Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 19Б.10-мм

Уткин Илья Николаевич

Разработка решения для поддержки новых парсеров и языков программирования в инструменте Astminer

Отчёт по учебной практике

Научный руководитель:
доц. кафедры СП, к. т. н. Т.А. Брыксин

Консультант:
программист-исследователь JetBrains Е.С. Спирин

Санкт-Петербург
2021

Оглавление

1. Введение

Сегодня написание кода не единственная ответственность для программного инженера. Почти любой код, помимо написания и внедрения в существующую архитектуру, также надо проанализировать, проверить на возможные ошибки, сопроводить документацией и тестами. Недостаточное внимание к этим факторам может привести к большим потерям денег и времени в будущем. Чтобы немного освободить программистов от этой рутины, можно использовать инструменты, основанные на машинном обучении [1].

Применение подобных техник к процессу создания программных продуктов не ново и активно изучается. Современные модели уже могут в некоторой степени предсказывать названия переменных, функций [2], оценивать общее качество кода [3], находить скрытые ошибки [4]. Используя данные техники, можно ускорить жизненный цикл программных продуктов и упростить работу программиста.

Для того, чтобы подобные инструменты имели существенную точность, необходимо правильно подбирать данные для обучения. Качество исходных данных и их предварительная обработка может повлиять на конечную точность модели, однако одним из самых существенных факторов является вид представления исходных данных. Выбор конечного представления данных определяет архитектуру модели, что в конечном итоге влияет и на общее качество обучения.

Ранее исходный код для обучения представлялся в качестве обычного текста, пробовались идеи из сферы обработки естественных языков [5]. Сегодня одними из самых популярных методов представления являются абстрактные синтаксические деревья (Abstract Syntax Tree, AST) и их производные (например представление в виде мешка путей [6] или дополненных графов [7]), которые позволяют лучше отражать структуру программы и ее семантику, чем обычный текст. Еще одним плюсом таких представления является простота получения AST из исходного кода программы, так как уже существует большое количество инструментов для синтаксического анализа. Однако, разные

синтаксические анализаторы продуцируют разные деревья для одной и той же программы, а так как затем с каждым деревом необходимо будет работать, то это сильно затормаживает процесс нахождения оптимального анализатора под конкретную проблему.

Для решения данной проблемы, был создан инструмент *astminer*[1] [8], позволяющий конвертировать исходный код в различные представления, фильтровать его и подготавливать в качестве тренировочных данных для обучения. На данный момент он поддерживает такие языки как Java, Javascript, C, C++ и Python и объединяет несколько синтаксических анализаторов под единым интерфейсом, позволяя использовать один и тот же код для разных парсеров и разбираемых языков.

Несмотря на единые интерфейсы для разных синтаксических анализаторов, добавление новых становилось все более трудоемким. Нагромождение сущностей, созданных новыми парсерами, сильно переплеталось с архитектурой, которая не должна зависеть от конкретных разбираемых языков и парсеров. В рамках данной работы рассматривается решение этой проблемы.

[1]Страница на github: https://github.com/JetBrains-Research/astminer

2. Цель и задачи

Целью данной работы является создание единого интерфейса, позволяющего взаимодействовать со специфичными для языков компонентами *astminer*, для дальнейшего масштабирования на новые языки программирования и анализаторы.

Для достижения этой цели были поставлены следующие задачи:

1. Изучить предметную область и текущую архитектуру *astminer*, отвечающую за разбор исходного кода
2. Спроектировать и разработать решение, позволяющее поддерживать новые парсеры и языки программирования в не зависимости от их числа
3. Интегрировать полученное решение в существующий инструмент, сопроводить его тестами и документацией
4. Провести апробацию внедренного решения

3. Обзор

3.1. Абстрактные синтаксические деревья

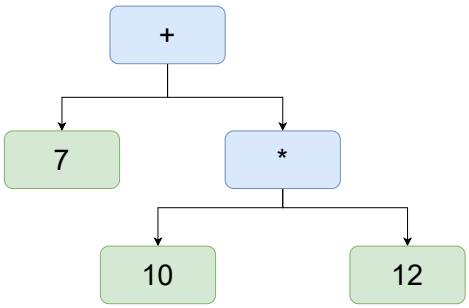
Любой текст описывается некоторыми наборами правил — грамматиками (Листинг 1). Правила в грамматиках выделяют некоторые синтаксические структуры, которые затем выделяют более сложные структуры и т.д.

