
from states: 1 2 11 12
trasitions: to: 7 symbol: a
to: 2 symbol: b

state 2
from states: 3 4 13 14
trasitions: to: 3 symbol: a
to: 7 symbol: b

state 3
from states: 5 6 15 16
trasitions: to: 7 symbol: a
to: 4 symbol: b

state 4
from states: 7 8 17 18
trasitions: to: 5 symbol: a
to: 6 symbol: b

state 5 is final
from states: 9 21
trasitions: to: 7 symbol: a
to: 7 symbol: b

state 6 is final
from states: 19 21
trasitions: to: 7 symbol: a
to: 7 symbol: b

state 7
```

```
from states:
trasitions: to: 7 symbol: a
to: 7 symbol: b

Trasitions table
  |a          |b      |
0 |1          |7      |S
1 |7          |2      |
2 |3          |7      |
3 |7          |4      |
4 |5          |6      |
5 |7          |7      |F
6 |7          |7      |F
7 |7          |7      |

Back trasitions table
  |a          |b      |
0 |           |       |S
1 |0          |       |
2 |           |1      |
3 |2          |       |
4 |           |3      |
5 |4          |       |F
6 |           |4      |F
7 |1 3 5 6 7 |0 2 5 6 7 |
min DFA: state 0 is final
from states: 5 6
trasitions: to: 5 symbol: a
to: 5 symbol: b

state 1
from states: 4
trasitions: to: 0 symbol: a
to: 0 symbol: b

state 2
from states: 3
trasitions: to: 5 symbol: a
to: 1 symbol: b

state 3
from states: 2
trasitions: to: 2 symbol: a
to: 5 symbol: b

state 4
from states: 1
trasitions: to: 5 symbol: a
to: 3 symbol: b

state 5
from states: 7
trasitions: to: 5 symbol: a
to: 5 symbol: b

state 6 is start
from states: 0
trasitions: to: 4 symbol: a
to: 5 symbol: b

Trasitions table
  |a          |b      |
```

```
0 |5          |5     |F
1 |0          |0     |
2 |5          |1     |
3 |2          |5     |
4 |5          |3     |
5 |5          |5     |
6 |4          |5     |S
```

Введем слово, не соответствующее регулярному выражению - ababab.
Вывод:

```
-6-> ababab -4-> babab -3-> abab -2-> bab -1-> ab -0-> b -5-> | fail
Given word doesn't match your regular expression
```

3) (acr+f*)+

```
NFA: start 12 end 13
state 0
trasitions: from: 0 to: 1 symbol: a
---------------
state 1
trasitions: from: 1 to: 2 symbol: eps

---------------
state 2
trasitions: from: 2 to: 3 symbol: c
---------------
state 3
trasitions: from: 3 to: 6 symbol: eps

---------------
state 4
trasitions: from: 4 to: 5 symbol: r
---------------
state 5
trasitions: from: 5 to: 7 symbol: eps

from: 5 to: 4 symbol: eps

---------------
state 6
trasitions: from: 6 to: 4 symbol: eps

---------------
state 7
trasitions: from: 7 to: 10 symbol: eps

---------------
state 8
trasitions: from: 8 to: 9 symbol: f
---------------
state 9
trasitions: from: 9 to: 11 symbol: eps

from: 9 to: 8 symbol: eps

---------------
state 10
trasitions: from: 10 to: 8 symbol: eps

from: 10 to: 11 symbol: eps

---------------
state 11
```

```
trasitions: from: 11 to: 13 symbol: eps

from: 11 to: 0 symbol: eps

---------------
state 12
trasitions: from: 12 to: 0 symbol: eps

---------------
state 13 is final
---------------
number of transitions 8
DFA: state 0 is start
from states: 0 12
trasitions: to: 1 symbol: a
to: 5 symbol: c
to: 5 symbol: f
to: 5 symbol: r

state 1
from states: 1 2
trasitions: to: 5 symbol: a
to: 2 symbol: c
to: 5 symbol: f
to: 5 symbol: r

state 2
from states: 3 4 6
trasitions: to: 5 symbol: a
to: 5 symbol: c
to: 5 symbol: f
to: 3 symbol: r

state 3 is final
from states: 0 4 5 7 8 10 11 13
trasitions: to: 1 symbol: a
to: 5 symbol: c
to: 4 symbol: f
to: 3 symbol: r

state 4 is final
from states: 0 8 9 11 13
trasitions: to: 1 symbol: a
to: 5 symbol: c
to: 4 symbol: f
to: 5 symbol: r

state 5
from states:
trasitions: to: 5 symbol: a
to: 5 symbol: c
to: 5 symbol: f
to: 5 symbol: r

Trasitions table
  |a   |c     |f     |r     |
0 |1   |5     |5     |5     |S
1 |5   |2     |5     |5     |
2 |5   |5     |5     |3     |
3 |1   |5     |4     |3     |F
4 |1   |5     |4     |5     |F
5 |5   |5     |5     |5     |
```

