МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение
высшего 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 №2
«Конечный автомат и регулярное выражение для распознавания форматов вещественных чисел»
по дисциплине
«Математическая логика и теория автоматов»
Вариант 15

Студент,
группы 5130201/20102 _____ Тищенко А. А.

Преподаватель _____ Востров А. В.

«_____»_____ 2025г.

Санкт-Петербург, 2025

Содержание

Введение

Лабораторная №2 по дисциплине «Математическая логика» заключается в следующем. Необходимо построить регулярное выражение для заданного варианта, затем создать недетерминированный конечный автомат и детерминировать его. Реализовать программу, которая проверяет введённый текст через реализацию конечного автомата, с вариантами вывода: строка соответствует, не соответствует, символы не из алфавита. Также необходимо реализовать функцию случайной генерации верной строки по полученному конечному автомату.

Вариант 15. Соответствие вещественного числа разным форматам представления.

В данной лабораторной работе рассматриваются следующие форматы представления вещественных чисел.

- Целые числа, например: ''123'', ''-456'', ''0'', в т. ч. ''+0'', ''-0''.
- Десятичные числа с разделителем точка, например: ''123.456'', ''-456.789'', ''.5''.
- Экспоненциальная форма (буква E может быть как в верхнем, так и в нижнем регистре), например: ''1.23E4'', ''-4.56e-7'', ''7e8''.

Формальное определение синтаксиса предложенного формата вещественных чисел в БНФ нотации:

```
real           ::= [sign] (integer | float | exponentnumber)
sign           ::= "+" | "-"

integer        ::= nonzerodigit {digit} | "0" {"0"}
nonzerodigit   ::= "1" | "2" | ... | "9"
digit          ::= "0" | nonzerodigit

float          ::= {digit} "." digitpart | digitpart "."
digitpart      ::= digit {digit}

exponentnumber ::= (digitpart | float) exponent
exponent       ::= ("e" | "E") [sign] digitpart
```

Где:

- [R]– необязательный элемент (0 или 1 раз)
- {R} – повторение элемента (0 или более раз)
- P|Q – альтернатива (либо P, либо Q)

1 Математическое описание

1.1 Языки и грамматики

Языком над конечным словарем Σ называется произвольное множество конечных цепочек над этим словарем.

- Цепочки языка называются словами (предложениями).
- Над конечным непустым словарем можно определить бесконечное количество слов конечной длины (счетное множество).
- Над конечным непустым словарем можно определить бесконечное количество языков, т.е. подмножество множества всех возможных слов (континуум, как число подмножеств счетного множества).

Языки могут быть конечными и бесконечными (содержать бесконечное число цепочек). Словарь всегда конечен.

Формальная грамматика – способ описания того, какие предложения возможны в языке. Существует два вида грамматик:

- порождающие грамматики – правила, позволяющие строить любое предложение языка,
- распознающие грамматики (алгоритмы) - позволяют определить, принадлежит ли данное предложение языку.

Распознающая грамматика – это конечный набор правил, алгоритм, который по введенной цепочке определяет, принадлежит цепочка языку, или нет.

1.2 Регулярные множества

Регулярные множества, как множества цепочек, построенные над конечным словарем (по определенным правилам) – это языки. Их называют регулярными языками.

Правила построения регулярных множеств:

- Объединение двух регулярных множеств L_1 и L_2 обозначается $L_1 \cup L_2$ и состоит из цепочек, которые принадлежат хотя бы одному из множеств L_1 или L_2.

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

- Конкатенация (произведение) двух регулярных множеств L_1 и L_2 обозначается $L_1 \cdot L_2$ и состоит из цепочек, которые можно разбить на две части, одна из которых принадлежит L_1, а другая L_2.

$$L_1 \cdot L_2 = \{\alpha\beta \mid \alpha \in L_1 \text{ и } \beta \in L_2\}$$

  Обозначим $L^0 = \{\varepsilon\}$, $L^1 = L$, $L^2 = L \cdot L$, $L^{k+1} = L^k \cdot L$, и так далее. Обозначение ε обозначает пустую цепочку.

