Автономная некоммерческая организация высшего образования
«Университет Иннополис»

АННОТАЦИЯ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
(БАКАЛАВРСКУЮ РАБОТУ)
ПО НАПРАВЛЕНИЮ ПОДГОТОВКИ
09.03.01 – «ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»

НАПРАВЛЕННОСТЬ (ПРОФИЛЬ) ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ
«ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»

Тема	**Проецирование Java в φ-исчисление и его реализацию — EOLANG /** **Mapping Java to φ-calculus and its implementation — EOLANG**

Выполнил	**Степанов Максим Александрович**	подпись

Иннополис, Innopolis, 2022

Оглавление

Список таблиц

Список иллюстраций

Аннотация

Данный диплом описывает способ проецирования традиционного объектно-ориентированного языка программирования (в частности — Java) в φ-исчисление (конкретнее — в его реализацию — EOLANG). EOLANG накладывает определенные дополнительные ограничения поверх φ-исчисления, следовательно, транслятор, который разрабатывается как часть этого диплома, делает больший упор на реализацию проекта, чем на теорию проецирования.

Глава 1

Введение

Этот проект направлен на предоставление инструмента транспиляции исходного кода Java (*исходный язык*) в EOLANG (*целевой язык*) для передачи результата в статический анализатор Polystat [1] и его дальнейшего анализа.

Транспилятор (англ. transpiler), также часто называемый транслятором (англ. translator), представляет собой программный инструмент, который преобразует исходный код одного языка в исходный код другого языка. Транспиляция (англ. transpilation) или трансляция (англ. translation) — это операция преобразования исходного кода с помощью такого инструмента.

EOLANG [2] — универсальное промежуточное представление семантики объектно-ориентированного языка, активно разрабатываемое в рамках проекта Polystat. Академическое сообщество еще не полностью изучило EOLANG и его формальную основу — φ-исчисление, что придает диссертации **новизну** и является основной причиной его выбора в качестве целевого языка для проекта.

Согласно индексу TIOBE [3], Java является третьим по популярности языком программирования. Индекс TIOBE — авторитетный показатель в об-

ласти разработки программного обеспечения. Популярность языка придает **актуальность** проекту, что является основной причиной его выбора в качестве исходного языка для проекта.

Статический семантический анализ [4], [5] является одним из популярных подходов к минимизации количества ошибок в программном проекте, и область статического анализа активно исследуется в настоящее время.

Общая цель Polystat — предоставить инструменты для перевода различных языков программирования в EOLANG и инструмент для выполнения обобщенного статического анализа промежуточного представления EOLANG. Такой подход позволяет реализовать единый универсальный инструмент анализа, охватывающий несколько объектно-ориентированных языков, сводя к минимуму усилия команды разработчиков анализатора Polystat, реализуя алгоритм анализа только один раз. Этот проект, J2EO, относится к первому типу инструментов — переводу Java в EOLANG. Структура всего проекта показана в (Fig. 1.1).

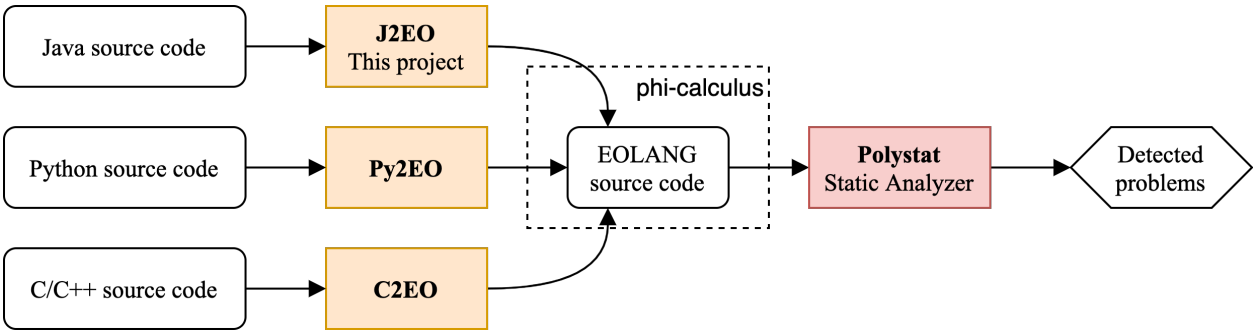

Рис. 1.1: Общая структура проекта Polystat

1.1 Обзор содержания тезиса

В главе «Обзор литературы» будет изложена общая информация о трансляторах языков программирования, работе, проделанной в этой обла-

сти, целях таких проектов, часто встречающихся проблемах и их решениях. Он также охватывает некоторые функции EOLANG и связанные с ними проблемы.