```
Back trasitions table
  |a   |c     |f     |r     |
0 |    |      |      |      |S
1 |0 3 4  |   |      |      |
2 |    |1     |      |      |
3 |    |      |      |2 3   |F
4 |    |      |3 4   |      |F
5 |1 2 5  |0 2 3 4 5 | 0 1 2 5 |0 1 4 5 |
min DFA: state 0 is final
from states: 4
trasitions: to: 3 symbol: a
to: 4 symbol: c
to: 0 symbol: f
to: 4 symbol: r

state 1
from states: 2
trasitions: to: 4 symbol: a
to: 4 symbol: c
to: 4 symbol: f
to: 2 symbol: r

state 2 is final
from states: 3
trasitions: to: 3 symbol: a
to: 4 symbol: c
to: 0 symbol: f
to: 2 symbol: r

state 3
from states: 1
trasitions: to: 4 symbol: a
to: 1 symbol: c
to: 4 symbol: f
to: 4 symbol: r

state 4
from states: 5
trasitions: to: 4 symbol: a
to: 4 symbol: c
to: 4 symbol: f
to: 4 symbol: r

state 5 is start
from states: 0
trasitions: to: 3 symbol: a
to: 4 symbol: c
to: 4 symbol: f
to: 4 symbol: r

Trasitions table
  |a   |c     |f     |r     |
0 |3   |4     |0     |4     |F
1 |4   |4     |4     |2     |
2 |3   |4     |0     |2     |F
3 |4   |1     |4     |4     |
4 |4   |4     |4     |4     |
5 |3   |4     |4     |4     |S
```

Введем слово, соответствующее регулярному выражению - acracrff.

Вывод:

```
-5-> acracrff -3-> cracrff -1-> racrff -2-> acrff -3-> crff -1-> rff -2-> ff -0-> f -0-> |
success
Given word matches your regular expression
```

Введем слово, не соответствующее регулярному выражению - acracacrf.

Вывод:

```
-5-> acracacrf -3-> cracacrf -1-> racacrf -2-> acacrf -3-> cacrf -1-> acrf -4-> crf -4-> rf
-4-> f -4-> | fail
Given word doesn't match your regular expression
```

Заключение

В результате лабораторной работы была разработана программа на языке C++, принимающая на вход регулярное выражение и слово на проверку, соответствует ли оно данному регулярному выражению. В процессе работы программа строит НКА, ДКА, минимизирует ДКА и осуществляет требуемую проверку.
По результатам тестирования программы можно сделать вывод, что программа работает корректно. Приобретены практические навыки реализации важнейших элементов лексических анализаторов на примере распознавания цепочек регулярного языка.

Листинг

```
#include <string>
#include <iostream>
#include <stack>
#include <vector>
#include <set>
#include <map>
#include <utility>
#include "stdio.h"

// STEP 1: insert concatenation operator to regular expression
void addConcatSymbol(std::string &s) {
   for (int i = 0; i < s.length(); i++) {
        if (s[i] == '(' || s[i] == '|')
             continue;

        if (i < s.length() - 1) {
             if (s[i + 1] == '*' || s[i + 1] == '+' || s[i + 1] == '*' || s[i + 1] == '|' || s[i + 1] == ')')
                  continue;
             else {
                  s.insert(i + 1, ".");
                  i++;
             }
```

```
        }
    }
}

// STEP 2: convert regular expression to postfix form
bool isOperator(char c) {
    return c == '.' || c == '*' || c =='|' || c == '+';
}

int getPrecedence(char c) {
    if (c == '|')
        return 0;
    else if (c == '.')
        return 1;
    else if (c == '*' || c == '+')
        return 2;
    else return -1;
}

std::string makePostfixForm(std::string &s, std::set<char> &alfabet) {
    std::stack<char> operatorStack;
    std::string outputQueue;
    int len = s.length();

    for (int i = 0; i < len; i++) {
        if ((s[i] >= 'a' && s[i] <= 'z') || (s[i] >= 'A' && s[i] <= 'Z')) {
            outputQueue.push_back(s[i]);
            alfabet.insert(s[i]);
        }
        else if (isOperator(s[i])) {
            while (    not(operatorStack.empty()) &&
                    isOperator(operatorStack.top()) &&
                    (getPrecedence(operatorStack.top()) >= getPrecedence(s[i])))
{
                outputQueue.push_back(operatorStack.top());
                operatorStack.pop();
            }
            operatorStack.push(s[i]);
        } else if (s[i] == '(') {
            operatorStack.push(s[i]);
        } else if (s[i] == ')') {
            while (operatorStack.top() != '(') {
                outputQueue.push_back(operatorStack.top());
                operatorStack.pop();
            }
```

