**Министерство науки и высшего образования Российской Федерации**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА**

# ГЕНЕРАЦИЯ KOTLIN КОДА С ПОМОЩЬЮ НЕЙРОННОЙ СЕТИ АРХИТЕКТУРЫ TRANSFORMER ДЛЯ ФАЗЗИНГА КОМПИЛЯТОРА

Автор: Тихонов Виталий Андреевич _____

Направление подготовки: 01.03.02 Прикладная математика и информатика

Квалификация: Бакалавр

Руководитель ВКР: Фильченков А.А., канд. физ.-мат. наук _____

Санкт-Петербург, 2021 г.

Обучающийся Тихонов Виталий Андреевич
Группа М3436  Факультет ИТиП

Направленность (профиль), специализация
Математические модели и алгоритмы в разработке программного обеспечения

Консультанты:

а) Петухов В.А., без степени, без звания _____

ВКР принята «_____» _____ 20___ г.

Оригинальность ВКР _____%

ВКР выполнена с оценкой _____

Дата защиты «15» июня 2021 г.

Секретарь ГЭК Павлова О.Н. _____

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

**Министерство науки и высшего образования Российской Федерации**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»**

**УТВЕРЖДАЮ**

Руководитель ОП

проф., д.т.н. Парфенов В.Г. _____

«____» _____ 20___ г.

# ЗАДАНИЕ
## НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

**Обучающийся** Тихонов Виталий Андреевич

**Группа** M3436 **Факультет** ИТиП

**Квалификация:** Бакалавр

**Направление подготовки:** 01.03.02 Прикладная математика и информатика

**Направленность (профиль) образовательной программы:** Математические модели и алгоритмы в разработке программного обеспечения

**Тема ВКР:** Генерация Kotlin кода с помощью нейронной сети архитектуры Transformer для фаззинга компилятора

**Руководитель** Фильченков А.А., канд. физ.-мат. наук, доцент, научный сотрудник Университета ИТМО

**2 Срок сдачи студентом законченной работы до:** «31» мая 2021 г.

**3 Техническое задание и исходные данные к работе**

Требуется попробовать применить нейронные сети, построенные на архитектуре Transformer для генерации кода на языке Kotlin. В архитектуру сети должна быть заложена генерация кода не только в соответствии с грамматикой языка (правила синтаксиса), но и в соответствие с некоторыми правилами семантики.

**4 Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)**

а) Провести обзор существующих архитектур нейронных сетей Transformer и сделать аргументированный выбор трансформера для генерации кода.

б) Изучить существующие способы делать code embedding и выбрать наиболее подходящий для решаемой задачи.

в) Обучить сеть на наборе тестов компилятора и/или фрагментов кода в багтрекере.

г) Применить построенный генератор кода для поиска проблем в компиляторе Kotlin: выбрасываемых исключений и проблем с производительностью.

**5 Перечень графического материала (с указанием обязательного материала)**

Графические материалы и чертежи работой не предусмотрены

**6 Исходные материалы и пособия**

Исходные материалы и пособия работой не предусмотрены

**7 Дата выдачи задания** «01» сентября 2020 г.

Руководитель ВКР _____

Задание принял к исполнению _____ «01» сентября 2020 г.

**Министерство науки и высшего образования Российской Федерации**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»**

# АННОТАЦИЯ
## ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

**Обучающийся:** Тихонов Виталий Андреевич

**Наименование темы ВКР:** Генерация Kotlin кода с помощью нейронной сети архитектуры Transformer для фаззинга компилятора

**Наименование организации, в которой выполнена ВКР:** Университет ИТМО

## ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

1 Цель исследования: Разработать модель поддерживающую обучение элементам семантики языка для более качественной генерации кода

2 Задачи, решаемые в ВКР:

а) Реализовать модель для генерации кода

б) Раширить модель, поддержав обработку некоторой семантичекой информации

в) Оценить пользу от добавления семантичекой информации

г) Использовать модель для генерации кода и выявления проблем компилятора

3 Число источников, использованных при составлении обзора: 0

4 Полное число источников, использованных в работе: 5

5 В том числе источников по годам:

| **Отечественных** | | | **Иностранных** | | |
|---|---|---|---|---|---|
| Последние 5 лет | От 5 до 10 лет | Более 10 лет | Последние 5 лет | От 5 до 10 лет | Более 10 лет |
| 0 | 0 | 0 | 5 | 0 | 0 |

6 Использование информационных ресурсов Internet: да, число ресурсов: 5

7 Использование современных пакетов компьютерных программ и технологий:

| **Пакеты компьютерных программ и технологий** | **Раздел работы** |
|---|---|
| IntelliJ IDEA | 2, 3 |
| PyCharm | 2, 3 |
| Google Colab | 3 |

8 Краткая характеристика полученных результатов

На данный момент результатов получено не было

9 Гранты, полученные при выполнении работы

нет

10 Наличие публикаций и выступлений на конференциях по теме выпускной работы

нет

Обучающийся Тихонов В.А. _____

Руководитель ВКР Фильченков А.А. _____

«____» _____ 20___ г.