В главе «Реализация» будет подробно представлена методология проектирования, а также описаны детали реализации проекта.

В главе «Результаты и обсуждение» будут представлены результаты реализованного проекта и обсуждены будущие возможности и актуальность проекта.

Глава 2

Реализация

В этой главе рассматриваются теоретические аспекты, связанные с переводом Java на EO, разработанные прогнозы и описание программного проекта, разработанного вместе с этим тезисом.

2.1 Концептуальное сравнение Java и EO

Чтобы приступить к переводу одного языка на другой, важно понимать их отличия. Даже один и тот же термин может по-разному интерпретироваться разработчиками нескольких языков.

2.1.1 Парадигма

Важно помнить, что в прошлом различные языки, включая Java, искажали исходное представление об объектно-ориентированной парадигме. Таким образом, объектно-ориентированная парадигма по-разному интерпретируется разработчиками разных языков.

Java реализует традиционную в настоящее время парадигму ООП. Он

строго вынуждает разработчиков помещать все в классы или интерфейсы, которые являются единственными доступными конструкциями верхнего уровня.

EO ближе к функциональной парадигме с упором на объекты. Он также претендует на то, чтобы быть языком ООП, хотя EO подразумевает исходное значение этого термина.

2.1.2 Видимость

Java имеет строгую систему видимости, включающую четыре уровня, описанные в подразделе **??**. EO, с другой стороны, не имеет система видимости вообще. Такая система видимости невозможна в EO из-за его динамической природы, включая динамическую типизацию, отсутствие спецификации структуры объектов и неограниченный доступ к созданию и модификации объектов. Эти функции необходимы для того, чтобы структура языка оставалась достаточно гибкой для перевода на нее различных языков.

2.1.3 Изменчивость

В Java все переменные по умолчанию изменяемы (англ. mutable) (хотя это может быть ограничено с помощью final). В EO нет встроенной поддержки изменяемости (англ. immutable), но есть обходные пути в стандартной библиотеке: `memory` и `cage`.

2.1.4 Стиль программирования

В языке Java используется императивный стиль: разработчик пишет код, который говорит *как* выполнить программу. Декларативный стиль может

быть достигнут с помощью специально реализованной пары исполнителя и структуры данных, но он не будет работать с библиотеками (как стандартной библиотекой, так и внешними библиотеками). Для поддержки библиотек необходимы пользовательские обертки.

EO имеет декларативный стиль — разработчик пишет *чего он хочет добиться*, а не как этого добиться. Императивный стиль может быть достигнут с помощью встроенного атома с именем `seq`, но из-за особенностей реализации область `seq` более ограничена, чем область действия объекта, например, невозможно объявить переменные внутри.

2.2 Проекции

В этом разделе будут рассмотрены теоретические подходы к проектированию исходного кода Java в семантически эквивалентный исходный код EO.

2.2.1 Именование

Java и EO имеют разные соглашения по именованию языковых компонентов, и если Java не обязывает использовать соглашения об именовании, EO это делает. Следовательно, отображение имен должно быть разработано и согласовано во всех трансляторах, чтобы результирующий код работал.

В частности: имена объектов и атрибутов в EO могут начинаться только с букв нижнего регистра и продолжаться буквами верхнего/нижнего регистра, цифрами, дефисами или символами подчеркивания. Это отличается от Java только первым символом, поэтому решение состоит в том, чтобы добавлять перед именами фиксированную текстовую последовательность.

Кроме того, у EO есть проблемы с затенением атрибутов. Затенение атрибутов — это функция языка, которая позволяет переопределять переменные (атрибуты в терминах EO) во вложенных областях. Эта функция работает непредсказуемым образом, особенно учитывая нестабильную природу EO (она все еще находится на ранней стадии разработки, и поведение часто меняется на момент написания этой статьи). Поэтому транслятор максимально избегает дублирования имен. Первым шагом к минимизации таких дубликатов является добавление разных типов токенов исходного кода с разными префиксами.