Абстрактное синтаксическое дерево — результат применения некоторой грамматики к тексту.

Listing 1: Пример ANTLR грамматики для математических выражений

```
grammar Exp;
eval
    :    additionExp
    ;
additionExp
    :    multiplyExp
         ( '+' multiplyExp
         | '-' multiplyExp
         )*
    ;
multiplyExp
    :    Number
         ( '*' Number
         | '/' Number
         )*
    ;
Number
    :    ('0'..'9')+ ('.' ('0'..'9')+)?
    ;
WS
    :   (' ' | '\t' | '\r'| '\n') {channel=HIDDEN;}
    ;
```

Рис. 1: Полученное на выражении 7+10*12 AST

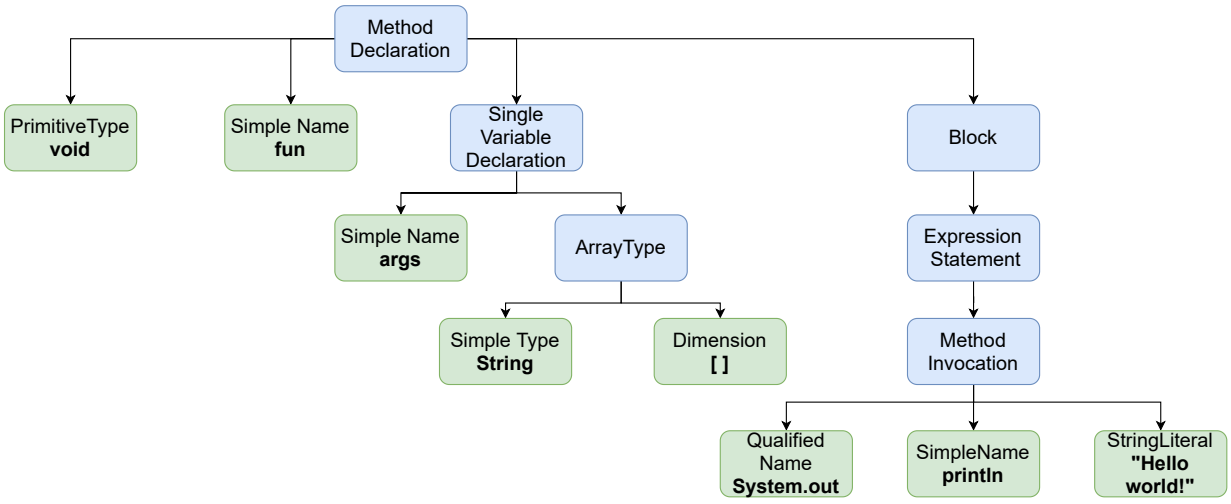


Применяя похожие грамматики к программам (как на примере листинга 2) мы можем получить представление программы в качестве древовидной структуры, где каждая вершина в дереве представляет собой элемент исходного кода: внутренние вершины - операторы (присваивания, циклы, условия и т.д.), а листья - операнды (переменные, вызовы функций и т.д.).

Listing 2: а) Функция написанная на Java

```
void fun(String[] args) {
    System.out.println(”Hello world!”);
}
```

Рис. 2: б) Полученное AST



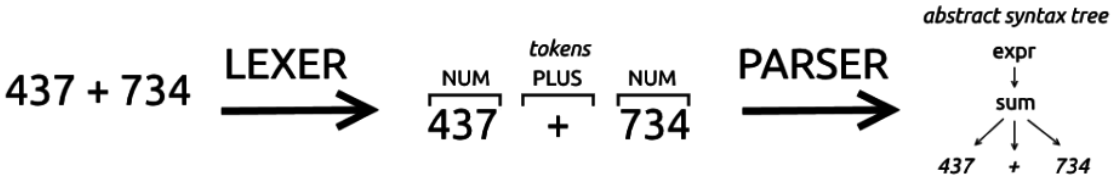
3.2. Обзор существующих синтаксических анализаторов

Создание собственных парсеров кропотливый и трудоемкий процесс. Даже написав парсер для языка программирования, его также необходимо будет постоянно поддерживать и обновлять вместе с языком,поэтому для получения AST в *astminer* используются готовые синтаксические анализаторы. Далее приводится обзор некоторых из них.

