Федеральное государственное бюджетное образовательное учреждение
высшего образования «Сибирский государственный университет
телекоммуникаций и информатики»
(СибГУТИ)

Кафедра ПМиК

ОТЧЕТ
по курсовой работе по предмету
Теория языков программирования и методы трансляции

Тема «Построение конструкций, задающих язык.»

Выполнил:
студент. гр. ИП-216
Андрущенко Филипп Александрович

Проверил:
ст. преподаватель, Осипова У.В.

оценка_____

подпись_____

Новосибирск 2025 г.

Оглавление

Задание

Вариант - 1

Разработанное программное приложение должно иметь графический интерфейс. Выбор конкретного средства разработки оставляется за студентом.

Программа должна управляться посредством меню с пунктами "Автор", "Тема" (с полной информацией о разработчике и теме задания), "Расчёты", "Запись результатов в файл" и другими, определяемыми конкретным заданием. При вводе данных с клавиатуры использовать соответствующую форму, а также предусмотреть возможность вызова справки с примером формата данных. Во всех вариантах заданий все результаты расчётов должны отображаться на экране и выводиться в файл (по требованию пользователя).

Замечание: необходимо предусмотреть обработку ошибок. Программа не должна "зависать" или прекращать выполнение по неизвестной причине - обязательна выдача соответствующей диагностики.

При введении автором каких-либо ограничений на исходные данные (размер алфавита и т.п.) они должны быть описаны в пояснительной записке и в справке.

Написать программу, которая по предложенному описанию языка построит детерминированный конечный автомат, распознающий этот язык, и проверит вводимые с клавиатуры цепочки на их принадлежность языку. Предусмотреть возможность поэтапного отображения на экране процесса проверки цепочек. Функция переходов ДКА может изображаться в виде таблицы и графа (выбор вида отображения посредством меню). Язык задается следующими параметрами:

- Алфавит, обязательная конечная подцепочка всех цепочек языка и кратность вхождения выбранного символа алфавита в любую цепочку языка.

Описание алгоритма решения задачи

Общий алгоритм работы программы

Программа решает задачу построения детерминированного конечного автомата, который распознаёт язык, заданный алфавитом, обязательной конечной подцепочкой и кратностью вхождения выбранного символа. Алгоритм состоит из четырех основных этапов:

Этап 1: Построение суффикс-автомата

Алгоритм вычисляет префикс-функцию суффикса по алгоритму Кнута-Морриса-Пратта для эффективных переходов. Для каждого состояния и символа определяется следующее состояние: начиная с текущей позиции, проходим по таблице border до совпадения с символом или до позиции ноль, затем сдвигаемся на одну позицию вперёд. Это создаёт подмашину, распознающую все суффиксы заданной подцепочки.

Этап 2: Расширение до полного автомата с кратностью

К состояниям суффикс-автомата добавляется счётчик кратности от нуля до k-минус-один. Полные состояния имеют вид "позиция в суффиксе и остаток кратности". Переходы работают так: берём переход по суффикс-автомату, если символ — это счётный символ, то увеличиваем счётчик по модулю k, иначе оставляем счётчик без изменений. Начальное состояние — ноль позиция и ноль счётчик, принимающее состояние — только когда суффикс полностью прочитан и счётчик равен нулю.

Этап 3: Проверка цепочек

Для заданной цепочки выполняем последовательные переходы: начинаем с начального состояния, для каждого символа цепочки применяем соответствующую функцию перехода. Цепочка принимается, если после обработки всех символов оказались в принимающем состоянии. Все шаги записываются для пошаговой демонстрации. Проверяется принадлежность всех символов алфавиту.

Этап 4: Визуализация

Таблица переходов заполняется в таблицу: строки — это состояния, столбцы — символы алфавита, ячейки содержат следующее состояние. Граф рисуется на холсте: состояния располагаются по окружности, между ними проводятся стрелки с метками переходов, несколько символов с одинаковым переходом объединяются в одну метку.

Пример работы программы:

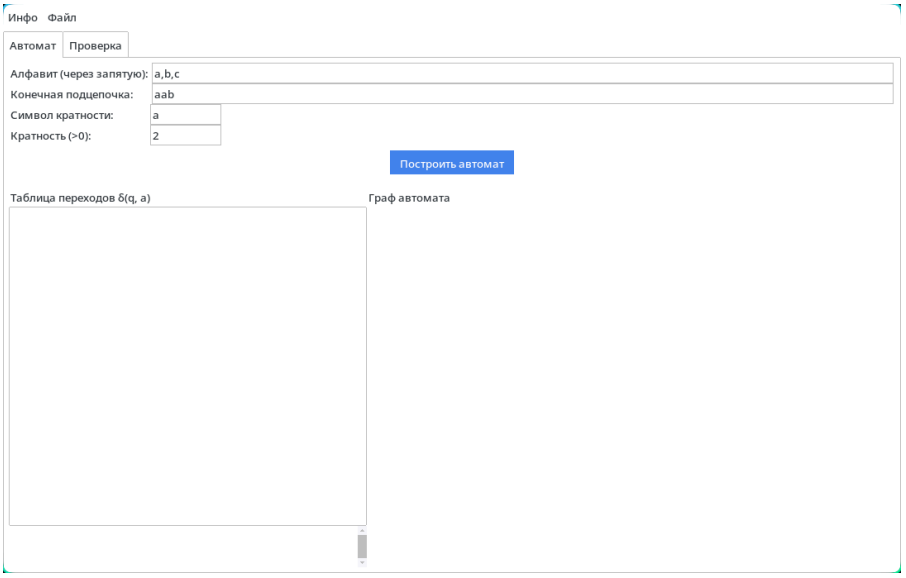

Рисунок 1. Пример входных данных

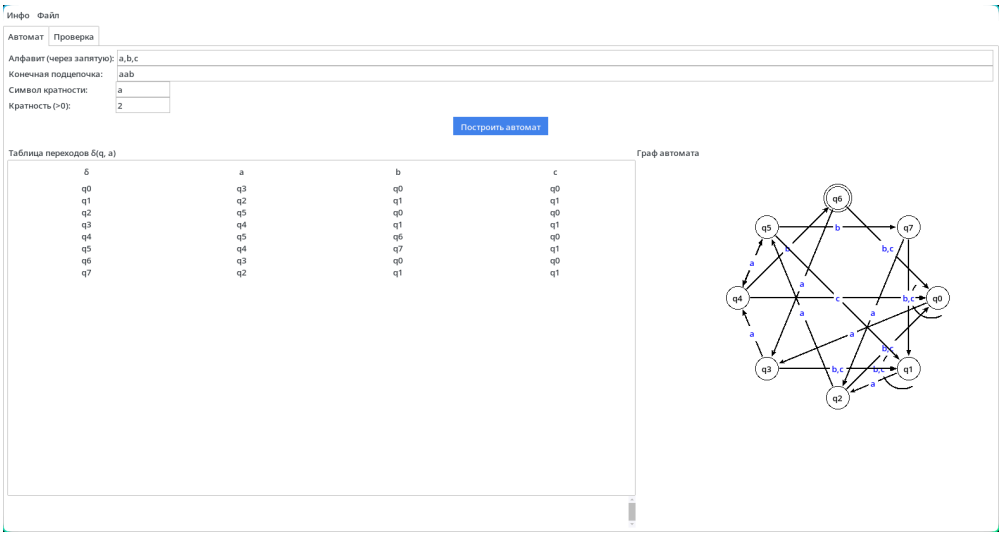

Рисунок 2. Визуализация построенного ДКА: таблица переходов и граф состояний

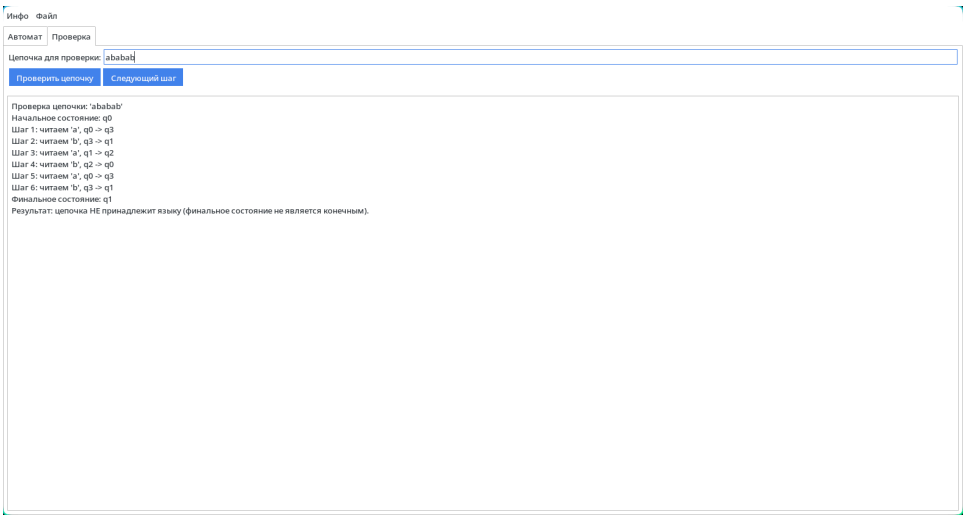

Рисунок 3. Результат проверки цепочки с пошаговым выводом переходов

Основные блоки программы

Блок 1: build_suffix_automaton

Функция строит суффикс-автомат для распознавания заданной подцепочки по алгоритму Кнута-Морриса-Пратта.

