МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»
Институт компьютерных наук и кибербезопасности
Высшая школа технологий искусственного интеллекта
Направление **02.03.01** : Математика и компьютерные науки

Лабораторная работа «Проверка корректности адреса с использованием регулярных выражений и конечного автомата» по дисциплине «Теория алгоритмов»

Обучающийся: _____ Яшнова Дарья Михайловна
группа 5130201/20002

Руководитель: _____ Востров Алексей Владимирович

«___»_____ 2025г

Санкт-Петербург, 2025

Содержание

Введение

Целью данной работы является разработка программы, реализующей преобразование регулярного выражения в конечный автомат, его детерминизацию, а также создание инструментов для проверки строк на соответствие заданному шаблону и генерации корректных тестовых данных.

Задание: По заданному варианту построить регулярное выражение, затем недетерминированный конечный автомат и детерминировать его (переходы можно задавать диапазонами). Реализовать программу, которая проверяет введенный текст через реализацию конечного автомата (варианты вывода: строка соответствует, не соответствует, символы не из алфавита). Также необходимо реализовать функцию случайной генерации верной строки по полученному конечному автомату.

Вариант 21: Проверка корректности адреса в формате (регион, город, улица/проспект/бульвар. . . , номер дома и квартира (если есть)) с разделителем ;

1 Математическое описание

1.1 Регулярные выражения

Регулярные выражения — это формальный способ описания регулярных языков над конечным словарём Σ. Они строятся из следующих элементов:

- **Базовые компоненты**:
  - $\emptyset$ — пустое множество (не содержит цепочек)
  - ε — пустая цепочка
  - $a \in \Sigma$ — любой символ из словаря
- **Операции**:
  - **Объединение** ($R_1 + R_2$ или $R_1|R_2$): представляет множество цепочек, принадлежащих R_1 или R_2
  - **Конкатенация** (R_1R_2): множество цепочек, образованных соединением цепочки из R_1 и цепочки из R_2
  - **Итерация Клини** (R^*): множество цепочек, состоящих из нуля или более повторений цепочек из R
  - **Усечённая итерация** (R^+): аналогична итерации, но требует минимум одного повторения ($R^+ = RR^*$)

Приоритет операций:

1. Итерация ($*$) и усечённая итерация ($+$) — наивысший приоритет
2. Конкатенация (без явного символа)
3. Объединение ($+$ или $|$)

Скобки изменяют порядок выполнения операций.

1. Регулярные множества и языки

   Регулярное множество — это бесконечное множество цепочек, построенное из символов Σ с использованием операций объединения, конкатенации и итерации. Например:

   - Регулярное выражение $ab + ba^*$ задаёт множество:

     $$\{ab\} \cup \{b, ba, baa, baaa, \ldots\} = \{ab, b, ba, baa, \ldots\}$$

   - Выражение $(ac)^*b + c^*$ описывает объединение двух множеств:
     - Цепочки вида $(ac)^n b$ (например, $b, acb, acacb, \ldots$)
     - Цепочки из любого числа символов c (включая ε)

   Регулярный язык — это подмножество Σ^*, которое может быть описано регулярным выражением. Например, язык всех цепочек из символов a, b, c является регулярным, так как Σ^* задаётся выражением $(a + b + c)^*$.

2. Формальные операции над языками

   Для языков $L_1, L_2 \subseteq \Sigma^*$ определены следующие операции:

(a) **Объединение**:

$$L_1 \cup L_2 = \{w \mid w \in L_1 \text{ или } w \in L_2\}$$

Пример: $\{abbc, cb\} \cup \{ba, ccc\} = \{abbc, cb, ba, ccc\}$

(b) **Конкатенация**:

$$L_1 \cdot L_2 = \{w_1 w_2 \mid w_1 \in L_1, w_2 \in L_2\}$$

Пример: $\{abbc, cb\} \cdot \{ba, ccc\} = \{abbcba, abbccc, cbba, cbccc\}$

(c) **Итерация Клини**:

$$L^* = \bigcup_{k=0}^{\infty} L^k, \text{ где } L^0 = \{\varepsilon\},\ L^{k+1} = L \cdot L^k$$

