Труды лаборатории языковых инструментов

Выпуск 4

Санкт-Петербург, 2016

Предисловие

Четвертый выпуск сборника трудов лаборатории языковых инструментов JetBrains на математико-механическом факультете СПбГУ продолжает традицию публикации курсовых и дипломных работ. Представляемые работы выполнены студентами лаборатории в 2015–2016 учебном году и разрабатывают такие темы, как управление памятью, анализ встроенных языков, функциональное программирование. По результатам некоторых из данных работ были подготовлены статьи, принятые для публикации в российских и зарубежных реферируемых журналах и сборниках конференций.

куратор лаборатории Д.Булычев

Оглавление

Ослабленный синтаксический анализ динамически формируемых выражений на основе алгоритма GLL

Рагозина Анастасия Константиновна

Санкт-Петербургский государственный университет
ragozina.anastasiya@gmail.com

Аннотация Данная работа посвящена описанию подхода к синтаксическому анализу регулярных множеств. Подобная задача возникает при анализе встроенных языков или при поиске соответствий в метагеномных сборках в задачах биоинформатики. Регулярное множество задаётся с помощью конечного автомата, порождающего цепочки, и необходимо проверить, принадлежат ли эти цепочки языку, задаваемому грамматикой. В качестве основы использовался алгоритм обобщённого синтаксического анализа GLL из-за его высокой скорости работы.

Введение

При работе с формальными языками и грамматиками выводимость цепочки в грамматике можно рассматривать как следующее свойство: цепочка ω обладает свойством S, если ω выводима из S: $(S \Rightarrow^* \omega)$. При решении практических задач, как правило, выполняют проверку свойства выводимости в грамматике для отдельно взятых цепочек, либо же для конечного множества цепочек, представленных в явном виде (например, в виде множества файлов с исходным текстом программ). На практике такие множества могут оказаться бесконечными, что делает такую проверку невозможной. Подобная ситуация может возникнуть, если цепочки генерируются

автоматически. Множество порождаемых генератором цепочек в этом случае будет регулярным. Для описания регулярных множеств часто используются конечные автоматы. Таким образом возникает задача проверки свойства выводимости в КС-грамматике для всех элементов множества, заданного конечным автоматом. Такую задачу будем называть синтаксическим анализом регулярных множеств.

Описанная задача имеет практическое применение и возникает в ряде областей. Например, широкое распространение при разработке информационных систем получил подход, использующий динамически формируемые программы. Код таких программ формируется в процессе выполнения внешней программы из строковых литералов и в дальнейшем выполняется соответствующим окружением. Такой подход может использоваться для генерации SQL-запросов, Web-страниц, запросов к XML-подобным структурам данных. Однако проблема заключается в том, что динамически формируемый код не подвергается статической проверке стандартными инструментами, так как компилятором он воспринимается как обычные строки. Статический анализ встроенного кода позволил бы выявлять ошибки до того, как программа будет запущена. Кроме того, общепринятой практикой при разработке информационных систем является использование интегрированных сред разработки, которые упрощают процесс разработки путём подсветки синтаксиса, автодополнения и других функций. Для встроенных языков подобные возможности также были бы полезны. Для решения таких задач необходимо иметь структурное представление кода (дерево разбора), которое строится в процессе синтаксического анализа. При этом задача анализа динамически формируемого кода осложняется тем, что все возможные значения программы нельзя задать простым перечислением. Это происходит из-за того, что для формирования кода могут использоваться циклы, потенциально бесконечные. Для представления такого кода можно построить регулярную аппроксимацию сверху, представленную

в виде конечного автомата над алфавитом встроенного языка. Таким образом мы приходим к задаче синтаксического анализа регулярных множеств при обработке динамически формируемого кода. Важной особенностью задач в данной области является необходимость построения дерева разбора, что выдвигает дополнительные требования к алгоритму анализа.

Другим примером использования синтаксического анализа регулярных множеств является поиск подпоследовательностей генома (таких как РНК, например) в задачах биоинформатики. Для того, чтобы классифицировать образцы, взятые из окружающей среды, для них строится метагеномная сборка, являющаяся комбинацией генов всех организмов, находящихся в образце. Сборка представляется в виде графа с последовательностями символов на рёбрах. В таком графе необходимо найти подстроки, позволяющие провести классификацию. Искомые подстроки могут быть описаны КС-грамматикой [4], то есть необходимо искать подстроки, обладающие свойством выводимости. В данном случае не обязательно строить дерево разбора, необходимо лишь ответить на вопрос, порождается ли цепочка данным автоматом. Данная задача может быть решена с помощью синтаксического анализа регулярных множеств.

Анализу динамически формируемых программ посвящён ряд работ [1–3]. Изучению данной проблемы посвящена также работа [36], в которой описан алгоритм анализа встроенных языков с использованием алгоритма RNGLR, строящий структурное представление динамически формируемого кода. Однако в этой работе указано, что у предложенного решения существуют проблемы с производительностью. Синтаксический анализ также применяется в задачах биоинформатики, однако только для линейных входных данных. При этом отдельное внимание необходимо уделять производительности решения. Это обуславливается тем, что входные данные при анализе метагеномной сборки, как правило, имеют очень большой размер: порядка 10^5 рёбер, 10^5 вершин, 10^8 — суммарного количества символов в метках рёбер.

1 Постановка задачи

Целью данной работы является создание решения для синтаксического анализа регулярных множеств, применимого для работы со входными данными большого размера. Для достижения поставленной цели были поставлены следующие задачи:

- Разработать алгоритм синтаксического анализа динамически формируемого кода на основе алгоритма GLL.
- Доказать корректность предложенного алгоритма.
- Применить к задаче поиска на входных данных большого размера.
- Реализовать предложенный алгоритм в рамках проекта YaccConstructor.
- Произвести эксперименты и сравнение.

2 Обзор

2.1 Обобщённый синтаксический анализ

Большинство языков программирования могут быть описаны однозначной КС-грамматикой, но создание такой грамматики является трудоёмким и долгим процессом. На практике, как правило, спецификацию необходимо получить быстро, по этой причине доступная для разработчиков спецификация языка часто содержит неоднозначности. Приведение грамматики к детерминированной форме — процесс сложный, часто приводящий к появлению ошибок. Для работы с неоднозначными грамматиками используются алгоритмы обобщённого синтаксического анализа. Такие алгоритмы рассматривают все возможные пути разбора входной цепочки и строят все деревья вывода для неё.

Впервые такой подход был предложен в работе [15]: на основе алгоритма восходящего синтаксического анализа LR

был разработан алгоритм GLR. Такой алгоритм позволил обрабатывать конфликты типа shift/reduce и reduce/reduce, которые не могут быть корректно обработаны обычными LR-анализаторами. Принцип работы GLR-алгоритма остался таким же, как и у LR-алгоритма, но для заданной грамматики GLR-парсер строит все возможные выводы входной последовательности, используя поиск в ширину. В ячейках LR-таблиц хранится не более одного правила свёртки для текущего символа во входном потоке. В ячейках таблицы GLR-парсера может храниться несколько правил свёртки. Эта ситуация соответствует конфликту типа reduce/reduce. Когда возникает конфликт, т.е. символ на входе может быть разобран несколькими разными способами, стек парсера разветвляется на два или больше параллельных стека. Верхние состояния этих стеков соответствуют возможным переходам. Если для какого-либо верхнего состояния и входного символа в таблице не существует ни одного перехода, то эта ветка разбора считается ошибочной и отбрасывается.

Позже было предложено множество модификаций GLR-алгоритма: RIGLR [15], RNGLR [17], BRNGLR [14]. Восходящие анализаторы в отличие от нисходящих, как правило, имеют сложную структуру и трудны для разработки. В 2010 году был предложен алгоритм обобщённого анализа Generalised LL (GLL) [4] на основе алгоритма нисходящего синтаксического анализа. Основная идея осталась прежней — просмотр всех возможных путей разбора и ветвление стека в случае возникновения неоднозначностей. В силу особенностей работы LL-алгоритма соответствующие анализаторы более просты для разработки, и, кроме того, они имеют более высокую скорость работы.

Прежде чем переходить к описанию процесса работы алгоритма, рассмотрим принцип работы синтаксических анализаторов, созданных с помощью метода рекурсивного спуска (Recursive Descent, RD). RD-анализаторы являются процедурной интерпретацией грамматики. Их непосредственная связь с

грамматикой упрощает их разработку, отладку и внесение изменений при необходимости. Рассмотрим простую грамматику G_0 (листинг 1) и соответствующий анализатор, написанный методом рекурсивного спуска.

Listing 1 Грамматика G_0

$$s \to a\ b \mid b\ C$$
$$b \to A \mid C$$
$$a \to A$$

Анализатор, созданный с помощью метода рекурсивного спуска, для данной грамматики будет состоять из функций для разбора нетерминалов — $parseS()$, $parseB()$, $parseA()$, и управляющей процессом разбора функции main(). Но поскольку данная грамматика не является однозначной, разбор даже корректного входа, например, цепочки *"ac"*, будет заканчиваться ошибкой. Кроме того, в худшем случае время работы таких парсеров может экспоненциально зависеть от размера входа для сильнонеоднозначных грамматик.

Алгоритм Generalised LL является обобщением алгоритмов LL и рекурсивного спуска и способен обрабатывать все контекстно свободные грамматики, в том числе содержащие левую рекурсию. Как и в классическом рекурсивном спуске, в GLL-анализаторах сохраняется тесная связь с грамматикой, что упрощает реализацию и отладку. Вместе с этим, расход памяти и время работы в худшем случае является кубическим относительно размера входа и линейным для LL-грамматик. Это обеспечивается за счёт использования специализированных структур данных хранения стека и леса разбора.

2.2 Структурированный в виде графа стек

Стек в алгоритмах синтаксического анализа используется для того, чтобы запоминать ранее разобранные символы.

Поскольку алгоритмы обобщённого синтаксического анализа строят все возможные выводы входной строки, необходимо для каждого варианта разбора хранить свой стек. Как только в процессе разбора происходит конфликт, создаётся несколько новых процессов анализа и каждый из них будет иметь свой собственный стек. Каждый такой стек будет иметь общую часть и разным у них будет только верхнее состояние. Проблема данного подхода заключается в том, что хранение отдельных стеков в явном виде требует слишком больших накладных расходов. Для того, чтобы бороться с этим, стеки комбинируются с помощью структуры данных GSS (Graph Structured Stack). В результате всё множество стеков представляется в виде графа, а в качестве вершины конкретного стека можно хранить только указатель на соответствующую вершину графа. На рис. 1 показан пример объединения нескольких стеков: для стеков S1, S2 и S3 в результате будет получен S1.

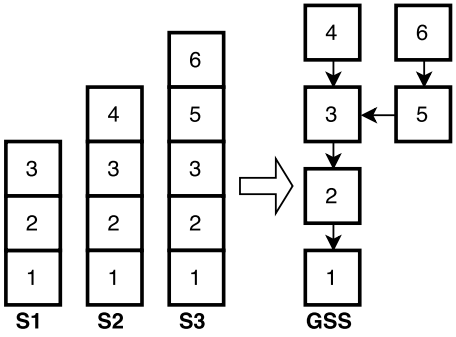


Рис. 1: Стек, структурированный в виде графа

В вершине GSS хранится правило грамматики с указанием позиции в правой части (обозначается $X \to \alpha x \cdot \beta$, дальше просто *слот*) и позиция во входном потоке. Позиция во вход-

ном потоке используется для того, чтобы различать вершины стека. На ребре GSS хранится часть леса разбора, построенное на соответствующем шаге работы анализатора. Описанное представление GSS обладает существенным недостатком: многие вершины и рёбра дублируются. Для примера рассмотрим грамматику G_1 (листинг 2).

Listing 2 Грамматика G_1

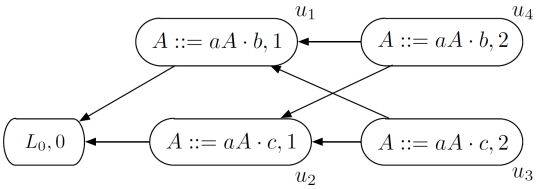
$$a \to A\ a\ B \mid A\ a\ C \mid A$$

Для такой грамматики в процессе разбора будет построен GSS (в дальнейшем просто стек), показанный на рис. 2a. На рисунке видно, что рёбра дублируются. Этот же стек может быть представлен как показано на рис. 2b. Таким образом можно значительно уменьшить размер графа без потери информации.

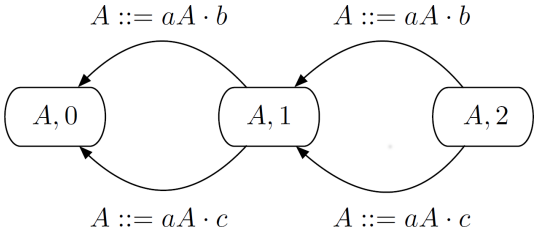
Данная модификация стека предложена в работе [7], посвященной улучшению производительности работы алгоритма GLL. Уменьшение количества рёбер в стеке достигается за счёт хранения в вершинах не слота целиком, а только имени нетерминала и позиции во входном потоке. Соответствующий слот в описанном варианте хранится на ребре стека. В конечном итоге уменьшение количества вершин и рёбер позволяет значительно ускорить время работы алгоритма и уменьшить объём потребляемой памяти [7]. В данной работе были использованы эти результаты.

2.3 Сжатое представление леса разбора

Результатом работы синтаксического анализатора является дерево разбора. Для неоднозначных грамматик для одной и той же входной цепочки может быть построено несколько различных деревьев, для некоторых грамматик количество



(a) Старая версия GSS



(b) Модифицированный GSS

Рис. 2: Изменения GSS

деревьев может экспоненциально зависеть от размера входа. Для того, чтобы уменьшить количество требуемой для хранения деревьев памяти, используется структура данных Shared Packed Parse Forests (SPPF) [1], которая позволяет хранить лес разбора более компактно. Это достигается за счёт того, что в SPPF узлы с одинаковыми деревьями под ними переиспользуются, а узлы, которые соответствуют разным выводам одной и той же цепочки из одного и того же нетерминала, комбинируются. Например, для сильно неоднозначной грамматики G_2 (листинг 3) и входной цепочки *"bbb"* может быть построено два дерева, показанных на рис. 3*(а)* и рис. 3*(б)*, которые могут быть сжаты в SPPF так, как показано на рис. 3*(в)*.

Listing 3 Грамматика G_2

$$s \to s\ s \mid B$$

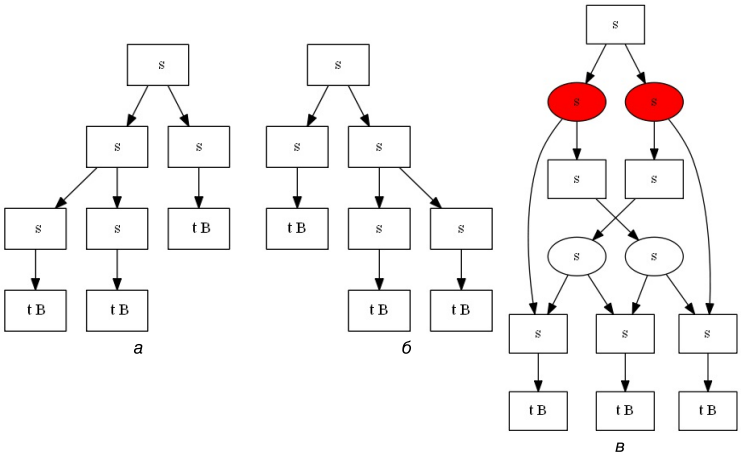


Рис. 3: *а* — первый вывод; *б* — второй вывод; *в* — сжатое представление леса разбора

В алгоритме GLL строится бинаризированная версия SPPF, в которой используется три типа узлов: символьные, упакованные и промежуточные узлы. Символьные узлы представляют собой тройку (x, i, j), где x имя нетерминала или терминала (различают, соответственно, терминальные и нетерминальные узлы), а i и j — позиции во входном потоке, которым соответствует рассматриваемая часть дерева (для корня — узла, помеченного стартовым нетерминалом — начальной позицией будет 0, а конечной — длина строки). Позиции во входном потоке дальше будем называть правой и левой координатами узла. Промежуточные узлы хранят слот и позиции во входном потоке. Упакованные узлы имеют вид $(a \to \gamma\cdot, k)$, где k — правая позиция для левого сына, которая используется для того, чтобы различать узлы. Терминальные узлы не имеют потомков и являются листьями в SPPF. Нетерминальные и промежуточные узлы могут иметь несколько потомков, упакован-

ные узлы имеют только оного или двух, где правым потомком является символьный узел вида (x, k, i), а левым, если существует — символьный или промежуточный узел вида (s, j, k). Упакованные узлы используются для описания фактов неоднозначности в выводе, т.е. если у нетерминала более одного упакованного узла в потомках, то для этого нетерминала было построено несколько выводов одной и той же цепочки. На рис. 4 изображен SPPF для грамматики G_2.

На рис. 4 точками обозначены упакованные узлы. Узлы, помеченные слотами, являются промежуточными, а терминальные и нетерминальные узлы подписаны. У стартового нетерминала имеется два потомка, что свидетельствует о существовании двух выводов для данной входной цепочки.

2.4 Алгоритм GLL

Процесс работы GLL-анализатора можно рассматривать как обход грамматики в соответствии со входным потоком. На каждом шаге выполнения GLL анализатор находится в некоторой позиции в грамматике вида $x \to \alpha X \cdot \beta$, которая обозначается L, и поддерживает три переменные: текущая позиция во входном потоке (cI), текущая вершина стека (cU) и текущий узел в дереве (cN). Эта четвёрка называется дескриптором и позволяет полностью описать текущий шаг разбора. Дескрипторы извлекаются из очереди и разбор каждый раз начинается заново с точки, описанной в дескрипторе. Если какой-то путь не может быть продолжен, то процесс анализа не заканчивается ошибкой: вместо этого из очереди извлекается следующий дескриптор и процесс возобновляется с точки, описанной в нём. Алгоритм завершает работу, как только очередь дескрипторов становится пустой.

Синтаксический анализатор состоит из набора функций с уникальными метками, управление между которыми передаётся с помощью оператора goto(). В отличие от парсеров, созданных с помощью метода рекурсивного спуска, в GLL-

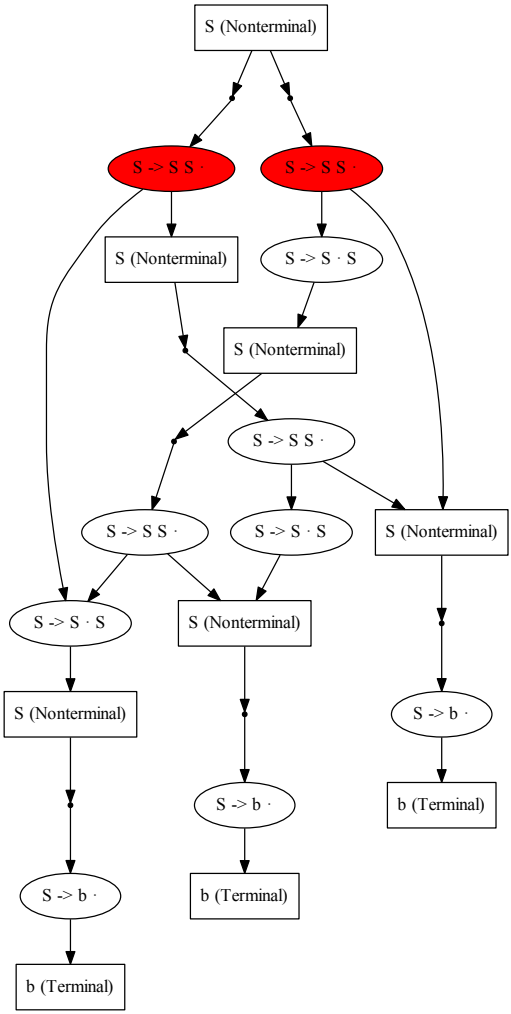


Рис. 4: Бинаризированное представление сжатого леса разбора для грамматики G_2 и входной строки *“bbb”*

анализаторах функции генерируется для каждой альтернативы и у goto() может быть несколько целевых меток. Функции бывают двух видов: для каждой альтернативы нетерминала, соответствующие слотам в грамматике вида $a \to \cdot\gamma$, и функциями для слотов вида $y \to \delta x \cdot \mu$. При создании дескриптора в качестве текущей позиции в грамматике запоминается имя целевой функции. Кроме того, имеется управляющая функция, которая извлекает очередной дескриптор из очереди и вызывает соответствующую функцию с помощью операции goto(). Также есть функция для построения стека create() и для построения дерева getNodeP() и getNodeT(). Функция getNodeT() используется для создания терминального узла, а getNodeP() для создания всех остальных видов узлов.

Стек частично заменяет вызов функции в парсерах, написанных методом рекурсивного спуска. Вершины стека создаются, как только в процессе обхода грамматики встречается нетерминал (например, $x \to \alpha \cdot a\beta$). Как упоминалось ранее, на вершинах стека находятся пары (N, i), где N — имя нетерминала, который необходимо разобрать, i — позиция во входном потоке на момент создания вершины. На ребре хранится слот и часть леса разбора. Слот позволяет сохранить информацию о том, в какую позицию в грамматике необходимо вернуться после того, как нетерминал будет разобран (в случае слота $x \to \alpha a \cdot \beta$ разбирается нетерминал a). Второй элемент пары — часть леса разбора, которая была построена на момент создания вершины стека (для рассматриваемого слота $x \to \alpha a \cdot \beta$ это часть SPPF для цепочки α). После того, как нетерминал будет разобран, вершина извлекается из стека и для всех исходящих рёбер создаются новые дескрипторы (l, i, u, t), где l — слот с ребра, i — текущая позиция во входном потоке, u — целевая вершина ребра и t — часть леса разбора, полученная объединением части леса с ребра и построенной для нетерминала (для рассматриваемого слота $x \to \alpha a \cdot \beta$ объединяется часть леса для цепочки α и нетерминала A).

Listing 4 Грамматика G_3

$$s \to A\ s\ B \mid D \mid A\ D\ B$$

Кроме того, для корректной работы алгоритма используются дополнительные множества. Все дескрипторы добавляются в очередь R и последовательно извлекаются из неё в процессе работы анализатора. Процесс работы синтаксического анализатора недетерминирован, и может возникнуть ситуация, когда один и тот же дескриптор будет создаваться снова и снова. Повторное создание дескриптора приводит к тому, что процесс зациклится и никогда не завершится. Для того, чтобы избежать дублирования дескрипторов, отдельно хранится множество U ранее созданных дескрипторов. В очередь добавляются только дескрипторы, не содержащиеся во множестве U. Для того, чтобы хранить и переиспользовать уже разобранные нетерминалы, используется множество P. В нём хранятся пары вида (u, z), где z — узел SPPF, а u — вершина GSS. На рис. 5 приведён пример GLL-анализатора для грамматики G_3 (листинг 4). В приведённом примере функция L_0 содержит основной цикл, который извлекает дескрипторы из очереди и присваивает переменным cU, cI и cN значения из извлечённого дескриптора.

2.5 Подходы к анализу встроенных языков

В области анализа встроенных языков ведутся исследования и разрабатываются инструменты, реализующие различные подходы.

– Java String Analyzer (JSA) [2, 14] — инструмент для анализа строк и строковых операций в программах на Java. Основан на проверке включения регулярной аппроксимации встроенного языка в контекстно-свободное описание эталонного языка.

$c_I := 0$; $c_U := u_0$
$\mathcal{R} := \emptyset$; $\mathcal{P} := \emptyset$
for $0 \leq j \leq m$ { $\mathcal{U}_j := \emptyset$ }
goto L_S

L_0: **if** ($\mathcal{R} \neq \emptyset$) { remove (L, u, i, w) from $\mathcal{R}$
$c_U := u$; $c_I := i$; $c_N := w$; **goto** L }
else if (there is an SPPF node $(S, 0, m)$) report success
else report failure

L_S: **if** ($I[c_I] \in \{a\}$) { $add(L_{S_1}, c_U, c_I, \$)$; $add(L_{S_3}, c_U, c_I, \$)$ }
if ($I[c_I] \in \{d\}$) $add(L_{S_2}, c_U, c_I, \$)$
goto L_0

L_{S_1}: $c_N := getNodeT(a, c_I)$; $c_I := c_I + 1$
if ($I[c_I] \in \{a, d\}$) { $c_U := create(R_{S_1}, c_U, c_I, c_N)$; **goto** L_S }
else goto L_0

R_{S_1}: **if** ($I[c_I] = b$) $c_R := getNodeT(b, c_I)$ **else goto** L_0
$c_I := c_I + 1$; $c_N := getNodeP(S ::= aSb., c_N, c_R)$;
$pop(c_U, c_I, c_N)$; **goto** L_0

L_{S_2}: $c_R := getNodeT(d, c_I)$
$c_I := c_I + 1$; $c_N := getNodeP(S ::= d., c_N, c_R)$
$pop(c_U, c_I, c_N)$; **goto** L_0

L_{S_3}: $c_N := getNodeT(a, c_I)$; $c_I := c_I + 1$
if ($I[c_I] = d$) $c_R := getNodeT(d, c_I)$ **else goto** L_0
$c_I := c_I + 1$; $c_N := getNodeP(S ::= ad.b, c_N, c_R)$
if ($I[c_I] = b$) $c_R := getNodeT(b, c_I)$ **else goto** L_0
$c_I := c_I + 1$; $c_N := getNodeP(S ::= adb., c_N, c_R)$
$pop(c_U, c_I, c_N)$; **goto** L_0

Рис. 5: GLL-анализатор для грамматики G_3

– PHP String Analyzer (PHPSA) [15, 27] — инструмент для статического анализа строк в программах на PHP. Расширяет подход инструмента JSA. В инструменте уточнена проводимая аппроксимация, и это повышает точность проводимого анализа.

– Alvor [4–6] — плагин к среде разработки Eclipse, предназначенный для статической проверки корректности SQL-выражений, встроенных в Java. Для компактного представления множества динамически формируемого строкового выражения используется понятие абстрактной строки, которая, фактически, является регулярным выражением над используемыми в строке символами. В инструменте Alvor отдельным этапом выделен лексический анализ. Поскольку абстрактную строку можно преобразовать в конечный автомат, то лексический анализ заключается в преобразовании этого конечного автомата в конечный автомат над терминалами при использовании конечного преобразователя, полученного генератором лексических анализаторов JFlex. Несмотря на то, что абстрактная строка позволяет описывать строковые выражения, при конструировании которых использовались циклы, плагин в процессе работы выводит сообщение о том, что данная языковая конструкция не поддерживается. Также инструмент Alvor не поддерживает обработку строковых операций, за исключением конкатенации, о чём также выводится сообщение во время работы.

– IntelliLang [7] — плагин к средам разработки IntelliJ IDEA и PHPStorm, предоставляющий поддержку встроенных строковых языков, таких как HTML, SQL, XML, JavaScript в указанных средах разработки. Данное расширение обеспечивает подсветку синтаксиса, автодополнение, статический поиск ошибок. Для среды разработки IntelliJ IDEA расширение IntelliLang также предоставляет отдельный текстовый редактор для работы со встроенным языком. Для использования данного плагина требуется ручная раз-

метка переменных, содержащих выражения на том или ином встроенном языке.
– Varis [9] — плагин для Eclipse, представленный в 2015 году и предоставляющий поддержку кода на HTML, CSS и JavaScript, встроенного в PHP. В плагине реализованы функции подсветки встроенного кода, автодополнения, перехода к объявлению (jump to declaration), построения графа вызовов (call graph) для встроенного JavaScript.

Все эти инструменты предназначены для решения достаточно узких задач и часто не предусматривают проведения сложного анализа динамически формируемого кода. Более сложные виды анализа могут быть произведены с применением абстрактного синтаксического анализа, который предложен в работе [2]. Алгоритм абстрактного синтаксического анализа комбинирует анализ потока данных и синтаксический LR-анализ. Входными данными для него является набор data-flow уравнений, описывающих множество значений динамически формируемого кода. Данные уравнения решаются в домене LR-стеков при помощи абстрактной интерпретации [9], обеспечивающей свойство завершаемости алгоритма. Результатом работы является набор абстрактных синтаксических деревьев, которые в дальнейшем могут использоваться для решения различных задач статического анализа. К сожалению, в работах отсутствует рассмотрение эффективного представление результатов анализа. Кроме того, инструментов, реализующих предложенный подход, в открытом доступе нет.