Основные компоненты:

- border — вычисление префикс-функции суффикса для эффективных переходов
- next_len(current_len, ch) — функция перехода: от текущей позиции проходим по border до совпадения с символом
- trans[(state, a)] — таблица переходов для всех состояний (0..len(suffix)) и символов алфавита

Логика: для каждого состояния и символа определяется следующее состояние через поиск по border-таблице с обработкой особого случая полного совпадения суффикса.

Блок 2: build_dfa

Функция создаёт полный детерминированный конечный автомат, комбинируя суффикс-автомат с счётчиком кратности.

Основные компоненты:

- internal_states — генерация состояний вида (позиция_в_суффиксе, остаток_кратности)
- state_to_name и name_to_state — словари для отображения состояний в читаемые имена q0, q1...
- dfa_trans[((i,r), a)] — таблица переходов полного автомата

Логика: вызывает суффикс-автомат, генерирует состояния (n+1)×k, строит переходы: next_i из суффикс-автомата + обновление счётчика (r+1 mod k если a=count_symbol).

Блок 3: run_dfa

Функция проверяет принадлежность цепочки языку с пошаговым логированием переходов.

Основные компоненты:

- steps — список всех переходов для пошаговой визуализации
- run_dfa - итеративный цикл по символам цепочки с проверкой алфавита

Логика: начинает с начального состояния (0,0), последовательно применяет переходы, возвращает принятие и список шагов. Принимает только в (len(suffix), 0).

Блок 4: DFAApp (GUI)

Класс реализует графический интерфейс с двумя вкладками, меню и визуализацией автомата.

Основные компоненты:

- Инициализация: __init__() создаёт notebook с вкладками "Автомат" и "Проверка", меню "Инфо"/"Файл"
- Ввод параметров: поля для алфавита, суффикса, символа кратности, k с валидацией

Визуализация:

- update_table() — Treeview таблица переходов (строки=состояния, столбцы=алфавит)

- draw_graph() — Canvas граф: состояния по окружности, группировка рёбер, петли для самопереходов
- Проверка: check_word() и next_step() — полная/пошаговая проверка с логированием
- Сохранение: save_results_to_file() — запись истории в UTF-8 файл

Блок 5: Визуализация графа (draw_graph)
Метод рисует граф автомата на Canvas с группировкой переходов.
Основные компоненты:

- positions — размещение состояний по окружности
- edge_labels[(q,nxt)] — группировка символов с одинаковым переходом

Результаты тестирования

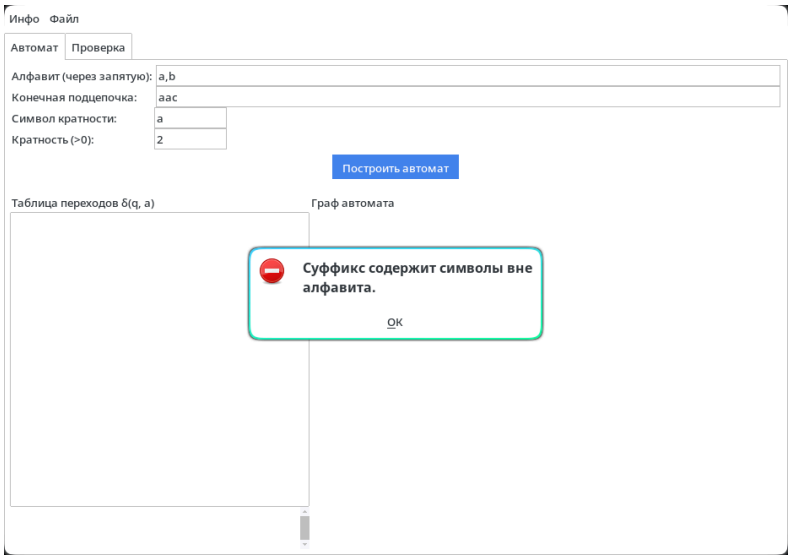

Рисунок 4. Попытка создания автомата с конечной подцепочкой не удовлетворяющей условию алфавита