```
                        if (operatorStack.top() == '(')
                                operatorStack.pop();
                }
        }
        while (not(operatorStack.empty())) {
                outputQueue.push_back(operatorStack.top());
                operatorStack.pop();
        }
        return outputQueue;
}

// STEP 3: build NFA from regular expression
struct trasition {
    int fromState;
    int toState;
    char symbol;
};

struct epsilonTrasition {
    int fromState;
    int toState;
};

struct state {
    int state;
    std::vector<trasition*> trasitions;
    std::vector<epsilonTrasition*> epsilonTrasitions;
    bool isFinal;
};

state* createState(int n, bool isFinal_) {
    state *newState = new state;
    newState->state = n;
    newState->isFinal = isFinal_;
    return newState;
}

trasition* createTransition(int fromState_, int toState_, char symbol_) {
    trasition *newTransition = new trasition;
    newTransition->fromState = fromState_;
    newTransition->toState = toState_;
    newTransition->symbol = symbol_;
    return newTransition;
}

epsilonTrasition* createEpsilonTransition(int fromState_, int toState_) {
```

```
    epsilonTrasition *newTransition = new epsilonTrasition;
    newTransition->fromState = fromState_;
    newTransition->toState = toState_;
    return newTransition;
}

void printTransition(trasition *transition_) {
    std::cout << "from: " << transition_->fromState << " to: " << transition_->toState << "
symbol: " << transition_->symbol << '\n';
}

void printEpsilonTransition(epsilonTrasition *transition_) {
    std::cout << "from: " << transition_->fromState << " to: " << transition_->toState << "
symbol: eps\n" << std::endl;;
}

void printState(state *state_) {
    std::cout << "state " << state_->state << (state_->isFinal ? " is final" : "") << std::endl;
    if (state_->isFinal) return;
    std::cout << "trasitions: ";
    for (int i = 0; i < state_->trasitions.size(); i++) {
        printTransition(state_->trasitions[i]);
    }
    for (int i = 0; i < state_->epsilonTrasitions.size(); i++) {
        printEpsilonTransition(state_->epsilonTrasitions[i]);
    }
}

class NFA {
public:
    std::map<int, state*> states;
    state *start, *end;

    NFA() {
        start = NULL;
        end = NULL;
    }

    ~NFA() {}

    void addState(state *newState, bool isStart, bool isEnd) {
        states.insert(std::pair<int, state*>(newState->state, newState));
        if (isStart) start = newState;
        if (isEnd) end = newState;
    }
```

```
    void statesUnion(std::map<int, state*> &newStates) {
        states.insert(newStates.begin(), newStates.end());
    }

    void printNFA() {
        std::cout << "\nNFA: start " << start->state << " end " << end->state << std::endl;
        std::map<int, state*>::iterator it;
        for (it = states.begin(); it != states.end(); ++it) {
            printState(it->second);
            std::cout << "---------------" << std::endl;
        }
    }
};

NFA postfixToNFA(std::string &postfix) {
    std::stack<NFA> automataStack;
    int len = postfix.length();
    int stateCounter = 0;

    for (int i = 0; i < len; i++) {
        switch (postfix[i]) {
            case '|': {
                NFA nfa1, nfa2;
                nfa2 = automataStack.top();
                automataStack.pop();
                nfa1 = automataStack.top();
                automataStack.pop();
                state *stateStart = createState(stateCounter, false);
                stateCounter++;
                state *stateEnd = createState(stateCounter, true);
                stateCounter++;
                nfa1.end->isFinal = false;
                nfa2.end->isFinal = false;

stateStart->epsilonTrasitions.push_back(createEpsilonTransition(stateStart->state,
nfa1.start->state));

stateStart->epsilonTrasitions.push_back(createEpsilonTransition(stateStart->state,
nfa2.start->state));

nfa1.end->epsilonTrasitions.push_back(createEpsilonTransition(nfa1.end->state,
stateEnd->state));

nfa2.end->epsilonTrasitions.push_back(createEpsilonTransition(nfa2.end->state,
stateEnd->state));
                nfa1.statesUnion(nfa2.states);
```

```
                    nfa1.states.insert(std::pair<int, state*>(stateStart->state, stateStart));
                    nfa1.states.insert(std::pair<int, state*>(stateEnd->state, stateEnd));
                    nfa1.start = stateStart;
                    nfa1.end = stateEnd;
                    //nfa1.printNFA();
                    automataStack.push(nfa1);
                    break;
                }
                case '*': {
                    NFA nfa;
                    nfa = automataStack.top();
                    automataStack.pop();
                    state *stateStart = createState(stateCounter, false);
                    stateCounter++;
                    state *stateEnd = createState(stateCounter, true);
                    stateCounter++;
                    nfa.end->isFinal = false;

stateStart->epsilonTrasitions.push_back(createEpsilonTransition(stateStart->state,
nfa.start->state));