Также существует подход к обработке динамически формируемого кода, основанный на алгоритме обобщённого восходящего анализа RNGLR [36]. Подход основан на проверке включения регулярного языка в некоторые подклассы КС-языков. В качестве входных данных в работе используется регулярное приближение множества всех значений динамически формируемого кода в некоторой точке программы-генератора. Данное приближение описывается регулярным языком L и является приближением сверху (over-approximation) для множества

возможных значений: регулярный язык содержит все предложения, генерируемые программой и, возможно, ещё какие-то. Благодаря этому можно говорить о достоверности многих видов статического анализа. Таким образом, регулярная аппроксимация для множества значений динамически формируемых выражений позволяет решать многие важные задачи, например, поиск ошибок во встроенном коде. Для представления регулярной аппроксимации используются конечные автоматы, так как для любого регулярного языка L можно построить такой конечный автомат, что он принимает те и только те цепочки, которые принадлежат языку L. Далее построенный конечный автомат подаётся на вход лексическому анализу (преобразуется из автомата над символами в автомат над токенами), а затем синтаксическому анализу, алгоритм которого основан на алгоритме RNGLR. Достоинствами предложенного решения являются модульность, позволяющая рассматривать различные этапы анализа отдельно, и возможность построения конечного леса вывода для всех корректных цепочек из L, что позволяет реализовывать различные виды анализа динамически формируемого кода, требующие его структурного представления.

2.6 YaccConstructor

YaccConstructor [2] — исследовательский проект лаборатории языковых инструментов JetBrains, направленный на изучение алгоритмов лексического и синтаксического анализа. Проект включает в себя одноимённый модульный инструмент с открытым исходным кодом, предоставляющий платформу для разработки лексических и синтаксических анализаторов, содержащую большое количество готовых компонент, таких как различные преобразования грамматик, язык описания атрибутных грамматик YARD и др. Большинство компонент реализованы на платформе .NET, основным языком разработки является F# [10]. Предоставляемый язык спецификации грамматик YARD поддерживает атрибутные граммати-

ки, грамматики в расширенной форме Бэкуса-Наура и многое другое.

В рамках проекта была создана платформа для статического анализа динамически формируемого кода, основанная на модульной архитектуре, предложенной в [11]. В рамках соответствующих модулей реализованы механизмы лексического анализа, описанный в [7], и алгоритм синтаксического анализа, описанный в [36]. Благодаря модульной архитектуре данные компоненты могут использоваться независимо.

2.7 Анализ метагеномной сборки

В биологических исследованиях часто необходимо ответить на вопрос, к какому виду относится тот или иной организм. Образцы часто берутся из окружающей среды и могут быть не идентифицированы. По таким образцам строится метагеномная сборка, являющаяся множеством участков ДНК различных организмов. Такое множество может быть представлено в виде конечного автомата, порождающего геномы всех организмов, содержащихся в образце. Саму последовательность ДНК можно рассматривать, как строку в алфавите $\{A, C, G, T\}$, однозначно определяющую организм (или штамм), к которому она относится.

Наиболее часто используемый подход для идентификации образцов, взятых из окружающей среды, — проведение сравнительного анализа последовательности рРНК, поскольку большая часть информации об организме хранится именно в рРНК. При этом для поиска и идентификации подцепочек используется несколько основных механизмов: скрытые цепи Маркова, используемые в таких инструментах, как HMMER [30], REAGO [20], ковариационые модели (covariation model, CM) [37], основанные на вероятностных грамматиках и применении синтаксического анализа для задачи поиска. Однако проблема заключается в том, что восстановление небольших участков рРНК по объёмной метагеномной сборке оказы-

вается трудоёмким и не всегда её может быть решена эффективно.

Существующие подходы для решения такой задачи можно разделить на два класса. Первый класс сначала использует инструменты для анализа сборок de novo (полные геномные последовательности организмов, находящихся в метагеномной сборке, ещё не известны), которые восстанавливают геномные последовательности в линейном виде. Затем результат их работы передаётся инструментам геномного поиска рРНК. Проблема данных инструментов заключается в том, что восстановление генов в метагеномной сборке — задача очень трудоёмкая. Кроме того, большая часть генов de novo состоит из участков, не носящих информации об рРНК. Таким образом, подход построения метагеномой сборки и поиска рРНК по ней с использованием сторонних инструментов не оптимален.

Другой класс инструментов реализует поиск непосредственно в метагеномной сборке. Часть из них используют предварительную фильтрацию отдельных линейных участков (предполагается, что сборка ещё не представлена в виде конечного автомата), что позволяет избавиться от анализа последовательностей, не несущих необходимой информации. Затем из оставшихся частей производится сборка рРНК. Такой подход используется в EMIRGE [22]. Недостатком подхода является необходимость наличия большого числа известных рРНК и высокая вероятность не найти далеко стоящие друг от друга рРНК. Инструмент REAGO лишён этих проблем, однако он не работает с метагеномной сборкой, представленной в виде графа, а хранение сборки в виде набора прочитанных участков требует большого объёма памяти.

Обработка сборки, представленной в виде графа, реализована в инструменте Xander [23], который использует для решения задачи композицию скрытых моделей Маркова (HMM) и входного графа. Недостатком такого подхода является то, что использование HMM даёт существенно более низкую точность результата по сравнению с использованием CM [38]. Наибо-

лее известным инструментом, использующим CM для поиска рРНК является Infernal [10]. Однако он не применим к метагеномным сборкам, представленным в виде графа.

2.8 Выводы

В результате выполненного обзора были сделаны следующие выводы:

1. Алгоритм синтаксического анализа, предложенный в работе [36], позволяет решать задачу построения леса вывода для динамически формируемого кода. Поэтому идеи, предложенные в этих работах, взяты за основу данной.
2. Алгоритм синтаксического анализа GLL имеет более очевидную связь с грамматикой, чем восходящие алгоритмы анализа, поддерживает произвольные КС-грамматики и обладает высокой производительностью, что позволяет использовать его для задач синтаксического анализа регулярных множеств.
3. YaccConstructor содержит готовое решение для синтаксического анализа динамически формируемого кода, основанное на RNGLR-алгоритме. При этом архитектура этого решения позволяет легко заменять алгоритмы синтаксического анализа, оставляя без изменений другие компоненты, такие как лексический анализ или построение аппроксимации [5]. Таким образом, YaccConstructor является подходящей платформой для практической реализации.
4. Применение механизмов синтаксического анализа к конечным автоматам является важной задачей в биоинформатике.

3 Алгоритм анализа регулярных множеств

Целью данной работы является создание алгоритма синтаксического анализа регулярных множеств, который может

быть применим для анализа встроенных языков. Как упоминалось ранее, встроенный код порождается в момент выполнения основной программы. Для генерации могут использоваться строковые операторы, условные операторы и циклы, из-за чего такой код не может быть представлен линейно. Результатом работы лексического анализатора на таком коде является не линейный поток токенов, а конечный автомат над алфавитом токенов. В рамках данной работы на такие автоматы накладывается следующее ограничение: они должны быть детерминированными. Если это условие не будет выполняться, то нельзя будет однозначно выбрать, по какому пути производить разбор. Например, для встроенного кода на листинге 5 будет построен конечный автомат на рис. 6.

Listing 5 Код на C#, динамически формирующий скобочную последовательность

```
1 string expr = "" ;
2 for(int i = 0; i < len; i++)
3 {
4     expr = "()" + expr;
5 }
```

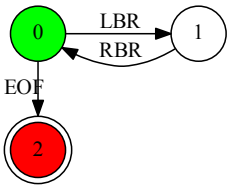


Рис. 6: Конечный автомат, представляющий аппроксимацию встроенного кода для листинга 5

В алгоритме вместо позиции во входном потоке теперь хранится номер вершины во входном графе. Поскольку вход является нелинейным, то вместо того, чтобы просматривать один текущий входной символ на каждом шаге, рассматриваются все исходящие рёбра для текущей вершины и выбирается одно (как упоминалось ранее, автомат детерминирован, поэтому это возможно), соответствующее текущему терминальному символу в грамматике. Если такого ребра нет, то алгоритм просто продолжает свою работу — из очереди достаётся новый дескриптор, и процесс возобновляется.

Для автоматического создания синтаксических анализаторов существует несколько подходов. В рамках первого подхода весь код парсера генерируется по грамматике. Чаще всего такой подход используется при генерации анализаторов, построенных методом рекурсивного спуска. При генерации нисходящих анализаторов для каждого нетерминала генерируются функции, которые последовательно вызываются в процессе разбора. Несмотря на то, что нисходящие анализаторы просты для разработки, и поэтому чаще всего создаются вручную, существуют инструменты для автоматической генерации таких анализаторов. Например, инструмент ANTLR [24] — генератор парсеров, позволяющий автоматически создавать анализаторы на одном из целевых языков программирования по описанию LL(*)-грамматики на языке, близком к EBNF. Структура генераторов такого типа изображена на рис. 7*(а)*.

Существует ещё один подход для генерации синтаксических анализаторов, который используется для получения табличных анализаторов. Отдельно создаётся интерпретатор, который содержит в себе основную логику алгоритма. Интерпретатор пишется вручную и переиспользуется. По грамматике каждый раз генерируется дополнительная информация, которая необходима интерпретатору в процессе работы. Структура такого генератора представлена на рис. 7*(б)*. Чаще всего в качестве дополнительной информации генерируются таблицы синтаксического анализа, управляющие процессом разбора.

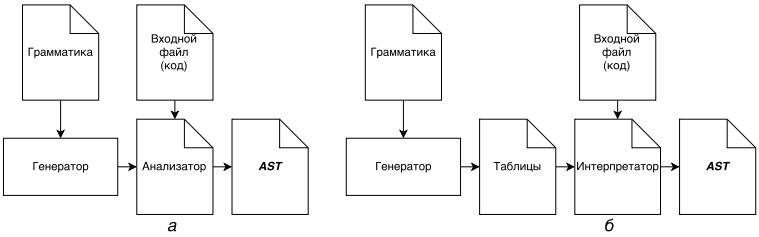


Рис. 7: Подходы к генерации синтаксических анализаторов

В оригинальных работах, описывающих GLL-алгоритм, используется первый подход. В рамках данной работы был выбран второй подход из-за его гибкости и универсальности. Вместо генерации функций по слотам грамматики и их последовательного вызова, в главном цикле алгоритма просто рассматриваются все возможные состояния, в которых может находиться парсер. В зависимости от того, какой символ во входном потоке и какая позиция в грамматике, в процессе разбора рассматриваются следующие ситуации:

– Если текущий символ в грамматике является терминалом x и существует исходящее из текущей вершины ребро, помеченное этим нетерминалом, то указатель в грамматике нужно сдвинуть на одну позицию вправо, $x \to \alpha X \cdot \beta$, и текущей вершиной назначить конечную вершину ребра. Никаких дополнительных действий со стеком при этом не производится. Иначе, если нет ребра, помеченного терминалом X, то текущая ветка разбора считается ошибочной, отбрасывается и разбор продолжается с использованием следующего дескриптора.
– Если текущий символ в грамматике является нетерминалом a, то необходимо в стек записать слот, по которому продолжить разбор после того, как правило для a будет разобрано. Указатель в грамматике перемещается на

$a \to \cdot\gamma$, а номер вершины во входном потоке остаётся без изменений.
- Если указатель в грамматике имеет вид $x \to \alpha\cdot$ и стек не пуст, то слот вида $y \to \delta x \cdot \mu$, который хранится в этот момент в текущей вершине стека, извлекается и становится текущим.
- Если текущий слот имеет вид $s \to \tau\cdot$, и весь входной поток рассмотрен, то разбор завершается успешно, иначе разбор заканчивается ошибкой. В случае успешного завершения разбора возвращается дерево, иначе — сообщение об ошибке.

Наличие циклов во входном графе никак не влияет на процесс разбора. Дескрипторы позволяют без каких-либо изменений процесса разбора обработать их. Это делает результирующий алгоритм более простым, чем алгоритм, основанный на RNGLR, в который потребовалось внести существенные изменения для поддержки циклов [36]. За счёт того, что каждый раз при добавлении дескриптора выполняется проверка всей четвёрки целиком (позиция во входе, слот, вершина стека и часть леса разбора), то лишние дескрипторы с одинаковыми деревьями не создаются. Переиспользование уже созданных узлов также позволяет избежать создания лишних деревьев: если дерево с определёнными координатами и соответствующим правилом вывода уже было создано, то повторно такое дерево создаваться не будет.

В алгоритме также поддерживается четвёрка: слот (вместо имени функции), номер вершины в графе, вершина стека и узел дерева. Поскольку вызов функций заменён на обработку ситуаций, возникающих в процессе анализа в теле основной функции, то появилась необходимость определять, какое правило вывода использовать для разбора. Для определения правила используются LL-таблицы, где в каждой ячейке может быть несколько правил для разбора, что соответствует ситуации наличия в грамматике неоднозначностей. Анализатор состоит из функции, содержащей основной цикл алгорит-

ма, функции, управляющей процессом разбора и функций для построения дерева и стека.

Listing 6 Функция, содержащая в себе основную логику алгоритма

```
function PARSING()
  condition ← true
  if isEpsilonRule(cL.rule) then
    cR ← newTerminalNode("Epsilon packExtension(cI, cI))
    cN ← getNodeP(cL, cN, cR)
    pop(cU, cI, cN)
  else
    if isEndOfRule(cL.rule, cL.position) then
      curSmb ← grammarRules[cL.rule][cL.position]
      if isTerminal(curSmb) then
        curSmb ← grammarRules[cL.rule][cL.position]
        if cI.OutEdges contains edge labeled with curSmb then
          curEdge ← edge labeled curSmb
          cR ← getNodeT(curEdge)
          cI ← curEdge.TargetVertex
          cL ← label(cL.rule, cL.position + 1)
          cN ← getNodeP(cL, cN, cR)
          condition ← false
      else
        cU ← create(cI, label(cL.rule, cL.position + 1), cU, cN)
        for all edge in outgoing edges of cI do
          for all ruleintable[curSymbol, edge.Token] do
            addContext(cI, packLabel(rule, 0), cU, $)
    else
      pop(cU, cI, cN)
```

На листинге 7 приведены две функции управляющие разбором. Функция control() в зависимости от значений булевых переменных stop и condition вызывает функции dispatcher() или parsing(). Функция dispatcher() извлекает из очере-

Listing 7 Функции, управляющие процессом разбора

```
function CONTROL()
  condition ← true
  while not stopr do
    if condition then dispatcher()
    elseprocessing()
function DISPATCHER()
  if Q is not empty then
    currentContext ← R.Dequeue()
    cI ← currentContext.Index
    cU ← currentContext.GSSNode
    cL ← currentContext.Label
    cN ← currentContext.SPPFNode
    cR ← DummySPPFNode
    condition ← false
  else
    stop ← true
```

ди дескриптор, присваивает значения переменным. Функция parsing() на листинге 6 содержит в себе основную логику алгоритма.

3.1 Пример работы алгоритма

Рассмотрим следующий пример. В качестве входных данных будем использовать конечный автомат M, представленный на рис. 8, который генерирует произвольные скобочные последовательности. Необходимо построить лес разбора для всех цепочек, порождаемых автоматом M, выводимых в грамматике G_4 (листинг 8), описывающей язык правильных скобочных последовательностей.

В результате работы описанного алгоритма построено сжатое представление леса разбора, представленное на рис. 9. Циклы в сжатом представлении леса разбора отображают наличие циклов во входном конечном автомате и позволяют из-

Listing 8 Грамматика G_4

$$s \to LBR\ s\ RBR\ s \mid \varepsilon$$

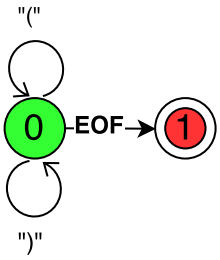


Рис. 8: Конечный автомат, подаваемый на вход анализатору

влекать потенциально бесконечное множество деревьев, каждое из которых соответствует цепочке, порождаемой автоматом.

3.2 Доказательство корректности

Для того чтобы показать, что предложенный алгоритм работает корректно, сначала нужно доказать, что процесс останавливается.

Теорема 1. *Алгоритм завершает свою работу для произвольного детерминированного конечного автомата и контекстно-свободной грамматики.*

Доказательство.

Алгоритм завершает свою работу как только очередь дескрипторов становится пустой. Дескриптор с определённым набором значений полей в очередь добавляется лишь единожды. Таким образом, чтобы показать завершаемость алгоритма достаточно доказать, что количество дескрипторов конечно.

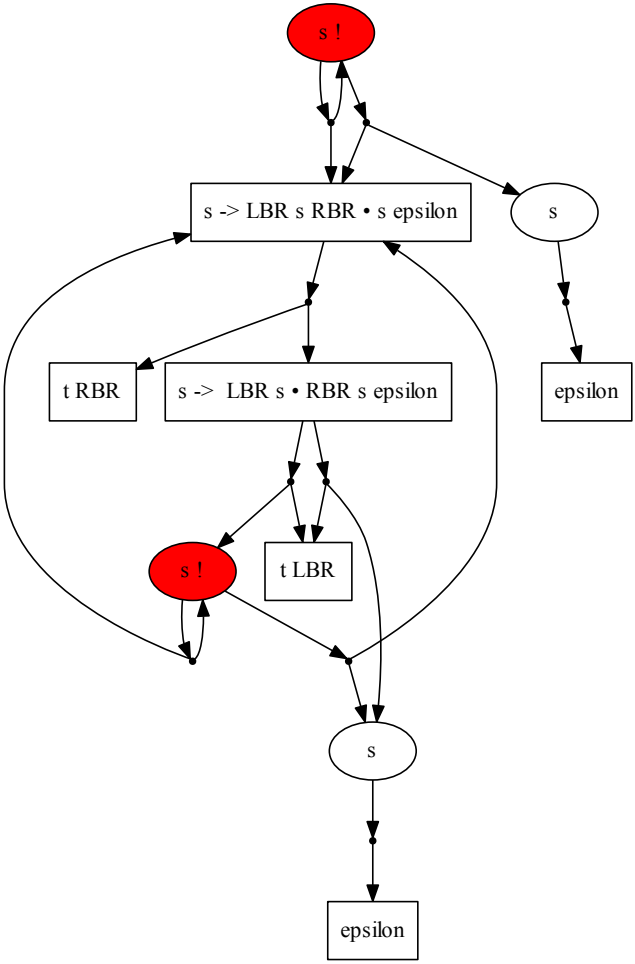


Рис. 9: Сжатое представление леса разбора для грамматики G_4 и конечного автомата на рис. 8

Дескриптор состоит из четырёх элементов: слот, индекс во входном потоке, вершина стека, дерево. Таким образом, общее количество дескрипторов не превышает прямого произведения возможного количества каждого из этих элементов. Количество индексов не больше количества вершин входного графа. Количество слотов конечно, потому что грамматика конечна. Вершина стека определяется парой — слот и индекс, и, значит тоже конечно. Часть леса, хранимая в дескрипторе, определяется однозначно именем нетерминала или слотом и двумя координатами во входном графе. Обе составляющие конечны. □

Таким образом было показано, что количество дескрипторов — конечное число.

Определение 1. *Корректное дерево* — это упорядоченное дерево со следующими свойствами:

1. Корень дерева соответствует стартовому нетерминалу грамматики G.
2. Листья соответствуют терминалам грамматики G. Упорядоченная последовательность листьев соответствует некоторому пути во входном графе.
3. Внутренние узлы соответствуют нетерминалам грамматики G. Потомки внутреннего узла (для нетерминала N) соответствуют символам правой части некоторой продукции для N в грамматике G.

Для того, чтобы доказать, что SPPF содержит только корректные деревья, сначала необходимо доказать следующую лемму.

Лемма 1. *Для любой части леса t, построенного в процессе вывода, существует путь в графе , такой, что крона t покрывает этот путь.*

Доказательство.

Для доказательство используется индукция по построению SPPF.

База.

Для терминальных узлов утверждение очевидно. Терминальный узел соответствует ровно одному ребру во входном графе и строится только после прохода по этому ребру. Построение эпсилон-узлов никак не зависит от входного графа, а производится только в соответствии с грамматикой.

Переход.

Достаточно доказать для упакованных ячеек, всё остальное доказывается аналогично. Создание упакованных ячеек происходит в двух случаях: при чтении нового терминала из входного потока или изъятии вершины стека, что значит, что текущий нетерминал был разобран и необходимо вернуться к точке, с которой этот разбор начался.

Рассмотрим первый случай. У нас есть часть леса, которая соответствует какому-то пути p в графе от вершины v_0 до v_1. Текущая позиция во входном потоке соответствует правой координате для этой части SPPF. При считывании нового терминала создаётся упакованная ячейка, левым сыном которой становится уже построенная часть SPPF, а правым — терминал. Получаем новую часть леса, соответствующее пути $P_1 = v_0 \dots v_1 v_{1+1}$.

Рассмотрим второй случай — изъятие вершины со стека. Первая часть леса разбора T1 хранится на ребре стека. Вторая часть T2 построена по только что разобранному правилу. Каждая из этих частей соответствует какому-то подпути в графе и необходимо показать, что правая координата T1 совпадает с левой координатой T2. Это соответствует тому факту, что объединение этих частей леса даст часть леса, покрывающую путь в графе без "дыр", то есть если в графе была цепочка *"abcd"* и T1 соответствует *"ab"*, то T2 будет соответствовать *"bc"*. Для того, чтобы показать, что это условие выполняется, достаточно рассмотреть, как происходит процесс разбора. Как только в процессе обхода грамматики (в слоте) встречается нетерминал, создаётся новая вершина стека, которая хранит в себе слот с позицией за этим нетерминалом, на ребре хранится уже построенная часть леса. Правая координата этой части

SPPF является номером вершины, с которой будет происходить дальнейший разбор, это число и записывается в новый дескриптор. Таким образом, после того, как нетерминал будет разобран до конца, будет создан новый упакованный узел, в котором в качестве левого потомка будет часть леса с ребра, а в качестве правого — нетерминальный узел, левая координата которого совпадает с правой координатой левой части леса, так как именно с того места и начался разбор этого нетерминала. □

Таким образом, для упакованных узлов в дереве доказано необходимое. Доказательство для остальных видов узлов проводится аналогично.

Теорема 2. *Любое дерево, извлечённое из SPPF, является корректным.*

Доказательство.

Рассмотрим произвольное извлечённое из SPPF дерево и докажем, что оно удовлетворяет определению. Первый и третий пункт определения корректного дерева следует из определения SPPF.

Второй пункт следует из Леммы 1. Необходимо только показать, что такой путь начинается в начальной вершине и заканчивается в конечной. Действительно, так как работа алгоритма может быть начата только из начальной вершины, то левой координатой для неё будет стартовая вершина. Результатом работы алгоритма является SPPF. Узел помечается как результирующий, если он помечен стартовым нетерминалом и его левая координата является стартовой вершиной, а правая — финальной. □

Теорема 3. *Пусть грамматика Γ порождает язык L. Тогда для каждого пути в графе p, соответствующего строке s из L, из SPPF может быть изъято корректное дерево.*

Доказательство.

Необходимо доказать, что SPPF содержит все корректные деревья вывода для всех корректных цепочек из входа. Как только процесс разбора начинается, в очередь дескрипто-

ров добавляются дескрипторы для всех альтернатив стартового правила, соответствующие терминалу во входном потоке. Аналогичная ситуация происходит, как только в грамматике встречается нетерминал. Рассматриваются все альтернативы нетерминала и добавляются те, по которым может быть продолжен синтаксический анализ в соответствии со входным символом. Это гарантирует, что все альтернативы в выводе будут рассмотрены. При этом во входном графе все пути, соответствующие входным цепочкам, тоже рассматриваются, так как переход по ребру осуществляется всегда, если оно продолжает корректный префикс.□

3.3 Анализ данных большого объёма

Одной из задач, сформулированных в данной работе, является использование предложенного алгоритма для анализа больших данных. Это востребовано, например, в задачах биоинформатики. Прежде чем формулировать задачу, следует ввести основные определения.

Исследование геномов является одной из распространённых задач биоинформатики. Информацию, содержащуюся в геноме, можно представить в виде последовательностей символов и в дальнейшем эти последовательности анализировать. Геномы извлекаются из ДНК и позволяют характеризовать тот или иной организм. Для этого из генома необходимо выделить определённые участки, позволяющие сделать выводы о его свойствах. Геном (последовательность ДНК) — строка в алфавите $\{A, C, G, T\}$, однозначно определяющая организм (или штамм), к которому она относится. Сборка — набор подстрок генома, длина которых на порядки меньше длины самого генома. Метагеномная сборка — смесь сборок нескольких геномов, то есть набор небольших подстрок нескольких геномов. Поскольку геном состоит из повторяющихся участков, то его можно представить в виде конечного автомата с последовательностями символов на рёбрах, который на практике часто представляется в виде графа Де Брауна [39].

Как упоминалось ранее, для решения задач, возникающих в биоинформатике, не нужно структурное представление вывода. Это значит, что дерево разбора, которое является результатом работы синтаксического анализатора, не нужно. Необходимо лишь ответить на вопрос: порождает ли входной автомат данную подстроку или нет, и вернуть координаты участка, на котором это происходит. При этом геном можно описать с помощью грамматики, т.е. про подцепочки, порождаемые входным конечным автоматом, известно, что они описываются некоторой грамматикой. Необходимо найти подавтоматы, принимающие цепочки, задаваемые некоторой грамматикой. Таким образом, предложенный алгоритм необходимо модифицировать таким образом, чтобы он решал данную задачу.

Для решения поставленной задачи не нужно строить лес разбора, поэтому от функций для его построения можно просто отказаться. Самое простое представление результата — набор путей. Однако для больших графов это может потребовать больших дополнительных расходов памяти. Чтобы этого избежать, можно предложить следующий подход: строить множество начальных и конечных вершин и контролировать длину путей. Это можно делать в процессе анализа, не накапливая дополнительной информации. Тогда после завершения работы можно будет выделить подграф, который, возможно, будет содержать лишние пути и потому потребуется его последующая обработка с накоплением путей. При этом извлечённые подграфы будут существенно меньше исходного графа и их повторная обработка не сильно скажется на производительности.

Таким образом, в местах, где раньше в алгоритме строились узлы дерева, теперь просто запоминаются координаты. Вместо хранения поддерева на рёбрах стека теперь хранится просто число — начало и конец подцепочки, созданной на момент создания вершины стека. Кроме координат начала и конца и длины можно ещё сохранять путь целиком. Для этого на рёбрах нужно просто сохранять цепочки, а не одно число.

4 Реализация

Предложенный алгоритм был реализован в рамках исследовательского проекта YaccConstructor. В данной главе описывается архитектура предложенного решения: основные модули и их взаимодействие. Кроме того, рассматриваются особенности практической реализации.

4.1 Архитектура предложенного решения

На основе предложенного алгоритма разработан новый модуль инструмента YaccConstructor, который является генератором в терминах, принятых в этом проекте. Это показано на рис. 10, где изображена архитектура инструмента YaccConstructor и цветом выделен реализованный модуль.

