Министерство образования и науки российской федерации

(минобрнауки россии)

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

«Санкт-Петербургский государственный политехнический университет» (ФГБОУ ВПО «СПбГПУ»)

**Институт менеджмента и информационных технологий**

(филиал)федерального государственного бюджетного образовательного учреждения высшего профессионального образования

«Санкт-Петербургский государственный политехнический университет» в г. Череповце (ИМИТ «СПбГПУ»)

Кафедра ПО ВТ и АС

ЛАБОРАТОРНАЯ РАБОТА №1

Дисциплина: «Теория языков программирования и методы трансляции»

Тема: «Лексический анализ и конечные автоматы»

Выполнил студент группы о.291 Шанин Игнат Леонидович

№ зачетной книжки о2080127

Проверил Михайлов Андрей Евгеньевич

«\_\_\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_201\_\_ г.

\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

отметка о зачете подпись преподавателя

г. Череповец

2012

# Задание

На основе порождающей грамматики (выбрать 2 варианта) построить конечный автомат и реализовать его на любом языке программирования. Программа должна принимать на вход текстовый файл с цепочками символов, разделённых пробелами и переводами строк.

На выходе должны формироваться три файла: один только с цепочками из входного файла, которые принадлежат языку, порождаемому первой грамматикой, второй – второй, третий – не принадлежат ни одному из двух языков.

# Краткая теоретическая справка

**Конечный автомат** — в [теории алгоритмов](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC%D0%BE%D0%B2) [математическая](http://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0) [абстракция](http://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%86%D0%B8%D1%8F), позволяющая описывать пути изменения [состояния](http://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D0%B5) объекта в зависимости от его текущего состояния и [входных данных](http://ru.wikipedia.org/wiki/%D0%92%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B5_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5), при условии что общее возможное количество состояний [конечно](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B5%D1%87%D0%BD%D0%BE%D0%B5_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE). Конечный автомат является частным случаем [абстрактного автомата](http://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82).

Существуют различные варианты задания конечного автомата. Например, конечный автомат может быть задан с помощью пяти параметров: \boldsymbol{M = (Q , \Sigma , \delta , q_0 , F)}где:

* Q — конечное *множество состояний* автомата;
* q0 — *начальное состояние* автомата ( q_0 \in Q);
* F — множество *заключительных*(или *допускающих*) состояний, таких что F \subset Q;
* Σ — допустимый *входной алфавит* (конечное множество допустимых входных символов), из которого формируются строки, считываемые автоматом;
* δ — заданное отображение множества Q \times \Sigma во множество \mathcal {P} (Q) подмножеств Q: \delta : Q \times \Sigma \rightarrow \mathcal {P} (Q) (иногда δ называют *функцией переходов автомата*).

Автомат начинает работу в состоянии q0, считывая по одному символу входной строки. Считанный символ переводит автомат в новое состояние из Q в соответствии с функцией переходов. Если по завершении считывания входного слова (цепочки символов) автомат оказывается в одном из допускающих состояний, то слово «принимается» автоматом. В этом случае говорят, что оно принадлежит языку данного автомата. В противном случае слово «отвергается».

Конечные автоматы широко используются на практике, например в [синтаксических](http://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%82%D0%BE%D1%80), [лексических анализаторах](http://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BD%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%82%D0%BE%D1%80), и [тестировании программного обеспечения](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F) [на основе моделей](http://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BD%D0%B0_%D0%BE%D1%81%D0%BD%D0%BE%D0%B2%D0%B5_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D0%B8).

Конечные автоматы подразделяются на детерминированные и недетерминированные.

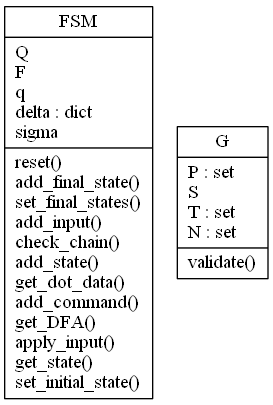
**Детерминированным конечным автоматом** (ДКА) называется такой автомат, в котором при любой данной последовательности входных символов существует лишь одно состояние, в которое автомат может перейти из текущего.