nfa.end->epsilonTrasitions.push_back(createEpsilonTransition(nfa.end->state,
stateEnd->state));

nfa.end->epsilonTrasitions.push_back(createEpsilonTransition(nfa.end->state,
nfa.start->state));

stateStart->epsilonTrasitions.push_back(createEpsilonTransition(stateStart->state,
stateEnd->state));
                    nfa.start = stateStart;
                    nfa.end = stateEnd;
                    nfa.states.insert(std::pair<int, state*>(stateStart->state, stateStart));
                    nfa.states.insert(std::pair<int, state*>(stateEnd->state, stateEnd));
                    //nfa.printNFA();
                    automataStack.push(nfa);
                    break;
                }
                case '.': {
                    NFA nfa1, nfa2;
                    nfa2 = automataStack.top();
                    automataStack.pop();
                    nfa1 = automataStack.top();
                    automataStack.pop();

nfa1.end->epsilonTrasitions.push_back(createEpsilonTransition(nfa1.end->state,
nfa2.start->state));
```

```
                    nfa1.end->isFinal = false;
                    nfa1.end = nfa2.end;
                    nfa1.statesUnion(nfa2.states);
                    //nfa1.printNFA();
                    automataStack.push(nfa1);
                    break;
                }
                case '+': {
                    NFA nfa;
                    nfa = automataStack.top();
                    automataStack.pop();
                    state *stateStart = createState(stateCounter, false);
                    stateCounter++;
                    state *stateEnd = createState(stateCounter, true);
                    stateCounter++;
                    nfa.end->isFinal = false;

stateStart->epsilonTrasitions.push_back(createEpsilonTransition(stateStart->state,
nfa.start->state));

nfa.end->epsilonTrasitions.push_back(createEpsilonTransition(nfa.end->state,
stateEnd->state));

nfa.end->epsilonTrasitions.push_back(createEpsilonTransition(nfa.end->state,
nfa.start->state));
                    nfa.start = stateStart;
                    nfa.end = stateEnd;
                    nfa.states.insert(std::pair<int, state*>(stateStart->state, stateStart));
                    nfa.states.insert(std::pair<int, state*>(stateEnd->state, stateEnd));
                    //nfa.printNFA();
                    automataStack.push(nfa);
                    break;
                }
                default: {    // letters
                    state *state1 = createState(stateCounter, false);
                    stateCounter++;
                    state *state2 = createState(stateCounter, true);
                    stateCounter++;
                    state1->trasitions.push_back(createTransition(state1->state,
state2->state, postfix[i]));
                    NFA newNFA;
                    newNFA.addState(state1, true, false);
                    newNFA.addState(state2, false, true);
                    //newNFA.printNFA();
                    automataStack.push(newNFA);
                    break;
```

```
                    }
            }
    }

    return automataStack.top();
}

// STEP 4: building DFA from NFA
struct DFAState
{
    int state;
    std::set<int> fromNFAstates;
    std::map<char, int> transitions;
    bool isFinal;
};

DFAState *createDFAState(int state_, std::set<int> fromStates, bool isFinal_) {
    DFAState *newState = new DFAState;
    newState->state = state_;
    newState->fromNFAstates = fromStates;
    newState->isFinal = isFinal_;
    return newState;
}

void printDFAState(DFAState *state, bool isStart) {
    std::cout << "state " << state->state << (isStart ? " is start " : "") <<
                                (state->isFinal ? " is final " : "") << std::endl;
    std::cout << "from states: ";
    for (std::set<int>::iterator itr = state->fromNFAstates.begin(); itr !=
state->fromNFAstates.end(); ++itr) {
            std::cout << (*itr) << " ";
    }
    std::cout << "\ntrasitions: ";
    if (state->transitions.empty()) {
            std::cout << "\n";
            return;
    }
    for (std::map<char, int>::iterator itr = state->transitions.begin(); itr !=
state->transitions.end(); ++itr) {
            std::cout << "to: " << (*itr).second << " symbol: " << (*itr).first << '\n';
    }
    std::cout << "\n";
}

class DFA
{
```

```
public:
    std::map<int, DFAState*> states;
    DFAState *start;

    DFA() {
            start = NULL;
    }
    ~DFA() {}
    void addDFAState(DFAState *newState, bool isStart) {
            states.insert(std::pair<int, DFAState*>(newState->state, newState));
            if (isStart) start = newState;
    }
    void printDFA() {
            printf("DFA: ");
            for (std::map<int, DFAState*>::iterator itr = states.begin(); itr != states.end(); ++itr) {
                    printDFAState((*itr).second, this->start->state == itr->first);
            }
    }