3.2.1. ANTLR

ANTLR (ANother Tool for Language Recognition — ещё одно средство распознавания языков) [9] — инструмент, позволяющий преобразовывать грамматики в парсеры на Java, Python и другие языки. Парсеры, полученные с помощью ANTLR токенизируют текст, а затем строят абстрактное синтаксическое дерево используя правила из грамматики (рис. 3).

Рис. 3



Использование ANTLR позволяет без особых проблем создавать новые синтаксические анализаторы из готовых грамматик, поэтому добавлять новые языки очень легко (ANTLR не привязан к конкретному языку). Однако широкие возможности ANTLR ведут к нестандартизированым грамматикам (токены в разных грамматиках могут выделяться по разному, по разному могут называться типы вершин) или к грамматикам, которые попросту плохо работают. Также деревья, полученные с помощью ANTLR стоит сжать: после парсинга могут получиться длинные ”бамбуковые” ветви, где у каждой вершины один потомок. Такие ветви занимают много места в памяти и могут быть сжаты до одной вершины.

3.2.2. GumTree

GumTree [10] —полноценный фреймворк для работы с исходным кодом в виде деревьев и вычисления различий между ними. Работа GumTree основана на сравнении двух AST: алгоритм рассчитывает какие базовые операции по модификации одного дерева надо применить, чтобы получить второе.

Полученные с помощью GumTree AST лучше стандартизированы (например для любого языка вершина с обьявлением функции будет называться `methodDeclaration`, обьявление класса — `classDeclaration` и т.д.), они эффективно хранятся и поэтому с ними проще работать (по сравнению с работой с ANTLR). Однако используя GumTree мы ограниченны с выбором языка и конкретной грамматики, поэтому если какая-то грамматика работает некорректно, то исправить это будет невозможно. Например, на момент написания, грамматика GumTree для Python полностью игнорирует указание возвращаемого типа после сигнатуры функции, когда как ANTLR может это находить.

3.2.3. TreeSitter

Данный парсер еще не был добавлен в *astminer*, однако результаты этой работы могут поспособствовать его добавлению.