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Процесс тестирования очень важен при разработке приложений. Причина в том, что продукт, работающий некачественно может доставлять пользователям дискомфорт, приводить к потере времени и денег, а это в свою очередь может стать поводом для отказа от продукта. Еще важнее тестировать приложения, не являющиеся конечным продуктом, ведь проблемы в них могут затронуть еще большее число пользователей. Одним из таких приложений являются компиляторы языков программирования.

Классический подход в разработке тестов - написание тестов программистами, сразу после обновления функционала программы. Очевидно, что таким способом сложно протестировать настолько большую программу как компилятор достаточно хорошо, поэтому необходимы и другие методы тестирования.

Примером такого метода может быть фаззинг. В процессе фаззинга для тестируемой программы генерируется большое количество экземпляров входных данных. Для каждого примера входных данных программа запускается независимо, и исследуются некоторые характеристики ее работы, такие, например, как затрачиваемые память и время. Входные данные на которых поведение программы аномально (например, фиксируется большое количество затраченной памяти или долгое время работы) выделяются для дальнейшего изучения программистом.

В данной работе в качестве тестируемого приложения выбран компилятор языка Kotlin. В случае с фаззингом компитлятора в качестве входных данных будут использоваться программы, написанные на соответствующем языке, а критериями аномальности могут служить время компиляции, затраченная память, выброшенные исключения, отличающееся поведение откомпилированных входных программ на разных платформах или разных версиях компилятора.

Получается, что в данном случае задача тестирования сводится к задаче генерации кода. Есть два основных аспекта, которые следует учитывать при генерации кода - это синтаксис и семантика языка. С синтаксисом языка программирования не должно возникать серьезных проблем - он описывается формальной грамматикой, и нет трудностей в генерации кода на ее основе. С семантикой все сложнее - у нее нет формального описания. В случае Kotlin’а она описывается спецификацией на естественном языке. Реализовать генера-

тор, полностью поддерживающий спецификацию - задача сопоставимая с разработкой самого компилятора. Более того, генератор будет иметь неформальную основу, а значит высоки шансы допустить ошибки при его разработке.

При этом опираться только на синтаксис нельзя - доля семантически некорректных, а значит некомпилируемых программ слишком высока. Поэтому возникает желание изучить семантику с помощью машинного обучения, вместо того чтобы формализовывать ее или писать сложный генератор, описывая всю спецификацию. Однако современные решения почти не предусматривают непосредственную передачу модели семантической информации в качестве входных данных.

Таким образом формулируется цель работы - разработать модель нейронной сети, обучаемую семантике языка Kotlin для генерации корректного кода для фаззинга компилятора.

В первой главе описываются некоторые существующие подходы к извлечению информации из исходного кода, выбирается базовый подход для дальнейшего улучшения. Во второй главе рассмативаются возможные улучшения базового подхода за счет добавления информации о семантике. Заключительная третья глава рассказывает о процессе реализации, проблемах, возникших на этом этапе и их устранении.

# ГЛАВА 1. ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ

Для начала необходимо разобраться разобраться как работать с такими сложными данными как исходный код. Модели машинного обучения отлично умеют работать с числами. Однако работать напрямую с текстовой информацией они не могут. Тут на помощь приходит механизм эмбедингов - данные разбиваются на структурные компоненты (например, текст можно разбить на слова, а исходный код на токены), и каждой такой компоненте сопостовляется вектор чисел, называемый эмбедингом. Таким образом каждый элемент кодируется точкой в некотором пространстве. Обработанные таким образом данные уже можно подавать на вход модели.

Существуют способы предлагать эти вектора по-умному, сохраняя некоторую семантическую информацию. Например, может получиться так, что вектор между точками, соответствующим словам ”Москва”и ”Россия”будет коллинеарен аналогичному вектору для слов ”Прага”и ”Чехия”. То есть такой вектор будет иметь смысл ”x является столицей y”.

Текст имеет достаточно простую структуру - это по сути просто последовательность слов. Для работы с последовательностями существуют рекурентные нейронные сети, принимающие данные фермент за фрагментом. Исходный же код имеет более сложную древовидную структуру. Это приводит к некоторым интересным спецэффектам. Например, на уровне файла мы можем поменять местами определеня двух функций, и код от этого никак не изменится. Однако, если в тексте поменять местами два абзаца, то смысл может значительно поменяться. Другим примером являются многозначные слова. В случае с Java это может быть слово `final` или `static`, имеющие разные значения в зависимости от контекста, а конструкция `String s` может быть как объявлением переменной, так и параметром функции. В Kotlin’е есть ключевые слова `get`, `set`, `field`, которые в то же время могут использоваться как идентификаторы. Выходит, что рассматривать код как обычный текст - не лучшая затея, так как часть информации из входных данных просто не будет использоваться.

Другое хорошо изученное представление данных - изображения. Для работы с ними так есть целый класс нейронных сетей - сверточные. Изображения рассматриваются как двумерные объекты, полностью заполняя некоторый прямоугольник.

Графовое предствление данных находится где-то по середине, не умещаясь в одномерное пространство, при этом не заполняя полностью двумерное. Поэтому для графов как правило используются специальные подходы.