};

std::set<int> findTransitionsByLetter(std::vector<trasition*> trasitions, char letter) {
    std::set<int> trasitionsTo;

    for (std::vector<trasition*>::iterator itr = trasitions.begin(); itr < trasitions.end(); ++itr) {
            if ((*itr)->symbol == letter) {
                    trasitionsTo.insert((*itr)->toState);
            }
    }
    return trasitionsTo;
}

std::set<int> epsilonClosure(NFA nfa, state *curState) {
    std::set<int> curEpsilonClosure;
    curEpsilonClosure.insert(curState->state);
    int epsilonTrasitionsNumber = curState->epsilonTrasitions.size();
    for (int i = 0; i < epsilonTrasitionsNumber; i++) {
            std::set<int> newSet;
            int from = curState->epsilonTrasitions[i]->toState;
            newSet = epsilonClosure(nfa, nfa.states[from]);
            curEpsilonClosure.insert(newSet.begin(), newSet.end());
    }
    std::set<int>::iterator itr; /*
    std::cout << "epsilonClosure " << curState->state << ": ";
    for (itr = curEpsilonClosure.begin(); itr != curEpsilonClosure.end(); ++itr)
           std::cout << (*itr) << ' ';
```

```
		std::cout << std::endl;*/
	return curEpsilonClosure;
}

bool isFinal(NFA nfa, std::set<int> states) {
	for (std::set<int>::iterator itr = states.begin(); itr != states.end(); ++itr)
		if (nfa.end->state == (*itr))
			return true;
	return false;
}

int stateExcists(DFA &dfa, std::vector<std::pair<int, std::set<int> > > queue, std::set<int> set)
{
	for (std::vector<std::pair<int, std::set<int> > >::iterator queueItr = queue.begin();
			queueItr != queue.end(); ++queueItr) {
		if (set == (*queueItr).second) {
			return (*queueItr).first;
		}
	}
	std::map<int, DFAState*> &states = dfa.states;
	for (std::map<int, DFAState*>::iterator itr = states.begin(); itr != states.end(); ++itr) {
		if ((*itr).second->fromNFAstates == set) {
			return (*itr).first;
		}
	}
	return -1;
}

DFA NFAtoDFA(NFA nfa, std::set<char> alfabet) {
	std::set<int> startState;
	std::vector<state> DFAStates;
	std::vector<std::pair<int, std::set<int> > > stateQueue;
	int stateCounter = 0;
	DFA dfa;

	startState = epsilonClosure(nfa, nfa.start);
	stateQueue.push_back(std::make_pair(stateCounter, startState));
	stateCounter++;

	while (not(stateQueue.empty())) { // в каждом состоянии из множества найти все
переходы по каждой букве из алфавита
		std::set<int> currentState = stateQueue.front().second;
		int nState = stateQueue.front().first;
		bool isFinal_ = isFinal(nfa, stateQueue.front().second);
		dfa.addDFAState(createDFAState(stateQueue.front().first,
stateQueue.front().second, isFinal_), nState == 0);
```

```
            // for each letter in the alfabet
            for (std::set<char>::iterator alfabetItr = alfabet.begin(); alfabetItr != alfabet.end();
++alfabetItr) {
                        std::set<int> newState;
                        // for each state of nfa from the set
                        for (std::set<int>::iterator stateItr = currentState.begin(); stateItr !=
currentState.end(); ++stateItr) {
                                    int nfaState = (*stateItr);
                                    // get transitions by the letter
                                    std::set<int> state_ =
findTransitionsByLetter(nfa.states[nfaState]->trasitions, (*alfabetItr));
                                    // create new state where all the transitions go
                                    newState.insert(state_.begin(), state_.end());
                        }

                        if (not(newState.empty())) {
                                    // find epsilon closure of these transitions
                                    std::set<int> finalSetOfStates;
                                    for (std::set<int>::iterator itr = newState.begin(); itr != newState.end();
++itr) {
                                                std::set<int> fromEpsClosure = epsilonClosure(nfa,
nfa.states[(*itr)]);
                                                finalSetOfStates.insert(fromEpsClosure.begin(),
fromEpsClosure.end());
                                    }
                                    // add new set of NFA states to queue if they haven't been there
                                    int n = stateExcists(dfa, stateQueue, finalSetOfStates);
                                    if (n >= 0) {

dfa.states[nState]->transitions.insert(std::make_pair((*alfabetItr), n));
                                    } else {
                                                stateQueue.push_back(std::make_pair(stateCounter,
finalSetOfStates));

dfa.states[nState]->transitions.insert(std::make_pair((*alfabetItr), stateCounter++));
                                    }
                        }

            }
            stateQueue.erase(stateQueue.begin());
    }
    //dfa.printDFA();
    return dfa;
}
```

