

**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)**

**ОТЧЕТ**

По лабораторной работе №1

По курсу «Конструирование компиляторов»

Тема: «Распознавание цепочек регулярного языка»

Вариант 2

Студент:

Апальков Ф.С.

Группа:

ИУ7-23М

Преподаватель:

Ступников А.А.

Москва 2022

**1. Цель и задачи работы**

Цель работы: приобретение практических навыков реализации важнейших элементов лексических анализаторов

на 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, право линейным языком, конечно автоматным языком и недетерминированным конечно-автоматным языком.

3) Разработать, тестировать и отладить программу распознавания цепочек регулярного или право линейного

языка в соответствии с предложенным вариантом грамматики

**Теоретическая часть**

Распознавание цепочек регулярного языка является центральной задачей лексического анализа, который образует первый этап процесса компиляции. На этом этапе символы, составляющие исходную программу, считываются и группируются в отдельные лексические элементы, называемые лексемами. Большую часть того, что происходит в течение лексического анализа можно моделировать с помощью конечных преобразователей, работающих последовательно или параллельно. Например, лексический анализатор может состоять из ряда последовательно соединенных конечных преобразователей. Первый преобразователь в этой цепи может устранять из исходной программы все несущественные пробелы, второй ликвидирует комментарии, третий ищет константы и т. д. Другая возможность - завести набор конечных преобразователей, каждый из которых ищет определенную лексическую конструкцию. Лексические анализаторы бывают по существу двух видов - прямые и непрямые. Можно показать как по регулярным выражениям, описывающим соответствующие лексемы, строятся анализаторы обоих видов. При непрямом лексическом анализе требуется, прочитав цепочку знаков, определить, появилась ли подцепочка, образующая некоторую конкретную лексему. Если множество возможных цепочек, которые могут образовывать эту лексему, обозначается, как это обычно бывает, регулярным выражением, то проблему построения непрямого лексического анализатора для данной лексемы можно представить себе как проблему реализации конечного преобразователя. Конечный преобразователь - это почти конечный автомат (распознаватель) в том смысле, что он читает вход, не производя выхода, пока не обнаружит присутствие лексемы данного типа (т. е. достигнет заключительного состояния). Тогда он сигнализирует о том, что эта лексема появилась, и выдает на выходе цепочку символов, образующих эту лексему. Проблема непрямого лексического анализа является, таким образом, по существу проблемой построения детерминированного конечного автомата (ДКА) по заданному регулярному выражению и его программной реализации. В простейшем случае сначала по регулярному выражению строят недетерминированный конечный автомат (НКА). Затем этот НКА превращают в ДКА, либо моделируют его работу, прослеживая параллельно всевозможные последовательности тактов. Пусть множество лексем данного типа обозначается регулярным выражением или задается в виде праволинейной грамматики. Во втором случае требуется преобразование праволинейной грамматики в регулярное выражение, которое выполняется за два шага: Шаг 1. Построение стандартной системы уравнений с регулярными коэффициентами по праволинейной грамматике. Шаг 2. Решение стандартной системы уравнений с регулярными коэффициентами. Построение НКА по регулярному выражению может выполняться либо с помощью алгоритма «Конструктор Томпсона», либо с помощью алгоритма построения НКА по расширенному регулярному выражению (см. Алгоритм 3.2. АУ1). Наконец, осуществляют детерминированное моделирование НКА для заданной входной цепочки знаков. При моделировании конечного автомата необходимо учитывать следующие обстоятельства. Работа автомата завершается, если обнаруживается хотя бы одна допускающая конфигурация, достижимая из начальной конфигурации автомата. Если входная цепочка построена синтаксически неправильно, то придется рассмотреть все возможные последовательности тактов автомата. Если исчерпаны все возможные последовательности тактов, а допустимая конфигурация не обнаружена, то надо выдать сообщение об ошибке. Внимание! При моделировании НКА или ДКА необходимо показать все конфигурации алгоритма

**Вариант**

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

1) По регулярному выражению строит НКА.

