Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.Б07-мм

# Разработка тестов для автоматизации проверки задач по курсу «Теория формальных языков»

***Кубышкин Ефим Алексеевич***

Отчёт по учебной практике
в форме «Производственное задание»

Научный руководитель:
доцент кафедры системного программирования, к. ф.-м. н., Григорьев С. В.

Санкт-Петербург
2024

# Оглавление

# Введение

Формальные языки нашли своё применение во многих областях: от машинного обучения [7] до биоинформатики [1]. Поэтому актуальность этой активно развивающейся области растёт. В связи с этим создаются различные учебные материалы по формальным языкам.

В то же время, продемонстрировать связь между практикой и теорией — важнейшая задача в обучении программистов. Один из курсов, который приоритезирует решение этой задачи — «Теория формальных языков», читающийся на мат-мехе (и не только). В нём ставится акцент на современных алгоритмах (например, [2, 4]) и их реализации, в значительной мере отдавая базовую теорию на самостоятельное изучение. Что помогает не только лучше усвоить материал, но и почувствовать влияние смежных математических дисциплин, таких как линейная алгебра и теория графов. Именно поэтому в каждом из 12 заданий курса нужно так или иначе писать код, сложность которого варьируется от простого использования библиотечных функций до реализации интерпретатора языка запросов.

Основная проблема такого подхода — сложность проверки написанного кода, ведь без инструментов автоматического тестирования преподавателю приходилось бы лично просматривать каждую из реализаций. Это не только трудоёмко, но ещё и крайне подвержено риску ошибок в силу человеческого фактора.

Одним из решений этой трудности являются автоматизированные тесты. Она позволяют уменьшить нагрузку на преподавателя и предоставить обучающимся набор необходимых условий, которым должны удовлетворять их решения. Таким образом, этот проект нацелен на реализацию тестов для курса «Теория формальных языков».

# 1. Постановка задачи

Целью работы является написание и внедрение тестов к домашним заданиям по курсу «Теория формальных языков». Для её реализации были поставлены следующие задачи:

- продумать инварианты, которые обязаны сохранятся для каждой задачи;
- реализовать и внедрить тесты в репозиторий курса[1];
- провести проверку домашних заданий с помощью тестов во время чтения семестрового курса.

[1]Репозиторий курса на GitHub: https://github.com/FormalLanguageConstrainedPathQuerying/formal-lang-course (дата обращения: 31 мая 2024 г.)

# 2. Обзор

В этом разделе будут рассмотрены инструменты и подходы, используемые во время реализации тестов. А также будет приведён весь перечень заданий из курса с формулировками. Отметим, что выбор почти всех технологий был продиктован тем, что они уже использовались в курсе.

## 2.1. Математические библиотеки

Для работы с грамматиками в различном виде: контекстно-свободная грамматика, регулярное выражение, конечные автоматы, рекурсивные автоматы — была выбрана библиотека pyformlang [6]. Она предоставляет интерфейс для взаимодействия с ними.

Для работы с графами была использована библиотека networkx [3].

Для синтаксического разбора использовался ANTLR [5]. Он позволяет по грамматике генерировать парсер, строящий абстрактные синтаксические деревья, которые нужны, чтобы легче анализировать программу, например, типизировать или интерпретировать её.

## 2.2. Тестирование, основанное на свойствах

Для функционального тестирования существует несколько подходов. Один из них — тестирование, основанное на свойствах. Он заключается в том, чтобы выделить некоторые инварианты, которым обязана удовлетворять корректная реализация. На наборе случайно выбранных данных проверяется, что выделенные свойства действительно выполняются. Таким образом можно относительно легко составить большое количество тестов, ведь не придётся в ручную просчитывать ответы для каждого входа.

## 2.3. Pytest

В качестве основного фреймворка для написания тестов был использован pytest [8]. У него есть довольно обширная документация, и

он является одним из наиболее распространённых инструментов для реализации не слишком больших функциональных тестов. Отдельно рассмотрим следующие инструменты, так как они активно использовались.