```
// STEP 5: minimize DFA
void addDeadState(DFA &dfa, std::set<char> &alfabet) { // add dead state where
non-excisting edges go
    std::set<int> emptySet;
    int deadStateN = dfa.states.size();
    DFAState *deadState = createDFAState(deadStateN, emptySet, false);
    for (std::set<char>::iterator itr = alfabet.begin(); itr != alfabet.end(); ++itr) {
        deadState->transitions.insert(std::make_pair((*itr), deadStateN));
        for (std::map<int, DFAState*>::iterator stateItr = dfa.states.begin(); stateItr !=
dfa.states.end(); ++stateItr) {
            if (stateItr->second->transitions.count((*itr)) < 1) {
                stateItr->second->transitions.insert(std::make_pair((*itr),
deadStateN));
            }
        }
    }
    dfa.addDFAState(deadState, false);
}

int printIntVector(std::vector<int> &vector) {
    int len = 0;
    for (std::vector<int>::iterator itr = vector.begin(); itr != vector.end(); ++itr) {
        printf("%d ", (*itr));
        len += std::to_string(*itr).length() + 1;
    }
    return len;
}

void printPartition(std::vector<std::vector<int> > &partition) {
    printf("partition: \n");
    int length = partition.size();
    for (int i = 0; i < length; i++) {
        printf("state set %d: ", i);
        printIntVector(partition[i]);
        printf("\n");
    }
    printf("======\n");
}

void initialPartition(DFA &dfa, std::vector<std::vector<int> > &partition, std::vector<int>
&classAttachment) {
    std::vector<int> final, notFinal;
    for (std::map<int, DFAState*>::iterator itr = dfa.states.begin(); itr != dfa.states.end(); ++itr)
{
        if (itr->second->isFinal) {
            final.push_back(itr->second->state);
```

```
                classAttachment[itr->first] = 0;
            } else {
                notFinal.push_back(itr->second->state);
                classAttachment[itr->first] = 1;
            }
    }
    //if (not(notFinal.empty()))

    partition.push_back(final);
    if (not(notFinal.empty()))
        partition.push_back(notFinal);
    //partition.push_back(notFinal);
}

void printQueue(std::vector<std::pair<int, char> > queue) {
    int length = queue.size();
    printf("queue: \n");
    for (int i = 0; i < length; i++) {
        printf("%d %c\n", queue[i].first, queue[i].second);
    }
    printf("======\n");
}

int findMax(std::set<char> &alfabet) {
        int max = 0;
        if (!alfabet.empty())
        max = *(alfabet.rbegin());
        return max;
}

void printTable(std::vector<std::vector<std::vector<int> > > &transitionsTable, std::set<char>
&alfabet, DFA &dfa) {

    int width = dfa.states.size() * 1.3;
    int maxStateLen = std::to_string(dfa.states.size()).length();
    printf("%*c", maxStateLen + 2, '|');
    for (std::set<char>::iterator itr = alfabet.begin(); itr != alfabet.end(); ++itr) {
        printf("%-*c|", width, *itr);
    }

    printf("\n");
    for (std::map<int, DFAState*>::iterator stateItr = dfa.states.begin(); stateItr !=
dfa.states.end(); ++stateItr) {
        printf("%-*d|", maxStateLen + 1, stateItr->first);
        for (std::set<char>::iterator itr = alfabet.begin(); itr != alfabet.end(); ++itr) {
                int len = printIntVector(transitionsTable[stateItr->first][(*itr)]);
```

```
				printf("%*c", width - len + 1, '|');
			}
			if (stateItr->first == dfa.start->state)
				printf("S ");
			if (stateItr->second->isFinal)
				printf("F ");
			printf("\n");
	}
}

void makeTransitionsTable(DFA &dfa, std::set<char> &alfabet) {
	int numberOfStates = dfa.states.size();
	int maxLetterCode = findMax(alfabet);
	std::vector<std::vector<std::vector<int> > > transitionsTable(numberOfStates,
			std::vector<std::vector<int> >(maxLetterCode+1, std::vector<int>(0)));
	// просмотриваем каждое состояние автомата
	for (std::map<int, DFAState*>::iterator stateItr = dfa.states.begin(); stateItr !=
dfa.states.end(); ++stateItr) {
			//и каждую букву алфавита (из каждого состояния есть переход по каждой
букве)
			for (std::set<char>::iterator alfabetItr = alfabet.begin(); alfabetItr != alfabet.end();
++alfabetItr) {
				// добавляем элемент таблицы по индексу [номер состояния, откуда
переход][буква]
				// куда переход

transitionsTable[stateItr->first][(*alfabetItr)].push_back(stateItr->second->transitions[(*alfabet
Itr)]);
			}
	}
	printf("Trasitions table\n");
	printTable(transitionsTable, alfabet, dfa);
}

void makeBackTransitionsTable(DFA &dfa, std::vector<std::vector<std::vector<int> > >
&transitionsTable, std::set<char> &alfabet) {
	// просмотриваем каждое состояние автомата
	for (std::map<int, DFAState*>::iterator stateItr = dfa.states.begin(); stateItr !=
dfa.states.end(); ++stateItr) {
			//и каждую букву алфавита (из каждого состояния есть переход по каждой
букве)
			for (std::set<char>::iterator alfabetItr = alfabet.begin(); alfabetItr != alfabet.end();
++alfabetItr) {
				// добавляем элемент таблицы по индексу [куда переход по
букве][буква]
				// номер состояния, откуда переход
```

