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 (ИУ)»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии (ИУ7)»

ОТЧЕТ

Лабораторная работа №1

по курсу «Конструирование компиляторов»

на тему: «Распознавание цепочек регулярного языка»

Вариант № 4

Студент ИУ7-22М (Группа) _____ (Подпись, дата) И. А. Цветков (И. О. Фамилия)

Преподаватель _____ (Подпись, дата) А. А. Ступников (И. О. Фамилия)

2024 г.

1 Задание

1.1 Условие

Вариант 4: Напишите программу, которая в качестве входа принимает произвольное регулярное выражение, и выполняет следующие преобразования:

1. По регулярному выражению строит НКА.
2. По НКА строит эквивалентный ему ДКА.
3. По ДКА строит эквивалентный ему КА, имеющий наименьшее возможное количество состояний (Алгоритм Бржозовского).
4. Моделирует минимальный КА для входной цепочки из терминалов исходной грамматики.

2 Выполнение лабораторной работы

2.1 Тесты

Таблица 2.1 – Набор тестов и ожидаемые результаты работы программы

Регулярное выражение	**Входная цепочка**	**Ожидаемый результат**	**Результат**
a*	a	соответствует	соответствует
a*	b	не соответствует	не соответствует
a*	aaaa	соответствует	соответствует
a*	пустая	соответствует	соответствует
(ab*)+ab	aab	соответствует	соответствует
(ab*)+ab	abab	соответствует	соответствует
(ab*)+ab	aaab	соответствует	соответствует
(ab*)+ab	ab	не соответствует	не соответствует
(ab*)+ab	пустая	не соответствует	не соответствует
a(ab*)*	a	соответствует	соответствует
a(ab*)*	aab	соответствует	соответствует
a(ab*)*	ab	не соответствует	не соответствует
a(ab*)*	пустая	не соответствует	не соответствует

2.2 Результат работы программы

Результаты работы программы для регулярного выражения `a(ab*)*` приведены на рисунках 2.1–2.3.

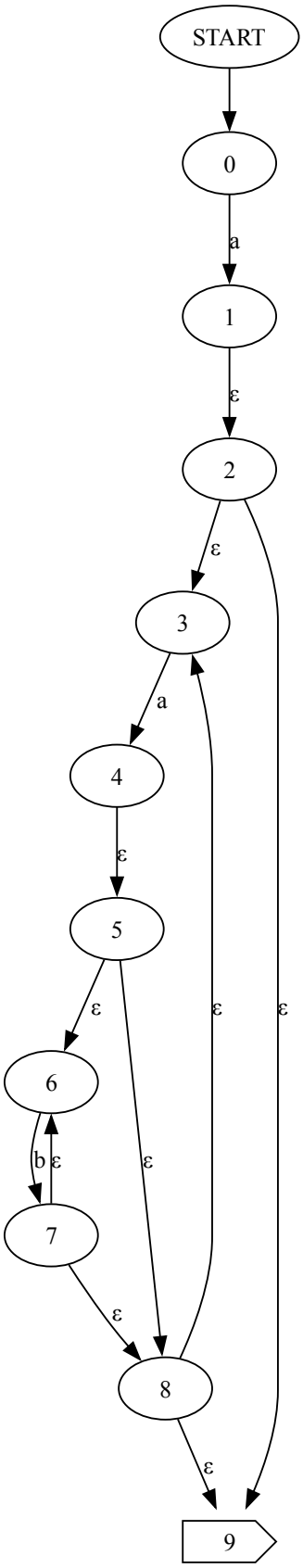


Рисунок 2.1 – НКА для регулярного выражения

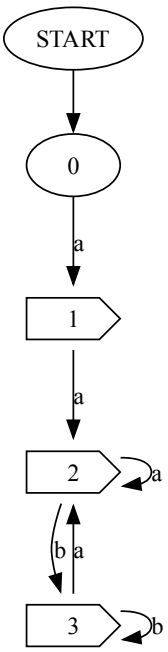


Рисунок 2.2 – ДКА для регулярного выражения

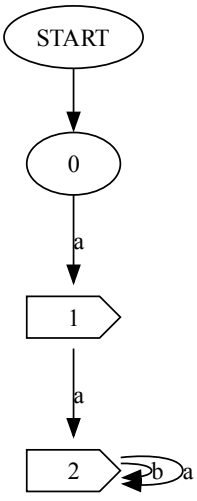


Рисунок 2.3 – Минимизированный ДКА алгоритмом Бржозовского

2.3 Контрольные вопросы

1. Какие из следующих множеств регулярны? Для тех, которые регулярны, напишите регулярные выражения.