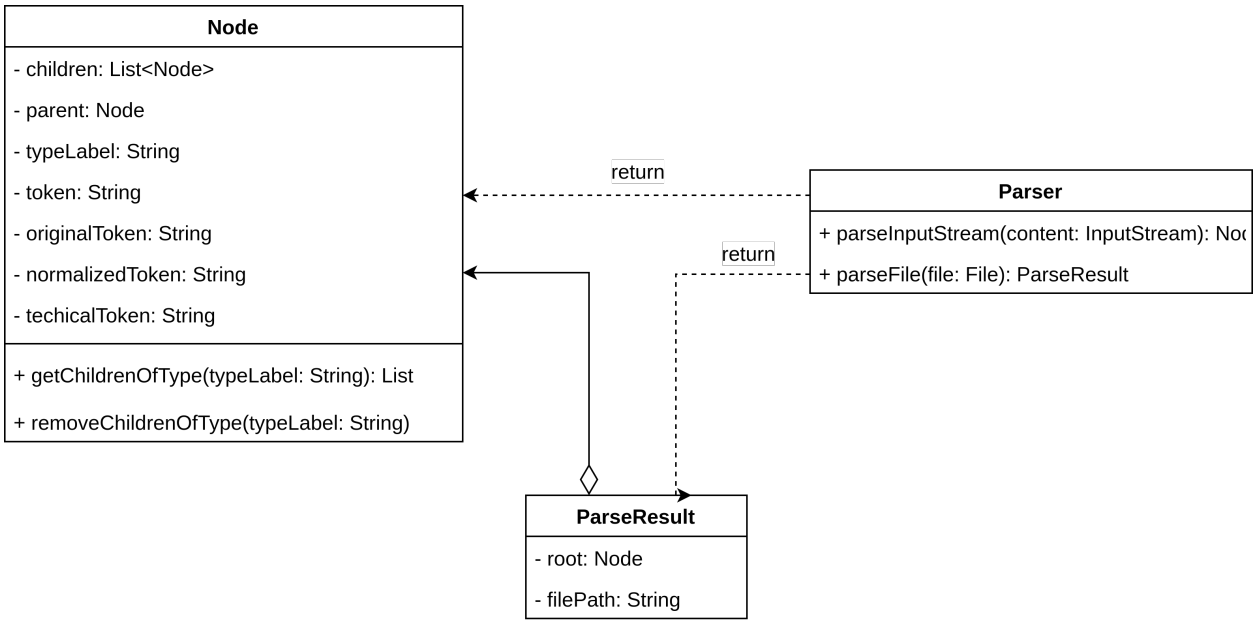
TreeSitter [11] — это инструмент генератора парсеров и библиотека инкрементного синтаксического анализа. Он может построить конкретное синтаксическое дерево для исходного файла и эффективно обновлять синтаксическое дерево по мере редактирования исходного файла.

TreeSitter быстрее (по сравнению с используемыми сейчас парсерами) и позволяет как и ANTLR строить деревья из готовых грамматик. Используя в комбинации с ANTLR и GumTree, он позволит расширить выбор среди граммматик для парсинга.

3.3. Обзор архитектуры парсинга

В данной секции рассматривается существующая архитектура компонентов, использующихся в *astminer* для майнинга кода.

Рис. 4: Отношения интерфейсов `Node, ParseResult, Parser`



AST хранится в виде рекурсивной структуры Node (рис. 4). В любом экземпляре класса хранятся ссылки на родителя и потомков.

Каждая вершина имеет некоторый тип, например `single variable` или `method declaration`, который определяет какому правилу грамматики соответствует данная вершина. Также реализованы методы для поиска и удаления вершин определенного типа. Все это позволяет видоизменять деревья или находить определенные вершины (например, все параметры функции).

Наконец, каждая вершина имеет несколько токенов. Original token — токен, полученный непосредственно из исходного кода программы (если вершина это оператор, то значение будет пустым). Normalized token — получается применением алгоритма нормализации к оригинальному (алгоритм нормализации включает в себя приведение к нижнему регистру, разделение на слова и т.д.). Technical token вершина получает, когда мы хотим ”спрятать” упомянутые выше токены, обрабатывая дерево для определенной задачи машинного обучения (например для

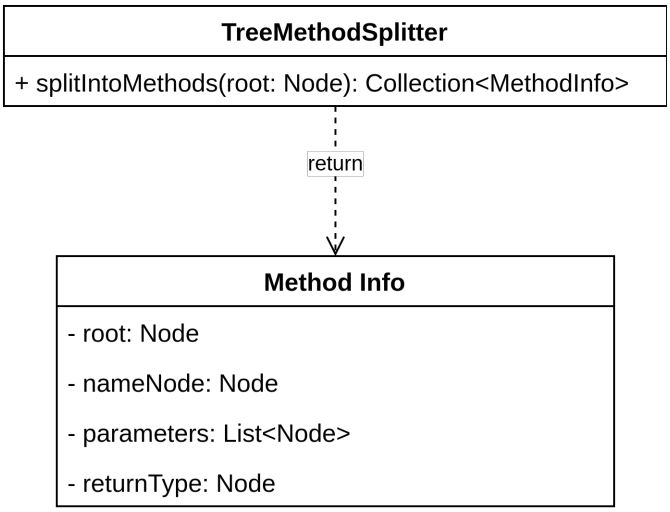
проблемы генерации имен функции необходимо спрятать это имя, тогда любой рекурсивный вызов будет заменен на `SELF`). Ранее эти токены записывались в поле `metadata`, однако во время одного из рефакторингов было решено выделить их в отдельные поля.

Любой парсер использующийся в *astminer* должен наследоваться от интерфейса `Parser` (рис. 4) и реализовывать метод `parseInputStream`.

Результатом парсинга является экземпляр класса `ParseResult`, где будет хранится корень дерева исходного кода, хранящийся в файле, и путь до файла. Путь хранится для того чтобы затем легко можно было восстановить структуру проекта после парсинга.

Также для некоторых сценариев использования, может понадобиться разбить исходное дерево на методы. Тогда для каждого парсера и языка существует свой `TreeMethodSplitter` (рис. 5), который проходит по дереву, разбивает его на методы и вытаскивает некоторые полезные для дальнейшей работы вершины: название метода, параметры и возвращаемый тип.

Рис. 5: `Tree splitting model`



Теперь можно заметить, что добавление любого языка или парсера сопровождается созданием целого ряда объектов. Эти объекты сильно переплетаются с основной архитектурой *astminer*, делая процесс добавления новых все труднее и труднее. Переработав взаимосвязь этих компонентов, добавив новые абстракции и обьекты, можно создать единый

API, который позволит добавлять новые языки и парсеры без изменения остальной архитектуры.

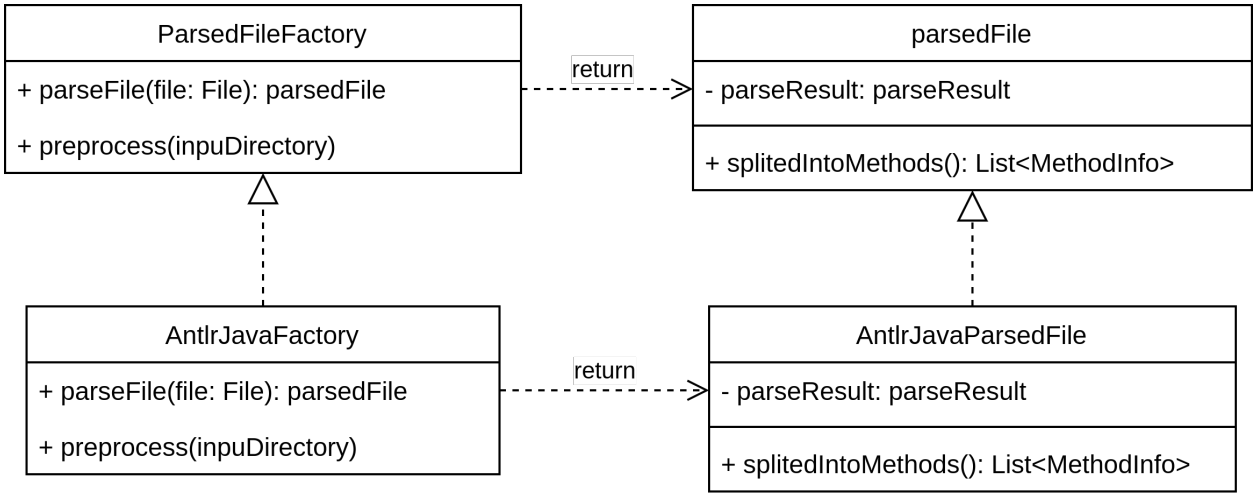
4. Реализация

4.1. Проектирование

Одной из основных причин переплетения главной логики приложения с логикой разбора исходного кода является слабая обособленность части приложения, отвечающей за парсинг. Парсеры и сплиттеры не обьединены между собой и за их инициализацию, использование и корректный выбор отвечает код, который непосредственно вызывает приложение. Это создает нагромождение разных обьектов и сложность в масштабировании.

Сделать архитектуру понятнее можно с помощью паттерна абстрактная фабрика [12].

Рис. 6: Parsed file model



Оставим текущую архитектуру парсинга и разбиения на методы, однако обернем результаты их работы в класс `ParsedFile`, за создание которых будут отвечать `ParsedFileFactory` (рис. 6).