**Недетерминированный конечный автомат** (НКА) является обобщением детерминированного. Недетерминированность автоматов достигается двумя способами:

Существует теорема, гласящая, что «Любой недетерминированный конечный автомат может быть преобразован в детерминированный так, чтобы их языки совпадали» (такие автоматы называются [эквивалентными](http://ru.wikipedia.org/wiki/%D0%AD%D0%BA%D0%B2%D0%B8%D0%B2%D0%B0%D0%BB%D0%B5%D0%BD%D1%82%D0%BD%D0%BE%D1%81%D1%82%D1%8C)). Однако, поскольку количество состояний в эквивалентном ДКА в худшем случае растёт экспоненциально с ростом количества состояний исходного НКА, на практике подобная детерминизация не всегда возможна. Кроме того, [конечные автоматы с выходом](http://ru.wikipedia.org/w/index.php?title=%D0%9A%D0%BE%D0%BD%D0%B5%D1%87%D0%BD%D1%8B%D0%B9_%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82_%D1%81_%D0%B2%D1%8B%D1%85%D0%BE%D0%B4%D0%BE%D0%BC&action=edit&redlink=1) в общем случае не поддаются детерминизации.

В силу последних двух замечаний, несмотря на б**о**льшую сложность недетерминированных конечных автоматов, для задач, связанных с обработкой текста, преимущественно применяются именно НКА.

# Листинг

*#!/usr/bin/python*

*# -\*- coding: utf-8 -\*-*

**import** random

**from** multiprocessing **import** Process, Pool

**from** collections **import** defaultdict

**from** itertools **import** combinations, permutations, chain

**def** show\_graph(dotdata, title="xdot viewer"):

**import** gtk

**import** gtk.gdk

**import** xdot

    window = xdot.DotWindow()

    window.set\_dotcode(dotdata)

    window.connect('destroy', gtk.main\_quit)

    window.set\_title(title)

    gtk.main()

*##*

*#  #*

*#      # #     ###   ## #   ## #    ###   # #*

*# ##   ## #   #  #   # # #  # # #  #  #   ## #*

*#  #   #      #  #   # # #  # # #  #  #   #*

*##    #       ####  #   #  #   #   ####  #*

*диаграмма классов*

**class** G:

    """Representation of a formal grammar G = (N, T, P, S).

   G(N, T, P, S) -> grammar

   \* N - set of nonterminal symbols

   \* T - set of terminal symbols

   \* P - set of production rules

   \* S - start symbol

   Any symbol must be a string of length one. Rule is a string with nonterminal

   symbol, symol "->" and >= 1 strings composed of terminals and nonterminals

   separateg by "|".

   """

**def** \_\_init\_\_(self, T, N, P, S):

        self.T = set(T)

        self.N = set(N)

        self.P = set(P)

        self.S = S

        self.validate()

**def** \_\_str\_\_(self):

        s = 'G = (T, N, P, S)**\n**'

        s += '    T = {' + ', '.join((t **for** t **in** self.T)) + '}**\n**'

        s += '    N = {' + ', '.join((t **for** t **in** self.N)) + '}**\n**'

        s += '    P = {' + ',**\n**         '.join((t **for** t **in** self.P)) + '}**\n**'

        s += '    S = ' + self.S + "**\n**"

**return** s

**def** validate(self):

        """G.validate() -> True

       Return validness of the grammar. True if every symbol is length one str,

       NUT is empty, S in N and every rule match specified format. Owerwise

       raise ValueError.

       """

*# every symbol is length one str*

        invalid\_symbols = list(filter(

**lambda** x: type(x) **is** **not** str **or** len(x) **is** **not** 1,

            chain(self.N, self.T)))

*# intersection must be empty*

        invalid\_symbols.extend(self.N.intersection(self.T))

*# every rule must match specified format*

        invalid\_rules = list()

**for** rule **in** self.P:

            L, D = rule.replace(" ", "").split('->')

            D = D.replace("|", "")

**if** list(filter(**lambda** x: x **not** **in** self.N.union(self.T), D)) **or** \

               L **not** **in** self.N:

                invalid\_rules.append(rule)

**if** invalid\_symbols **or** invalid\_rules:

**raise** ValueError("Invalid symbols: " + str(invalid\_symbols) + " " +

                             "Invalid rules: " + str(invalid\_rules))

*####    ##    #   #*

*#      #  #   ## ##*

*###     #     # # #*

*#        #    # # #*

*#      #  #   #   #*

*#       ##    #   #*

**class** FSM:

    """Finite state machine

   FSM(G) -> finite state machine

   \* G - right linear grammar.

   FSM(sigma, Q, q, delta, F) -> finite state machine

   \* sigma - input alphabet (a finite, non-empty set of symbols).

   \* Q - a finite, non-empty set of states.

   \* q - an initial state, an element of Q.

   \* delta - state-transition dictionary: delta[state1][input] -> set of states

   \* F - set of final states, subset of Q.