## 1.1. Способы обработки кода

Код можно рассматривать в некотором смысле как текст, а значит модели машинного обучения не смогут работать с ним напрямую. Поэтому необходимо разобраться в подходах, позволяющих представлять исходный код в виде векторов. Рассмотрим подробнее несколько подходов по представлению исходного кода и выберем наиболее подходящий для генерации кода.

### 1.1.1. Тривиальные подходы

Наиболее очевидным подходом к извлечению информации из кода являются разбение исходного кода программы на токены и обучение модели, основанной на рекуррентной сети на полученной последовательности. Как было сказано ранее, в таком случае часть информации утрачивается и не учитывается моделью.

Более разумным будет построение дерево разбора для исходного кода. Деревья разбора содержат гораздо больше полезной информации, чем код, представленный в виде текста. Например, если код из листинга 1 рассматривать последовательно, то мы поймем, что есть некоторый класс `A`, с которым ассоциируются поля `x: Int` и `y: Int`. При этом дерево разбора этого фрагмента, изображенное на рисунке 1 слева имеет более сложную структуру: тут явно указано, что `x: Int` относится к конструктору класса.

Листинг 1 – Пример исходного кода на Kotlin

```
class A(val x: Int) {
    val y: Int = 10
}
```

Чтобы модель могла работать с деревом предлагается обучить эмбединги для каждого типа вершин, а само дерево представять как последовательность вершин в порядке обхода в глубину. Полученную последовательность можно отдавать на вход модели на основе рекуррентной сети. Для графа, изображенного справа на рисунке 1 обход в глубину будет выгляеть следующим образом: $P$, $A$, $B$, $D$, $E$, $F$, $C$. Но не смотря на то что самом графе расстояние между

вершинами $B$ и $C$ равно двум, в полученной последовательности путь значительно вырос и стал равен четырем. Для решения такой проблемы можно добавить в путь стрелку вверх, если при переходе к следующей вершине делается подъем, и стрелку вниз, если делается спуск. В таком случае результат обхода будет следующим: $P, \downarrow, A, \downarrow, B, \downarrow, D, \downarrow, E, \downarrow, F, \uparrow, \uparrow, \uparrow, \uparrow, \downarrow, C$. Стрелки - специальные символы, для которых также обучаются эмбедингги.

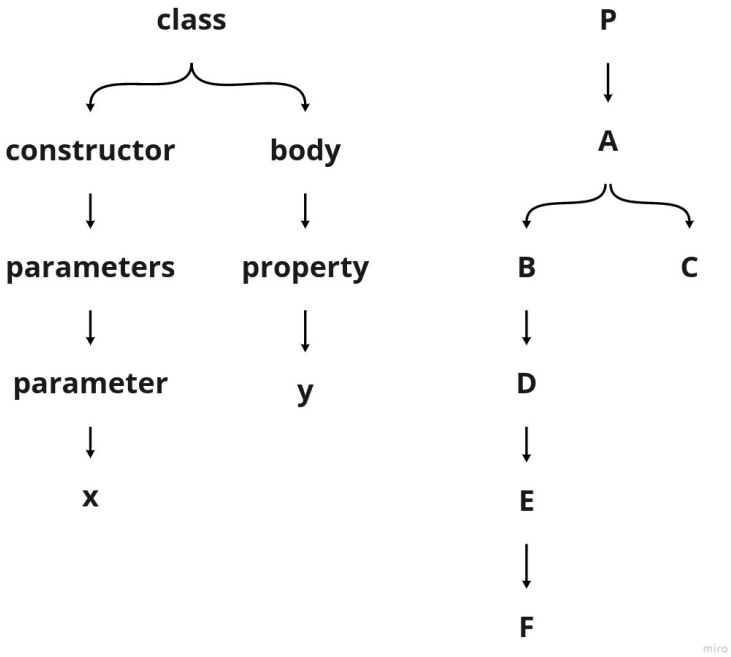


Рисунок 1 – Слева AST для листинга 1, справа - пример дерева

### 1.1.2. Графовые нейронные сети

Значительно более мощным является подход с использованием графовых нейронных сетей (Graph Neural Network - GNN). Каждой вершине сопостовляется вектор состояния, после чего происходит несколько этапов обмена информации между вершинами. Этап состоит из двух фаз: отправка сообщений и получение сообщений. Во время первой фазы для каждой вершины вычисляется сообщение для отправки. В качестве сообщения используется вектор состояния, преобразованный каким-либо образом. После того как все сообщения вычислины, вершины передают сообщения своим соседям, начинается вторая фаза. Во время фазы получения сообщений каждая вершина агрегирует получаемые от других вершин сообщения, например, в работе [2] сообщения взвешенно усреднются с использованием механизма внимания. Состояние вершины обновлятеся с помощью полученного только что агрегированно-

го сообщения. Это можно сделать, например, с помощью конкетенации этих векторов и линейного предобразования для уменьшения размерности конкатенированного вектора до исходной размерности состояния вершины.

Простой вариант - использовать AST в качестве графа. Однако в статье [1] предлагается дополнить этот граф ребрами data-flow графа. Например, можно соеденить вызов функции и возвращаемое значение из опредленение функции или аргументы вызова с параметрами функции. В таком случае информация по графовой нейронной сети в некоторых случаях будет передаваться также как и в исходном коде.

