Лабораторна робота 3

Оксана Дзюба

1. TerminationState

Я реалізувала метод `check_self`, оскільки термінальний стан не повинен приймати жодного символу.

```
def check_self(self, char: str) -> bool:
    return False
```

Це потрібно для того, щоб скінченний автомат точно знав, що завершальний стан не може приймати жодні вхідні символи, і не продовжував аналіз далі.

2. DotState

Стан для крапки повинен приймати абсолютно будь-який символ. Це я реалізувала в `check_self`, повертаючи `True` незалежно від вхідного символу.

```
def check_self(self, char: str):
    return True
```

Також важливо, що кожен `DotState` створюється в конструкторі скінченного автомату, якщо в рядку regex є символ `.`, тобто це універсальний стан.

3. AsciiState

Потрібен для обробки звичайних ASCII-символів. Я реалізувала зберігання символу в конструкторі та порівняння в `check_self`:

```
def __init__(self, symbol: str) -> None:
    super().__init__()
    self.curr_sym = symbol

def check_self(self, curr_char: str) -> bool:
    return curr_char == self.curr_sym
```

Це дозволяє скінченному автомату точно визначати відповідність кожного символу шаблону, інакше він би помилково приймав будь-які символи.

4. StarState

Мета стану — дозволити багаторазове повторення одного і того ж стану. Я реалізувала зберігання попереднього стану, якому належить оператор `*`, і додала його у `next_states`.

```
def __init__(self, checking_state: State):
    super().__init__()
    self.checking_state = checking_state
    checking_state.next_states.append(self)
    self.next_states = [checking_state]
```

Метод `check_self` просто делегує перевірку своєму внутрішньому стану:

```
def check_self(self, char):
    return self.checking_state.check_self(char)
```

Завдяки цьому реалізується логіка повторення: якщо `checking_state` приймає символ, то скінченний автомат може залишитись у `StarState`, або знову перейти на початок цього ж стану.

5. PlusState

Подібний до `StarState`, але вимагає хоча б одного входу. Я реалізувала майже так само, як і для зірочки, оскільки повторення виглядає подібно, але скінченний автомат створюється по-іншому — не додається прямий перехід на цей стан із поточного.

```
def __init__(self, checking_state: State):
    super().__init__()
    self.checking_state = checking_state
    checking_state.next_states.append(self)
    self.next_states = [checking_state]

def check_self(self, char):
    return self.checking_state.check_self(char)
```

Потрібно, щоб скінченний автомат знав: плюс означає не просто можливе повторення, а обов'язкову мінімальну присутність хоча б одного символу.

6. Побудова автомату (RegexFSM)

Після зчитування символу я створювала відповідний стан. Якщо після символу є `*` або `+`, я видаляла попередній стан із черги та обгортала його у `StarState` або `PlusState`. Я робила перезапис переходів, щоб скінченний автомат знав, що має працювати вже з обгорнутим станом.

```
if i < len(regex_expr) and regex_expr[i] in "*+":
    operator = regex_expr[i]
    prev = self.states.pop()
    wrapped = StarState(prev) if operator == "*" else PlusState(prev)

    for state in self.states[::-1]:
        if prev in state.next_states:
            state.next_states = [wrapped if s is prev else s
                                 for s in state.next_states]
            break
    self.states.append(wrapped)
    i += 1
```

Цей блок дозволяє зберегти логіку: кожен стан має знати, чи він підпорядковується оператору * або +, а також правильно повертатись до себе.

7. check_string

Основний метод перевірки рядка. Рекурсивно проходжу всі стани скінченного автомату, намагаючись дійти до TerminationState рівно в кінці рядка. Додала внутрішню функцію `explore`:

```
def explore(state: State, pos: int) -> bool:
    if isinstance(state, TerminationState):
        return pos == len(s)

    accepted = False
    if pos < len(s) and state.check_self(s[pos]):
        for nxt in state.next_states:
            if explore(nxt, pos + 1):
                return True

    if isinstance(state, (StarState, PlusState)):
        for nxt in state.next_states:
            if explore(nxt, pos):
                return True

    return accepted
```

Це дозволяє скінченному автомату пробувати різні варіанти шляху. У випадку з `StarState` чи `PlusState` я також перевіряю можливість залишитися на місці (тобто, повторити).

Раніше, якщо стан мав перехід на AsciiState, а потім його замінювали на StarState, то попередній стан продовжував вказувати на старий об'єкт. Я реалізувала перезапис усіх переходів, які посилались на старий вузол:

```
for state in self.states[::-1]:
    if prev in state.next_states:
        state.next_states = [wrapped if s is prev else s
                             for s in state.next_states]
        break
```

Цей цикл проходить у зворотному порядку, оскільки найближчий перехід найімовірніше останній. Таким чином уникнула дублювання і зайвих переходів.

8. Завершальний перехід

Наприкінці побудови автомату обов'язково додається перехід від останнього створеного стану до TerminationState:

```
self.states[-1].next_states.append(self.exit)
```

Це потрібно для того, щоб рекурсія знала, коли вона закінчила опрацювання рядка.

Також я додала обробку помилок — не можна починати регулярний вираз з * або +:

```
if char in "*+":
    raise ValueError(f"Regex cannot start with '{char}'")
```

Це допомагає виявити помилки на етапі побудови скінченного автомату, а не під час виконання.

Висновок

Всі стани автомату були представлені окремими класами, які визначали поведінку при перевірці символів у рядку. Клас `StartState` є початковим станом, який не перевіряє жоден символ, але задає точку входу для автомату. Клас `TerminationState` є термінальним, що позначає кінець обробки рядка і не приймає жодних символів. Клас `DotState` відповідає символу `.`, що дозволяє автомату приймати будь-який символ. Це реалізовано шляхом того, що метод `check_self()` завжди повертає `True`, незалежно від символа. Клас `AsciiState` реалізує перевірку конкретного символу, що відповідає літерам або цифрам. Він приймає лише той символ, який був заданий при створенні стану. Клас `StarState` реалізує квантифікатор `*` (нуль або більше повторень), що дозволяє автомату залишатися на тому самому стані або рухатися далі, залежно від наявності символів для повторення. Клас `PlusState` аналогічний до `StarState`, але він вимагає хоча б одного символу для виконання переходу. Це забезпечує перевірку на мінімальну кількість повторень. Конструктор класу `RegexFSM` послідовно будує автомат, додаючи відповідні стани для кожного символу регулярного виразу. Після цього автомат перевіряє рядок, чи відповідає він шаблону, використовуючи рекурсивну функцію `check_string()` для обробки переходів між станами. Таким чином, основний принцип роботи цього коду полягає в побудові скінченного автомату з окремих станів, які відповідають різним частинам регулярного виразу. Кожен стан має свій набір правил переходу, що дозволяє автомату визначити, чи підходить йому певний рядок.