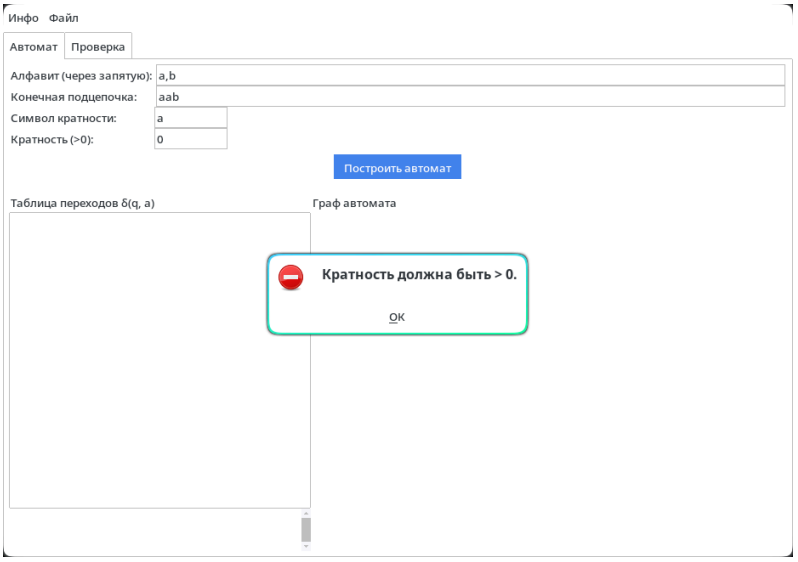

Рисунок 5. Тестирование создания автомата с кратностью символа равной 0

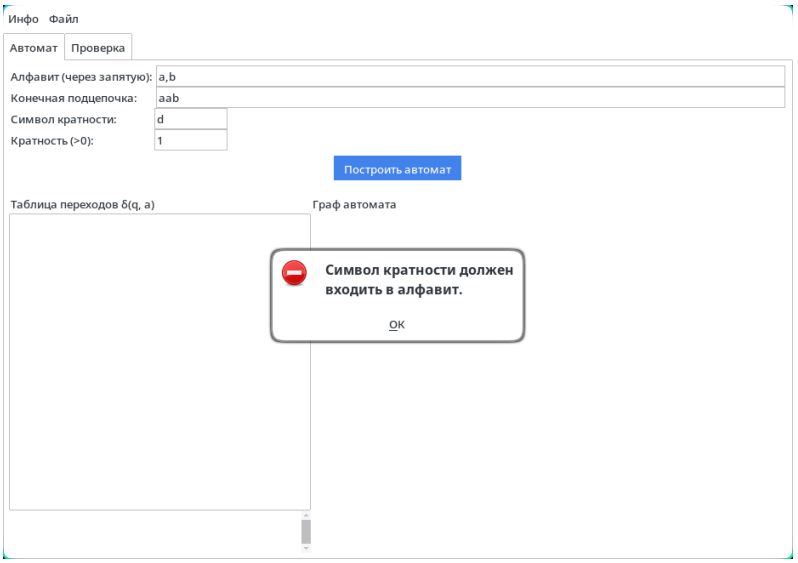

Рисунок 6. Тестирование создания автомата с условием кратности символа, не входящим в алфавит