- Декоратор fixture позволяет написать функцию для динамического создания какого-то объекта по запросу и использовать его в теле теста. Это делается, передав имя написанной функции в качестве аргумента. Пример приведён на листинге 1. Здесь для случайно сгенерированных восьми графов будут запущены тесты, проверяющие на то, что они направленные.

- Декоратор parametrize позволяет передать любую коллекцию в качестве параметра, и будут запущены тесты со всеми объектами из неё. Объект можно использовать в тесте, передав название параметра в качестве аргумента, пример использования этого декоратора приведён на листинге 2. Здесь для каждой грамматики из коллекции `GRAMMARS` будет проверено, что она задаёт не пустой язык.

**Листинг 1: Пример использования fixture**

```
@pytest.fixture(scope="function", params=range(8))
def graph(request) -> MultiDiGraph:
    return generate_rnd_graph(1, 40, LABELS)

def test_graph(graph):
    assert graph.is_directed()
```

**Листинг 2: пример использования параметризации**

```
@pytest.mark.parametrize("grammar", GRAMMARS)
def test_grammar(grammar):
    assert not grammar.is_empty()
```

## 2.4. Формулировки задач

Для понимания того, какая именно функциональность тестируется, будут представлены формулировки всех заданий курса в сокращённом виде.

### Задача 1. Инициализация рабочего окружения

- Сделать fork репозитория курса.
- Занести всю необходимую информацию о себе в таблицу с текущим прогрессом и добавить в совладельцы форка одного из ассистентов.
- С помощью cfpq-data[2] реализовать следующие функции.
  - По имени графа вернуть количество вершин, рёбер и перечислить различные метки, встречающиеся на рёбрах.
  - По количеству вершин в циклах и именам меток строить граф из двух циклов и сохранять его в указанный файл в формате DOT (использовать pydot).
- Добавить запуск тестов с помощью pytest в `.github/workflows/`.

### Задача 2. Построение детерминированного конечного автомата по регулярному выражению и недетерминированного конечного автомата по графу

- Используя возможности pyformlang, реализовать функцию построения минимального ДКА по заданному регулярному выражению.
- Используя возможности pyformlang, реализовать функцию построения недетерминированного конечного автомата по графу.

[2]Репозиторий cfpq_data: https://github.com/FormalLanguageConstrainedPathQuerying/CFPQ_Data (дата обращения: 31 мая 2024 г.)

### Задача 3. Регулярные запросы для всех пар вершин

- Реализовать тип `FiniteAutomaton`, представляющий конечный автомат в виде разреженной матрицы смежности из sciPy (или сразу её булевой декомпозиции) и информации о стартовых и финальных вершинах.
- Реализовать функцию-интерпретатор для типа `FiniteAutomaton`, выясняющую, принимает ли автомат заданную строку и является ли язык, задающийся автоматом, пустым.
- Используя разреженные матрицы из sciPy, реализовать функцию пересечения двух конечных автоматов через тензорное произведение.
- На основе предыдущей функции реализовать функцию выполнения регулярных запросов к графам: по графу с заданными стартовыми и финальными вершинами и регулярному выражению вернуть те пары вершин из заданных стартовых и финальных, которые связанны путём, формирующем слово из языка, задаваемого регулярным выражением.

### Задача 4. Регулярные запросы для нескольких стартовых вершин

Используя разреженные матрицы, реализовать функцию достижимости с регулярными ограничениями с несколькими стартовыми вершинами.

- Для конструирования регулярного запроса и графа использовать Задачи 2.
- Для каждой из стартовых вывести множество достижимых из неё.

### Задача 5. Экспериментальное исследование алгоритмов для регулярных запросов

Задача посвящена анализу производительности алгоритма решения задачи достижимости между всеми парами вершин и с заданным множеством стартовых вершин с регулярными ограничениями.

Исследуются следующие задачи достижимости, решаемые в предыдущих работах.