   (a) Множество цепочек с равным числом нулей и единиц.

   Ответ: Не является регулярным множеством.

   (b) Множество цепочек из `{0, 1}*` с четным числом нулей и нечетным числом единиц.

   Ответ: Является регулярным множеством.

   Пример: ((0110)|(1001)|(1010)|(0101)|(11)|(00))*1
   ((0110)|(1001)|(1010)|(0101)|(11)|(00))*

   (c) Множество цепочек из `{0, 1}*`, длины которых делятся на 3.

   Ответ: Является регулярным множеством.

   Пример: ((0|1)(0|1)(0|1))*

   (d) Множество цепочек из `{0, 1}*`, не содержащих подцепочки 101.

   Ответ: Является регулярным множеством.

   Пример: ((0*00)|1)*

2. Найдите праволинейные грамматики для тех множеств из вопроса 1, которые регулярны.

(a)

$$
\begin{aligned}
S &\to 0110S \\
S &\to 1001S \\
S &\to 1010S \\
S &\to 0101S \\
S &\to 11S \\
S &\to 00S \\
S &\to 1A \\
A &\to 0110A \\
A &\to 1001A \\
A &\to 1010A \\
A &\to 0101A \\
A &\to 11A \\
A &\to 00A \\
A &\to \epsilon
\end{aligned}
\tag{2.1}
$$

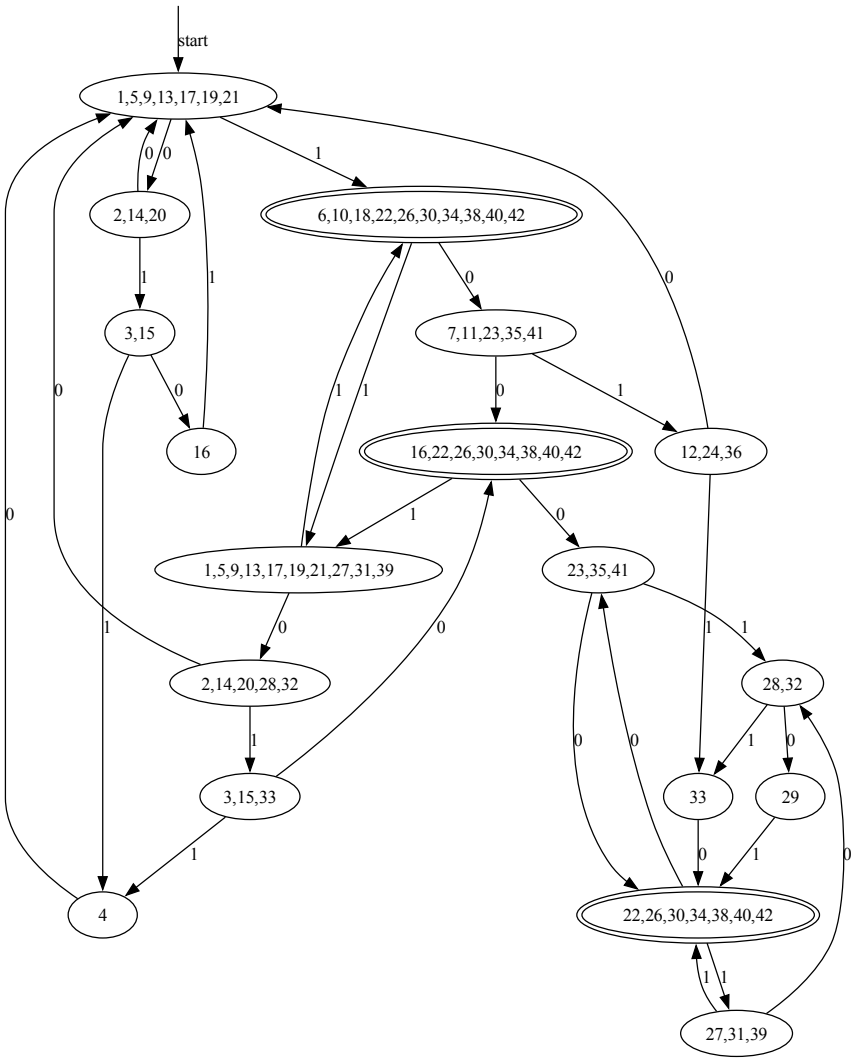
(b)