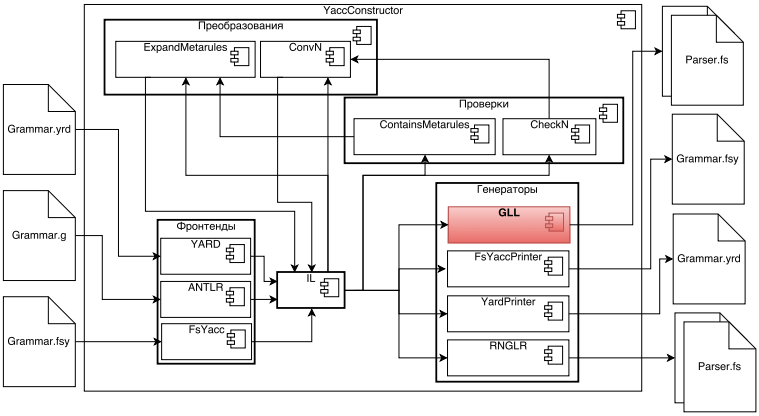


Рис. 10: Архитектура инструмента YaccConstructor (рисунок взят из работы [11])

Внутреннее устройство этого модуля показано на рис. 11. Основными компонентами являются генератор, который по

грамматике строит управляющие таблицы и дополнительные структуры данных, компонента с описанием SPPF и функциями работы с ним, два интерпретатора управляющих таблиц, различающиеся тем, что один из них строит лес разбора, а другой нет. Интерпретаторы разделены в силу того, что структуры для хранения элементов дерева тесно связаны с другими структурами, используемыми при анализе, например, стеком, а отказ от построения леса был вызван необходимостью получить алгоритм, расходующий меньше памяти. По этой причине реализовано два набора структур данных, каждая из которых оптимальна при решении соответствующей задачи.

На вход генератор принимает внутреннее представление в формате IL, которое строится по грамматике и может быть получено с помощью соответствующего фронтенда. Так как в язык описания грамматики позволяет использовать конструкции, которые не обрабатываются генератором (например, метаправила), то необходимо применить соответствующие преобразования, что достигается заданием специальных параметров при запуске инструмента. Результатом работы генератора является файл с исходным кодом, в котором описаны управляющие таблицы и вспомогательная информация, которая в дальнейшем используется интерпретатором.

Интерпретатор написан вручную и содержит в себе основную логику алгоритма. Он подключается в виде отдельной сборки к целевому приложению и позволяет на основе сгенерированнх данных выполнять анализ входа.

Пользователь при создании приложения, использующего модуль, добавляет в свой проект сгенерированный файл, ссылку на интерпретатор и файл, содержащий лексический анализатор (полученный с помощью другого модуля YC, который не описывается в данной работе) и вызывает соответствующую функцию для синтаксического анализа. Результатом работы такой функции является либо SPPF, либо набор координат во входном графе, позволяющих определить положение в нём

участка, порождающего строку, принимаемую соответствующей грамматикой.

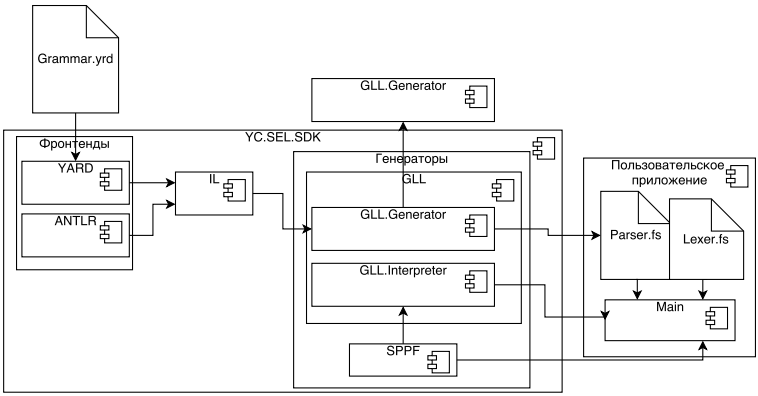


Рис. 11: Принцип работы реализованного модуля (рисунок взят и модифицирован из работы [11])

4.2 Особенности используемых структур данных

Алгоритм реализован в рамках проекта YaccConstructor на языке программирования F#. Исходный код свободно доступен в репозитории https://github.com/YaccConstructor/YaccConstructor, автор вёл разработку под учётной записью *AnastasiyaRagozina*.

Заявленная производительность алгоритма — в худшем случае куб по памяти и времени — обоснована теоретически [18]. На практике же, для достижения высокой производительности алгоритма, написанного с использованием языков высокого уровня, необходимо приложить некоторые усилия. Рассуждениям на данную тему и описанию эффективных структур данных посвящена

работа [19]. При реализации описанного алгоритма возникли похожие проблемы: высокий расход памяти и медленные структуры данных. Основной проблемой было хранение леса разбора и поиск уже существующих узлов. Хранение узлов в многомерных массивах, как было предложено в [19], накладывало значительные ограничения на длину входа. Кроме того, хранение в каждом узле дерева нескольких чисел (имени нетерминала и координаты начала и конца подцепочки, соответствующей данному поддереву) делало его громоздким. В результате для того, чтобы уменьшить расход памяти при хранении SPPF, было использовано сжатие хранимых в узлах координат в одно число. Это позволило вместо хранения двух чисел хранить одно, которое можно было использовать в качестве ключа при поиске уже созданных поддеревьев. Аналогичное сжатие использовалось для хранения слотов. Для хранения терминальных узлов в алгоритме было предложено использовать динамически изменяемый массив, размер которого сравним с размером входных данных, что приводит к выделению большого количества лишней памяти при использовании стандартного типа ResizeArray<_> при больших размерах входа. Для решения этой проблемы использовалась модификация динамически изменяемого массива, в которой память выделяется блоками константного размера. Данная структура данных была реализована в рамках работы над RNGLR-алгоритмом. В рамках данной работы она была выделена в библиотеку структур данных FSharpx.Collections, поддерживаемую FSharp-сообществом [26]. Подобные задачи являются интересными с инженерной точки зрения и часто возникают на практике.

Важной задачей также является представление метагеномной сборки и её обработка, так как, в отличие от графа, являющегося аппроксимацией встроенных языков, граф, представляющий метагеномную сборку, как правило, существенно большего размера. Для того, чтобы уменьшить

размер самого графа, на рёбрах хранится не по одному токену, а цепочки токенов. Это приводит к тому, что теперь в качестве координаты начала и конца подстроки используется не два числа, а четыре: номера концов рёбер и позиция на них. По аналогии, эти числа сжимались. Для того, чтобы эффективно использовать такие индексы, была создана структура данных, доступ к элементам которой как у массива, но по сжатому числу.

При обработке графа метагеномной сборки были использованы следующие знания о его структуре и особенностях решаемой задачи. В графе есть рёбра, на которых лежат подстроки длины большей, чем длина искомой подстроки. Это означает, что такие рёбра можно удалить из графа и обработать отдельно, как линейные данные. При этом граф распадается на набор связанных компонент, что позволяет обрабатывать части входного графа полностью независимо. Это, в свою очередь, существенно упрощает параллельную обработку данных: возникает классическая параллельность по данным, когда к большому количеству независимых данных нужно применить одну и ту же функцию обработки.

Однако, несмотря на то, что параллельность по данным может быть реализована очевидным образом, использование нескольких потоков в рамках одного многоядерного процессора не даёт ожидаемого прироста производительности, что наглядно продемонстрировано на рис. 12.

Замеры, результаты которых представлены на рис. 12 и рис. 13 проводились на машине со следующей конфигурацией:

- OS Name Microsoft Windows 10 Pro;
- System Type x64-based PC;
- Processor Intel(R) Core(TM) i7-4790 CPU 3.60GHz, 3601 Mhz, 4 Core(s), 4 Logical Processor(s);
- Installed Physical Memory (RAM) 32.0 GB.

Из рис. 12 видно, что максимальная производительность наблюдается при использовании двух потоков. Однако

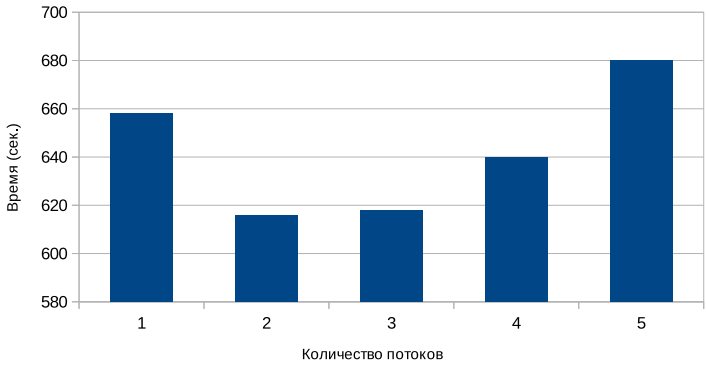


Рис. 12: Сравнение производительности предложенного решения при запуске на нескольких потоках

прирост производительности по сравнению с использованием одного потока составляет всего 6.4%, что значительно меньше теоретически возможного.

Такое поведение системы связано с тем, что при обработке одного графа происходит активное обращение к вспомогательным структурам данных большого объёма. При этом обращения, в силу особенностей алгоритма, плохо локализованы. В связи с чем, при попытке обработать несколько графов одновременно на одном процессоре, учащаются промахи кэшей. Частично решить эту проблему удалось заменив очередь дескрипторов на стек, это сделало обращения к данным более локализованными и позволило улучшить производительность решения.

Результаты измерений после замены очереди на стек представлены на рис. 13. Максимальная производительность достигается при использовании трёх потоков и прирост производительность составляет 14.2%. На рис. 13

представлено сравнение производительности до и после модификации.

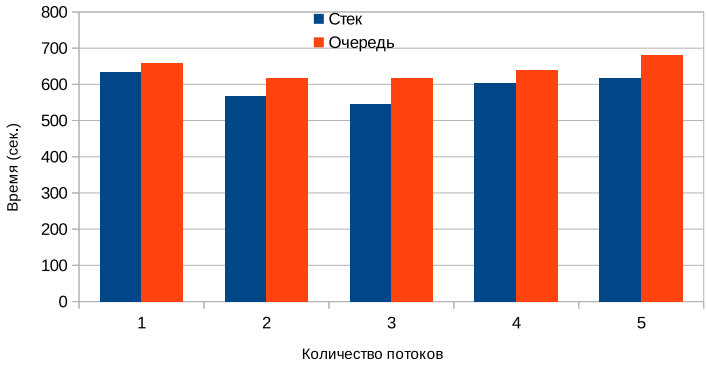


Рис. 13: Сравнение производительности предложенного решения замене стека на очередь для хранения дескрипторов

Для того, чтобы избавиться от проблем с кэшами при многопоточной обработке, можно использовать многопроцессорные системы, такие как вычислительные кластеры. При этом, как показали проведённые ранее эксперименты, имеет смысл запускать не более двух потоков на одном процессоре. Так как время обработки одного подграфа занимает время порядка нескольких секунд, то затраты на передачу по сети не должны заметно уменьшать выигрыш, получаемый за счёт параллельной обработки при достаточном количестве графов для обработки на одном узле.

Для реализации вычислений в кластере была выбрана технология MBrace, которое позволяет, с одной стороны, управлять кластером в облаке Microsoft.Azure с помощью скриптов на F#. Предоставляется полный набор функций,

позволяющий сконфигурировать кластер “с нуля”, а затем управлять им (например, изменять количество машин). С другой стороны, MBrace позволяет прозрачно использовать кластер в коде на F#. Это достигается благодаря предоставлению набора высокоуровневых функций и окружения cloud, благодаря которому код, предназначенный для выполнения в кластере можно задать следующим образом.

Listing 9 Код для запуска предложенного решения в кластере

```
let parallelTask =
    [ for i in 1 .. 10 ->
          cloud { return sprintf "i'm work item %d" i } ]
    |> Cloud.Parallel
    |> cluster.CreateProcess
```

В скобках cloud { } может находиться произвольный код на F#. Все необходимые для выполнения этого кода в кластере дополнительные действия и коммуникации (передача данных, подготовка и передача бинарных файлов) осуществляется автоматически и не требует участия разработчика. Таким образом, в предположении, что функция обработки графа processGraph реализована, всё, что необходимо для реализации обработки массива графов о кластере, это “завернуть” её вызов в окружение cloud следующим образом.

5 Эксперименты

В рамках данной работы были произведены эксперименты по сравнению алгоритмов синтаксического анализа регулярных множеств на основе алгоритма RNGLR и GLL.

Listing 10 Код для запуска предложенного решения в кластере с параметризацией входных данных

```
let parallelGraphProcessing graphs =
    [ for g in graphs -> cloud { return processGraph g } ]
    |> Cloud.Parallel
    |> cluster.CreateProcess
```

Замеры производились на компьютере со следующими характеристиками:

- Операционная система: Microsoft Windows 8.1 Pro;
- Тип системы: x64-based PC;
- Процессор: Intel(R) Core(TM) i7-4790 CPU 3.60GHz, 3601 Mhz, 4 Core(s), 8 Logical Processor(s);
- Объём оперативной памяти: 16.0 GB.

Для сравнения были выбраны несколько грамматик, представляющих как практический, так и теоретический интерес. Одна из таких грамматик — сильно неоднозначная грамматика G_5 (листинг 11), реализующая худший случай для анализатора, и позволяющая оценить производительность алгоритмов в задачах биоинформатики, так как большинство шаблонов в них являются сильно неоднозначными. Результаты измерений представлены на рис. 14.

Listing 11 Грамматика G_5

$$s \to s\ s\ s \mid s\ s \mid B$$

Следующей рассматриваемой грамматикой является неоднозначная грамматика правильных скобочных последовательностей G_6 (листинг 12). Данная грамматика представляет интерес с практический точки зрения, так как

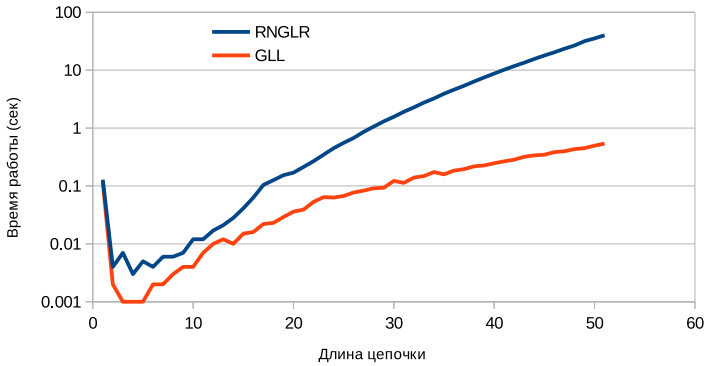


Рис. 14: Сравнение времени работы решений на основе алгоритмов GLL и RNGLR для грамматики G_5

она близка к шаблонам для поиска, используемым в задачах биоинформатики.

Listing 12 Грамматика G_6

$$s \rightarrow s\ s \mid LBR\ s\ RBR \mid \varepsilon$$

Результаты измерений представлены на рис. 15. Видно, что время работы алгоритма на основе RNGLR растёт значительно быстрее, чем алгоритма на основе GLL, с ростом длины входной цепочки.

Следующий эксперимент производился на грамматике подмножества языка T-SQL [35]. Входные графы строились с помощью последовательной конкатенации блоков с параллельными путями, которые соответствуют

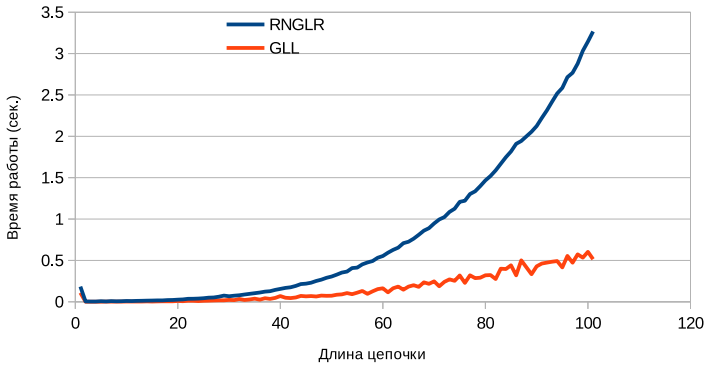


Рис. 15: Сравнение времени работы решений на основе алгоритмов GLL и RNGLR для грамматики G_6

использованию операторов ветвления при построении выражения. Пример входного графа приведён на рис. 16.

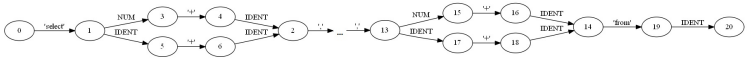


Рис. 16: Структура графа для экспериментов на грамматике T-SQL

Результаты измерений приведены на рис. 17 В данном случае GLL оказался медленнее. Однако необходимо заметить, что, с одной стороны, на входах практически значимой длины замедление несущественно, а с другой, алгоритм на основе GLR длительное время оптимизировался. Техническая доработка GLL может позволить улучшить его производительность.

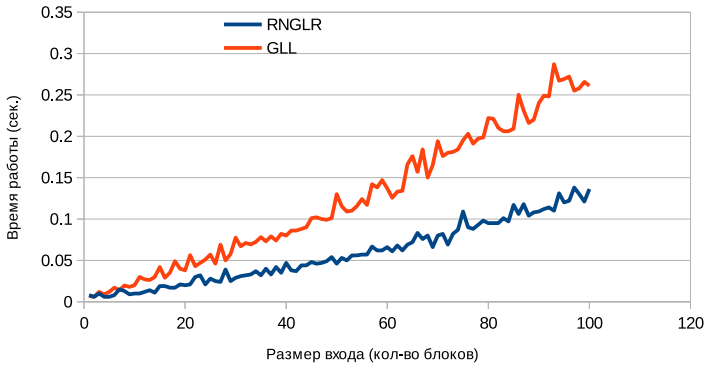


Рис. 17: Сравнение времени работы решений на основе алгоритмов GLL и RNGLR для грамматики T-SQL

Также было проведено сравнение алгоритмов на основе RNGLR и GLL на задаче поиска подцепочки в геноме. В качестве искомой подцепочки была выбрана транспортная РНК, шаблон для поиска которой был описан грамматикой, представленной на листинге 13. Данная грамматика является сильно неоднозначной. В качестве входа была выбрана последовательность из тестов инструмента Infernal [10], от которой последовательно брались участки длины $100 + 10 * k$, начинающиеся с нулевой позиции. Таким образом был получен набор тестовых данных: множество цепочек с увеличивающейся длиной.

График зависимости времени работы от длины цепочки приведён на рис. 18. Для RNGLR приведены первые 10 замеров, так как дальнейшие измерения для алгоритма на основе RNGLR было решено прекратить из-за его низкой производительности. Приведённые результаты показывают, что реализованный в рамках данной работы алгоритм более

Listing 13 Пример грамматики для описания транспортной РНК

```
stem<s>:
      A stem<s> U | U stem<s> A
    | C stem<s> G | G stem<s> C
    | G stem<s> U | U stem<s> G
    | s

any: A | U | G | C

[<Start>]
full: folded any?

folded: stem<(any*[1..3] stem<any*[7..10]>
             any*[1..3] stem<any*[5..8]>
             any*[3..6]
             stem<any*[5..8]>)>
```

чем в 400 раз быстрее алгоритма на основе RNGLR, что согласуется с результатами сравнений на сильно неоднозначной грамматике.

Заключение

В данной работе получены следующие результаты:

- Разработан алгоритм синтаксического анализа динамически формируемого кода на основе алгоритма GLL, результатом работы которого является лес разбора, который компактно представляется с помощью структуры данных SPPF.
- Доказана корректность и завершаемость предложенного алгоритма.
- Предложенный алгоритм реализован на языке F# в виде модуля инструмента YaccConstructor. Исходный код

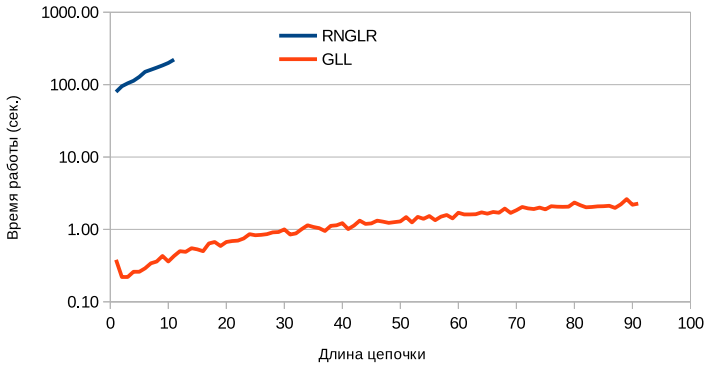


Рис. 18: Сравнение времени работы решений на основе алгоритмов GLL и RNGLR для грамматики на листинге 13

доступен в репозитории YaccConstructor [2], автор работал под учётной записью *AnastasiyaRagozina*.

– Проведён ряд экспериментов и выполнено сравнение с алгоритмом, реализующим аналогичный подход.
– Выполнена модификация предложенного алгоритма, позволяющая обрабатывать входные данные большого размера, что продемонстрировано на примере поиска подпоследовательностей в метагеномных сборках.

По результатам работы сделан доклад “Обобщённый табличный LL-анализ” на конференции “ТМПА-2014”, тезисы опубликованы в сборнике материалов конференции, и выполнена публикация “Средство разработки инструментов статического анализа встроенных языков” в сборнике “Наука и инновации в технических университетах материалы Восьмого Всероссийского форума студентов, аспирантов и молодых ученых”. Исследовательская работа поддержана грантом УМНИК: договор №5609ГУ1/2014.

Существует несколько направлений дальнейшего развития полученных результатов. Во-первых, важной задачей является оценка теоретической сложности представленного алгоритма. Во-вторых, необходимо исследовать возможности по непосредственной поддержке грамматик в EBNF и поддержке булевых грамматик. Использование булевых, или даже конъюнктивных, грамматик позволит более точно задавать критерии поиска, например, это позволит специфицировать высоту stem-а. Эта возможность продемонстрирована в листинге 14: правило stem_3_5<s> описывает stem высотой от 3 до 5 пар.

Listing 14 Пример конъюнктивной грамматики для описания stem-ов фиксированной высоты

```
1  stem<s>:
2        A stem<s> U
3      | U stem<s> A
4      | C stem<s> G
5      | G stem<s> C
6      | G stem<s> U
7      | U stem<s> G
8      | s
9
10 any: A | U | G | C
11 stem_3_5<s>: stem <s> & (any*[3..5] s any*[3..5])
```

Кроме этого, необходимо выполнить ряд технических доработок, таких как оптимизация реализации.

Список литературы

1. Doh K. G., Kim H., Schmidt D. A. Abstract Parsing: Static Analysis of Dynamically Generated String Output Using LR-Parsing Technology. Proceedings of the 16th International

Symposium on Static Analysis. — SAS '09. — Berlin, Heidelberg : Springer-Verlag, 2009. — P. 256–272.
2. Doh K. G., Kim H., Schmidt D. A. Abstract LR-parsing. Formal Modeling. Ed. by Gul Agha, José Meseguer, Olivier Danvy. — Berlin, Heidelberg : Springer-Verlag, 2011. — P. 90–109.
3. Doh K. G., Kim H., Schmidt D. A. Static Validation of Dynamically Generated HTML Documents Based on Abstract Parsing and Semantic Processing. Static Analysis. — Springer Berlin Heidelberg, 2013. — Vol. 7935 of Lecture Notes in Computer Science. — P. 194–214.
4. Anderson J. Novák Á. Sükösd Z. Quantifying variances in comparative RNA secondary structure prediction. BMC Bioinformatics. — 2013. — P. 14–149.
5. S. Grigorev, E. Verbitskaia, A. Ivanov et al. String-embedded Language Support in Integrated Development Environment. Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia. — CEE-SECR '14. — New York, NY, USA : ACM, 2014. — P. 21:1–21:11.
6. Syme D., Granicz A., Cisternino A. Expert F#. (Expert's Voice in .Net). — ISBN: 1590598504, 9781590598504.
7. A. Afroozeh, A. Izmaylova. Faster, Practical GLL Parsing. Compiler Construction. Springer. — 2015. — P. 89–108.
8. A. Annamaa, A. Breslav, J. Kabanov, V. Vene. An Interactive Tool for Analyzing Embedded SQL Queries. Proceedings of the 8th Asian Conference on Programming Languages and Systems. — APLAS'10. — Berlin, Heidelberg : Springer-Verlag, 2010. — P. 131–138.
9. Cousot P., Cousot R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. — POPL '77. — New York, NY, USA : ACM, 1977. — P. 238–252.
10. Tomita M. An Efficient Context-free Parsing Algorithm for Natural Languages. Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2. — IJCAI'85. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1985. — P. 756–764.
11. Rekers J. G. Parser Generation for Interactive Environments: Ph. D. thesis. J. G. Rekers ; Universiteit van Amsterdam. — 1992.

12. С. Григорьев. Синтаксический анализ динамически формируемых программ: Ph. D. thesis. Санкт-Петербургский государственный университет. — 2016.
13. Scott E., Johnstone A. GLL Parsing. Electron. Notes Theor. Comput. Sci. — 2010. — Vol. 253, no. 7. — P. 177–189.
14. Scott E., Johnstone A., Economopoulos R. BRNGLR: A Cubic Tomitastyle GLR Parsing Algorithm. Acta Inf. — 2007. — Vol. 44, no. 6. — P. 427–461.
15. Scott E., Johnstone A. Generalized Bottom Up Parsers With Reduced Stack Activity. Comput. J. — 2005. — Vol. 48, no. 5. — P. 565–587.
16. Annamaa A., Breslav A., Vene V. Using Abstract Lexical Analysis and Parsing to Detect Errors in String-Embedded DSL Statements. Proceedings of the 22nd Nordic Workshop on Programming Theory. — 2010. — P. 20–22.
17. Nguyen H., Kästner C., Nguyen T. Varis. IDE Support for Embedded Client Code in PHP Web Applications. Proceedings of the 37th International Conference on Software Engineering (ICSE). — New York, NY : ACM Press, 2015. — Formal Demonstration paper.
18. Johnstone A., Scott E. Principled software microengineering. Science of Computer Programming. — 2015. — Vol. 97, Part 1. — P. 64 – 68. — Special Issue on New Ideas and Emerging Results in Understanding Software.
19. Johnstone A., Scott E. Modelling GLL Parser Implementations. Software Language Engineering: Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers / Ed. by Brian Malloy, Steffen Staab, Mark van den Brand. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2011. — P. 42–61. — ISBN: 978-3-642-19440-5.
20. C. Yuan, J. Lei, J. Cole, Y. Sun. Reconstructing 16S rRNA genes in metagenomic data. Bioinformatics. — 2015. — Vol. 31, no. 12. — P. i35–i43.
21. Nawrocki E., Eddy S. Infernal 1.1: 100-fold faster RNA homology searches. Bioinformatics. — 2013. — Vol. 29, no. 22. — P. 2933–2935.
22. Miller C. Baker B. Thomas B. Singer S. Banfield J. EMIRGE: reconstruction of full-length ribosomal genes from microbial community short read sequencing data. Genome Biology. — 2011. — Vol. 12, no. 5. — P. 1–14.

23. Qiong Wang, Jordan A. Fish, Mariah Gilman et al. Xander: employing a novel method for efficient gene-targeted metagenomic assembly. Microbiome. — 2015. — Vol. 3, no. 1. — P. 1–13.
24. ANTLR [Электронный ресурс]. http://www.antlr.org/ urldate = 11.05.2016
25. Alvor [Электронный ресурс]. http://code.google.com/p/alvor/ urldate = 11.05.2016
26. Репозиторий FSharpx.Collections [Электронный ресурс]. https://github.com/fsprojects/FSharpx.Collections/ urldate = 13.05.2016
27. PHP String Analyzer [Электронный ресурс]. http://www.score.cs.tsukuba.ac.jp/ minamide/phpsa/ urldate = 11.05.2016
28. Java String Analyzer [Электронный ресурс]. http://www.brics.dk/JSA/ urldate = 11.05.2016
29. "YaccConstructor [Электронный ресурс]"https://github.com/YaccConstructor/ urldate = 11.05.2015
30. HMMER [Электронный ресурс] http://hmmer.org/ urldate = 14.05.2016
31. IntelliLang [Электронный ресурс] https://www.jetbrains.com/idea/help/intellilang.html urldate = 11.05.2016
32. Christensen A. S., Møller A., Schwartzbach M. I. Precise Analysis of String Expressions. Proc. 10th International Static Analysis Symposium (SAS). — Vol. 2694 of LNCS. — Springer-Verlag, 2003. — P. 1–18.
33. Minamide Y. Static Approximation of Dynamically Generated Web Pages. Proceedings of the 14th International Conference on World Wide Web. — WWW '05. — New York, NY, USA : ACM, 2005. — P. 432–441.
34. Полубелова М. Лексический анализ динамически формируемых строковых выражений. Санкт-Петербургский государственный университет. — 2015.
35. Репозиторий проекта YaccConstructor, содержащий набор различных грамматик [Электронный ресурс]. https://github.com/YaccConstructor/YC.GrammarZOO urldate = 29.04.2016
36. Verbitskaia E., Grigorev S., Avdyukhin D. Relaxed Parsing of Regular Approximations of String-Embedded Languages.