### 1.1.3. Дерево разбора как список правил

Дерево разбора можно представить в виде последовательности, но значительно хитрее, чем в описанном ранее случае, где дерево заменялось на список вершин в порядке обхода в глубину. Подход, используемый в статье [5] рассматривает синтаксическое дерево как последовательность приминения правил грамматики.

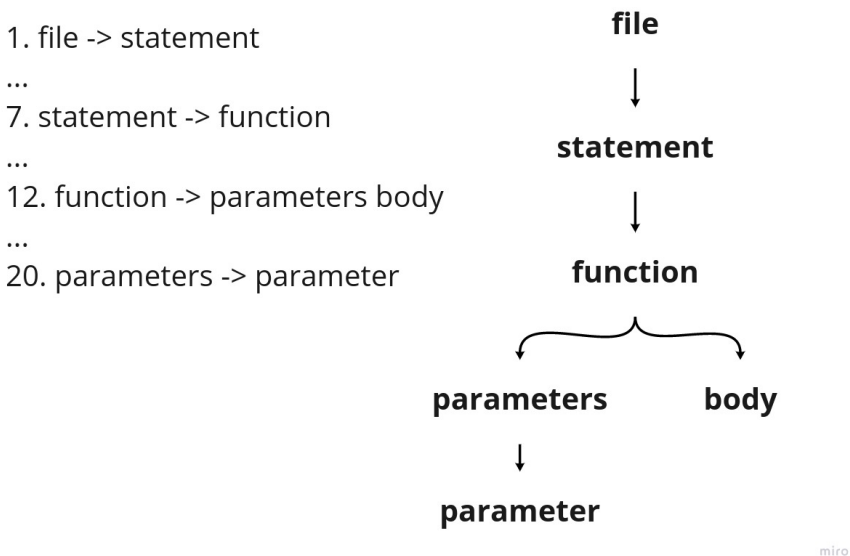


Рисунок 2 – Слева некоторые правила грамматики, справа - дерево, получающееся при последовательном применении правил 1, 7, 12, 20

Рассмотрим пример. Слева на рисунке 2 указаны некоторые правила грамматики. Будем считать, что `file` - это стартовый нетерминал, тогда после использования правил 1, 7, 12, 20 дерево примет вид изображенный справа на том же рисунке. В качестве текущего нетерминала для раскрытия выбирается первый в порядке обхода в глубину. То есть следующее правило будет раскры-

вать нетерминал `parameter`. Такое решение описывает исходный код очень близко к грамматике.

### 1.1.4. Дерево разбора как набор путей

Еще один подход для представления дерева разбора кода рассмотрен в статье [3]. При таком способе AST представляется в виде набора путей между всеми парами вершин в дереве. Далее эбмединги путей проходят через блок трансформера, механизм внимания позволяет выделить наиболее важные пути, после чего они взвешенно суммируются. Таким образом вычисляется векторное представление примера кода.

Логику этого подхода продолжает статья [4]. Там ставится задача дополнения кода (code completion), то есть предсказания вершины по контексту. В качестве входных данных на очередном шаге используются пути от всех листьев до родителя предсказываемой вершины, а также путь от корня дерева.