```
transitionsTable[stateItr->second->transitions[(*alfabetItr)]][(*alfabetItr)].push_back(stateItr->f
irst);
            }
    }

}

bool hasAllTransitions(DFA &dfa, int alfabetPower) {
    int statesPower = dfa.states.size();
    int transitionsPower = 0;

    for (std::map<int, DFAState*>::iterator itr = dfa.states.begin(); itr != dfa.states.end(); ++itr)
            transitionsPower += itr->second->transitions.size();
    printf("number of transitions %d\n", transitionsPower);
    return statesPower * alfabetPower == transitionsPower;
}

DFA buildMinDFA(DFA &dfa, std::set<char> &alfabet, int numberOfStates,
std::vector<std::vector<int> > &partition,
    std::vector<int> &classAttachment) {
    DFA minDFA;
    int partitionPower = partition.size();

    for (int i = 0; i < partitionPower; i++) {
            std::set<int> fromStates;
            bool isFinal = false, isStart = false;
            std::map<char, int> transitions;
            DFAState *newState;
            for (int m = 0; m < partition[i].size(); m++) {
                    fromStates.insert(partition[i][m]);
                    isFinal = isFinal || dfa.states[partition[i][m]]->isFinal;
                    isStart = isStart || (dfa.start->state == dfa.states[partition[i][m]]->state);
                    for (std::set<char>::iterator itr = alfabet.begin(); itr != alfabet.end(); ++itr) {
                            transitions[(*itr)] =
classAttachment[dfa.states[partition[i][m]]->transitions[(*itr)]];
                    }

            }
            newState = createDFAState(i, fromStates, isFinal);
            newState->transitions = transitions;
            minDFA.addDFAState(newState, isStart);
    }
    return minDFA;
}
```

```
void makePartition(DFA &dfa, std::set<char> &alfabet, int numberOfStates,
   std::vector<std::vector<int> > &partition, std::vector<int> &classAttachment) {
   int maxLetterCode = findMax(alfabet);

   printf("\n");
   std::vector<std::pair<int, char> > queue; // очередь
   std::vector<std::vector<std::vector<int> > > transitionsTable(numberOfStates+1,
         std::vector<std::vector<int> >(maxLetterCode+1, std::vector<int>(0)));
   std::map<int, std::vector<int> > classConsistency; // какому номеру класса какие
состояния соответствуют

   //dfa.printDFA();
   // начальное разбиение: допускающие и недопускающие состояния, заполнение
вектора classAttachment
   initialPartition(dfa, partition, classAttachment);
   //printPartition(partition);

   // заполняем очередь парами: класс, буква алфавита
   for (int i = 0; i < partition.size(); i++) {
         for (std::set<char>::iterator alfabetItr = alfabet.begin(); alfabetItr != alfabet.end();
               ++alfabetItr) {
               queue.push_back(std::make_pair(i, (*alfabetItr)));
         }
   }
   //printQueue(queue);

   // заполняем обратную таблицу переходов
   makeBackTransitionsTable(dfa, transitionsTable, alfabet);
   printf("Back trasitions table\n");
   printTable(transitionsTable, alfabet, dfa);

   while (not(queue.empty())) {
         std::pair<int, char> splitter = queue.front();
         std::vector<int> splitterClass = partition[splitter.first];
         char splitterLetter = splitter.second;
         std::map<int, DFAState*> dfaStates = dfa.states;

         classConsistency.clear();
         queue.erase(queue.begin());
         // для каждого состояния из класса в сплиттере
         for (int i = 0; i < splitterClass.size(); i++) {
               // для каждого состояния автомата с ребром в сплиттер
               std::vector<int> statesToSplitter =
transitionsTable[splitterClass[i]][splitterLetter];
               int statesToSplitterPower = statesToSplitter.size();
```

```
			for (int r = 0; r < statesToSplitterPower; r++) {
				// из какого класса состояние с ребром в сплиттер?
				int fromClass =
classAttachment[transitionsTable[splitterClass[i]][splitterLetter][r]];
				if (classConsistency.find(fromClass) == classConsistency.end()) {
					std::vector<int> v;
					classConsistency[fromClass] = v;
				}

classConsistency[fromClass].push_back(transitionsTable[splitterClass[i]][splitterLetter][r]);
			}
		}