Preliminary Proceedings of the PSI 2015: 10th International Andrei Ershov Memorial Conference. — PSI'15. — 2015. — P. 1–12.
37. R. Durbin, S. Eddy, A. Krogh, G. Mitchison. Biological sequence analysis: probabilistic models of proteins and nucleic acids. Cambridge university press, 1998.
38. Nawrocki E., Eddy S. Computational identification of functional RNA homologs in metagenomic data. RNA biology. — 2013. — Vol. 10, no. 7. — P. 1170–1179.
39. Compeau P., Pevzner P., Tesler G. How to apply de Bruijn graphs to genome assembly. Nature biotechnology. — 2011. — Vol. 29, no. 11. — P. 987–991.

Библиотека параллельной сборки мусора для C++

Моисеенко Евгений Александрович

Санкт-Петербургский государственный университет
evg.moiseenko@mail.ru

Аннотация В работе описывается алгоритм сборки мусора для языка C++, реализованный в виде библиотеки, не требующей поддержки со стороны компилятора. Реализованный сборщик мусора является *точным*, т.е. способным точно идентифицировать указатели на управляемые объекты, *инкрементальным*, *копирующим*, т.е. способным улучшать локальность памяти за счёт перемещения объектов, *потокобезопасным* и *параллельным*. В работе также представлен обзор существующих решений задачи сборки мусора для языка C++, и проведена апробация результата.

Введение

C++ является популярным языком общего назначения, разработанным в 1980-х годах как улучшение широко используемого языка C. Главным новшеством языка было введение объектно-ориентированных конструкций, позволивших моделировать предметную область программы на языке объектов. С самого начала C++ разрабатывался с учётом высоких требований к производительности таким образом, чтобы введение новых языковых конструкций не нарушало обратную совместимость и не накладывало дополнительных накладых расходов (принцип “don’t pay for

what you don't use"). C++ используется в приложениях, где критичными являются требования ко времени выполнения, потребляемой памяти, размеру исполняемых файлов. Среди таких приложений можно выделить финансовые и банковские системы (high-frequency trading), приложения, активно работающие с графикой (графические редакторы, системы виртуальной реальности, компьютерные игры), программы, работающие на встраиваемых платформах (embedded systems)[1].

Одним из основных ресурсов приложения является память. Большинство современных языков программирования активно использует *динамическое распределение памяти*, при котором выделение памяти осуществляется во время исполнения программы. Динамическое управление памятью вводит два основных примитива — функции выделения и освобождения блоков памяти. Существуют два способа управления динамической памятью: *ручное* и *автоматическое*. При *ручном управлении памятью* программист должен следить за освобождением выделенной памяти, что приводит к возможности возникновения труднообнаружимых ошибок. Более того, в некоторых ситуациях (например, при программировании на функциональных языках или в многопоточной среде) время жизни объекта не всегда очевидно для разработчика. *Автоматическое управление памятью* избавляет программиста от необходимости вручную освобождать выделенную память, устраняя тем самым целый класс возможных ошибок и увеличивая безопасность исходного кода программы. *Сборка мусора* (garbage collection) давно стала стандартом в области автоматического управления памятью, хотя её использование может накладывать дополнительные расходы по памяти и времени исполнения.

[1] С обзором современного использования C++ можно ознакомиться здесь: http://blog.jetbrains.com/clion/2015/07/infographics-cpp-facts-before-clion/

На сегодняшний день среды времени выполнения многих популярных языков программирования, таких как Java, C#, Python, Ruby и др., активно используют сборку мусора. Язык C++ разрабатывался с расчётом на использование ручного управления памятью. Некоторые языковые возможности, такие как адресная арифметика и приведение типов указателей, затрудняют реализацию сборщиков мусора для этого языка. Несмотря на это, существует ряд подходов к разработке сборщиков мусора для языка C++.

В рамках проекта лаборатории JetBrains была реализована сборка мусора для языка C++ [1–3] на уровне библиотеки. Сборщик приостанавливает работу приложения на всё время сборки мусора, из-за чего в работе программы могут возникать длительные паузы. Продолжительные остановки на сборку мусора неприемлимы для целого класса приложений, для которых важна низкая латентность, то есть малое время отклика на запросы пользователей. Существуют подходы, призванные уменьшить время паузы на сборку мусора, в частности, инкрементальная параллельная маркировка. Отличительной особенностью данного подхода является выполнение части работы по сборке мусора параллельно с приложением, без его полной остановки.

1 Постановка задачи

В данной работе были поставлены следующие задачи:

- проанализировать ограничения и недостатки текущей версии библиотеки сборки мусора;
- предложить и реализовать новые методы для модернизации библиотеки;
- реализовать и встроить в библиотеку алгоритм инкрементальной параллельной маркировки;
- протестировать функциональность и производительность новой версии библиотеки.

2 Обзор

Существуют различные подходы к автоматическому управлению памятью, каждый из которых имеет свои преимущества и недостатки [4, 5]. В данной главе будут рассмотрены основные алгоритмы сборки мусора, а также подходы к автоматическому управлению памятью, актуальные в контексте C++. Кроме того, представлен обзор существующих решений задачи сборки мусора для языка C++.

2.1 Обзор алгоритмов сборки мусора

Различают два принципиально различных подхода к автоматическому управлению памятью: *подсчёт ссылок* (*reference counting*) и *трассирующая сборка мусора* (*tracing garbage collection*). Существует множество различных разновидностей трассирующей сборки мусора, мы рассмотрим две из них: *сжимающую* (*compacting garbage collection*) и *инкрементальную* (*incremental garbage collection*).

Подсчёт ссылок Одним из наиболее простых методов автоматического управления памятью является *подсчёт ссылок*. С каждым объектом, выделенным в куче, связывается целое число — *счётчик ссылок*. Изначально, значение счётчика инициализируется единицей. При копировании указателя на объект его счётчик увеличивается на единицу, а при удалении указателя — уменьшается на единицу. Когда значение счётчика падает до нуля, объект может быть удалён.

К достоинствам данного подхода можно отнести то, что он прост для понимания, легко реализуем, и, кроме того, момент освобождения памяти строго детерминирован — как только в программе не остаётся ссылок на определённый объект, занимаемая им память может быть освобождена и переиспользована. Однако данный подход обладает рядом недостатков.

1. Поддержка счётчика ссылок накладывает дополнительные накладные расходы на все операции с указателями.
2. Имеется вероятность *лавинного освобождения памяти*, т.е. ситуации, при которой уничтожение одной ссылки на "корень" структуры данных приводит к цепному освобождению целой области памяти, вызывая тем самым *непредсказуемую* по продолжительности паузу в процессе исполнения программы.
3. *Циклические структуры данных* (множества объектов, содержащих взаимные ссылки) не могут быть корректно обработаны без модификации алгоритма (наличие таких структур приводит к утечке памяти).

Ясно, что второй недостаток прямо связан с детерминированным моментом освобождения памяти, и попытка преодолеть его также приведёт и к потере этого преимущества. Для решения третьей проблемы были предложены различные модификации алгоритма, однако они либо достаточно сложны и вычислительно трудоёмки, либо перекладывают ответственность за разрешение циклических ссылок на программиста [4]. Например, может быть введён новый примитив — *невладеющий указатель* (weak pointer). Невладеющий указатель не участвует в подсчёте ссылок. Используя этот указатель для хранения ссылок, замыкающих цикл, можно предотвратить утечку памяти.

Трассирующая сборка мусора Принципиально другой подход к сборке мусора основан на использовании критерия доступности объекта. *Доступность* определяется индуктивно, а именно, область памяти называется *доступной*, если либо указатель на неё принадлежит корневому множеству, либо существует указатель на неё из другой области памяти, являющейся доступной. *Корневое множество* представляет собой множество априори доступных

объектов. Конкретное определение корневого множества зависит от языка программирования и среды времени выполнения. Зачастую, корневым множеством является множество указателей, расположенных в регистрах, на стеке и в статической области памяти. Все объекты, которые не являются доступными, объявляются *мусором*. Сборщики мусора, использующие данный подход, называются *трассирующими*. Инвариантом данного подхода является то, что память, занимаемая мусором, в любой момент может быть безопасно освобождена. Тем самым доступность является эвристическим приближением понятия *используемости*. Заметим, что проблема используемости памяти неразрешима[2].

Базовым алгоритмом трассирующий сборки мусора является алгоритм *пометить и освободить* (*mark-and-sweep*). Сборщик мусора обходит граф достижимых объектов, начиная с корневого множества, помечая все достижимые объекты как живые (для хранения метки с каждым объектом связывается специальный бит). Затем, память из-под всех непомеченных объектов освобождается. Компонент, выполняющий пометку и освобождение, называют *сборщиком*, а компонент, модифицирующий память и запрашивающий её выделение, — *мутатором*.

Ввиду того, что мутатор может модифицировать объекты и менять структуру графа достижимых объектов, перед запуском маркировки и освобождения он должнен быть остановлен. Алгоритмы сборки мусора, требующие полной остановки работы приложения на время своей работы, известны как *алгоритмы с полной остановкой мира* (*stop-the-world algorithms*).

---

[2] Может быть показано, что проблема используемости памяти сводится к проблеме остановки:
https://en.wikipedia.org/wiki/Tracing_garbage_collection#Reachability_of_an_object

Трассирующие сборщики мусора можно разделить на два класса:

- *точные* (*precise*/*accurate*) сборщики мусора, которые способны обнаружить все недоступные объекты;
- *консервативные* (*conservative*) — сборщики мусора, не являющиеся точными.

Для точной трассирующей сборки мусора необходимо выполнение следуюших условий:

- возможность построения корневого множества;
- возможность определить все указатели из любого объекта на другие области управляемой памяти.

Часто, в силу особенности языка программирования, устройства среды времени выполнения или по другим специфическим причинам, выполнение этих условий невозможно. В таком случае, без наложения дополнительных ограничений на использование языковых средств, возможна только консервативная сборка мусора. При этом используются различные эвристические методы построения корневого множества или идентификации ячейки памяти как указателя. Это приводит к тому, что для каждого консервативного сборщика мусора существует целый класс программ, на которых выбранные эвристики работают недостаточно хорошо [9]. Следствием этого могут стать длительные задержки в работе сборщика или же полное исчерпание памяти.

Сжимающая сборка мусора *Сжимающая сборка* (*compacting garbage collection*) — это разновидность трассирующей сборки, предназначенная для решения проблемы *фрагментации памяти*. *Фрагментацией* называется ситуация, при которой в куче выделено большое число блоков, не образующих непрерывную область памяти. В таком случае попытка выделения объекта большого

размера может закончиться неудачей, несмотря на то, что суммарного количества свободной памяти вполне достаточно для удовлетворения запроса.

Алгоритм сжимающей сборки *"пометить и сжать"* (*mark-and-compact*) после этапа маркировки запускает процедуру сжатия кучи. Все живые объекты перемещаются на место непереживших сборку, образуя тем самым непрерывные области памяти, а программные указатели на них перенаправляются на новое расположение объектов (*pointer forwarding*).

Для сжатия кучи могут применяться различные алгоритмы. Базовым алгоритмом сжатия кучи является *двухпальцевый алгоритм* (*two-finger-compact*). Мы рассмотрим одну из модификаций двухпальцевого алгоритма, которая может применяться для сжатия области памяти, содержащей объекты фиксированного размера. Заметим, что существуют и другие вариации алгоритма, позволяющие сжимать кучу с объектами произвольного размера, но мы не рассматриваем их, так как в текущей версии библиотеки они не применяются. Алгоритм поддерживает два указателя: указатель на свободный объект и указатель на перемещаемый объект. Первый указатель инициализируется адресом начала сжимаемой области памяти, а второй — адресом конца. Затем выполняется поиск очередного свободного блока (первым указателем) и очередного перемещаемого объекта (вторым указателем). Память, занимаемая перемещаемым объектом, копируется в свободный блок, а на её место записывается так называемый *перемещающий указатель* (*forward pointer*), указатель на новое расположение объекта. Перемещающий указатель затем используется в фазе модификации указателей на перемещённые объекты. Алгоритм повторяет вышеописанный процесс, пока указатель на свободный блок меньше указателя на перемещаемый объект.

Инкрементальная параллельная маркировка Алгоритмы, позволяющие чередовать работу коллектора и мутатора называются *инкрементальными*. Инкрементальные алгоритмы разбивают цикл сборки мусора на несколько частей, тем самым размывая паузы, вызываемые работой сборщика мусора, по исполнению программы. При этом снижается время каждой конкретной паузы мутатора, и уменьшаются задержки в работе приложения, вызванные остановкой на сборку мусора. Однако инкрементальная сборка, как правило, привносит накладные расходы, так что суммарное время работы приложения может увеличиться по сравнению с подходами с полной остановкой мутатора. Алгоритмы сборки, способные выполнять большую часть своей работы (часто, всю работу лишь за исключением сканирования корневого множества) по сборке мусора одновременно с работой мутатора, называются *почти параллельными* (*mostly-concurrent*).

Далее мы рассмотрим модификацию алгоритма “пометить и сжать”, которая позволяет производить маркировку без остановки мутатора, остановка мира необходима только для сканирования корневого множества, а также на этапе сжатия кучи. Чтобы рассмотреть такую модификацию, введём несколько определений. Разделим все объекты на три класса:

- *белые* — объекты, которые еще не были сканированы сборщиком мусора;
- *серые* — объекты, которые были помечены сборщиком, но не все поля которого были просканированы;
- *чёрные* — объекты, которые были помечены сборщиком, и все их поля были просканированы.

При инкрементальной маркировке может возникнуть ситуация, в которой чёрный объект будет содержать ссылку на белый. После завершения маркировки белый объект будет освобождён, а чёрный будет хранить некорректный указатель. Такая ситуация может возникнуть при выполнении одновременно двух условий:

– мутатор сохраняет указатель на белый объект в чёрном;
– не существует серого объекта, такого, что существует путь от него до рассматриваемого белого объекта.

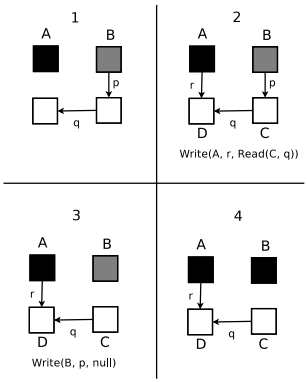


Рис. 1: Пример ошибочной ситуации

Рассмотрим конкретный пример. На Рис. 1 показано, как последовательность действий сборщика и мутатора может привести к подобному некорректному состоянию кучи. В начальном состоянии куча содержит четыре объекта — чёрный объект A, серый объект B и два белых объекта C и D. При этом имеются ссылки p и q из объекта B в C и из C в D, соответственно. Затем мутатор создаёт ещё одну ссылку r — из A в D. Хотя теперь появилась ссылка из чёрного объекта в белый (первое условие выполнено), объект D всё ещё достижим из серого объекта B (второе условие нарушено), следовательно рано или поздно объект D будет просканирован сборщиком. Однако затем мутатор удаляет ссылку из B в C, что приводит к выполнению сразу двух условий, и возникновению некорректного состояния.

Для предотвращения ошибочной ситуации сборщик и мутатор должны поддерживать один из двух инвариантов.

1. *Слабый трёхцветный инвариант* (*weak tricolor invariant*): для любого белого объекта, на который указывает чёрный объект, существует серый объект, такой, что существует путь от этого серого объекта до рассматриваемого белого.
2. *Сильный трёхцветный инвариант* (*strong tricolor invariant*): не существует чёрных объектов, указывающих на белые.

Для поддержки инвариантов могут быть использованы *барьеры чтения* или *барьеры записи*. Барьером чтения называется перехват чтения полей белого объекта, барьером записи — прехват записи ссылок на белый объект в чёрный. Заметим, что последовательность инструкций, реализующая барьер, выполняется при каждой операции чтения/записи во время фазы маркировки объектов, накладывая тем самым дополнительные расходы на время выполнения работы операций с указателями. Барьеры могут быть реализованы либо программно, либо с помощью аппаратной поддержки, либо с использованием функций операционной системы. Далее мы будем рассматривать только барьер записи, так как именно он используется в нашей работе. Этот выбор обоснован тем, что операции записи, как правило, выполняются реже, следовательно, и барьер записи вызывается реже, а значит и накладные расходы меньше. Существуют различные реализации барьеров записи, каждая из которых поддерживает один из двух упомянутых выше инвариантов. Приведём псевдокод барьера записи, впервые предложенного в работе [6]:

Listing 2.1: Dijkstra Barier

```
Write(src, field, ptr):
    src->field = ptr
    Shade(ptr)

Shade(ptr):
    if white(ptr)
        colour(ptr) = grey
```

Заметим, что для корректной работы сборщика мусора необходимо, чтобы обе операции в функции *Write* были атомарны.

2.2 Сборка мусора в C++

Как уже отмечалось, язык C++ разрабатывался с расчётом на использование ручного управления памятью. Поддержка сборки мусора не была изначально добавлена разработчиками языка и по сей день не включена в стандарт[3]. В [7] было упомянуто три подхода для добавления сборки мусора в язык, подробно описанных ниже.

1. Подход, основанный на *умных указателях* (*smart pointer*) — объектах специального класса, оборачивающих сырые (*raw pointer*) указатели и добавляющие к ним некоторую функциональность. Зачастую, умные указатели используются для реализации подсчёта ссылок, однако могут быть реализованы и другие алгоритмы автоматического управления памятью.
2. Подход, основанный на введении нового типа данных для хранения указателей на управляемые объекты. Экземпляры этого нового типа имеют

[3] ISO/IEC, Working Draft, Standard for Programming Language C++, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf

более бедный интерфейс по сравнению с обычными указателями, например, запрещена адресная арифметика. Существенное отличие от предыдущего подхода — использование поддержки со стороны компилятора. Хорошим примером реализации данного подхода служит нестандартное расширение C++/CLI[4], разработанное компанией Microsoft.

3. *Прозрачная* (*transparent*) сборка мусора. Этот подход подразумевает использование обычных указателей с возможностью переиспользования памяти. Примером реализации прозрачной сборки является сборщик мусора Бёма-Демерса-Вайзера[5].

В языках C/C++ указатели хранятся в памяти как целые числа. Иными словами, представление указателей в памяти никак не отличается от представления других примитивных типов данных. Из сказанного выше ясно, что без наложения дополнительных ограничений или введения новых типов данных для хранения указателей, точная сборка невозможна в C/C++. Поэтому, третий подход подразумевает использование консервативного сборщика. Также стоит отметить, что в рамках третьего подхода затруднительно реализовать сжимающий сборщик мусора.

2.3 Существующие решения

Рассмотрим более подробно существующие решения для автоматического управления памятью в C++.

Шаблонный класс shared_ptr<T> из стандартной библиотеки C++–11 — умный указатель, реализующий алгоритм подсчёта ссылок для автоматического управления памятью. *Умный указатель* (*smart pointer*) — это

[4] C++/CLI Language Specification, http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-372.pdf

[5] Официальный сайт проекта: http://www.hboehm.info/gc/

класс–декоратор, добавляющий некоторую функциональность к обычному интерфейсу указателей. Умные указатели используют целый ряд языковых возможностей C++ (в том числе новые возможности, добавленные в стандарте C++11), такие как переопределение операторов, семантика перемещения (move semantics) и другие, для предоставления удобного и интуитивно понятного интерфейса. Также, как правило, используется идиома RAII (Resource Acquisition Is Initialization). В самом общем смысле, использование этой идиомы позволяет связывать некоторые операции с конструированием и уничтожением объекта. Для этого создается специальный класс–обёртка над каким-либо ресурсом, в конструкторе этого класса вызываются функции, связанные с инициализацией ресурса, а в деструкторе — с освобождением ресурса. Например, конструктор shared_ptr<T> инициализирует контрольный блок (специальный объект, хранящий счётчик ссылок и указатель на функцию зачистки), а деструктор выполняет проверку счётчика ссылок и при необходимости освобождает память, занятую объектом и его контрольным блоком.

Как отмечалось в разделе 2.1, один из главных недостатков наивного подсчёта ссылок — некорректная обработка циклических ссылок. Для преодоления этой проблемы программисту предлагается использовать объекты класса weak_ptr<T> — невладеющие указатели. Перекладывание задачи по обработке циклических ссылок на пользователя библиотеки приводит к возможности возникновения в программе труднообнаружимых ошибок.

Расширение C++/CLI было разработано компанией Microsoft для возможности интеграции программ на C++ с программной платформой .Net. .Net использует общеязыковую среду исполнения (Common Language Runtime), управление памятью в которой происходит при помощи трассирующего сборщика мусора. Для того чтобы обеспечить совместимость C++ с CLR в язык

необходимо было добавить поддержку управляемых объектов, что и было сделано. Для хранения указателей на управляемые объекты был введён новый тип данных T^, который, однако, предоставляет более бедный интерфейс по сравнению с обычными указателями. Подход C++/CLI позволяет совмещать ручное и автоматическое управление памятью, сборка мусора является точной и сжимающей. Но скомпилировать программы, написанные на C++/CLI, можно только специальным компилятором, поддерживающим нестандартные расширения языка.

Среди консервативных сборщиков мусора для C++ наиболее популярным является уже упоминавшийся сборщик BoehmGC. Одна из главных целей, которую преследовали разработчики этого сборщика — возможность использования сборки мусора в существующих проектах с минимальной модификацией исходного кода и минимальными ограничениями на использование языковых возможностей. BoehmGC использует алгоритм “пометить и освободить”. Для выделения памяти программист должен использовать функцию GC_malloc, которая сохраняет необходимую для сборщика метаинформацию. BoehmGC используется в таких проектах, как Mono, Portable.NET, GNU Compiler for Java и других. Также он может быть использован для поиска утечек памяти, и в таком качестве он применяется в продуктах компании Mozilla. BoehmGC не требует поддержки от компилятора. К его недостаткам можно отнести консервативность и отсутствие поддержки сжимающей сборки мусора.

3 Анализ предыдущей версии библиотеки

В рамках проекта лаборатории JetBrains была реализована библиотека сборки мусора для языка C++ [1–3]. Реализованный сборщик мусора является почти точным, копирующим, не требует поддержки от компилятора и

позволяет совмещать ручное и автоматическое управление памятью. Используется модификация алгоритма “пометить и сжать”, мутатор останавливается на время маркировки и сжатия. От пользователя библиотеки ожидается выполнение некоторых соглашений, необходимых для корректной работы сборщика. Библиотека может быть скомпилирована только компилятором gcc для 64 битной архитектуры и операционной системы Linux, однако в дальнейшем возможен перенос на другие платформы, который потребует переработки лишь небольшой части платформозависимого кода.

Далее в данной главе будет приведён детальный обзор реализации предыдущей версии библиотеки, будут рассмотрены интерфейс библиотеки, реализация корневого множества и кучи, структура метаинформации, необходимой для сборки мусора, используемый алгоритм остановки мира и механизм взаимодействия с сырыми указателями.

Интерфейс Библиотека основана на подходе, использующем умные указатели для поддержки автоматического управления памятью. Пользователю предоставляется шаблонный класс gc_ptr<T> для хранения умных указателей и шаблонная функция gc_new<T> для создания управляемый объектов в куче. Класс gc_ptr<T> определяет конструктор по умолчанию, конструктор от нулевого указателя (nullptr), конструктор копирования, оператор присваивания, оператор доступа к члену класса (operator->()), оператор разыменования (operator*()), оператор преобразования в тип bool (для проверки, является ли указатель нулевым), метод reset для обнуления указателя. Пример использования примитивов библиотеки приводится в листинге 2.2.

Корневое множество Корневым множеством полагается множество указателей на управляемые объекты на стеке и в статической памяти. Корневое множество хранится в памяти как список указателей на объекты gc_ptr, являющиеся

Listing 2.2: Пример использования примитивов библиотеки

```
struct Node {
    Node(const gc_ptr<Node>& left,
         const gc_ptr<Node>& right)
        : left_(left)
        , right_(right)
    {}

    gc_ptr<Node> left_;
    gc_ptr<Node> right_;
};
...
gc_ptr<Node> a, b;
...
gc_ptr<Node> c = gc_new<Node>(a, b);
```

корнями. Чтобы снизить издержки на синхронизацию, каждый поток имеет свой экземпляр списка указателей на корни. Корневое множество поддерживается в консистентном состоянии при помощи `gc_new` и конструктора `gc_ptr`. В конструкторе `gc_ptr` проверяется уровень вложенности вызовов `gc_new`, и если он равен нулю, то создаваемый указатель является корнем. Новые корни добавляются в голову списка, при удалении корня список также просматривается, начиная с головы. Это связано с тем, что в C++ порядок вызова деструкторов почти всегда обратен порядку вызова конструкторов (временные объекты могут нарушать это предположение), поэтому такая оптимизация позволяет в большинстве случаев быстро добавлять и удалять корни.

Метаинформация Для того, чтобы сборщик имел возможность построить граф достижимых объектов, каждый объект должен содержать дополнительную

метаинформацию, позволяющую определить, указатели на какие объекты он содержит. В нашей библиотеке используются два типа метаинформации: метаинформация класса и метаинформация объекта. Метаинформация класса хранит размер экземпляра данного класса и список смещений gc_ptr, содержащихся в экземпляре класса, относительно начала объекта. Метаинформация класса создается один раз при первом выделении объекта данного класса в управляемой куче. С каждым объектом, выделенным в управляемой куче, связана метаинформация объекта. Она хранится в куче сразу после самого объекта. Метаинформация объекта содержит указатель на метаинформацию класса, указатель на начало объекта и количество экземпляров класса внутри управляемого объекта[6] (для поддержки массивов). Метаинформация создается и поддерживается с помощью специального протокола взаимодействия gc_new и конструктора gc_ptr, подробное описание которого может быть найдено в работе [1].

Куча Для хранения управляемых объектов используется собственная реализация кучи. Для выделения объектов больших (больше 4096 байт) и маленьких размеров используются различные стратегии. Куча для объектов маленьких размеров представляет собой набор пулов (segregated storage[7]), каждый из которых обслуживает объекты определённого размера. Адресное пространство кучи разделено на страницы, по умолчанию размер страницы равен 4096 байт. Размеры объектов являются степенями двойки от 32 (минимальный размер управляемого объекта) до 4096 байт. Пул выделяет память блоками размера, кратного размеру страницы, с помощью системного вызова mmap. Затем

[6] Под объектом в данном случае понимается не экземпляр какого-либо класса, а область памяти в управляемой куче.

[7] http://www.memorymanagement.org/glossary/s.html#term-simple segregated-storage

этот блок делится на подблоки равного размера. С каждым блоком также связывается дескриптор, хранящий два битовых массива: массив битов маркировки (mark_bits) и массив битов закрепления (pin_bits) — а также другую метаинформацию. Блоки индексируются с помощью двух-трёх уровневого сильноветвящегося дерева. Дерево индексации позволяет по указателю на управляемый объект за константное время определить связанный с ним дескриптор и соответствующие биты маркировки и закрепления, а также по произвольному указателю определить, является ли объект, на который он указывает, управляемым. Большие объекты выравниваются по размеру страницы и выделяются напрямую с помощью системного вызова mmap. С ними также связывается дескриптор, который хранит размер объекта а также бит маркировки и бит закрепления.

Остановка мира На стадии сжатия и освобождения кучи все потоки приложения приостанавливаются. Однако поток может быть приостановлен не в любой момент, а только когда соблюдаются определенные инварианты, необходимые для корректной работы сборщика. К примеру, предположим, что сборщик мусора решит приостановить поток мутатора, в то время как он выделяет новый блок памяти из кучи. Поток может быть приостановлен в середине этой операции, и куча окажется в неконсистентном состоянии.

