МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

АЭРОКОСМИЧЕСКОГО ПРИБО6РОСТРОЕНИЯ»

КАФЕДРА № 43

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Старший преподаватель |  |  |  | С.А. Рогачев |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №5 |
| Синтез КНА |
| по курсу: Теория Вычислительных Процессов |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4134к |  |  |  | Д. В. Самарин |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2024

**Цель работы:**

Создать программу на языке высокого уровня реализующую алгоритм синтеза конечного автомата на основе заданного регулярного выражения.

**Постановка задачи**

11) <<a><b>d>kc(b|d)x– регулярное выражение

### **Строим конечный автомат Мили:**

Чтобы построить конечный автомат Мили для проверки допустимости слов, соответствующих данному регулярному выражению, нужно:

1. Определить состояния:

**Состояния:**

* q0 — начальное состояние.
* q1 — после <a>.
* q2 — после <b>.
* q3 — после d.
* q4 — после k.
* q5 — после c.
* q6 — после b или d (последний символ перед x).
* q7 — принимающее состояние после символа x

1. Алфавит:

Входной алфавит: {a, b, d, k, c, x}.

Алфавит выходных символов: {0,1}

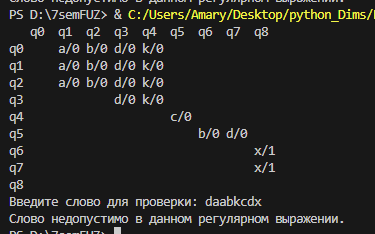
1. Функции переходов (δ) и выходов (λ):
   * Функция переходов **δ**: Она определяет, в какое состояние автомат перейдёт при поступлении определённого символа.
   * Функция выходов λ : Она определяет, какой выход будет сгенерирован при поступлении определённого символа в конкретном состоянии