- Достижимость между всеми парами вершин.
- Достижимость для каждой из заданного множества стартовых вершин.

Вопросы, на которые необходимо ответить в ходе исследования.

- Какое представление разреженных матриц и векторов лучше подходит для каждой из решаемых задач?
- Начиная с какого размера стартового множества выгоднее решать задачу для всех пар и выбирать нужные?

### Задача 6. Преобразование грамматики в ОНФХ, алгоритм Хеллингса

- Используя возможности pyformlang для работы с контекстно-свободными грамматиками, реализовать функцию преобразования контекстно-свободной грамматики в ослабленную нормальную форму Хомского (ОНФХ).
- Реализовать функцию, основанную на алгоритме Хеллингса, решающую задачу достижимости между всеми парами вершин для заданного графа и заданной КС грамматики (не обязательно в ОНФХ).

### Задача 7. Матричный алгоритм решения задачи достижимости с КС ограничениями

Реализовать функцию, основанную на матричном алгоритме, решающую задачу достижимости между всеми парами вершин для заданного графа и заданной КС грамматики.

### Задача 8. Тензорный алгоритм решения задачи достижимости с КС ограничениями

Реализовать функцию, основанную на тензорном алгоритме, решающую задачу достижимости между всеми парами вершин для заданного графа и заданной КС грамматики.

Для преобразования грамматики в RSM использовать результаты предыдущих работ. Явно описать функции преобразования CFG $\rightarrow$ RSM и EBNF $\rightarrow$ RSM.

### Задача 9. Алгоритм решения задачи достижимости с КС ограничениями, основанный на GLL

Реализовать функцию, основанную на алгоритме Generalized LL (работающего с RSM), решающую задачу достижимости между всеми парами вершин для заданного графа и заданной КС грамматики.

### Задача 10. Экспериментальное исследование алгоритмов решения задачи достижимости с КС ограничениями

Провести анализ производительности различных алгоритмов решения задачи достижимости между всеми парами вершин с контекстно-свободными ограничениями: алгоритма Хеллингса, матричного алгоритма, тензорного алгоритма, алгоритма на основе GLL. В ходе анализа необходимо ответить на следующие вопросы.

- Какой из трёх указанных алгоритмов обладает лучшей производительностью?

- Имеет ли смысл для решения задачи достижимости с регулярными ограничениями использовать алгоритмы для КС ограничений (ведь регулярные — частный случай КС) или всё же лучше использовать специализированные алгоритмы для регулярных ограничений?

- Как влияет грамматика на производительность тензорного алгоритма и алгоритма на основе GLL? Если зафиксировать язык, то как свойства грамматики (размер, (не)однозначность) влияют на производительность?

### Задача 11. Язык запросов к графам

Для данного синтаксиса необходимо.

- С использованием ANTLR реализовать синтаксический анализатор предложенного языка. А именно, реализовать функцию, которая принимает строку и возвращает дерево разбора.

- Реализовать функцию, которая по дереву разбора возвращает количество узлов в нём.

- Реализовать функцию, которая по дереву разбора строит ранее разобранную строку.

### Задача 12. Интерпретатор языка запросов к графам

- Реализовать механизм вывода типов, гарантирующий корректность построения запросов.

- Из множества реализованных в предыдущих работах алгоритмов выполнения запросов к графам выбрать те, которые будут использоваться в интерпретаторе.

- Используя парсер из предыдущей работы, разработанную систему вывода типов, выбранные алгоритмы, реализовать интерпретатор языка, описанного в предыдущей задаче.

Можно заметить, что некоторые задания тесно связаны друг с другом: 3-4, как и 6-9 задания содержательно решают одну и ту же задачу.

# 3. Реализация

В этом разделе будет показано, какие инварианты были придуманы для каждой задачи, а также некоторые подробности реализации тестов.

Общая структура разделов с заданиями будет следующей.

1. Сигнатуры функций, которые необходимо было реализовать студентам.

