Лабораторна робота: Реалізація регулярних виразів за допомогою скінченних автоматів

Мета роботи

Закріпити знання з теми «Скінченні автомати», набути навички побудови та реалізації детермінованих та недетермінованих автоматів. Ознайомитися з патерном проєктування `State` в об'єктно-орієнтованому програмуванні. Імплементувати простий компілятор регулярних виразів у вигляді скінченного автомата.

Задача

Потрібно реалізувати клас `RegexFSM`, який приймає регулярний вираз у спрощеній формі (підтримуються символи: букви, цифри, `.`, `*`, `+`) та компілює його у скінченний автомат. Після цього автомат можна використовувати для перевірки, чи підходить рядок під даний вираз.

У даній роботі застосовано поняття **скінченного автомата (Finite State Machine)**:

- **Q** — множина станів (класи-нащадки `State`).
- — алфавіт (латинські букви, цифри, спецсимволи `. * +`).
- **:** $\mathbf{Q} \times \ \rightarrow \mathbf{Q}$ — функція переходу, реалізована методом `check_next`.
- **q** — початковий стан (`StartState`).
- **F** — множина кінцевих станів (`TerminationState`).

Алгоритм побудови автомата полягає у проходженні рядка регулярного виразу та створенні об'єктів-станів, які зв'язуються між собою через список `next_states`. Перевірка рядка здійснюється рекурсивним обходом автомата.

Опис структури програми

Класи станів

Усі класи наслідуються від абстрактного класу `State`, який визначає методи:

- `check_self(char)` — перевіряє, чи підходить символ для поточного стану.
- `check_next(char)` — переходить у наступний стан, який відповідає символу.

Конкретні реалізації:

- `StartState` — початковий стан.
- `TerminationState` — кінцевий стан.
- `AsciiState` — стан для конкретного символу (букви/цифри).
- `DotState` — універсальний стан, приймає будь-який символ.
- `StarState` — реалізація квантора “нуль або більше разів”.
- `PlusState` — реалізація квантора “один або більше разів”.

Клас RegexFSM

Основні частини:

- Метод `__init__` будує автомат із регулярного виразу.
- Метод `check_string(input_str)` перевіряє відповідність рядка регулярному виразу.
- Внутрішня функція `can_match(state, s, pos)` — реалізує рекурсивну перевірку переходів.

Фрагмент коду

```
class RegexFSM:
    ...
    def check_string(self, input_str: str):
        ...
        def can_match(state: State, s: str, pos: int) -> bool:
            if pos == len(s):
                return any(isinstance(n, TerminationState) or can_match(n, s, p
                           for n in state.next_states)

            if state.check_self(s[pos]):
                if can_match(state, s, pos + 1):
                    return True
```

```
            for next_state in state.next_states:
                if next_state.check_self(s[pos]) and can_match(next_state, s, p
                    return True
                if isinstance(next_state, StarState) and can_match(next_state,
                    return True

            return False

        return can_match(self.start_state, input_str, 0)
```

Приклад використання

```
regex_pattern = "a*4.+hi"
regex_compiled = RegexFSM(regex_pattern)

print(regex_compiled.check_string("aaaaaa4uhi"))  # True
print(regex_compiled.check_string("4uhi"))        # True
print(regex_compiled.check_string("meow"))        # False
```

Висновки

У ході виконання лабораторної роботи було реалізовано прототип регулярного виразу за допомогою скінченних автоматів. Використано патерн `State`, що спрощує структуру переходів та дозволяє гнучко розширювати систему.

Програма ілюструє, як із базових принципів теорії автоматів та дискретної математики можна побудувати простий парсер регулярних виразів, який працює з текстовими рядками.

Посилання

- https://pages.cs.wisc.edu/~hasti/cs536/readings/scanning.html
- https://wiki.eecs.yorku.ca/course_archive/2014-15/W/6339/_media/fsa.pdf
- Код доступний у репозиторії: https://github.com/LidaSemsichko/discrete_regex.git