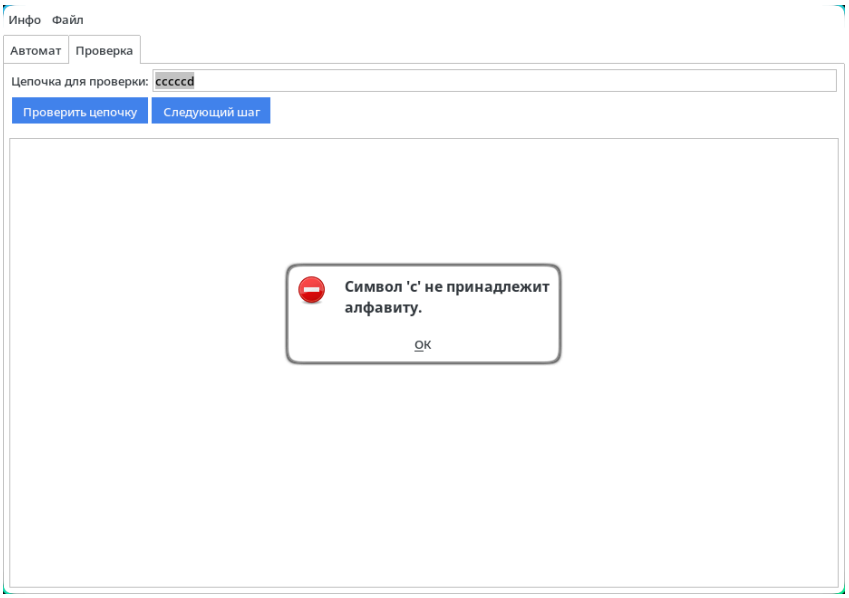

Рисунок 7. Попытка проверки цепочки с символами, не входящих в алфавит

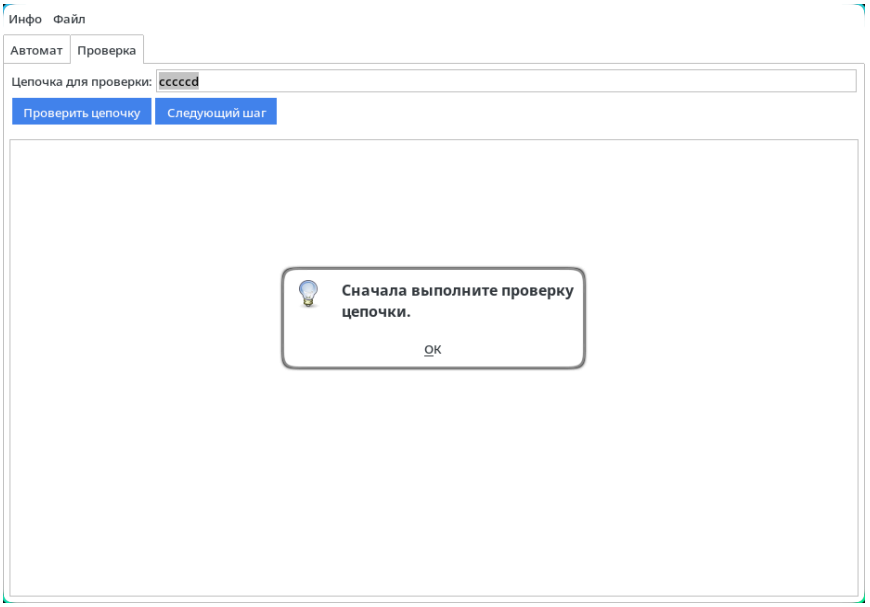

Рисунок 8. Попытка просмотра шагов проверки цепочки, без изначального ввода

Листинг

```
import tkinter as tk
from tkinter import messagebox, filedialog
from tkinter import ttk
import ttkbootstrap as tkb
import math


def build_suffix_automaton(alphabet, suffix):
    n = len(suffix)

    border = [0] * n
    k = 0
    for i in range(1, n):
        while k > 0 and suffix[k] != suffix[i]:
            k = border[k - 1]
        if suffix[k] == suffix[i]:
            k += 1
        border[i] = k

    def next_len(current_len, ch):
        while current_len > 0 and suffix[current_len] != ch:
            current_len = border[current_len - 1]
        if suffix[current_len] == ch:
            current_len += 1
        return current_len

    trans = {}
    for state in range(n + 1):
        for a in alphabet:
            if n == 0:
                trans[(state, a)] = 0
                continue
            if state == n:
                nxt = next_len(border[n - 1], a)
            else:
                if state > 0 and suffix[state] != a:
                    tmp = state
                    while tmp > 0 and suffix[tmp] != a:
                        tmp = border[tmp - 1]
                    if suffix[tmp] == a:
                        tmp += 1
                    nxt = tmp
                else:
                    if suffix[state] == a:
                        nxt = state + 1
                    else:
                        nxt = 0
            trans[(state, a)] = nxt
```

```
    return trans


def build_dfa(alphabet_str, suffix, count_symbol, k):
    alphabet = [s.strip() for s in alphabet_str.split(",") if s.strip()]

    if not alphabet:
        raise ValueError("Алфавит пустой.")

    if any(len(x) != 1 for x in alphabet):
        raise ValueError("Каждый символ алфавита должен быть одной
буквой/цифрой.")

    if count_symbol not in alphabet:
        raise ValueError("Символ кратности должен входить в алфавит.")

    for ch in suffix:
        if ch not in alphabet:
            raise ValueError("Суффикс содержит символы вне алфавита.")

    if k <= 0:
        raise ValueError("Кратность должна быть положительным целым
числом.")

    suff_trans = build_suffix_automaton(alphabet, suffix)

    internal_states = []
    for i in range(len(suffix) + 1):
        for r in range(k):
            internal_states.append((i, r))

    state_to_name = {}
    name_to_state = {}
    for idx, st in enumerate(internal_states):
        name = f"q{idx}"
        state_to_name[st] = name
        name_to_state[name] = st

    start_state = (0, 0)
    accept_states = [(len(suffix), 0)]

    dfa_trans = {}
    for i, r in internal_states:
        for a in alphabet:
            next_i = suff_trans[(i, a)]
            next_r = (r + 1) % k if a == count_symbol else r
            dfa_trans[((i, r), a)] = (next_i, next_r)

    return {
        "alphabet": alphabet,
```

```
        "states": internal_states,
        "start": start_state,
        "accepts": accept_states,
        "trans": dfa_trans,
        "suffix": suffix,
        "count_symbol": count_symbol,
        "k": k,
        "state_to_name": state_to_name,
        "name_to_state": name_to_state,
    }


def run_dfa(dfa, word):
    state = dfa["start"]
    steps = []
    for idx, ch in enumerate(word):
        if ch not in dfa["alphabet"]:
            raise ValueError(f"Символ '{ch}' не принадлежит алфавиту.")
        next_state = dfa["trans"][(state, ch)]
        steps.append({"pos": idx, "char": ch, "from": state, "to":
next_state})
        state = next_state
    accepted = state in dfa["accepts"]
    return accepted, steps, state


class DFAApp:
    def __init__(self, root):
        self.root = root
        self.root.title("ДКА по описанию языка")

        self.dfa = None
        self.last_results_log = ""
        self.step_index = 0
        self.current_steps = []

        self.create_menu()
        self.create_notebook()
        self.create_automaton_tab()
        self.create_check_tab()

    def create_menu(self):
        menu_bar = tk.Menu(self.root)

        info_menu = tk.Menu(menu_bar, tearoff=0)
        info_menu.add_command(label="Автор", command=self.show_author)
        info_menu.add_command(label="Тема", command=self.show_topic)
        info_menu.add_command(label="Справка", command=self.show_help)
        menu_bar.add_cascade(label="Инфо", menu=info_menu)
```

```
        file_menu = tk.Menu(menu_bar, tearoff=0)
        file_menu.add_command(
            label="Запись результатов в файл",
command=self.save_results_to_file
        )
        menu_bar.add_cascade(label="Файл", menu=file_menu)

