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»

Отчет по лабораторной работе №1

Название Преобразование грамматик

Дисциплина Конструирование компиляторов

Студент Шацкий Н. С.

Группа ИУ7-21М

Преподаватель Ступников А. А.

Москва — 2024 г.

1 Задание

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

1. Преобразует регулярное выражение непосредственно в ДКА.
2. По ДКА строит эквивалентный ему КА, имеющий наименьшее возможное количество состояний.
3. Моделирует минимальный КА для входной цепочки из терминалов исходной грамматики (воспользоваться алгоритмом минимизации ДКА Бржозовского).

2 Результаты работы программы

Результаты работы программы для регулярного выражения `(a|b)*aabb` приведены на рисунках 1 – 4.

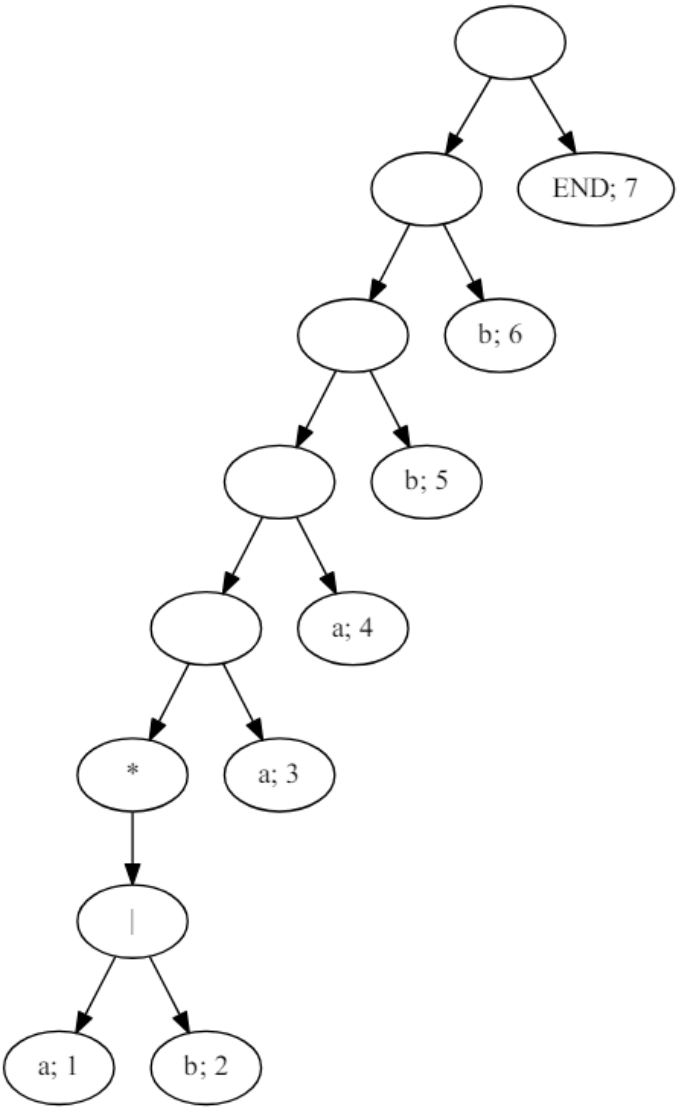

Рисунок 1 – Дерево синтаксического разбора

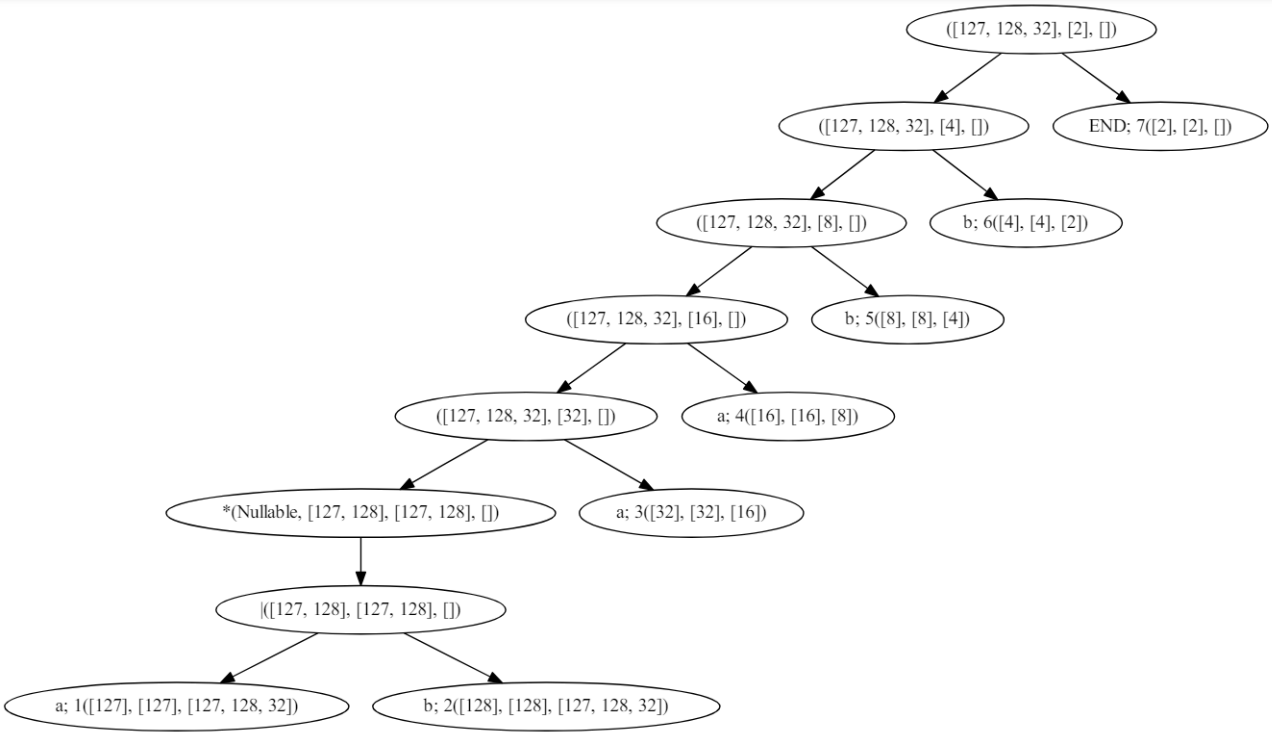

Рисунок 2 – Дерево синтаксического разбора с функциями firstpos, lastpos, followpos