$$
\begin{aligned}
S &\to 0A \\
S &\to 1A \\
S &\to \epsilon \\
A &\to 0B \\
A &\to 1B \\
B &\to 0S \\
B &\to 1S
\end{aligned}
\tag{2.2}
$$

(c)

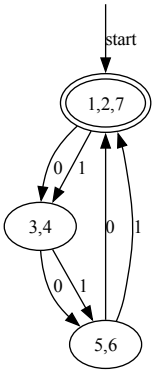
$$
\begin{aligned}
S &\to A \\
S &\to 1S \\
S &\to \epsilon \\
A &\to 0A \\
A &\to 00S
\end{aligned}
\tag{2.3}
$$

3. Найдите детерминированные и недетерминированные конечные автоматы для тех множеств из вопроса 1, которые регулярны.



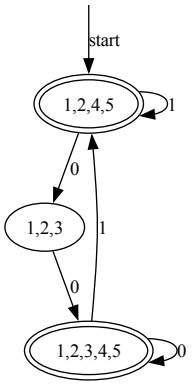
(a)

Рисунок 2.4 – ДКА для первого регулярного выражения



(b)

Рисунок 2.5 – ДКА для второго регулярного выражения



(c)

Рисунок 2.6 – ДКА для третьего регулярного выражения

4. Найдите конечный автомат с минимальным числом состояний для языка, определяемого автоматом M = ({A, B, C, D, E}, {0, 1}, d, A, {E, F}), где функция в задается таблицей

Состояние	Вход	
	0	1
A	B	C
B	E	F
C	A	A
D	F	E
E	D	F
F	D	E

Рисунок 2.7 – Таблица для 4 вопроса

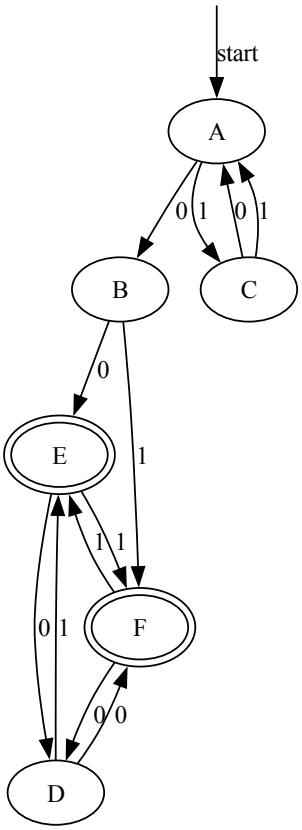


Рисунок 2.8 – ДКА для языка, определяемого автоматом M

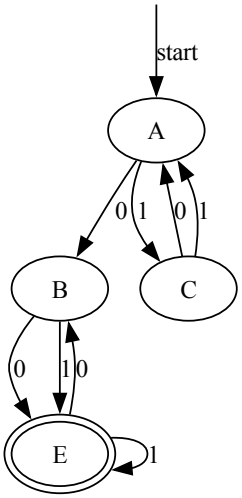


Рисунок 2.9 – Минимизированный ДКА для языка, определяемого автоматом M

2.4 Код прогаммы

В листингах 2.1–2.3 представлен код программы.