2. Придуманные инварианты для них. Заметим, что они могут быть как для каждой функции по отдельности, так и для некоторой их комбинации.

3. Подробности реализации вышеописанных свойств в тестах.

В заданиях 5, 10 нужно сделать эксперимент, так что для них тестов не предполагалось. Тесты для первого задания также не предусмотрены.

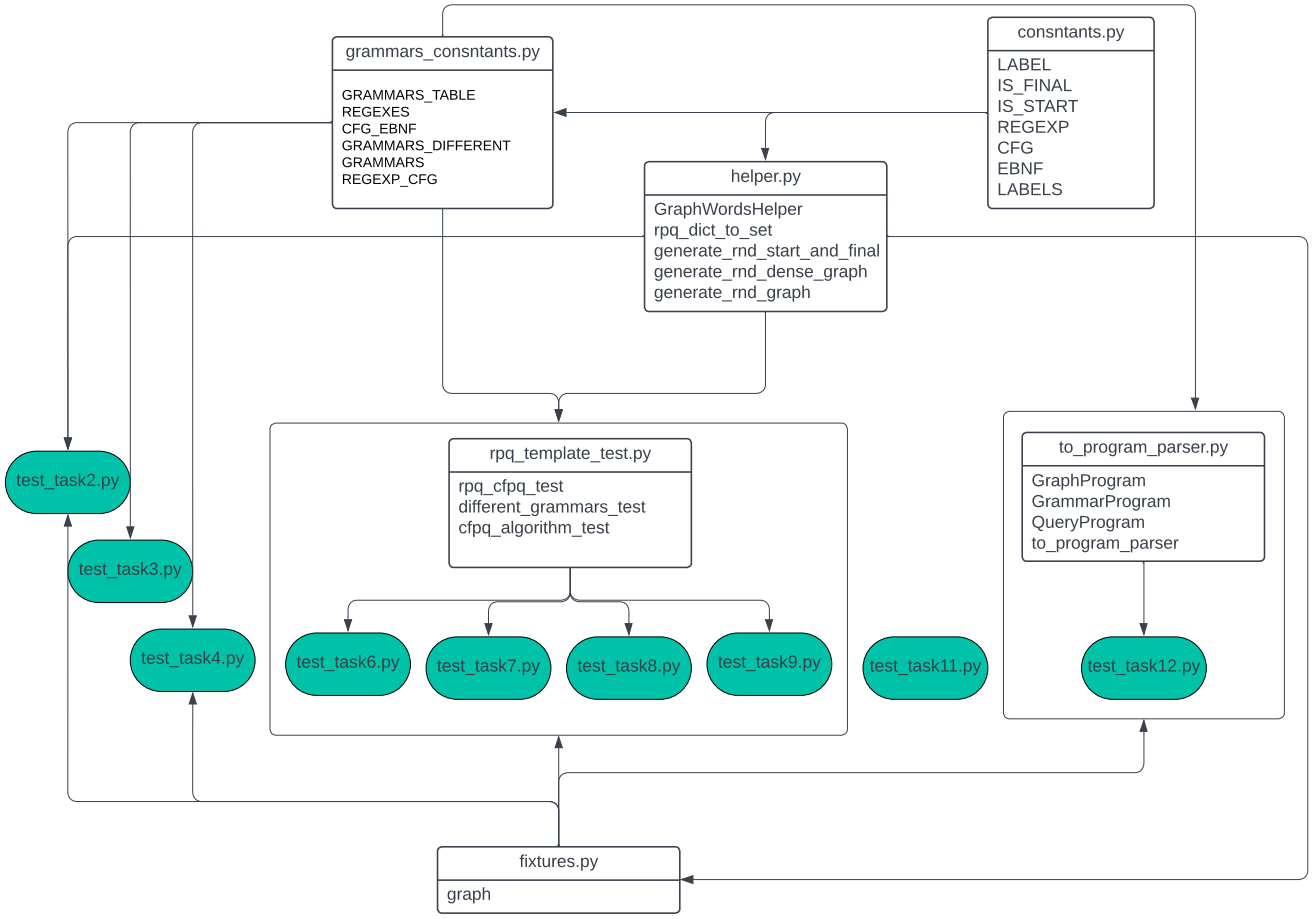
## 3.1. Архитектура системы тестирования

Все файлы с тестами лежат в одной папке и названы по шаблону `test_task<номер задания>.py`. В этом же папке лежат все вспомогательные файлы. Модули и взаимосвязи между ними представлены на uml-диаграмме 1.