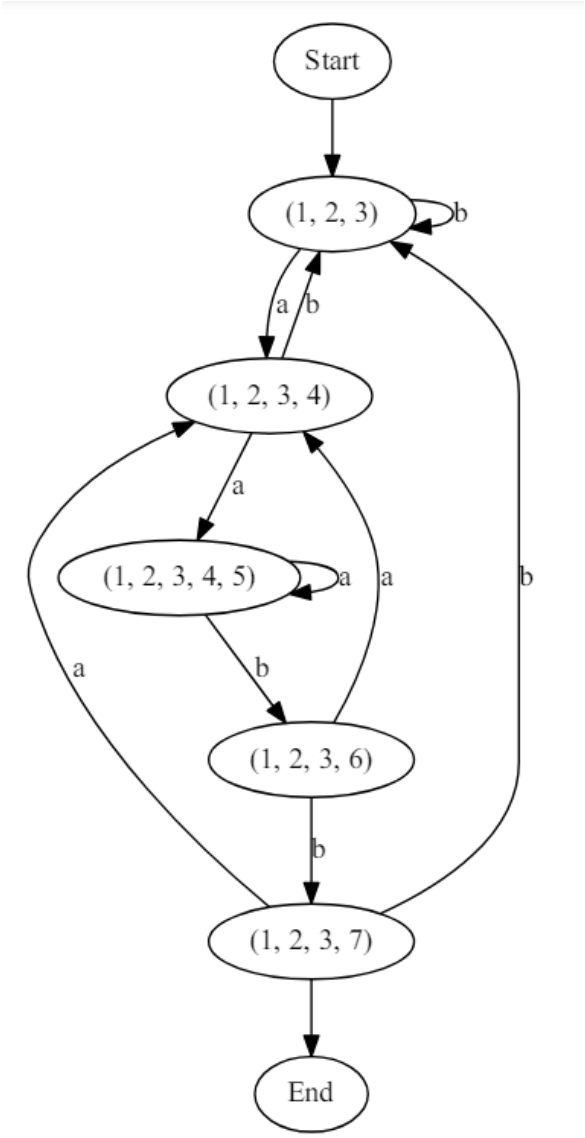

Рисунок 3 – ДКА для регулярного выражения

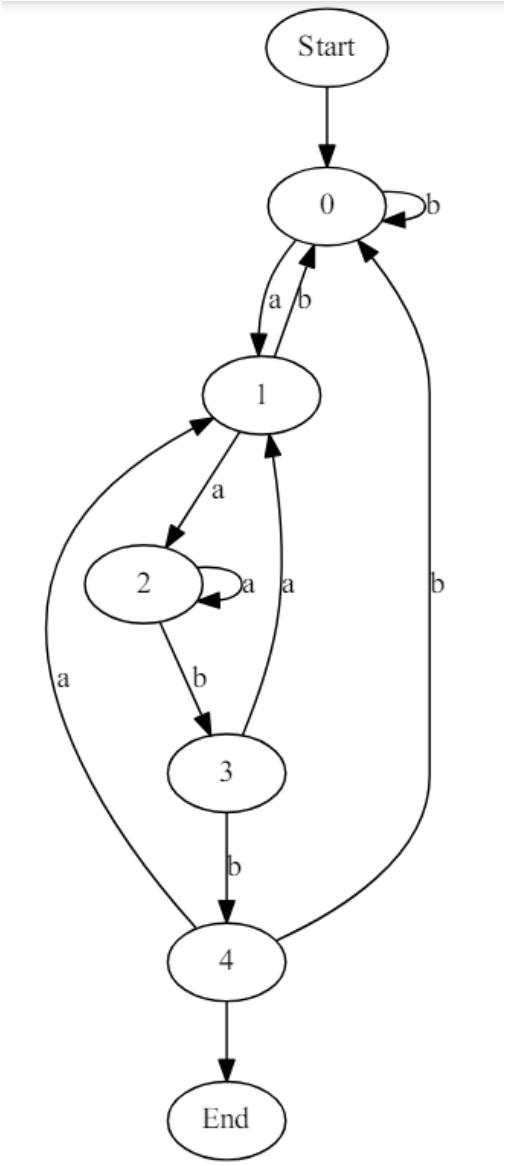

Рисунок 4 – МДКА алгоритмом Бржозовского

3 Минимизация ДКА

Алгоритм Томпсона [1] строит по НКА эквивалентный ДКА следующим образом:

Начало.

Шаг 1. Помещаем в очередь Q множество, состоящее только из стартовой вершины.

Шаг 2. Затем, пока очередь не пуста выполняем следующие действия:

- Достаем из очереди множество, назовем его q;

- Для всех $c \in \Sigma$ посмотрим в какое состояние ведет переход по символу c из каждого состояния в q;

- Полученное множество состояний положим в очередь Q только если оно не лежало там раньше. Каждое такое множество в итоговом ДКА будет отдельной вершиной, в которую будут вести переходы по соответствующим символам.

- Если в множестве q хотя бы одна из вершин была терминальной в НКА, то соответствующая данному множеству вершина в ДКА также будет терминальной.

Конец.

На рисунках 5 – 8 представлен результат работы алгоритмы Бржозовского для минимизации ДКА по шагам.