Листинг 2.1 — Основной модуль программы

```
from builder import *
import sys


def main():
    regexp = input("Введите регулярное выражение: ")
    # regexp = "a(ab*)*"
    # regexp = "(ab*)+ab"

    nfa = create_nfa(regexp)
    nfa.show_automata(1)

    dfa = convert_to_dfa(nfa)
    dfa.show_automata(2)

    mdfa = minimize_fda(dfa)
    mdfa.show_automata(3)

    while(True):
        check = input("Введите строку для моделирования МКА (для выхода введите
        ↪  'N'): ")
        if check.upper() == 'N':
            exit()
        else:
            mdfa.model_check(check)

if __name__ == "__main__":
    main()
```

Листинг 2.2 — Классы НКА и ДКА

```
import os
os.environ["PATH"] += os.pathsep + 'C:/Program Files/Graphviz/bin/'

from graphviz import Digraph
from typing import Dict, List, Set
import copy
import numpy as np

FDA_table = Dict[str, List[int]]
NFDA_table = Dict[str, List[List[int]]]
EPSILON = 'e'
```

Продолжение листинга 2.2

```
12
13
14  class CharCantBeAccepted(Exception):
15      pass
16
17
18  class Automata:
19      def accepts(self, input_string):
20          raise NotImplementedError()
21
22      def num_of_states(self):
23          raise NotImplementedError()
24
25      def getAlphabet(self):
26          raise NotImplementedError()
27
28      def show_automata(self):
29          raise NotImplementedError()
30
31
32  class NFA(Automata):
33      def __init__(self, table: NFDA_table, final_states):
34          self.table = table
35          self.final_states = final_states
36
37          self.states = None
38
39      def next_state(self, state, char):
40          if char not in self.table:
41              raise CharCantBeAccepted
42          return self.table[char][state]
43
44      def forward(self, old_state, char):
45          new_state = set()
46          for state in old_state:
47              new_state.update(self.next_state(state, char))
48              if EPSILON in self.table.keys():
49                  new_state.update(sum([self.eps_close(s) for s in new_state], []))
50          return list(new_state)
51
52      def add_transition(self, start, char, finish):
53          if char not in self.table:
54              self.table[char] = [[] for _ in range(self.num_of_states())]
55          self.table[char][start].append(finish)
56
```

Продолжение листинга 2.2

```
57      def accepts(self, input_string):
58          self.states = self.eps_close(0)
59          try:
60              for c in input_string:
61                  self.states = self.forward(self.states, c)
62              for state in self.states:
63                  if set(self.eps_close(state)).intersection(self.final_states):
64                      return True
65              return False
66          except CharCantBeAccepted:
67              return False
68
69      def num_of_states(self):
70          return len(list(self.table.values())[0])
71
72      def copy(self):
73          new_table = copy.deepcopy(self.table)
74          new_final = copy.deepcopy(self.final_states)
75          return NFA(new_table, new_final)
76
77      def show_automata(self, type):
78          dot = Digraph()
79
80          for char, state_list in self.table.items():
81              for i, s in enumerate(state_list):
82                  if len(s) != 0:
83                      for t in s:
84                          dot.edge(str(i), str(t), char)
85
86          for i in self.final_states:
87              dot.node(str(i),shape='cds')
88
89          dot.edge('START', '0')
90
91          if type == 1:
92              dot.render('НКА', view=True)
93          if type == 2:
94              dot.render('ДКА', view=True)
95          if type == 3:
96              dot.render('МДКА', view=True)
97
98
99      def eps_close(self, state: int) -> List[int]:
100         if EPSILON not in self.table.keys():
101             return [state]
```

Продолжение листинга 2.2

```
102             visited = []
103             active = [state]
104             while len(active) != 0:
105                 new_active = []
106                 for s in active:
107                     new_active.extend(self.table[EPSILON][s])
108                 visited = list(set(visited + active))
109                 active = list(set(new_active).difference(visited))
110             return visited
111
112         def getAlphabet(self):
113             return list(self.table.keys())
114
115     class DFA(Automata):
116         def __init__(self, table: FDA_table, final_states: List[int]):
117             # Q = states
118             # A = alhabet
119             # T = table
120             # S = start_states
121             # F = final_state
122             self.table = table
123             self.final_states = final_states
124             self.proxy = NFA(self.__getNFATableFromDFATable(), final_states)
125
126             num_of_states = self.num_of_states()
127             self.states = [i for i in range(num_of_states)]
128             self.start_states = self.__getStartStates(self.states[:])
129             self.alphabet = self.getAlphabet()
130
131         def __getNFATableFromDFATable(self):
132             proxy_table = {}
133
134             for char, states in self.table.items():
135                 proxy_table[char] = [[state] if state is not None else [] for state in
                    ↪  states]
136
137             return proxy_table
138
139         def __getDFATableFromNFATable(self):
140             table = {char:[] for char in self.alphabet}
141             isStop = False
142
143             for char, states in self.proxy.table.items():
144                 if isStop:
145                     break
```