В качестве решения проблемы затенения атрибутов были разработаны следующие сопоставления:

- Имена классов начинаются с `class__`
- Переменные начинаются с `var__`
- Аргументы начинаются с `arg__`
- Методы начинаются с `method__`

2.2.2 Конструкторы

Java, как и многие современные языки ООП, имеет конструкторы классов. В EO нет классов, поэтому реализация конструкторов находится под полным контролем транслятора. В текущей форме конструкторы Java отображаются в атрибуты функций с дополнительным именем `cons`.

2.2.3 Перегрузка

Кроме того, EO не поддерживает перегрузку методов/конструкторов и даже не имеет функций напрямую. Это простые объекты, как и все в языке. Поэтому к именам методов/конструкторов добавляются имена типов, разделенные __. Информация о дженериках опущена, как это делает и JVM.

Пример результирующего сопоставления с учетом всего вышеперечисленного:

```
class A {
  A(int member) {...}
  void functionName(int arg1, Option<Value> arg2)
    ↪ {...}
}
```

```
[] > class__A
  [] > new
    [] > this
      [var__member] > cons__int
        ...

      [] > method__functionName__int__Option
        ...
```

2.2.4 Выражения

Java предоставляет способ создания сложных вложенных и связанных (разделенных точкой) выражений. Эти выражения могут вызывать различные побочные эффекты в разном порядке.

Связанные выражения

Сложный случай — цепные выражения. Из-за особого подхода к вызову функций в EO вызов метода экземпляра класса требует передачи этого экземпляра в качестве первого аргумента, то есть `obj.function(obj)`. Это легко сделать, когда используются операторы с одним выражением, но цепочка с точкой требует где-то хранить промежуточный объект, чтобы можно было продолжить цепочку.

Рассмотрим следующий пример на Java:

```
13 var result = obj.setValue(value).computeResult();
```

```
14 ... > obj
15 ... > value
16 ...
17 [obj] > <uuid_1>
18   seq > @
19     obj.setValue obj value
20 [obj] > <uuid_2>
21   seq > @
22     obj.computeResult obj
23 ...
24 seq > @
25   ...
26   <uuid_2>
27     <uuid_1>
28       obj
```

Хотя вариант EO требует значительно больше кода для выполнения той же операции, язык никогда не предназначался для написания кода вручную, а только для перевода с других языков.

Побочные эффекты в выражениях

Рассмотрим следующий фрагмент Java:

```
29 var out =
    ↪  obj.setValue(obj2.setValue(obj2.getCount()).i++);
```

Он отличается от предыдущего фрагмента одной особенностью — операцией постинкремента, которая является побочным эффектом мутации на месте.

В этом коде в заданном порядке выполняются следующие операции:

- Вызывается функция `getCount` из `obj2` и получается результат
- Функция `setValue` объекта `obj2` вызывается с полученным значением
- `obj2` возвращает себя для дальнейшего связанного вызова
- Получено свойство `i` объекта `obj2`
- Свойство `i` у `obj2` увеличивается на 1
- Функция `setValue` из `obj` вызывается со значением, полученным до увеличения, так как это тип значения, а не ссылочный тип
- Результат `setValue` присваивается `out`

Как видно из списка выше, порядок выполнения не соответствует абстрактному синтаксическому дереву, в частности — операция постинкремента. Эту операцию нужно было ставить перед получением результата, как в

преинкременте, чтобы соответствовать структуре AST. Этот факт измененного порядка затрудняет перевод Java-кода в EO, который не имеет соответствующего синтаксического сахара, а также переменных с типизированным значением.

Общая проблема заключается в том, что выражение вызывает побочные эффекты: EO не поощряет любые операции с побочными эффектами и предоставляет только «серые» обходные атомы для имитации побочных эффектов, для создания уровня совместимости с программами, основанными на побочных эффектах. Но поскольку типов-значений по-прежнему нет, нам приходится обходить операции, которые полагаются на них, с помощью специальной логики (т. е. создавать новую переменную).