2) По НКА строит эквивалентный ему ДКА.

3) По ДКА строит эквивалентный ему КА, имеющий наименьшее возможное количество состояний. Указание. Воспользоваться алгоритмом, приведенным по адресу http://neerc.ifmo.ru/wiki/index.php?title=Минимизация\_ДКА,\_алгоритм\_за\_O(n%5E2)\_с\_построением\_пар\_разл ичимых\_состояний

4) Моделирует минимальный КА для входной цепочки из терминалов исходной грамматики.

**Текст программы**

**Main.py**

from graphConverter import \*

print('Введите регулярное выражение')

s = input()

rs = convertToPoland(s)

g = NKAEps(rs)

print('НКА с эпсилонами')

prettyPrint(g.createTable())

NKA(g)

LOF = g.getListOfNodes()

print('НКА без эпсилонов')

prettyPrint(g.createTable())

k = DKA(g)

print('ДКА')

prettyPrint(k.createTable())

rerere = k.getListOfNodes()

mg = minGraph(k)

print('Минимальный ДКА')

prettyPrint(mg.createTable())

print('сделаны автоматы')

while True:

    print('введите строку для проверки')

    psn = input()

    flag = sumulation(k, psn)

    if flag:

        print('Прошла')

    else:

        print('Не прошла')

**GarphCOnverter.py**

def getNoda(note):

    return list(note.values())[0]

def getSymbol(note):

    return list(note.keys())[0]

def getAllNodes(connection):

    buf = []

    for i in connection:

        buf.append(getNoda(i))

    return buf

class Graph():

    def \_\_init\_\_(self, startPoint, endPoint) -> None:

        self.Start = startPoint

        self.End = endPoint

        self.Finish = []

    def getListOfNodes(self):

        numeratedNodes = [self.Start]

        connectionsSymbols = []

        index = 0

        while True:

            connectedNodes = []

            countOfAppended = 0

            for nodeInCurrentConnections in numeratedNodes[index].Connections:

                connectedNodes.append(list(nodeInCurrentConnections.values())[0])

                connectionsSymbols.append(list(nodeInCurrentConnections.keys())[0])

            for nodeInCurrentConnections in connectedNodes:

                if not nodeInCurrentConnections in numeratedNodes:

                    numeratedNodes.append(nodeInCurrentConnections)

                    countOfAppended += 1

            if countOfAppended == 0 and index == len(numeratedNodes) - 1:

                break

            else:

                index += 1

        return {'numeratedNodes': numeratedNodes, 'connectionsSymbols': list(set(connectionsSymbols))}

    def createMap(self):

        listAllNodes = [self.Start]

        Map = {}

        index = 0

        while True:

            appended = 0

            Map[listAllNodes[index]] = {'Connections': listAllNodes[index].getConnections(), 'BackConnections': listAllNodes[index].getBackConnections()}

            for i in getAllNodes(Map[listAllNodes[index]]['Connections']):

                if not i in listAllNodes:

                    listAllNodes.append(i)

            index += 1

            if index == len(listAllNodes): break

        return Map

    def createTable(self):

        listOfNodes = self.getListOfNodes()

        sumbHref = listOfNodes['connectionsSymbols']

        returnsTable = []

        annotation = {}

        for i in range(len(sumbHref)):

            annotation[sumbHref[i]] = i

        for element in listOfNodes['numeratedNodes']:

            columnToInsert = [[] for i in range(len(sumbHref))]

            for oneConnect in element.Connections:

                columnToInsert[annotation[list(oneConnect.keys())[0]]].append(listOfNodes['numeratedNodes'].index(list(oneConnect.values())[0]))

            returnsTable.append(columnToInsert)

        return {'table': returnsTable, 'annotation': sumbHref}

class Node():

    def \_\_init\_\_(self) -> None:

        self.Connections = []

        self.BackConnections = []

    def appendConnection(self, letter, pointToConnect):

        self.Connections.append({letter: pointToConnect})

    def appendBackConnections(self, letter, pointToConnect):

        self.BackConnections.append({letter: pointToConnect})

    def getConnections(self):

        return self.Connections

    def getBackConnections(self):

        return self.BackConnections

def createLetterGraph(letter: str) -> Graph:

    """

        Создание графа по типу "символ", на вход принимает символ

    """

    localStartPoint = Node()

    localEndPoint = Node()

    localStartPoint.appendConnection(letter, localEndPoint)

    localEndPoint.appendBackConnections(letter, localStartPoint)

    return Graph(localStartPoint, localEndPoint)

def createConcatGraph(g1: Graph, g2 : Graph) -> Graph:

    g1.End.appendConnection('eps', g2.Start)

    g2.Start.appendBackConnections('eps', g1.End)

    return Graph(g1.Start, g2.End)

def createOrGraph(listOfGraphs) -> Graph:

    localStartPoint = Node()

    localEndPoint = Node()

    for iterGraph in listOfGraphs:

        localStartPoint.appendConnection('eps', iterGraph.Start)

        iterGraph.Start.appendBackConnections('eps', localStartPoint)

        iterGraph.End.appendConnection('eps', localEndPoint)

        localEndPoint.appendBackConnections('eps', iterGraph.End)

    return Graph(localStartPoint, localEndPoint)

def createStarGraph(g:Graph) -> Graph:

    localStartPoint = Node()

    localEndPoint = Node()

    localStartPoint.appendConnection('eps', localEndPoint)

    localEndPoint.appendBackConnections('eps', localStartPoint)

    localStartPoint.appendConnection('eps', g.Start)

    g.Start.appendBackConnections('eps', localStartPoint)

    g.End.appendConnection('eps', localEndPoint)

    localEndPoint.appendBackConnections('eps', g.End)

    g.End.appendConnection('eps', g.Start)

    g.Start.appendBackConnections('eps', g.End)

    return Graph(localStartPoint, localEndPoint)

def allSymbToGraphs(convertedList, sepcSymbolls = ['|', '.', '\*']):

    ret = []

    for element in convertedList:

        if not element in sepcSymbolls:

            ret.append(createLetterGraph(element))

        else:

            ret.append(element)

    return ret

def createGraphFromList(convertedList):

    listTograph = allSymbToGraphs(convertedList)

    index = 0

    while len(listTograph) != 1:

        if listTograph[index] == '.':

            if type(listTograph[index+1]) == Graph and type(listTograph[index+2]) == Graph:

                listTograph[index] = createConcatGraph(listTograph[index+1], listTograph[index+2])

                listTograph.pop(index+1)

                listTograph.pop(index+1)

            else:

                index += 1

        elif listTograph[index] == '|':

            if type(listTograph[index+1]) == Graph and type(listTograph[index+2]) == Graph:

                listTograph[index] = createOrGraph([listTograph[index+1], listTograph[index+2]])

                listTograph.pop(index+1)

                listTograph.pop(index+1)

            else:

                index += 1

        elif listTograph[index] == '\*':

            if type(listTograph[index+1]) == Graph:

                listTograph[index] = createStarGraph(listTograph[index+1])

                listTograph.pop(index+1)

            else:

                index += 1

        else:

            if index == len(listTograph) - 1:

                index = 0

            else:

                index += 1

    return listTograph[0]

from prettytable import PrettyTable

def prettyPrint(tableWithAnnotation):

    th = ['N']

    for i in tableWithAnnotation['annotation']: th.append(i)

    td = []

    index = 0

    for column in tableWithAnnotation['table']:

        td.append(str(index))

        for elem in column:

            if elem == []:

                td.append('-')

            else:

                td.append(','.join([str(i) for i in elem]))