Продолжение листинга 2.2

```
146
147             for state in states:
148                 statesNum = len(state)
149                 if statesNum == 0:
150                     table[char].append(None)
151                 elif statesNum == 1:
152                     table[char].append(state[0])
153                 else:
154                     # print("ERROR: __getDFATableFromNFATable:", self.proxy.table)
155                     table = {}
156                     isStop = not isStop
157                     break
158
159         return table
160
161     def accepts(self, input_string: str) -> bool:
162         return self.proxy.accepts(input_string)
163
164     def show_automata(self, type):
165         self.proxy.show_automata(type)
166
167     def num_of_states(self):
168         return self.proxy.num_of_states()
169
170     def getAlphabet(self):
171         return self.proxy.getAlphabet()
172
173     def revertDFA(self) -> None:
174         final_state_tmp = self.final_states[:]
175         self.final_states = self.start_states
176         self.start_states = final_state_tmp
177
178         new_table = {char: [[] for _ in range(len(self.states))] for char in
            ↪ self.alphabet}
179
180         for char, state_list in self.proxy.table.items():
181             for start_state, char_states in enumerate(state_list):
182                 if len(char_states) != 0:
183                     for end_state in char_states:
184                         new_table[char][end_state].append(start_state)
185
186         self.proxy.table = new_table
187         self.table = self.__getDFATableFromNFATable()
188
189     def detDFA(self) -> None:
```

Продолжение листинга 2.2

```
190             def reachable(q, state):
191                 t = dict()
192                 for char in self.alphabet:
193                     ts = set()
194                     for i in q[state]:
195                         ts |= set(self.proxy.table[char][i])
196                     if not ts:
197                         t[char] = []
198                         continue
199                     try:
200                         i = q.index(ts)
201                     except ValueError:
202                         i = len(q)
203                         q.append(ts)
204                     t[char] = [i]
205                 return t
206 
207             q = [set(self.start_states)]
208             new_table = {char: [] for char in self.alphabet}
209 
210             while len(list(new_table.values())[0]) < len(q):
211                 tmp = reachable(q, len(list(new_table.values())[0])) # -> {a: [[]], b:
                    ↪  [[]]}
212                 for char in self.alphabet:
213                     new_table[char].append(tmp[char])
214 
215             self.start_states = [0]
216             self.states = [i for i in range(0, len(q))]
217             self.final_states = [q.index(i) for i in q if set(self.final_states) & i]
218             self.proxy.table = new_table
219             self.table = self.__getDFATableFromNFATable()
220 
221         def model_check(self, check_str):
222             check_arr = [*check_str]
223             size = len(self.table[list(self.table.keys())[0]])
224             Ssize = len(self.table)
225             true_table = np.full((size,size,Ssize), None)
226             j = 0
227             for char, state_list in self.table.items():
228                 for i, s in enumerate(state_list):
229                     if s != None:
230                         true_table[i][s][j] = char
231                 j += 1
232 
233             carette = 0
```

Продолжение листинга 2.2

```
234
235         while(True):
236             if not check_arr:
237                 break
238             for i in range(size):
239                 if check_arr:
240                     arr = []
241                     for a in true_table[carette]:
242                         for b in a:
243                             arr.append(b)
244                     if check_arr[0] not in arr:
245                         print('ERROR')
246                         return
247                     for symbol in true_table[carette][i]:
248                         if check_arr[0] == symbol:
249                             check_arr.pop(0)
250                             carette = i
251                             break
252
253         if not check_arr and carette in self.final_states:
254             print('OK')
255         else:
256             print('ERROR')
257
258         return
259
260     def __getStartStates(self, states: List[int]) -> List[int]:
261         for _, state_list in self.proxy.table.items():
262             for start_state, char_states in enumerate(state_list):
263                 if len(char_states) != 0:
264                     for end_state in char_states:
265                         if start_state != end_state and end_state in states:
266                             states.remove(end_state)
267         return states
```