Учитывая методы, описанные выше, мы можем получить следующее отображение EO:

```
... > obj
... > obj2
...
[obj] > <uuid_1>
  memory > result
  seq > @
    result.write
      obj.getCount obj
    result
[obj arg] > <uuid_2>
  seq > @
    obj.setValue obj arg
[obj] > <uuid_3>
  memory > result
  seq > @
    result.write
      obj.i
    obj.i.write
```

```
48        add
49          obj.i
50          1
51      result
52  [obj arg] > <uuid_4>
53    seq > @
54      obj.setValue
55        obj
56        arg
57  ...
58  seq > @
59    ...
60    <uuid_4>
61      obj
62      <uuid_3>
63        <uuid_2>
64          obj2
65          <uuid_1>
66            obj2
67    ...
```

Обратите внимание, как реализован постинкремент в `<uuid_3>`: создается новый объект `memory` (представьте его как ячейку, которая может хранить какое-то значение), туда копируется текущее значение, а затем значение исходного объекта обновляется.

Тот же подход используется везде, где используются типы значений (для Java список включает `int`, `long`, `short`, `char`, `float`, `double`). В данном случае он используется в `<uuid_1>`. Это обеспечивает надежный способ гарантировать, что исходное значение не будет изменено последующими вызовами функций, поскольку оно не изменяется в Java.

2.2.5 Разрешение составных имен в Java

В Java есть очень неприятная особенность со ссылками на вложенные классы/члены: каждый токен в любом случае отделяется точкой. Поэтому невозможно сказать, какая именно операция представлена конкретной точкой: является ли она частью имени пакета, ссылкой на класс из пакета или ссылкой на член класса.

Для определения работы составного имени анализируется блок импорта: если строка из блока импорта Java (исключая последнюю часть, либо `*`, либо имя класса) является началом составного имени, части пакета не изменяются в составном имени EO; файл, на который ссылается имя пакета, загружается, анализируется и анализируется. Части, которые соответствуют именам классов, начинаются с `class__`, части, которые соответствуют именам переменных, начинаются с `var__`.

2.2.6 Отображение классов

Тела классов достаточно просто перевести.

- Класс – это объект со всеми статическими членами и статическими методами, а также фабрикой объектов `new` и конструкторами `cons<n>`. Он также украшает суперкласс, наследуя его статические члены и методы.

- Метод `new` украшает объект суперкласса `new` и определяет члены класса с их значениями по умолчанию.

- `cons<n>`, содержащий соответствующий код конструктора Java.

Что представляет собой проблему, так это наследование, особенно множественное наследование (реализованное через интерфейсы в Java). Наблюдаемых подходов к интерфейсам карт в проекте пока нет, поэтому пока их пропустим. Основное внимание будет уделено урокам картирования.

Суперкласс записывается в атрибут `super` экземпляра при его создании.

Ниже приведен пример сопоставления классов:

Оригинальный фрагмент кода Java:

```
68 class A {
69   int i = 42;
70   int getI() { return i; }
71 }
72 class B extends A {
73   int i = 1;
74   B() { super(); i = 2; }
75   int getI() { return i; }
76   int getSuperI() { return super.i; }
77   int setSuperI(int i) { super.i = i; }
78 }
```

Производится семантически эквивалентный код EO:

```
79 [] > class__A
80 [] > new
81   [] > this
82     memory > i
83     [this] > getI
84       this.i > @
85   seq > @
86     this.i.write 42
87     this
88 [this] > cons1
89   this > @
```

```
90
91  [] > class__B
92  [] > new
93    [] > this
94      class__A.new > super
95      super > @
96      memory > i
97      [this] > getI
98        this.i > @
99      [this] > getSuperI
100       this.super.i > @
101     [this i] > setSuperI
102       seq > @
103         this.super.i.write i
104         this
105   seq > @
106     this.i.write 1
107     this
108 [this] > cons1
109   class__A.cons1 this > this2
110   seq > @
111     this2.i.write 2
112     this2
```

2.2.7 Отображение интерфейсов

Особенностью EO, которая позволяет расширять другие объекты, является декорация. Однако декорация позволяет моделировать только одиночное наследование. Java 8 привнесла в интерфейсы методы по умолчанию, что позволило реализовать множественное наследование. Это не вписывается в φ-исчисление, поэтому его можно только обойти, а не реализовать.

Текущее решение просто игнорирует интерфейсы, поскольку в основном они используются для облегчения статической проверки когда необхо-

димо скрыть данные. Однако этот подход некорректно работает с методами по умолчанию.