Каждый экземпляр класса `ParsedFile` будет самостоятельно парсить исходный код одного файла в AST и хранить для дальнейшей работы. Также он будет иметь возможность разбить это дерево на несколько поддеревьев в зависимости от конкретной грамматики, так как любой парсер и язык создает своего наследника класса `parsedFile`. Таким образом, вся логика, связанная с применением грамматики к конкрет-

ному файлу и его разбиение на поддеревья, может находится в одном классе. Затем, разные части программы могут получать необходимую им интерпретацию AST.

Наследники `ParsedFileFactory` будут отвечать как за создание экземпляров `ParsedFile`, так и подготовку исходного кода к парсингу (препроцессинг), так как тоже знают какую именно грамматику они будут применять. Например обработка макросов в C и C++ может быть реализованая через метод фабрики `preprocess`.

Это позволит локализовать всю работу с конкретными языками и парсерами в отдельные блоки, когда как остальному коду приложения всего лишь надо будет создать фабрику соответствующую выбранному парсеру и языку.

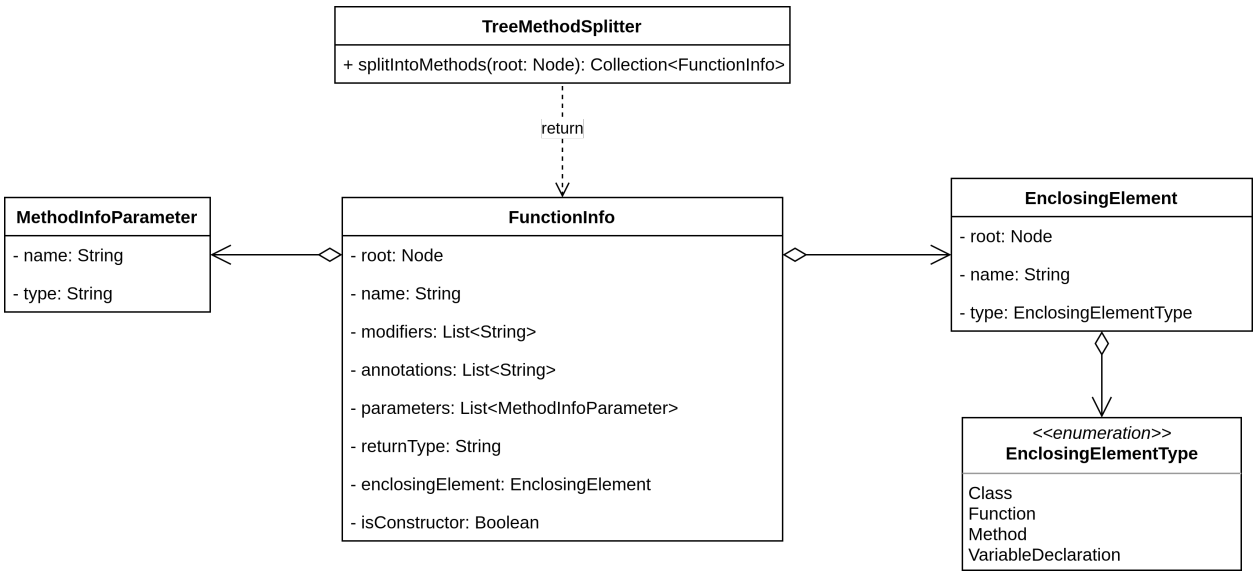
Также, одной из специфичных для грамматики частей приложения является разбиение дерева на методы, которая на самом деле также отвечает за разбиение на функции, поэтому далее будет упоминаться разбиение на функции.

Проблема с разбиением на функции и последующим использованием обьектов `MethodInfo` заключается в том, что они хранят как корень функции, так и информацию о ней (её название, параметры, возвращаемый тип и др.) в виде вершин. Это сильно усложняет дальнейшее использование, так как чтобы получить более точную информацию о функции, необходимо повторно обработать эти вершины, а обработка будет зависеть от конкретной грамматики.

