**УДК 004.4'413**

# АНАЛИЗ КАЧЕСТВА ВОССТАНОВЛЕНИЯ ПОСЛЕ ОШИБОК В СИНТАКСИЧЕСКИХ АНАЛИЗАТОРАХ

**Воробьев Я. С.** (ВШЭ), **Бачище О. И.** (ИТМО)
**Научный руководитель – кандидат физико-математических наук, Григорьев С.В.**
(СПбГУ)

**Введение.** При написании кода разработчики часто допускают синтаксические ошибки. Задача восстановления заключается в том, чтобы определить такие ошибки и, при возможности, предложить пользователю способы их исправления или даже исправить автоматически. Это свойство особенно важно для интегрированных сред разработки (IDE) и средств статического анализа кода.

Большинство синтаксических анализаторов в той или иной степени способны определять ошибки и предлагать способы исправления. Однако, тяжело сформулировать критерий качества восстановления, который позволил бы выбрать инструмент для конкретной задачи.

Среди общих метрик выделяются *потребление оперативной памяти* и *скорость* анализа. Такой критерий нередко вычисляется для синтаксических анализаторов, но преимущественно на файлах без ошибок.

Более специфичный критерий качества восстановления — это количество каскадных ошибок. Такие ошибки отсутствуют в исходном файле, но могут появиться после неудачного локального восстановления. Можно предположить, что с уменьшением общего числа ошибок уменьшаются именно каскадные[1]. Но подобная метрика не всегда корректна: если анализатор не фиксирует определённые классы ошибок, то их общее число может быть занижено. Поэтому для более точного анализа необходимо учитывать не только количество, но и *типы возникающих ошибок*.

Непростой задачей является определение *качества* восстановления или того, насколько точно был восстановлен код в месте ошибки. В натуральных тестовых данных нет “верного способа исправления”. Поэтому в качестве меры используется *редакционное расстояние* — показатель того, насколько восстановленный код близок к исходному.

В рамках данной работы реализован набор метрик для вычисления характеристик восстановления после ошибок для синтаксических анализаторов языка программирования Java8. Подход апробирован на трёх синтаксических анализаторах с использованием пользовательских данных с ошибками из датасета blackbox.

**Основная часть.** Так как синтетические данные могут не в полной мере соответствовать пользовательским ошибкам, в качестве датасета для анализа был выбран набор blackbox, состоящий из пользовательских файлов на Java8 с ошибками.

В работе рассмотрены три синтаксических анализатора. Два из них были сгенерированы с помощью ANTLR — для официальной грамматики Java8 и для грамматики, специфически ускоренной для алгоритма ALL(*)[2], а также синтаксический анализатор из инструмента javac.

Подсчет *качества* восстановления состоит из следующих этапов:

1. Исходный код на Java подвергается токенизации. Полученный список токенов передается анализатору для построения синтаксического дерева (лексические ошибки в данном подходе пока не рассматриваются).

2. Производится обход полученного дерева в глубину с извлечением токенов, включая ошибочные узлы (error nodes) и отсутствующие элементы (missing nodes), которые учитываются как фиктивные токены.

3. Рассчитывается мера сходства между исходным и восстановленным списками токенов, основанная на редакционном расстоянии. Каждый фиктивный токен уменьшает

значение метрики, которое варьируется от 0 (полное несоответствие) до 1 (полное совпадение).

Для анализа *полноты обнаруженных ошибок* используется синтаксический анализатор инструмента javac. Ошибки, обнаруживаемые этим инструментом во-первых, классифицированы и фиксированы в документации. Во-вторых, понятны пользователю, потому что именно их получает разработчик от JVM при обработке некорректного файла.

Таким образом, каждый файл из датасета был размечен типами ошибок по классификации javac. Такой подход позволил выделить классы ошибок, которые хуже обрабатываются анализатором или полностью нарушают процесс синтаксического анализа.

Все синтаксические анализаторы реализованы на java, поэтому замер *скорости* обработки производился стандартными для таких программ методом и определялся как среднее нескольких запусков на разогретой JVM. Для анализа *оперативной памяти* используется подход, предложенный в работе Измайловой[3]: минимально необходимый объем памяти определяется при помощи бинарного поиска по соответствующему параметру JVM.

Во время исследования были выявлены нетривиальные свойства рассматриваемых синтаксический анализаторов, не описанные ранее. Например, ANTLR для оптимизированной версии грамматики в среднем потребляет почти в 11 раз меньше памяти и работает в 55 раз быстрее версии для стандартной грамматики. Количество ошибок, обнаруженных исходной версией меньше на 66%. Однако, анализ качества восстановления позволил определить, что более 10 типов ошибок (по классификации javac) ускоренная версия восстанавливает гораздо хуже или не определяет вообще.

**Выводы.** Предлагаемое решение позволяет оценить качество механизма восстановления ошибок, выделить сильные и слабые места синтаксических анализаторов. Это может быть полезно как при выборе инструмента в областях, где это свойство важно (например, среды разработки), так и при улучшении существующих анализаторов или грамматик для их генерации.

Решение уже используется в компании SRC (Saint Petersburg Research Center), разрабатывающей среды разработки для языка программирования Java. За счет того, что метрики не привязаны к конкретному языку программирования, разработанный инструмент может быть внедрен в большее число сфер. Например, уже есть запрос от SRC на расширение инфраструктуры для анализа анализаторов для языка Python.

**Список использованных источников**:

1. Diekmann L., Tratt L.. Don't Panic! Better, Fewer, Syntax Errors for LR Parsers // 34th European Conference on Object-Oriented Programming (ECOOP 2020). Leibniz International Proceedings in Informatics (LIPIcs), Volume 166, pp. 6:1-6:32, Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2020) https://doi.org/10.4230/LIPIcs.ECOOP.2020.6

2. Terence Parr, Sam Harwell, and Kathleen Fisher. 2014. Adaptive LL(*) parsing: the power of dynamic analysis. SIGPLAN Not. 49, 10 (October 2014), 579–598. https://doi.org/10.1145/2714064.2660202

3. Anastasia Izmaylova, Ali Afroozeh, and Tijs van der Storm. 2016. Practical, general parser combinators. In Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '16). Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2847538.2847539