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 № 6

Студент ИУ7-22М (Группа) _____ (Подпись, дата) К.Э. Ковалец (И. О. Фамилия)

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

2024 г.

1 Выполнение лабораторной работы

1.1 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 Хопкрофта).

1.2 Набор тестов

Таблица 1.1 – Набор тестов и ожидаемые результаты работы программы

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

1.3 Результаты работы программы

Результаты работы программы для регулярного выражения `(a|b)*abb` приведены на рисунках 1.1–1.5.

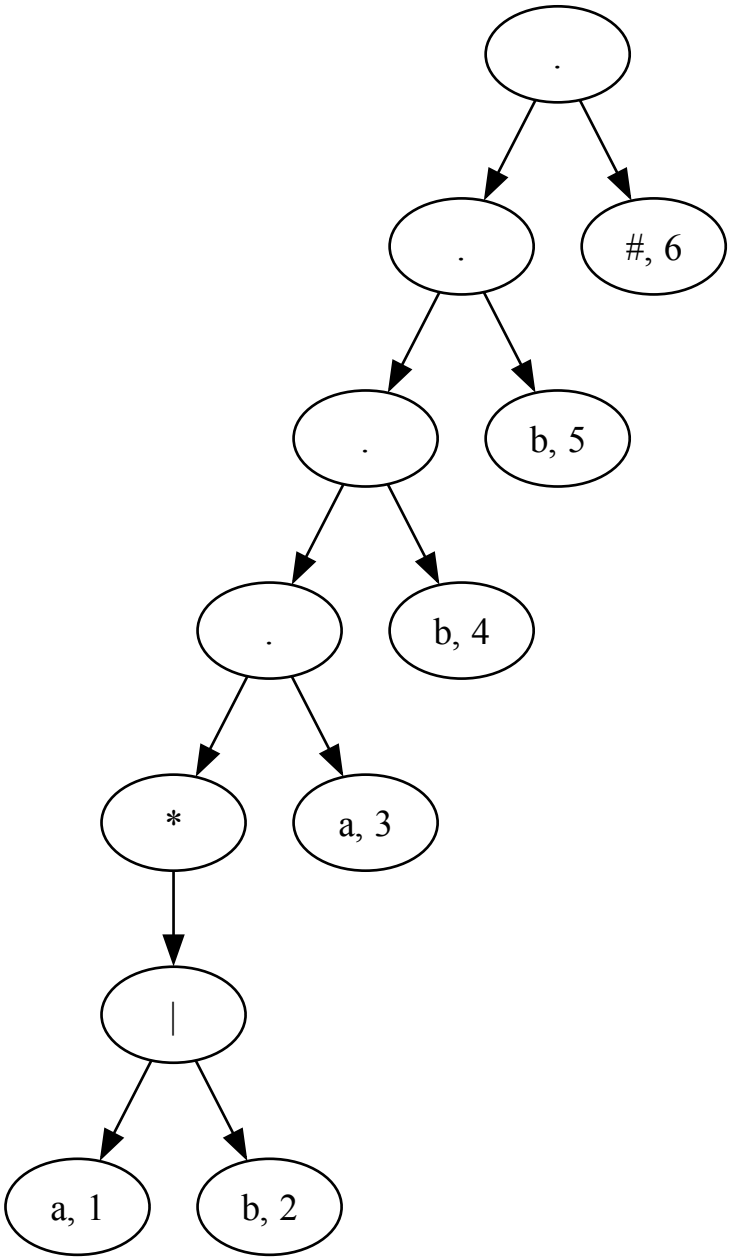


Рисунок 1.1 – Синтаксическое дерево для регулярного выражения

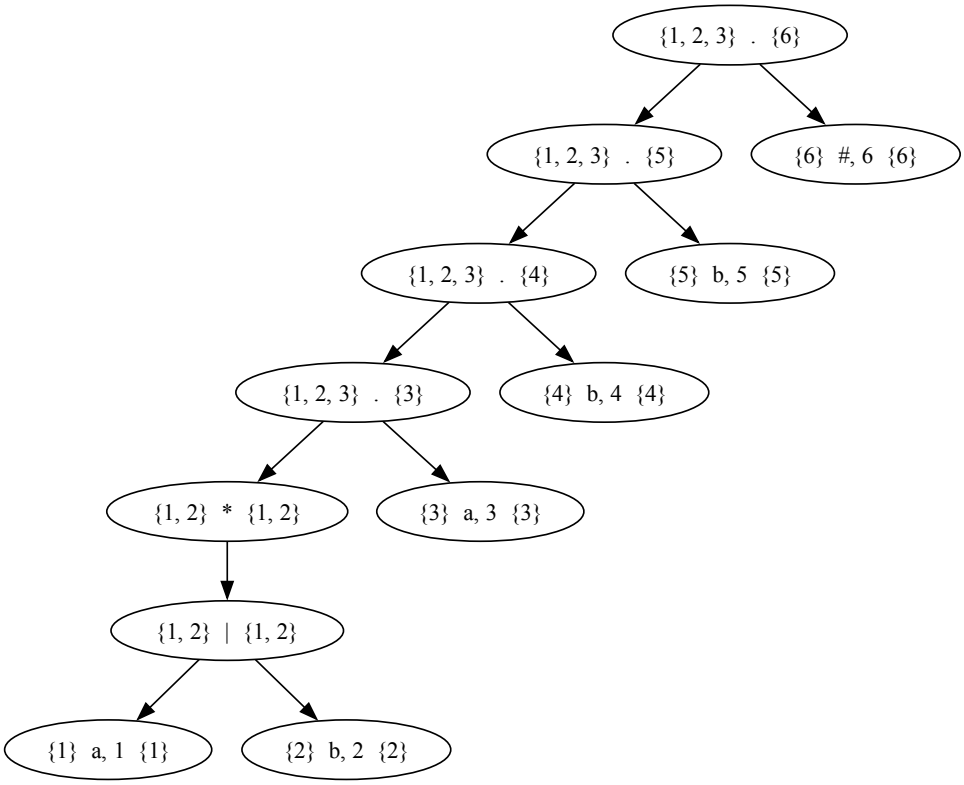


Рисунок 1.2 – Значения функций firstpos и lastpos в узлах синтаксического дерева

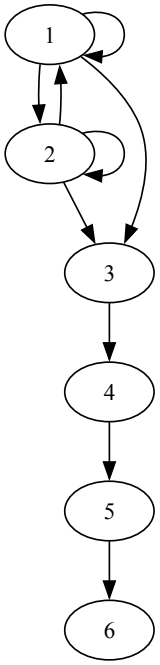


Рисунок 1.3 – Ориентированный граф для функции followpos

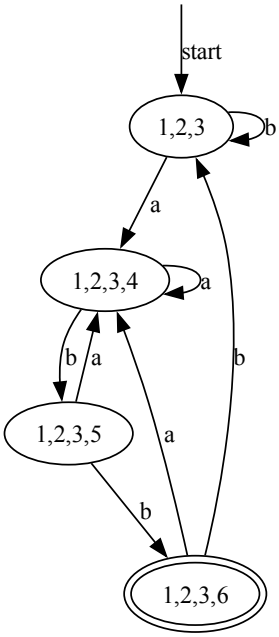


Рисунок 1.4 – ДКА для регулярного выражения

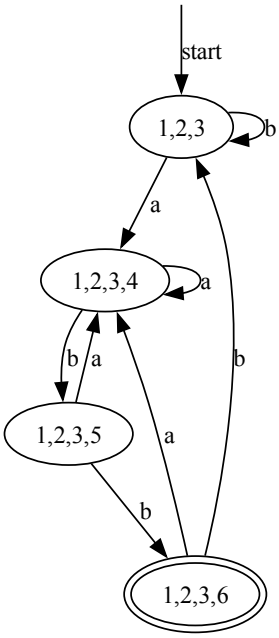


Рисунок 1.5 – Минимизированный ДКА алгоритмом Хопкрофта

2 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)

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

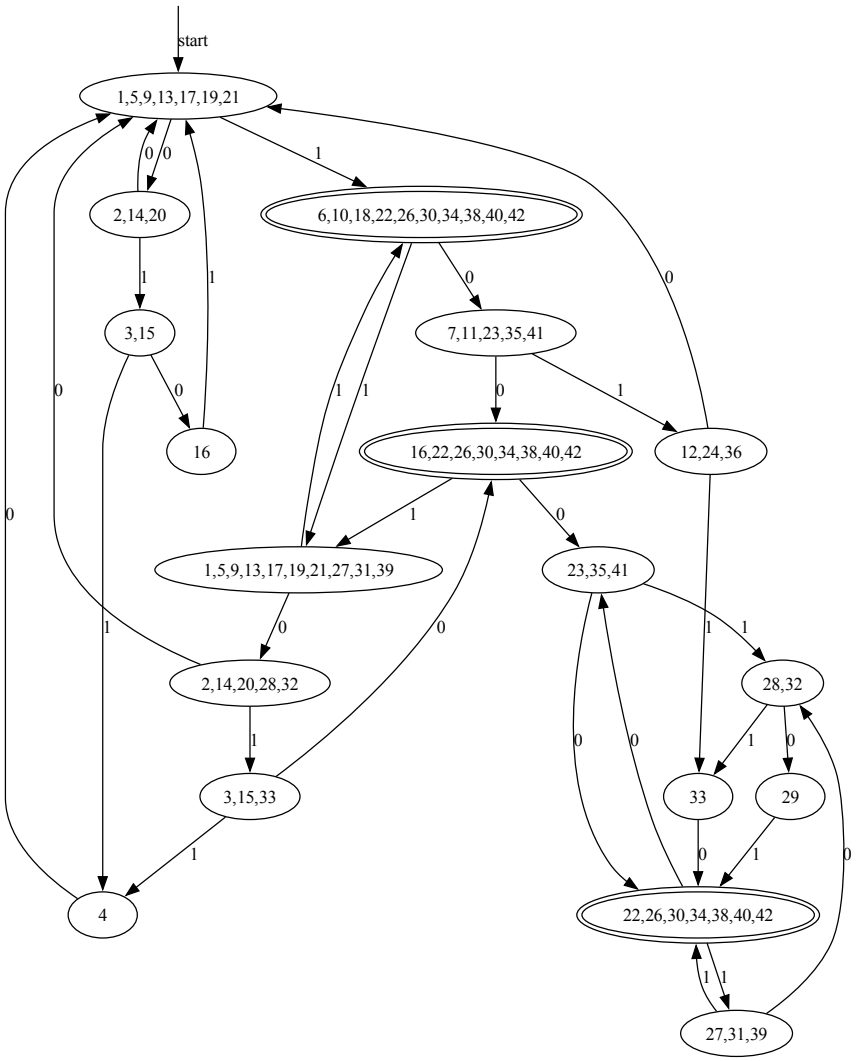
(b)

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

(c)

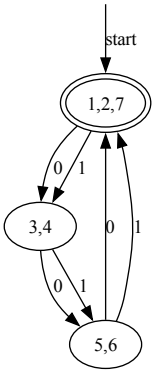
$$
\begin{aligned}
S &\to A \\
S &\to 1S \\
S &\to \epsilon \\
A &\to 0A \\
A &\to 00S
\end{aligned}
\tag{2.3}
$$

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



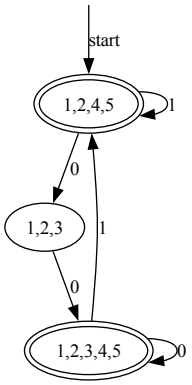
(a)

Рисунок 2.1 – ДКА для первого регулярного выражения



(b)

Рисунок 2.2 – ДКА для второго регулярного выражения



(c)

Рисунок 2.3 – 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 2.4 – Таблица для 4 вопроса

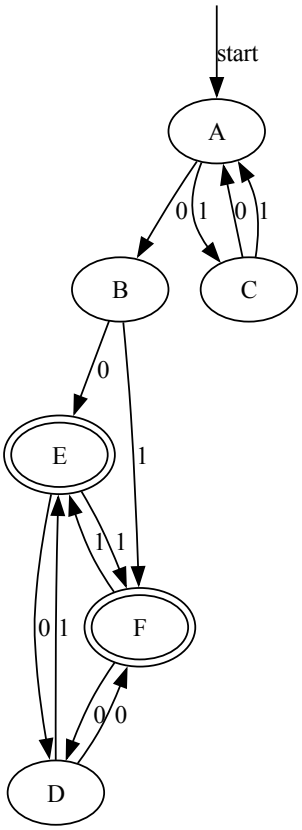


Рисунок 2.5 – ДКА для языка, определяемого автоматом M

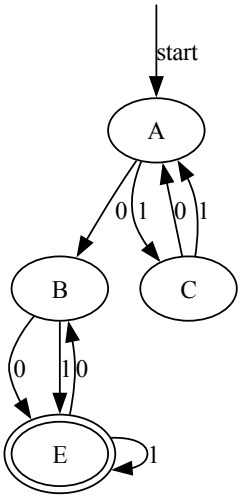


Рисунок 2.6 – Минимизированный ДКА для языка, определяемого автоматом M

3 Текст программы

В листингах 3.6–3.7 представлен код программы.

Листинг 3.1 — Основной модуль программы

```
from color import *
from regularExpression import convertRegexToDesiredFormat, ALPHABET
from parseTree import ParseTree
from dfa import DFA
from minDfa import MinDFA
from chain import inputChainCheckCorrespondence


MSG = f"""
    {YELLOW}\tМеню\n
    {YELLOW}1.{BASE}  Синтаксическое дерево для регулярного выражения;
    {YELLOW}2.{BASE}  Значения функций firstpos и lastpos в узлах синтаксического
↪   дерева;
    {YELLOW}3.{BASE}  Ориентированный граф для функции followpos;
    {YELLOW}4.{BASE}  ДКА для регулярного выражения;
    {YELLOW}5.{BASE}  Минимизированный ДКА алгоритмом Хопкрофта;
    {YELLOW}6.{BASE}  Проверка входной цепочки на соответсвие регулярному
↪   выражению;

    {YELLOW}0.{BASE}  Выход.\n
    {GREEN}Выбор:{BASE} """


def inputOption():
    try:
        option = int(input(MSG))
    except:
        option = -1

    if option < 0 or option > 6:
        print("%s\nОжидался ввод целого числа от 0 до 6%s" %(RED, BASE))

    return option


def main():
    regex = input(f"\n{BLUE}Введите регулярное выражение: {BASE}")
    convertedRegex = convertRegexToDesiredFormat(regex)
    if convertedRegex is None:
        return

```

Продолжение листинга 3.1

```
        parseTree = ParseTree(convertedRegex)
        parseTree.printTree()

        dfa = DFA(parseTree)
        dfa.printFirstposLastpos()
        dfa.printFollowpos()
        dfa.printDFA()

        minDFA = MinDFA(dfa, ALPHABET)
        minDFA.printGroupList()
        minDFA.printMinDFA()

        option = -1
        while option != 0:
            option = inputOption()
            match option:
                case 1:
                    parseTree.buildGraph(view=True)
                case 2:
                    dfa.buildFirstposLastposGraph(view=True)
                case 3:
                    dfa.buildFollowposGraph(view=True)
                case 4:
                    dfa.buildDFAGraph(view=True)
                case 5:
                    minDFA.buildMinDFAGraph(view=True)
                case 6:
                    inputChainCheckCorrespondence(regex, minDFA)


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

Листинг 3.2 — Модуль обработки регулярных выражений

```
from pythonds.basic.stack import Stack
from color import *


ALPHABET = "qwertyuiopasdfghjklzxcvbnm0123456789"


def convertRegexToDesiredFormat(regex: str) -> str | None:
    regex = regex.replace(" ", "").lower()
    try:
```

Продолжение листинга 3.2

```
            checkRegex(regex)
        except ValueError as exc:
            print(f"\n{RED}{exc}{BASE}\n")
            return None

        regex = convertToDesiredFormat(regex)
        print(f"\n{GREEN}Обработанное регулярное выражение:{BASE}\n{regex}\n")

        return regex


    def checkRegex(regex: str) -> None:
        alphabet = ALPHABET + "()*|"
        for symbol in regex:
            if symbol not in alphabet:
                raise ValueError(f"Недопустимый символ для регулярного выражения
                ↪  '{symbol}'")

        openBracketsCount = 0
        stack = Stack()
        lettersBetween = 0
        for symbol in regex:
            if symbol == '(':
                openBracketsCount += 1
                stack.push(lettersBetween + 1)
                lettersBetween = 0
            elif symbol == ')':
                if openBracketsCount > 0:
                    openBracketsCount -= 1
                else:
                    raise ValueError("Неверная постановка скобок в регулярном
                    ↪  выражении")

                # в скобках должно быть хотя бы одно выражение
                if lettersBetween < 2:
                    raise ValueError("Неверная постановка скобок в регулярном
                    ↪  выражении")
                else:
                    lettersBetween = stack.pop()
            elif symbol != '|':
                lettersBetween += 1

        if openBracketsCount > 0:
            raise ValueError("Не все скобки в регулярном выражении были закрыты")

```

Продолжение листинга 3.2

```
53      lenRegex = len(regex)
54      for i in range(lenRegex):
55          if regex[i] == '|' and (
56              i == 0 or \
57              i == lenRegex - 1 or \
58              regex[i - 1] in ['|', '('] or \
59              regex[i + 1] in ['|', '*', ')']
60          ):
61              raise ValueError("Недопустимое расположение символа '|'")
62
63          if regex[i] == '*' and (
64              i == 0 or \
65              regex[i - 1] in ['|', '*', '(']
66          ):
67              raise ValueError("Недопустимое расположение символа '*'")
68
69
70  def convertToDesiredFormat(regex: str):
71      resRegex = ""
72      lenRegex = len(regex)
73      for i in range(lenRegex):
74          resRegex += regex[i]
75          if regex[i] in ALPHABET + "*)" and \
76              i != lenRegex - 1 and \
77              regex[i + 1] not in ['|', '*', ')']:
78              resRegex += '.'
79
80      # учет приоритета оператора '*'
81      i = 1
82      while i < len(resRegex):
83          if resRegex[i] == "*":
84              if resRegex[i - 1] != ")":
85                  resRegex = f"{resRegex[:i - 1]}({resRegex[i - 1:i +
                    ↪  1]}){resRegex[i + 1:]}"
86              else:
87                  openingBracketIndex = findOpeningBracketIndex(resRegex, i - 1)
88                  resRegex =
                    ↪  f"{resRegex[:openingBracketIndex]}({resRegex[openingBracketIndex::i
                    ↪  + 1]}){resRegex[i + 1:]}"
89              i += 2
90          i += 1
91
92      return resRegex + ".#"
93
94
```

Продолжение листинга 3.2

```
95  def findOpeningBracketIndex(regex: str, closingBracketIndex: int) -> int:
96      regex = regex[:closingBracketIndex][::-1]
97      closingBracketsCount = 0
98      openingBracketIndex = 0
99      for i in range(len(regex)):
100         if regex[i] == ')':
101             closingBracketsCount += 1
102         elif regex[i] == '(':
103             if closingBracketsCount > 0:
104                 closingBracketsCount -= 1
105             else:
106                 openingBracketIndex = i
107                 break
108
109     return closingBracketIndex - openingBracketIndex - 1
110
111
112 def findClosingBracketIndex(regex: str, openingBracketIndex: int) -> int:
113     regex = regex[openingBracketIndex + 1:]
114     openBracketsCount = 0
115     closingBracketIndex = 0
116     for i in range(len(regex)):
117         if regex[i] == '(':
118             openBracketsCount += 1
119         elif regex[i] == ')':
120             if openBracketsCount > 0:
121                 openBracketsCount -= 1
122             else:
123                 closingBracketIndex = i
124                 break
125
126     return openingBracketIndex + closingBracketIndex + 1
```

Листинг 3.3 — Модуль для работы с синтаксическим деревом регулярного выражения

```
1  import graphviz
2  from pythonds.basic.stack import Stack
3  from color import *
4  from regularExpression import findClosingBracketIndex
5
6
7  class Node:
8      def __init__(self, leftNode=None, rightNode=None) -> None:
9          self.nodeNumber = None
```

Продолжение листинга 3.3

```
10            self.letterNumber = None
11            self.value = None
12            self.leftChild = leftNode
13            self.rightChild = rightNode
14
15            self.nullable = None
16            self.firstpos = set()
17            self.lastpos = set()
18
19
20    class ParseTree():
21        def __init__(self, regex: str) -> None:
22            self.followpos = dict()
23            self.letterNumbers = dict()
24            self.root = self.__buildTree(regex)
25
26        def printTree(self) -> None:
27            print(f"{GREEN}Синтаксическое дерево для регулярного выражения:{BASE}")
28            self.__printNode(self.root)
29            print("\n")
30
31        def buildGraph(self, view: bool = False) -> None:
32            dot = graphviz.Digraph(
33                comment='Синтаксическое дерево для регулярного выражения'
34            )
35            self.__addNodeToGraph(self.root, dot)
36            dot.render('../docs/parse-tree.gv', view=view)
37
38        def __buildTree(self, regex: str) -> Node:
39            root, _, _ = self.__buildTreeRecursion(
40                regex=regex,
41                nodeNumber=0,
42                letterNumber=0
43            )
44            if root.value is None:
45                root = root.leftChild
46
47            return root
48
49        def __buildTreeRecursion(
50                self,
51                regex: str,
52                nodeNumber: int,
53                letterNumber: int,
54            ) -> list[Node, int, int]:
```

Продолжение листинга 3.3

```
55          stackNode = Stack()
56          node = Node()
57
58          i = 0
59          while i < len(regex):
60              symbol = regex[i]
61              if stackNode.isEmpty():
62                  root = Node(leftNode=node)
63                  stackNode.push(root)
64
65              if symbol == '(':
66                  closingBracketIndex = findClosingBracketIndex(regex, i)
67                  subtreeRoot, nodeCount, letterCount = self.__buildTreeRecursion(
68                      regex=regex[i + 1: closingBracketIndex],
69                      nodeNumber=nodeNumber,
70                      letterNumber=letterNumber
71                  )
72                  if subtreeRoot.value is None:
73                      subtreeRoot = subtreeRoot.leftChild
74                  node.leftChild = subtreeRoot.leftChild
75                  node.rightChild = subtreeRoot.rightChild
76                  node.value = subtreeRoot.value
77                  node.nodeNumber = subtreeRoot.nodeNumber
78                  nodeNumber = nodeCount
79                  letterNumber = letterCount
80                  i = closingBracketIndex
81                  node = stackNode.pop()
82
83              elif symbol not in ['.', '|', '*', ')']:
84                  nodeNumber += 1
85                  letterNumber += 1
86                  node.nodeNumber = nodeNumber
87                  node.letterNumber = letterNumber
88                  node.value = symbol
89                  self.letterNumbers[letterNumber] = symbol
90                  self.followpos[letterNumber] = set()
91                  node = stackNode.pop()
92
93              elif symbol in ['.', '|']:
94                  if node.value is not None:
95                      node = stackNode.pop()
96                  nodeNumber += 1
97                  node.nodeNumber = nodeNumber
98                  node.value = symbol
99                  node.rightChild = Node()
```

Продолжение листинга 3.3

```
100                 stackNode.push(node)
101                 node = node.rightChild
102
103             elif symbol == '*':
104                 if node.value is not None:
105                     node = stackNode.pop()
106                 nodeNumber += 1
107                 node.nodeNumber = nodeNumber
108                 node.value = symbol
109                 node.nullable = True
110
111             i += 1
112
113         return root, nodeNumber, letterNumber
114
115     def __printNode(self, node: Node, end: str = ' ') -> None:
116         if node is not None:
117             if node.leftChild:
118                 print('(', end=end)
119                 self.__printNode(node.leftChild)
120
121             print(node.value, end=end)
122
123             if node.rightChild:
124                 self.__printNode(node.rightChild)
125                 print(')', end=end)
126             elif node.leftChild: # для оператора '*'
127                 print(')', end=end)
128
129     def __addNodeToGraph(self, node: Node, dot: graphviz.Digraph) -> None:
130         if node is not None:
131             if node.leftChild:
132                 self.__addNodeToGraph(node.leftChild, dot)
133                 dot.edge(str(node.nodeNumber), str(node.leftChild.nodeNumber))
134
135             dot.node(
136                 name=str(node.nodeNumber),
137                 label=f"{node.value}{f", {node.letterNumber}" if node.letterNumber
                    ↪  else ""}"
138             )
139
140             if node.rightChild:
141                 self.__addNodeToGraph(node.rightChild, dot)
142                 dot.edge(str(node.nodeNumber), str(node.rightChild.nodeNumber))
```

Листинг 3.4 — Модуль для работы с ДКА

```
import graphviz
from color import *
from parseTree import ParseTree, Node


class DFA():
    def __init__(self, parseTree: ParseTree):
        self.root = parseTree.root
        self.followpos = parseTree.followpos
        self.letterNumbers = parseTree.letterNumbers

        self.__completeTree(self.root)
        self.initialState = self.__convertSetToString(self.root.firstpos)
        self.dStates = self.__findDStates()
        self.finalStates = self.__findFinalStates()

    def printFirstposLastpos(self) -> None:
        print(f"{GREEN}Значения функций firstpos и lastpos в узлах синтаксического
        ↪  дерева для регулярного выражения:{BASE}")
        self.__printNode(self.root)
        print("\n")

    def printFollowpos(self) -> None:
        print(f"{GREEN}Ориентированный граф для функции followpos:{BASE}")
        for key, value in self.followpos.items():
            print(f"{key}: {value}")
        print()

    def printDFA(self) -> None:
        print(f"{GREEN}ДКА для регулярного выражения:{BASE}")
        for key, value in self.dStates.items():
            print(f"{key}: {value}")
        print()

    def buildFirstposLastposGraph(self, view: bool = False) -> None:
        dot = graphviz.Digraph(
            comment='Значения функций firstpos и lastpos в узлах синтаксического
            ↪  дерева для регулярного выражения'
        )
        self.__addNodeToGraph(self.root, dot)
        dot.render('../docs/firstpos-lastpos.gv', view=view)

    def buildFollowposGraph(self, view: bool = False) -> None:
        dot = graphviz.Digraph(
            comment='Ориентированный граф для функции followpos'
        )
```

Продолжение листинга 3.4

```
45          for i in self.followpos:
46              dot.node(str(i))
47              for j in self.followpos[i]:
48                  dot.edge(str(i), str(j))
49
50          dot.render('../docs/followpos.gv', view=view)
51
52      def buildDFAGraph(self, view: bool = False) -> None:
53          dot = graphviz.Digraph(
54              comment='ДКА для регулярного выражения'
55          )
56          dot.node("", peripheries="0")
57          dot.edge("", self.initialState, label="start")
58          for state in self.dStates.keys():
59              if state in self.finalStates:
60                  linesCount = '2'
61              else:
62                  linesCount = '1'
63
64              dot.node(state, peripheries=linesCount)
65              for key, value in self.dStates[state].items():
66                  dot.edge(state, value, label=key, constraint='true')
67
68          dot.render('../docs/dfa.gv', view=view)
69
70      def __printNode(self, node: Node, end: str = ' ') -> None:
71          if node is not None:
72              if node.leftChild:
73                  print('(', end=end)
74                  self.__printNode(node.leftChild)
75
76              print(f"{node.firstpos} {node.value} {node.lastpos}", end=end)
77
78              if node.rightChild:
79                  self.__printNode(node.rightChild)
80                  print(')', end=end)
81              elif node.leftChild: # для оператора *
82                  print(')', end=end)
83
84      def __completeTree(self, node: Node) -> None:
85          if node is not None:
86              if node.leftChild:
87                  self.__completeTree(node.leftChild)
88              if node.rightChild:
89                  self.__completeTree(node.rightChild)
```

Продолжение листинга 3.4

```
90
91              node.nullable = self.__calcNullable(node)
92              node.firstpos = self.__calcFirstpos(node)
93              node.lastpos = self.__calcLastpos(node)
94
95              if node.value == '.':
96                  for i in node.leftChild.lastpos:
97                      for j in node.rightChild.firstpos:
98                          self.followpos[i].add(j)
99              elif node.value == '*':
100                 for i in node.lastpos:
101                     for j in node.firstpos:
102                         self.followpos[i].add(j)
103
104     def __calcNullable(self, node: Node) -> bool:
105         if node.value == '|':
106             nullable = \
107                 node.leftChild.nullable or \
108                 node.rightChild.nullable
109         elif node.value == '.':
110             nullable = \
111                 node.leftChild.nullable and \
112                 node.rightChild.nullable
113         elif node.value == '*':
114             nullable = True
115         else:
116             nullable = False
117
118         return nullable
119
120     def __calcFirstpos(self, node: Node) -> set:
121         if node.value == '|':
122             firstpos = node.leftChild.firstpos.union(node.rightChild.firstpos)
123         elif node.value == '.':
124             firstpos = \
125                 node.leftChild.firstpos.union(node.rightChild.firstpos) \
126                 if node.leftChild.nullable else node.leftChild.firstpos
127         elif node.value == '*':
128             firstpos = node.leftChild.firstpos
129         else:
130             firstpos = {node.letterNumber}
131
132         return firstpos
133
134     def __calcLastpos(self, node: Node) -> set:
```

Продолжение листинга 3.4

```
135             if node.value == '|':
136                 lastpos = node.leftChild.lastpos.union(node.rightChild.lastpos)
137             elif node.value == '.':
138                 lastpos = \
139                     node.leftChild.lastpos.union(node.rightChild.lastpos) \
140                     if node.rightChild.nullable else node.rightChild.lastpos
141             elif node.value == '*':
142                 lastpos = node.leftChild.lastpos
143             else:
144                 lastpos = {node.letterNumber}
145 
146             return lastpos
147 
148         def __addNodeToGraph(self, node: Node, dot: graphviz.Digraph) -> None:
149             if node is not None:
150                 if node.leftChild:
151                     self.__addNodeToGraph(node.leftChild, dot)
152                     dot.edge(str(node.nodeNumber), str(node.leftChild.nodeNumber))
153 
154                 dot.node(
155                     name=str(node.nodeNumber),
156                     label=f"{node.firstpos}  {node.value}{f", {node.letterNumber}" if
                        ↪  node.letterNumber else ""}  {node.lastpos}"
157                 )
158 
159                 if node.rightChild:
160                     self.__addNodeToGraph(node.rightChild, dot)
161                     dot.edge(str(node.nodeNumber), str(node.rightChild.nodeNumber))
162 
163         def __findDStates(self) -> dict:
164             dStates = {}
165             newStates = [self.initialState]
166             while len(newStates) > 0:
167                 state = newStates.pop()
168                 dStates[state] = {}
169                 for i in state.split(','):
170                     i = int(i)
171                     if self.letterNumbers[i] == '#':
172                         continue
173                     elif not dStates[state].get(self.letterNumbers[i]):
174                         dStates[state][self.letterNumbers[i]] = self.followpos[i]
175                     else:
176                         dStates[state][self.letterNumbers[i]] =
                            ↪  self.followpos[i].union(
177                             dStates[state][self.letterNumbers[i]]
```

Продолжение листинга 3.4

```
                    )

            for letter, nextState in dStates[state].items():
                nextState = self.__convertSetToString(nextState)
                dStates[state][letter] = nextState
                if nextState not in dStates and nextState not in newStates:
                    newStates.append(nextState)

        return dStates

    def __findFinalStates(self) -> list:
        finalStates = []
        for state in self.dStates.keys():
            for i in state.split(','):
                if int(i) == self.root.rightChild.letterNumber:
                    finalStates.append(state)
                    break

        return finalStates

    def __convertSetToString(self, item: set) -> str:
        item = list(item)
        item.sort()
        itemStr = ""
        for i in item:
            itemStr += f"{i},"

        return itemStr[:-1]
```

Листинг 3.5 — Модуль для работы с минимизированным ДКА

```
import graphviz
from color import *
from dfa import DFA


class MinDFA():
    def __init__(self, dfa: DFA, alphabet: str):
        self.dStates = dfa.dStates

        self.groupList = self.__minimizeNumberOfStates(dfa.finalStates.copy(),
        ↪  alphabet)
        self.initialState = self.__findInitialState(dfa.initialState)
        self.finalStates = self.__findFinalStates(dfa.finalStates)
        self.minDstates = self.__findMinDstates()
```

Продолжение листинга 3.5

```
14
15      def printGroupList(self) -> None:
16          print(f"{GREEN}Группы состояний, полученные после минимизации ДКА
            ↪  алгоритмом Хопкрофта:{BASE}")
17          for i in range(len(self.groupList)):
18              print(f"{i + 1}: {self.groupList[i]}")
19          print()
20
21      def printMinDFA(self) -> None:
22          print(f"{GREEN}Минимизированный ДКА алгоритмом Хопкрофта:{BASE}")
23          for key, value in self.minDstates.items():
24              print(f"{key}: {value}")
25
26      def buildMinDFAGraph(self, view: bool = False) -> None:
27          dot = graphviz.Digraph(
28              comment='Минимизированный ДКА алгоритмом Хопкрофта'
29          )
30          dot.node("", peripheries="0")
31          dot.edge("", self.initialState, label="start")
32
33          for state in self.minDstates.keys():
34              if state in self.finalStates:
35                  linesCount = '2'
36              else:
37                  linesCount = '1'
38
39              dot.node(state, peripheries=linesCount)
40              for key, value in self.minDstates[state].items():
41                  dot.edge(state, value, label=key, constraint='true')
42
43          dot.render('../docs/min-dfa.gv', view=view)
44
45      def __minimizeNumberOfStates(self, finalStates: list, alphabet: str) -> list:
46          nonFinalStates = []
47          for state in self.dStates.keys():
48              if state not in finalStates:
49                  nonFinalStates.append(state)
50
51          if len(nonFinalStates):
52              groupList = [nonFinalStates, finalStates]
53          else:
54              groupList = [finalStates]
55
56          groupListLen = len(groupList)
57          while True:
```

Продолжение листинга 3.5

```
58              for group in groupList:
59                  newGroup = []
60                  groupDict = {}
61                  for state in group:
62                      for letter in alphabet:
63                          nextState = self.dStates[state].get(letter)
64                          firstGroupIndex = groupDict.get(letter)
65                          groupIndex = \
66                              self.__getGroupIndexOfState(nextState, groupList)
67                          if firstGroupIndex is None:
68                              groupDict[letter] = groupIndex
69                          elif firstGroupIndex != groupIndex:
70                              newGroup.append(state)
71                              break
72
73                  if len(newGroup):
74                      groupList.append(newGroup)
75                      for state in newGroup:
76                          group.remove(state)
77
78              if groupListLen != len(groupList):
79                  groupListLen = len(groupList)
80              else:
81                  break
82
83          return groupList
84
85      def __findInitialState(self, dfaInitialState: str) -> str:
86          for group in self.groupList:
87              if dfaInitialState in group:
88                  return group[0]
89
90      def __findFinalStates(self, dfaFinalStates: list) -> list:
91          finalStates = []
92          for group in self.groupList:
93              state = group[0]
94              if state in dfaFinalStates:
95                  finalStates.append(state)
96
97          return finalStates
98
99      def __findMinDstates(self) -> dict:
100         minDstates = {}
101         for group in self.groupList:
102             state = group[0]
```

Продолжение листинга 3.5

```
103                 minDstates[state] = {}
104                 for letter, nextState in self.dStates[state].items():
105                     groupIndex = self.__getGroupIndexOfState(nextState,
                        ↪  self.groupList)
106                     minDstates[state][letter] = self.groupList[groupIndex][0]
107
108             return minDstates
109
110         def __getGroupIndexOfState(self, nextState: str | None, groupList: list) ->
            ↪  int:
111             if nextState is None:
112                 return -1
113
114             for i in range(len(groupList)):
115                 for state in groupList[i]:
116                     if state == nextState:
117                         return i
```

Листинг 3.6 — Модуль обработки входных цепочек

```
1   from color import *
2   from minDfa import MinDFA
3
4
5   def checkChain(chain: str, minDfa: MinDFA) -> bool:
6       state = minDfa.initialState
7       for symbol in chain:
8           nextState = minDfa.minDstates[state].get(symbol)
9           if nextState:
10              print(f"{symbol}: {state} ---> {nextState}")
11              state = nextState
12          else:
13              print(f"{symbol}: {state} ---> None")
14              return False
15
16      if state not in minDfa.finalStates:
17          print(f"Состояние '{state}' не является конечным")
18          return False
19
20      return True
21
22
23  def inputChainCheckCorrespondence(regex: str, minDFA: MinDFA) -> None:
24      chain = input(f"\nВведите входную цепочку, которую хотите проверить на
        ↪  соответсвие регулярному выражению '{regex}': ")
```

Продолжение листинга 3.6

```
25      if checkChain(chain, minDFA):
26          print(f"\nВходная цепочка '{chain}' {GREEN}соответствует{BASE} регулярному
            ↪  выражению '{regex}'.")
27      else:
28          print(f"\nВходная цепочка '{chain}' {RED}не соответствует{BASE}
            ↪  регулярному выражению '{regex}'.")
```

Листинг 3.7 — Модуль с вариантами цветов при выводе сообщений в консоль

```
1  BASE   = "\x1B[0m"
2  GREEN  = "\x1B[32m"
3  RED    = "\x1B[31m"
4  YELLOW = "\x1B[33m"
5  BLUE   = "\x1B[34m"
6  PURPLE = "\x1B[35m"
```