        index += 1

    columns = len(th)

    table = PrettyTable(th)

    td\_data = td[:]

    while td\_data:

        table.add\_row(td\_data[:columns])

        td\_data = td\_data[columns:]

    print(table)  # Печатаем таблицу

def deleteLongEps(g: Graph):

    listOfnodes = g.getListOfNodes()['numeratedNodes']

    reconnections = {}

    candidatesToRemove = []

    for node in listOfnodes:

        listOfEpsConnections = []

        for elem in node.Connections:

            if list(elem.keys())[0] == 'eps':

                listOfEpsConnections.append(list(elem.values())[0])

        reconnections[node] = listOfEpsConnections

        index = 0

        if len(reconnections[node]) == 0:

            continue

        while True:

            listOfEpsConnections = []

            countOfAppend = 0

            for elem in reconnections[node][index].Connections:

                if list(elem.keys())[0] == 'eps':

                    noda = list(elem.values())[0]

                    listOfEpsConnections.append(noda)

                    countOfAppend += 1

                    if not reconnections[node][index] in candidatesToRemove:

                        candidatesToRemove.append(reconnections[node][index])

            for elemToAppend in listOfEpsConnections:

                reconnections[node].append(elemToAppend)

            if countOfAppend == 0 and index == len(reconnections[node]) - 1:

                break

            else:

                index += 1

    graphMap = g.createMap()

    #добавить новые епс соединения из списка новых соединений

    for key in reconnections.keys():

        for new\_connect in reconnections[key]:

            if not new\_connect in getAllNodes(graphMap[key]['Connections']):

                graphMap[key]['Connections'].append({'eps': new\_connect})

    #delete all connections and backconnections to candidates to remove

    for node in graphMap.keys():

        index\_in\_node = 0

        while True:

            if len(graphMap[node]['Connections']) > 0:

                if getNoda(graphMap[node]['Connections'][index\_in\_node]) in candidatesToRemove:

                    graphMap[node]['Connections'].pop(index\_in\_node)

                else:

                    index\_in\_node += 1

            if index\_in\_node >= len(graphMap[node]['Connections']): break

        index\_in\_node = 0

        while True:

            if len(graphMap[node]['BackConnections']) > 0:

                if getNoda(graphMap[node]['BackConnections'][index\_in\_node]) in candidatesToRemove:

                    graphMap[node]['BackConnections'].pop(index\_in\_node)

                else:

                    index\_in\_node += 1

            if index\_in\_node >= len(graphMap[node]['BackConnections']): break

    #Создать конечную точку и добавить новые конечные точки

    candidatesToFinish = [g.End]

    while True:

        appended = 0

        for node in graphMap.keys():

            for connect in graphMap[node]['Connections']:

                if getNoda(connect) in candidatesToFinish and getSymbol(connect) == 'eps' and not node in candidatesToFinish:

                    candidatesToFinish.append(node)

                    appended += 1

        if appended == 0:

            break

    for i in candidatesToFinish:

        g.Finish.append(i)

    #рекомпановка соединений через епсилоны

def recomposition(g: Graph):

    graphMap = g.createMap()

    recomposition = {}

    for noda in graphMap.keys():

        epsilons = [] #list on nodes

        for connection in noda.getConnections():

            if getSymbol(connection) == 'eps':

                epsilons.append(getNoda(connection))

        next = [] #list of list of connections

        for eps\_noda in epsilons:

            buffer\_connections = []

            for connection in eps\_noda.getConnections():

                if getSymbol(connection) != 'eps':

                    buffer\_connections.append(connection)

            next.append(buffer\_connections)

        recomposition[noda] = {'epsilons': epsilons, 'next':next}

    for noda in recomposition:

        #уалить эпсилон переходы из начала

        for epsilon\_noda\_index in range(len(recomposition[noda]['epsilons'])):

            epsilon\_noda = recomposition[noda]['epsilons'][epsilon\_noda\_index]

            if len(recomposition[noda]['next'][epsilon\_noda\_index]):

                c\_i\_b = graphMap[noda]['Connections']

                if {'eps': epsilon\_noda} in c\_i\_b:

                    c\_i\_b.pop(c\_i\_b.index({'eps': epsilon\_noda}))

        #удалить буквенные переходы после епсилон переходад

        for i\_it\_c in range(len(recomposition[noda]['next'])):

            list\_con = recomposition[noda]['next'][i\_it\_c]

            if list\_con != []:

                for iter\_con in list\_con:

                    del\_name = recomposition[noda]['epsilons'][i\_it\_c]

                    gr\_con = graphMap[del\_name]['Connections']

                    buf = {getSymbol(iter\_con): getNoda(iter\_con)}

                    if buf in gr\_con:

                        gr\_con.pop(gr\_con.index(buf))

        # соединение нод между которыми разорваны связи

        for i\_it\_c in range(len(recomposition[noda]['next'])):

            list\_con = recomposition[noda]['next'][i\_it\_c]

            if list\_con != []:

                for iter\_con in list\_con:

                    gr\_con = graphMap[noda]['Connections']

                    buf = {getSymbol(iter\_con): getNoda(iter\_con)}

                    if not buf in gr\_con:

                        gr\_con.append(buf)

    # return graphMap

    return recomposition

def deleteEpsConnections(g: Graph):

    graphMap = g.createMap()

    for noda in graphMap:

        index = 0

        if len(graphMap[noda]['Connections']) != 0:

            while True:

                if getSymbol(graphMap[noda]['Connections'][index]) == 'eps':

                    graphMap[noda]['Connections'].pop(index)

                else:

                    index += 1

                if index == len(graphMap[noda]['Connections']): break

    nodes = g.getListOfNodes()['numeratedNodes']

    index = 0

    if g.Finish != []:

        while True:

            if not g.Finish[index] in nodes:

                g.Finish.pop(index)

            else:

                index += 1

            if index == len(g.Finish): break

def NKAEps(parsedPoland: list):

    return createGraphFromList(parsedPoland)

def NKA(g:Graph):

    deleteLongEps(g)

    recomposition(g)

    deleteEpsConnections(g)

def getConnectionsInMap(Map, node):

    return Map[node]['Connections']

def convTIntL(a):

    k = a[1:-1].split(', ')

    return list(int(i) for i in k)

def DKA(g:Graph):

    nodes = g.getListOfNodes()['numeratedNodes']

    letters = g.getListOfNodes()['connectionsSymbols']

    numbers = {}

    newConnections = {}

    for i in range(len(nodes)): numbers[nodes[i]] = i

    Qstack = [[numbers[g.Start]]]

    index = 0

    while True:

        iterNodes = []

        for i in Qstack[index]: iterNodes.append(nodes[i])

        iterConnections = []

        for noda in iterNodes:

            for connection in noda.getConnections():

                iterConnections.append(connection)

        bufferConnections = {}

        for i in letters:

            bufferConnections[i] = []

        for i in iterConnections:

            if not numbers[getNoda(i)] in bufferConnections[getSymbol(i)]:

                bufferConnections[getSymbol(i)].append(numbers[getNoda(i)])

        for i in bufferConnections.keys():

            bufferConnections[i].sort()

        newConnections[str(Qstack[index])] = bufferConnections

        for i in bufferConnections.values():

            if not i in Qstack: Qstack.append(i)

        index += 1

        if index == len(Qstack): break

    fin = []

    for i in g.Finish:

        fin.append(numbers[i])