Безопасной точкой называется такое состояние мутатора, в котором он может быть приостановлен сборщиком. Если сборщик мусора имеет поддержку со стороны компилятора, как правило, именно компилятор расставляет безопасные точки в пользовательском коде. То есть компилятор вставляет последовательность инструкций, которая проверяет, была ли запрошена сборка мусора, и останавливает поток, если необходимо. Как уже упоминалось, наша библиотека не имеет поддержки компилятора. Поэтому, безопасные точки в предыдущей версии были вручную расставлены в некоторых

примитивах библиотеки, например, в функции gc_new. Такое решение имеет существенный недостаток: если какой-либо поток редко использует примитивы библиотеки, он не сможет быть остановлен сборщиком. Сборка мусора не может быть начата, пока все потоки мутатора не будут приостановлены. Таким образом, пользователь библиотеки мог столкнуться с ситуацией, при которой сборка мусора никогда бы не была вызвана. Для преодоления этой проблемы в данной работе был предложен новый алгоритм остановки мира, описанный в разделе 4.1.

Стоит также отметить, что для остановки мира, как в старой, так и в новой реализации, необходимо поддерживать список активных потоков. Для поддержки этого списка была реализована обёртка над библиотекой std::thread. Пользователь библиотеки должен использовать шаблонную функцию create_managed_thread для создания нового управляемого потока. Эта функцию принимает функтор и набор аргументов, создаёт новый поток, регистрирует его в списке активных потоков, вызывает функтор на переданных аргументах и затем удаляет поток из списка. Функция create_managed_thread возвращает экземпляр класса std::thread. Заметим, что пользователь не обязан регистрировать все потоки приложения, достаточно лишь тех, что будут обращаться к управляемой куче. Таким образом, в программе одновременно могут существовать неуправляемые и управляемые потоки, причём в фазе остановки мира приостановлены будут только последние.

Преобразование управляемого указателя в сырой Для хранения управляемых указателей пользователь библиотеки должен использовать объекты класса gc_ptr. Однако иногда может понадобиться получить сырой[8] указатель на управляемый объект (например, для передачи

[8] *Сырым* указателем называется обычный указатель C/C++ вида T*, где T — тип объекта.

такого указателя в функцию сторонней библиотеки, не знающей о существовании сборки мусора). Отметим также, что некоторые указатели в программе не могут быть умными [4], например, указатель this. Проиллюстрируем это нижеследующим примером.

Listing 2.3: Неявное разыменование gc_ptr

```
struct A {
    int x;
    int y;
    void f() {
        x = 0;
        y = 42;
    }
}

gc_ptr<A> a = gc_new<A>();
a->f();
```

В данном примере при вызове метода f() сырой указатель на объект a (this) будет неявно сохранен на стеке. Если между строками 5 и 6 работа приложения будет остановлена сборщиком мусора, а объект a будет перемещен, то после возобновления исполнения на стеке окажется некорректный указатель, после чего поведения программы становится неопределенным (*undefined behaviour*).

Для предотвращения этой ситуации в предыдущей версии библиотеки разыменованные управляемые указатели заносились в хэш-таблицу разыменованных указателей. Перед запуском сборки мусора просматривались стеки потоков. Указатели на стеке, содержащиеся в хеш-таблице, считались корнями. Также у объектов, на которые они указывали, выставлялся бит закрепления. Наличие этого бита говорило сборщику, что указанный объект не может быть перемещён. Момент, когда закрепление может быть снято, не отслеживался точно. Все указатели из хэш-таблицы,

которые не были найдены на стеке, удалялись из неё перед возобновлением работы мутатора.

Так как сборщик мусора использовал консервативный обход стека для идентификации разыменованных указателей, строго говоря, он не являлся точным. Была возможна ситуация, при которой не оставалось указателей на объект, и тем не менее занимаемая им память не освобождалась. Поэтому предыдущая версия сборщика мусора являлась *почти-точной* (*mostly-precise*). В данной работе предложен новый механизм преобразования управляемых указателей в сырые указатели, который возвращает сборщику свойство точности и позволяет точно отслеживать момент, когда закрепление может быть снято.

4 Реализация

В данном разделе приводится описание выполненных модификаций библиотеки и деталей их реализации.

4.1 Остановка мира

В разделе 3 были перечислены недостатки использовавшегося алгоритма остановки мутатора. Для их преодоления был предложен новый подход. Вместо расстановки безопасных точек в примитивах библиотеки предлагается помечать небезопасные участки кода, а другие участки считать безопасными. Преимущество данного подхода в том, что теперь мутатор может быть уведомлён об инициации сборки мусора в любой момент своей работы. Если в момент инициации сборки поток находится в небезопасном участке, то его приостановка откладывается до момента, когда он покинет этот участок. Заметим, что небезопасными являются только участки кода, выполняющие некоторые операции с глобальными структурами данных (корневым

множеством, кучей, и т.д.) и расположенные в примитивах самой библиотеки.

Для реализации этого подхода необходимо иметь возможность уведомить поток об инициации сборки мусора в любой момент времени. После получения такого уведомления поток должен приостановить свою работу до момента, когда сборка будет окончена. Подобный функционал может быть реализован при помощи механизма сигналов операционной системы Linux и примитивов синхронизации, являющихся асинхронно-сигналобезопасными. Далее в данной главе будут рассмотрены сигналы и реализация механизма их временной блокировки, учитывающая специфику их использования в нашей библиотеке. Также будет описана реализация асинхронно-сигналобезопасных примитивов синхронизации. В последнем подразделе описан новый алгоритм "остановки мира".

Сигналы ОС Linux Сигналы в Linux — один из механизмов межпроцессорного взаимодействия [8]. Linux поддерживает несколько типов сигналов, каждый из которых имеет уникальный идентификатор и мнемоническое имя. Сигнал может быть отправлен процессу либо ядром операционной системы, либо другим процессом с помощью системного вызова kill. При получении сигнала вызывается соответствующий обработчик сигнала. Для некоторых типов сигналов разрешается переопределять обработчик сигнала (при помощи системного вызова sigaction). Также имеется возможность временно заблокировать доставку сигналов определенного типа (системный вызов sigprocmask).

Библиотека pthreads позволяет отправлять сигналы конкретному потоку процесса с помощью системного вызова pthread_kill, тем самым позволяя использовать сигналы для межпоточного взаимодействия.

Блокирование сигналов В библиотеке pthreads имеется функция pthread_sigmask аналогичная sigprocmask, но блокирующая доставку сигналов только данному потоку. Однако при каждом вызове pthread_sigmask происходит переход в *режим ядра* (*kernel mode*), что вызывет длительные задержки. Временное блокирование доставки сигнала об инициации сборки мусора очень частая операция. Например, она необходима при каждой модификации корневого множества. Из-за её низкой скорости выполнения может пострадать производительность всего приложения, использующего сборку мусора.

Для того, чтобы ускорить блокирование доставки сигнала инициации сборки мусора, был реализован механизм, учитывающий специфику задачи. Идея заключается в том, чтобы отследить попытку доставки сигнала потоку, выполнившему его блокирование, и отложить выполнение обработчика до момента разблокировки. В каждом потоке объявляются две локальные для потока (thread_local) переменные depth и pending_flag. depth отслеживает вложенность блокировок, pending_flag инициализируется значением false. В обработчике сигнала сначала проверяется значение depth. Если оно не равно нулю, значит в текущий момент доставка сигнала заблокирована, выставляется pending_flag, и происходит выход из обработчика. Иначе происходит нормальное исполнение обработчика сигнала. Для того чтобы заблокировать доставку сигнала поток просто инкрементирует переменную depth. При разблокировании доставки сигнала значение depth декрементируется, и если оно становится равным нулю, и при этом установлен флаг pending_flag, вызывается код обработчика сигнала, но уже без дополнительных проверок. Код обработчика сигнала, блокирования и разблокирования доставки сигналов на C++ может быть найден в листинге 2.4.

В таблице 1 приводится сравнение времени работы механизма блокирования сигналов с помощью функции

pthread_sigmask и механизма, предложенного в нашей работе. Измерялось суммарное время работы пары операций блокирования и разблокирования сигнала. В таблице приведено среднее время работы операций $\bar{x}$ и стандартное отклонение s. Заметим, что ускорения удалось достичь за счёт того, что предложенный подход к блокированию сигналов является менее общим и учитывает специфику нашей задачи.

Таблица 1: Сравнение времени работы двух методов блокирования сигналов (среднее время $\bar{x}$ и стандартное отклонение s указано в наносекундах)

	$\bar{x}$	s
pthread_sigmask	143.661	6.741
siglock/sigunlock	3.965	0.033

Асинхронно-сигналобезопасные примитивы синхронизации Стандарт POSIX[9] накладывает жёсткие ограничения на код, работающий внутри обработчика сигнала. Обработчик сигнала должен быть асинхронно-сигналобезопасной функцией [10] (*async-signal safe function*).

После получения сигнала об инициации сборки мусора поток должен приостановить свою работу до тех пор,

[9] Стандарт POSIX описывает интерфейс для доступа к функциям операционной системы. Обеспечивает переносимость прикладных программ между ОС, поддерживающими POSIX. http://pubs.opengroup.org/onlinepubs/9699919799/

[10] Определение асинхронно-сигналобезопасной функции может быть найдено по ссылке https://www.securecoding.cert.org/confluence/display/c/BB.+Definitions#BB.Definitions-asynchronous-safefunction

пока он не получит уведомление об окончании сбоки. Код, выполняющий эту задачу, должен работать в контексте обработчика сигнала, так как иного способа уведомить поток о некотором событии в произвольный момент времени, кроме использования сигналов, нет. Так как никакие из примитивов синхронизации библиотеки pthreads не являются асинхронно-сигналобезопасными[11], необходимо использовать собственную реализацию примитивов синхронизации.

В нашей библиотеке было реализована два примитива: signal_safe_barrier и signal_safe_event.

- signal_safe_barrier имеет два метода:
  1. notify() для уведомления о том, что поток достиг некоторой точки в программе;
  2. wait(size_t n) для приостановки текущего потока до момента, когда n других потоков достигнут барьера.
- signal_safe_event также определяет два метода:
  1. notify(size_t n) для уведомления n потоков о наступлении события;
  2. wait() для ожидания события.

Оба примитива реализованы с помощью одной и той же техники. Системные вызовы read и write, предназначенные для чтения и записи данных из потока ввода-ввывода, являются асинхронно-сигналобезопасными. Кроме того, read является блокирующим, то есть поток, вызвавший read, будет приостановлен, пока из потока не будет прочитано указанное количество байт. На основе этих двух системных вызовов, а также системного вызова pipe, открывающего безымянный двунаправленный поток ввода-ввывода, и построены наши примитивы.

[11] Список async-signal safe функций ОС Linux может быть найден по ссылке
http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_04_03

Оба примитива реализованы как классы. В конструкторе они открывают безымянный поток ввода-вывода. Метод notify() класса signal_safe_barrier записывает один байт в этот поток. Метод wait(size_t n) читает из потока n байт. Аналогично для signal_safe_event, notify(size_t n) записывает n байт в поток, а wait() читает из потока один байт.

Алгоритм "остановки мира" Опишем теперь сам алгоритм "остановки мира". Рассмотрим отдельно часть, отвечающую за приостановку потока, выполняющуюся в обработчике сигнала, и часть, инициирующую сборку. В ходе выполнения обе части синхронизируют свою работу с помощью одного объекта класса signal_safe_barrier и одного объекта signal_safe_event. В обработчике сигнала поток сначала вызывает метод notify() барьера, чтобы уведомить поток, инициировавший "остановку мира", что он достиг обработчика. Затем вызывается метод wait() события. Когда сборка мусора окончится, поток, выполнявший сборку, возобновит работу остальных потоков, вызвав метод notify(size_t) этого события. Затем в обработчике события будет вызван notify() барьера ещё раз, чтобы уведомить поток инициатор, что обработка сигнала завершена, и поток возвращается к нормальному исполнению. Для того, чтобы два потока одновременно не инициировали "остановку мира", код инициирования защищён мьютексом. Псевдокод инициирования "остановки мира" приведён в листинге 2.5.

4.2 Закрепление объектов

Механизм закрепления объектов, использовавшийся в предыдущей версии библиотеки (раздел 3), включал процедуру консервативного обхода стека потоков. Поэтому в некоторых ситуациях сборщик мог ошибочно не идентифицировать мусор. В новой версии реализован механизм закрепления объектов, лишённый этого недостатка,

Listing 2.4: Блокирование доставки сигналов

```
thread_local volatile sig_atomic_t depth = 0;
thread_local volatile sig_atomic_t pending_flag = false;

void check_gc_siglock(int signum) {
    if (depth > 0) {
        pending_flag = true;
        return;
    }
    gc_signal_handler();
}

void siglock() {
    if (depth == 0) {
        std::atomic_signal_fence();
        depth = 1;
        std::atomic_signal_fence();
    } else {
        depth++;
    }
}

void sigunlock() {
    if (depth == 1) {
        std::atomic_signal_fence();
        depth = 0;
        std::atomic_signal_fence();
        bool pending = pending_flag;
        pending_flag = false;
        if (pending) {
            gc_signal_handler();
        }
    } else {
        depth--;
    }
}
```

Listing 2.5: Остановка мира

```
lock(gc_mutex)
for (thread in threads)
    send(gc_signal, thread)
signal_safe_barrier.wait(threads.count - 1)
gc()
signal_safe_event.notify(threads.count - 1)
signal_safe_barrier.wait(threads.count - 1)
unlock(gc_mutex)
```

и кроме того, точно отслеживающий время, когда закрепление может быть снято.

Используется идиома RAII, упомянутая в разделе 2.3. Вводится новый примитив — шаблонный класс gc_pin<T>. Конструктор класса gc_pin<T> принимает в качестве аргумента ссылку на объект типа gc_ptr<T> и выполняет закрепление объекта, на который указывает этот gc_ptr. В деструкторе gc_pin<T> закрепление объекта снимается. Указатели на все закрепленные объекты хранятся в структуре данных, аналогичной той, что используется для хранения корневого множества (раздел 3). То есть используется список указателей, причём новые указатели вставляются в голову и при удалении список просматривается с головы. Также как и в случае корневого множества, каждый поток имеет собственный экземпляр списка закрепленных объектов. Таким образом, в конструкторе gc_pin<T> указатель на объект вносится в список закрепленных объектов, а в деструкторе удаляется из него.

На этапе сжатия/освобождения кучи, после “остановки мира”, сборщик просматривает списки закрепленных объектов всех потоков. У объектов из этих списков выставляется бит закрепления (наличие этого бита говорит сборщику о запрете перемещать данный объект), также они сканируются

сборщиком (для того, чтобы пометить все объекты, достижимые из закрепленного, и предотвратить их удаление).

Как уже упоминалось, закрепление объектов необходимо в двух случаях:

1. Для передачи сырого указателя на управляемый объект в функцию, принимающую сырой указатель как аргумент.
2. Для закрепления указателя this при вызове оператора доступа к члену класса (operator->()).

В первом случае пользователь может самостоятельно создать объект типа gc_pin и получить сырой указатель с помощью метода get() этого объекта. Стоит однако отметить, что время жизни полученного таким образом сырого указателя не должно превышать время жизни соответствующего объекта gc_pin. Во втором случае объект gc_pin создаётся неявно самой библиотекой. Используется следующее соглашения языка C++ [12] : если оператор доступа к члену класса (operator->()) возвращает объект типа, отличного от T*, вызывается оператор доступа к члену класса этого объекта. То есть, перегруженный оператор доступа к члену класса gc_ptr создаёт на стеке объект типа gc_pin и возвращает его, затем, согласно соглашению языка C++, вызывается оператор доступа к члену класса этого временного объекта, который уже возвращает сырой указатель. После этого вызывается деструктор временного объекта gc_pin.

4.3 Инкрементальная параллельная маркировка

Рассмотрим реализацию алгоритма инкрементальной параллельной маркировки, разработанную в рамках данной работы и внедренную в библиотеку точной сборки мусора для языка C++.

[12] Working Draft, Standard for Programming Language C++, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf

В нашей работе в фазе маркировки работа мутатора не приостанавливается, сборщик мусора работает параллельно с потоками мутатора. “Остановка мира” происходит два раза за цикл сборки мусора: перед запуском этапа маркировки для сканирования корневого множества и для освобождения памяти после окончания маркировки. Выполнять маркировку параллельно могут сразу несколько потоков. Фаза маркировки начинается, когда размер кучи достигает некоторого порогового значения.

Для распределения работы между потоками-маркировщиками используется механизм *пакетов работы* (*work packet*), подобный описанному в статье [11]. Пакет представляет собой массив фиксированного размера, который содержит указатели на объекты, ожидающие сканирования. Сборщик мусора поддерживает глобальный пул пакетов. Каждый поток–маркировщик использует два пакета — входной (input) и выходной (output). Маркировщик по очереди вынимает указатели из входного пакета и сканирует объекты, на которые они указывают. Если обнаружится, что объект содержит указатели на другие объекты, они заносятся в выходной пакет. Когда входной пакет оказывается пуст или выходной пакет заполнен, маркировщик возвращает пакет в глобальный пул и получает новый пакет.

Глобальный пул пакетов поддерживает три списка: список пустых пакетов, список частично (менее чем на 50%) заполненных пакетов и список почти полностью (более чем на 50%) заполненных пакетов. Каждый список также содержит мьютекс, для того, чтобы синхронизировать операции добавления и удаления элементов при обращении нескольких потоков. Суммарное количество пакетов фиксированно, память для пакетов выделяется один раз при инициализации сборщика мусора. Когда поток–маркировщик запрашивает входной пакет у пула, выбирается пакет из почти полностью заполненных, а если таковых нет — из частично заполненных.

При запросе выходного пакета возвращается пустой пакет, если же список пустых пакетов пуст, то пакет из частично заполненных. Изначально все пакеты находятся в списке пустых пакетов. При сканированни корневого множества сборщик запрашивает у пула необходимое количество выходных пакетов, копирует в них указатели на корневые объекты и возвращает пакеты в пул. После этого потоки-маркировщики начинают запрашивать входные пакеты и сканировать их.

Использование механизма пакетов позволяет достаточно просто определить момент окончания фазы маркировки — когда количество пустых пакетов становится равным общему количеству пакетов, фаза маркировки считается законченной.

Рассмотрим также реализацию барьера записи. В нашей библиотеке барьер записи реализован программно и вызывается из конструктора копирования и оператора присваивания `gc_ptr`. Барьер записи перехватывает операции записи только если сборщик находится в фазе маркировки. Мы используем барьер записи Дейкстры, его псевдокод приведён в листинге 2.1. Напомним, что барьер записи Дейкстры перекрашивает объект, ссылка на который записывается, в серый цвет, если раньше его цвет был белым. Для этого каждый поток мутатора запрашивает у глобального пула пакетов один выходной пакет и помещает в него ссылки на “перекрашенные” в серый объекты. Когда пакет заполняется, поток возвращает его в пул и запрашивает новый. Заметим, что после окончания фазы маркировки у потоков мутатора могут остаться частично заполненные пакеты серых объектов. Данные пакеты сканируются уже после “остановки мира”. Ожидается, что количество объектов в этих частично заполненных пакетах будет небольшим.

Также стоит отметить, что в фазе маркировки новые объекты создаются чёрными, то есть гарантированно переживают текущий цикл сборки мусора.

Операции присваивания указателей и создания новых объектов очень часты, поэтому необходимо, чтобы проверка текущей фазы была достаточно быстрой. В нашей реализации для этого используется переменная phase типа перечисления (допустимые значения: IDLE, MARKING, COMPACTING). Изменения значения переменной phase происходят только во время остановок мира, а операции присваивания указателей и создания нового объекта помечены как небезопасные, то есть во время их выполнения не может произойти "остановки мира" и, следовательно, не может измениться фаза сборки мусора.

4.4 Параллельное сжатие

В процедуру сжатия кучи также были внесены изменения с целью её распараллеливания. Как уже упоминалось в разделе 3, куча для объектов маленьких размеров представляет собой набор пулов, каждый из которых обслуживает объекты определенного размера. Ясно, что каждый пул можно сжимать независимо от остальных в отдельном потоке. В нашей реализации множество пулов разбивается на n равных частей, где n — количество доступных на данной системе ядер процессора, полученное с помощью функции из стандартной библиотеки std::thread::hardware_concurrency. Затем создаётся n потоков, каждый из которых поочередно сжимает пулы из соответствующей части с помощью двухпальцевого алгоритма 2.1. Преимущество подобного статического разбиения работ в том, что потокам нет необходимости синхронизировать свою работу. С другой стороны, если окажется, что объекты распределены по пулам неравномерно, то сборщик не сможет полностью задействовать аппаратный параллелизм, так как некторые потоки закончат свою работу намного раньше остальных.

Объекты большого размера выделяются с помощью системного вызова mmap и сохраняются в список больших

объектов. Для таких объектов сжатие не выполняется. Память из-под объектов, объявленных сборщиком недоступными, возвращается операционной системе с помощью системного вызова munmap.

5 Апробация

Функциональность сборщика мусора была протестирована на модульных тестах, написанных вручную с помощью фреймворка **googletest**[13]. Также было произведено тестирование производительности реализованного алгоритма инкрементальной параллельной маркировки. Для этой цели использовался известный тест Бёма, активно работающий с динамической памятью. В этом тесте строятся двоичные деревья различной глубины с различным количеством узлов. Деревья строятся двумя способами: от листов к корню и наоборот, от корня к листьям. Было выполнено сравнение библиотеки с аналогами, в частности, с ручным управлением памятью с помощью new/delete, с классом умного указателя std::shared_ptr, реализующим подсчёт ссылок, а также с консервативным сборщиком мусора Бёма-Демерса-Вайзера с включенной и выключенной опцией инкрементальной маркировки. Наша библиотека тестировалась с четырьмя стратегиями: включенной/выключенной инкрементальной маркировкой и включенным/выключенным сжатием кучи.

Измерялось суммарное время работы теста (рис. 2), количество сборок мусора (на графике показано как целое число над соответствующим столбцом) и среднее время “остановки мира” (рис. 3). Все измерения были взяты как среднее и стандартное отклонение по 20 запускам теста. Хотя суммарное время работы сборщика с инкрементальной параллельной маркировкой и увеличилось, среднее время

[13] Google Test, Google’s C++ test framework, https://github.com/google/googletest

паузы существенно уменьшились. В частности, среднее время паузы при отключенном сжатии при инкрементальной маркировке примерно в шесть раз меньше, чем без неё. При одновременном включении и сжатия, и инкрементальной маркировки время работы теста существенно увеличивается. Это объясняется тем, что при инкрементальной маркировки доля объектов, переживающих текущий цикл сборки, больше, а время работы текущей реализации процедуры сжатия существенно увеличивается при увелечении размера кучи. Стоит отметить, что наша библиотека работает примерно в 4 раза медленее чем другие решения, участвовавшие в эксперименте, что свидетельствует о необходимости дальнейшей оптимизации.

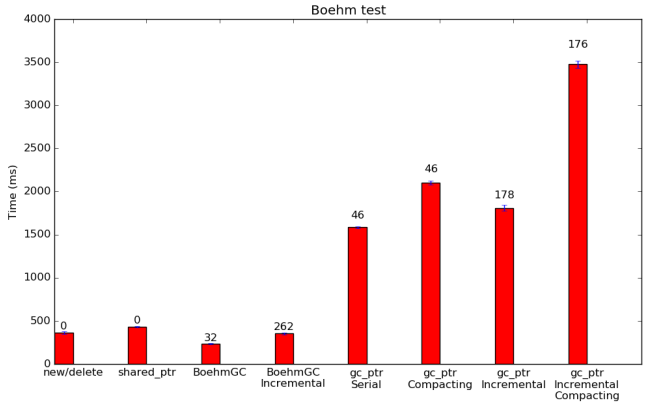


Рис. 2: Тест Бёма. Суммарное время работы.

Также было протестирована работа сборщика с многопоточным приложением. Для этой цели был написан тест, использующий алгоритм сортировки слиянием. В этом

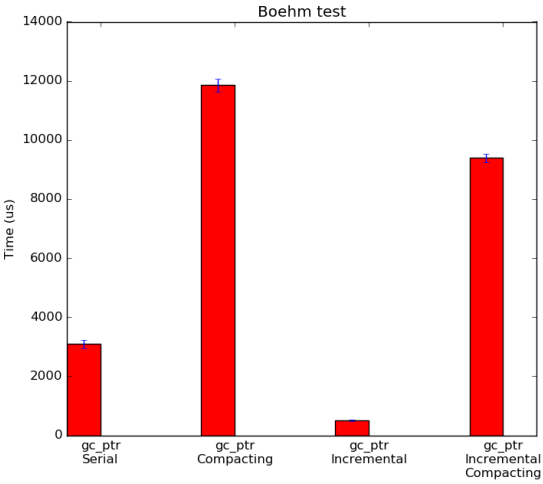


Рис. 3: Тест Бёма. Среднее время паузы.

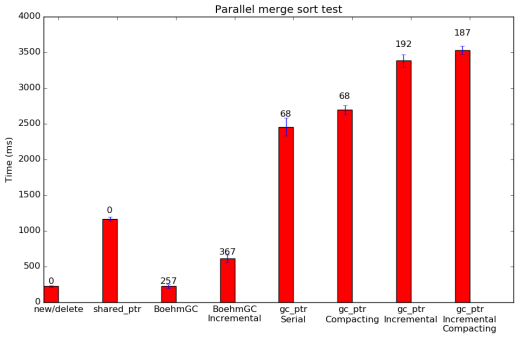


Рис. 4: Многопоточная сортировка слиянием. Суммарное время работы.

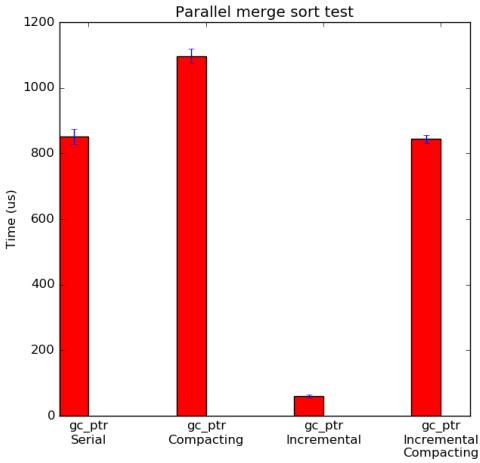


Рис. 5: Многопоточная сортировка слиянием. Среднее время паузы.

тесте генерируются односвязные списки. Каждый узел списка содержит одно случайное значение типа int. Список разделяется на несколько частей (по числу доступных ядер процессора), каждая часть сортируется в отдельном потоке сортировкой слиянием, затем части сливаются в единый список. Результаты замеров работы библиотеки, а также аналогов, на этом тесте представлены на рис. 4 и рис. 5. Видно, что результаты данного теста в целом повторяют результаты теста Бёма.

Результаты

В ходе работы были достигнуты следующие результаты:

– проанализированы ограничения и недостатки предыдущей версии библиотеки;

– реализован новый алгоритм остановки мира;
– реализован новый алгоритм закрепления объектов;
– реализована модификация алгоритма инкрементальной параллельной маркировки;
– выполнена апробация: протестирована функциональность библиотеки, измерена её производительность, произведено сравнение с аналогами.

Список литературы

1. D. Berezun, D. Boulytchev. Precise Garbage Collection for C++ with a Non-cooperative Compiler. Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia, 2014.
2. Д. Березун. Реализация основных примитивов библиотеки неконсервативной сборки мусора для C++. Труды лаборатории языковых инструментов JetBrains, выпуск 2, 2014.
3. А.Самофалов. Библиотека почти точной копирующей сборки мусора для C++. Труды лаборатории языковых инструментов JetBrains, выпуск 3, 2015
4. Jones, Richard and Lins, Rafael D. Garbage collection: algorithms for automatic dynamic memory management. Wiley, 1996.
5. Jones, Richard and Hosking, Antony and Moss, Eliot. The garbage collection handbook: the art of automatic memory management. Chapman & Hall/CRC, 2011.
6. Dijkstra, Edsger W and Lamport, Leslie and Martin, Alain J and Scholten, Carel S and Steffens, Elisabeth FM. On-the-fly garbage collection: An exercise in cooperation. Communications of the ACM, vol. 21, num. 11, pp. 996—975, 1978.
7. Boehm, Hans-J and Spertus, Michael. Transparent Programmer-Directed Garbage Collection for C+. URL: http://www.open-std.org/jtc1/sc22/wg21/docs/papers, num. 2310, 2007.
8. А.М. Робачевский. Операционная система UNIX, 2 изд., БХВ-Петербург, 2010.
9. Boehm, Hans-Juergen. Space efficient conservative garbage collection. ACM SIGPLAN Notices, vol. 28, num. 6, pp. 197–206, 1993.