- Итерация регулярного множества L обозначается L^* и состоит из цепочек, которые можно разбить на произвольное количество повторений множества L.

$$L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \dots$$

1.3 Регулярные выражения

Регулярные выражения – формальный язык шаблонов для поиска и выполнения манипуляций с подстроками в тексте. Регулярное выражение – это формула (pattern, шаблон), задающая правило поиска подстрок в потоке символов.

Регулярное выражение показывает, как можно построить регулярное множество цепочек из одноэлементных множеств с использованием трех операций: конкатенации, объединения и итерации.

Примеры регулярных выражений:

- $ab + ba^*$ – представляет регулярное множество:

$$\{a\}\{b\} \cup \{b\}\{a\}^*$$

- $(ac)^*b + c^*$ – представляет регулярное множество:

$$\{b, acb, acacb, acacacb, \dots, \varepsilon, c, cc, ccc, \dots\}$$

В реальных программах и языках программирования используется расширенный синтаксис регулярных выражений, который добавляет множество удобных конструкций. Среди дополнительных возможностей: классы символов (например, `[a-z0-9]`), квантификаторы (`?`, `+`, `{n,m}`), группировка с помощью скобок, обратные ссылки, опережающие и ретроспективные проверки. Эти расширения делают регулярные выражения более компактными и удобными для использования, но не меняют их выразительную мощность с теоретической точки зрения.

Если регулярные множества это языки, то регулярные выражения – это распознающие грамматики этих языков.

1.4 Регулярные выражения для заданного варианта

Для разных форматов представления вещественных чисел были построены следующие регулярные выражения в соответствии с определённой БНФ нотацией:

- Целые числа:
  `[+-]?(0+|[1-9][0-9]*)`
- Десятичные числа:
  `[+-]?([0-9]*\.[0-9]+|[0-9]+\.)`
- Экспоненциальная форма:
  `[+-]?([0-9]+|[0-9]*\.[0-9]+|[0-9]+\.)[eE][+-]?[0-9]+`

Объединяя, получаем следующее регулярное выражение, которое распознает все форматы вещественных чисел в соответствии с БНФ нотацией:

```
[+-]?(
    0+|
    [1-9][0-9]*|
    [0-9]*\.[0-9]+|
    [0-9]+\.|
    ([0-9]+|[0-9]*\.[0-9]+|[0-9]+\.)[eE][+-]?[0-9]+
)
```

Разберём структуру этого выражения:

- `[+-]?` – необязательный знак числа (плюс или минус)
- `0+|[1-9][0-9]*` – целое число, которое может быть либо последовательностью нулей, либо цифрой от 1 до 9, за которой следует произвольное количество цифр.
- `[0-9]*\.[0-9]+|[0-9]+\.` – десятичное число, которое может быть представлено произвольным количеством цифр до и как минимум одной цифрой после точки, либо произвольным количеством цифр и одной точкой в конце.
- `([0-9]+|[0-9]*\.[0-9]+|[0-9]+\.)[eE][+-]?[0-9]+` – экспоненциальная форма числа, состоящая из произвольного количества цифр, либо десятичного числа, за которым следует буква E (в любом регистре), необязательный знак и как минимум одна цифра.

Таким образом, полученное регулярное выражение распознаёт формат представления вещественных чисел, рассматриваемый в данной работе, и полностью соответствует формальному определению, представленному в БНФ нотации.

1.5 Конечный автомат-распознаватель

Конечный автомат-распознаватель – это математическая модель, которая используется для распознавания цепочек символов в соответствии с заданным формальным языком.

Конечный автомат-распознаватель $A = (S, \Sigma, s_0, \delta, F)$, где:

- S – конечное множество состояний
- Σ – конечное множество входных символов
- $s_0 \in S$ – начальное состояние
- $\delta : S \times \Sigma \to S$ – функция переходов
- $F \subseteq S$ – множество финальных (допускающих) состояний

Автомат A допускает (распознает) цепочку, если эта цепочка переводит A из начального в одно из финальных состояний. Автомат A допускает язык L, если он допускает все цепочки этого языка – и только их.