Пример: $\{abb, c\}^* = \{\varepsilon, abb, c, abbabb, abbc, cc, cabb, \ldots\}$

(d) **Усечённая итерация**:

$$L^+ = \bigcup_{k=1}^{\infty} L^k$$

3. Связь регулярных выражений и языков

Регулярные выражения являются формализмом для описания регулярных языков. Каждое выражение соответствует языку, построенному через комбинацию операций:

- Базовые элементы:
  - $\emptyset$ соответствует пустому языку
  - ε — языку, содержащему только пустую цепочку
  - $a \in \Sigma$ — языку $\{a\}$
- Операции расширяют базовые языки:
  - $R_1 + R_2$ задаёт объединение их языков
  - $R_1 R_2$ — конкатенацию
  - R^* — итерацию

4. Важность регулярных языков

Регулярные языки играют ключевую роль в теории формальных языков и компиляторов. Они лежат в основе:

- Лексического анализа (распознавание токенов)
- Валидации строк (например, проверка email)
- Поиска шаблонов в тексте

Мощность регулярных языков: Хотя словарь Σ конечен, регулярные языки могут быть бесконечными (например, язык всех цепочек чётной длины). Однако их мощность ограничена — они образуют счётное множество, в отличие от всех возможных языков над Σ, мощность которых континуум.

1.2 Регулярное выражение для данной работы

Для региона, названия улицы и города используется аналогичное выражение:

```
[А-Я]+ (? : [- ])[А-Я]+)*
```

Выражение целиком:

```

[А-Я]+ (? : [- ])[А-Я]+)* \s* ;\s* [А-Я]+ (? : [- ])[А-Я]+)* \s* ;\s*

(улица|бульвар|проспект|переулок|аллея|линия)
[А-Я]+ (? : [- ])[А-Я]+)* \s* ;\s*

 д. [1-9][0-9]* (кв.\s* ;\s*[1-9][0-9]*|\s* ;\s*)
```

Регулярные множества:

`[А-Я]+ - последовательность русских заглавных букв.`

`(?:[- ]) - дефис или пробел.`

`\s* - пробельные символы.`

`; - точка с запятой.`

`(улица|бульвар|проспект|переулок|аллея|линия) - тип улицы.`

`[1-9][0-9]* - натуральное число (без ведущих нулей).`

`д. и кв. - литералы для обозначения дома и квартиры.`

1.3 Недетерминированные конечные автоматы (НДКА)

1.3.1 Определение и особенности

Недетерминированный конечный автомат (НДКА) — это математическая модель, описываемая пятёркой $(Q, \Sigma, \delta, q_0, F)$, где:

- Q — конечное множество состояний,
- Σ — конечный входной алфавит,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$ — функция переходов,
- $q_0 \in Q$ — начальное состояние,
- $F \subseteq Q$ — множество допускающих состояний.

Ключевые особенности НДКА:

- Возможность **неоднозначных переходов**
- Наличие ε**-переходов**
- **Несколько начальных состояний** (в обобщённой модели)

1.3.2 ε-переходы

ε-переходы полезны для:

- Моделирования внутренних событий
- Упрощения конструкции автомата
- Реализации неявных состояний

1.3.3 НДКА как распознаватели языков

1.3.4 Язык, допускаемый НДКА

Недетерминированные конечные автоматы являются важным инструментом для распознавания формальных языков, особенно регулярных.

Преимущества НДКА как распознавателей

- Компактность представления:
  - Меньше состояний чем эквивалентный ДКА
  - Проще конструировать для сложных шаблонов
- Гибкость:
  - eps-переходы упрощают композицию автоматов
  - Легко комбинировать через объединение, конкатенацию, итерацию
- Прямое соответствие:
  - Близкое соответствие регулярным выражениям
  - Простота преобразования regex -> НДКА

1.3.5 НДКА как генераторы языков

Недетерминированные конечные автоматы (НДКА) могут использоваться не только для распознавания, но и для генерации строк принадлежащих языку. В качестве генератора НДКА работает следующим образом:

- На каждом шаге случайным образом выбирается один из возможных переходов
- При наличии eps-переходов автомат может менять состояние без генерации символа
- Процесс продолжается пока не будет достигнуто конечное состояние

1.3.6 Преимущества и ограничения НДКА

Преимущества	**Недостатки**
Компактное представление языков	Сложность реализации
Простота операций над автоматами	Неочевидная семантика
Естественное моделирование параллелизма	Требуется детерминизация

1.3.7 Автоматы в данной работе

На рис.1 представлен недетерминированный конечный автомат для разбора адреса.