10. Э. Уильямс. Параллельное программирование на C++ в действии. Практика разработки многопоточных программ. Litres, 2014.
11. Barabash, Katherine and Ben-Yitzhak, Ori and Goft, Irit and Kolodner, Elliot K and Leikehman, Victor and Ossia, Yoav and Owshanko, Avi and Petrank, Erez. A parallel, incremental, mostly concurrent garbage collector for servers, ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 27, num. 6, pp. 1097–1146, 2005.

Генерация декларативных принтеров по грамматике в форме Бэкуса-Наура

Озерных Игорь Станиславович

Санкт-Петербургский государственный университет
igor.ozernykh@gmail.com

Аннотация Средства автоматического форматирования программ являются неотъемлемой частью сред разработки. Одним из способов их реализации является метод синтаксических шаблонов, который требует подробного описания конструкций целевого языка. Ранее такие описания составлялись вручную в формате XML. В данной работе представлен метод, позволяющий получать описания по БНФ-грамматике целевого языка. Метод был реализован для системы генерации синтаксических анализаторов GrammarKit и апробирован на целевых языках While и Erlang.

Введение

Интегрированные среды разработки (Integrated Development Environment, IDE) являются неотъемлемой частью современного программирования и разработки программных продуктов. Появившись в 1970-х годах, они прошли длительный путь развития и совершенствования. В наши дни типичная среда разработки включает в себя текстовый редактор, компилятор и/или интерпретатор, средства автоматической сборки, отладчик и другие инструменты. Популярными IDE для языка Java являются Eclipse[1] и IntelliJ IDEA[2]. Они также включают средства

[1] https://eclipse.org
[2] http://jetbrains.com/idea

для работы с системами контроля версий, рефакторинга, форматирования кода. Кроме того, существует возможность расширения функциональности этих IDE путем создания плагинов — программных компонентов, добавляющих новые возможности. В частности, плагины могут добавлять поддержку новых языков в IDE. Для разработки подобного плагина необходимо в большинстве случаев реализовать синтаксический анализатор целевого языка, который в случае IntelliJ IDEA можно получить с помощью плагина Grammar-Kit по грамматике этого языка в форме Бэкуса-Наура. Также, при реализации поддержки нового целевого языка часто возникает потребность в реализации форматера для программ на этом языке.

Задача автоматического форматирования текстов программ является классической в области разработки языковых процессоров. Она подразумевает построение текста программы по ее синтаксическому дереву или дереву разбора. Для каждого дерева существует множество различных текстовых представлений. Так, на рис. 1 приведен пример синтаксического дерева оператора ветвления языка C, а на рис. 2 различные текстовые представления этого дерева.

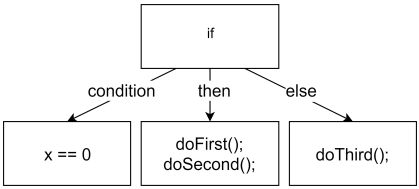


Рис. 1: Оператор ветвления в виде дерева разбора

Программный компонент, занимающийся форматированием, мы будем называть *принтером*. Как правило, результат работы принтера должен соответствовать некоторому

```
if (x == 0) {
    doFirst();
    doSecond();
} else {
    doThird();
}
```

а)

```
if (x==0)
  {
    doFirst();
    doSecond();
  }
else
  {
    doThird();
  }
```

б)

Рис. 2: Текстовые представления синтаксического дерева с рис. 1

стандарту кодирования (СК, coding convension) — набору правил и соглашений, используемых при написании кода на соответствующем языке программирования. Так, стандарт кодирования для языка Java задает расположение фигурных скобок и операторов внутри тела функции, размер и формат отступов и др.

Одним из способов форматирования программных текстов является метод, основанный на *синтаксических шаблонах* [1]. Под *шаблоном* понимаются данные, сопоставление которых с элементом синтаксического дерева дает текстовое представление этого элемента (и его потомков). На рис. 3, *а* представлен шаблон форматирования, который может быть применен к дереву разбора с рис. 1. На рис. 3, *б* представлен результат применения шаблона к этому дереву.

Такой подход был применен в работе [1] при разработке принтер-плагина для среды разработки IntelliJ IDEA. На рис. 4 приведен процесс работы принтер-плагина. На вход принтеру подаются соответствующий требуемому форматированию код (эталонный репозиторий), из которого

```
if (@condition) {
    @expr
} else {
    @expr
}
```

а) Шаблон для оператора ветвления

```
if (x == 0) {
    doFirst();
    doSecond();
} else {
    doThird();
}
```

б) Текст, полученный при применении шаблона к дереву разбора

Рис. 3: Оператор ветвления и шаблон для него

извлекаются шаблоны форматирования, и код, который необходимо отформатировать (целевой код). Полученные шаблоны применяются к целевому коду, и результатом работы принтера становится отформатированный целевой код.

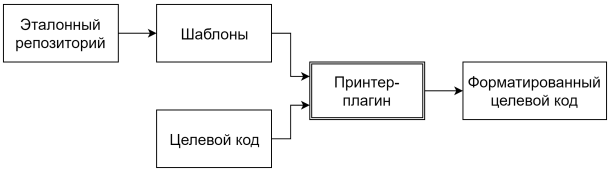


Рис. 4: Принцип работы принтер-плагина

Список поддерживаемых принтер-плагином языков можно расширять. Для этого необходимо создать языкозависимую прослойку между ядром платформы и представлением синтаксического дерева, полученного в результате работы анализатора. Метод, описанный в [1], позволяет генерировать такую прослойку по языкозависимому описанию *компонент*

```
<component name="While" psiComponentClass="PsiWhileStmt">
 <subtree name="condition" psiGetMethod="Bexpr"
      hasSeveralElements="false" isRequired="true" />
 <subtree name="body" psiGetMethod="StmtList"
      hasSeveralElements="false" isRequired="true" />
</component>
```

Рис. 5: Описание компоненты принтера, форматирующей циклы с предусловием

принтера — классов, описывающих, как форматировать соответствующие элементы синтаксического дерева. Описание каждой компоненты принтера (component) является XML-файлом. Каждый такой файл содержит имя компоненты (name), класс внутреннего представления в IDE (psiComponentClass), а так же список поддеревьев (subtree) и их свойств (name, psiGetMethod, hasSeveralElements, isRequired). На рис. 5 приведено XML-описание компоненты принтера, соответствующей циклу с предусловием. Современные языки программирования содержат большое число структур, поэтому необходимо большое число подобных XML-описаний для задания принтера языка. Процесс создания такого описания трудоемок и требует глубоких знаний о системе [1].

Грамматика языка, которая используется в Grammar-Kit и по которой генерируется код синтаксического анализатора, содержит большую часть необходимой информации для создания этой языкозависимой прослойки. Поэтому принтер для целевого языка можно получать по грамматике этого языка.

1 Постановка задачи

Целью данной работы является автоматизация процесса задания декларативных принтеров путем их генерации по грамматике языка в форме Бэкуса-Наура. Данная грамматика также используется для получения синтаксического анализатора программ на целевом языке с помощью плагина Grammar-Kit.

Поставлены следующие задачи:

- разработка методики генерации декларативных принтеров по грамматике в форме Бэкуса-Наура;
- интеграция метода в проект Grammar-Kit путем реализации генератора принтеров;
- реализация интеграции полученных принтеров с принтер-плагином для IDE IntelliJ IDEA;
- апробация метода на основе грамматики языков While [6] и Erlang.

While — учебный язык, содержащий самые основные конструкции. На его примере производилась апробация метода генерации принтеров по языкозависимому описанию компонент [1]. Erlang[3] — промышленный язык программирования, для которого уже существуют грамматика в нужной форме и синтаксический анализатор, полученный по грамматике с помощью плагина Grammar-Kit.

2 Обзор

В данном обзоре рассматриваются некоторые подходы к заданию принтеров и форматеров, а также плагин Grammar-Kit для IntelliJ IDEA, позволяющий по БНФ-грамматике задавать синтаксический анализатор целевого языка. Используемые плагином грамматики рассматриваются на примере грамматики языка While [6].

[3] https://www.erlang.org

2.1 Подходы к заданию принтеров и форматеров

Рассмотрим некоторые подходы к заданию принтеров и форматеров.

Задание форматеров по описанию синтаксиса. Существуют различные подходы к заданию принтеров для целевого языка. Один из них описан в [8]. Авторы предлагают язык описания синтаксиса, по которому можно получить и синтаксический анализатор, и принтер для языка. Описание синтаксиса состоит из правил, которые схожи с правилами формальных грамматик, но которые также задают и стиль форматирования для данной структуры языка. Приведенное ниже правило вывода описывает основные арифметические выражения:

```
templates
  Exp.Num = <<INT>>
  Exp.Plus = <<Exp> + <Exp>>
  Exp.Times = <<Exp> * <Exp>>
```

Первое преобразование определяет шаблон для выражений, состоящих из чисел. Остальные преобразования задают шаблоны для операций сложения и умножения. Каждое из них состоит из двух меток <Exp> для подстановки выражений, арифметического знака, а также двух пробелов вокруг знака. Само правило явно задает способ, которым будут отформатированы арифметические выражения полученного языка. Вместо пробелов можно также указать табуляции и/или переносы строк. Недостатком данного подхода является то, что правила форматирования задаются заранее, и, чтобы их изменить, необходимо менять описание синтаксиса языка. Кроме того, каждое такое правило задает единственный вариант форматирования данной структуры.

Наиболее близкий к нашей работе метод описан в [3]. Принтер языка может быть получен по ASF+SDF–описанию языка [5], что представляет собой контекстно-свободную

грамматику. При этом правила форматирования не указываются. Недостатком данного подхода является то, что при генерации принтера задается базовый стиль форматирования, и, чтобы его изменить, необходимо редактировать сгенерированный код, тогда как подход с использованием синтаксических шаблонов позволяет пользователю декларативным образом настраивать принтер.

Форматеры, встроенные в IDE. Также широко распространены форматеры, встроенные в IDE. Они используют множество настроек для задания стиля форматирования (рис. 6). Среди них: тип и размер отступов, расположение фигурных скобок в C-подобных языках, политика переноса списочных выражений на новую строку и десятки других. Набор этих настроек выбирается разработчиком форматера на основе его представлений о возможных стилях кодирования для целевого языка. Такие форматеры обычно тесно интегрированы с IDE, имеют высокую скорость работы, и их выразительности, как правило, достаточно для задания необходимого стиля форматирования. Однако для поддержки нового языка необходимо определить нужный набор настроек и вручную реализовать форматер. В случае, если пользователю необходимо придерживаться стиля кодирования некоторой существующей кодовой базы, то ему нужно самостоятельно определить значения этих настроек. Данный недостаток отсутствует в работе [4], где авторы предлагают инструмент, позволяющий получить некоторые настройки форматера из существующего программного кода. Недостатком подхода является то, что число этих настроек невелико: система позволяет определять только стиль отступов, стили именования идентификаторов, необходимое количество комментариев. Другой подход [7] предлагает использование генетического алгоритма для поиска настроек форматера в исходном коде программ на языке C.

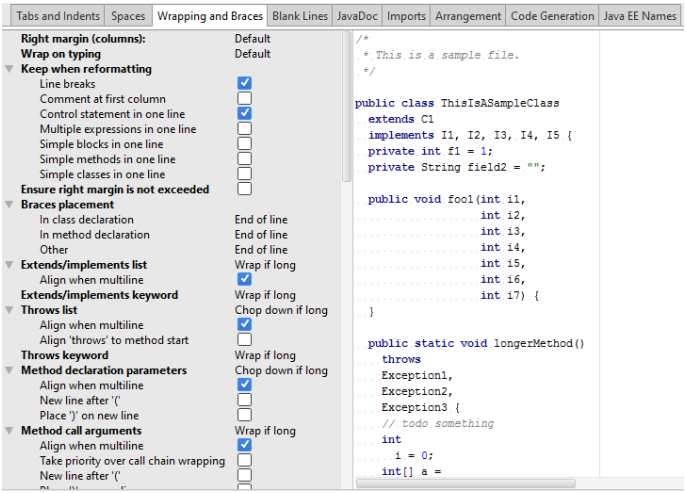


Рис. 6: Настройки форматера языка Java в IntelliJ IDEA

2.2 Grammar-Kit

Как уже было отмечено, для поддержки нового языка в платформе IntelliJ IDEA необходимо разработать синтаксический анализатор. Для его генерации можно использовать плагин Grammar-Kit. В качестве системы описания синтаксиса языка используется БНФ-грамматика. Результатом работы плагина является код синтаксического анализатора и иерархия классов внутреннего представления.

Рассмотрим грамматику языка While, для которого производилась апробация метода, описанного в [1]. While — язык программирования, содержащий следующие конструкции: чтение из стандартного потока (read) и запись в стандартный поток (write), оператор ветвления (if), цикл с предусловием (while); процедуры (proc), бинарные выражения (binary_expr), в том числе булевы (binary_bexpr) и т. д. На рис. 7 представлена часть грамматики языка While, задающая

```
1 whileFile ::= proc_list stmt_list
2 stmt_list ::= stmt*
3 proc_list ::= procedure*
4 stmt ::= skip|assign|if|while|write|read
5 skip ::= 'skip' ';'
6 write ::= 'write' '(' expr ')' ';'
7 read ::= 'read' '(' id ')' ';'
8 assign ::= id ':=' expr ';'
9 if ::= 'if' '(' bexpr ')' 'then' stmt_list ('else' stmt_list)? 'fi'
10 while ::= 'while' '(' bexpr ')' 'do' stmt_list 'od'
11 procedure ::= 'proc' id '(' param_list ')' stmt_list 'endp'
12 param_list ::= ref_expr? (',' ref_expr)*
13 ...
```

Рис. 7: Грамматики языка While. Операторы языка

множество операторов. Рассмотрим правило грамматики, задающее оператор ветвления. Правая часть правила состоит из терминалов 'if', 'then' '(' и др.; нетерминалов: bexpr, stmt_list, а также условного вхождения ('else' stmt_list)? (то есть конструкция может отсутствовать в программах на данном языке). Некоторые правила грамматики имеют модификаторы, которые используются для дополнительных указаний генератору синтаксического анализатора (рис. 8). Модификатор fake указывает системе, что соответствующий нетерминал не должен участвовать в синтаксическом разборе, однако соответствующие классы синтаксического дерева должны быть сгенерированы; private указывает на обратное — нетерминал участвует в анализе, но соответствующий класс не генерируется; left — используется для разбора левоассоциативных операций. Полный список модификаторов может быть найден на странице плагина GrammarKit[4].

[4] https://github.com/JetBrains/Grammar-Kit

```
1 ...
2 fake ar_op ::= plus_op|mul_op
3 fake binary_expr ::= expr ar_op expr
4
5 expr ::= factor plus_expr *
6 left plus_expr ::= plus_op factor
7 plus_op ::= '+'|'-'
8 private factor ::= primary mul_expr *
9 left mul_expr ::= mul_op primary
10 mul_op ::= '*'|'/'|'%'
11 private primary ::= literal_expr | ref_expr | paren_expr
12 paren_expr ::= '(' expr ')'
13 ref_expr ::= id
14 literal_expr ::= number
15
16 fake bl_op ::= or|and
17 fake binary_bexpr ::= bexpr bl_op bexpr
18 ...
```

Рис. 8: Грамматика языка While. Выражения с модификаторами

Модификатор private используется для нетерминалов, которые не имеют представления в синтаксическом дереве. Среди них те, которые используются для устранения левой рекурсии. Например, на рис. 9 представлено описание правил с рис. 8 (строки 5–14), но в более простой для восприятия леворекурсивной форме. Однако грамматика, с которой работает Grammar-Kit, не должна содержать леворекурсивных правил. Устраняя левую рекурсию, мы получим описание выражений на рис. 8 (строки 5–14). Кроме того, появляются новые правила, которые с точки зрения синтаксического анализа (а следовательно, и форматирования) являются избыточными. В данном случае такими являются factor и primary на рис. 8.

```
expr ::= plus_expr | mul_expr | paren_expr | ref_expr | literal_expr
plus_expr ::= expr plus_op expr
plus_op ::= '+'|'-'
mul_expr ::= expr mul_op expr
mul_op ::= '*'|'/'|'%'
paren_expr ::= '(' expr ')'
ref_expr ::= id
literal_expr ::= number
```

Рис. 9: Грамматики языка While. Выражения в естественной леворекурсивной форме

Каждый файл с грамматикой языка содержит в себе заголовок, в котором описывается различная дополнительная информация: используемые в сгенерированных файлах классы, префиксы и суффиксы сгенерированных классов внутреннего представления, Java-пакеты, множество терминальных символов грамматики (*tokens*) и др. (см рис. 10).

```
{ parserClass="com.intellij.whileLang.parser.WhileParser"
    psiClassPrefix="Psi"
    psiImplClassSuffix="Impl"
    psiPackage="com.intellij.whileLang.psi.impl"
    tokens=[...]
    ...
}
...
```

Рис. 10: Заголовок файла с грамматикой

3 Метод генерации декларативных принтеров

В данном разделе описывается соответствие между XML-описаниями [1] и правилами грамматики, необходимые преобразования грамматики, а также процесс отбора правил, необходимых для получения принтера языка.

3.1 Определение поддеревьев и их свойств для правил грамматики

Текущий способ добавления новых языков в принтер-плагин предполагает создание описания всех значимых компонент языка. На рис. 11 представлено XML-описание структуры языка While. Количество таких структур в языке может быть большим, а значит, процесс создания такого описания может быть весьма трудоемким. Для каждой компоненты (*component*) принтера в XML-описании необходимо указать ее имя (*name*), класс внутреннего представления (*psiComponentClass*), а так же список поддеревьев (*subtree*) и их свойства. Данные свойства приведены ниже.

1. Имя поддерева (*name*).

```
<component name="While" psiComponentClass="PsiWhileStmt">
 <subtree name="condition" psiGetMethod="Bexpr"
      hasSeveralElements="false" isRequired="true" />
 <subtree name="body" psiGetMethod="StmtList"
      hasSeveralElements="false" isRequired="true" />
</component>
```

Рис. 11: XML-описание цикла с предусловием языка While

2. Метод класса внутреннего представления, возвращающий данное поддерево или коллекцию поддеревьев такого типа (*psiGetMethod*).
3. Является ли поддерево обязательным для синтаксической корректности структуры (*isRequired*).
4. Является ли поддерево набором однотипных элементов (*hasSeveralElements*).

Третье свойство с точки зрения принтера указывает на необходимость построения текстового представления для данного поддерева. Так, в языке Java блок else оператора ветвления необязателен, соответственно, для него может не существовать представления в синтаксическом дереве, значит, текстовое представление дерева оператора ветвления не будет содержать этот блок. В свою очередь, блок then является обязательным, и невозможность получения текстового представления для него приведет к невозможности получения текстового представления и для всего оператора ветвления. Четвертое свойство указывает, что поддерево является набором однотипных узлов синтаксического дерева. Например, тело функции состоит из нескольких операторов, и чтобы получить текстовое представление тела функции, необходимо получить текстовые представления для этих операторов и объединить их, расположив друг под другом. В случае поддерева body на рис. 11 hasSeveralElements="false", так как синтаксического дерево имеет отдельный узел для представления набора операторов (stmt_list), то есть body состоит из одного stmt_list.

Все указанные выше свойства можно получить прямо из грамматики языка. Название компоненты, класса синтаксического анализатора, метода для получение поддерева задаются с использованием логики генератора синтаксического анализатора. Поддеревьями будут являться нетерминалы в правой части. Поддерево будет обязательным, если оно не является элементом выбора ($A \to B \mid C \mid D$), элементом с условным вхождением (“?” — 0 или 1)

или элементом с повторением (“*” — любое число раз). Если нетерминал в правой части допускает множественное вхождение (“+” — 1 или более раз, “*” — любое число раз), то считается, что поддерево состоит из нескольких элементов.

Ниже приведено правило грамматики языка While, задающее ту же структуру языка, что и описание на рис. 11:

while_stmt ::= ’while’ ’(’ bexpr ’)’ ’do’ stmt_list ’od’

Данное правило грамматики имеет два нетерминала в правой части bexpr и stmt_list. При этом конструкция в правой части не является выбором ($B \mid C \mid D$), и каждый из нетерминалов не имеет условного или множественного вхождения. Таким образом, мы получаем те же данные, что указаны на рис. 11.

3.2 Необходимые преобразования правил грамматики

Грамматика языка содержит множество правил, однако некоторые из них могут не иметь представления в синтаксическом дереве или быть ненужными с точки зрения принтера. Генерация компонент принтера по таким правилам может привести к его некорректности или увеличению времени его работы. Чтобы полученный принтер был корректным и производительным, необходимо определить набор правил, задающих структуры языка, которые могут иметь различные текстовые представления, то есть будут форматироваться принтером. Такие правила будем называть *значимыми* (в контексте принтера). Остальные правила назовём *избыточными*, так как для них либо существует одно текстовое представление, либо не существует никакого. Поэтому генерация кода принтера для них бессмысленна, так как увеличивается время работы и генератора, и принтера.

Устранение левой рекурсии. Как уже было отмечено, грамматика, с которой работает Grammar-Kit, не может иметь

леворекурсивных правил (непосредственная левая рекурсия: $A \to A\alpha$). Существует способ устранения левой рекурсии [2], однако при этом создаются вспомогательные правила, которые не имеют представления в синтаксическом дереве, а значит, не могут иметь и текстового представления. Такие правила также считаются избыточными.

Другие правила грамматики. Грамматика языка может содержать особые правила, которые должны помечаться специальными модификаторами. Среди них: *private*, *external* — для правил с такими модификаторами не генерируются классы внутреннего представления. Так как компоненты принтера зависят от классов внутреннего представления и их методов, то генерация компонент по таким правилам не нужна.

3.3 Поиск правил, описывающих списочных структуры

Все структуры языка можно разделить на содержащие списочные поддеревья и не содержащие. Например, структура “вызов метода” имеет поддерево “список параметров”. В статье [1] описываются особенности форматирования таких структур, поэтому их необходимо обрабатывать иным образом, то есть генерировать другой код. Для этого необходимо заранее знать, какая перед нами структура. Из грамматики языка мы не можем узнать, являются ли поддеревья данной структуры списками. Для решения этой проблемы в грамматику был введен новый модификатор для правил *list*. Предполагается, что пользователь системы найдет списочные правила в грамматике и отметит их вручную. Также списки в некоторых языках могут иметь отличные от запятой разделители: “;”, “|” – и другие. Поэтому пользователю предлагается указывать еще и тип разделителя для списочной структуры, задаваемой данным правилом, если

этот разделитель отличен от запятой. Для этого требуется указать значение параметра listSep:

list alt_list ::= expr ('|' expr)* { listSep='|' }

3.4 Итоги

Таким образом, было установлено соотношение между XML-описаниями, использовавшимися для задания компонент принтера, и правилами грамматики, что позволяет получить требуемую нам информацию. Из всего множества правил были выделены те, которые задают структуры, форматирование которых будет изменяться. По полученному множеству правил далее будут генерироваться компоненты принтера.

4 Реализация

В данном разделе описывается реализация генератора принтеров и изменения в архитектуре принтер-плагина, позволяющие интегрировать в него полученные генератором принтеры, а так же некоторые ограничения, накладываемые на полученный код.

4.1 Изменение архитектуры принтер-плагина

Изначально плагин создавался с целью форматирования только программ на языке Java, и не предполагалось никаких механизмов расширения числа поддерживаемых языков. Метод добавления поддерживаемых языков [1] по сути предполагал создание аналогичного плагина, но для программ на другом языке. То есть помимо принтера требовалось реализовывать интерфейс взаимодействия плагина с пользователем. Структура кода была монолитной, и поэтому большую его часть приходилось переносить в каждый новый плагин. Намного удобнее было бы

свести все поддерживаемые языки в единую систему, чтобы пользователю не приходилось задумываться об установке отдельного плагина для каждого языка, а затем, при форматировании, выбирать соответствующий язык программирования. Для этого требовалось изменить архитектуру системы путем выделения языконезависимой части в отдельный модуль. При этом было необходимо минимизировать количество кода, который нужно будет генерировать (только принтер и компоненты). На рис. 12 представлена конечная архитектура принтер-плагина[5]. Для добавления поддержки нового языка в принтер-плагин требуется генерировать только языкозависимую часть (с помощью Grammar-Kit).

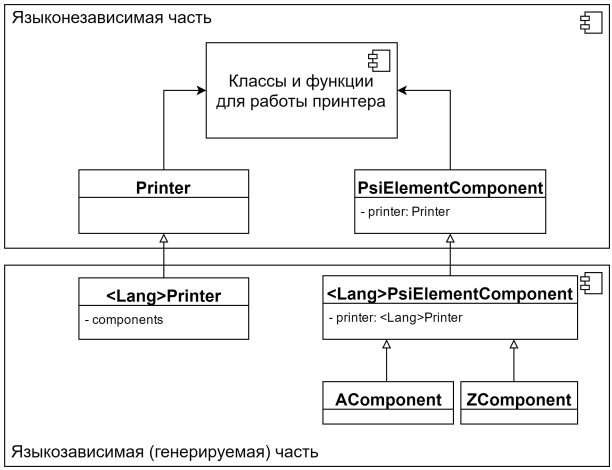


Рис. 12: Архитектура принтер-плагина

[5] https://bitbucket.org/igorozernykh/printerplugin, https://github.com/IgorOzernykh/printer-core

4.2 Генератор принтеров в плагине Grammar-Kit

После определения значимых правил, поддеревьев и их свойств происходит генерация кода компонент принтера. Генерация кода происходит с использованием текстовых файлов, которые имеют набор меток вида @TEXT@ для вставки соответствующей информации (рис. 13). Метки могут быть предназначены как для вставки названий классов, типов, так и для целых методов этих классов. Для каждого метода также имеется свой шаблон с набором меток. Таким образом, построение итогового кода осуществляется путём заполнения более мелких шаблонов нужной информацией и подстановкой их в более крупные.

Существует два основных шаблона: для обычных компонент и для списочных. Необходимость такого разграничения появилась из-за различного набор методов и их реализации. Упомянутый ранее модификатор *list*, применяемый к правилам грамматики, явно задает шаблон для генерации компонент. Аналогичным образом генерируется класс принтера.

4.3 Генерация файловой компоненты

Полученный в результате работы генератора принтер и его компоненты зависят от классов внутреннего представления программ. Так как классы внутреннего представления и соответствующие компоненты принтера генерируются по правилам грамматики, то мы можем гарантировать их корректное взаимодействие. Однако принтер вынужден также взаимодействовать и с классами, которые создаются вручную. Одним из них является класс внутреннего представления, описывающий структуру файла данного языка. Разработчик плагина для поддержки нового языка может задать данный класс разными способами, поэтому возникают трудности при генерации компоненты стандартным способом (разное число поддеревьев, которое можно получить при генерации

```
<..>
class @NAME_CC@Component(printer: @LANG@Printer)
    : @LANG@PsiElementComponent<@COMP_CLASS@, <..>>
{
    @DECL_TAGS@
    @GEN_SUBTREES@
    @GET_NEW_ELEM@
    @UPDATE_SUBTREES@
    @PREPARE_SUBTREES@
    @GET_TAGS@
    @IS_TEMPL_SUIT@
    @GET_TEMPLATE@
}
```

Рис. 13: Шаблон для генерации компонент

и которое указано в синтаксическом узле; несоответствие имен методов, их типов). Решением этой проблемы стало явное задание поддеревьев, которые будут сгенерированы. Например, поддеревья для структуры, описывающей файла языка Java, можно задать следующим образом:

fileSubtrees=‘‘PackageStatement, ImportList, Classes!*’’.