Листинг 2.3 — Функции перевода из автомата в автомат

```
1  import copy
2  from automata import NFA, DFA, EPSILON
3  from typing import Dict, List, Tuple
4
5
6  def merge_tables(A, B):
7      keys = set(list(A.table.keys()) + list(B.table.keys()))
8      new_final = [state + A.num_of_states() for state in B.final_states]
```

Продолжение листинга 2.3

```
9       new_final.extend(A.final_states)
10      new_table = {}
11      for k in keys:
12          new_row = []
13          if k in A.table:
14              new_row.extend(A.table[k])
15          else:
16              new_row.extend([[] for _ in range(A.num_of_states())])
17          if k in B.table:
18              new_row.extend([[s + A.num_of_states() for s in states] for states in
                ↪  B.table[k]])
19          else:
20              new_row.extend([[] for _ in range(B.num_of_states())])
21          new_table[k] = new_row
22      return NFA(table=new_table, final_states=new_final)
23
24
25  def concatenate(A, B):
26      merged = merge_tables(A, B)
27      for start in A.final_states:
28          merged.add_transition(start, EPSILON, A.num_of_states())
29      merged.final_states = [s + A.num_of_states() for s in B.final_states]
30      return merged
31
32
33  def alternate(A, B):
34      merged = merge_tables(A, B)
35      shifted_finals = [f + 1 for f in merged.final_states]
36      shifted_table = {}
37      for char, state_list in merged.table.items():
38          shifted_table[char] = [[]] + [[state + 1 for state in states] for states
            ↪  in state_list] + [[]]
39      new = NFA(table=shifted_table, final_states=[])
40      new.add_transition(0, EPSILON, 1)
41      new.add_transition(0, EPSILON, A.num_of_states() + 1)
42      for f in shifted_finals:
43          new.add_transition(f, EPSILON, new.num_of_states() - 1)
44      new.final_states = [new.num_of_states() - 1]
45      return new
46
47
48  def star(A):
49      shifted_finals = [f + 1 for f in A.final_states]
50      shifted_table = {}
51      for char, state_list in A.table.items():
```

Продолжение листинга 2.3

```
            shifted_table[char] = [[]] + [[state + 1 for state in states] for states
            ↪  in state_list] + [[]]
        new = NFA(table=shifted_table, final_states=[])
        for f in shifted_finals:
            new.add_transition(f, EPSILON, 1)
            new.add_transition(f, EPSILON, new.num_of_states() - 1)
        new.add_transition(0, EPSILON, 1)
        new.add_transition(0, EPSILON, new.num_of_states() - 1)
        new.final_states = [new.num_of_states() - 1]
        return new


    def plus(A):
        return concatenate(A, star(A))


    def primitive_nfda(actual_string):
        table: Dict[str, List[List[int]]] = {}
        for i, c in enumerate(actual_string):
            if c not in table:
                table[c] = [[] for _ in range(len(actual_string) + 1)]
            table[c][i].append(i + 1)
        return NFA(table=table, final_states=[len(actual_string)])


    operations = {
        '*': star,
        '+': plus,
        '|': alternate,
        ',': concatenate
    }
    priorities = {
        '|': 0,
        ',': 1,
        '*': 2,
        '+': 2
    }

    binary = ['|', ',']
    unary = ['*', '+', '?']


    def is_character(c):
        return c not in (list(operations.keys()) + ['(', ')'])