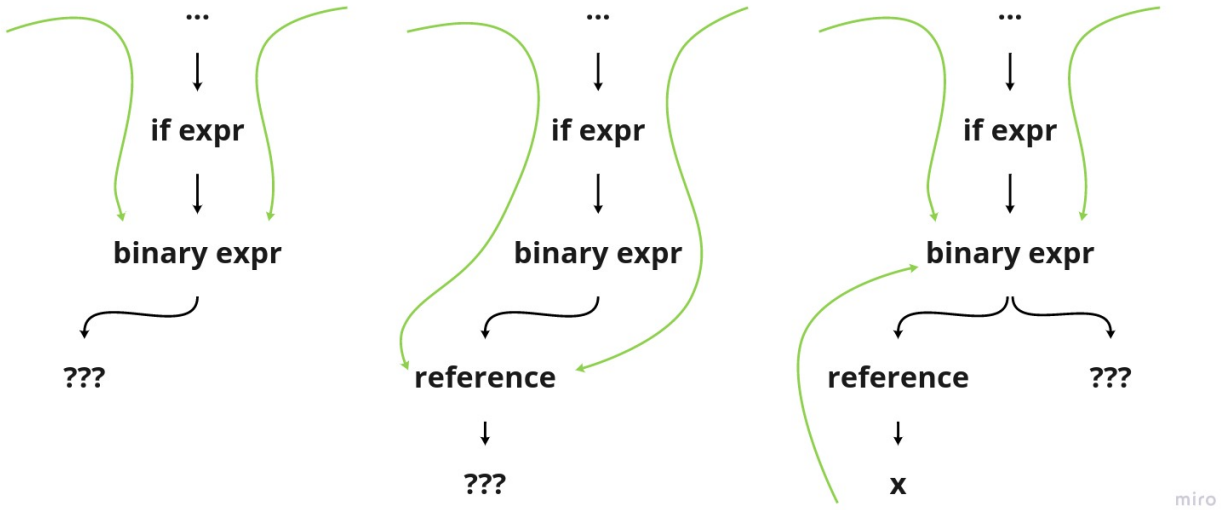


Рисунок 3 – Несколько шагов работы модели по дополнению кода

На рисунке 3 демострируется несколько шагов такого подхода. За `???` обозначается вершина, которую необходимо предсказать, к ее родителю идут зеленые стрелки, показывающие пути от листьев. Интересно отметить переход от второй картинки к третьей. На втором шаге было предсказан `x` - ребенок вершины `reference`. Так как `x` является листом, то модель прекращет генерацию в этом поддереве и начитает собирать пути ко второму ребенку `binary expr`. Важным моментом является то, что также учитывается путь от только что сгенерированного поддерева с `reference` и `x`.

нужно рассказать что-то про трансформеры

## 1.2. Подходы к извлечению семантической информации

Ранее описанные механизмы делают больший акценкт на синтаксис языка, за исключением некоторых деталей. Как уже было сказано, ислючить семантику совсем нельзя, а значит данные модели самостоятельно обучаются семантике. Рассмотрим некоторые способы непосредственной передачи семантической информации модели.

### 1.2.1. Семантические ребра

В обзоре подхода из статьи [1] уже было сказано о том, что графовая нейронная сеть обучается не просто на дереве разбора кода, а на его расшрении за счет ребер из data-flow графа. По идее такой способ представления кода должен способствовать усвоению не только синтаксической информации (за счет использования AST), но и семантической (за счет ребер data-flow графа).

Также в данной статье присутствует способ работы с типами для случая статически типизируемых языков, когда типы выражений известны на этапе компиляции. Для каждого типа предлагается использовать обучаемое векторное представление. Эмбедингом типа считается максимальное значение среди таких представлений для самого типа и его родительских типов. Использование операции максимум по сути позволяет обучить решетку на типах, то есть упорядочить их в иерархию.

### 1.2.2. Предсказание типов

Использование в [1] информации о типах наталкивает на мысль о важности этих данных. Действительно, знание типов в процессе разработки значительно помогает программисту - так разработчик понимает устройство сущности с которой он работает не только из ее названия, но и из структурного описания.

рассказ о LambdaNet

## 1.3. Улучшение модели

Практически во всех указанных работах модели не используют напрямую семантическую информацию. Исключением является статья [1] о применении графовых нейронных сетей для представления исходного кода, где внедряют информацию о типах объектов, обучая решетку на типах. Такой подход позволяет частично сохранить иерархию типов, даже для новых типов, не

видимых на этапе обучения. Однако типы определяются не только иерархией. Кроме нее, есть как минимум поля и методы, которые можно вызвать на экземпляре класса.

Важно отметить, что информация о типе не является чем-то недоступным, ее можно получить на этапе компиляции. Поэтому лучше попытаться использовать имеющуюся информацию, чем надеятся, что модель самостоятельно ее распознает и выучит.

Таки образом ставится задача разработать и внедрить механизмы работы с семантикой в модель, в частности для передачи более сложной информации о типах выражений.

## Выводы по главе 1

В данной главе рассматриваются некоторые способы представления исходного кода программы для работы с моделями машинного обучения, а также подходы к извлечению семантической инфорации из кода.

# ГЛАВА 2. ОБРАБОТКА СЕМАНТИКИ

В данной главе будет выбрана базовая модель для последующего улучшения. Также будут предложены подходы для ее расширения с целью использования семантичекой информации.

## 2.1. Выбор модели

Для задачи генерации кода графовые нейронные сети не самый лучший вариант. Во-первых, не приминялись ??. Во-вторых, они тяжело учатся ??.

Подход со списком правил выглядит неплохим вариантом, но что? можно сказать что он постарее, чем SLM и нигде(?) больше не применялся. Тем не менее его можно рассмотреть в дальнейших исследованиях.

В данной работе в качестве основы выбран подход из статьи Structural language models of code. В целом подход себя зарекомендовал для решения таких задач как предсказание имени метода и генерация описания кода на естественном языке. Более того, конкретно в этой статье подход используется для генерации (пусть и небольших фрагментов) кода.

Генерация среднего кода. Почему это не плохо - статья Даниила

## 2.2. Интерграция типов

Типы в языках программирования устроены как правило достаточно сложным образом. Поэтому необходимо выбрать некоторое упрощенное описание типов и на его основе разработать и обучить модель. Каким же образом можно описать типы? В первом приближении тип описывается набором супертипов, а также его членами - методами и полями. Однако, есть и более сложные моменты - inner классы, вложенные классы, типовые параметры и многое другое, но в данной работе они не рассматриваются.

### 2.2.1. Особенности описания типов

Даже с таким описанием класса встречаются интересные ситуации. Например, наличие возможности переопределения полей и методов в классах-наследниках. В таком случае информация об этих сущностях будет содержаться и в родительском классе и в наследнике. Для избежания дублирования информации есть следующее предложение: давайте будем игнорировать все переопределения. Инфомация о наличии метода с определенной сигнатурой или поля определенного типа в любом случае есть в описании родительского класса.