    #create DKA

    for i in newConnections.keys():

        for k in newConnections[i].keys():

            newConnections[i][k] = str(newConnections[i][k])

    sp = None

    resGraph = None

    newPoints = {}

    deleted\_keys = []

    for key in newConnections.keys():

        if key == '[]':

            deleted\_keys.append(key)

    for key in deleted\_keys:

        newConnections.pop(key)

    deleted\_keys = {}

    for key in newConnections.keys():

        deleted\_keys[key] = []

        for key\_in in newConnections[key]:

            if newConnections[key][key\_in] == '[]':

                deleted\_keys[key].append(key\_in)

    for key in deleted\_keys.keys():

        for key\_in in deleted\_keys[key]:

            newConnections[key].pop(key\_in)

    for nodeName in newConnections.keys():

        if 0 in convTIntL(nodeName):

            sp = Node()

            resGraph = Graph(sp, sp)

            newPoints[nodeName] = sp

        else:

            newPoints[nodeName] = Node()

    for nodeName in newPoints.keys():

        for i in fin:

            if i in convTIntL(nodeName):

                resGraph.Finish.append(newPoints[nodeName])

    for nodeName in newConnections.keys():

        for letterConnection in newConnections[nodeName].keys():

            newPoints[nodeName].appendConnection(letterConnection, newPoints[newConnections[nodeName][letterConnection]])

    return resGraph

def sumulation(g: Graph, reg: str) -> bool:

    currentNode = g.Start

    for letter in reg:

        connections\_in = currentNode.getConnections()

        flagHas = False

        index = None

        for index\_connect in range(len(connections\_in)):

            if getSymbol(connections\_in[index\_connect]) == letter:

                flagHas = True

                index = index\_connect

                break

        if flagHas:

            currentNode = getNoda(connections\_in[index])

        else:

            return False

    if currentNode in g.Finish:

        return True

    else:

        return False

def convertToPoland(s:str) -> str:

    #расставить точки

    operators = ['\*', '|']

    alloperators = ['\*', '(', ')', '|']

    allop = ['\*', ')', '|']

    buf = ''

    for i in range(len(s) - 1):

        if s[i] == ')':

            if not s[i+1] in allop:

                buf += s[i]

                buf += '.'

            else:

                buf += s[i]

        elif not s[i] in alloperators:

            if not s[i+1] in allop:

                buf += s[i]

                buf += '.'

            else:

                buf += s[i]

        elif s[i] == '\*':

            buf += s[i]

            buf += '.'

        else:

            buf += s[i]

    buf += s[-1]

    nbuf = ''

    for i in buf[::-1]:

        if i == ')':

            nbuf+= '('

        elif i == '(':

            nbuf += ')'

        else:

            nbuf += i

    #convert to polka

    ops = ['(', ')', '.', '|', '\*']

    prioritet = {'.': 1, '|':2, '\*': 3, '(': 0}

    rets = ''

    Qstack = []

    for let in nbuf:

        if not let in ops:

            rets += let

        else:

            if let == '(':

                Qstack.append(let)

            elif let == ')':

                while True:

                    ntr = Qstack.pop()

                    if ntr == '(':

                        break

                    else:

                        rets += ntr

            else:

                if len(Qstack) == 0:

                    Qstack.append(let)

                else:

                    if prioritet[let] > prioritet[Qstack[-1]]:

                        Qstack.append(let)

                    else:

                        while True:

                            if len(Qstack) == 0: break

                            top = Qstack[len(Qstack)-1]

                            priortop = prioritet.get(top)

                            priorlet = prioritet.get(let)

                            if priortop <= priorlet and len(Qstack) != 0:

                                ntr = Qstack.pop()

                                rets += ntr

                            else:

                                break

                        Qstack.append(let)

    while True:

        if len(Qstack) != 0:

            rets += Qstack.pop()

        else:

            break

    return rets[::-1]

def decart(s1,s2):

   return list([[a,b] for a in s1 for b in s2])

def andReplace(s):

    buf = s.copy()

    buf2 = list([[i[1], i[0]] for i in s])

    buf.extend(buf2)

    return buf

def minGraph(g: Graph) -> Graph:

    nodeList = g.getListOfNodes()

    gTable = g.createTable()

    index = 0

    nameOfNodes = {}

    for noda in nodeList['numeratedNodes']:

        nameOfNodes[noda] = index

        index += 1

    marked = []