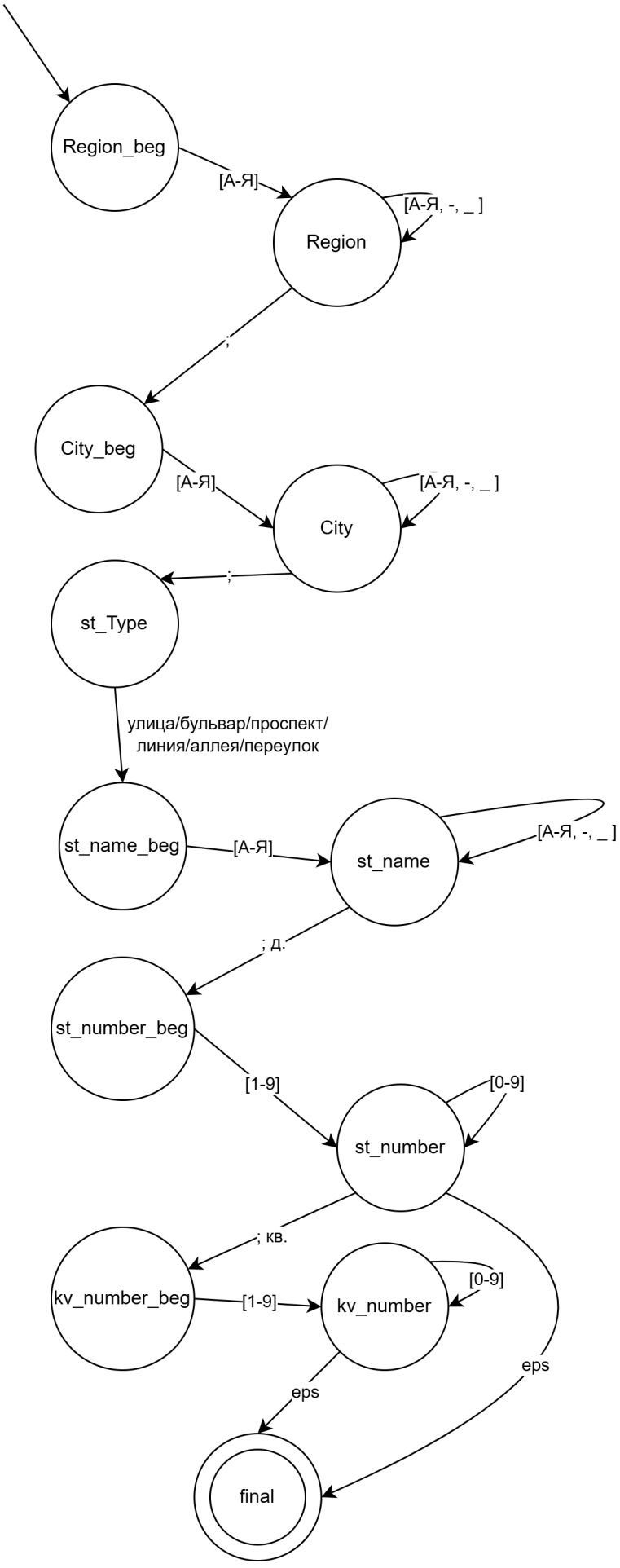

Рис. 1: Недетерминированный конечный автомат

Таблица 1: Таблица переходов НДКА

Текущее состояние	Входной символ	Следующее состояние
Region_beg	[А-Я]	Region
Region	[А-Я, -, _]	Region
Region	;	City_beg
City_beg	[А-Я]	City
City	[А-Я, -, _]	City
City	;	st_Type
st_Type	улица|бульвар|аллея| линия|проспект|переулок	st_name_beg
st_name_beg	[А-Я]	st_name
st_name	[А-Я, -, _]	st_name
st_name	; д.	st_number_beg
st_number_beg	[1-9]	st_number
st_number	[0-9]	st_number
st_number	; кв.	w_number_beg
st_number	eps	final
kv_number_beg	[1-9]	w_number
kv_number	[0-9]	w_number
kv_number	eps	final

Конечные состояния детерминированного конечного автомата - все состояния, содержащие final:

q0 = st_number, final - конечное

q2 = kv_number, final - конечное

Состояния:

- q0 = st_number, final
- q1 = kv_number_beg
- q2 = kv_number, final

На рис.2 представлен ДКА для данной программы.