Другой интересный вопрос - это имена классов. Есть ли разница между классами `A` и `B` в листинге 2? С одной стороны они абслютно идентичны в структурном плане и нет необходимости различать их как два различных типа - с экземпляром каждого из них можно делать один и тот же набор действий. С другой стороны такой подход имеет слабое место, например, экземляр `d` класса `D` можно привести к супертипу `A`, который, как сказано ранее эквивалентен классу `B`, а значит в таком подходе валидно приведение типа `d as E`, хотя в иерархии классов `D` и `E` расположены в разных местах. Тем не менее это не должно приводить к некомпилируемым результатам, поэтому мы пренебрежем данной особенностью.

Листинг 2 – Пример исходного кода на Kotlin

```
class A(val x: Int)
class B(val y: Int)
class C(val x, Int, val y: Int)
class D(val y: Int) : A(100)
class E(val z: Double) : B(100)
```

Отметим, что классы `C` и `D` будут считаться различным, не смотря на то, что оба содержат два поля типа `Int` (`C` напрямую, `D` - один напрямую, а второй через наследоавние от `A`).

Выходит, что формулируемый подход описывает класс с точки зрения того, что можно делать с его экземпляром - какие методы вызывать и к полям какого типа обращаться. Однако, наличие у класса членов с модификаторами `private` и `protected` совсем не сочитаются с данным подходом из-за того, что доступа к ним из вне класса нет. Заметим, что наличие для приватного поля публичных методов для его модификации наполнит описание класса информацией ровно об этих методах, но не о соответсвующем поле. Вопрос что конкретно делают эти методы, действительно ли что-то модифицируют не имеет смысла в рамках поставленной задачи. Понимая, что `private` и `protected` члены класса не играют важной роли, а возможно даже вредят, мы исключим информарцию о них из описания класса.

### 2.2.2. Определение порядка построения эмбедингов типов

Вновь обращаясь к листингу 2 заметим, что нет особого смысла создавать эмбединг сначала для класса `D`, а только потом для класса `A`, ведь `D` должен содержить информацию о классе `A`. Сразу на ум приходит разумная идея -

давайте посторим граф зависимостей типов, проведем на вершинах топологическую сортировку и построим эмбединги типов в порядке обратном данной сортировке.

Однако такой граф будет содержать циклы, что делает построение сортировки невозможным. Примеры ”плохого”кода представлены в листинге 3. В первом случае `Tree` - обычный рекурсивный тип, порождающий петлю в графе зависимостей. Во втором случае ни `A`, ни `B` не являются рекурсивными сами по себе, но образуют циклическую зависимость, ссылаясь друг на друга. В последнем примере класс `D` не имеет ни одного поля непостредственно, но становится рекурсивным из-за наследования от класса `C`, содержащего поля типа `D`.

Листинг 3 – Типы с циклическими зависимостями

```
class Tree(var t: Tree?)

class A(var b: B?)
class B(var a: A?)

class C(var d: D?)
class D : C
```

Нужна оценка на долю рекурсивных классов Так как доля таких ситуаций крайне мала, предлагается просто исключить все рекурсивные зависимости, а именно удалять одно из ребер, в каждом цикле. Важно понимать, что после того как класс лишается зависимости на какой-либо тип, он больше не может содержать поля такого типа, а его функции-члены не могут иметь этот типа в качестве параметра или возвращаемого значения. Такие члены класса придется удалить. После указанного преобразования представления классов будут менее точными, но все еще насыщенными большим объемом информации.

При внимательном рассмотрении структуры типа `Any`, а также примитивных типов становится ясно, что все они содержат рекурсивные зависимости. Например, `Any` содержит метод `equals: (Any?)` $\rightarrow$ `Boolean`, что создает один цикл с самим собой и еще один цикл с классом `Boolean`. А любой числовой тип содержит в частности методы `compareTo(X)`, где в качестве `X` может быть любой из числовых типов `Byte, Short, Int, Long, Float, Double`.

Лишаться информации об этих методах совсем не хочется, поэтому предгалается следующее решение. Определим набор базовых типов, которые будут своего рода кирпичиками при построении более сложных типов. В этот набор войдут числовые типы - `Byte, Short, Int, Long, Float, Double`, типы, связанные со строками - `Char, CharSequence`, логический тип `Boolean` и типы общего вида `Any, Unit`. Так как для базовых типов постороение эмбедингов исходя из структуры этих классов невозможно по причине рекурсивных зивисимостей, эмбеденги будут предложены напрямую. При этом информация о их методах будет использована в процессе обучения для корректировки их векторных представлений.

Типы из выбранного базового набора можно запросто увидеть в любой программе, более того этот набор ограничен, а значит не должно возникнуть проблемы с обучением эмбедингов для этих типов. Важно, что такой подход никак не поможет для других типов с циклическими зависимостями - их пространство слишком велико, и мы не можем выдать всем им эмбединги напрямую.