		// теперь обновить разбиение с учетом того, разделились ли состояния
		for (std::map<int, std::vector<int> >::iterator itr = classConsistency.begin();
			itr != classConsistency.end(); ++itr) {
			int fromClass = itr->first;
			// если не все состояния из класса переходят в сплиттер, то это
состояния надо разделить на два
			if (classConsistency[fromClass].size() < partition[fromClass].size()) {
				// добавляем пустое состояние в разбиение
				std::vector<int> v;
				partition.push_back(v);
				int newClassNumber = partition.size() - 1;
				// каждое состояния в выделяемом классе
				for (int i = 0; i < classConsistency[fromClass].size(); i++) {
					// удаляем из старого класса

partition[fromClass].erase(std::find(partition[fromClass].begin(),
						partition[fromClass].end(),
classConsistency[fromClass][i]));
					// добавляем в новый класс

partition[newClassNumber].push_back(classConsistency[fromClass][i]);
				}
				if (partition[newClassNumber] > partition[fromClass])
					std::swap(partition[fromClass], partition[newClassNumber]);
				// меняем номера класса в массиве
				for (int i = 0; i < partition[newClassNumber].size(); i++)
					classAttachment[partition[newClassNumber][i]] =
newClassNumber;
				// добавляем новые классы в очередь
				for (std::set<char>::iterator alfabetItr = alfabet.begin(); alfabetItr !=
alfabet.end();
					++alfabetItr)
```

```
                    queue.push_back(std::make_pair(newClassNumber,
(*alfabetItr)));
                }
            }
    }

    // если образовалось пустое состояние, удаляем его
    for (std::vector<std::vector<int> >::iterator itr = partition.begin(); itr != partition.end(); ++itr)
{
            if (itr->empty()) {
                    partition.erase(itr);
            }
    }

    //printPartition(partition);
}

DFA minimizeDFA(DFA &dfa, std::set<char> &alfabet) {
    std::vector<std::vector<int> > partition; // разбиение

    // добавляем состояние, в которое ведут ребра из всех вершин по всем символам
    // если количество переходов != количество состояний * мощность алфавита
    //dfa.printDFA();
    if (not(hasAllTransitions(dfa, alfabet.size()))) {
            addDeadState(dfa, alfabet);
    }
    dfa.printDFA();
    makeTransitionsTable(dfa,alfabet);
    int numberOfStates = dfa.states.size();
    std::vector<int> classAttachment(numberOfStates); // classAttachment[i] - какому классу
разбиения принадлежит состояние i
    makePartition(dfa, alfabet, numberOfStates, partition, classAttachment);

    DFA minimizedDFA = buildMinDFA(dfa, alfabet, numberOfStates, partition,
classAttachment);
    printf("min ");
    minimizedDFA.printDFA();
    makeTransitionsTable(minimizedDFA, alfabet);
    return minimizedDFA;
}

// STEP 6: string recognition

bool stringMatchesDFA(std::string word, DFA dfa, std::set<char> &alfabet) {
    int wordLength = word.size();
    printf("1\n");
```

```
    DFAState *currentState = dfa.start;
    std::cout << "-" << currentState->state << "-> ";
    for (int i = 0; i < wordLength; i++) {
        char c = word[i];
        if (alfabet.find(c) == alfabet.end()) {
            std::cout << "| fail.\nLetter " << c << " from your word is not in the alfabet.\n";
            return false;
        }
        currentState = dfa.states[currentState->transitions[c]];
        std::cout << word.substr(i, wordLength) << " -" << currentState->state << "-> ";
    }
    std::cout << (currentState->isFinal ? "| success" : "| fail") << std::endl;
    return currentState->isFinal;
}

int main() {
    std::string regex, procRegex;
    std::string postfix, word;
    std::set<char> alfabet;
    NFA nfa;
    DFA dfa;

    std::cout << "Enter your regex in infix form: ";
    std::cin >> regex;
    procRegex = regex;
    addConcatSymbol(procRegex);
    std::cout << "Postfix form of your regex: " << procRegex << std::endl;
    postfix = makePostfixForm(procRegex, alfabet);
    std::cout << postfix << std::endl;
    std::set<char>::iterator itr;
    std::cout << "alfabet: ";
    for (itr = alfabet.begin(); itr != alfabet.end(); itr++)
        std::cout << (*itr) << ' ';
    std::cout << std::endl;
    nfa = postfixToNFA(postfix);
    nfa.printNFA();
    dfa =  NFAtoDFA(nfa, alfabet);
    dfa.printDFA();
    //printf("qwerty\n");
    //makeTransitionsTable(dfa, alfabet);
    DFA minDFA = minimizeDFA(dfa, alfabet);
    std::cout << "\nYour regex is: " << regex;
    std::cout << "\nEnter the word to check if it matches your regular expression: ";
    std::cin >> word;
    std::cout << (stringMatchesDFA(word, minDFA, alfabet) ? "Given word matches your
regular expression\n" :
```

```
            "Given word doesn't match your regular expression\n");
}
```