```

Продолжение листинга 2.3

```
96
97  def prepare_regexp(regexp):
98      if len(regexp) == 0:
99          return ''
100     new = []
101     last = None
102     for c in regexp:
103         if last is None:
104             last = c
105             new.append(c)
106             continue
107         if last in unary and c == '(' \
108             or last in unary and is_character(c) \
109             or is_character(last) and is_character(c) \
110             or last == ')' and is_character(c) \
111             or is_character(last) and c == '(':
112             new.append(',')
113         new.append(c)
114         last = c
115     return ''.join(new)
116
117
118 def create_nfa(regexp):
119     if not regexp:
120         raise ValueError("Error: Empty regexp")
121
122     op_stack = []
123     automata_stack = []
124     buffer = ''
125
126     def avalanche(priority=-1):
127         while len(op_stack) != 0 \
128                 and op_stack[-1] != '(' \
129                 and (op_stack[-1] not in operations.keys() or
                    ↪  priorities[op_stack[-1]] > priority):
130             op = op_stack[-1]
131             if op in binary:
132                 automata_stack.append(operations[op](automata_stack[-2],
                    ↪  automata_stack[-1]))
133                 automata_stack.pop(-2)
134                 automata_stack.pop(-2)
135                 op_stack.pop()
136             elif op in unary:
137                 automata_stack.append(operations[op](automata_stack[-1]))
138                 automata_stack.pop(-2)
```

Продолжение листинга 2.3

```
                op_stack.pop()
        if priority == -1 and len(op_stack) != 0 and op_stack[-1] == '(':
            op_stack.pop()

    regexp = prepare_regexp(regexp)

    for c in regexp:
        if c in list(operations.keys()) + ['(', ')']:
            if buffer != '':
                automata_stack.append(primitive_nfda(buffer))
            buffer = ''
        if c in operations:
            if len(op_stack) == 0 or op_stack[-1] in ['(', ')'] or
            ↪ priorities[op_stack[-1]] < priorities[c]:
                op_stack.append(c)
            else:
                avalanche(priorities[c])
                op_stack.append(c)
        elif c == '(':
            op_stack.append('(')
        elif c == ')':
            avalanche()
        else:
            buffer += c

    if buffer != '':
        automata_stack.append(primitive_nfda(buffer))
    avalanche()

    return automata_stack[-1]


def convert_to_dfa(nfda):
    links = []
    newStates = [set(nfda.eps_close(0))]
    visitedStates = []
    alphabet = [x for x in list(nfda.table.keys()) if x != EPSILON]
    while len(newStates) > 0:
        tmp = newStates.pop()
        if tmp in visitedStates:
            continue
        visitedStates.append(tmp)
        for char in alphabet:
            newTmp = set(nfda.forward(tmp, char))
            if len(newTmp) != 0:
```

Продолжение листинга 2.3

```
183                     newStates.append(newTmp)
184                     links.append((tmp, char, newTmp))
185         formatted_links = []
186         for link in links:
187             formatted_links.append((visitedStates.index(link[0]), link[1],
                ↪  visitedStates.index(link[2])))
188         links = formatted_links
189         old_final = set(nfda.final_states)
190         new_final = [i for i, s in enumerate(visitedStates) if
            ↪  s.intersection(old_final)]
191         new_table = {}
192         for k in alphabet:
193             new_table[k] = [None for _ in enumerate(visitedStates)]
194         for link in links:
195             new_table[link[1]][link[0]] = link[2]
196         return DFA(table=new_table, final_states=new_final)
197
198
199
200 class DfaStandart: # ранее рассмотренное представление не подходит для
    ↪  использования алгоритма Бржозовского
201     def __init__(self) -> None:
202         self.Q = [] # состояния
203         self.A = [] # алфавит
204         self.T = [] # функции переходов -> состояние: в какие переходит и по какой
            ↪  букве
205         self.S = [] # начальные состояния
206         self.F = [] # конечные состояния
207
208     def convertFromDFA(self, dfa: DFA) -> None:
209         if dfa.table:
210             self.A = dfa.alphabet()
211             self.F = dfa.final_states
212
213             num_of_states = dfa.num_of_states()
214             self.Q = [i for i in range(num_of_states)]
215
216             self.S = self.__getStartStatesFromDFA(dfa, self.Q[:])
217             self.T = copy.deepcopy(dfa.proxy.table)
218
219     def revertDFA(self) -> None:
220         final_state_tmp = self.F[:]
221         self.F = self.S
222         self.S = final_state_tmp
223
```