Конечный автомат-распознаватель является распознающей грамматикой.

1.6 Недетерминированный КА-распознаватель

Недетерминизм - очень удобное свойство формальной модели, его можно определенным образом трактовать, даже и не реализовывая, ограничиваясь только формальными аналитическими преобразованиями.

В недетерминированном конечном автомате-распознавателе могут быть следующие неоднозначности:

- несколько начальных состояний,
- несколько переходов, помеченных одним и тем же символом,
- переходы, помеченные пустым символом ε.

Цепочка допускается конечным автоматом, если существует путь, по которому эта цепочка переводит автомат из какого-нибудь начального состояния в какое-нибудь финальное состояние.

Недетерминированный конечный автомат-распознаватель $A = (S, \Sigma, S_0, \delta, F)$, где:

- S – конечное множество состояний
- Σ – конечное множество входов
- $S_0 \subseteq S$ – множество начальных состояний
- $\delta : S \times \Sigma \to 2^S$ – функция переходов
- $F \subseteq S$ – множество финальных состояний

1.7 Теорема Клини

Теорема Клини. Классы регулярных множеств и автоматных языков совпадают. Это значит, что:

- Любой язык, распознаваемый конечным автоматом, может быть задан регулярным выражением.
- Для любого регулярного выражения существует конечный автомат, распознающий соответствующий язык.

1.8 Недетерминированный КА-распознаватель для заданного варианта

На Рис. 1 представлен недетерминированный КА-распознаватель, соответствующий регулярному выражению для заданного варианта.