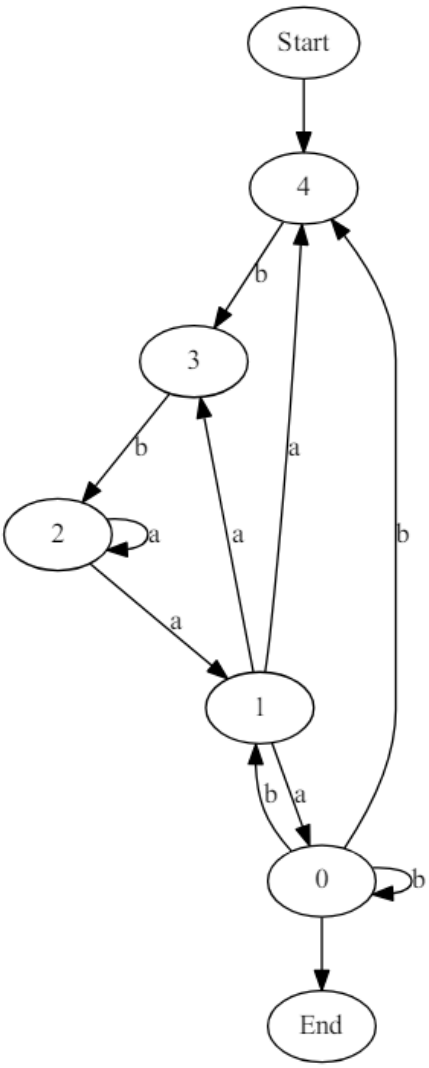

Рисунок 5 – Разворот КА (шаг 1)

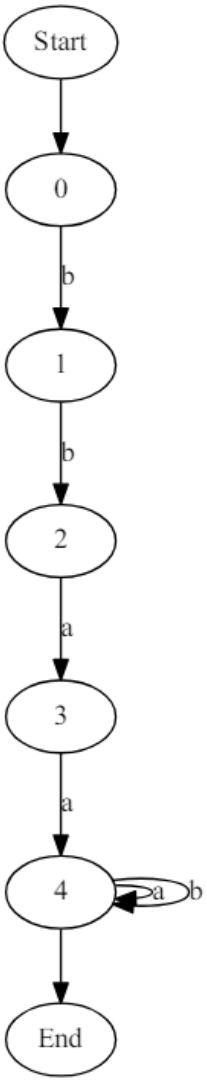

Рисунок 6 – Детерминированный КА (шаг 2)

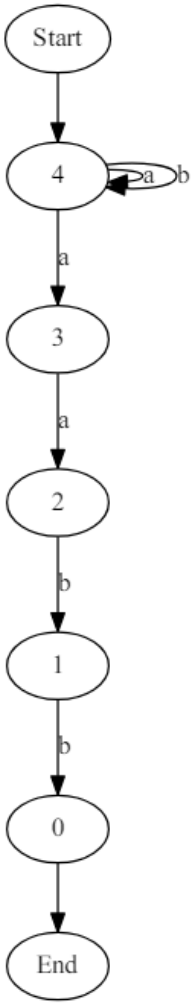

Рисунок 7 – Разворот КА (шаг 3)