### 2.2.3. Алгоритм построения эмбедингов типов

Для конструирования модели для эмбедингов типов небходимо ответить на следующие вопросы в общем случае: “сколько параметров в функции?”, “сколько членов в классе?”, “сколько супертипов имеет класс?”. Ответ оказывается достаточно простым - сколько угодно. В виду отсутствия каких-либо ограничений предлагается использовать реккурентные нейронные сети для сбора информации о параметрах, членах класса и супертипах. Так как в каждом таком случае информация может иметь разные свойства, будем использовать три различных нейронных сети с разными весами.

Опишем алгоритм более строго:

а) Для базовых типов выберем случайные обучаемые векторные предствления
б) Для каждого класса построим его эмбединг, если это не сделано ранее:
    1) Рекурсивно получим эмбединги для всех классов, от которых зависит текущий
    2) Построим эмбденги для всех функций членов текущего класса:
        i. Пропустим через рекуррентную нейронную сеть все параметры, получая векторное представление списка параметров

   ii. С помощью линейного преобразования объеденим векторное представления списка параметров и возвращаемого типа
3) Пропустим через вторую рекуррентную нейронную сеть эмбединги всех функций членов и эмбединги типов всех полей, получая векторное предствление класса вне иерархии типов
4) Пропустим через третью рекуррентную нейронную сеть эмбединги всех супертипов, а также эмбединг с предыдущего шага

Стоит отметить, что не все классы содержат члены, и не все функции имеют хотя бы один параметр. Поэтому для корректной работы необходимы обучаемые вектора, означающие пустой список параметров и пустой список членов класса.

После выполнения алгоритма у нас имеются векторные представления для всех классов и функций в экземпляре исходного кода. Однако совершенно недостаточно получить случайные векторные представления, необходимо получить именно такие, которые сохраняют полезную информацию о типе. Проверка этой информации и будет основой при обучении.

Мы обладаем некоторыми знаниями о всех типах, например, что тип `A` является подтипом `B` или что функция `F` возвращает значение типа `C`. Поэтому оценивать качество получившихся векторных представлений типов можно на основе соответствующих вопросов - “`A` подтип `B`?”, “`F` возвращает `C`?”, решая задачу бинарной классификации.

При интеграции типов в основную модель нужно помнить о двух вещах. Во-первых, так как мы обладаем информацией о типах некоторых вершин в AST, например, тип вершины, отвечающей за вызов функции может соответствовать типу возвращаемого значения этой функции, хочется передавать эту информацию модели. Во-вторых, нужно каким-то образом получать результаты на основе типов, например, для новых переменных можно предсказывать конкретный тип, чтобы указывать его при генерации подставлять его при генерации.

Исходя из этого предлагается для описания вершины в AST использовать конкатенацию векторного представления вида вершины (например, `RETURN` или `FUN`) и векторного представления типа, если он есть. Аналогичные изменения должны быть с выходом модели, то есть полученный вектор можно разбить на два - отвечающий за вид вершины и отвечающий за тип.

### 2.3. Механизм обучения

В оригинальной модели в качестве ожидаемого результата при обучении используется тот вид вершины, который был в изначальном примере. Такой подход не соответствует цели, поставленной в данной работе, поэтому необходимо изменить механизм обучения.

#### 2.3.1. Обучение синтаксису

Первое, с чем приходится сталкиваться, это синтаксическая корректность - мы хотим, чтобы код был правильным с точки зрения грамматики. Поэтому будем штрафовать модель за предсказание заведомо не подходящих видов вершин. Соответсвенно, предсказаниямя с наибольшей вероятностью будут подходящими с точки зрения синтаксиса.

Существуют такие случаи, когда добавление вершины в AST делает его синтаксически некорректным, однако при продолжении наращивания дерева ошибки пропадают. Примерами могу послужить операции доступа через точку $\star$`.`$\star$ или бинарная операция $\star$ `+` $\star$. В обоих случаях за $\star$ обозначаются отсутствующие дети, необходимые для компилируемости кода.

Из-за того, что в ходе работы не обнаружено простого способа понять является ли добавление очередной вершины в дерево синтаксинтаксически некорректным в принципе, при любых дальнейших действиях, или только по причине необходимости наличия детей у нее самой, был реализован простой механизм такой проверки. Основываясь на датасете, определим следующее отношение: {(A, B) | сущетсвует такое AST, в котором вершина вида A содержит как ребенка вершину вида B}. Использовать его как критерий синтаксической корректности не очень надежно, например, оно не мешает иметь вершине вида `IF` сразу двух детей вида `CONDITION`.

Первое, что мы сделаем для усложения этого отношения будет использование информации о видах детей слева от предсказываемой вершины: {(A, B, LB) | сущетсвует такое AST, в котором вершина вида A содержит как детей вершины видов LB, а затем вершину вида B}. Такой подход позволит избавиться от ошибки, указанной выше, однако порождает новую: число различных комбинаций детей, например, у `FILE` крайне велико (в отличие от `IF`, который имеет только $[\varnothing]$, `[CONDITION]`, `[CONDITION, THEN]` и `[CONDITION, THEN, ELSE]`), а значит мы очень сильно ограничиваемся разнообразием датасета. Поэтому вторая модификация залючается в том, что для подобно-

го рода вершин мы не будем требовать порядка на детях. Помимо `FILE` это коснется блоков кода, выражений `when`, строк (из-за интерполяции), списков параметров и аргументов значений и типов.