Названия поддеревьев соответствуют методам узла синтаксического дерева, “*” указывает, что возвращаемых тип — коллекция (массив или список), а “!” — является ли поддерево обязательным (аналог аннотаций @NotNull и @Nullable).

4.4 Ограничения

Для преобразования программного текста в объекты, которыми оперирует система, используется *фабрика элементов*. Она, как и класс, описывающий файл языка,

создается вручную. Причем заранее неизвестно, каким образом инстанциируются объекты данного класса, неизвестна сигнатура методов, поэтому, чтобы обеспечить взаимодействие принтера и фабрики элементов, необходимо вручную реализовать метод *createElementFromText* в классе принтера.

5 Апробация

Апробация производится для языков While и Erlang.

5.1 Принтер для языка While

Выбор данного языка был обусловлен тем, что количество структур в языке невелико и что для него уже существовал принтер, заданный с использованием метода [1] и позволяющий форматировать программы на данном языке. Грамматику потребовалось дополнить следующим образом. В заголовок необходимо было добавить имя пакета для генерируемого принтера (*printerPackage*), имя класса фабрики элементов (*factoryClass*), имя класса внутреннего представления для файла данного языка (*fileClass*). Эти значения необходимо указывать, чтобы верно определить зависимости генерируемых классов. Также в заголовке грамматики указывается список поддеревьев для файла данного языка (*fileSubtrees*), о чем было упомянуто в разделе 4. В данном случае, fileSubtrees="procList, stmtList". Далее необходимо было найти списочные правила и отметить их модификатором *list*. В языке While существует единственное списочное правило param_list. Таким образом, чтобы получить грамматику языка, потребовалось внести следующие изменения:

```
{
    <..>
```

```
    printerPackage="com.intellij.whileLang"
    factoryClass="com.intellij.whileLang.WhileElementFactory"
    fileClass="com.intellij.whileLang.WhileFile"
    fileSubtrees="procList, stmtList"
    <..>
}
<..>
list param_list ::= ref_expr? (COMMA ref_expr)*
<..>
```

Полученный в результате работы генератора принтер[6] корректно форматировал программы на данном языке. На рис. 14 представлена программа на языке While, которую требуется отформатировать. В качестве образцов форматирования будем использовать программы, представленные на рис. 15, *а* и 15, *б*. Результат работы принтера представлен на рис. 16, *а* и 16, *б* соответственно.

```
read(a); read(b);
res:=1;
while(b>0) do
  if (b%2=1) then
    res:=res*a; fi
  b:=b/2;
  a:=a*a;
od
write(res);
```

Рис. 14: Пример кода на языка While

[6] https://github.com/prettyPrinting/whileLang-idea-plugin

```
read(a);
read(b);
while (a <> b)
do
     if (a > b)
        then a := a - b;
        else b := b - a; fi od
write(a);
```

а)

```
read(a);
read(b);
while (a <> b) do
     if (a > b) then
          a := a - b;
     else
          b := b - a;
     fi
od
write(a);
```

б)

Рис. 15: Образцы форматирования

5.2 Принтер для языка Erlang

Язык Erlang был выбран исходя из того, что для этого языка существовала грамматика, по которой генерировался синтаксический анализатор и классы внутреннего представления с помощью плагина Grammar-Kit. Для генерации принтера по этой грамматики нужно было найти и отметить списочные правила, а также указать некоторую дополнительную информацию в заголовке файла, содержащего грамматику.

Однако полученный в результате реализованного генератора принтер некорректно форматировал некоторые структуры языка. Во-первых, возникла проблема с форматированием некоторых списочных структур. Связано это с тем, что в Erlang существуют списки, в которых одновременно могут быть различные типы разделителей. Рассмотрим конструкцию сопоставления с образцом (pattern matching) для списков: [X,Y,Z|ZS], где X, Y, Z – элементы списка, ZS – его “хвост”. X, Y, Z, ZS с точки зрения

```
read(a);
read(b);
res:=1;
while (b > 0)
do
  if (b % 2 = 1)
    then res := res * a; fi
  b := b / 2;
  a := a * a; od
write(res);
```

а)

```
read(a);
read(b);
res := 1;
while (b > 0) do
    if (b % 2 = 1) then
        res := res * a;
    fi
    b := b / 2;
    a := a * a;
od
write(res);
```

б)

Рис. 16: Результаты форматирования программы из рис. 14

синтаксического дерева являются выражениями, “,” и “|” – разделителями. Существующий способ форматирования поддерживает лишь единственный тип разделителей, поэтому при форматировании таких списков все разделители будут заменены на явно заданный или установленный по умолчанию разделитель (в данном случае “|” будет заменено на “,”), что приведет к некорректности конструкции.

Во-вторых, некоторые правила грамматики данного языка недостаточно выразительны, что не влияет на синтаксический разбор выражений языка, но негативно отражается на их форматировании. Например, бинарные операторы в грамматике Erlang задаются следующим образом:

```
private add_op ::= '+' |'-' | bor | bxor | bsl | bsr | or | xor
private mult_op ::= '/' |'*' | div 0| rem | band | and
```

По данным правилам не генерируются классы внутреннего представления, а значит, по бинарному выражению невозможно узнать, какой именно оператор в нем задан

(так как не генерируются соответствующие get-методы), то есть бинарный оператор не является поддеревом бинарных выражений. Поэтому и принтер не будет строить представления для операторов, то есть выбирать тот, который соответствует данному узлу дерева и который требуется подставить в шаблон. Таким образом, оператор будет явно "зашит" в шаблон для данной структуры: @expression + @expression, где @expression – место для вставки подвыражений. Применяя такой шаблон к выражению с оператором "-", получится выражение с оператором "+", то есть семантика программы изменится, чего не должно происходить при переформатировании. Чтобы решить эту проблему в грамматике языка Erlang, требуется полностью изменить способ задания бинарных выражений, что изменит классы внутреннего представления, а значит, может повлиять на работу плагина языка.

Однако некоторые структуры языка форматируются полученным принтером корректно. На рис. 17 представлен образец кода на языке Erlang. Применяя шаблоны, полученные из этого образца к коду, представленному на рис. 18, *а*, мы получаем код на рис. 18, *б*.

```
foo(A, 0) -> A;
foo(A, B) -> foo(B, 0).
```

Рис. 17: Образец кода на языке Erlang

5.3 Итоги

Апробация для языка Erlang показала, что не любая грамматика позволит получить принтер для языка, задаваемого этой грамматикой. Во-первых, для структур

```
b_not(true)->false;
b_not(false)->true.
b_and(true,true)->true;
b_and(_,_)->false.
b_or(false,false)->false;
b_or(_,_)->true.
```

а) Неотформатированный код

```
b_not(true) -> false;
b_not(false) -> true.
b_and(true, true) -> true;
b_and(_, _) -> false.
b_or(false, false) -> false;
b_or(_, _) -> true.
```

б) Код с заданным форматированием

Рис. 18: Пример форматирования программы на языке Erlang

языка, которые могут иметь различные текстовые представления, правила должны быть заданы таким образом, чтобы по ним генерировались классы внутреннего представления. Например, чтобы избежать проблем с описанными выше бинарными выражениями, требуется, чтобы отдельно было задано правило для бинарных операций, что позволило бы рассматривать их как поддерево бинарных выражений, а не как терминальные символы. Во-вторых, поддеревья структуры не должны пересекаться или быть вложенными друг в друга. По умолчанию это условие выполняется, однако пользователь может также явно задавать поддеревья[7]. В случае пересечения поддеревьев будет невозможно построить для них текстовые представления.

Заключение

В рамках этой работы были достигнуты следующие результаты:

[7] https://github.com/JetBrains/Grammar-Kit/blob/master/HOWTO.md#32-organize-psi-using-fake-rules-and-user-methods

– Предложен метод генерации декларативных принтеров по грамматике в форме Бэкуса-Наура.
– Реализован генератор принтеров в рамках проекта Grammar-Kit[8].
– Добавлена возможность интеграции сгенерированных принтеров с принтер-плагином для IDE IntelliJ IDEA.
– Получен принтер для языка While, позволяющий форматировать программы на этом языке.
– Установлено, что не по любой грамматике языка можно получить корректный принтер. Представлены требования к грамматике.

Список литературы

1. А. Подкопаев, А. Коровянский, И. Озерных. Языконезависимое форматирование текстов программ на основе сопоставления с образцом и синтаксических шаблонов. Научно-технические ведомости СПбГПУ 4' (224), 2015.
2. Д. Хопкрофт, Р. Мотвани, Д. Ульман. Введение в теорию автоматов, языков и вычислений, 2-е изд. *Вильямс*, Москва, 2002.
3. M.G.J. van den Brand, E. Visser. Generation of Formatters for Context-free Languages. ACM Trans. Softw. Eng. Meth. 5, 1, p1-41, 1996.
4. F. Corbo, C. Del Grosso, M. Di Penta. Smart Formatter: Learning Coding Style from Existing Source Code. ICSM, 2007.
5. P. Klint. A Meta-environment for Generating Programming Environments. ACM Trans. Softw. Eng. Meth. 2, 2, p176-201, 1993.
6. F. Nielson, H.R. Nielson, C. Hankin. Principles of Program Analysis. Springer-Verlag New York Inc., 1999.
7. A. Utkin, R. Shein, A. Kazakova. Applying Genetic Algorithms to Automatic Code Formatting. https://blog.jetbrains.com/clion/2015/11/applying-genetic-algorithms-to-automatic-code-formatting/. urldate = "09.05.2016"
8. T. Vollebregt, L.C.L. Kats, E. Visser. Declarative Specification of Template-Based Textual Editors. Proceedings of the Twelfth

[8] https://github.com/IgorOzernykh/Grammar-Kit

Workshop on Language Descriptions, Tools, and Applications. Article No. 8, 2012.

Диагностика синтаксических ошибок в динамически формируемом коде

Азимов Рустам Шухратуллович

Санкт-Петербургский государственный университет
rustam.azimov19021995@gmail.com

Аннотация Использование встроенных языков затрудняет статическую диагностику ошибок, что снижает надёжность целевых систем. В рамках проекта YaccConstructor разрабатывается платформа для статического анализа динамически формируемого кода, алгоритм синтаксического анализа которой строит лес разбора для всех корректных цепочек, но игнорирует синтаксические ошибки. По этой причине её практическая применимость ограничена. В данной работе представлен алгоритм синтаксического анализа динамически формируемого кода, расширенный механизмом обнаружения ошибок. Задача обнаружения всех синтаксических ошибок в рамках данного алгоритма является неразрешимой, поэтому предложенный механизм выделяет аппроксимацию сверху множества всех синтаксических ошибок. Также в данной работе доказана корректность модифицированного алгоритма и выполнена его реализация в проекте YaccConstructor.

Введение

При разработке сложных программных систем существует проблема недостатка выразительности и гибкости языков программирования общего назначения. Часто для решения данной проблемы помимо основного языка используют один

или несколько других (встроенных) языков. В данном подходе программа, написанная на основном языке, динамически формирует строковое представление кода на встроенном языке, и далее сформированная строка анализируется и выполняется специальными компонентами (база данных, веб-браузер) во время исполнения основной программы.

Как правило, динамически формируемые выражения, описывающие код программ на встроенных языках, конструируются с использованием конкатенаций строковых литералов в ветках условных операторов, циклах и рекурсивных процедурах. Это приводит к множеству возможных значений встроенного кода, что не позволяет в общем виде использовать статический анализ для проверки корректности формируемого выражения. В результате этого становятся недоступными такие типы функциональности, как информирование о синтаксических ошибках, автодополнение и подсветка синтаксиса. Отсутствие этих возможностей повышает вероятность ошибок, которые обнаруживаются лишь во время выполнения программы, а также усложняет процесс разработки и тестирования.

Существует ряд инструментов, позволяющих проводить анализ динамически формируемых строковых выражений: Java String Analyzer [2, 14], PHP String Analyzer [15], Alvor [4–6], IntelliLang [7], PHPStorm [8], Varis [9] (анализ этих технологий приведен в работе [18]). Большинство реализаций проводят диагностику синтаксических ошибок, но плохо расширяемы как в смысле поддержки других языков, так и в смысле решения новых задач. Существует алгоритм, описанный в статье [12] и реализованный как часть проекта YaccConstructor [11], который позволяет провести синтаксический анализ динамически формируемых выражений. В отличие от других инструментов, реализация данного алгоритма хорошо расширяема и строит конечное представление леса разбора относительно входной грамматики, которое может быть использовано

в дальнейшем семантическом анализе. Недостаток данного алгоритма заключается в отсутствии механизма диагностики синтаксических ошибок. Устранению этого недостатка и посвящена данная работа.

1 Постановка задачи

Целью данной работы является разработка механизма диагностики ошибок в алгоритме ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения, реализованного в проекте YaccConstructor [11].

Для достижения этой цели были сформулированы следующие задачи.

- Определить понятие синтаксической ошибки в терминах регулярной аппроксимации динамически формируемого выражения.
- Разработать механизм диагностики ошибок в синтаксическом анализе регулярной аппроксимации динамически формируемого выражения.
- Доказать корректность механизма.
- Реализовать предложенный механизм в рамках проекта YaccConstructor.
- Провести экспериментальное исследование.

2 Обзор

Алгоритм, доработке которого посвящена данная работа, основан на алгоритме RNGLR [17] (Right-Nulled Generalized LR). В свою очередь, RNGLR является модификацией алгоритма Generalized LR (GLR) [15], предназначенного для анализа естественных языков. GLR-алгоритм использует управляющие таблицы семейства LR (Left-to-right Rightmost) алгоритмов [20] с возможностью содержать несколько

действий в одной ячейке. Таким образом, при описании работы алгоритма ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения, необходимо рассмотреть все вышеперечисленные алгоритмы.

В данном разделе дано краткое описание работы алгоритма ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения. Также рассмотрено понятие синтаксической ошибки в алгоритмах LR-семейства и описан проект, в рамках которого велась разработка предложенного алгоритма.

2.1 Регулярная аппроксимация динамически формируемого выражения

Под регулярной аппроксимацией подразумевается приближение сверху множества значений динамически формируемого выражения некоторым регулярным выражением. В терминах, привычных для задач распознавания, проверка корректности выражений из аппроксимирующего множества формулируется как задача проверки включения регулярного языка в другой (как правило, контекстно-свободный язык), которая разрешима во многих важных на практике случаях [14]. Метод построения рассматриваемой аппроксимации, представленный в работе [21], реализован [13] в проекте YaccConstructor.

2.2 Алгоритм ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения

Чтобы понять принцип работы данного алгоритма, сначала рассмотрим алгоритмы LR-семейства, такие как LR(1), SLR(1), LALR(1). Все эти алгоритмы принимают на вход строку и контекстно-свободную грамматику. Если удалось

построить детерминированную управляющую таблицу по данной грамматике, то перечисленные алгоритмы позволяют ответить на вопрос о принадлежности произвольной строки языку, порождаемому данной грамматикой. Входная строка читается слева направо и в процессе чтения программа, выполняющая синтаксический анализ (*парсер*), меняет свои состояния по правилам, установленным в управляющей таблице. Нужная ячейка таблицы определяется текущим состоянием и правым контекстом (одним или несколькими следующими терминалами) входной строки. История состояний программы хранится в виде стека. Кроме перехода из состояния в состояние ячейка управляющей таблицы содержит действия, которые бывают двух видов: *сдвиг* (*push, shift*) — чтение следующей части входной строки, — и *свертка* (*reduce*) — применение одного из правил входной грамматики. Строка принимается алгоритмом лишь в том случае, если она была полностью обработана, и конечное состояние парсера — одно из заранее определенных *принимающих состояний*.

Иногда построить детерминированную таблицу не удается, и в одной ячейке управляющей таблицы могут быть конфликты следующих видов: сдвиг/свертка (shift/reduce) и свертка/свертка (reduce/reduce). Существует два подхода, применяемых для обработки таких неоднозначностей. Первый из них подразумевает разрешение конфликтов, а второй — анализ всех возможных последовательностей действий, предпринимаемых парсером. Алгоритмы семейства GLR, разработанные для анализа произвольных неоднозначных контекстно-свободных грамматик, используют второй подход. При работе этих алгоритмов порождается множество стеков состояний и деревьев разбора, для эффективного хранения которых используются специализированные структуры данных: структурированный в виде графа стек GSS (Graph Structured Stack) и компактное представление леса разбора SPPF [1] (Shared Packed Parse Forest).

Следует отметить, что оригинальный GLR-алгоритм не способен анализировать все контекстно-свободные грамматики, поэтому в работе [17] был предложен RNGLR-алгоритм, который является его расширением. RNGLR специальным способом обрабатывает *обнуляемые справа правила* входной грамматики (имеющие вид $A \to \alpha\beta$, где β выводит пустую строку ϵ). RNGLR, как и GLR, строит GSS "слоями", т.е. сначала выполняются все возможные операции свертки для текущего терминала, после чего осуществляется операция сдвига к следующему терминалу входной строки. Вершина GSS представляется в виде пары (s, l), где s — состояние парсера, а l — уровень (позиция текущего терминала во входном потоке).

Алгоритм, доработке которого посвящена данная работа, является расширением алгоритма RNGLR, способным производить синтаксический анализ регулярного множества входных строк. При анализе рассматриваемым алгоритмом регулярного множества, состоящего из единственной строки конечной длины, результат будет аналогичен результату анализа алгоритма RNGLR. В расширении вместо строки на вход подается конечный недетерминированный автомат с единственными начальным и конечным состояниями, который порождает регулярную аппроксимацию значений динамически формируемого выражения. Данный автомат представляется в виде ориентированного графа (далее *входного графа*) с вершинами — состояниями автомата и ребрами — переходами автомата. Строится *внутренний граф*, который получается из входного ассоциацией вершин GSS и некоторых коллекций с вершинами графа. Основная идея расширения заключается в перемещении по внутреннему графу и последовательном построении GSS. В качестве "слоев" выступают вершины внутреннего графа. Таким образом каждая вершина GSS хранит состояние парсера *state* и уровень *level* (который отождествляется с вершиной внутреннего графа). Теперь операция сдвига выполняется не

по одному токену, а по множеству токенов, нагруженных на дуги, исходящие из текущей вершины графа.

Для организации порядка обработки вершин внутреннего графа используется глобальная очередь Q. При добавлении новой вершины GSS сначала все *свертки длины 0* (*zero-reductions*) добавляются в очередь операций reduce, после этого выполняется операция сдвига следующих токенов со входа, а соответствующие вершины графа добавляются в очередь Q. Так как добавление нового ребра GSS может порождать новые свертки, то в очередь на обработку Q необходимо добавить вершину внутреннего графа, которой соответствует начальная вершина добавленного ребра (процесс построения GSS описан в алгоритме 3). Операции свертки проводятся вдоль путей в GSS; таким образом, если начальная вершина нового ребра ранее присутствовала в GSS, то необходимо заново вычислить свертки путей, проходящих через эту вершину (функция applyPassingReductions в алгоритме 2).

Кроме состояния анализатора *state* и уровня *level*, в вершине GSS хранится коллекция *проходящих сверток*. Проходящая свертка — это тройка $(startV, N, l)$, соответствующая свертке, чей путь содержит данную вершину GSS, а длина оставшейся части пути равна l. Проходящие свертки сохраняются в каждой вершине пути (кроме первой и последней) во время поиска путей в функции *makeReductions* (алгоритм 2).

В вершинах внутреннего графа хранятся следующие коллекции:

- *processed* — вершины GSS, для которых ранее были вычислены все операции push;
- *unprocessed* — вершины GSS, операции push для которых ещё только предстоит выполнить;
- *reductions* — очередь операций reduce, которые ещё только предстоит выполнить;

– *passingReductionsToHandle* — пары из вершины GSS и ребра GSS, вдоль которых необходимо обновить проходящие свертки.

Algorithm 1 Алгоритм ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения

```
 1: function PARSE(grammar, fsa)
 2:     inputGraph ← construct inner graph representation of fsa
 3:     parserSource ← generate RNGLR parse tables for grammar
 4:     if inputGraph contains no edges then
 5:         if parserSource accepts empty input then report success
 6:         else report failure
 7:     else
 8:         ADDVERTEX(inputGraph.startVertex, startState)
 9:         Q.Enqueue(inputGraph.startVertex)
10:         while Q is not empty do
11:             v ← Q.Dequeue()
12:             MAKEREDUCTIONS(v)
13:             PUSH(v)
14:             APPLYPASSINGREDUCTIONS(v)
15:         if ∃v_f : v_f.level = q_f and v_f.state is accepting then
16:             report success
17:         else report failure
```

Algorithm 2 Обработка вершины внутреннего графа

```
1: function PUSH(innerGraphV)
2:   U ← copy innerGraphV.unprocessed
3:   clear innerGraphV.unprocessed
4:   for all v_h in U do
5:     for all e in outgoing edges of innerGraphV do
6:       push ← calculate next state by v_h.state and token on e
7:       ADDEDGE(v_h, e.Head, push, false)
8:       add v_h in innerGraphV.processed
9: function MAKEREDUCTIONS(innerGraphV)
10:  while innerGraphV.reductions is not empty do
11:    (startV, N, l) ← innerGraphV.reductions.Dequeue()
12:    find the set of vertices X reachable from startV
13:     along the path of length (l − 1), or 0 if l = 0;
14:    add (startV, N, l − i) in v.passingReductions,
15:     where v is an i-th vertex of the path
16:    for all v_h in X do
17:      state_t ← calculate state by v_h.state and nonterminal N
18:      ADDEDGE(v_h, startV, state_t, (l = 0))
19: function APPLYPASSINGREDUCTIONS(innerGraphV)
20:  for all (v, e) in innerGraphV.passingReductionsToHandle do
21:    for all (startV, N, l) ← v.passingReductions.Dequeue() do
22:      find the set of vertices X,
23:       reachable from e along the path of length (l − 1)
24:      for all v_h in X do
25:        state_t ← calculate state by v_h.state and N
26:        ADDEDGE(v_h, startV, state_t, false)
```

Algorithm 3 Построение GSS

```
 1: function addVertex(innerGraphV, state)
 2:   v ← find a vertex with state = state in
 3:    innerGraphV.processed ∪ innerGraphV.unprocessed
 4:   if v is not null then                ▷ Вершина была найдена в GSS
 5:     return (v, false)
 6:   else
 7:     v ← create new vertex for innerGraphV with state state
 8:     add v in innerGraphV.unprocessed
 9:     for all e in outgoing edges of innerGraphV do
10:       calculate the set of zero-reductions by v
11:        and the token on e and
12:        add them in innerGraphV.reductions
13:     return (v, true)
14: function addEdge(v_h, innerGraphV, state_t, isZeroRed)
15:   (v_t, isNew) ← addVertex(innerGraphV, state_t)
16:   if GSS does not contain edge from v_t to v_h then
17:     edge ← create new edge from v_t to v_h
18:     Q.Enqueue(innerGraphV)
19:     if not isNew and v_t.passingReductions.Count > 0 then
20:       add (v_t, edge) in innerGraphV.passingReductionsToHandle
21:     if not isZeroRed then
22:       for all e in outgoing edges of innerGraphV do
23:         calculate the set of reductions by v
24:          and the token on e and
25:          add them in innerGraphV.reductions
```

2.3 Понятие синтаксической ошибки в алгоритмах LR-семейства

Конкретная формальная грамматика делит множество всевозможных строк на *принимаемые* (принадлежащие языку, порожденному данной грамматикой) и *непринимаемые*. Несоответствие синтаксическим правилам, определенным грамматикой, называется *синтаксической ошибкой*. Таким образом, в строке присутствует синтаксическая ошибка

(может быть не одна) тогда и только тогда, когда она не является принимаемой для соответствующей грамматики.

Алгоритмы из LR-семейства обладают свойством *корректного префикса* (*correct-prefix* property) [20], то есть если данные алгоритмы корректно определяют, принимается ли данная строка для рассматриваемой грамматики, то они обнаруживают ошибку на первом терминале, который образует из прочтенной части входной строки *некорректный префикс* (ни одна принимаемая строка не начинается с данного префикса).

Если алгоритмы из LR-семейства обнаружили синтаксическую ошибку, то возможен один из двух вариантов:

- парсер обработал всю входную строку, но итоговое состояние не принадлежит множеству принимающих состояний;
- парсер выполнил все операции свертки для текущего терминала, но не имеется ни одной операции сдвига к следующему терминалу входной строки.

Таким образом, данные алгоритмы продолжают свою работу до тех пор, пока обработанная часть строки является корректным префиксом.

2.4 Проект YaccConstructor

В рамках исследовательского проекта YaccConstructor [11] лаборатории языковых инструментов JetBrains на математико-механическом факультете СПбГУ проводятся исследования в области лексического, синтаксического анализа, а также статического анализа встроенных языков. Проект YaccConstructor является модульным инструментом, имеющим собственный язык спецификации грамматик. Также в нем объединены различные алгоритмы лексического и синтаксического анализа. В рамках проекта была создана платформа для статического анализа встроенного кода.

На рис. 1 представлена диаграмма последовательности, иллюстрирующая взаимодействие модулей платформы. Выделена компонента, осуществляющая синтаксический анализ множества значений динамически формируемого выражения.

Предыдущая реализация платформы игнорировала синтаксические ошибки, что усложняло процесс разработки и тестирования программных систем, использующих данную платформу. Однако это повлекло необходимость разработки механизма диагностики ошибок в рамках алгоритма синтаксического анализа, чему и посвящена данная работа.

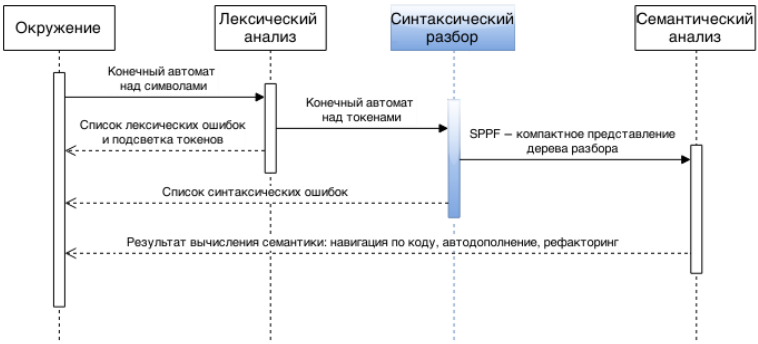


Рис. 1: Диаграмма последовательности: взаимодействие компонентов инструмента YaccConstructor (рисунок взят из работы [19])

3 Понятие синтаксической ошибки

Прежде чем описывать алгоритм диагностики ошибок, необходимо определить понятие синтаксической ошибки для алгоритма ослабленного синтаксического анализа регулярной

аппроксимации динамически формируемого выражения. На вход данного алгоритма подается не одна строка, а множество строк, поэтому необходимо обнаруживать ошибки во всех строках из данного множества. Для этого необходимо определить понятие синтаксической ошибки для алгоритма RNGLR. Определим сначала понятие синтаксической ошибки для GLR-алгоритма.

GLR-алгоритм, получивший на вход строку, не принимаемую соответствующей грамматикой, останавливает свою работу в двух случаях:

- синтаксический анализатор обработал всю входную строку, но среди множества итоговых состояний ни одно состояние не принадлежит множеству принимающих состояний;
- синтаксический анализатор выполнил все операции свертки для текущего терминала, но среди множества состояний текущего уровня не имеется ни одного состояния, по которому в управляющей таблице существует операция сдвига к следующему терминалу входной строки.

Таким образом, пока обработанная часть входной строки является корректным префиксом данной грамматики, хотя бы один стек из множества стеков состояний текущего уровня может быть продолжен с использованием операции сдвига к следующему терминалу входной строки. Другими словами, GLR-алгоритм, как и алгоритмы LR-семейства, обладает свойством корректного префикса. *Ошибочным терминалом* входной строки GLR-алгоритма будем называть первый терминал из необработанной части входной строки на момент обнаружения синтаксической ошибки (если обработана вся строка, то это специальный терминал, обозначающий конец строки).

Алгоритм RNGLR также обладает свойством корректного префикса, т.к. является расширением GLR-алгоритма

со специальной обработкой некоторых правил входной грамматики. Определение *ошибочного терминала* входной строки RNGLR-алгоритма аналогично определению ошибочного терминала GLR алгоритма.