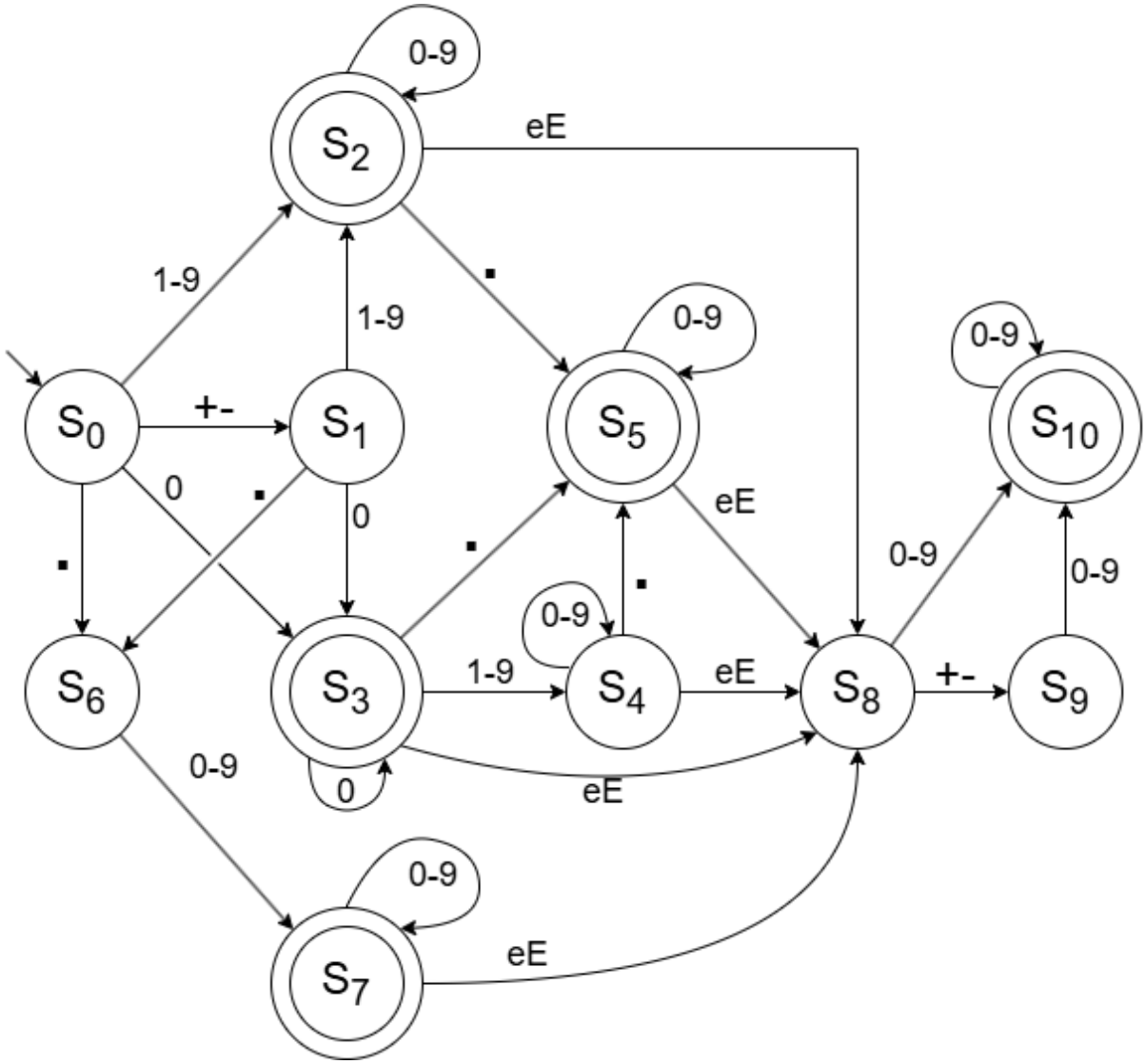

Рис. 1. Недетерминированный КА-распознаватель для заданного варианта.

Матрица переходов для данного автомата представлена в Таблице 1.

1.9 Детерминированный КА-распознаватель для заданного варианта

Для того, чтобы преобразовать недетерминированный КА-распознаватель (Рис. 1) в детерминированный, достаточно добавить ещё одно состояние S_E, соответствующее недопустимой цепочке символов, и переходы в него из всех остальных состояний.

Таблица 1. Таблица переходов для недетерминированного КА-распознавателя.

Состояние\Вход	+-	**0**	**1-9**	**.**	**eE**
S_0	S_1	S_3	S_2	S_6	–
S_1	–	S_3	S_2	S_6	–
S_2	–	S_2	S_2	S_5	S_8
S_3	–	S_3	S_4	S_5	S_8
S_4	–	S_4	S_4	S_5	S_8
S_5	–	S_5	S_5	–	S_8
S_6	–	S_7	S_7	–	–
S_7	–	S_7	S_7	–	S_8
S_8	S_9	S_{10}	S_{10}	–	–
S_9	–	S_{10}	S_{10}	–	–
S_{10}	–	S_{10}	S_{10}	–	–

На Рис. 2 представлен детерминированный КА-распознаватель. Символом C (от англ. Complement – дополнение) обозначены переходы, соответствующие любым символам, кроме тех, по которым уже есть переходы в другие состояния. Символом A (от англ. Any – любой) обозначен переход, соответствующий любому символу.

Таблица 2. Таблица переходов для детерминированного КА-распознавателя.

Состояние\Вход	+-	**0**	**1-9**	**.**	**eE**
S_0	S_1	S_3	S_2	S_6	S_E
S_1	S_E	S_3	S_2	S_6	S_E
S_2	S_E	S_2	S_2	S_5	S_8
S_3	S_E	S_3	S_4	S_5	S_8
S_4	S_E	S_4	S_4	S_5	S_8
S_5	S_E	S_5	S_5	S_E	S_8
S_6	S_E	S_7	S_7	S_E	S_E
S_7	S_E	S_7	S_7	S_E	S_8
S_8	S_9	S_{10}	S_{10}	S_E	S_E
S_9	S_E	S_{10}	S_{10}	S_E	S_E
S_{10}	S_E	S_{10}	S_{10}	S_E	S_E