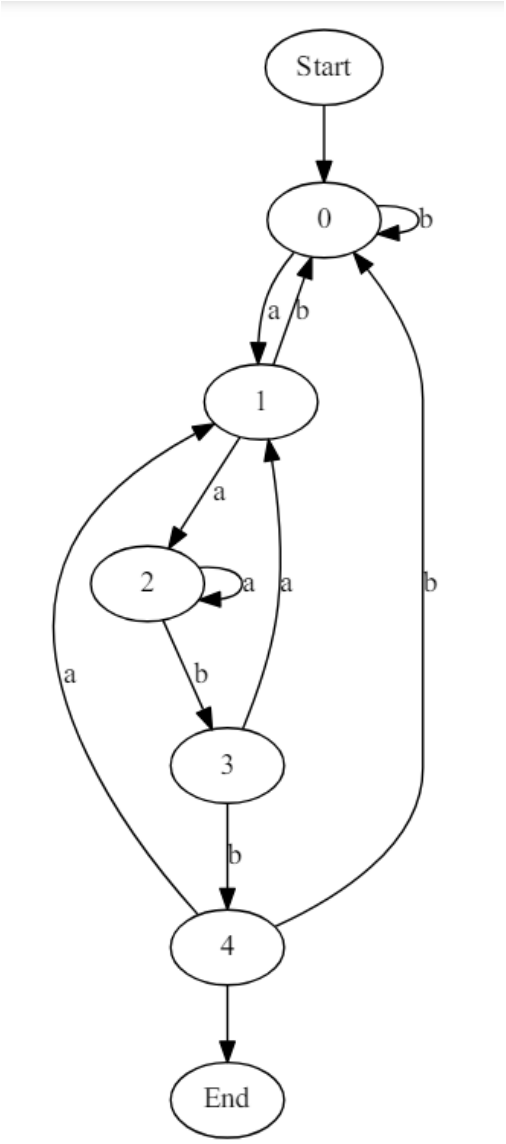

Рисунок 8 – Детерминированный КА (шаг 4 – заключительный)

4 Набор тестов

Таблица 1 – Набор тестов и ожидаемые результаты работы программы

Регулярное выражение	**Входная цепочка**	**Ожидаемый результат**	**Результат**
a*	a	валидно	валидно
a*	aaa	валидно	валидно
a*	b	не валидно	не валидно
a*	пустая	валидно	валидно
(a|b)*abb	abb	валидно	валидно
(a|b)*abb	aaabb	валидно	валидно
(a|b)*abb	babaabb	валидно	валидно
(a|b)*abb	ababbb	не валидно	не валидно
(a|b)*abb	пустая	не валидно	не валидно
((aa)|(bb)|c)*	aabb	валидно	валидно
((aa)|(bb)|c)*	bbcccbbc	валидно	валидно
((aa)|(bb)|c)*	aacab	не валидно	не валидно
((aa)|(bb)|c)*	пустая	валидно	валидно