2.3 Обзор реализации J2EO

В дополнение к теоретическим отображениям, продуктом диссертации является программный проект под названием J2EO. Проект написан на Java/Kotlin, а парсер использует ANTLR [6].

Обработка одного файла Java через разные модули показана ниже.

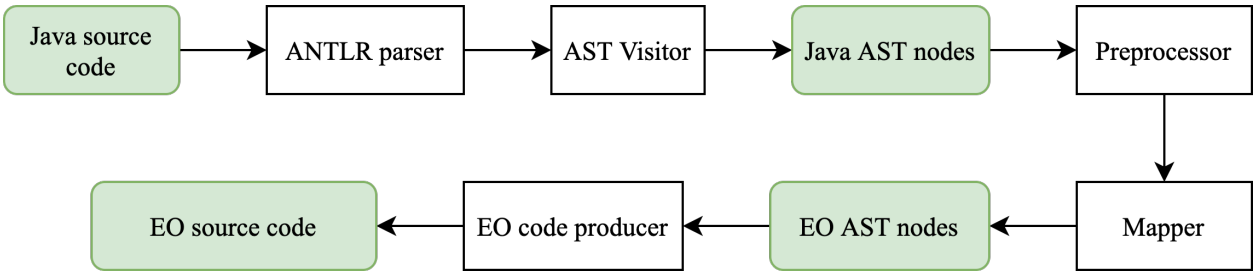

Рис. 2.1: Обработка одного файла Java через конвейер J2EO

Обертка, которая обрабатывает ввод-вывод, выполнение конвейера и другие рутинные задачи, не рассматривается в этом тексте.

Следующая часть описывает каждый из шагов более подробно.

2.3.1 Парсер ANTLR

Парсеров Java для современных версий языка почти нет в Интернете. Единственный синтаксический анализатор, который охватывает Java 17, реализован в ANTLR и доступен на GitHub [7]. Требуются небольшие модификации, чтобы заставить его работать в случае использования J2EO.

2.3.2 Посетитель AST

Этот этап конвейера запускает синтаксический анализатор ANTLR и посещает каждый узел для создания внутреннего узла Java AST. Этот шаг необходим, поскольку ANTLR сам по себе не генерирует узлы AST, а только предоставляет контекст, который можно использовать для получения частей дерева.

2.3.3 Препроцессор

Препроцессор применяет исправления соглашения об именах к коду. Этот шаг включает префикс классов с `class__`, переименование импорта в соответствии с соглашениями EO и другие семантически-нейтральные изменения.

2.3.4 Проектор

Это основной шаг конвейера, который создает узлы EO AST из предварительно обработанных узлов Java AST. Методология проецирования узлов AST представлена в предыдущем разделе.

2.3.5 Принтер кода EO

Последним этапом конвейера является создание исходного кода из EO AST. Эта часть реализована в виде функций расширения Kotlin, которые добавляют методы к узлам EO AST неинвазивным способом.

2.3.6 Подведение итогов

Все описанные выше этапы пайплайна реализованы в функциональном стиле. Таким образом, ни одна из функций не использует общее изменяемое состояние, что позволяет распараллелить перевод файлов. Параллельная версия J2EO успешно протестирована.

Проект J2EO имеет открытый исходный код и доступен на GitHub. Любой может использовать его и внести свой вклад в него.

Глава 3

Результаты и обсуждение

3.1 Результаты

Результатом этой диссертации является часть программного обеспечения J2EO, написанного на Java/Kotlin. Реализованная часть включает в себя абстрактное синтаксическое дерево EO (AST), алгоритм, который печатает EO AST в файл с исходным кодом, и алгоритм, который переводит выбранные узлы Java AST в EO AST.

Основным вариантом использования J2EO является преобразование реальных проектов Java в EO для последующего выполнения статического анализа с другими проектами Polystat. Таким образом, важно без сбоев переводить большие проекты и максимизировать количество переведенных функций Java. Создание полного семантически эквивалентного кода не является обязательным требованием для статического анализа.

В качестве эталона для оценки результатов я выбрал несколько крупных проектов, активно используемых в промышленных системах.