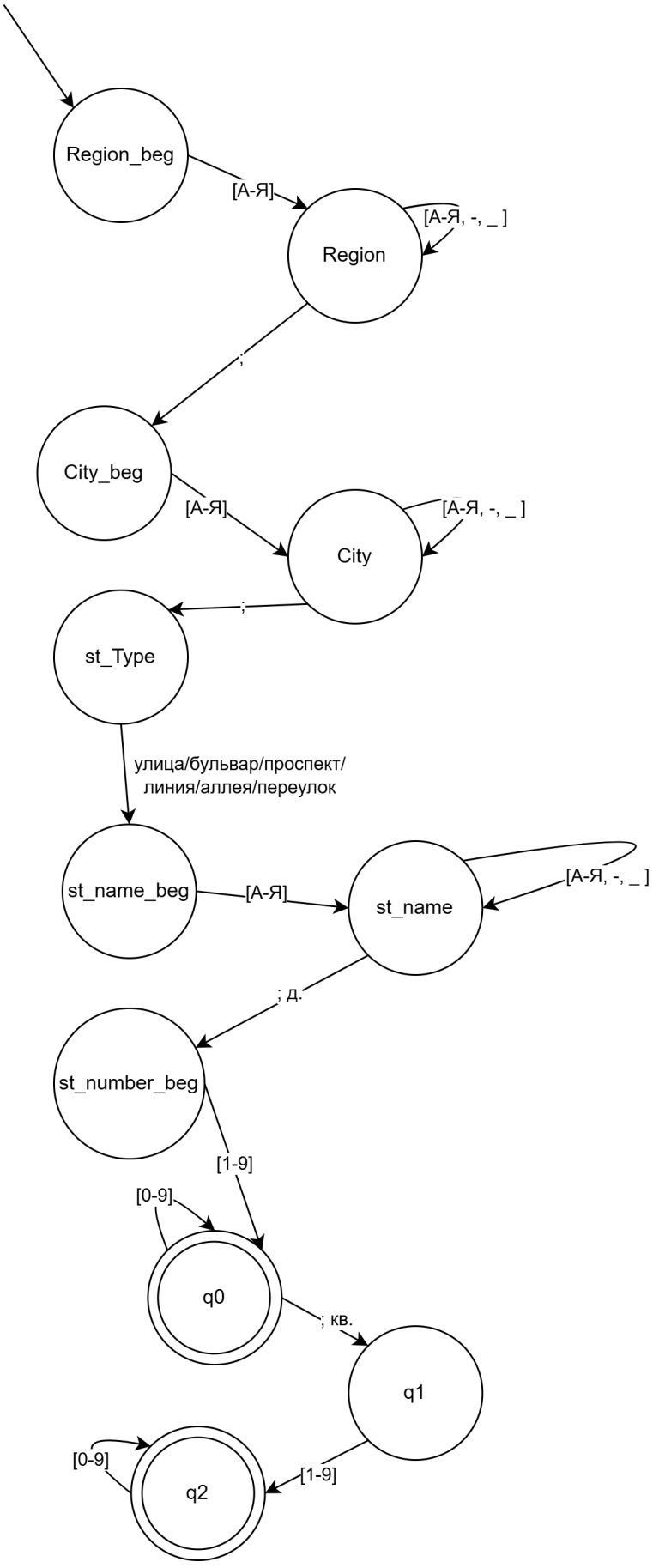

Рис. 2: ДКА для данной работы

Таблица 2: Таблица переходов ДКА

Текущее состояние	Входной символ	Следующее состояние
Region_beg	[А-Я]	Region
Region	[А-Я, -, _]	Region
Region	;	City_beg
City_beg	[А-Я]	City
City	[А-Я, -, _]	City
City	;	st_Type
st_Type	улица|бульвар|аллея| линия|проспект|переулок	st_name_beg
st_name_beg	[А-Я]	st_name
st_name	[А-Я, -, _]	st_name
st_name	; д.	st_number_beg
st_number_beg	[1-9]	q0
q0	[0-9]	q0
q0	; кв.	q1
q1	[1-9]	q2
q2	[0-9]	q2

2 Особенности реализации

2.1 Структуры данных

1. **Множество**

   `allowed_chars` - множество разрешенных символов в адресе. Используется для быстрой проверки принадлежности символа к допустимым.