        self.root.config(menu=menu_bar)

    def show_author(self):
        messagebox.showinfo(
            "Автор",
            "ФИО: Андрущенко Филипп Александрович\n"
            "Группа: ИП-216\n"
            "Тема: Построение конструкций, задающих язык.\n"
            "Контакт: darkoogle24@gmail.com",
        )

    def show_topic(self):
        messagebox.showinfo(
            "Тема",
            "Тема 1. Построение конструкций, задающих язык.\n"
            "Программа строит детерминированный конечный автомат по
алфавиту,\n"
            "обязательной конечной подцепочке и кратности вхождения
выбранного символа,\n"
            "затем проверяет цепочки на принадлежность языку.",
        )

    def show_help(self):
        help_text = (
            "Формат ввода:\n\n"
            "Алфавит: символы через запятую, например 0,1 или a,b,c\n"
            "Подцепочка: подцепочка, которой должны заканчиваться все
цепочки.\n"
            "Символ кратности: один символ из алфавита, кратность
которого будет проверяться.\n"
            "Кратность: целое число > 0; число вхождений выбранного
символа делится на него.\n\n"
            "Конечное состояние -- это состояние автомата,\n"
            "в котором оно останавливается после чтения всей строки и
принимает её."
        )
        messagebox.showinfo("Справка", help_text)

    def save_results_to_file(self):
        if not self.last_results_log:
            messagebox.showwarning("Нет данных", "Нет результатов для
сохранения.")
            return
```

```
        filename = filedialog.asksaveasfilename(
            defaultextension=".txt",
            filetypes=[("Text files", "*.txt"), ("All files", "*.*")],
        )
        if filename:
            try:
                with open(filename, "w", encoding="utf-8") as f:
                    f.write(self.last_results_log)
                messagebox.showinfo(
                    "Успех", f"Результаты сохранены в файл:\n{filename}"
                )
            except Exception as e:
                messagebox.showerror("Ошибка записи файла", str(e))

    def create_notebook(self):
        self.notebook = ttk.Notebook(self.root)
        self.notebook.pack(fill=tk.BOTH, expand=True)

        self.tab_automaton = ttk.Frame(self.notebook)
        self.tab_check = ttk.Frame(self.notebook)

        self.notebook.add(self.tab_automaton, text="Автомат")
        self.notebook.add(self.tab_check, text="Проверка")

    def create_automaton_tab(self):
        frame = self.tab_automaton

        top_frame = tk.Frame(frame)
        top_frame.pack(fill=tk.X, padx=5, pady=5)

        tk.Label(top_frame, text="Алфавит (через запятую):").grid(
            row=0, column=0, sticky="w"
        )
        self.entry_alphabet = tk.Entry(top_frame, width=35)
        self.entry_alphabet.insert(0, "a,b,c")
        self.entry_alphabet.grid(row=0, column=1, sticky="we", padx=2)

        tk.Label(top_frame, text="Конечная подцепочка:").grid(
            row=1, column=0, sticky="w"
        )
        self.entry_suffix = tk.Entry(top_frame, width=35)
        self.entry_suffix.insert(0, "aab")
        self.entry_suffix.grid(row=1, column=1, sticky="we", padx=2)

        tk.Label(top_frame, text="Символ кратности:").grid(row=2,
column=0, sticky="w")
        self.entry_symbol = tk.Entry(top_frame, width=10)
        self.entry_symbol.insert(0, "a")
        self.entry_symbol.grid(row=2, column=1, sticky="w")
```

```
        tk.Label(top_frame, text="Кратность (>0):").grid(row=3, column=0,
sticky="w")
        self.entry_k = tk.Entry(top_frame, width=10)
        self.entry_k.insert(0, "2")
        self.entry_k.grid(row=3, column=1, sticky="w")

        self.btn_build = tk.Button(
            top_frame, text="Построить автомат",
command=self.build_automaton
        )
        self.btn_build.grid(row=4, column=0, columnspan=2, pady=5)

        top_frame.grid_columnconfigure(1, weight=1)

        bottom_frame = tk.Frame(frame)
        bottom_frame.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

        table_frame = tk.Frame(bottom_frame)
        table_frame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

        tk.Label(table_frame, text="Таблица переходов δ(q,
a)").pack(anchor="w")

        self.table = ttk.Treeview(table_frame, show="headings")
        self.table.pack(fill=tk.BOTH, expand=True)

        scrollbar_y = ttk.Scrollbar(
            table_frame, orient="vertical", command=self.table.yview
        )
        self.table.configure(yscrollcommand=scrollbar_y.set)
        scrollbar_y.pack(side=tk.RIGHT, fill=tk.Y)

        graph_frame = tk.Frame(bottom_frame)
        graph_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)

        tk.Label(graph_frame, text="Граф автомата").pack(anchor="w")

        self.canvas = tk.Canvas(graph_frame, bg="white")