Пусть в `Function info` в виде вершины хранится только корень поддерева AST, который является обьявлением функции, а вся остальная информация будет хранится в виде, независимой от конкретной грамматики. Тогда для любого языка и парсера будут заполнятся одни и те же поля и остальной код приложения не будет знать как именно их надо получать.

Здесь также необходимо предусмотреть разницу между отсутствием некоторой информации и невозможностью ее получить, используя текущие инструменты. В первом случае необходимо, чтобы соответствующее поле имело значение `null`, а во втором, чтобы при обращении к

Рис. 7: Function info model



соответствующему полю кидалось исключение. Для этого можно воспользоваться возможностями языка Kotlin, который позволяет задать методы get в интерфейсе (Листинг 3). Затем мы можем переопределить некоторые поля и присвоить им соответствующие значения.

Listing 3: Интерфейс Function info

```
interface FunctionInfo<T : Node> {
    val nameNode: T?
        get() = notImplemented(''nameNode'')
    val name: String?
        get() = nameNode?.token
    val root: T
        get() = notImplemented(''root'')
    ...
    val isConstructor: Boolean
        get() = notImplemented(''isConstructor'')
}
```

4.2. Интегрирование

После проектирования решения, для его внедрения необходимо изменить работу всех затронутых компонентов.

Для начала было замечено, что в сплиттерах используется большое количество кастов. Происходит это из-за плохой реализации функции `preOrder`, которая выдает вершины поддерева в виде списка в определенном порядке. Полностью переработав как работают обходы деревьев, мы можем избавиться от кастов как в сплиттерах, так и в многих других частях проекта.

После проведения рефакторинга, создадим наследников FunctionInfo для каждого языка и парсера, затем перенесем туда логику, использующуюся в сплиттерах для нахождения существенных вершин (вершин параметров, имени функции и т.д.). После этого для каждой грамматики разработаем и добавим алгоритм, позволяющий собирать информацию из вершин и выдавать в релевантном нам виде. При этом стоит заметить, что большая часть кода в сплиттерах повторяется. Чтобы решить эту проблему, реализуем небольшой API, который позволит лучше ориентироваться в дереве, а затем переработаем код в более читаемый вид.

Затем создадим фабрики для каждого языка и парсера и функцию `getParsedFileFactory`, позволяющую получить необходимую фабрику.

5. Тестирование

В качестве апробации спроектированного и внедренного решения было решено добавить поддержку нового разбираемого языка в инструмент. Такой эксперимент позволит проверить насколько полученное решение упрощает добавление новых языков и парсеров. Для добавления был выбран язык PHP и парсер на граммматике ANTLR, так как для PHP уже существует готовая грамматика, а также уже давно существовал запрос[2] на его необходимость в *astminer*.

Текущая реализация парсинга требовала создание таких классов, как `PHPParser` [13], `PHPFunctionSplitter` [14] и `PHPFunctionInfo` [15] для парсинга исходного кода, а также класс `PHPParsedFile` и объект `PHPParsedFileFactory` для работы с вышеуказанными компонентами, что на 3 сущности больше чем в предыдущей реализации (предыдущая реализация требовала только `PHPParser` и `PHPFunctionSplitter`).

Для полноценной интеграции новых компонентов в старой реализации необходимо было добавить создание экземпляров классов `PHPParser` и `PHPFunctionSplitter` в файлы [16] и [17], а также добавить разбор всех необходимых вершин во всех существующих фильтрах функций [18]. Используя текущую реализацию, достаточно добавить ссылку на фабрику в функции `getAntlrParsedFileFactory` [19]. При дальнейшей работе с PHP, для добавления новых функций по разбору языка (например разбиение на классы или фильтрацию функций по возвращаемому типу) в предыдущей архитектуре понадобилось бы повторно вносить изменения во все вышеуказанные файлы, когда как все изменения в текущей архитектуре проводятся на момент добавления нового языка.

Таким образом, текущая реализация требует создания большего количества объектов, однако дальнейшее внедрение в архитектуру и использование стало значительно проще.