2. **Словарь**

   `states` - словарь, который сопоставляет названия состояний (этапов разбора адреса) с числовыми значениями. Это помогает управлять текущим состоянием парсера.

   `street_types` - словарь, где ключи - это названия типов улиц (например, "улица "бульвар"), а значения - списки символов, из которых состоят эти названия. Используется для проверки корректности типа улицы.

3. **Списки**

   В словаре `street_types` значения представляют собой списки символов, которые последовательно проверяются при разборе типа улицы.

4. **Строки**

   Сам адрес и его части обрабатываются как строки. В коде используются методы строк, такие как `isalpha()`, `isdigit()`, `isupper()`, а также индексация и срезы для проверки символов.

5. **Переменные-флаги**

   `house_number_started` и `flat_number_started` - булевы переменные, которые используются для отслеживания, начался ли номер дома или квартиры (чтобы проверить, что они не начинаются с нуля).

   `current_state` - переменная, которая хранит текущее состояние парсера (например, разбор региона, города и т.д.).

   `prev_char` - переменная, которая хранит предыдущий символ для проверки корректности последовательности (например, чтобы не было двух пробелов подряд).

   `street_type_step` - переменная, которая отслеживает текущую позицию в разборе типа улицы.

   `expected_street_type` - переменная, которая хранит ожидаемый тип улицы во время разбора.

2.2 Описание функции `check_address`

2.2.1 Назначение функции

Функция выполняет валидацию строки почтового адреса согласно заданному формату.

2.2.2 Сигнатура функции

```
def check_address(address: str) -> str:
```

2.2.3 Входные и выходные данные

- **Вход**: Строка `address` в формате:

```
"РЕГИОН; ГОРОД; ТИП_УЛИЦЫ НАЗВАНИЕ_УЛИЦЫ; д.НОМЕР_ДОМА; кв.НОМЕР_КВАРТИРЫ"
```

- **Выход**:
  - `"соответствует"` — если адрес валиден
  - `"не соответствует"` — если адрес не соответствует формату
  - `"символы не из алфавита"` — при наличии недопустимых символов

2.2.4 Алгоритм работы

1. Проверка символов:

```
if any(c.upper() not in allowed_chars...):
    return "символы не из алфавита"
```

2. Разбор компонентов через конечный автомат:
   - Регион → Город → Тип улицы → Название улицы → Номер дома → Номер квартиры
3. Проверка формата:
   - Регион/город: только заглавные буквы
   - Тип улицы: соответствие шаблонам
   - Номера: начинаются не с нуля

Переменная current_state отслеживает текущее состояние автомата. Начинается автомат с состояния region (проверка региона).

Автомат переходит между состояниями в строго определённом порядке, анализируя символы входной строки (address). Переходы происходят при обнаружении определённых символов (например, точки с запятой ;) или после полной проверки определённой части адреса (например, типа улицы).

Примеры переходов:

- Из region в city при обнаружении ;.
- Из city в street_type при обнаружении ;.
- Из street_type в street_name после полной проверки типа улицы (например, "улица "проспект").
- Из street_name в house_prefix (и далее в house_number) при обнаружении ;.
- Из house_number в flat_prefix (и далее в flat_number) при обнаружении ;.