        self.canvas.pack(fill=tk.BOTH, expand=True)

    def create_check_tab(self):
        frame = self.tab_check

        top_frame = tk.Frame(frame)
        top_frame.pack(fill=tk.X, padx=5, pady=5)

        tk.Label(top_frame, text="Цепочка для проверки:").grid(
            row=0, column=0, sticky="w"
        )
        self.entry_word = tk.Entry(top_frame, width=40)
```

```
        self.entry_word.insert(0, "ababab")
        self.entry_word.grid(row=0, column=1, sticky="we", padx=2)

        self.btn_check = tk.Button(
            top_frame, text="Проверить цепочку", command=self.check_word
        )
        self.btn_check.grid(row=1, column=0, pady=5)

        self.btn_step = tk.Button(
            top_frame, text="Следующий шаг", command=self.next_step
        )
        self.btn_step.grid(row=1, column=1, sticky="w", pady=5)

        top_frame.grid_columnconfigure(1, weight=1)

        self.text_output = tk.Text(frame, height=20, width=80)
        self.text_output.pack(fill=tk.BOTH, expand=True, padx=5, pady=5)

    def build_automaton(self):
        try:
            alphabet = self.entry_alphabet.get().strip()
            suffix = self.entry_suffix.get().strip()
            symbol = self.entry_symbol.get().strip()
            k_str = self.entry_k.get().strip()

            if not suffix:
                raise ValueError("Подцепочка не должна быть пустой.")
            if not symbol:
                raise ValueError("Символ кратности не должен быть
пустым.")
            if len(symbol) != 1:
                raise ValueError("Символ кратности должен быть одним
знаком.")
            if not k_str.isdigit():
                raise ValueError("Кратность должна быть положительным
целым числом.")
            k = int(k_str)
            if k <= 0:
                raise ValueError("Кратность должна быть > 0.")

            self.dfa = build_dfa(alphabet, suffix, symbol, k)
            self.update_table()
            self.draw_graph()
            messagebox.showinfo("Готово", "ДКА успешно построен.")
        except Exception as e:
            messagebox.showerror("Ошибка построения автомата", str(e))

    def update_table(self):
        for col in self.table["columns"]:
            self.table.heading(col, text="")
```

```
        self.table.delete(*self.table.get_children())

        if not self.dfa:
            return

        alphabet = self.dfa["alphabet"]
        states = self.dfa["states"]

        columns = ["state"] + alphabet
        self.table["columns"] = columns

        self.table.column("state", width=80, anchor="center")
        self.table.heading("state", text="δ")

        for a in alphabet:
            self.table.column(a, width=80, anchor="center")
            self.table.heading(a, text=a)

        for q in states:
            row = [self.format_state(q)]
            for a in alphabet:
                nxt = self.dfa["trans"][(q, a)]
                row.append(self.format_state(nxt))
            self.table.insert("", tk.END, values=row)

        out_lines = []
        out_lines.append("Построенный ДКА:")
        out_lines.append(f"Алфавит: {self.dfa['alphabet']}")
        out_lines.append(f"Суффикс: '{self.dfa['suffix']}'")
        out_lines.append(
            f"Символ для кратности: '{self.dfa['count_symbol']}',
кратность: {self.dfa['k']}"
        )
        out_lines.append(f"Начальное состояние:
{self.format_state(self.dfa['start'])}")
        out_lines.append(
            "Конечное состояние: "
            + ", ".join(self.format_state(s) for s in
self.dfa["accepts"])
        )

        self.last_results_log = "\n".join(out_lines)

    def format_state(self, state):
        if not self.dfa:
            return str(state)
        return self.dfa["state_to_name"].get(state, str(state))

    def draw_graph(self):
        self.canvas.delete("all")
```

```
        if not self.dfa:
            return

        states = self.dfa["states"]
        accepts = set(self.dfa["accepts"])
        start = self.dfa["start"]

        n = len(states)
        if n == 0:
            return

        width = self.canvas.winfo_width() or 600
        height = self.canvas.winfo_height() or 400
        cx, cy = width // 2, height // 2
        radius = min(width, height) // 2 - 60

        positions = {}
        for idx, q in enumerate(states):
            angle = 2 * math.pi * idx / n
            x = cx + radius * math.cos(angle)
            y = cy + radius * math.sin(angle)
            positions[q] = (x, y)

        r_node = 18

        edge_labels = {}
        for (q, a), nxt in self.dfa["trans"].items():
            edge_labels.setdefault((q, nxt), []).append(a)

        for (q, nxt), labels in edge_labels.items():
            x1, y1 = positions[q]
            x2, y2 = positions[nxt]