## 3.2. Общие детали

Как будет понятно далее, почти во всех задачах необходимо генерировать случайный помеченный направленный граф (разреженный или плотный). Для его построения были написаны вспомогательные функции в файле `helper.py`, там же функция для генерации случайных стартовых и финальных вершин. В файл `fixtures.py` была вынесена fixture для генерации графа (которая вызывает функции из `helper.py`), это позволило упростить использование его в тестах до простого импортирования соответствующей функции в модуль.

Рис. 1: Архитектура системы тестирования




Также общим почти для всех тестов было использование грамматик в том или ином виде (регулярное выражение, стандартное представление контекстно-свободной грамматики, расширенная форма форма Бэкуса-Наура). Причём иногда необходимо знать, какие грамматики задают один и тот же язык. В качестве решения этой проблемы был создан файл `grammars_constants.py`, который содержит в себе таблицу вида 1.

Все используемые в тестах множества составлены на её основе и лежат в том же файле.

## 3.3. Задание 2

Требуемые от студентов функции с сигнатурами представлены в листинге 3.

Таблица 1: Пример части таблицы с грамматиками

| **Language** | **Regexp** | **CFG** | **EBNF** |
|---|---|---|---|
| $L_1$ | $a^*$ | $S \to \varepsilon$<br>$S \to aS$ | $S \to a^*$ |
| | | $S \to \varepsilon$<br>$S \to SS$<br>$S \to a$ | |
| | $a^* \mid a$ | $S \to \varepsilon$<br>$S \to SaS$ | |
| $L_2$ | – | $S \to S_1$<br>$S \to S_2$<br>$S_1 \to S_{ab}$<br>$S_1 \to S_1 c$<br>$S_{ab} \to \varepsilon$<br>$S_{ab} \to aS_{ab}b$<br>$S_2 \to S_{bc}$<br>$S_2 \to aS_2$<br>$S_{bc} \to \varepsilon$<br>$S_{bc} \to bS_{bc}c$ | $S \to S_{ab}c^* \mid a^*S_{bc} \mid \varepsilon$<br>$S_{ab} \to aS_{ab}b \mid \varepsilon$<br>$S_{bc} \to bS_{bc}c \mid \varepsilon$ |

...

**Листинг 3: Сигнатуры функций для задания 2**

```
def regex_to_dfa(regex: str) -> DeterministicFiniteAutomaton

def graph_to_nfa(
    graph: MultiDiGraph,
    start_states: set[int],
    final_states: set[int]
    ) -> NondeterministicFiniteAutomaton:
```

Инвариант первой функции заключается в том, что любое слово, задающиеся регулярным выражением обязано приниматься построенным конечным автоматом.

Реализация заключается в том, что для набора регулярных выражений запускается функция `regex_to_nfa`, генерируется случайное слово, и у полученного автомата вызывается функция `accepts` со сгенерированным словом, которая проверяет, принимает ли автомат данное слово или нет.

Для второй функции идея похожа — любое слово, которое можно получить из графа, должно приниматься функцией `accepts` полученного после применения функции `graph_to_nfa` автомата. Так как ни в pyformlang, ни в networkx нет функциональности, которая бы позволила генерировать слова, задающиеся путями в помеченном графе, был написан класс `GraphWordsHelper`, который решает эту задачу.

Алгоритм генерации слов для фиксированной вершины основан на обходе в ширину. Код представлен на листинге 4. Ключевые особенности алгоритма заключаются в следующем.