    Qstack = []

    nameOfFinishes = []

    for noda in g.Finish:

        nameOfFinishes.append(nameOfNodes[noda])

    for noda in nodeList['numeratedNodes']:

        bufStr = []

        for noda\_in in nodeList['numeratedNodes']:

            if nameOfNodes[noda] == nameOfNodes[noda\_in]:

                bufStr.append(False)

            else:

                if nameOfNodes[noda] in nameOfFinishes:

                    if nameOfNodes[noda\_in] in nameOfFinishes:

                        bufStr.append(False)

                    else:

                        bufStr.append(True)

                        Qstack.append([nameOfNodes[noda\_in], nameOfNodes[noda]])

                else:

                    if nameOfNodes[noda\_in] in nameOfFinishes:

                        bufStr.append(True)

                        Qstack.append([nameOfNodes[noda\_in], nameOfNodes[noda]])

                    else:

                        bufStr.append(False)

        marked.append(bufStr)

    #создать дельту

    delta = []

    terminals = []

    for t in gTable['annotation']:

        terminals.append(t)

    for noda in nodeList['numeratedNodes']:

        bufStr = []

        for letter in terminals:

            bufLetter = []

            for connect in noda.getConnections():

                if getSymbol(connect) == letter:

                    bufLetter.append(nameOfNodes[getNoda(connect)])

            bufStr.append(bufLetter)

        delta.append(bufStr)

    #обновить marked по delta

    for noda in nodeList['numeratedNodes']:

        strName = nameOfNodes[noda]

        for noda\_in in nodeList['numeratedNodes']:

            stlbName = nameOfNodes[noda\_in]

            if marked[strName][stlbName] == False:

                buf = []

                for index in range(len(terminals)):

                    if delta[strName][index] != [] and delta[stlbName][index] != []:

                        buf.extend( andReplace(decart(delta[strName][index], delta[stlbName][index])) )

                for elem in buf:

                    if marked[elem[0]][elem[1]] or marked[elem[1]][elem[0]]:

                        marked[strName][stlbName] = True

                        marked[stlbName][strName] = True

                        break

    #создать deltaminus

    deltaminus = []

    for noda in nodeList['numeratedNodes']:

        bufStr = []

        for i in terminals:

            bufStr.append([])

        deltaminus.append(bufStr)

    for noda in nodeList['numeratedNodes']:

        for connect in noda.getConnections():

            deltaminus[nameOfNodes[getNoda(connect)]][terminals.index(getSymbol(connect))].append(nameOfNodes[noda])

    #удалить все из Qstack и обновить marked по deltaminus

    while True:

        if len(Qstack) != 0:

            para = Qstack.pop()

            if marked[para[0]][para[1]] == False or marked[para[1]][para[0]] == False:

                buf = []

                for ind\_l in range(len(terminals)):

                    if deltaminus[para[0]][ind\_l] != [] and deltaminus[para[1]][ind\_l] != []:

                        buf.extend(andReplace(decart(deltaminus[para[0]][ind\_l], deltaminus[para[1]][ind\_l])))

                for elem in buf:

                    if marked[elem[0]][elem[1]] == False or marked[elem[1]][elem[0]] == False:

                        marked[elem[0]][elem[1]] = True

                        marked[elem[1]][elem[0]] = True

                        Qstack.append(elem)

        else:

            break

    #выбрать компоненты

    components = []

    for stroka in marked:

        bufTocomp = []

        for index in range(len(stroka)):

            if stroka[index] == False:

                bufTocomp.append(index)

        if not bufTocomp in components:

            components.append(bufTocomp)

    componentsFromNodes = []

    for comp in components:

        bufTocomp = []

        for c\_in in comp:

            bufTocomp.append(nodeList['numeratedNodes'][c\_in])

        componentsFromNodes.append(bufTocomp)