Таблица использованных проектов с соответствующими им фактиче-

скими хэшами коммитов указана в таблице I:

Project	Java version	LoC	Actual commit hash
Hadoop [8]	8	1631465	ec0ff1dc04b2ced199d7-1543a8260e9225d9e014
Kafka [9]	8	499373	f36de0744b915335de6b-636e6bd6b5f1276f34f6
J2EO [10]	17	41200	a762a903eb55f3e11403-d4630654f4c89397d75a

Таблица I: Java projects used for benchmarking

Используемая версия J2EO — 0.5.3, она присутствует в репозитории GitHub [10] проекта.

Метрика Lines of Code (LoC) была рассчитана с помощью утилиты `cloc`[11] и включает только физические строки Java, исключая пустые строки и комментарии.

Данные версии программного обеспечения и хэши коммитов позволяют читателям воспроизводить результаты, приведенные в тексте, на своей машине.

3.1.1 Расчетные проекции

В рамках этой диссертации разработаны теоретические прогнозы для многих конструкций и функций Java. Полный список представлен в главе 2.

3.1.2 Реализованные прогнозы

На момент написания проекции, реализованные в J2EO, охватывают отображение классов, их статических и нестатических членов, статических и нестатических методов, поддержку большинства операторов, которые имеет смысл статически анализировать. Любая структура проекта анализируется правильно, поэтому в качестве входных данных могут быть переданы как проекты Maven, так и проекты Gradle с произвольной структурой каталогов.

Приоритет реализации проекций активно обсуждался с командой разработчиков анализатора, поэтому включенные отображения актуальны для дальнейшего развития всего суперпроекта.

Поддерживаемая версия Java — 17, и, учитывая обратную совместимость, поддерживаются любые проекты более ранних версий. Несколько записей в таблицах бенчмаркинга подтверждают это.

3.1.3 Оценка результатов

На момент написания статьи объективная оценка результатов невозможна, так как анализатор Polystat находится на ранних этапах разработки и, следовательно, не существует полного пайплайна анализа. Основным используемым методом оценки является перевод больших проектов Java и проверка правильности созданных файлов EO вручную. Проекты из таблицы бенчмаркинга предоставляют кодовую базу из нескольких миллионов строк кода на Java и факт того, что транслятор успешно их обрабатывает звучит многообещающе для будущего этого проекта.

3.2 Обсуждение и заключение

В тексте представлен обзор используемых технологий, включая Java, EOLANG и φ-исчисление, затем дана оценка этих технологий для целей проекта, теоретические проекции конструкций между исходным и целевым языками и, наконец, описаны реализация самого инструмента.

Проект J2EO уже охватывает значительный диапазон функций Java, облегчая дальнейшую разработку анализатора Polystat. Поскольку Полистат сильно зависит от результатов работы транспиляторов, его разработка была приостановлена на длительный период, и поэтому на момент написания статьи не было представлено никаких подробных результатов анализа.

Комбинация J2EO/Polystat все еще находится в активной разработке. Однако, по результатам J2EO, EO обеспечивает необходимую базу для представления других языков. Эта комбинация имеет все шансы стать конкурентоспособной с другими популярными статическими анализаторами Java, такими как PMD, SpotBugs и встроенным статическим анализатором IntelliJ IDEA (который недоступен отдельно).

Текущее исследование проекций Java в EOLANG и инструмент перевода J2EO будут представлены академическому сообществу в готовящейся к выпуску статье позже в этом году.

Использованная библиография

[1] *Polystat repository*, `https://github.com/polystat/polystat`.

[2] *EOLANG repository*, `https://github.com/yegor256/eo`.

[3] *TIOBE programming language popularity index*, `https://www.tiobe.com/tiobe-index/`.

[4] A. G. Bardas и др., «Static code analysis,» *Journal of Information Systems & Operations Management*, т. 4, № 2, с. 99—107, 2010.

[5] C. Epure и A. Iftene, «Semantic analysis of source code in object oriented programming. A case study for C#,» *Romanian Journal of Human-Computer Interaction*, т. 9, № 2, с. 103, 2016.

[6] *ANTLR*, `https://www.antlr.org`.

[7] *ANTLR Java parser*, `https://github.com/antlr/grammars-v4/tree/master/java/java`.

[8] *Apache Hadoop*, `https://github.com/apache/hadoop`.

[9] *Apache Kafka*, `https://github.com/apache/kafka`.

[10] *J2EO — Java to EOLANG Transpiler*, `https://github.com/polystat/j2eo`.

[11] *cloc*, `https://github.com/AlDanial/cloc`.