2.2.5 Код функции

```
def check_address(address):
    allowed_chars = set('АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ -;0123456789улицабврптккпереулокаллеялинияеосяндкв.')
    if any(c.upper() not in allowed_chars and c != ';' for c in address):
        return "символы не из алфавита"

    states = {
        'region': 0,
        'city': 1,
        'street_type': 2,
        'street_name': 3,
        'house_prefix': 4,
        'house_number': 5,
        'flat_prefix': 6,
        'flat_number': 7
    }
    current_state = states['region']
    prev_char = None
    street_type_step = 0
    expected_street_type = None
    street_types = {
        'улица': ['у', 'л', 'и', 'ц', 'а'],
        'бульвар': ['б', 'у', 'л', 'ь', 'в', 'а', 'р'],
        'проспект': ['п', 'р', 'о', 'с', 'п', 'е', 'к', 'т'],
        'переулок': ['п', 'е', 'р', 'е', 'у', 'л', 'о', 'к'],
        'аллея': ['а', 'л', 'л', 'е', 'я'],
        'линия': ['л', 'и', 'н', 'и', 'я']
    }

    house_number_started = False
    flat_number_started = False

    i = 0
    n = len(address)

    while i < n:
        c = address[i]

        if current_state == states['region']:
            if c == ' ':
                i += 1
                continue
            if i == 0:
                if not (c.isalpha() and c.isupper()):
                    return "не соответствует"
                prev_char = c
                i += 1
            else:
                if c == ';':
                    if prev_char in [' ', '-']:
                        return "не соответствует"
                    current_state = states['city']
                    prev_char = None
                    i += 1
                    while i < n and address[i] == ' ':
                        i += 1
                else:
                    if not (c.isalpha() and c.isupper() or c in [' ', '-']):
```

```
58                          return "не соответствует"
59                      if prev_char in [' ', '-'] and c in [' ', '-']:
60                          return "не соответствует"
61                      prev_char = c
62                      i += 1
63
64          elif current_state == states['city']:
65              if c == ' ':
66                  i += 1
67                  continue
68              if prev_char is None:
69                  if not (c.isalpha() and c.isupper()):
70                      return "не соответствует"
71                  prev_char = c
72                  i += 1
73              else:
74                  if c == ';':
75                      if prev_char in [' ', '-']:
76                          return "не соответствует"
77                      current_state = states['street_type']
78                      prev_char = None
79                      i += 1
80                      while i < n and address[i] == ' ':
81                          i += 1
82                  else:
83                      if not (c.isalpha() and c.isupper() or c in [' ',
                            '-']):
84                          return "не соответствует"
85                      if prev_char in [' ', '-'] and c in [' ', '-']:
86                          return "не соответствует"
87                      prev_char = c
88                      i += 1
89
90          elif current_state == states['street_type']:
91              if expected_street_type is None:
92                  if c == 'у':
93                      expected_street_type = 'улица'
94                      street_type_step = 1
95                      i += 1
96                  elif c == 'б':
97                      expected_street_type = 'бульвар'
98                      street_type_step = 1
99                      i += 1
100                 elif c == 'п':
101                     if i + 1 >= n:
102                         return "не соответствует"
103                     next_c = address[i+1]
104                     if next_c == 'р':
105                         expected_street_type = 'проспект'
106                         street_type_step = 2
107                         i += 2
108                     elif next_c == 'е':
109                         expected_street_type = 'переулок'
110                         street_type_step = 2
111                         i += 2
112                     else:
113                         return "не соответствует"
114                 elif c == 'а':
115                     expected_street_type = 'аллея'
116                     street_type_step = 1
117                     i += 1
```

```
118                 elif c == 'л':
119                     expected_street_type = 'линия'
120                     street_type_step = 1
121                     i += 1
122                 else:
123                     return "не соответствует"
124             else:
125                 expected_char = street_types[expected_street_type][street
                        _type_step]
126                 if c != expected_char:
127                     return "не соответствует"
128                 street_type_step += 1
129                 i += 1
130                 if street_type_step == len(street_types[expected_street_
                        type]):
131                     if i >= n or address[i] != ' ':
132                         return "не соответствует"
133                     i += 1
134                     while i < n and address[i] == ' ':
135                         i += 1
136                     current_state = states['street_name']
137                     prev_char = None
138 
139         elif current_state == states['street_name']:
140             if c == ';':
141                 if prev_char in [' ', '-']:
142                     return "не соответствует"
143                 current_state = states['house_prefix']
144                 prev_char = None
145                 i += 1
146                 while i < n and address[i] == ' ':
147                     i += 1
148                 # Проверяем префикс дома
149                 if i + 1 >= n or address[i] != 'д' or address[i+1] !=
                        '.':
150                     return "не соответствует"
151                 i += 2
152                 # Пропустить пробел после точки, если есть
153                 if i < n and address[i] == ' ':
154                     i += 1
155                 current_state = states['house_number']
156                 house_number_started = False
157             else:
158                 if not (c.isalpha() and c.isupper() or c in [' ', '-']):
159                     return "не соответствует"
160                 if prev_char in [' ', '-'] and c in [' ', '-']:
161                     return "не соответствует"
162                 prev_char = c
163                 i += 1
164 
165         elif current_state == states['house_number']:
166             if c == ';':
167                 current_state = states['flat_prefix']
168                 prev_char = None
169                 i += 1
170                 while i < n and address[i] == ' ':
171                     i += 1
172                 # Проверяем префикс квартиры
173                 if i + 2 >= n or address[i] != 'к' or address[i+1] != 'в'
                        or address[i+2] != '.':
174                     return "не соответствует"
```

```
                    i += 3
                    # Пропустить пробел после точки, если есть
                    if i < n and address[i] == ' ':
                        i += 1
                    current_state = states['flat_number']
                    flat_number_started = False
                elif not c.isdigit():
                    return "не соответствует"
                else:
                    if not house_number_started:
                        house_number_started = True
                        if c == '0':
                            return "не соответствует"
                    i += 1

            elif current_state == states['flat_number']:
                if not c.isdigit():
                    return "не соответствует"
                else:
                    if not flat_number_started:
                        flat_number_started = True
                        if c == '0':
                            return "не соответствует"
                    i += 1

        if current_state in [states['house_number'], states['flat_number']]:
            if current_state == states['house_number'] and not house_number_
               started:
                return "не соответствует"
            if current_state == states['flat_number'] and not flat_number_
               started:
                return "не соответствует"
            return "соответствует"
        else:
            return "не соответствует"
```