    #собрать граф

    points = []

    mapa = {}

    for index\_comp in range(len(componentsFromNodes)):

        comp = componentsFromNodes[index\_comp]

        for noda in comp:

            mapa[noda] = index\_comp

    for comp in componentsFromNodes:

        points.append(Node())

    for index\_comp in range(len(componentsFromNodes)):

        comp = componentsFromNodes[index\_comp]

        for noda in comp:

            for connect in noda.getConnections():

                points[index\_comp].appendConnection(getSymbol(connect), points[mapa[getNoda(connect)]])

    retG = Graph(points[0], points[0])

    bufFin = []

    for index in range(len(components)):

        comp = components[index]

        for elem in comp:

            if elem in nameOfFinishes:

                if not index in bufFin:

                    bufFin.append(index)

    f = []

    for elm in bufFin:

        f.append(points[elm])

    retG.Finish = f

    mapping = retG.createMap()

    for noda in retG.getListOfNodes()['numeratedNodes']:

        buf = noda.getConnections()

        buf1 = []

        for i in buf:

            if not i in buf1:

                buf1.append(i)

        while True:

            if len(noda.getConnections()) == 0: break

            noda.getConnections().pop()

        for i in buf1:

            noda.getConnections().append(i)

    return retG

# kk = [1,2]

# kj = [3,4,5]

# ff = andReplace(decart(kk, kj))

# s = convertToPoland('(a|b)\*c|d')

# a = ".a\*|bc"

# g = NKAEps(a)

# prettyPrint(g.createTable())

# NKA(g)

# LOF = g.getListOfNodes()

# prettyPrint(g.createTable())

# DKA = DKA(g)

# prettyPrint(DKA.createTable())

# rerere = DKA.getListOfNodes()

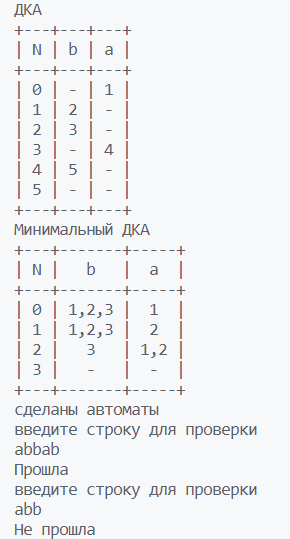
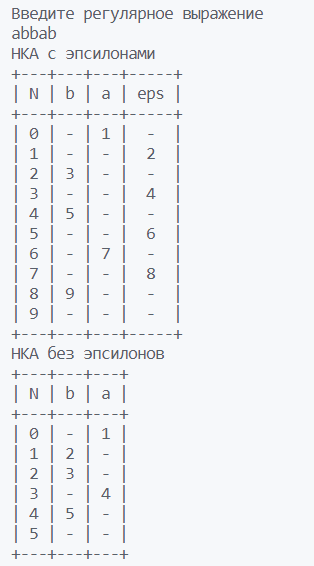
# minG = minGraph(DKA)

# prettyPrint(minG.createTable())

# flag = sumulation(minG, 'bc')

# print('all')

**Тесты**



**Вывод**

Освоен процесс создания и минимизации конечных автоматов и их сокращенных версий. Работа с автоматами происходит с графом как структурой, а не с таблицей смежности графа. Это занимает больше памяти, но способствует любой модификации программы. В качестве символов в регулярном выражении можно использовать что угодно. Процесс первоначального синтеза по строке токенов происходит ее преобразованием в обратную польскую запись, поэтому невозможна запись вида a|b|c, она будет распознана алгоритмом сортировочной станции как (a | b) | c, это можно исправить модификацией алгоритма сортировочной станции. При этом процессы над автоматом – минимизации и т д, будут работать так же, из-за устойчивой структуры графа.