**Конечный автомат заданный тремя способами**

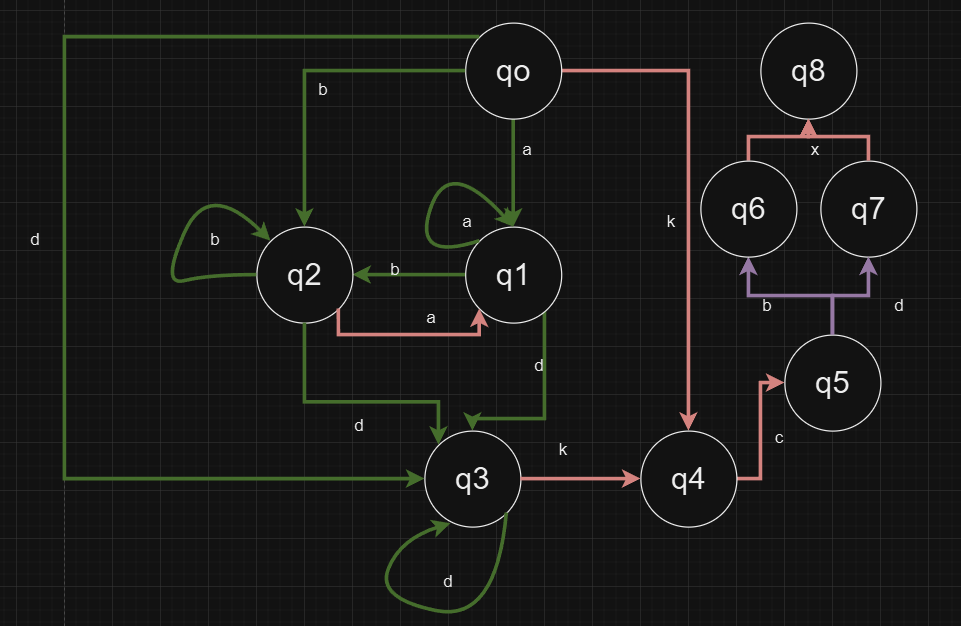
Матричный

<<a><b>d>kc(b|d)x

|  |  |  |  |
| --- | --- | --- | --- |
| Состояние | Символ | Следующее состояние | Выход |
| q0 (k) | k | q4 | 0 |
| q0 (a) | <a> | q1 | 0 |
| q0 (b) | <b> | q2 | 0 |
| q0 (d) | <d> | q3 | 0 |
| q1(aa) | <a> | q1 | 0 |
| q1 (ad) | <d> | q3 | 0 |
| q1 (ab) | <b> | q2 | 0 |
| q1(ak) | k | q4 | 0 |
| q2(bb) | <b> | q2 | 0 |
| q2 (bd) | <d> | q3 | 0 |
| q2(aba) | <a> | q1 | 0 |
| q2(bk) | k | q4 | 0 |
| q3(dd) | <d> | q3 | 0 |
| q3 | k | q4 | 0 |
| q4 | c | q5 | 0 |
| q5 | d | q6 | 0 |
| q5 | b | q7 | 0 |
| q6 | x | q8 | 1 |



Граф переходов

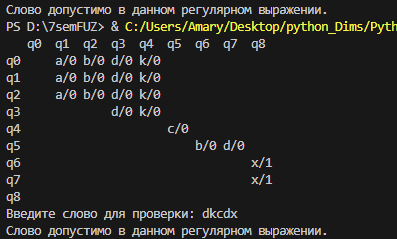


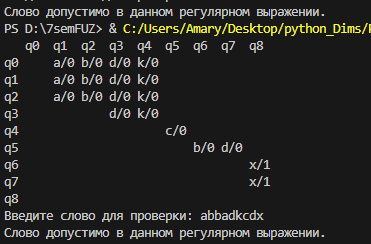
11) <<a><b>d>kc(b|d)x– регулярное выражение

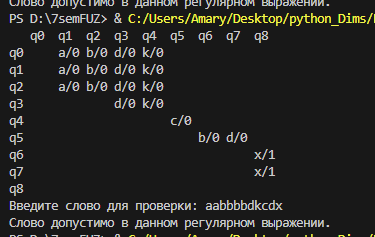
Автоматная таблица

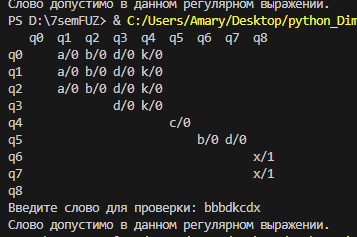
<<a><b>d>kc(b|d)x

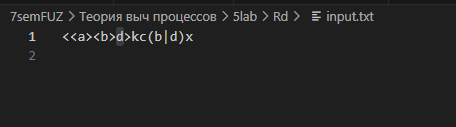
|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Состояние | Вход <a> | Вход  <b> | Вход  <d> | Вход  k | Вход  c | Вход  d | Вход  b | Вход  x | Выход |
| q0 | q1 | q2 | q3 | q4 |  |  |  |  | 0 |
| q1 | q1 | q2 | q3 | q4 |  |  |  |  | 0 |
| q2 |  |  | q3 | q4 |  |  |  |  | 0 |
| q3 |  |  | q3 | q4 |  |  |  |  | 0 |
| q4 |  |  |  |  | q5 |  |  |  | 0 |
| q5 |  |  |  |  |  | q6 | q7 |  | 0 |
| q6 |  |  |  |  |  |  |  | q8 | 1 |
| q7 |  |  |  |  |  |  |  |  | принято |











Листинг кода:

|  |
| --- |
| import sys  import os  #<<a><b>d>kc(b|d)x  # Принятые слова: bdkcdx, bdkcbx, addkcbx, aabbbbdkcdx, bbbdkcdx abdkcdx  # Важные допустимые местами поменять ab и повторить  - abbadkcdx  # Важные не допустимые: daabkcdx  # Константы  ROUND\_OPENING\_BRACKETS = ['(']  ROUND\_CLOSING\_BRACKETS = [')']  ANGLE\_OPENING\_BRACKETS = ['<']  ANGLE\_CLOSING\_BRACKETS = ['>']  OPERATORS = ROUND\_OPENING\_BRACKETS + ROUND\_CLOSING\_BRACKETS + ANGLE\_OPENING\_BRACKETS + ANGLE\_CLOSING\_BRACKETS + ['|']  # Функция для чтения алфавита из регулярного выражения  def read\_alphabet(regex):      return sorted({char for char in regex if char not in OPERATORS})  # Функция для печати выходной матрицы в файл и в консоль  def print\_output(output\_matrix, conditions):      output\_file\_path = r'D:/GIT/VuzUC/7SEM/TVP/LAB5/output.txt'      with open(output\_file\_path, 'w', encoding='utf-8') as f:          original\_stdout = sys.stdout          sys.stdout = f          print\_table(output\_matrix, conditions)          sys.stdout = original\_stdout      print\_table(output\_matrix, conditions)  # Общая функция для печати таблицы  def print\_table(output\_matrix, conditions):      headers = '   ' + '  '.join(['q' + str(i) for i in range(len(conditions))])      print(headers)      for i, row in enumerate(output\_matrix):          print('q' + str(i), ' '.join(row))  # Функции для обработки правил подчиненности для круглых и угловых скобок  def apply\_first\_rule\_round\_brackets(regex, subordination\_dependencies, i):      if regex[i] in ROUND\_OPENING\_BRACKETS:          subordination\_dependencies[i + 1].append(i)          update\_subordination\_dependencies\_for\_round\_brackets(regex, subordination\_dependencies, i)  def apply\_first\_rule\_angle\_brackets(regex, subordination\_dependencies, i, levels):      if regex[i] in ANGLE\_OPENING\_BRACKETS:          subordination\_dependencies[i + 1].append(i)          update\_subordination\_dependencies\_for\_angle\_brackets(regex, subordination\_dependencies, i, levels)  def track\_angle\_bracket\_levels(regex):      levels = [0] \* len(regex)      bracket\_counter = 0      for i, char in enumerate(regex):          if char in ANGLE\_OPENING\_BRACKETS:              bracket\_counter += 1              levels[i] = bracket\_counter          elif char in ANGLE\_CLOSING\_BRACKETS:              levels[i] = bracket\_counter              bracket\_counter -= 1          else:              levels[i] = bracket\_counter      return levels  # Обновление зависимостей для круглых скобок  def update\_subordination\_dependencies\_for\_round\_brackets(regex, subordination\_dependencies, i):      bracket\_counter = 0      for j in range(i, len(regex)):          if regex[j] in ROUND\_OPENING\_BRACKETS:              bracket\_counter += 1          if regex[j] in ROUND\_CLOSING\_BRACKETS:              if bracket\_counter == 1:                  break              bracket\_counter -= 1          if regex[j] == '|' and bracket\_counter == 1:              subordination\_dependencies[j + 1].append(i)  # Обновление зависимостей для угловых скобок с поддержкой вариативности  def update\_subordination\_dependencies\_for\_angle\_brackets(regex, subordination\_dependencies, i, levels):      bracket\_counter = 0      for j in range(i, len(regex)):          if regex[j] in ANGLE\_OPENING\_BRACKETS:              bracket\_counter += 1          elif regex[j] in ANGLE\_CLOSING\_BRACKETS:              bracket\_counter -= 1              if bracket\_counter == 0:                  break          # Проверяем уровень вложенности          if bracket\_counter == 1 and levels[j] == 2:              # Этот символ находится в двойных угловых скобках              subordination\_dependencies[j + 1].append(i)  # Применение второго правила для круглых скобок  def apply\_second\_rule\_round\_brackets(regex, subordination\_dependencies, A, i):      bracket\_counter = 0      if regex[i] in ROUND\_OPENING\_BRACKETS:          helper = []          for j in range(i, len(regex)):              if regex[j] in ROUND\_OPENING\_BRACKETS:                  bracket\_counter += 1              if regex[j] in ROUND\_CLOSING\_BRACKETS:                  if bracket\_counter == 1:                      subordination\_dependencies[j + 1] = helper                      break                  else:                      bracket\_counter -= 1              if (regex[j] in A and (j + 1 < len(regex) and regex[j + 1] not in A)) and bracket\_counter == 1:                  helper.append(j + 1)  # Применение