5 Листинг программы

В листинге 5 приведена реализация программы на ЯП Python.

```
0 from graphviz import Digraph
1
2
3 class Node:
4     def __init__(self, v = None, l = None, r = None):
5         self.left = l
6         self.right = r
7         self.value = v
8         self.followpos = []
```

```
9           self.label_i = ''
10      def copy(cur):
11          return Node(cur.value, cur.left, cur.right)
12      left = None
13      right = None
14      value = None
15
16      def show_tree(pt, dot = None, show_params=False):
17          if not dot:
18              dot = Digraph()
19          label = str(pt.value) + ("; " + str(pt.label_i) if pt.label_i else
        "")
20          def nodes_to_i(nodes):
21              return [node.i for node in nodes]
22          if show_params:
23              label = label + "(" + ("Nullable, " if pt.nullable else "") +
    str(nodes_to_i(pt.firstpos)) + ", " + str(
24                  nodes_to_i(pt.lastpos)) + ", " + str(nodes_to_i(pt.
    followpos)) + ")"
25          dot.node(str(pt.i), label)
26          if pt.left:
27              dot.edge(str(pt.i), str(pt.left.i))
28              pt.left.show_tree(dot, show_params=show_params)
29          if pt.right:
30              dot.edge(str(pt.i), str(pt.right.i))
31              pt.right.show_tree(dot, show_params=show_params)
32          return dot
33
34      def count_i(self, i = 0):
35          self.i = i
36          if self.left:
37              self.left.count_i(i*2+1)
38          if self.right:
39              self.right.count_i(i*2+2)
40
41      def count_nullable(self):
42          if self.left:
43              self.left.count_nullable()
44          if self.right:
45              self.right.count_nullable()
```

```
46
47          if self.value == '*':
48              self.nullable = True
49          elif self.value== '|':
50              self.nullable = self.left.nullable or self.right.nullable
51          else:
52              self.nullable = False
53          return self.nullable
54
55      def count_firstpos(self):
56          if self.left:
57              self.left.count_firstpos()
58          if self.right:
59              self.right.count_firstpos()
60
61          if self.value in ['*', '+']:
62              self.firstpos = self.left.firstpos.copy()
63          elif self.value == '':
64              self.firstpos = self.left.firstpos.copy()
65              if self.left.nullable:
66                  self.firstpos += self.right.firstpos
67          elif self.value== '|':
68              self.firstpos = self.left.firstpos + self.right.firstpos
69          else:
70              self.firstpos = [self]
71          return self.firstpos
72
73      def count_lastpos(self):
74          if self.left:
75              self.left.count_lastpos()
76          if self.right:
77              self.right.count_lastpos()
78
79          if self.value in ['*', '+']:
80              self.lastpos = self.left.lastpos.copy()
81          elif self.value == '':
82              self.lastpos = self.right.lastpos.copy()
83              if self.right.nullable:
84                  self.lastpos += self.left.lastpos
85          elif self.value== '|':
```

```
86              self.lastpos = self.left.lastpos + self.right.lastpos
87          else:
88              self.lastpos = [self]
89          return self.lastpos
90
91      def count_followpos(self):
92          if self.left:
93              self.left.count_followpos()
94          if self.right:
95              self.right.count_followpos()
96          if self.value == '':
97              for i in self.left.lastpos:
98                  i.followpos += self.right.firstpos
99          elif self.value == '*':
100             for i in self.lastpos:
101                 i.followpos += self.firstpos
102
103     def foreach_deep(self, funcIn = None, FuncOut = None):
104         if funcIn:
105             funcIn(self)
106         if self.left:
107             self.left.foreach_deep(funcIn, FuncOut)
108         if self.right:
109             self.right.foreach_deep(funcIn, FuncOut)
110         if FuncOut:
111             FuncOut(self)
112
113
114 def buildParseTree(INITexpr):
115     def get_expr(exp):
116         def cat_exprs(left, right):
117             return Node('', left, right)
118         s = 0
119         tree = None
120         if exp[s] == '(':
121             s += 1
122             e, n = get_expr(exp[s:])
123             s += n
124             tree = e
125             while exp[s] != ')':
```

```
126                 e, n = get_expr(exp[s:])
127                 s += n
128                 tree = cat_exprs(tree, e)