### 2.3.2. Обучение семантике

На самом деле проблема из предыдущего раздела про некомпилируемость кода из-за недостроенности дерева разброра не ограничивается синтаксисом. Рассмотрим пример из листинга 4. Пусть решено инициализировать переменную с помощью условоного оператора, однако конструкция (2) не валидна даже с точки зрения синтаксиса, так как отсутвует условие. Но добавления условия тоже недостаточно, ведь `if` используется для инициализации и типы блоков `THEN` и `ELSE` должны соответствовать типу пемеменной `x`. Поможет только генерация всего поддерава вершины вида `IF`.

Листинг 4 – Возможная последовательная генерация

```
(1) val x: Int = ???                                    //не компилируется
(2) val x: Int = if (???) { ??? } else { ??? }          //не компилируется
(3) val x: Int = if (true) { ??? } else { ??? }         //не компилируется
(4) val x: Int = if (true) { 4 } else { ??? }           //не компилируется
(5) val x: Int = if (true) { 4 } else { 5 }             //компилируется
```

Исходя из этого примера, стратегия обучения должна давать возможность модели предсказать поддерево целиком. Если получен компилируемый результат, модель должна быть поощрена за свой выбор. Если результат не коспилируется, это может означать либо что полученный код не приводим к компилируемому ни при каких условиях, либо он пока еще не приведен, как в пунктах 2-4 в листинге 4. Чтобы понять с чем мы столкнулись, дадим модели завершить несколько поддеревьев. Если даже после завершения нескольких поддеревьев код не стал компилируемым, значит высока вероятность, что он таким и не станет, и модель должна быть оштрафована за принятие своих решенией.

## 2.4. Подход к оцениваю

1. Метрики для генерации

предсказываемая вершина vs оригинальная

обучить без семантики????

доля семантически-верного кода

2. Метрики для фаззинга

запуск компилятора на сгенерированном (в т.ч. с разными параметрами)

время работы / число вершин

запуск под разными платформами

## Выводы по главе 2

В данной главе приянто и обосновано решение об использовании представления исходоного кода на основе путей в дереве разбора, предложен механизм построения векторных представлений для типов и способ внедрения информации о типах в основную модель, описан способ обучения основной модели.

# ГЛАВА 3. РЕАЛИЗАЦИЯ И РЕЗУЛЬТАТЫ

## 3.1. Реализация на стороне Kotlin'а

кэширование деревьев, ALTree и извлечение путей

Архитектура генератора

## 3.2. Особенности реализации модели для типов и результаты обучения

Реализация модели для эмбедингов типов

(ну хотя бы конктреные вопросы)

Результаты ее обучения

## 3.3. Результат работы генератора

Результат генерации кода

## 3.4. Возможные улучшения

Для сохранения информации о типах выражений можно использовать более продвинутые подходы, например, на основе графовых нейронных сетей. Это позволит сохранять больше информации, в частности может позволить поддержать рекурсивные типы.

Также в ходе работы возникала, но не была реализована следующая идея. По аналогии с подходом из работы ссылка, где в дерево разбора добавлялись ребра, означающие семантические связи, хотелсь бы поддержать нечто подобное и в подходе на основе путей. Например, можно приминять к связанным вершинам некоторый оператор.

Листинг 5 – Примеры семантических операторов

```
(1) typed_node = node_with_type(node, type)

(2) private_node = is_private_node(old_node)
(3) new_call, new_return = call_return(old_call, old_return)
(4) new_param, new_arg = param_arg(old_param, old_arg)
```

Обратимся к листингу 5. В первой строке представлен оператор типизации вершины, по сути именно такой и используется в данной работе, однако три другие оператора - нет. Оператор (2) должен передать в вершину информацию о том, что она приватная. Для всех пар вызова и декларации соответсвующего метода можно приминить преобразование (3). То же касается и парных аргументов и параметров в случае оператора (4). Предполагается, что таким образом удастся внедрить дополнительную информацию в модель.

## Выводы по главе 3

фывфв

# ЗАКЛЮЧЕНИЕ

В данном разделе размещается заключение.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Miltiadis Allamanis Marc Brockschmidt M. K.* Learning to Represent Programs with Graphs [Электронный ресурс]. — 2018. — URL: https://arxiv.org/abs/1711.00740.

2 *имена не на английском*. Graph Attention Networks [Электронный ресурс]. — 2018. — URL: https://arxiv.org/abs/1710.10903.

3 *Uri Alon Meital Zilberstein O. L.* code2vec: Learning Distributed Representations of Code [Электронный ресурс]. — 2019. — URL: https://arxiv.org/abs/1803.09473.

4 *Uri Alon Roy Sadaka O. L.* Structural Language Models of Code [Электронный ресурс]. — 2020. — URL: https://arxiv.org/abs/1910.00577.

5 *Zeyu Sun Qihao Zhu Y. X.* TreeGen: A Tree-Based Transformer Architecture for Code Generation [Электронный ресурс]. — 2019. — URL: https://arxiv.org/abs/1911.09983.