   """

**def** \_\_init\_\_(self, \*args):

**if** len(args) **is** 5: *# FSM(sigma, Q, q, delta, F)*

            sigma, Q, q, delta, F = args

            self.sigma = sigma

            self.Q = Q

            self.q = q

            self.delta = delta

            self.F = F

            self.\_state = q

**elif** len(args) **is** 1: *# FSM(G)*

            G, = args

            sigma = set()

            Q = set()

            q = G.S

            delta = dict()

            F = set()

            self.\_\_init\_\_(sigma, Q, q, delta, F)

**for** state **in** G.N:

                self.add\_state(state)

            self.add\_state('Z')

            self.set\_final\_states(set('Z'))

**for** input\_ **in** G.T:

                self.add\_input(input\_)

**for** rule **in** G.P:

                extra\_state\_index = 1

                L, D = rule.replace(" ", "").split('->')

**for** d **in** D.split('|'):

                    state\_before\_last\_unput = L

**if** d[-1] **not** **in** self.Q:

                        d += 'Z'

**for** input\_ **in** d[:-2]: *# add extra states if necessary*

                        state\_before\_last\_unput = L + str(extra\_state\_index)

                        self.add\_state(state\_before\_last\_unput)

                        self.add\_command(L, input\_, state\_before\_last\_unput)

                        extra\_state\_index += 1

                    self.add\_command(state\_before\_last\_unput, d[-2], d[-1])

**def** add\_state(self, state):

        """m.add\_state(str) Add state in Q"""

        self.Q.add(state)

        self.delta[state] = defaultdict(set)

**def** add\_final\_state(self, state):

        """m.add\_final\_state(str) Add state in F"""

**if** state **in** self.Q:

            self.F.add(state)

**else**:

**raise** ValueError('Automaton does not have specified state!')

**def** add\_input(self, input\_):

        """m.add\_input(str) Add input in sigma"""

        self.sigma.add(input\_)

**def** set\_initial\_state(self, state):

        """m.set\_initial\_state(str)

       Set q

       """

**if** state **in** self.Q:

            self.\_state = state

**else**:

**raise** ValueError('Automaton does not have specified state!')

**def** set\_final\_states(self, states):

        """m.set\_final\_states(set) Set F"""

**if** states.issubset(self.Q):

            self.F = states

**else**:

**raise** ValueError('Automaton does not have specified state!')

**def** add\_command(self, state1, input\_, state2):

        """m.add\_command(state1, input, state2) Add command"""

**if** state1 **in** self.Q **and** state2 **in** self.Q **and** input\_ **in** self.sigma:

            self.delta[state1][input\_].add(state2)

**else**:

**raise** ValueError("Automaton doesn't have specified state or input!")

**def** get\_state(self):

        """m.get\_state() -> current state"""

**return** self.\_state

**def** reset(self):

        """m.reset() set curent state to q"""

        self.\_state = self.q

**def** apply\_input(self, input\_):

        """m.apply\_input(input) change state according to input"""

        self.\_state = random.choice(list(self.delta[self.\_state][input\_]))

**def** check\_chain(self, chain\_):

        """m.check\_chain(chain) -> True if chain matches grammar, else False"""

        self.reset()

**for** i **in** chain\_:

**try**:

                self.apply\_input(i)

**except** IndexError, e:

**return** False

**return** self.\_state **in** self.F

**def** get\_dot\_data(self):

        """m.get\_dot\_data() -> graphviz dot data"""

**def** sts(state): *# state to string*

**if** type(state) **is** frozenset:

**if** **not** state: *# empty set*

**return** 'Ø'

                symbols = list(state)

                symbols.sort()

**return** ''.join(symbols)

**return** state

        dotdata = ('digraph finite\_state\_machine {**\n**'

         + 'rankdir=LR;**\n**'

         + 'size="8,5"**\n**'

         + 'node [shape = doublecircle]; ' + sts(self.q) + ' '

         + ' '.join((sts(x) **for** x **in** self.F)) + '**\n**'

         + 'node [shape = circle];'

         )

**for** state1, inputs **in** self.delta.items():

**for** input\_, states **in** inputs.items():

**for** state2 **in** states:

                    dotdata += '{s1} -> {s2} [ label = "{i}" ];**\n**'.format(

                                 s1=sts(state1), s2=sts(state2), i=input\_)

**return** dotdata + '}'

**def** get\_DFA(self):

        """m.get\_to\_DFA() -> determenitive automaton"""

        DFA = FSM(set(), set(), frozenset([self.q]), dict(), set())