Ввиду того, что алгоритм ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения, в отличие от RNGLR-алгоритма, принимает на вход не строку, а множество строк, то необходимо искать синтаксические ошибки во всем входном множестве строк. Для каждой отдельной строки понятие синтаксической ошибки совпадает с понятием синтаксической ошибки в RNGLR-алгоритме. Множество входных строк выражено конечным недетерминированным автоматом с единственными начальным и конечным состояниями. Данный автомат представляется в виде ориентированного графа (внутреннего графа) с терминалами на ребрах. В таком случае, обработанная часть входных данных — не префикс строки, а терминалы, ассоциированные с ребрами (то есть нагруженные на ребрах) в пути во внутреннем графе из начальной вершины в вершину, соответствующую текущему уровню. Рассматриваемый путь будем называть *префиксом внутреннего графа*. Если строка p, образованная терминалами на ребрах рассматриваемого пути префикса внутреннего графа P, является корректным префиксом эталонной грамматики, то префикс внутреннего графа P назовем *корректным*, иначе — *некорректным*.

Пусть непустая строка p является корректным префиксом для рассматриваемой грамматики, тогда при обработке данной строки RNGLR-алгоритмом будут прочитаны все её терминалы и на последнем уровне GSS будет хотя бы одна вершина. Пусть $(s_1, .., s_n)$ — все состояния вершин рассматриваемого последнего уровня GSS. Тогда в последней вершине V пути префикса внутреннего графа P $\exists(v_1, .., v_n)$ — GSS-вершины, где $v_i.state = s_i, \forall i \in [1, .., n]$. Префикс внутреннего графа P назовем *корректным для GSS-*

вершины v, если $v \in (v_1, .., v_n)$. Если строка p пустая и она является корректным префиксом для рассматриваемой грамматики, то префикс внутреннего графа P состоит из единственной начальной вершины внутреннего графа; назовем его корректным для GSS-вершин $(u_1, .., u_n)$, где $\forall i, u_i$ — либо начальная GSS-вершина, либо существует последовательность сверток длины 0, приводящая парсер из начального состояния в состояние $u_i.state$.

Назовем *ошибочным ребром* ребро внутреннего графа e, которое не нагружено специальным терминалом конца строки EOF, такое, что существует хотя бы один корректный префикс внутреннего графа P, заканчивающийся в вершине, из которой исходит данное ребро, но при добавлении ребра e в конец префикса P образуется некорректный префикс внутреннего графа. Если ребро e нагружено терминалом конца строки EOF, то ребро e является *ошибочным*, если существует хотя бы один корректный префикс внутреннего графа P, заканчивающийся в вершине, из которой исходит данное ребро, но строка, образованная последовательностью терминалов на ребрах пути P, не является принимаемой эталонной грамматикой. Аналогом ошибочного терминала во входной строке RNGLR-алгоритма является ошибочное ребро внутреннего графа.

Таким образом, цель диагностики ошибок в рамках синтаксического анализа регулярной аппроксимации множества значений динамически формируемого выражения заключается в обнаружении ошибочных ребер внутреннего графа и выводе корректных префиксов внутреннего графа, заканчивающихся в вершине, из которой исходит рассматриваемое ошибочное ребро, и становящихся некорректными при добавлении в конец этого ребра.

4 Механизм диагностики ошибок

Предлагаемый механизм диагностики ошибок состоит из двух частей:

- алгоритм синтаксического анализа регулярной аппроксимации динамически формируемых выражений (далее основной анализ) модифицируется, позволяя для каждой GSS вершины строить все корректные для нее префиксы внутреннего графа;
- после основного анализа с помощью построенных префиксов обнаруживаются ошибочные ребра внутреннего графа.

Далее в этой главе будут рассмотрены: компактное представление префиксов внутреннего графа, алгоритм построения корректных префиксов внутреннего графа для GSS-вершин и алгоритм диагностики ошибок, проводящий анализ построенных префиксов.

4.1 Алгоритм построения префиксов

4.2 Компактное представление префиксов внутреннего графа

Так как внутренний граф может иметь циклы, то множество различных префиксов внутреннего графа может быть бесконечным. В качестве компактного представления всех корректных префиксов внутреннего графа для вершины GSS используется ориентированный граф с выделенным множеством начальных вершин (далее *граф префиксов*). Из-за особенностей операций, используемых при построении графов префиксов, начальные вершины представляют не начало, а конец хранимых префиксов внутреннего графа в рассматриваемом графе префиксов. Каждая вершина в графе префиксов ассоциируется с ребром GSS или является специальной вершиной *EOP* (End Of Prefix). Ребра GSS, в свою очередь, будем делить на три вида:

Algorithm 4 Модификация построения GSS

```
1: function addEdge(v_h, innerGraphV, state_t, isZeroRed, pToAdd)
2:   (v_t, isNew) ← addVertex(innerGraphV, state_t)
3:   if GSS does not contain edge from v_t to v_h then
4:     edge ← create new edge from v_t to v_h
5:     Q.Enqueue(innerGraphV)
6:     if not isNew and v_t.passingReductions.Count > 0 then
7:       add (v_t, edge) in innerGraphV.passingReductionsToHandle
8:     if not isZeroRed then
9:       for all e in outgoing edges of innerGraphV do
10:        calculate the set of reductions by v and token on e
11:         and add them in innerGraphV.reductions
12:    V ← vertex of the prefix graph,
13:     associated with edge and connected to v_h.prefixes
14:    add V to initial vertexes of v_t.prefixes
15:  if pToAdd is not empty then
16:    e ← edge from v_t to v_h
17:    add all paths in pToAdd to e.paths
```

– *терминальное* ребро — ребро, порожденное операцией сдвига по какому-то терминалу t;
– *нетерминальное* — ребро, порожденное операцией свертки длины l, где $l > 0$;
– *обнуляемое* — ребро, порожденное операцией свертки длины l, где $l = 0$.

Будем говорить, что начальная вершина V графа префиксов GP_1 *соединена* с графом префиксов GP_2, если для любой начальной вершины U графа префиксов GP_2 существует ребро из вершины V в вершину U. Каждая начальная вершина графа префиксов, кроме EOP, соединена с одним графом префиксов. Вершина EOP не имеет исходящих дуг.

С каждым нетерминальным ребром ассоциируется множество путей в GSS, по которым произведена операция свертки для данного нетерминального ребра. Будем говорить

Algorithm 5 Модификация операции сдвига

```
1: function PUSH(innerGraphV)
2:   U ← copy innerGraphV.unprocessed
3:   clear innerGraphV.unprocessed
4:   for all v_h in U do
5:     for all e in outgoing edges of innerGraphV do
6:       push ← calculate next state by v_h.state and token on e
7:       pathsToAdd ← empty set
8:       ADDEDGE(v_h, e.Head, push, false, pathsToAdd)
9:       add v_h in innerGraphV.processed
```

что данное нетерминальное ребро *порождает* каждый путь из рассмотренного множества путей GSS.

Рассмотрим путь $(V_1, .., V_n)$ в графе префиксов GP как последовательность вершин графов префиксов. Удалим вершину EOP, если она присутствует, а также заменим все вершины в данном пути на ребра GSS, с которыми они ассоциируются. Получим последовательность $(e_1, .., e_n)$ ребер GSS. *Раскрытием* данной последовательности будем называть последовательность, получающуюся в результате применения следующих действий:

- все нетерминальные ребра GSS e заменяются на последовательность ребер, соответствующую одному из порожденных ребром e пути;
- все обнуляемые ребра GSS удаляются из последовательности.

Если после конечного числа раскрытий в последовательности останутся только терминальные ребра GSS, то будем говорить, что изначальный путь в графе префиксов *сводится* к строке, получающейся инвертированием этой последовательности терминальных ребер и их замены на терминалы, с которыми они ассоциированы. Если полученная строка получается заменой ребер в пути префикса внутреннего графа P на терминалы,

Algorithm 6 Модификация операции свертки

```
 1: function makeReductions(innerGraphV)
 2:   while innerGraphV.reductions is not empty do
 3:     (startV, N, l) ← innerGraphV.reductions.Dequeue()
 4:     find the set of vertices 𝒳 reachable from startV
 5:      along the path of length (l − 1), or 0 if l = 0;
 6:     add (startV, N, l − i) in v.passingReductions,
 7:      where v is an i-th vertex of the path
 8:     for all v_h in 𝒳 do
 9:       state_t ← calculate state by v_h.state and N
10:       if l > 0 then
11:         pToAdd ← paths, by which v_h is reachable from startV
12:       else pToAdd ← empty set
13:       addEdge(v_h, startV, state_t, (l = 0), pToAdd)
```

которыми нагружены эти ребра, то путь в графе префиксов *сводится* к префиксу внутреннего графа P. Будем говорить, что граф префиксов GP *порождает* префикс внутреннего графа P, если $\exists(V_1, .., V_n, EOP)$ — путь в графе префиксов GP, где V_1 — одна из начальных вершин графа префиксов GP, который сводится к префиксу внутреннего графа P. В графе префиксов также могут быть циклы, что позволяет сводиться к бесконечному множеству префиксов внутреннего графа.

Алгоритм модифицирует алгоритм ослабленного синтаксического анализа регулярной аппроксимации динамического выражения. Вершины GSS ассоциируются с коллекцией $prefixes$, которая представляет граф всех корректных префиксов внутреннего графа для данной вершины. В графе префиксов начальной GSS-вершины создается вершина EOP. Нетерминальные ребера GSS ассоциируются с коллекцией $paths$ — множеством путей в GSS, порождаемых ими. В функцию $addEdge$ добавляется параметр $pToAdd$ — множество путей в GSS, которое необходимо добавить ко множеству путей, порождаемых

Algorithm 7 Модификация обработки проходящих редукций

```
1: function applyPassingReductions(innerGraphV)
2:   for all (v, e) in innerGraphV.passingReductionsToHandle do
3:     for all (startV, N, l) ← v.passingReductions.Dequeue() do
4:       find the set of vertices 𝒳,
5:        reachable from e along the path of length (l − 1)
6:       for all v_h in 𝒳 do
7:         state_t ← calculate new state by v_h.state and N
8:         pToAdd ← paths, by which v_h is reachable from startV
9:         addEdge(v_h, startV, state_t, false, pToAdd)
```

Algorithm 8 Алгоритм диагностики ошибок

```
1: function findErrors(inputGraph, parserSource)
2:   𝒬.Enqueue(inputGraph.startVertex)
3:   while Q is not empty do
4:     v ← 𝒬.Dequeue()
5:     processVertex(v)
6:   while F is not empty do
7:     e ← ℱ.Dequeue()
8:     processEOF(e)
```

ребром $v_t \xrightarrow{e} v_h$. При обработке терминальных или обнуляемых ребер GSS $pToAdd$ является пустым. Создается начальная вершина графа префиксов $v_t.prefixes$, ассоциированная с e и соединенная с $v_h.prefixes$. Функция $addVertex$ не изменилась.

После описанных модификаций алгоритм синтаксического анализа регулярной аппроксимации динамически формируемого выражения для каждой вершины GSS дополнительно конструирует все корректные для нее префиксы внутреннего графа.

Algorithm 9 Анализ конечных ребер внутреннего графа

```
 1: function processEOF(edge)
 2:   acceptedPrefixes ← all prefixes graphs
 3:    from all GSS vertexes of edge.Tail with accepted state;
 4:   withCycle ← all cyclical prefixes graphs in acceptedPrefixes
 5:   withoutCycle ← all non-cyclical graphs in acceptedPrefixes
 6:   notAcceptedPrefixes ← all prefixes graphs
 7:    from all GSS vertexes of edge.Tail with not accepted state;
 8:   for all p in notAcceptedPrefixes do
 9:     if prefixes graph p has a cycle then
10:       add edge to probErrors
11:       add p to probErrors[edge]
12:     else
13:       for all prefixPath in paths of prefixes graph p do
14:         pathG ← prefixes graph generated by prefixPath
15:         if withoutCycle does not have any prefixes graph
16:            with equivalent to prefixPath path  then
17:           if withCycle is not empty then
18:             add edge to probErrors
19:             add pathG to probErrors[edge]
20:           else
21:             add edge to errors
22:             add pathG to errors[edge]
```

4.3 Алгоритм диагностики ошибок

Данный алгоритм получает на вход внутренний граф с построенными в ходе основного синтаксического анализа структурами (в том числе и префиксами внутреннего графа GSS вершин), а также сгенерированные RNGLR-таблицы. Алгоритм делает обход внутреннего графа и для каждого исходящего из вершины внутреннего графа ребра анализирует множества графов префиксов соседних вершин.

Для обхода внутреннего графа используется глобальная очередь Q. Все ребра внутреннего графа, ведущие не в конечную вершину, обрабатываются в функции

Algorithm 10 Анализ неконечных ребер внутреннего графа

```
 1: function processVertex(innerGraphV)
 2:   for all e in outgoing edges of innerGraphV do
 3:     if token on e is EOF then
 4:       F.Enqueue(e)
 5:     else
 6:       if e.Head was not processed then
 7:         Q.Enqueue(e.Head)
 8:       headPrefixes ← all prefixes graphs
 9:        from all GSS vertexes of e.Head;
10:       pushedPrefixes ←
11:        all prefixes graphs, to which connected
12:        some initial vertex of prefixes graph in headPrefixes,
13:        associated with terminal edge;
14:       withCycle ←
15:        all cyclical prefixes graphs in pushedPrefixes
16:       withoutCycle ←
17:        all non-cyclical prefixes graphs in pushedPrefixes
18:       notPushedPrefixes ←
19:        prefixes graphs from GSS vertexes of innerGraphV,
20:        without prefixes graphs from pushedPrefixes;
21:       for all p in notPushedPrefixes do
22:         if prefixes graph p has a cycle then
23:           add e to probErrors
24:           add p to probErrors[e]
25:         else
26:           for all pPath in paths of prefixes graph p do
27:             pathG ← prefixes graph generated by pPath
28:             if withoutCycle does not have then
29:                any prefixes graph with
30:                equivalent to prefixPath path
31:               if withCycle is not empty then
32:                 add e to probErrors
33:                 add pathG to probErrors[e]
34:               else
35:                 add e to errors
36:                 add pathG to errors[e]
```

$processVertex$. Для ребер, ведущих в конечную вершину внутреннего графа, используется глобальная очередь F, с последующей обработкой в функции $processEOF$.

В результате работы алгоритма все ребра из множества $errors$ являются ошибочными, а ребра из множества $probErrors$ — возможно ошибочными. То есть множество всех ошибочных ребер принадлежит объединению двух данных множеств. Кроме того, с элементами этих множеств ассоциируются множества графов префиксов (элемент e множества $errors$ или множества $probErrors$ ассоциируется со множеством $errors[e]$ или $probErrors[e]$ соответственно), которые порождают корректные префиксы, заканчивающиеся в вершине, из которой исходит рассматриваемое ребро. Данные множества могут быть использованы при создании сообщения пользователю о возможных ошибках динамически формируемого выражения. Все префиксы, порождаемые графами префиксов из множества $errors[e]$, становятся некорректными при добавлении в конец ребра e. Все префиксы, порождаемые графами префиксов из множества $probErrors[e]$, возможно становятся некорректными при добавлении в конец ребра e. Но множество всех корректных префиксов внутреннего графа, заканчивающихся в вершине, из которой исходит ребро e, и становящихся некорректными при добавлении ребра e в конец, порождается хотя бы одним графом префиксов из объединения множеств $errors[e]$ и $probErrors[e]$, где множество $errors[e]$ пусто, если $e \notin errors$, и множество $probErrors[e]$ пусто, если $e \notin probErrors$.

5 Корректность механизма диагностики ошибок

5.1 Корректность алгоритма построения префиксов

В данном разделе приведено доказательство того, что построенные графы префиксов порождают только корректные

для соответствующей вершины стека префиксы внутреннего графа.

Теорема 1. *Каждый корректный для GSS-вершины v префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_l)$ во внутреннем графе, порождается графом префиксов $v.prefixes$, построенным алгоритмом построения префиксов.*

Доказательство. Обозначим входную эталонную грамматику через G. Пусть p — строка, образованная терминалами на ребрах пути P. Докажем теорему методом математической индукции по l — количеству вершин в пути P.

База $l = 1$**.** Тогда p — пустая строка. Так как рассматриваемый префикс внутреннего графа корректен для GSS-вершины v, то либо v — начальная вершина GSS v_0, либо существует последовательность сверток длины 0, приводящая парсер из начального состояния $v_0.state$ в состояние $v.state$. В первом случае $v = v_0$, а значит, среди начальных вершин графа префиксов $v.prefixes$ имеется вершина EOP. Так как путь в графе префиксов $v.prefixes$, состоящий из единственной начальной вершины EOP, сводится к пустой строке, то граф префиксов $v.prefixes$ порождает префикс внутреннего графа, соответствующий пути P. Во втором случае в GSS существует путь из вершины v в вершину v_0, состоящий из обнуляемых ребер $(e_1, .., e_a)$. Значит, в графе префиксов $v.prefixes$ существует путь $S = (E_1, .., E_a, EOP)$, где E_1 — начальная вершина графа префиксов $v.prefixes$ и $\forall i \in [1, .., a], E_i$ — ассоциируется с обнуляемым ребром e_i. Путь S также сводится к пустой строке, следовательно, граф префиксов $v.prefixes$ порождает префикс внутреннего графа, соответствующий пути P. База доказана.

Индукционный переход. Пусть теорема доказана для всех префиксов внутреннего графа, соответствующих путям с количеством вершин не большим k, где $k > 0$. Докажем

теорему для префикса внутреннего графа, соответствующего пути $P = (V_1, .., V_{k+1})$.

Так как рассматриваемый префикс внутреннего графа корректен для GSS-вершины v, то непустая строка p — корректный префикс для грамматики G. Значит, получив на вход строку p, алгоритм RNGLR обработает все терминалы этой строки и построит GSS, имеющий вершины $(v_1, .., v_n)$ на последнем уровне. Причем, $\exists i : v_i.state = v.state$. По построению GSS существует хотя бы один путь из вершины v_i в начальную GSS-вершину v_0. Пусть $Q = (w_1, .., w_m)$ — один из таких путей, где $w_1 = v_i$, а $w_m = v_0$. Рассмотрим ребро $e = (w_1, w_2)$. Ребро e может быть терминальным, нетерминальным или обнуляемым.

Если e — терминальное ребро, то пусть оно ассоциировано с терминалом t, нагруженным на ребро (V_k, V_{k+1}) внутреннего графа. Префикс внутреннего графа, соответствующий пути $P' = (V_1, .., V_k)$, корректен в силу корректности префикса, соответствующего пути P. То есть в вершине внутреннего графа V_{k+1} существуют вершины GSS $(u_1, .., u_b)$, для которых префикс внутреннего графа, соответствующий P', корректен. Тогда $\exists j \in [1, .., b] : u_j.state = w_2.state$. По индукционному предположению префикс внутреннего графа, соответствующий пути P', порождается графом префиксов $u_j.prefixes$. Следовательно, существует путь $R = (R_1, .., R_c, EOP)$ графа префиксов $u_j.prefixes$, где R_1 — начальная вершина графа префиксов, такой, что путь R сводится к строке p', где p' — строка p без последнего терминала t. Так как в построенном RNGLR-алгоритмом GSS присутствует ребро $e = (w_1, w_2)$, то в управляющих таблицах, соответствующих грамматике G, существует операция сдвига по терминалу t из состояния $w_2.state$ в состояние $w_1.state$. Значит, при обработке основным анализом GSS-вершины u_j будет добавлено ребро $e' = (v, u_j)$, т.к. $u_j.state = w_2.state$ и $v.state = w_1.state$. Значит, в граф префиксов $v.prefixes$ будет добавлена начальная вершина E', соответствующая

терминальному ребру $e' = (v, u_j)$ и соединенная с графом префиксов $u_j.prefixes$. Поэтому в графе префиксов $v.prefixes$ существует путь $R' = (E', R_1, .., Rc, EOP)$, который сводится к строке p. Таким образом, граф префиксов $v.prefixes$ порождает префикс внутреннего графа, соответствующий пути P.

Если e — нетерминальное ребро, то пусть оно ассоциировано с нетерминалом N и путями GSS $(P_1, .., P_d)$. Пусть в GSS, сконструированным RNGLR-алгоритмом, вершина w_2 была создана после обработки первых $s < k$ терминалов строки p. Обозначим через p_1 строку, составленную из первых s терминалов строки p, а через p_2 — строку, составленную из последних $(k - s)$ терминалов строки p. Так как префикс, соответствующий пути $P_s = (V_1, .., V_{s+1})$ — корректен, то в вершине V_{s+1} существует GSS-вершина h такая, что этот префикс корректен для GSS-вершины h и $h.state = w_2.state$. Количество вершин пути P_s равно $(s + 1)$, что не превышает k, поэтому по индукционному предположению префикс внутреннего графа, соответствующий пути P_s, порождается графом префиксов $h.prefixes$. Значит, существует путь $R = (R_1, .., R_c, EOP)$ в графе префиксов $h.prefixes$, где R_1 — начальная вершина графа префиксов, такой, что путь R сводится к строке p_1. В GSS, построенном RNGLR-алгоритмом, имеется нетерминальное ребро e, значит, в GSS, построенным основным анализом, имеется нетерминальное ребро $e' = (v, h)$, ассоциированное с нетерминалом N, так как $h.state = w_2.state$ и $v.state = w_1.state$. Значит, в граф префиксов $v.prefixes$ будет добавлена вершина E', ассоциированная с нетерминальным ребром e'. Среди путей GSS, ассоциированных с ребром e', будут также присутствовать пути, аналогичные путям $(P_1, .., P_d)$ (последовательности состояний вершин GSS в таких путях совпадают, соответствующие ребра будут иметь одинаковый вид, соответствующие

терминальные ребра будут ассоциированы с одинаковыми терминалами, а соответствующие нетерминальные ребра — с одинаковыми нетерминалами). Так как в GSS, построенном RNGLR-алгоритмом, нет циклов (кроме циклов, состоящих только из обнуляемых ребер), а количество ребер конечно, то из нетерминального ребра e с помощью конечного числа раскрытий можно получить последовательность терминальных ребер, соответствующих строке p_2. Значит и в GSS, построенном основным анализом, можно с помощью аналогичного конечного числа раскрытий из ребра e' получить последовательность терминальных ребер, соответствующих строке p_2. Таким образом, в графе префиксов $v.prefixes$ существует путь $(E', R_1, .., R_c, EOP)$, который сводится к строке p. Следовательно граф префиксов $v.prefixes$ порождает префикс внутреннего графа, соответствующий пути P.

Если e — обнуляемое ребро, обозначим $e_f = (w_f, w_{f+1})$ — первое не обнуляемое ребро в пути Q. Префикс внутреннего графа, соответствующий пути P, является корректным для GSS-вершины w_f. Граф префиксов $w_f.prefixes$ порождает префикс внутреннего графа, соответствующий пути P, так как в GSS, построенном RNGLR-алгоритмом, существует путь из вершины w_f, который начинается с терминального или нетерминального ребра, а такие случаи уже рассмотрены. Пусть $(F_1, .., F_g, EOP)$ — путь графа префиксов $w_f.prefixes$, который сводится к строке p. Тогда $\exists(Z_1, .., Z_{f-1}, F_1, .., F_g, EOP)$ — путь графа префиксов $w_1.prefixes$, где $\forall j \in [1, .., (f-1)], Z_j$ ассоциируется с обнуляемым ребром (w_j, w_{j+1}). Значит, этот путь также сводится к строке p. А так как $w_1.state = v.state$, то $w_1 = v$. Следовательно, граф префиксов $v.prefixes$ порождает префикс внутреннего графа, соотвествующий пути P. □

Для доказательства того, что каждый префикс внутреннего графа, порождаемый графом префиксов

$v.prefixes$, является корректным для GSS-вершины v, нам понадобится следующая лемма.

Лемма. *Пусть GSS-вершина v_m корректна для префикса внутреннего графа, соответствующего пути $P_1 = (V_1, .., V_l)$, терминалы на ребрах которого образуют строку $p_1 = (t_1, .., t_{l-1})$ ($p_1 = \epsilon$, если $l = 1$). Пусть $\exists Q = (v_1, .., v_m)$ — путь в GSS, в котором $E = (e_1, .., e_{m-1})$ — последовательность ребер, сводящаяся за конечное число раскрытий к строке $p_2 = (t_l, .., t_{l+r-1})$ ($p_2 = \epsilon$, если $r = 0$). Тогда префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_l, V_{l+1}, .., V_{l+r})$, корректен для GSS-вершины v_1, и терминалы на ребрах пути P образуют строку $p = p_1 \cdot p_2$.*

Доказательство. Докажем лемму методом математической индукции по длине строки p_2, равной r.

База $r = 0$ или $r = 1$. Так как GSS-вершина v_m корректна для префикса внутреннего графа, соответствующего пути P, то RNGLR-алгоритм, получив на вход строку p_1, обработает все терминалы этой строки и на последнем уровне построенного GSS создаст вершину w_m такую, что $w_m.state = v_m.state$.

Если $r = 0$, то строка $p_2 = \epsilon$, следовательно, $\forall i, e_i$ — обнуляемо. Следовательно, $\forall i, v_i$ ассоциировано с V_l. Таким образом, существует последовательность операций сверток длины 0, приводящих анализатор из состояния $v_m.state$ в состояние $v_1.state$. Следовательно, в GSS, построенном RNGLR-алгоритмом, в результате применения аналогичной последовательности операций сверток длины 0 на последнем уровне будет существовать путь из обнуляемых ребер, начинающийся в GSS-вершине w_1 и заканчивающийся в вершине w_m, где $w_1.state = v_1.state$. Из существования на последнем уровне GSS, построенного RNGLR-алгоритмом, вершины w_1 следует, что префикс внутреннего графа, соответствующий пути P_1, также является корректным и для

GSS-вершины v_1. А так как в данном случае путь P_1 равен пути P, то утверждение леммы при $r = 0$ доказано.

Если $r = 1$, то строка p_2 — не пустая, а значит, среди ребер в последовательности E существует ровно одно не обнуляемое ребро. Пусть j такое, что $e_j = (v_j, v_{j+1})$ — не обнуляемое ребро последовательности E. Если $j = (m - 1)$, то префикс внутреннего графа, соответствующий пути P_1, корректен для GSS-вершины v_{j+1}, так как $v_{j+1} = v_m$. Если $j < (m - 1)$, то существует путь в GSS из обнуляемых ребер, начинающийся в вершине v_{j+1} и заканчивающийся в вершине v_m. Последовательность ребер в данном пути сводится к пустой строке. Так как утверждение леммы при $r = 0$ было доказано, то префикс внутреннего графа, соответствующий пути P_1, является корректным для GSS-вершины v_{j+1}. Теперь рассмотрим не обнуляемое ребро e_j. Если e_j — терминальное ребро, то оно ассоциировано с терминалом t_l. Тогда в управляющих таблицах существует операция сдвига по терминалу t_l из состояния $v_{j+1}.state$ в состояние $v_j.state$. Значит, RNGLR-алгоритм, обработав строку p_1, может выполнить операцию сдвига из GSS-вершины с состоянием $v_{j+1}.state$ в вершину с состоянием $v_j.state$. Значит, префикс внутреннего графа, соответствующий пути $(V_1, .., V_l, V_{l+1})$, является корректным для GSS-вершины v_j. Если $j = 1$, то $v_j = v_1$ и утверждение леммы доказано. Пусть $j > 1$. Последовательность ребер $(e_1, .., e_{j-1})$ за конечное число раскрытий сводится к пустой строке. Так как случай при $r = 0$ был рассмотрен, то доказано, что префикс внутреннего графа, соответствующий пути P, корректен для GSS-вершины v_1. Теперь пусть e_j — нетерминальное ребро. Покажем, что префикс внутреннего графа, соответствующий пути P, также является корректным и для GSS-вершины v_j.

Рассмотрим ту конечную последовательность раскрытий, примененную к последовательности ребер E, после которой останется единственное терминальное ребро, ассоциированное с терминалом t_l. