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

по лабораторной работе № 1

по курсу «Конструирование компиляторов»

на тему: «Распознавание цепочек регулярного языка»

Вариант № 6

Студент ИУ7-22М (Группа) _____ (Подпись, дата) И. А. Малышев (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.

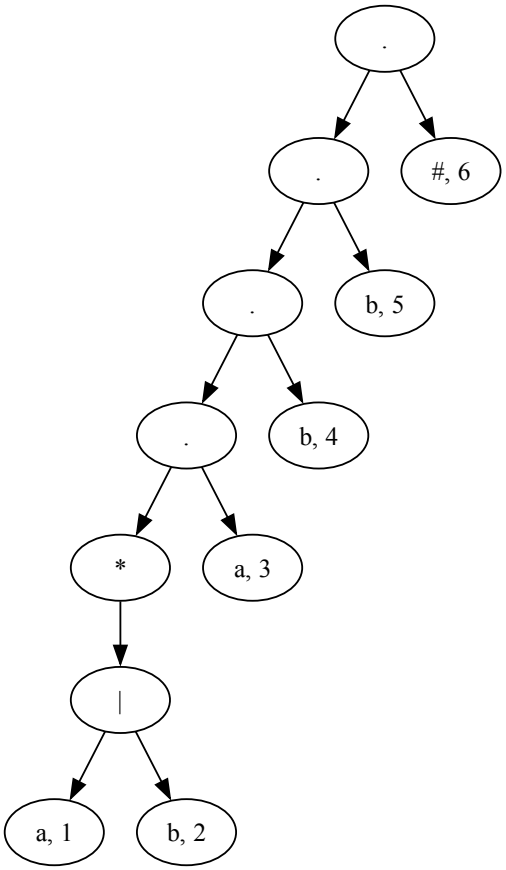


Рисунок 1.1 – Синтаксическое дерево для регулярного выражения

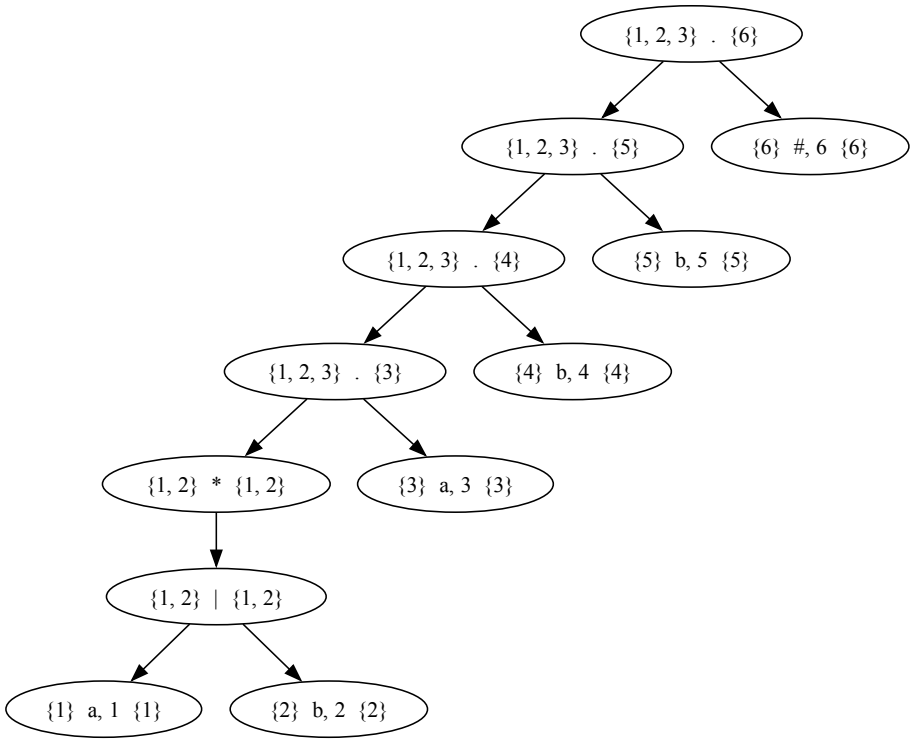


Рисунок 1.2 – Значения функций firstpos и lastpos в узлах синтаксического дерева

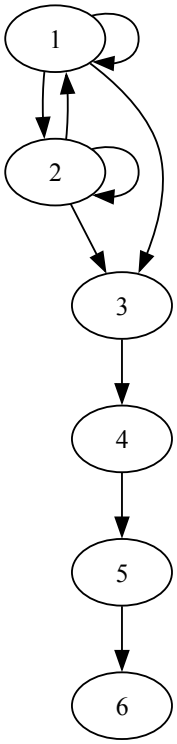


Рисунок 1.3 – Ориентированный граф для функции followpos

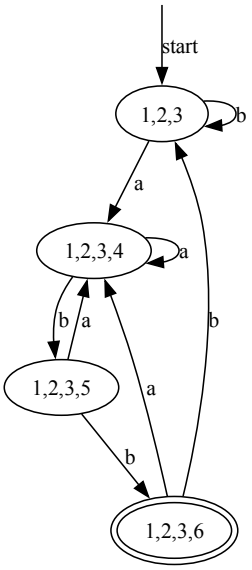


Рисунок 1.4 – ДКА для регулярного выражения

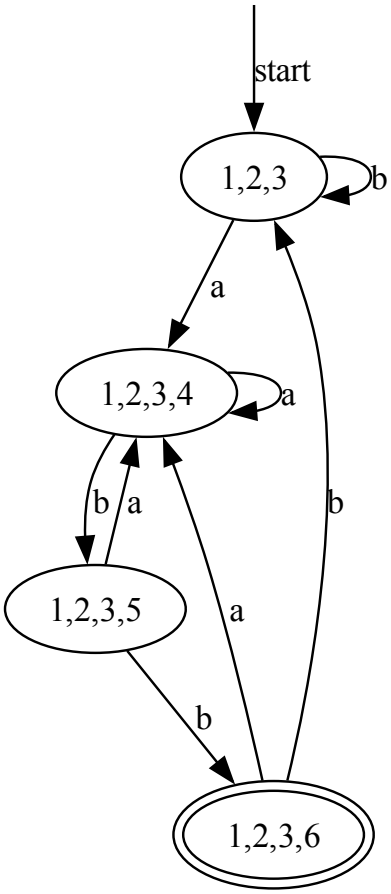


Рисунок 1.5 – Минимизированный ДКА алгоритмом Хопкрофта

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)

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

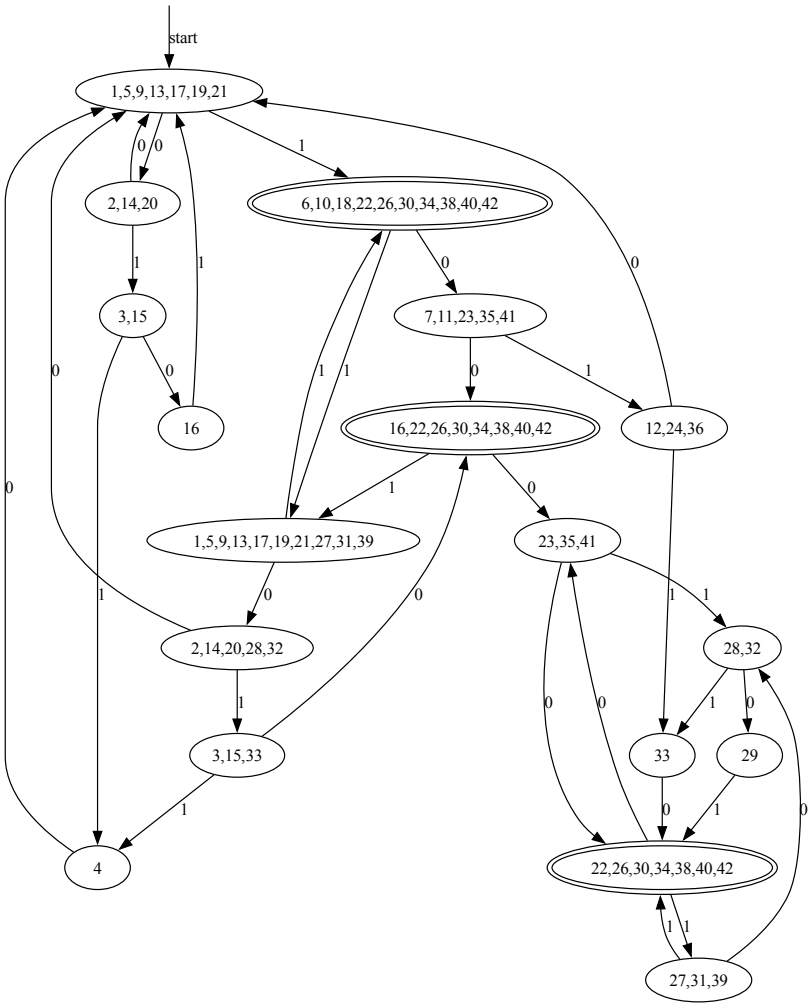
(b)

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

(c)

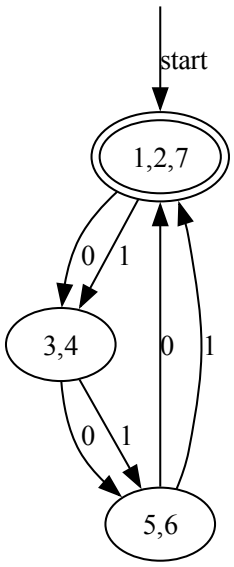
$$
\begin{aligned}
S &\to A \\
S &\to 1S \\
S &\to \epsilon \\
A &\to 0A \\
A &\to 00S
\end{aligned}
\tag{2.3}
$$

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



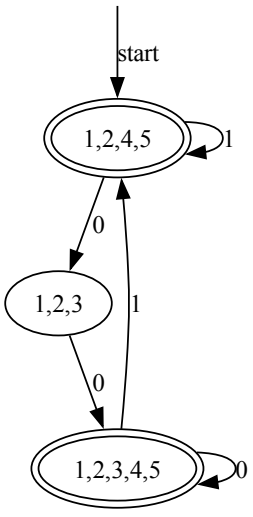
(a)

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

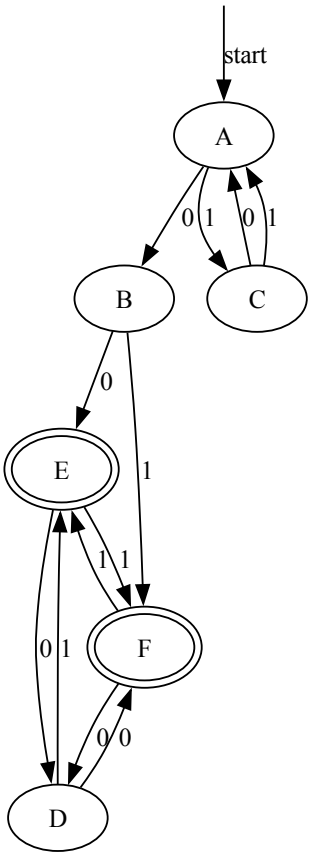


Рисунок 2.5 – ДКА для языка, определяемого автоматом M

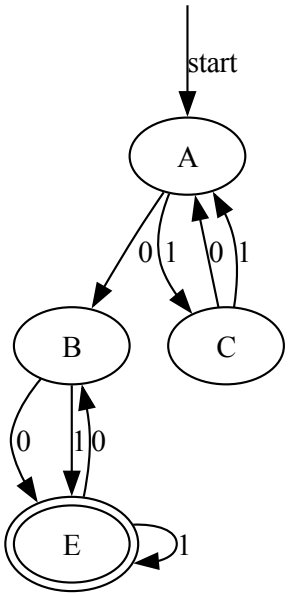


Рисунок 2.6 – Минимизированный ДКА для языка, определяемого автоматом M

3 Текст программы

В листингах 3.1–3.6 представлен код программы.

Листинг 3.1 – Основной модуль программы

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


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

    0. Выход.\n
    Выбор: """


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"\nВведите регулярное выражение: ")
# regex = "(a|b)*abb"
# regex = "((abba)|(baab)|(baba)|(abab)|(bb)|(aa))*"
# regex =
   "((0110)|(1001)|(1010)|(0101)|(11)|(00))*1((0110)|(1001)|(1010)
# regex = "((0|1)(0|1)(0|1))*"
# regex = "((0*00)|1)*"
convertedRegex = convertRegexToDesiredFormat(regex)
if convertedRegex is None:
    return

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


ALPHABET = "qwertyuiopasdfghjklzxcvbnm0123456789"


def convertRegexToDesiredFormat(regex: str) -> str | None:
    regex = regex.replace(" ", "").lower()
    try:
        checkRegex(regex)
    except ValueError as exc:
        print(f"\n{exc}\n")
        return None

    regex = convertToDesiredFormat(regex)
    print(f"\nОбработанное регулярное выражение:\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")

    lenRegex = len(regex)
    for i in range(lenRegex):
        if regex[i] == '|' and (
            i == 0 or \
            i == lenRegex - 1 or \
            regex[i - 1] in ['|', '('] or \
            regex[i + 1] in ['|', '*', ')']
        ):
            raise ValueError("Недопустимое расположение символа
               '|'")

        if regex[i] == '*' and (
            i == 0 or \
            regex[i - 1] in ['|', '*', '(']
        ):
            raise ValueError("Недопустимое расположение символа
               '*'")


def convertToDesiredFormat(regex: str):
    resRegex = ""
    lenRegex = len(regex)
    for i in range(lenRegex):
        resRegex += regex[i]
```

```
        if regex[i] in ALPHABET + "*)" and \
            i != lenRegex - 1 and \
            regex[i + 1] not in ['|', '*', ')']:
            resRegex += '.'

    # учет приоритета оператора '*'
    i = 1
    while i < len(resRegex):
        if resRegex[i] == "*":
            if resRegex[i - 1] != ")":
                resRegex = f"{resRegex[:i - 1]}({resRegex[i -
                    1:i + 1]}){resRegex[i + 1:]}"
            else:
                openingBracketIndex =
                    findOpeningBracketIndex(resRegex, i - 1)
                resRegex =
                    f"{resRegex[:openingBracketIndex]}({resRegex[openi
                    + 1]}){resRegex[i + 1:]}"
            i += 2
        i += 1

    return resRegex + ".#"


def findOpeningBracketIndex(regex: str, closingBracketIndex:
   int) -> int:
    regex = regex[:closingBracketIndex][::-1]
    closingBracketsCount = 0
    openingBracketIndex = 0
    for i in range(len(regex)):
        if regex[i] == ')':
            closingBracketsCount += 1
        elif regex[i] == '(':
            if closingBracketsCount > 0:
                closingBracketsCount -= 1
            else:
                openingBracketIndex = i
                break

    return closingBracketIndex - openingBracketIndex - 1
```

```
def findClosingBracketIndex(regex: str, openingBracketIndex:
   int) -> int:
    regex = regex[openingBracketIndex + 1:]
    openBracketsCount = 0
    closingBracketIndex = 0
    for i in range(len(regex)):
        if regex[i] == '(':
            openBracketsCount += 1
        elif regex[i] == ')':
            if openBracketsCount > 0:
                openBracketsCount -= 1
            else:
                closingBracketIndex = i
                break

    return openingBracketIndex + closingBracketIndex + 1
```

Листинг 3.3 – Модуль для работы с синтаксическим деревом регулярного выражения

```
import graphviz
from pythonds.basic.stack import Stack
from regularExpression import findClosingBracketIndex


class Node:
    def __init__(self, leftNode=None, rightNode=None) -> None:
        self.nodeNumber = None
        self.letterNumber = None
        self.value = None
        self.leftChild = leftNode
        self.rightChild = rightNode

        self.nullable = None
        self.firstpos = set()
        self.lastpos = set()


class ParseTree():
    def __init__(self, regex: str) -> None:
        self.followpos = dict()
        self.letterNumbers = dict()
```

```
        self.root = self.__buildTree(regex)

    def printTree(self) -> None:
        print(f"Синтаксическое дерево для регулярного
           выражения:")
        self.__printNode(self.root)
        print("\n")

    def buildGraph(self, view: bool = False) -> None:
        dot = graphviz.Digraph(
            comment='Синтаксическое дерево для регулярного
               выражения'
        )
        self.__addNodeToGraph(self.root, dot)
        dot.render('../docs/parse-tree.gv', view=view)

    def __buildTree(self, regex: str) -> Node:
        root, _, _ = self.__buildTreeRecursion(
            regex=regex,
            nodeNumber=0,
            letterNumber=0
        )
        if root.value is None:
            root = root.leftChild

        return root

    def __buildTreeRecursion(
            self,
            regex: str,
            nodeNumber: int,
            letterNumber: int,
        ) -> list[Node, int, int]:
        stackNode = Stack()
        node = Node()

        i = 0
        while i < len(regex):
            symbol = regex[i]
            if stackNode.isEmpty():
                root = Node(leftNode=node)
```

```
            stackNode.push(root)

        if symbol == '(':
            closingBracketIndex =
               findClosingBracketIndex(regex, i)
            subtreeRoot, nodeCount, letterCount =
               self.__buildTreeRecursion(
                regex=regex[i + 1: closingBracketIndex],
                nodeNumber=nodeNumber,
                letterNumber=letterNumber
            )
            if subtreeRoot.value is None:
                subtreeRoot = subtreeRoot.leftChild
            node.leftChild = subtreeRoot.leftChild
            node.rightChild = subtreeRoot.rightChild
            node.value = subtreeRoot.value
            node.nodeNumber = subtreeRoot.nodeNumber
            nodeNumber = nodeCount
            letterNumber = letterCount
            i = closingBracketIndex
            node = stackNode.pop()

        elif symbol not in ['.', '|', '*', ')']:
            nodeNumber += 1
            letterNumber += 1
            node.nodeNumber = nodeNumber
            node.letterNumber = letterNumber
            node.value = symbol
            self.letterNumbers[letterNumber] = symbol
            self.followpos[letterNumber] = set()
            node = stackNode.pop()

        elif symbol in ['.', '|']:
            if node.value is not None:
                node = stackNode.pop()
            nodeNumber += 1
            node.nodeNumber = nodeNumber
            node.value = symbol
            node.rightChild = Node()
            stackNode.push(node)
            node = node.rightChild
```

```
            elif symbol == '*':
                if node.value is not None:
                    node = stackNode.pop()
                nodeNumber += 1
                node.nodeNumber = nodeNumber
                node.value = symbol
                node.nullable = True

            i += 1

        return root, nodeNumber, letterNumber

    def __printNode(self, node: Node, end: str = ' ') -> None:
        if node is not None:
            if node.leftChild:
                print('(', end=end)
                self.__printNode(node.leftChild)

            print(node.value, end=end)

            if node.rightChild:
                self.__printNode(node.rightChild)
                print(')', end=end)
            elif node.leftChild: # для оператора '*'
                print(')', end=end)

    def __addNodeToGraph(self, node: Node, dot:
       graphviz.Digraph) -> None:
        if node is not None:
            if node.leftChild:
                self.__addNodeToGraph(node.leftChild, dot)
                dot.edge(str(node.nodeNumber),
                   str(node.leftChild.nodeNumber))

            dot.node(
                name=str(node.nodeNumber),
                label=f"{node.value}{f", {node.letterNumber}" if
                   node.letterNumber else ""}"
            )
```

```
            if node.rightChild:
                self.__addNodeToGraph(node.rightChild, dot)
                dot.edge(str(node.nodeNumber),
                   str(node.rightChild.nodeNumber))
```

Листинг 3.4 – Модуль для работы с ДКА

```
import graphviz
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()

        # self.initialState = "A"
        # self.dStates = {
        #     "A": {"0": "B", "1": "C"},
        #     "B": {"0": "E", "1": "F"},
        #     "C": {"0": "A", "1": "A"},
        #     "D": {"0": "F", "1": "E"},
        #     "E": {"0": "D", "1": "F"},
        #     "F": {"0": "D", "1": "E"},
        # }
        # self.finalStates = ["E", "F"]

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

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

```
            print(f"{key}: {value}")
        print()

    def printDFA(self) -> None:
        print(f"ДКА для регулярного выражения:")
        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'
        )
        for i in self.followpos:
            dot.node(str(i))
            for j in self.followpos[i]:
                dot.edge(str(i), str(j))

        dot.render('../docs/followpos.gv', view=view)

    def buildDFAGraph(self, view: bool = False) -> None:
        dot = graphviz.Digraph(
            comment='ДКА для регулярного выражения'
        )
        dot.node("", peripheries="0")
        dot.edge("", self.initialState, label="start")
        for state in self.dStates.keys():
            if state in self.finalStates:
                linesCount = '2'
            else:
                linesCount = '1'
```

```
            dot.node(state, peripheries=linesCount)
            for key, value in self.dStates[state].items():
                dot.edge(state, value, label=key,
                   constraint='true')

        dot.render('../docs/dfa.gv', view=view)

    def __printNode(self, node: Node, end: str = ' ') -> None:
        if node is not None:
            if node.leftChild:
                print('(', end=end)
                self.__printNode(node.leftChild)

            print(f"{node.firstpos} {node.value}
               {node.lastpos}", end=end)

            if node.rightChild:
                self.__printNode(node.rightChild)
                print(')', end=end)
            elif node.leftChild: # для оператора *
                print(')', end=end)

    def __completeTree(self, node: Node) -> None:
        if node is not None:
            if node.leftChild:
                self.__completeTree(node.leftChild)
            if node.rightChild:
                self.__completeTree(node.rightChild)

            node.nullable = self.__calcNullable(node)
            node.firstpos = self.__calcFirstpos(node)
            node.lastpos = self.__calcLastpos(node)

            if node.value == '.':
                for i in node.leftChild.lastpos:
                    for j in node.rightChild.firstpos:
                        self.followpos[i].add(j)
            elif node.value == '*':
                for i in node.lastpos:
                    for j in node.firstpos:
                        self.followpos[i].add(j)
```

```
    def __calcNullable(self, node: Node) -> bool:
        if node.value == '|':
            nullable = \
                node.leftChild.nullable or \
                node.rightChild.nullable
        elif node.value == '.':
            nullable = \
                node.leftChild.nullable and \
                node.rightChild.nullable
        elif node.value == '*':
            nullable = True
        else:
            nullable = False

        return nullable

    def __calcFirstpos(self, node: Node) -> set:
        if node.value == '|':
            firstpos =
               node.leftChild.firstpos.union(node.rightChild.firstpos)
        elif node.value == '.':
            firstpos = \
                node.leftChild.firstpos.union(node.rightChild.firstpos
                   \
                if node.leftChild.nullable else
                   node.leftChild.firstpos
        elif node.value == '*':
            firstpos = node.leftChild.firstpos
        else:
            firstpos = {node.letterNumber}

        return firstpos

    def __calcLastpos(self, node: Node) -> set:
        if node.value == '|':
            lastpos =
               node.leftChild.lastpos.union(node.rightChild.lastpos)
        elif node.value == '.':
            lastpos = \
                node.leftChild.lastpos.union(node.rightChild.lastpos)
```

```
                \
            if node.rightChild.nullable else
                node.rightChild.lastpos
        elif node.value == '*':
            lastpos = node.leftChild.lastpos
        else:
            lastpos = {node.letterNumber}

        return lastpos

    def __addNodeToGraph(self, node: Node, dot:
       graphviz.Digraph) -> None:
        if node is not None:
            if node.leftChild:
                self.__addNodeToGraph(node.leftChild, dot)
                dot.edge(str(node.nodeNumber),
                    str(node.leftChild.nodeNumber))

            dot.node(
                name=str(node.nodeNumber),
                label=f"{node.firstpos}  {node.value}{f",
                    {node.letterNumber}" if node.letterNumber
                    else ""}  {node.lastpos}"
            )

            if node.rightChild:
                self.__addNodeToGraph(node.rightChild, dot)
                dot.edge(str(node.nodeNumber),
                    str(node.rightChild.nodeNumber))

    def __findDStates(self) -> dict:
        dStates = {}
        newStates = [self.initialState]
        while len(newStates) > 0:
            state = newStates.pop()
            dStates[state] = {}
            for i in state.split(','):
                i = int(i)
                if self.letterNumbers[i] == '#':
                    continue
                elif not
```

```
                    dStates[state].get(self.letterNumbers[i]):
                        dStates[state][self.letterNumbers[i]] =
                            self.followpos[i]
                else:
                    dStates[state][self.letterNumbers[i]] =
                        self.followpos[i].union(
                            dStates[state][self.letterNumbers[i]]
                        )

            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 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()

    def printGroupList(self) -> None:
        print(f"Группы состояний, полученные после минимизации
           ДКА алгоритмом Хопкрофта:")
        for i in range(len(self.groupList)):
            print(f"{i + 1}: {self.groupList[i]}")
        print()

    def printMinDFA(self) -> None:
        print(f"Минимизированный ДКА алгоритмом Хопкрофта:")
        for key, value in self.minDstates.items():
            print(f"{key}: {value}")

    def buildMinDFAGraph(self, view: bool = False) -> None:
        dot = graphviz.Digraph(
            comment='Минимизированный ДКА алгоритмом Хопкрофта'
        )
        dot.node("", peripheries="0")
        dot.edge("", self.initialState, label="start")

        for state in self.minDstates.keys():
            if state in self.finalStates:
                linesCount = '2'
            else:
                linesCount = '1'

            dot.node(state, peripheries=linesCount)
            for key, value in self.minDstates[state].items():
                dot.edge(state, value, label=key,
```

```
                constraint='true')

        dot.render('../docs/min-dfa.gv', view=view)

    def __minimizeNumberOfStates(self, finalStates: list,
       alphabet: str) -> list:
        nonFinalStates = []
        for state in self.dStates.keys():
            if state not in finalStates:
                nonFinalStates.append(state)

        if len(nonFinalStates):
            groupList = [nonFinalStates, finalStates]
        else:
            groupList = [finalStates]

        groupListLen = len(groupList)
        while True:
            for group in groupList:
                newGroup = []
                groupDict = {}
                for state in group:
                    for letter in alphabet:
                        nextState =
                           self.dStates[state].get(letter)
                        firstGroupIndex = groupDict.get(letter)
                        groupIndex = \
                            self.__getGroupIndexOfState(nextState,
                               groupList)
                        if firstGroupIndex is None:
                            groupDict[letter] = groupIndex
                        elif firstGroupIndex != groupIndex:
                            newGroup.append(state)
                            break

                if len(newGroup):
                    groupList.append(newGroup)
                    for state in newGroup:
                        group.remove(state)

            if groupListLen != len(groupList):
```

```
                groupListLen = len(groupList)
            else:
                break

        return groupList

    def __findInitialState(self, dfaInitialState: str) -> str:
        for group in self.groupList:
            if dfaInitialState in group:
                return group[0]

    def __findFinalStates(self, dfaFinalStates: list) -> list:
        finalStates = []
        for group in self.groupList:
            state = group[0]
            if state in dfaFinalStates:
                finalStates.append(state)

        return finalStates

    def __findMinDstates(self) -> dict:
        minDstates = {}
        for group in self.groupList:
            state = group[0]
            minDstates[state] = {}
            for letter, nextState in self.dStates[state].items():
                groupIndex =
                   self.__getGroupIndexOfState(nextState,
                   self.groupList)
                minDstates[state][letter] =
                   self.groupList[groupIndex][0]

        return minDstates

    def __getGroupIndexOfState(self, nextState: str | None,
       groupList: list) -> int:
        if nextState is None:
            return -1

        for i in range(len(groupList)):
            for state in groupList[i]:
```

```
                if state == nextState:
                    return i
```

Листинг 3.6 – Модуль обработки входных цепочек

```
from minDfa import MinDFA


def checkChain(chain: str, minDfa: MinDFA) -> bool:
    state = minDfa.initialState
    for symbol in chain:
        nextState = minDfa.minDstates[state].get(symbol)
        if nextState:
            print(f"{symbol}: {state} ---> {nextState}")
            state = nextState
        else:
            print(f"{symbol}: {state} ---> None")
            return False

    if state not in minDfa.finalStates:
        print(f"Состояние '{state}' не является конечным")
        return False

    return True


def inputChainCheckCorrespondence(regex: str, minDFA: MinDFA) ->
   None:
    chain = input(f"\nВведите входную цепочку, которую хотите
       проверить на соответсвие регулярному выражению '{regex}':
       ")
    if checkChain(chain, minDFA):
        print(f"\nВходная цепочка '{chain}' соответствует
           регулярному выражению '{regex}'.")
    else:
        print(f"\nВходная цепочка '{chain}' не соответствует
           регулярному выражению '{regex}'.")
```