129         elif exp[s] not in ['+', '*', '|', ')']:
130             tree = Node(exp[s])
131         else:
132             raise Exception('Waiting for "(", term: ' +  "".join(exp) + '
    '+ str(s))
133         s += 1
134         if s < len(exp)and exp[s] in ['+', '*']:
135                 tree = Node(exp[s], tree)
136                 s+=1
137 
138         if s < len(exp) and exp[s] == '|':
139                 next_expr, l = get_expr(exp[s+1:])
140                 return Node('|', tree, next_expr), s + 1 + l
141 
142         return tree, s
143 
144     x, r =  get_expr(['('] + INITexpr + [')'])
145     return x
146 
147 
148 
149 
150 def make_ka(root: Node, symbols, end_symb='END'):
151     def  make_state(nodes):
152         return tuple(sorted([node.label_i for node in nodes]))
153 
154     def is_end(nodes):
155         for node in nodes:
156             if node.value == end_symb:
157                 return True
158         return False
159     Dstates = []
160 
161     table = {}
162     queue = [root.firstpos]
163 
164     last = []
```

```
165     while len(queue):
166         state = queue.pop()
167         Dstates += [make_state(state)]
168         for c in symbols:
169             U = []
170             for p in state:
171                 if p.value == c:
172                     U += p.followpos
173             U = list(set(U))
174             if not len(U):
175                 continue
176 
177             if make_state(U) not in Dstates:
178                 queue.append(U)
179 
180 
181 
182             table[make_state(state), c] = [make_state(U)]
183 
184             #print(make_state(state), c, [make_state(U)])
185 
186         if is_end(state):
187                 last += [make_state(state)]
188 
189 
190     return [Dstates[0]], last, table
191 
192 
193 
194 
195 def fix_names(first, last, table):
196     names = []
197     for f,v in table:
198         if f not in names:
199             names += [f]
200         for t in table[f,v]:
201             if not t in names:
202                 names += [t]
203     newTable = {}
204     for f,v in table:
```

```
205         newTable[(names.index(f), v)]  = [names.index(t) for t in table[f,
    v]]
206
207     return [names.index(t) for t in first], [names.index(t) for t in last
    ], newTable
208
209 def print_ka(first, end, Dtran):
210     g = Digraph()
211
212     g.node('S', 'Start')
213     g.node('E', 'End')
214
215     for i in first:
216         g.edge('S', str(i))
217
218     for i in end:
219         g.edge(str(i), 'E')
220
221
222     for f,v in Dtran:
223         for t in Dtran[(f,v)]:
224             g.node(str(f))
225             g.node(str(t))
226             g.edge(str(f), str(t), v )
227
228     return g
229
230
231
232
233
234 def reverseKa(first, last, Dtran):
235     newDtran = {}
236     for f,v in Dtran:
237         for t in Dtran[(f,v)]:
238             if (t,v) not in newDtran:
239                 newDtran[(t, v)] = []
240             newDtran[(t, v)] += [f]
241     return last, first, newDtran
242
```

```
243
244 def toDFA(first, last, table):
245     """
246     Алгоритм Томпсона строит по НКА эквивалентный ДКА следующим образом:
247
248     Начало.
249     Шаг 1. Помещаем в очередь Q
250         множество, состоящее только из стартовой вершины.
251     Шаг 2. Затем, пока очередь не пуста выполняем следующие действия:
252         Достаем из очереди множество, назовем его q
253         Для всех c
254         посмотрим в какое состояние ведет переход по символу c из каждого состояния
    в q. Полученное множество состояний положим в очередь Q
255         только если оно не лежало там раньше. Каждое такое множество в итоговом ДКА
    будет отдельной вершиной, в которую будут вести переходы по соответствующим
    символам.
256         Если в множестве q хотя бы одна из вершин была терминальной в НКА, то
    соответствующая данному множеству вершина в ДКА также будет терминальной.
257
258     Конец.
259     """
260     def nodes_to_state_name(l):
261         return tuple(set(l))
262
263     symbols = []
264     for _, v in table:
265         symbols += [v]
266     symbols = list(set(symbols))
267
268     queue = [first.copy()]
269
270     newTable = {}
271     states = []
272
273     while len(queue):
274         nodes = queue.pop()
275         states.append(nodes_to_state_name(nodes))
276         for c in symbols:
277             state = []
278             for node in nodes:
```

```
                if (node, c) in table:
                    state += table[(node,c)]
            state = nodes_to_state_name(state)

            if not len(state):