**for** i **in** range(len(self.Q)):

**for** state **in** combinations(self.Q, i+1):

                DFA.add\_state(frozenset(state))

**if** 'Z' **in** state:

                    DFA.add\_final\_state(frozenset(state))

        DFA.add\_state(frozenset())

**for** input\_ **in** self.sigma:

            DFA.add\_input(input\_)

**for** dstate **in** DFA.Q:

**if** dstate == frozenset():

**continue**

**for** input\_ **in** self.sigma:

                dstate2 = set()

**for** state **in** dstate:

                    dstate2.update(self.delta[state][input\_])

                DFA.add\_command(dstate, input\_, frozenset(dstate2))

*# remove unreachable states*

        reachable = set([DFA.q])

        search\_queue = [DFA.q]

**while** search\_queue:

            state = search\_queue.pop()

**for** input\_ **in** DFA.sigma:

**for** state2 **in** DFA.delta[state][input\_]:

**if** state2 **not** **in** reachable:

                        reachable.add(state2)

                        search\_queue.append(state2)

**if** frozenset() **in** reachable:

            reachable.remove(frozenset())

        DFA.delta = dict((k, v) **for** k, v **in** DFA.delta.items() **if** k **in** reachable)

        DFA.Q = DFA.Q.intersection(reachable)

        DFA.F = DFA.F.intersection(reachable)

*# there are still empty state*

**for** state1 **in** DFA.delta:

**for** input\_ **in** DFA.delta[state1].keys():

**if** DFA.delta[state1][input\_] == set([frozenset([])]):

                    DFA.delta[state1].pop(input\_)

**return** DFA

*# Вариант 1*

G1 = G({'0', '1', '#'}, {'S', 'N'}, {

        'S -> 0S | 1S | 0#N | 1#N',

        'N -> 0 | 1 | 11 | 0N | 1N'

    }, 'S')

*# Вариант 2*

G2 = G({'0', '1'}, {'S', 'B', 'C'}, {

        'S -> 0B | 1S ',

        'B -> 0C | 1B | 01 ',

        'C -> 0B | 1S '

    }, 'S')

*# Вариант 3*

G3 = G({'a', 'b', '+'}, {'S', 'A', 'B'}, {

        'S -> aA | aB | bA ',

        'A -> b+S ',

        'B -> a+S | bB | a '

    }, 'S')

*# Вариант 4*

G4 = G({'0', '1', '+'}, {'S', 'M', 'N'}, {

        'S -> 0S | 1S | 0M | 1M ',

        'M -> +N ',

        'N -> 0 | 1 | 0N | 1N '

    }, 'S')

*# Вариант 5*

G5 = G({'x', 'y', '+'}, {'S', 'B', 'C'}, {

        'S -> xB ',

        'B -> yC | y+S ',

        'C -> x '

    }, 'S')

*# Вариант 6*

G6 = G({'m', 'n', '-', '\*'}, {'S', 'B', 'C'}, {

        'S -> -B ',

        'B -> m\*C | m | nB ',

        'C -> nB ',

    }, 'S')

*# Вариант 7*

G7 = G({'0', '1', '+'}, {'H', 'A', 'B'}, {

        'H -> 0A | 1A ',

        'A -> 0A | 1A | +B | 0 | 1 ',

        'B -> 0+A | 1A '

    }, 'H')

*# Вариант 8*

G8 = G({'a', 'b'}, {'S', 'B', 'C', 'D'}, {

        'S -> bC ',

        'B -> aB | ab | bD ',

        'C -> aB | aaC     ',

        'D -> bD | b '

    }, 'S')

**if** \_\_name\_\_ == '\_\_main\_\_':

    M1 = FSM(G1)

    M2 = FSM(G2)

    D1 = M1.get\_DFA()

    D2 = M2.get\_DFA()

    args = [(M1.get\_dot\_data(), u"НДКА первой грамматики"),

            (M2.get\_dot\_data(), u"НДКА второй грамматики"),

            (D1.get\_dot\_data(), u"ДКА первой грамматики"),

            (D2.get\_dot\_data(), u"ДКА второй грамматики")]

    threads = [Process(target=show\_graph, args=a) **for** a **in** args]

    map(Process.start, threads)

    map(Process.join, threads)

**from** pygraphviz **import** AGraph

**for** data, name **in** args:

        G = AGraph(data)

        G.draw(name + '.png', prog='dot')

**with** open("test.txt") **as** f:

        chains = f.read().replace(" ", "").split('**\n**')

    chains1 = []

    chains2 = []

    chains\_no\_one = []