второго правила для угловых скобок  def apply\_second\_rule\_angle\_brackets(regex, subordination\_dependencies, A, i):      bracket\_counter = 0      if regex[i] in ANGLE\_OPENING\_BRACKETS:          helper = [i]          for j in range(i, len(regex)):              if regex[j] in ANGLE\_OPENING\_BRACKETS:                  bracket\_counter += 1              if regex[j] in ANGLE\_CLOSING\_BRACKETS:                  if bracket\_counter == 1:                      subordination\_dependencies[j + 1] = helper                      break                  else:                      bracket\_counter -= 1              if (regex[j] in A and (j + 1 < len(regex) and regex[j + 1] not in A)) and bracket\_counter == 1:                  helper.append(j + 1)  # Применение третьего правила для угловых скобок  def apply\_third\_rule\_angle\_brackets(regex, subordination\_dependencies, A, i):      bracket\_counter = 0      if regex[i] in ANGLE\_CLOSING\_BRACKETS:          place = i + 1          for j in range(i, -1, -1):              if regex[j] in ANGLE\_CLOSING\_BRACKETS:                  bracket\_counter += 1              if regex[j] in ANGLE\_OPENING\_BRACKETS:                  if bracket\_counter == 1:                      break                  else:                      bracket\_counter -= 1              if (regex[j] in A or regex[j] in ANGLE\_OPENING\_BRACKETS) and (j - 1 >= 0 and regex[j - 1] not in A) and bracket\_counter == 1:                  subordination\_dependencies[j].append(place)  # Функция для применения всех правил подчиненности к регулярному выражению  def subordination\_rules\_with\_levels(regex, markup, A):      subordination\_dependencies = [[] for \_ in range(len(regex) + 1)]      levels = track\_angle\_bracket\_levels(regex)  # Получение уровня вложенности для каждого символа      for i in range(len(regex)):          apply\_first\_rule\_round\_brackets(regex, subordination\_dependencies, i)          apply\_first\_rule\_angle\_brackets(regex, subordination\_dependencies, i, levels)  # Передача levels в вызов функции          update\_subordination\_dependencies\_for\_angle\_brackets(regex, subordination\_dependencies, i, levels)          apply\_second\_rule\_round\_brackets(regex, subordination\_dependencies, A, i)          apply\_second\_rule\_angle\_brackets(regex, subordination\_dependencies, A, i)          apply\_third\_rule\_angle\_brackets(regex, subordination\_dependencies, A, i)      # Обновляем разметку на основе зависимостей      for i in range(len(subordination\_dependencies)):          for j in range(len(subordination\_dependencies)):              if i in subordination\_dependencies[j]:                  markup[j].extend(x for x in markup[i] if x not in markup[j])      return markup  # Основная функция программы  def main():      input\_file\_path = r'D:/GIT/VuzUC/7SEM/TVP/LAB5/input.txt'      # Проверка на существование файла      if not os.path.exists(input\_file\_path):          print("Ошибка: файл не найден.")          return      with open(input\_file\_path, 'r', encoding='utf-8') as f:          regex = f.readline().strip()      A = read\_alphabet(regex)      markup = [[] for \_ in range(len(regex) + 1)]      pre\_primary\_places = []      counter = 1      # Заполнение начальных разметок для символов регулярного выражения      for i in range(len(regex) + 1):          if i == 0:              markup[i].append(0)              continue          if regex[i - 1] in A:              markup[i].append(counter)              pre\_primary\_places.append(i - 1)              counter += 1      # Применение правил подчиненности к разметке с учетом уровней вложенности      markup = subordination\_rules\_with\_levels(regex, markup, A)      # Построение множества состояний и переходов      conditions, table = build\_automaton\_states(regex, A, markup, pre\_primary\_places)      # Обработка альтернативных состояний      conditions = process\_alternative\_states(conditions)      # Формирование выходной матрицы переходов автомата      output\_matrix = create\_transition\_matrix(conditions, table, markup)      # Печать выходной матрицы переходов      print\_output(output\_matrix, conditions)      # Проверка слова на допустимость в автомате      check\_word(table, markup[-1])    # Функция для построения состояний автомата  def build\_automaton\_states(regex, A, markup, pre\_primary\_places):      conditions = [[0]]      i = 0      table = {a: [] for a in A}      while i < len(conditions):          for a in A:              flag = False              