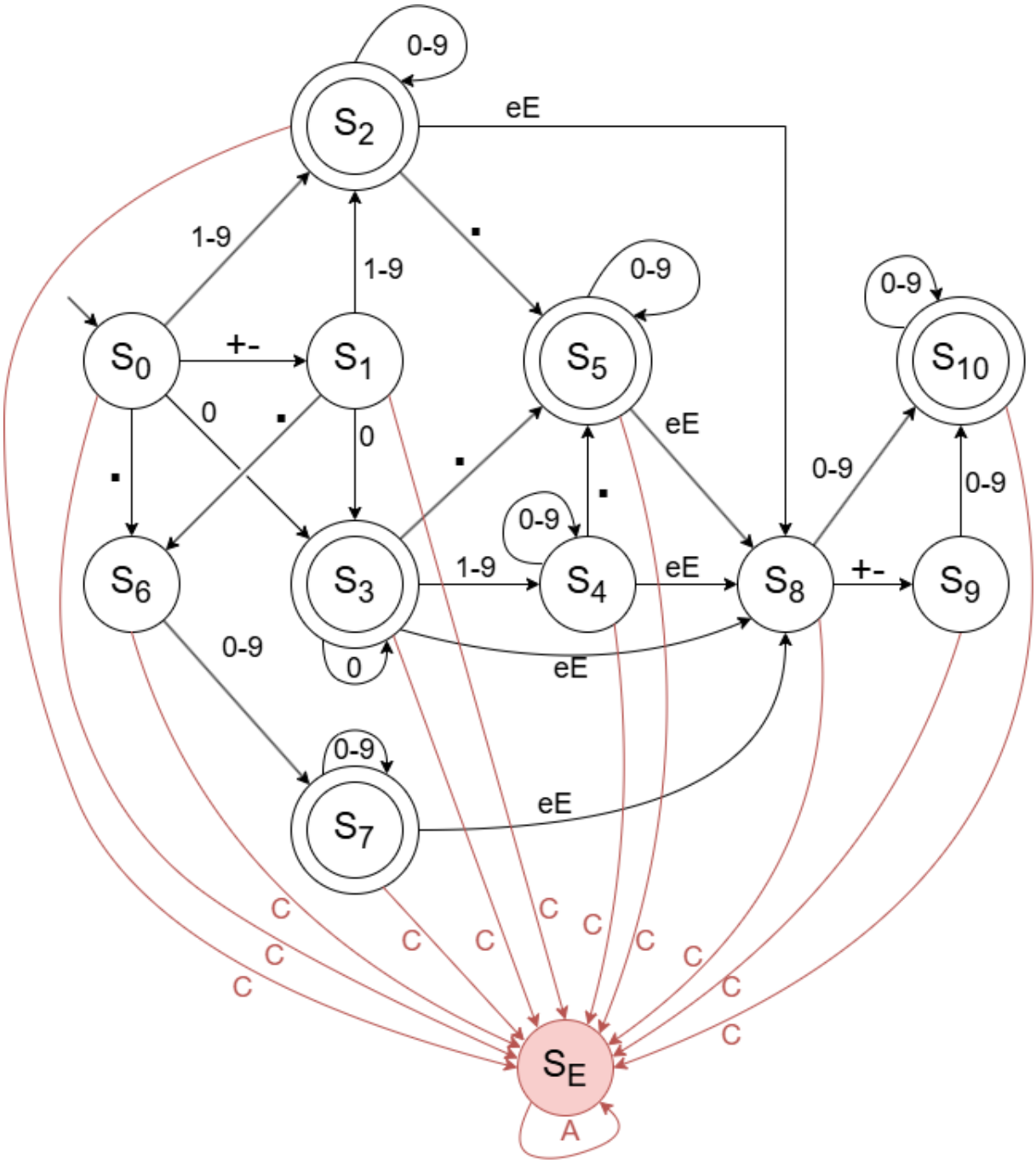

Рис. 2. Детерминированный КА-распознаватель для заданного варианта.