[2] https://github.com/JetBrains-Research/astminer/issues/81

6. Заключение

В ходе данной работы были достигнуты следующие задачи:

1. Были изучена предметная область и архитектура приложения *astminer*
2. Спроектирован единый API для добавления новых языков и парсеров
3. Решение было интегрировано и сопровождено тестами
4. В качестве апробации была добавлена поддержка языка PHP

Полученное решение, а также компоненты для поддержки языка PHP уже добавлены в основную ветку в оффициальном репозитории *astminer* [20] на Github. Дальнейшими задачами является добавление новых парсеров и языков, а также добавление алгоритмов получения большей информации о функциях (например алгоритмы для получения документации и аннотаций).

Список литературы

[1] Saad Shafiq, Atif Mashkoor, Christoph Mayr-Dorn, and Alexander Egyed. Machine learning for software engineering: A systematic mapping, 2020.

[2] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code, 2018.

[3] Nakarin Maneerat and Pomsiri Muenchaisri. Bad-smell prediction from software design model using machine learning techniques. In *2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 331–336, 2011.

[4] Jaswitha Abbineni and Ooha Thalluri. Software defect detection using machine learning techniques. In *2018 2nd International Conference on Trends in Electronics and Informatics (ICOEI)*, pages 471–475, 2018.

[5] Michael D. Ernst. Natural language is a programming language: Applying natural language processing to software development. In *SNAPL 2017: the 2nd Summit oN Advances in Programming Languages*, pages 4:1–4:14, Asilomar, CA, USA, May 2017.

[6] Zhensu Sun, Yan Liu, Chen Yang, and Yu Qian. Pscs: A path-based neural model for semantic code search, 2020.

[7] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs, 2018.

[8] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. Pathminer: a library for mining of path-based representations of code. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 13–17. IEEE Press, 2019.

[9] Antlr official web page. https://www.antlr.org/.

[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 313–324, 2014.

[11] Tree sitter github page. https://github.com/tree-sitter/tree-sitter.

[12] Описание паттерна абстрактная фабрика. https://refactoring.guru/ru/design-patterns/abstract-factory.

[13] Файл phpparser.kt. https://github.com/JetBrains-Research/astminer/blob/c1e35b0a3d4d4ee4f6f13ac3d52a876313ce2cf7/src/main/kotlin/astminer/parse/antlr/php/PHPParser.kt. Accessed: 05-06-2021.

[14] Файл phpfunctionsplitter.kt. https://github.com/JetBrains-Research/astminer/blob/c1e35b0a3d4d4ee4f6f13ac3d52a876313ce2cf7/src/main/kotlin/astminer/parse/antlr/php/PHPFunctionSplitter.kt. Accessed: 05-06-2021.

[15] Файл antlrphpfunctioninfo.kt. https://github.com/JetBrains-Research/astminer/blob/c1e35b0a3d4d4ee4f6f13ac3d52a876313ce2cf7/src/main/kotlin/astminer/parse/antlr/php/ANTLRPHPFunctionInfo.kt. Accessed: 05-06-2021.

[16] Файл labelextractors.kt. https://github.com/JetBrains-Research/astminer/blob/24f86628efbdb437f4e17d3542598095a74b2829/src/main/kotlin/astminer/cli/LabelExtractors.kt. Accessed: 05-06-2021.

[17] Файл utils.kt. https://github.com/JetBrains-Research/astminer/blob/24f86628efbdb437f4e17d3542598095a74b2829/src/main/kotlin/astminer/cli/utils.kt. Accessed: 05-06-2021.

[18] Файл filterpredicates.kt. https://github.com/JetBrains-Research/astminer/blob/24f86628efbdb437f4e17d3542598095a74b2829/src/main/kotlin/astminer/cli/FilterPredicates.kt. Accessed: 05-06-2021.

[19] Файл factory.kt. https://github.com/JetBrains-Research/astminer/blob/c1e35b0a3d4d4ee4f6f13ac3d52a876313ce2cf7/src/main/kotlin/astminer/parse/factory.kt. Accessed: 05-06-2021.

[20] Astminer github page. https://github.com/JetBrains-Research/astminer.