adding = []              for c in conditions[i]:                  for p in pre\_primary\_places:                      if c in markup[p] and regex[p] == a:                          adding.extend(markup[p + 1])                          flag = True                            # Проверяем, находится ли символ внутри двойных угловых скобок и добавляем соответствующие переходы                          if p > 0 and regex[p - 1] == '<':                              # Определение начала и конца секции с двойными угловыми скобками                              start\_index = p                              while start\_index > 0 and regex[start\_index - 1] == '<':                                  start\_index -= 1                              end\_index = p                              while end\_index < len(regex) - 1 and regex[end\_index + 1] == '>':                                  end\_index += 1                              # Добавляем переходы между всеми символами внутри этих двойных угловых скобок                              for k in range(start\_index, end\_index + 1):                                  if regex[k] in A and k != p:                                      adding.extend(markup[k + 1])              if not flag:                  table[a].append(None)              else:                  adding = sorted(set(adding))                  existing\_condition = next((condition for condition in conditions if set(condition) == set(adding)), None)                  if existing\_condition is None:                      conditions.append(adding)                      table[a].append(adding)                  else:                      table[a].append(existing\_condition)          i += 1      # Добавляем связь между состояниями для переходов из q2 в q1, если это необходимо      for cond in conditions:          if 2 in cond and 1 not in cond:              cond.append(1)      return [cond for cond in conditions if cond], table  # Функция для обработки альтернативных состояний  def process\_alternative\_states(conditions):      unique\_conditions = []      condition\_mapping = {}      for condition in conditions:          found = False          for unique in unique\_conditions:              if set(condition) == set(unique):                  condition\_mapping[tuple(condition)] = unique                  found = True                  break          if not found:              unique\_conditions.append(condition)              condition\_mapping[tuple(condition)] = condition      return unique\_conditions  # Обновленная функция для создания выходной матрицы переходов  def create\_transition\_matrix(conditions, table, markup):      exit\_symbols = markup[-1]      is\_condition\_in\_regex = [1 if any(c in exit\_symbols for c in condition) else 0 for condition in conditions]      output\_matrix = [['   ' for \_ in range(len(conditions))] for \_ in range(len(conditions))]      for key, elements in table.items():          for i, element in enumerate(elements):              if element is None:                  continue              index = conditions.index(element)              if output\_matrix[i][index] == '   ':                  output\_matrix[i][index] = key + '/' + str(is\_condition\_in\_regex[index])              elif key not in output\_matrix[i][index]:                  output\_matrix[i][index] += ',' + key + '/' + str(is\_condition\_in\_regex[index])      # Добавляем связь для возврата из q2 в q1, если необходимо      for i, condition in enumerate(conditions):          if 2 in condition and 1 in condition:              output\_matrix[i][1] = 'a/0'      return output\_matrix  # Функция для проверки слова и печати результата  def check\_word(table, exit\_symbols):      input\_word = input("Введите слово для проверки: ").strip()      current\_states = [0]      for symbol in input\_word:          next\_states = set()          if symbol in table:              for state in current\_states:                  transitions = table[symbol][state]                  if transitions is not None:                      next\_states.update(transitions)          current\_states = list(next\_states)      if any(state in exit\_symbols for state in current\_states):          print("Слово допустимо в данном регулярном выражении.")      else:          print("Слово недопустимо в данном регулярном выражении.")  if \_\_name\_\_ == '\_\_main\_\_':      main() |

**Вывод**

Я задал конечный автомат, который проверяет входное слово на допустимость в заданном регулярном выражении тремя способами