Матрица переходов для данного автомата представлена в Таблице 2.

2 Особенности реализации

2.1 Общая структура программы

Программа состоит из двух файлов:

- `finite_automaton.py` – содержит класс `FiniteAutomaton` для создания конечных автоматов.
- `main.py` – содержит определение конечного автомата для распознавания вещественных чисел, а также функцию `main`, которая реализует интерактивный интерфейс для проверки и генерации строк.

2.2 Класс `FiniteAutomaton`

Класс `FiniteAutomaton` это класс для представления конечных автоматов. Код конструктора класса представлен в листинге 1.

В классе определены четыре поля.

- `transitions` – `dict[str, list[tuple[str, str]]]` – словарь, определяющий переходы между состояниями. Ключами словаря являются состояния, значениями – списки кортежей вида `(transition, next_state)`, определяющие переходы из данного состояния. `transition` это строка, содержащая символы, по которым можно перейти из данного состояния в следующее состояние – `next_state`.
- `initial_state` – `str` – начальное состояние.
- `final_states` – `set[str]` – множество финальных состояний.
- `alphabet` – `set[str]` – множество символов, которые может содержать входная строка.

Листинг 1. Код конструктора класса FiniteAutomaton.

```
1  class FiniteAutomaton:
2      def __init__(
3          self,
4          transitions: dict[str, list[tuple[str, str]]],
5          initial_state: str,
6          final_states: set[str],
7          alphabet: set[str],
8      ):
9          self.transitions = transitions
10         self.initial_state = initial_state
11         self.final_states = final_states
12         self.alphabet = alphabet
```

Класс `FiniteAutomaton` содержит три метода: `_get_next_state`, `process_input` и `generate_random_string`.

2.3 Метод `_get_next_state`

Приватный метод `_get_next_state` принимает три параметра: `self` – ссылку на объект класса, `current_state` (`str`) – текущее состояние, и `char` (`str`) – символ, по которому нужно перейти из текущего состояния. Возвращает следующее состояние (`str`), либо `None`, если переход невозможен.

Листинг 2. Код метода `_get_next_state`.

```
def _get_next_state(self, current_state: str, char: str) -> str | None:
    if current_state in self.transitions:
        for transition, next_state in self.transitions[current_state]:
            if char in transition:
                return next_state
    return None
```

2.4 Метод `process_input`

Метод `process_input` принимает два параметра: `self` – ссылку на объект класса, и входную строку `input_string`. Возвращает кортеж из двух элементов: строку с сообщением о результате проверки, и список пройденных состояний.

Листинг 3. Код метода `process_input`.

```
def process_input(self, input_string: str) -> tuple[str, list[str]]:
    current_state = self.initial_state
    transitions_path = [current_state]

    for char in input_string:
        if char not in self.alphabet:
            return f"Символ '{char}' не из алфавита", transitions_path

        next_state = self._get_next_state(current_state, char)

        if next_state is None:
            return "Строка не соответствует", transitions_path

        transitions_path.append(next_state)
        current_state = next_state

    return (
        "Строка соответствует"
        if current_state in self.final_states
        else "Строка не соответствует"
    ), transitions_path
```

2.5 Метод `generate_random_string`

Метод `generate_random_string` принимает два параметра: `self` – ссылку на объект класса, и вероятность остановки в финальном состоянии – `stop_probability` (`float`). Возвращает сгенерированную строку.

Шаг алгоритма заключается в следующем:

1. Проверяем условие остановки: если текущее состояние является финальным и случайное число оказывается меньше заданной вероятности остановки, то генерация завершается.
2. Случайным образом выбираем переход из текущего состояния в следующее.
3. Случайным образом выбираем символ из множества символов выбранного перехода.
4. Добавляем выбранный символ в результат и переходим в следующее состояние.