            if q == nxt:
                loop_r = r_node + 14
                self.canvas.create_arc(
                    x1 - loop_r - 10,
                    y1 - loop_r,
                    x1 + loop_r - 10,
                    y1 + loop_r,
                    start=140,
                    extent=160,
                    style="arc",
                    width=2,
                )
                self.canvas.create_line(
                    x1 - loop_r, y1, x1 - r_node, y1, arrow=tk.LAST,
width=2
                )
```

```
                label = ",".join(labels)
                tx, ty = x1 - loop_r - 15, y1
                pad = 2
                tid = self.canvas.create_text(tx, ty, text=label,
fill="blue")
                bb = self.canvas.bbox(tid)
                self.canvas.create_rectangle(
                    bb[0] - pad,
                    bb[1] - pad,
                    bb[2] + pad,
                    bb[3] + pad,
                    fill="white",
                    outline="white",
                )
                self.canvas.tag_raise(tid)
                continue

            dx = x2 - x1
            dy = y2 - y1
            dist = math.hypot(dx, dy) or 1

            offset_start = r_node + 2
            offset_end = r_node + 2
            sx = x1 + dx * offset_start / dist
            sy = y1 + dy * offset_start / dist
            ex = x2 - dx * offset_end / dist
            ey = y2 - dy * offset_end / dist

            self.canvas.create_line(sx, sy, ex, ey, arrow=tk.LAST,
width=2)

            label = ",".join(labels)
            mx = (sx + ex) / 2
            my = (sy + ey) / 2
            pad = 2
            tid = self.canvas.create_text(mx, my, text=label,
fill="blue")
            bb = self.canvas.bbox(tid)
            self.canvas.create_rectangle(
                bb[0] - pad,
                bb[1] - pad,
                bb[2] + pad,
                bb[3] + pad,
                fill="white",
                outline="white",
            )
            self.canvas.tag_raise(tid)

        for q in states:
            x, y = positions[q]
```

```
            r = r_node
            self.canvas.create_oval(x - r, y - r, x + r, y + r,
fill="white")
            if q in accepts:
                self.canvas.create_oval(x - r - 4, y - r - 4, x + r + 4,
y + r + 4)
            if q == start:
                self.canvas.create_line(x - 35, y, x - r, y,
arrow=tk.LAST, width=2)
            self.canvas.create_text(x, y, text=self.format_state(q))

    def check_word(self):
        if not self.dfa:
            messagebox.showwarning(
                "Нет автомата", "Сначала постройте автомат на вкладке
'Автомат'."
            )
            return
        word = self.entry_word.get()
        try:
            accepted, steps, final_state = run_dfa(self.dfa, word)
            self.current_steps = steps
            self.step_index = 0

            log_lines = []
            log_lines.append(f"Проверка цепочки: '{word}'")
            log_lines.append(
                f"Начальное состояние:
{self.format_state(self.dfa['start'])}"
            )
            for s in steps:
                log_lines.append(
                    f"Шаг {s['pos'] + 1}: читаем '{s['char']}', "
                    f"{self.format_state(s['from'])} ->
{self.format_state(s['to'])}"
                )
            log_lines.append(f"Финальное состояние:
{self.format_state(final_state)}")
            if final_state in self.dfa["accepts"]:
                log_lines.append("Результат: цепочка ПРИНИМАЕТСЯ
(конечное состояние).")
            else:
                log_lines.append(
                    "Результат: цепочка НЕ принадлежит языку (финальное
состояние не является конечным)."
                )

            text = "\n".join(log_lines)
            self.text_output.delete("1.0", tk.END)
            self.text_output.insert(tk.END, text)
```

```
            self.last_results_log += "\n\n" + text
        except Exception as e:
            messagebox.showerror("Ошибка проверки цепочки", str(e))

    def next_step(self):
        if not self.current_steps:
            messagebox.showinfo("Нет шагов", "Сначала выполните проверку
цепочки.")
            return
        if self.step_index >= len(self.current_steps):
            messagebox.showinfo("Конец", "Шаги проверки завершены.")
            return
        s = self.current_steps[self.step_index]
        msg = (
            f"Шаг {s['pos'] + 1}:\n"
            f"Читается символ '{s['char']}'.\n"
            f"Переход: {self.format_state(s['from'])} ->
{self.format_state(s['to'])}."
        )
        self.step_index += 1
        messagebox.showinfo("Пошаговый просмотр", msg)


if __name__ == "__main__":
    root = tk.Tk()
    app = DFAApp(root)
    root.mainloop()
```