- В очереди лежат пары: вершина, в которую будет совершён переход и слово, составленное метками рёбер пути до этой вершины.
- Во фронте обхода будут все смежные вершины.
- Если вершина является финальной, то возвращаем слово, но продолжаем обход. Ключевое слово `yield` в python3 позволяет сделать

**Листинг 4: Алгоритм генерации слов**

```
def generate_words_by_node(self, node):
queue = [(node, [])]
while len(queue) != 0:
    (n, word) = queue.pop(0)
    for node_to, label in self._take_a_step(n):
        tmp = word.copy()
        tmp.append(label)
        if self._is_final_node(node_to):
            yield tmp.copy()
        if self._exists_any_final_path(node_to):
            queue.append((node_to, tmp.copy()))
```

это. Таким образом, создаётся генератор, который возвращает потенциально бесконечное множество слов.

Этот алгоритм запускается для всех стартовых вершин, для каждой из них берётся конечное количество слов.

Таким образом, для первой функции берётся случайное регулярное выражение, средствами pyformlang генерируется набор слов (все слова из языка, если он конечен и $2^9$ слов, если он бесконечен) и для случайного слова проверяется, что оно принимается конечным автоматом, полученным после применения к данному регулярному выражению функции `regex_to_dfa`.

Для второй функции берётся случайный направленный помеченный граф, по нему генерируются слова и для случайного слова проверяется, что оно принимается конечным автоматом, полученным после применения к данному графу функции `graph_to_nfa`.

## 3.4. Задания 3-4

Студентам нужно было реализовать класс `FiniteAutomaton` и функции, представленные на листинге 5.

Четвёртое задание решает ту же задачу. Сигнатура требуемой функции представлена на листинге 6.

**Листинг 5: Сигнатуры функций для задания 3**

```
def accepts(self, word: Iterable[Symbol]) -> bool

def is_empty(self) -> bool

def intersect_automata(
    automaton1: FiniteAutomaton,
    automaton2: FiniteAutomaton
    ) -> FiniteAutomaton

def paths_ends(
    graph: MultiDiGraph,
    start_nodes: set[int],
    final_nodes: set[int],
    regex:str
    ) -> list[tuple[NodeView, NodeView]]
```

**Листинг 6: Сигнатуры функции для задания 4**

```
def reachability_with_constraints(
    fa: FiniteAutomaton,
    constraints_fa: FiniteAutomaton
    ) -> dict[int, set[int]]:
```

Вспомогательные функции `intersect_automata`, `is_empty`, `accepts` можно проверить через соответствующие функции в pyformlang для контекстно-свободных грамматик. Соответственно, любое слово из языка, задающегося пересечением двух грамматик функцией из pyformlang, должно приниматься и пересечением двух автоматов. Если же пересечение пусто, оно должно быть таковым и для автоматов.

Инвариантом для основных функций будет являться то, что для любого графа и любого регулярного выражения, результат работы функции `paths_ends` должен быть равен результату работы функции `reachability_with_constraints`, если предварительно переделать

граф и регулярное выражение в конечные автоматы.

В реализации для случайного графа, случайного регулярного выражения проверяется на равенство результаты двух функций.

## 3.5. Задания 6-9

Задачи с шестой по девятую решает одну и ту же проблему — поиск путей в графе с контекстно свободными ограничениями. То есть надо найти все пары вершин, между которыми существует путь из заданного КС языка. Разница заключается только в алгоритме, который решает эту задачу и в представлении входных данных. Сигнатуры функций, которые необходимо было написать студентам представлены на листингах 7, 8, 9, 10.