Листинг 4. Код метода `generate_random_string`.

```
1  def generate_random_string(
2      self, stop_probability: float = 0.3
3  ) -> tuple[str, list[str]]:
4      result = []
5      current_state = self.initial_state
6      path = [current_state]
7
8      while True:
9          if (
10             current_state in self.final_states
11             and random.random() < stop_probability
12         ):
13             break
14
15         transition, next_state = random.choice(self.transitions[current_state])
16
17         char = random.choice(transition)
18         result.append(char)
19         current_state = next_state
20         path.append(current_state)
21
22     return "".join(result), path
```

2.6 Функция `main`

В функции `main` (листинг 5) заданы параметры конечного автомата и реализован интерактивный интерфейс пользователя. Функция не принимает параметров и ничего не возвращает.

Пользователю доступны следующие команды:

- `check <строка>` – проверяет, соответствует ли строка автомату.
- `gen [<вероятность_остановки>]` – сгенерировать случайную строку.
- `q` – выход из программы.

Листинг 5. Функция main.

```
def main():
    alphabet = set("+−0123456789.eE")
    initial_state = "S0"
    final_states = {"S2", "S3", "S5", "S7", "S10"}

    transitions = {
        "S0": [("+−", "S1"), ("123456789", "S2"), ("0", "S3"), (".", "S6")],
        "S1": [("123456789", "S2"), ("0", "S3"), (".", "S6")],
        "S2": [("0123456789", "S2"), (".", "S5"), ("eE", "S8")],
        "S3": [("0", "S3"), ("123456789", "S4"), (".", "S5"), ("eE", "S8")],
        "S4": [("0123456789", "S4"), (".", "S5"), ("eE", "S8")],
        "S5": [("0123456789", "S5"), ("eE", "S8")],
        "S6": [("0123456789", "S7")],
        "S7": [("0123456789", "S7"), ("eE", "S8")],
        "S8": [("+−", "S9"), ("0123456789", "S10")],
        "S9": [("0123456789", "S10")],
        "S10": [("0123456789", "S10")],
    }

    automaton = FiniteAutomaton(
        transitions=transitions,
        initial_state=initial_state,
        final_states=final_states,
        alphabet=alphabet,
    )

    print("Конечный автомат для распознавания форматов вещественных чисел")
    print("=" * 60)
    print("Варианты команд:")
    print(" − check строка<> − проверить, соответствует ли строка автомату")
    print(
        " − gen вероятность[<остановки_>] − сгенерировать случайную строку по( умолчанию
      0.3)"
    )
    print(" − q − выход из программы")
    print("=" * 60)

    while True:
        command = input("\Введитеп команду: ").strip()

        if not command:
            continue

        parts = command.split()
        cmd = parts[0].lower()

        if cmd == "q":
            print("Выход из программы.")
            break

        elif cmd == "check":
            input_string = ""
            if len(parts) > 1:
                input_string = " ".join(parts[1:]).strip()

```

```
55              message, transitions = automaton.process_input(input_string)
56
57              print(f"Результат: {message}")
58              print("Путь переходов:", " -> ".join(transitions))
59
60          elif cmd == "gen":
61              stop_prob = 0.3
62              if len(parts) > 1:
63                  try:
64                      stop_prob = float(parts[1])
65                      if not (0 < stop_prob <= 1):
66                          raise ValueError(
67                              "Вероятность должна быть больше 0 и меньше либо равна 1"
68                          )
69                  except ValueError as e:
70                      print(f"Ошибка: {e}")
71                      continue
72
73              random_string, path = automaton.generate_random_string(stop_prob)
74              print(f"Сгенерированная строка: {random_string}")
75              print("Путь переходов:", " -> ".join(path))
76
77          else:
78              print(f"Неизвестная команда: {cmd}")
```