2.3 Функция generate_text_part()

Назначение: Генерирует текстовую часть адреса (регион, город или название улицы).
Параметры:

- `min_len` (по умолчанию 3) - минимальная длина строки
- `max_len` (по умолчанию 15) - максимальная длина строки

Возвращаемое значение: Строка, соответствующая требованиям к части адреса.
Логика работы:

1. Начинается с заглавной буквы
2. Может содержать дефисы и пробелы (но не подряд)
3. Удаляет лишние разделители в конце строки
4. Рекурсивно перегенерирует, если результат слишком короткий

2.4 Функция generate_valid_address()

Назначение: Генерирует полный валидный почтовый адрес.

Параметры: Нет входных параметров.

Возвращаемое значение: Строка адреса в формате: "РЕГИОН; ГОРОД; ТИП_УЛИЦЫ НАЗВАНИЕ; д.НОМЕР[; кв.НОМЕР]"

Логика работы:

1. Использует конечный автомат с состояниями из словаря `states`
2. Последовательно генерирует компоненты адреса:
   - Регион (3-15 заглавных букв)
   - Город (3-10 заглавных букв)
   - Тип улицы (из предопределенного списка)
   - Название улицы (3-15 символов)
   - Номер дома (1-999)
   - Номер квартиры (1-999, добавляется в 70% случаев)
3. Соблюдает формальные требования:
   - Правильное использование разделителей
   - Номера не начинаются с нуля
   - Пробелы после точек опциональны

2.4.1 Код функции

```
def generate_valid_address():
    current_state = states['region']
    address_parts = []
    chars = 'АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ'
    street_types = ['улица', 'бульвар', 'проспект', 'переулок', 'аллея',
       'линия']

    def generate_text_part(min_len=3, max_len=15):
        part = []
        length = random.randint(min_len, max_len)
        part.append(random.choice(chars))

        for _ in range(length - 1):
            choice = random.choices([0, 1, 2], weights=[10, 1, 1], k=1)
               [0]

            if choice == 1 and part[-1] not in ['-', ' ']:
                part.append('-')
            elif choice == 2 and part[-1] not in ['-', ' ']:
                part.append(' ')
            else:
                part.append(random.choice(chars))

        part_str = ''.join(part)
        part_str = re.sub(r'[- ]+$', '', part_str.strip())
        return part_str if len(part_str) >= min_len else generate_text_
           part(min_len, max_len)

    while current_state != states['flat_prefix']:  # Цикл до flat_prefix
```

```
27            if current_state == states['region']:
28                address_parts.append(generate_text_part(3, 15))
29                current_state = states['city']
30
31            elif current_state == states['city']:
32                address_parts.append(generate_text_part(3, 10))
33                current_state = states['street_type']
34
35            elif current_state == states['street_type']:
36                street_type = random.choice(street_types)
37                address_parts.append(street_type)
38                current_state = states['street_name']
39
40            elif current_state == states['street_name']:
41                street_name = generate_text_part(3, 15)
42                address_parts[-1] = f"{address_parts[-1]} {street_name}"
43                current_state = states['house_prefix']
44
45            elif current_state == states['house_prefix']:
46                house_space = ' ' if random.random() > 0.5 else ''
47                address_parts.append(f"д.{house_space}")
48                current_state = states['house_number']
49
50            elif current_state == states['house_number']:
51                address_parts[-1] = address_parts[-1] + str(random.randint(1,
                      999))
52                current_state = states['flat_prefix']
53
54        if random.random() > 0.3:  # 70% chance to add flat
55            flat_space = ' ' if random.random() > 0.5 else ''
56            address_parts.append(f"кв.{flat_space}{random.randint(1, 999)}")
57
58        return '; '.join(address_parts)
```

3 Результаты работы программы

Далее представлены результаты проверки нескольких адресов на соответствие.

```
Адрес: РЕГИОН; ГОРОД; аллея ПАРКОВАЯ;д. 5; кв.1
Результат: соответствует

Адрес: МОСКОВСКАЯ ОБЛАСТЬ; ДЕДОВСК; переулок ШКОЛЬНЫЙ; д.1; кв.1
Результат: соответствует

Адрес: КРАЙ; ПОСЕЛОК; линия ВЕРХНЯЯ; д.3; кв.5
Результат: соответствует

Адрес: НИЖЕГОРОДСКАЯ ОБЛАСТЬ; САРОВ; проспект МИРА; д.2
Результат: соответствует

Адрес: РЕСПУБЛИКА ТАТАРСТАН; КАЗАНЬ; улица КРЕМЛЁВСКАЯ; д.1; кв.1
Результат: соответствует

Адрес: МОСКВА;; улица ТВЕРСКАЯ; д.10
Результат: не соответствует   \\нет города

Адрес: РЕСПУБЛИКА БАШКОРТОСТАН; ЧИШМЫ; улица РЕВОЛЮЦИОННАЯ; д.135; кв.0
Результат: не соответствует   \\квартира 0
```

Далее представлены результаты генерации адресов и проверки их на соответствие.

```
Сгенерирован: ЗЮЛФЯЯЫШВХЭПЮ-Л; ПФЗЦ-ЗО; улица ЕЖЯВНЭШ; д.981; кв. 637
Проверка: соответствует

Сгенерирован: ЭЮЫОЖПМФ; Ч-СЬТШ Ч; аллея ЖПЫЪУМ; д. 445; кв.636
Проверка: соответствует

Сгенерирован: О ОНКМТЪС ЕИВ; ЕФЕДНИЬЧЁ; бульвар ЁЮБУ; д. 251; кв.704
Проверка: соответствует

Сгенерирован: ОЭЧЫ-УЁГ-Х-ШЮЛ; ЭЬ Я; улица У-СЕЫИФЖСАП; д. 905
Проверка: соответствует

Сгенерирован: НЕЮЪХДЕЮГРЩЛК; УЮЙУАЪ; линия АТХЭД; д. 683; кв. 868
Проверка: соответствует
```

Заключение

В ходе данной работы было составлено регулярное выражение соответствующее структуре почтового адреса, а также был составлен конечный автомат, распознающий строку с адресом. В данной работе язык почтовых адресов является регулярным, поскольку его можно описать конечным автоматом без необходимости использования стека (в отличие от контекстно-свободных языков). Согласно теореме Клини, класс языков, распознаваемых конечными автоматами, совпадает с классом регулярных языков. Разработанное регулярное выражение для адресов является конечным автоматом в другой форме, что подтверждает их эквивалентность. На основе ДКА была разработана программа для генерации и валидации почтовых адресов на Python, основанная на концепции конечного автомата.

Достоинства

- Поддержка опциональных компонентов (номера квартир);
- Возможность регулирования длины генерируемых частей адреса;
- Простота добавления новых состояний или модификации существующих;

Недостатки

- Генерация случайных сочетаний букв вместо реальных топонимов;
- Строгая последовательность компонентов адреса;

Масштабируемость

Благодаря реализации через конечный автомат довольно легко добавить новое состояние или модифицировать существующее. Также в данную реализацию легко добавить поддержку распознавания и генерации адреса на иностранном языке.

Список литературы

[1] Электронный ресурс ВШТИИ
URL:https://tema.spbstu.ru/compiler/
(Дата обращения: 13.03.2025).

[2] Карпов, Ю. Г. Теория автоматов, Санкт-Петербург : Питер, 2003.
URL:https://djvu.online/file/eeLVKnyRZPXfl
(Дата обращения: 12.03.2025).