**Листинг 7: Сигнатуры функции для задания 6**

```
def cfg_to_weak_normal_form(cfg: CFG) -> CFG

def cfpq_with_hellings(
    cfg: CFG,
    graph: DiGraph,
    start_nodes: set[int] = None,
    final_nodes: set[int] = None,
    ) -> set[tuple[int, int]]
```

**Листинг 8: Сигнатуры функции для задания 7**

```
def cfpq_with_matrix(
    cfg: CFG,
    graph: DiGraph,
    start_nodes: Set[int] = None,
    final_nodes: Set[int] = None,
    ) -> set[tuple[int, int]]
```

Для всех задач инвариант почти одинаковый: на одном графе для разных грамматик (возможно, в разных форматах), задающих один

**Листинг 9: Сигнатуры функции для задания 8**

```
def cfpq_with_tensor(
    rsm: RecursiveAutomaton,
    graph: DiGraph,
    final_nodes: set[int] = None,
    start_nodes: set[int] = None,
    ) -> set[tuple[int, int]]
def cfg_to_rsm(cfg: CFG) -> RecursiveAutomaton

def ebnf_to_rsm(ebnf: str) -> RecursiveAutomaton
```

**Листинг 10: Сигнатуры функции для задания 9**

```
def cfpq_with_gll(
    rsm: RecursiveAutomaton,
    graph: DiGraph,
    start_nodes: set[int] = None,
    final_nodes: set[int] = None,
    ) -> set[tuple[int, int]]
```

язык, ответ должен быть одинаков. Дополнительно, для одинаковых входных данных, с точностью до представления, все алгоритмы должны иметь одинаковый результат.

Для реализации в файле `rpq_template_test.py` были написаны шаблонные функции, которые абстрагировали проверку от конкретного алгоритма. Также в каждом следующим после 6 задания тесте, реализована проверка результатов всех ранее написанных алгоритмов. Таким образом, фиксировался случайный граф и для всех грамматик, задающих один язык, запускались тестовые функции.

Отдельно можно отметить, что реализация рекурсивных автоматов в pyformlang вызывала вопросы, так что был сделан пулл реквест с их изменениями[3]. Они затронули интерфейсы и названия в сторону большей

[3]Пулл реквест на GitHub: https://github.com/Aunsiels/pyformlang/pull/20 (дата обращения: 31 мая 2024 г.)

строгости, что позволило более интуитивно работать с рекурсивными автоматами.

## 3.6. Задание 11

Сигнатуры требуемых функций представлены на листинге 11 Где функция `prog_to_tree` строит дерево по тексту программы, `nodes_count` возвращает количество узлов в дереве, `tree_to_prog` по дереву возвращает программу.

**Листинг 11: Сигнатуры функций для задания 11**

```
def prog_to_tree(program: str) -> tuple[ParserRuleContext, bool]:

def nodes_count(tree: ParserRuleContext) -> int:

def tree_to_prog(tree: ParserRuleContext) -> str:
```

Инвариантом для этого задания таков, что исходная программа должна остаться неизменной после поочерёдного применения к ней функций `prog_to_tree` и `tree_to_prog`. В реализации для множества корректных программ проверялся вышеописанный инвариант. Дополнительно была написана функция, которая «ломала» программу: удаляла из неё некоторые служебные символы, без которых она должна была стать некорректной, и проверялось, что реализованная студентами функция правильно обрабатывает такие ошибки.

## 3.7. Задание 12

Сигнатуры требуемых функций представлены на листинге 12. Где первая функция возвращает `True`, если данная программа может быть корректно протипизиорвана и `False` иначе. А вторая возвращает словарь, содержащий в качестве ключа название переменной, содержащую запрос, а значение — его результат.

**Листинг 12: Сигнатуры функций для задания 12**

```
def typing_program(program: str) -> bool:

def exec_program(program: str) -> dict[str, set[tuple]]:
```

В процессе написания интерпретатора студенты должны были использовать какой-либо алгоритм из 3.5. Таким образом, для любого графа, любой грамматики и соответствующего запроса, результат должен быть равным результату выполнения отдельной функции из заданий 6-9.

Для реализации в файле `to_program_parser.py` были написаны вспомогательные классы `GraphProgram, GrammarProgram, QueryProgram`, с помощью которых можно перевести граф, грамматику и запрос соответственно в программу на языке запросов. Таким образом, для случайного графа, грамматики и стартовых и финальных вершин строится программа, результат выполнения на ней функции `exec_program` должен быть равен результату выполнения любого из алгоритмов (был выбран `cfpq_with_matrix`) с этими же данными.

# Заключение

В рамках учебной практики были решены следующие задачи.

- Выделены основные свойства, которым должны удовлетворять все решения, придуманы инварианты.
- На их основе реализованы тесты. Внедрение же происходило постепенно в читающийся курс в ИТМО в этом семестре и сразу же использовались для верификации решений домашних заданий.
- Проведена проверка домашних заданий студентов на основе тестов.
- Внесены изменения для рекурсивных автоматов в библиотеку с открытым исходным кодом pyformlang[4].

Код можно найти на GitHub[5], работа выполнялась под ником KubEF.

В процессе разработки возникали проблемы, некоторые из которых остались нерешёнными. Так что дальнейшие пути развития курса в этом направлении видятся следующими.

- Из-за случайной природы генерации графа, может оказаться, что какие-то краевые случаи остались непротестированными, так что есть необходимость в наборе вручную составленных тестах, гарантирующие содержательную проверку.
- Тесты внедрялись «по ходу дела», так что некоторые архитектурные решения оставляют вопросы. Продумать этот момент и провести рефакторинг.
- В ходе проверок экспериментальных задач выяснилось, что некоторые решения, которые проходят тесты не соответствуют теоретическим выкладкам об их эффективности, так что можно добавить тесты на производительность.

[4]Пулл реквест: https://github.com/Aunsiels/pyformlang/pull/20 (дата обращения: 31 мая 2024 г.)

[5]Репозиторий курса на GitHub: https://github.com/FormalLanguageConstrainedPathQuerying/formal-lang-course (дата обращения: 31 мая 2024 г.)

- Продолжать вносить вклад в используемые библиотеки. Например, добавить в pyformlang генератор строк по регулярному выражению.

# Список литературы


[1] Estimating probabilistic context-free grammars for proteins using contact map constraints / Witold Dyrka, Mateusz Pyzik, François Coste, Hugo Talibart // PeerJ. — 2019. — Vol. 7. — P. e6559. — Publisher: PeerJ Inc. URL: https://peerj.com/articles/6559 (дата обращения: 2024-05-26).

[2] Abzalov Vadim, Pogozhelskaya Vlada, Kutuev Vladimir, Grigorev Semyon. GLL-based Context-Free Path Querying for Neo4j. — 2023. — 2312.11925.

[3] Hagberg Aric A., Schult Daniel A., Swart Pieter J. Exploring Network Structure, Dynamics, and Function using NetworkX // Proceedings of the 7th Python in Science Conference / Ed. by Gaël Varoquaux, Travis Vaught, Jarrod Millman. — Pasadena, CA USA, 2008. — P. 11 – 15.

[4] Shemetova Ekaterina, Azimov Rustam, Orachev Egor et al. One Algorithm to Evaluate Them All: Unified Linear Algebra Based Approach to Evaluate Both Regular and Context-Free Path Queries. — 2021. — 2103.14688.

[5] Parr Terence. The definitive ANTLR 4 reference // The Definitive ANTLR 4 Reference. — 2013. — P. 1–326.

[6] Romero Julien. Pyformlang: An Educational Library for Formal Language Manipulation // SIGCSE. — 2021.

[7] TAG Parsing with Neural Networks and Vector Representations of Supertags / Jungo Kasai, Robert Frank, R. Thomas Mccoy et al. // Conference on Empirical Methods in Natural Language Processing. — Copenhague, Denmark, 2017. — P. 1712 – 1722. — URL: https://hal.science/hal-01771494.

[8] Krekel Holger, Oliveira Bruno, Pfannschmidt Ronny et al. pytest x.y. — 2004. — URL: https://github.com/pytest-dev/pytest.