Продолжение листинга 2.3

```
224             new_table = {char: [[] for _ in range(len(self.Q))] for char in self.A}
225
226             for char, state_list in self.T.items():
227                 for start_state, char_states in enumerate(state_list):
228                     if len(char_states) != 0:
229                         for end_state in char_states:
230                             new_table[char][end_state].append(start_state)
231
232             self.T = new_table
233
234         def detDFA(self) -> None:
235             def reachable(q, state):
236                 t = dict()
237                 for char in self.A:
238                     ts = set()
239                     for i in q[state]:
240                         ts |= set(self.T[char][i])
241                     if not ts:
242                         t[char] = []
243                         continue
244                     try:
245                         i = q.index(ts)
246                     except ValueError:
247                         i = len(q)
248                         q.append(ts)
249                     t[char] = [i]
250                 return t
251
252             q = [set(self.S)]
253             new_table = {char: [] for char in self.A}
254
255             while len(list(new_table.values())[0]) < len(q):
256                 tmp = reachable(q, len(list(new_table.values())[0])) # -> {a: [[]], b:
                    ↪ [[]]}
257                 for char in self.A:
258                     new_table[char].append(tmp[char])
259
260             self.S = [0]
261             self.Q = [i for i in range(0, len(q))]
262             self.F = [q.index(i) for i in q if set(self.F) & i]
263             self.T = new_table
264
265
266         def __getStartStatesFromDFA(self, dfa: DFA, Q: List[int]) -> List[int]:
267             for _, state_list in dfa.proxy.table.items():
```

Продолжение листинга 2.3

```
268                 for start_state, char_states in enumerate(state_list):
269                     if len(char_states) != 0:
270                         for end_state in char_states:
271                             if start_state != end_state and end_state in Q:
272                                 Q.remove(end_state)
273             return Q
274 
275 
276     def min_brzhzovskiy(fda: DFA):
277         fda.revertDFA()
278         fda.detDFA()
279         fda.revertDFA()
280         fda.detDFA()
281         return fda
282 
283 
284     def minimize_fda(dfa: DFA):
285         dfa = min_brzhzovskiy(dfa)
286         return DFA(table=dfa.table, final_states=dfa.final_states)
287 
288         def split_set(target, splitter, split_char):
289             R1 = set()
290             R2 = set()
291             for v in target:
292                 if fda.table[split_char][v] in splitter:
293                     R1.add(v)
294                 else:
295                     R2.add(v)
296             return R1, R2
297 
298         sets = [{*fda.final_states}]
299         non_final = {*list(range(fda.num_of_states()))}.difference(fda.final_states)
300         if len(non_final) > 0:
301             sets.append(non_final)
302         queue = []
303         for c in fda.getAlphabet():
304             for s in sets:
305                 queue.append((s, c))
306         while len(queue) > 0:
307             splitter, char = queue.pop(0)
308             for s in sets:
309                 R1, R2 = split_set(s, splitter, char)
310                 if len(R1) > 0 and len(R2) > 0:
311                     sets.remove(s)
312                     sets.extend([R1, R2])
```

Продолжение листинга 2.3

```
313                     if (s, char) in queue:
314                         queue.remove((s, char))
315                         queue.append((R1, char))
316                         queue.append((R2, char))
317                     else:
318                         if len(R1) < len(R2):
319                             queue.append((R1, char))
320                         else:
321                             queue.append((R2, char))
322
323         first_state_index = [sets.index(s) for s in sets if 0 in s][0]
324         first_state = sets.pop(first_state_index)
325         sets.insert(0, first_state)
326
327         num_of_states = len(sets)
328         new_table = {k: [None] * num_of_states for k in fda.getAlphabet()}
329         for i, s in enumerate(sets):
330             for v in s:
331                 for c in fda.getAlphabet():
332                     new_indexes = [sets.index(s) for s in sets if fda.table[c][v] in
                        ↪ s]
333                     new_table[c][i] = None if len(new_indexes) == 0 else
                        ↪ new_indexes[0]
334         new_final = [sets.index(s) for s in sets if s.intersection(fda.final_states)]
335         return DFA(table=new_table, final_states=new_final)
```