**for** c **in** chains:

**if** D1.check\_chain(c):

            chains1.append(c)

**elif** D2.check\_chain(c):

            chains2.append(c)

**else**:

            chains\_no\_one.append(c)

**with** open("1.txt", "w") **as** f:

        f.write('**\n**'.join(chains1))

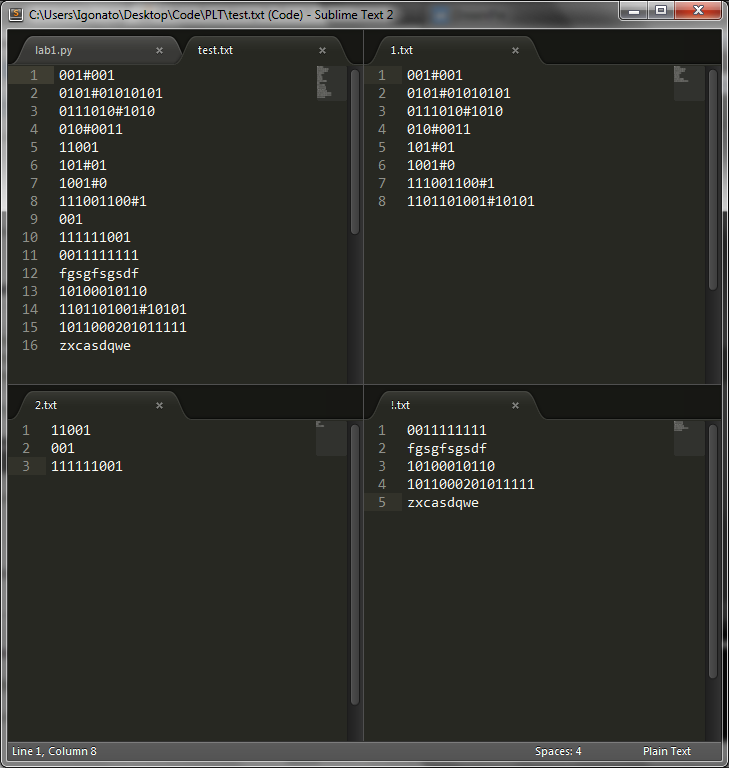
**with** open("2.txt", "w") **as** f:

        f.write('**\n**'.join(chains2))

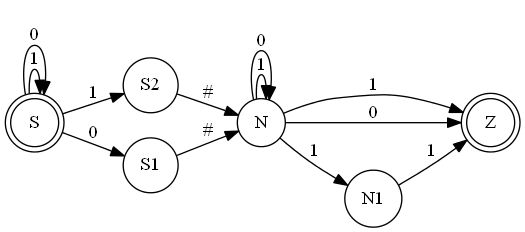
**with** open("!.txt", "w") **as** f:

        f.write('**\n**'.join(chains\_no\_one))

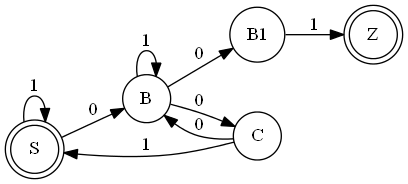
# Результат работы программы



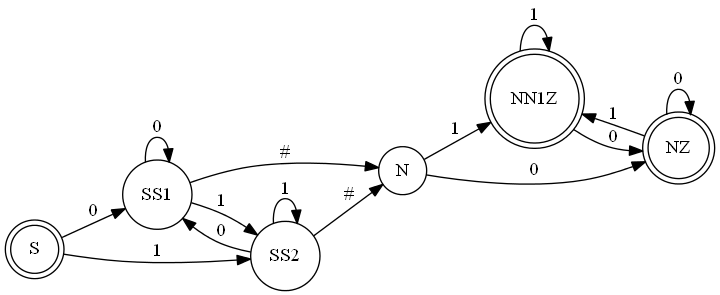
*Рис 1. Текстовые файлы*



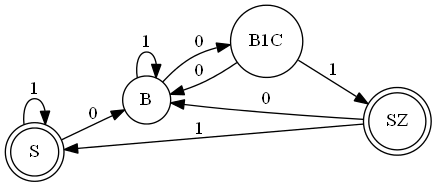
*Рис 2. НДКА первой грамматики*



*Рис 3. НДКА второй грамматики*



*Рис 4. ДКА первой грамматики*



*Рис 5. ДКА второй грамматики*

# Вывод

В результате проделанной работы была написана программа, позволяющая на основании заданной грамматики строить конечный автомат, приводить его к детерминированному случаю и определять, принадлежат ли грамматике цепочки символов.