Государственное образовательное учреждение
высшего профессионального образования
«Московский государственный технический университет
имени Н.Э. Баумана»

Отчет
По лабораторной работе
По курсу «Конструирование компиляторов»
На тему
«Распознавание цепочек регулярного языка»

Студент: Горин Д.И.
Группа: ИУ7-23М
Вариант: 3
Преподаватель: Ступников А.А.

Москва, 2020

Оглавление

1 Цель и задачи работы 2
2 Листинг 2
2.1 main.py 2
2.2 regexp_process.py 4
2.3 FSM.py 5
2.4 utils.py 17
3 Проверка корректности программы 18
4 Выводы 24
5 Список литературы 24

1 Цель и задачи работы

Цель работы: приобретение практических навыков реализации важнейших элементов лексических анализаторовна 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.

2 Листинг

2.1 main.py

```
1  from tabulate import tabulate
2  from utils import format_table_for_tabulate_dka,
       format_table_for_tabulate_nka
3  from regexp_process import get_postfix_regexp, TEST_ALPHABET,
       ALL_TEST_SYMBOLS, RegexpError
4  from FSM import generate_nfsm_from_pregexp, draw_nka_gz, \
5      generate_dfsm_from_nfsm, draw_dka_gz, \
6      generate_min_dka_from_dka, \
7      generate_min_dka_from_pregexp, \
8      dka_job
9
10 # TEST_REGEXP = 'ab|baba(abb*)+'
11 # TEST_REGEXP = '(b|ab*ab*)*'
12 # TEST_REGEXP = '(ab)*|ba'
13 # TEST_REGEXP = '(a|b)*abb'
14 # TEST_REGEXP = 'a|b|c'
15 # TEST_REGEXP = 'ab'
16 # TEST_REGEXP = 'a*'
17
18
19 if __name__ == '__main__':
20     # Вводрегулярки
21     TEST_REGEXP = input('Введите выражение, для которого необходимо
           построить автомат: ')
22     if [x for x in TEST_REGEXP if x not in ALL_TEST_SYMBOLS]:
23         print('Выражение не соответствует допустимому алфавиту')
24         exit(1)
25
26     # Построениепостфикснойформы
27     postfix_regexp = ''
```

```
28      try:
29          postfix_regexp = get_postfix_regexp(TEST_REGEXP)
30          print(f'Постфиксная запись регулярного выражения: {"".join(
                postfix_regexp)}')
31      except RegexpError as e:
32          print(e)
33          exit(1)
34
35      # ГенерацияНКАивывод
36      nka = generate_nfsm_from_pregexp(postfix_regexp, TEST_ALPHABET)
37      table, headers = format_table_for_tabulate_nka(nka.get_as_table(
            TEST_ALPHABET), TEST_ALPHABET)
38      print('\НКАn:')
39      print(tabulate(table, headers=headers))
40      draw_nka_gz(nka)
41
42      # ГенерацияДКАивывод
43      dka = generate_dfsm_from_nfsm(nka.get_as_table(TEST_ALPHABET),
            TEST_ALPHABET)
44      table, headers = format_table_for_tabulate_dka(dka,
            TEST_ALPHABET)
45      print('\ДКАn:')
46      print(tabulate(table, headers=headers))
47      draw_dka_gz(dka)
48
49      # ГенерацияминимальногоДКАивывод
50      min_dka = generate_min_dka_from_dka(dka, TEST_ALPHABET)
51      table, headers = format_table_for_tabulate_dka(min_dka,
            TEST_ALPHABET)
52      print('\Минимальныйn ДКА:')
53      print(tabulate(table, headers=headers))
54      draw_dka_gz(min_dka, is_min=True)
55
56      # ЦиклмоделированияработыКА
57      to_check = ''
58      while to_check != '$':
59          to_check = input('\Введитеn слово, которое нужно проверить ($
                для завершения): ')
60          try:
61              is_ok = dka_job(min_dka, to_check)
62          except ValueError as e:
63              if to_check == '$':
64                  break
65              print(e)
66              continue
67          if is_ok:
68              print('Автомат допускает введенное слово')
69          else:
70              print('Автомат не допускает введенное слово')
71
```

```
72      # Конец
73      print('Покапока-_-:)')
```

2.2 regexp_process.py

```
1  from typing import List
2
3
4  # TEST_ALPHABET = [chr(x) for x in range(ord('a'), ord('z')+1)] + [
       chr(x) for x in range(ord('A'), ord('Z')+1)] + \
5  #                 [str(x) for x in range(10)]
6  TEST_ALPHABET = ['a', 'b']
7  TEST_OPS_PRECEDENCE = {
8      '|': 0,
9      '+': 2,
10     '*': 2,
11     '(': -1,
12     ')': -1,
13     '.': 1,
14 }
15 TEST_OPS = list(TEST_OPS_PRECEDENCE.keys())
16 ALL_TEST_SYMBOLS = TEST_ALPHABET + TEST_OPS
17 CONCAT_OP = TEST_OPS[-1]
18
19
20 class RegexpError(Exception):
21     def __init__(self, message: str):
22         self.message = message
23
24     def __str__(self):
25         return self.message
26
27
28 def get_postfix_regexp(regexp: str) -> List[str]:
29     return __create_postfix_notation_regexp(__normalize_regexp(
           regexp))
30
31
32 def __normalize_regexp(regexp: str) -> str:
33     """Добавлениезнакаконкатенациикрегулярке
34
35     :param regexp: Регулярка
36     :return: Регуляркасознакамиконкатенации
37     """
38     nregexp = ''
39     for i in range(len(regexp)-1):
40         if regexp[i] not in ALL_TEST_SYMBOLS:
41             raise RegexpError(message=f'Неизвестный_символ_"{regexp[i
                   ]}"')
42         nregexp += regexp[i]
43         if regexp[i] in TEST_ALPHABET + ['*', '+', ')'] and regexp[i
```

```
            +1] in TEST_ALPHABET + ['(']:
44              nregexp += CONCAT_OP
45      if regexp[-1] not in ALL_TEST_SYMBOLS:
46          raise RegexpError(message=f'Неизвестный символ "{regexp[-1]}"'
                )
47      nregexp += regexp[-1]
48      return nregexp
49
50
51  def __create_postfix_notation_regexp(normalized_regexp: str) -> List
        [str]:
52      """Переводрегекспаизинфикснойзаписивпостфиксную
53
54      :param normalized_regexp: Регуляркассимволамиконкатенации
55      :return: Постфикснаянотациярегекспа
56      """
57      queue = []
58      stack = []
59      while len(normalized_regexp) != 0:
60          sym = normalized_regexp[0]
61          normalized_regexp = normalized_regexp[1:]
62          if sym in TEST_ALPHABET:
63              queue.append(sym)
64          elif sym == '(':
65              stack.append('(')
66          elif sym == ')':
67              while len(stack) > 0 and stack[-1] != '(':
68                  queue.append(stack.pop())
69              try:
70                  stack.pop()
71              except IndexError:
72                  raise RegexpError('Не хватает открывающей скобки')
73          elif sym in TEST_OPS:
74              while len(stack) > 0 and TEST_OPS_PRECEDENCE[stack[-1]]
                    >= TEST_OPS_PRECEDENCE[sym]:
75                  queue.append(stack.pop())
76              stack.append(sym)
77          else:
78              raise RegexpError('Неизвестный символ в построении
                    постфиксной формы')
79      while len(stack) > 0:
80          stack_sym = stack.pop()
81          if stack_sym == '(':
82              raise RegexpError('Не хватает закрывающей скобки')
83          queue.append(stack_sym)
84      return queue
```

2.3 FSM.py

```
1  from typing import List, Dict, Set, Tuple, Iterable, Union
2  from graphviz import Digraph
```

```
3
4
5  # MARK: - NFMS
6
7  class FiniteStateMachineNode:
8      """Состояниеконечногоавтомата
9
10     outputs вформате  [(node1, символ< перехода>), ...]
11     """
12
13     def __init__(self, state, outputs=None):
14         if outputs is None:
15             outputs = list()
16         self.state = state
17         self.outputs = outputs
18
19     @property
20     def is_end_state(self):
21         return len(self.outputs) == 0
22
23     def outputs_append(self, node, symbol='eps'):
24         self.outputs.append((node, symbol))
25
26     def __str__(self):
27         return str(self.state)
28
29     def __repr__(self):
30         return str(self)
31
32
33 class NKA:
34     """КлассдляНКА
35
36     """
37     state_num = 0
38
39     def __init__(self, root_state: FiniteStateMachineNode = None,
           symbol: str = 'eps'):
40         if root_state:
41             self.root_state = root_state
42         else:
43             st_node = FiniteStateMachineNode(state=NKA.state_num +
                   1)
44             end_node = FiniteStateMachineNode(state=NKA.state_num +
                   2)
45             st_node.outputs_append(end_node, symbol=symbol)
46             self.root_state = st_node
47             NKA.state_num += 2
48
49     def copy(self):
```

```
50          return NKA(self.root_state)
51
52      @property
53      def end_state(self):
54          node = self.root_state
55          while not node.is_end_state:
56              node = node.outputs[0][0]
57          return node
58
59      def concat(self, nka):
60          """
61          (self) -eps-> (nka)
62          """
63          onode_1, onode_2 = self.copy(), nka.copy()
64          onode_1.end_state.outputs_append(onode_2.root_state)
65          return NKA(root_state=onode_1.root_state)
66
67      def oorr(self, nka):
68          """
69              //eps->(self)-\
70          (S)-|               |-eps->(F)
71              \\eps->(nka)-/
72
73          """
74          st_node = FiniteStateMachineNode(state=NKA.state_num + 1)
75          end_node = FiniteStateMachineNode(state=NKA.state_num + 2)
76          onode_1, onode_2 = self.copy(), nka.copy()
77          onode_1.end_state.outputs_append(end_node)
78          onode_2.end_state.outputs_append(end_node)
79          st_node.outputs_append(onode_1.root_state)
80          st_node.outputs_append(onode_2.root_state)
81          NKA.state_num += 2
82          return NKA(root_state=st_node)
83
84      def plus(self):
85          """
86              /<---------eps----------\
87          (S) -eps-> (self) -eps-> (PF) -eps-> (F)
88          """
89          st_node = FiniteStateMachineNode(state=NKA.state_num + 1)
90          pre_end_node = FiniteStateMachineNode(state=NKA.state_num +
                2)
91          end_node = FiniteStateMachineNode(state=NKA.state_num + 3)
92          onode = self.copy()
93          pre_end_node.outputs_append(end_node)
94          pre_end_node.outputs_append(st_node)
95          onode.end_state.outputs_append(pre_end_node)
96          st_node.outputs_append(onode.root_state)
97          NKA.state_num += 3
98          return NKA(root_state=st_node)
```

```
99
100     def star(self):
101         """
102             /<--------eps----------\
103         (S) -eps-> (self) -eps-> (PF) -eps-> (F)
104             \\--------eps---------->/
105         """
106         st_node = FiniteStateMachineNode(state=NKA.state_num + 1)
107         pre_end_node = FiniteStateMachineNode(state=NKA.state_num +
                2)
108         end_node = FiniteStateMachineNode(state=NKA.state_num + 3)
109         onode = self.copy()
110         pre_end_node.outputs_append(end_node)
111         pre_end_node.outputs_append(st_node)
112         onode.end_state.outputs_append(pre_end_node)
113         st_node.outputs_append(end_node)
114         st_node.outputs_append(onode.root_state)
115         NKA.state_num += 3
116         return NKA(root_state=st_node)
117
118     def __get_table_row(self, alphabet: List[str]) -> Dict[str, List
            [str]]:
119         ans = {'eps': []}
120         for sym in alphabet:
121             ans[sym] = []
122         return ans
123
124     def get_as_table(self, alphabet: List[str]) -> Dict[str, Dict[
            str, List[str]]]:
125         """Автоматввидетаблицы
126
127         """
128         ans = {}
129         already_seen = []
130         stack = [self.root_state]
131         while len(stack) > 0:
132             node = stack.pop()
133             if str(node.state) in already_seen:
134                 continue
135             already_seen.append(str(node.state))
136             for nd, sym in node.outputs:
137                 stack.append(nd)
138                 try:
139                     row = ans[str(node.state)]
140                 except KeyError:
141                     ans[str(node.state)] = self.__get_table_row(
                            alphabet)
142                     row = ans[str(node.state)]
143                 row[sym].append(str(nd.state))
144         ans['f'] = self.__get_table_row(alphabet)
```

```
145         return ans
146
147
148 def generate_nfsm_from_pregexp(pregexp: List[str], alphabet: List)
    -> NKA:
149     """ГенерацияНКАизпостфикснойрегулярки
150
151     :param alphabet: Допустимыйалфавит
152     :param pregexp: Постфикснаярегулярка
153     :return: НачальноесостояниеНКА
154     """
155     stack = []
156     while len(pregexp) > 0:
157         cur_symbol = pregexp.pop(0)
158         if cur_symbol in alphabet:
159             stack.append(NKA(symbol=cur_symbol))
160         elif cur_symbol == '.':
161             nka2 = stack.pop()
162             nka1 = stack.pop()
163             new_nka = nka1.concat(nka2)
164             stack.append(new_nka)
165         elif cur_symbol == '|':
166             nka2 = stack.pop()
167             nka1 = stack.pop()
168             new_nka = nka1.oorr(nka2)
169             stack.append(new_nka)
170         elif cur_symbol == '*':
171             nka = stack.pop()
172             new_nka = nka.star()
173             stack.append(new_nka)
174         elif cur_symbol == '+':
175             nka = stack.pop()
176             new_nka = nka.plus()
177             stack.append(new_nka)
178     start_node = FiniteStateMachineNode(state='s')
179     end_node = FiniteStateMachineNode(state='f')
180     nka = stack.pop()
181     nka.end_state.outputs_append(end_node)
182     start_node.outputs_append(nka.root_state)
183     return NKA(root_state=start_node)
184
185
186 def draw_nka_gz(nka: NKA):
187     d = Digraph()
188     stack = [nka.root_state]
189     already_was = []
190     while len(stack) > 0:
191         node = stack.pop()
192         if node.state in already_was:
193             continue
```

```
194             already_was.append(node.state)
195             for tpl in node.outputs:
196                 d.edge(f'{node.state}', f'{tpl[0].state}', label=tpl[1])
197                 stack.append(tpl[0])
198     with open('nka', 'w') as f:
199         f.write(d.source)
200     d.render('nka', view=True)
201
202
203 # MARK: - DFMS
204
205 class DFSMState:
206     """КлассдлясостоянияДКА
207
208     """
209
210     def __init__(self, state: str, nka_states: Set[str], outputs:
        List[Tuple[str, str]] = None, is_final=False):
211         """
212         :param state: Состояние
213         :param nka_states: МножествосостоянийНКА ,
            соответствующихданномусостояниюДКА
214         :param outputs: Переходывдругиесостояниявформате (state,
            symbol)
215         :param is_final: Являетсялисостояниефинальным
216         """
217         self.state = state
218         self.nka_states = nka_states
219         self.outputs = outputs or []
220         self.is_final = is_final
221         if self.is_final:
222             self.state += 'f'
223
224     def append_output(self, state: str, symbol: str):
225         self.outputs.append((state, symbol))
226
227     def __str__(self):
228         return self.state
229
230     def __repr__(self):
231         return str(self)
232
233     def __eq__(self, other):
234         return self.nka_states == other.nka_states
235
236     def __hash__(self):
237         return hash(frozenset(self.nka_states))
238
239
240 def generate_dfsm_from_nfsm(nfsm: Dict[str, Dict[str, List[str]]],
```

```
    alphabet: List[str]) -> List[DFSMState]:
241     """ПреобразованиеНКАтабличного
242       (представления) вДКАпо (сутитожевтабличное )
            поалгоритмуизУльмана
243     :param nfsm: НКА
244     :param alphabet: Допустимыйалфавит
245     :return: ДКА
246     """
247     ans = []
248     __ec = __eps_closure_for_nka_state(nfsm, 's')
249     stack = [DFSMState('s', __ec, is_final='f' in __ec)]
250     marked_states = [(stack[0].nka_states, 's')]
251     states_cnt = 1
252     while len(stack) > 0:
253         dstate = stack.pop()
254         for asymbol in alphabet:
255             move_by_asymbol = __move_closure_for_set_of_nka_states(
                    nfsm, dstate.nka_states, asymbol)
256             u = __eps_closure_for_set_of_nka_states(nfsm,
                    move_by_asymbol)
257             new_dstate = DFSMState(state=str(states_cnt), nka_states
                    =u, is_final='f' in u)
258             if new_dstate not in ans and new_dstate not in stack:
259                 stack.append(new_dstate)
260                 states_cnt += 1
261                 marked_states.append((new_dstate.nka_states,
                        new_dstate.state))
262             dstate.append_output(state=[x[1] for x in marked_states
                    if x[0] == u][0], symbol=asymbol)
263         ans.append(dstate)
264     return __remove_states_without_inputs(ans)
265
266
267 def __remove_states_without_inputs(dka: List[DFSMState]) -> List[
    DFSMState]:
268     seen_in_output = {'s', 'sf'}
269     for state in dka:
270         for ostate, _ in state.outputs:
271             seen_in_output.add(ostate)
272     ans = []
273     for state in dka:
274         if state.state in seen_in_output:
275             ans.append(state)
276     return ans
277
278
279 def __eps_closure_for_nka_state(nfsm, state: str) -> Set[str]:
280     """Поискэпсилонзамыкания
281      - измножества state
282     :param nfsm: НКА
```

```
283         :param state: Состояние
284         :return: Множествосостояний, достижимыхизданноготолькопо eps
                переходам-
285         """
286         ans = {state}
287         stack = [state]
288         was_seen = []
289         while len(stack) > 0:
290             cur_state = stack.pop()
291             if cur_state in was_seen:
292                 continue
293             was_seen.append(cur_state)
294             ans.add(state)
295             to_ext = nfsm[cur_state]['eps']
296             ans.update(to_ext)
297             stack.extend(to_ext)
298         return ans
299
300
301 def __eps_closure_for_set_of_nka_states(nfsm, states) -> Set[str]:
302         """Поискэпсилонзамыкания
303          - измножествасостояний states
304         :param nfsm: НКА
305         :param states: Состояния
306         :return: Множествосостояний,
                достижимыхизданногомножествасостоянийтолькопо eps переходам-
307         """
308         ans = set()
309         for state in states:
310             eps_closure_for_state = __eps_closure_for_nka_state(nfsm,
                    state)
311             ans.update(eps_closure_for_state)
312         return ans
313
314
315 def __move_closure_for_nka_state(nfsm, state: str, symbol: str) ->
        Set[str]:
316         """Поисксостояний
317          , напрямуюдостижимыхизданногопосимволу
318         :param nfsm: НКА
319         :param state: Состояние
320         :param symbol: Символ
321         :return: Множествосостояний, достижимыхизданногопосимволу
                symbol
322         """
323         return set(nfsm[state][symbol])
324
325
326 def __move_closure_for_set_of_nka_states(nfsm, states, symbol: str)
        -> Set[str]:
```

```
327         """Поисксостояний
328           , напрямуюдостижимыхизданногомножествапосимволу
329         :param nfsm: НКА
330         :param states: Состояния
331         :param symbol: Символ
332         :return: Множествосостояний  ,
                достижимыхизданногомножествасостоянийпосимволу          symbol
333         """
334         ans = set()
335         for state in states:
336             cur_move_closure = __move_closure_for_nka_state(nfsm, state,
                    symbol)
337             ans.update(cur_move_closure)
338         return ans
339
340
341 # MARK: - Min DFMS
342
343 class MinDFSMState:
344     """Классдлясостоянияминимальногодка
345
346     """
347
348     def __init__(self, state: str, dka_states: Set[DFSMState],
           outputs: List[Tuple[str, str]] = None, is_final=False):
349         """
350         :param state: Состояние
351         :param dka_states: МножествосостоянийДКА   ,
               соответствующихданномусостояниюминимальногоДКА
352         :param outputs: Переходывдругиесостояниявформате        (state,
               symbol)
353         :param is_final: Являетсялисостояниефинальным
354         """
355         self.state = state
356         self.dka_states = dka_states
357         self.outputs = outputs or []
358         self.is_final = is_final
359         if is_final:
360             self.state += 'f'
361
362     @property
363     def dka_states_names(self) -> Set[str]:
364         return set([x.state for x in self.dka_states])
365
366     def append_output(self, state: str, symbol: str):
367         self.outputs.append((state, symbol))
368
369     def set_as_final(self):
370         self.is_final = True
371         if self.state[-1] != 'f':
```

```
372             self.state += 'f'
373
374     def state_as_not_final(self):
375         self.is_final = False
376         if self.state[-1] == 'f':
377             self.state = self.state[:-1]
378
379     def is_start_state(self):
380         return 's' in self.state
381
382     def __str__(self):
383         return self.state
384
385     def __repr__(self):
386         return str(self)
387
388     def __eq__(self, other):
389         return self.dka_states == other.dka_states
390
391     def __hash__(self):
392         return hash(frozenset(self.dka_states))
393
394
395 def generate_min_dka_from_dka(dka: List[DFSMState], alphabet: List[
    str]) -> List[MinDFSMState]:
396     """ГенерацияминимальногоДКАизДКА
397
398     :param dka: ДКА
399     :param alphabet: Допустимыйалфавит
400     :return: МинимальныйДКА
401     """
402     # ПрименениеалгоритмаХопкрофта
403     new_states_sets = __hopcroft_main_job(dka, alphabet)
404     # СозданиесостоянийминимальногоДКАбезпереходов
405     new_states = []
406     for i, states_set in enumerate(new_states_sets):
407         dka_states_set = set([x for x in states_set])
408         dka_states_names = set([x.state for x in states_set])
409         is_final = any([x.is_final for x in states_set])
410         state_name = f'{i}{"s" if len(dka_states_names.intersection
            ({"s", "sf"})) != 0 else ""}'
411         new_state = MinDFSMState(state=state_name, dka_states=
            dka_states_set, is_final=is_final)
412         new_states.append(new_state)
413     # Созданиепереходов
414     for state in new_states:
415         outputs = __create_outputs_for_min_dka_state(new_states,
            state)
416         state.outputs = outputs
417     return new_states
```

```
418
419
420 def __hopcroft_main_job(dka: List[DFSMState], alphabet: List[str])
        -> List[Set[DFSMState]]:
421     """ФункцияалгоритмаХопкрофта
422
423     """
424     fins, non_fins = __find_finals_and_rest(dka)
425     w = [fins]
426     p = [fins]
427     if len(non_fins) != 0:  # Можетбытьтакое , чтовсесостоянияДКА
            — финальные, ипустогомножестванамненадо
428         w.append(non_fins)
429         p.append(non_fins)
430     while len(w) > 0:
431         s = w.pop(0)
432         for asym in alphabet:
433             inputs = __find_all_inputs_for_states(dka, s, asym)
434             for r in p:
435                 if len(r.intersection(inputs)) == 0 or r.issubset(
                        inputs):
436                     continue
437                 r1 = r.intersection(inputs)
438                 r2 = r.difference(r1)
439                 p.remove(r)
440                 p.append(r1)
441                 p.append(r2)
442                 if r in w:
443                     w.remove(r)
444                     w.append(r1)
445                     w.append(r2)
446                 else:
447                     r_min = r1 if len(r1) < len(r2) else r2
448                     w.append(r_min)
449     return p
450
451
452 def __find_finals_and_rest(dka: Iterable[DFSMState]) -> Tuple[Set[
        DFSMState], Set[DFSMState]]:
453     """Поисквсехфинальныхинефинальныхсостояний ДКА
454
455     :param dka: ДКА
456     :return: Спискифинальныхинефинальныхсостояний
457     """
458     fin = set()
459     non_fin = set()
460     for state in dka:
461         if state.is_final:
462             fin.add(state)
463         else:
```

```
464                 non_fin.add(state)
465     return fin, non_fin
466
467
468 def __find_all_inputs_for_state(dka: List[DFSMState], dka_state:
        DFSMState, sym: str) -> Set[DFSMState]:
469     """Поисксостояний
470       , входящихвданное
471     :param dka: ДКА
472     :param dka_state: Состояние, входывкотороемыищем
473     :param sym: Поданномусимволу
474     :return: Названиявходящихсостояний
475     """
476     ans = set()
477     for state in dka:
478         if (dka_state.state, sym) in state.outputs:
479             ans.add(state)
480     return ans
481
482
483 def __find_all_inputs_for_states(dka: List[DFSMState], dka_states:
        Iterable[DFSMState], sym: str) -> Set[DFSMState]:
484     """Тожесамое
485       , нодлямножествасостояний
486     """
487     ans = set()
488     for state in dka_states:
489         ans.update(__find_all_inputs_for_state(dka, state, sym))
490     return ans
491
492
493 def __create_outputs_for_min_dka_state(min_dka: List[MinDFSMState],
        state: MinDFSMState) -> List[Tuple[str, str]]:
494     """СозданиеспискавыходовдлясостоянияминимальногоДКА
495
496     """
497     outputs = []
498     dka_outputs = next(iter(state.dka_states)).outputs
499     for dka_state_name, sym in dka_outputs:
500         state_to_go = [x for x in min_dka if dka_state_name in x.
              dka_states_names][0]
501         outputs.append((state_to_go.state, sym))
502     return outputs
503
504
505 def draw_dka_gz(dka: Union[List[DFSMState], List[MinDFSMState]],
        is_min=False):
506     d = Digraph()
507     filename = 'dka_min' if is_min else 'dka'
508     for state in dka:
```

```
509             for output_state, symbol in state.outputs:
510                 d.edge(state.state, output_state, label=symbol)
511         with open(filename, 'w') as f:
512             f.write(d.source)
513         d.render(filename, view=True)
514 
515 
516 def dka_job(dka: List[MinDFSMState], word: str) -> bool:
517     """Функция
518     , моделирующаяКА
519     """
520     cur_state = [x for x in dka if x.is_start_state()][0]
521     while len(word) > 0:
522         sym = word[0]
523         word = word[1:]
524         try:
525             state_to_go = [x[0] for x in cur_state.outputs if x[1]
                == sym][0]
526         except IndexError:
527             raise ValueError(f'Символа {sym} нет в допустимом алфавите!
                ')
528         cur_state = [x for x in dka if x.state == state_to_go][0]
529     return cur_state.is_final
530 
531 
532 # MARK: - All in one
533 def generate_min_dka_from_pregexp(pregexp, alphabet) -> List[
    MinDFSMState]:
534     """Получениеминимальногодкадляпостфиксногорегекспа
535 
536     """
537     nka = generate_nfsm_from_pregexp(pregexp, alphabet)
538     dka = generate_dfsm_from_nfsm(nka.get_as_table(alphabet),
        alphabet)
539     min_dka = generate_min_dka_from_dka(dka, alphabet)
540     return min_dka
```

2.4 utils.py

```
1 from typing import Tuple, List, Union, Dict
2 from FSM import DFSMState, MinDFSMState
3 
4 
5 def format_table_for_tabulate_nka(tbl: Dict[str, Dict[str, List[str
    ]]], alphabet) ->\
6         Tuple[List[Union[str, List]], List[str]]:
7     headers = ['eps'] + alphabet
8     rows = []
9     for state, row in tbl.items():
10         cur_row = [str(state)] + list(row.values())
11         rows.append(cur_row)
```

```
12      return rows, headers
13
14
15  def format_table_for_tabulate_dka(dka: Union[List[DFSMState], List[
        MinDFSMState]], alphabet) ->\
16          Tuple[List[str], List[str]]:
17      headers = alphabet
18      rows = []
19      for state in dka:
20          cur_row = [str(state)]
21          for symbol in alphabet:
22              state_lst = [x[0] for x in state.outputs if x[1] ==
                    symbol]
23              if len(state_lst) == 0:
24                  cur_row.append('')
25              elif len(state_lst) == 1:
26                  cur_row.append(state_lst[0])
27              else:
28                  raise ValueError('Not DKA was given')
29          rows.append(cur_row)
30      return rows, headers
```

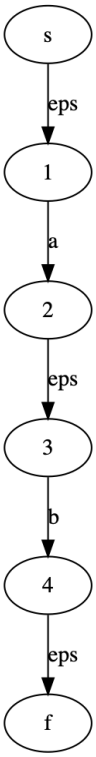
3 Проверка корректности программы

Все тесты производятся для алфавита $\Sigma = \{a, b\}$, конкатенация заменяется на символ «.», начальные состояния имеют в названии символ «s», а финальные «f».

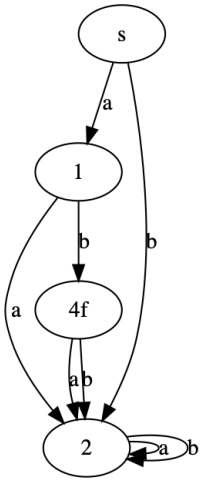
Выражение ab

Постфиксное регулярное выражение:
$ab.$

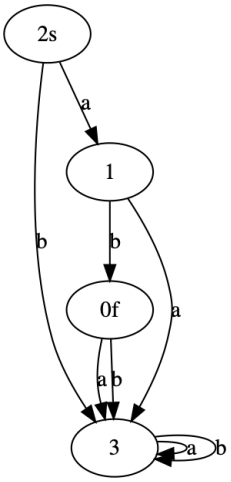
НКА:



ДКА:



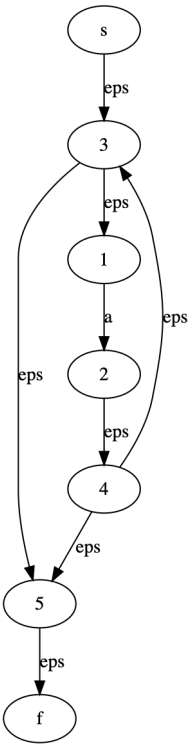
Минимальный ДКА:



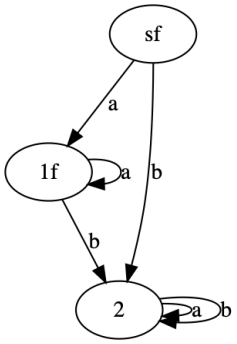
Выражение $a*$

Постфиксное регулярное выражение:
$a*$

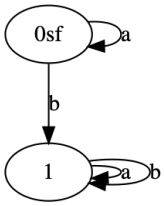
НКА:



ДКА:



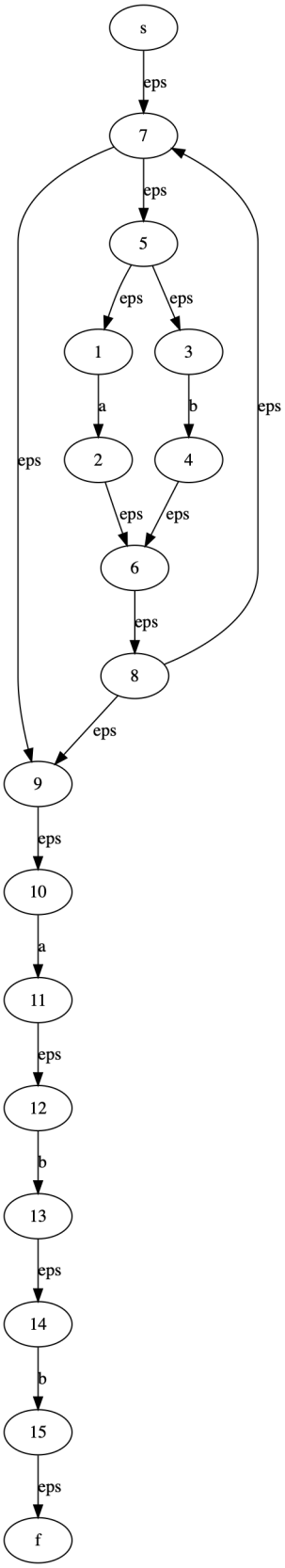
Минимальный ДКА:



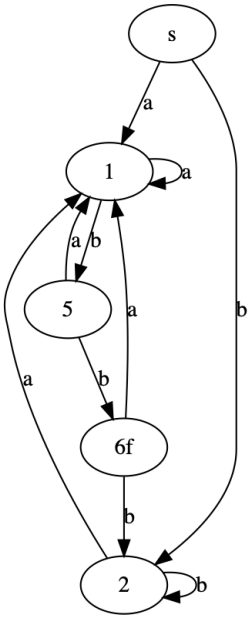
Выражение $(a|b) * abb$

Постфиксное регулярное выражение:
$ab| * a.b.b.$

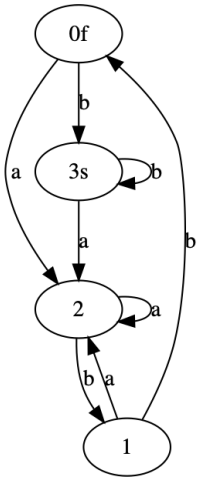
HKA:


s
eps
7
eps
5
eps
eps
1
3
a
b
eps
eps
2
4
eps
eps
6
eps
8
eps
9
eps
10
a
11
eps
12
b
13
eps
14
b
15
eps
f

ДКА:



Минимальный ДКА:



4 Выводы

По результатам проведенной работы студент ознакомился с основными определениями и понятиями, лежащими в основе построения лексических анализаторов и приобрел опыт в их реализации. В том числе была реализована программа, принимающая грамматику, по которой строится допускающий ее НКА, ДКА и минимальный ДКА, а также моделируется работа минимального ДКА для проверки входной цепочки символов

5 Список литературы

1. БЕЛОУСОВ А.И., ТКАЧЕВ С.Б. Дискретная математика: Учеб. Для вузов / Под ред. В.С. Зарубина, А.П. Крищенко. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2001.

2. АХО А., УЛЬМАН Дж. Теория синтаксического анализа, перевода и компиляции: В 2-х томах. Т.1.: Синтаксичечкий анализ. - М.: Мир, 1978.

3. АХО А.В, ЛАМ М.С., СЕТИ Р., УЛЬМАН Дж.Д. Компиляторы: принципы, технологии и инструменты. – М.: Вильямс, 2008.