                continue
            newTable[(nodes_to_state_name(nodes), c)] = [state]

            if state not in states:
                queue.append(state)


    def intersection(lst1, lst2):
        lst3 = [value for value in lst1 if value in lst2]
        return lst3


    return [nodes_to_state_name(first)], [
        state for state in states if intersection(state, last)
    ], newTable




def check_ka(st, fir, las, table):
    state = fir[0]
    for c in st:
        if (state, c) not in table:
            return False
        state = table[(state,c)][0]

    return state in las


def print_firstpos_info(node, indent=""):
    if node:
        print(indent + "Value:", node.value)
        print(indent + "Firstpos:", [n.value for n in node.firstpos])
```

```
319         print()
320         if node.left:
321             print(indent + "Left:")
322             print_firstpos_info(node.left, indent + "   ")
323         if node.right:
324             print(indent + "Right:")
325             print_firstpos_info(node.right, indent + "   ")
326 
327 def print_node_info(node, indent=""):
328     if node:
329         print(indent + "Value:", node.value)
330         print(indent + "Nullable:", node.nullable)
331         print(indent + "Firstpos:", [n.value for n in node.firstpos])
332         print(indent + "Lastpos:", [n.value for n in node.lastpos])
333         print(indent + "Followpos:", [n.value for n in node.followpos])
334         print()
335         if node.left:
336             print(indent + "Left:")
337             print_node_info(node.left, indent + "   ")
338         if node.right:
339             print(indent + "Right:")
340             print_node_info(node.right, indent + "   ")
341 
342 
343 def prepare_all(regK):
344     end_symb = 'END'
345     pt = buildParseTree(list(regK) + [end_symb])
346 
347 
348     pt.count_i()
349 
350     pt.count_nullable()
351     pt.count_firstpos()
352     pt.count_lastpos()
353 
354     pt.count_followpos()
355 
356     #print_node_info(pt)
357     #print_firstpos_info(pt)
358     nodes = []
```

```
359     def make_nodes(node):
360         if node.value not in ('+', '|', '', '*'):
361             nodes.append(node)
362 
363     pt.foreach_deep(FuncOut=make_nodes)
364     for i, node in enumerate(nodes, 1):
365         node.label_i = i
366 
367     symbols = list(set([node.value for node in nodes]))
368 
369     dfa = make_ka(pt, symbols, end_symb=end_symb)
370 
371     steps_V = []
372 
373     modified = dfa
374 
375 
376 
377     def step(ins):
378         fix = fix_names(*ins)
379         r = reverseKa(*fix)
380         dfa = toDFA(*r)
381         fix2 = fix_names(*dfa)
382         steps_V.append([fix, r, dfa, fix2])
383         return fix2
384 
385     modified = step(step(modified))
386 
387     return pt, dfa, modified, steps_V
388 
389 
390 
391 regK = "(a|b)*aabb"
392 z = prepare_all(regK)
393 
394 import traceback
395 def change_reg():
396     global regK, z
397     print("""Допустимые
398  терминальные символы: *, +, |, (, )Иные
```

```
 символы воспринимаются как нетерминальные
           """)
    x = input("Введите новое регулярное выражение: ")

    try:
        z = prepare_all(x)
    except Exception as e:
        print(traceback.format_exc())
        input("Произошла ошибка" + str(e))
        return change_reg()
    regK = x

def show_dla():
    z[0].show_tree().view()
def show_dfa():
    print_ka(*z[1]).view()
def show_mdfa():
    print_ka(*z[2]).view()

def show_mdfa_steps():
    for s in z[3]:
        for i in s:
            print_ka(*i).view()
            input()



def init_dfa():

    s = input("Введите строку для( выхода 'quit'): ")
    if s != "quit":
        res = check_ka(s, *z[2])
        print("Соответствует" if res else "Не соответствует")
        init_dfa()


while True:
    x = -1
    try:
        x = int(input("""Текущее
```

```
439  регулярное выражение {:}
440
441 0. Указать рег. выражение
442 1. Показать дерево
443 2. Показать ДКА
444 3. Показать МДКА
445 4. Проверить входную цепочку на соответствие рег. выражениюВведите
446
447  пункт меню: """.format(regK)))
448      except:
449          pass
450
451      actions = [
452          change_reg,
453          show_dla,
454          show_dfa,
455          show_mdfa,
456          init_dfa,
457          show_mdfa_steps,
458      ]
459
460      if x < 0 or x > len(actions):
461          input("Input error")
462          continue
463
464      actions[x]()
```

Список источников

1. Построение по НКА эквивалентного ДКА, алгоритм Томпсона | https://neerc.ifmo.ru/wiki/index.php?oldid=85540