3 Результаты работы программы

Результаты работы программы представлены на Рис. 3.

```
Конечный автомат для распознавания форматов вещественных чисел
============================================================
Варианты команд:
 - check <строка> - проверить, соответствует ли строка автомату
 - gen [<вероятность_остановки>] - сгенерировать случайную строку (по умолчанию 0.3)
 - q - выход из программы
============================================================

Введите команду: check 1.5
Результат: Строка соответствует ✓
Путь переходов: S0 -> S2 -> S5 -> S5

Введите команду: check
Результат: Строка не соответствует ✗
Путь переходов: S0

Введите команду: check 0001
Результат: Строка не соответствует ✗
Путь переходов: S0 -> S3 -> S3 -> S3 -> S4

Введите команду: check a2
Результат: Символ 'a' не из алфавита ✗
Путь переходов: S0

Введите команду: gen
Сгенерированная строка: +0
Путь переходов: S0 -> S1 -> S3
Путь переходов: S0 -> S1 -> S3

Введите команду: gen 0.1
Сгенерированная строка: .77E439021365796918
Путь переходов: S0 -> S6 -> S7 -> S7 -> S8 -> S10 -> S10 -> S10 -> S10 -> S10 -> S10
-> S10 -> S10 -> S10 -> S10 -> S10 -> S10 -> S10 -> S10 -> S10

Введите команду: q
Выход из программы.
```

Рис. 3. Результаты работы программы.

```
Введите команду: blabla
Неизвестная команда: blabla

Введите команду: gen -1
Ошибка: Вероятность должна быть больше 0 и меньше либо равна 1

Введите команду: gen blabla
Ошибка: could not convert string to float: 'blabla'
```

Рис. 4. Реакция программы на некорректный пользовательский ввод.

На Рис. 4 представлена реакция программы на некорректный пользовательский ввод.

Заключение

В ходе выполнения лабораторной работы было построено регулярное выражение для распознавания различных форматов вещественных чисел. В соответствии с теоремой Клини по заданному регулярному выражению, задающему регулярный язык, был построен недетерминированный конечный автомат-распознаватель. Затем полученный конечный автомат был детерминирован. На основе разработанного автомата была реализована программа, которая проверяет соответствие входной строки заданному формату и генерирует случайные корректные строки.

Из достоинств выполнения лабораторной работы можно выделить структурирование кода за счёт использования ООП. Вся логика работы с конечными автоматами вынесена в отдельный класс `FiniteAutomaton` с четко разделенными методами для проверки строк и генерации случайных строк. Создана удобная интерактивная консольная оболочка для взаимодействия с пользователем, позволяющая выполнять различные команды.

К недостаткам текущей реализации можно отнести следующие аспекты. Во-первых, переходы в автомате представлены в виде строк, содержащих допустимые символы, такой способ представления переходов не является самым оптимальным с точки зрения производительности. Во-вторых, в реализации генерации случайных строк вероятность остановки одинакова для всех финальных состояний, что может приводить к неравномерному распределению различных форматов чисел в генерируемых строках.

Функционал программы несложно масштабировать. Класс `FiniteAutomaton` может быть использован для работы с различными конечными автоматами-распознавателями. Для изменения распознаваемого языка достаточно задать новые параметры для автомата, не меняя базовую логику программы. Однако, текущая реализация работает только с символьными переходами, поэтому задать строчные переходы в виде, например, регулярных выражений не представляется возможным. Однако подобный функционал также несложно реализовать, взяв за основу существующий код.

На выполнение лабораторной работы ушло около 10 часов. Работа была выполнена в среде разработки Visual Studio Code. Программа написана на Python версии 3.10.

Список литературы

[1] Востров, А.В. Курс лекций по дисциплине «Математическая логика». URL https://tema.spbstu.ru/compiler/ (дата обращения 01.04.2025 г.)

[2] Лутц, М. Изучаем Python. 5-е изд. / М. Лутц. — СПб.: Питер, 2019. — 1216 с.

[3] Фридл, Дж. Регулярные выражения = Mastering Regular Expressions / Дж. Фридл. — СПб.: Питер, 2001. — 352 с. — (Библиотека программиста).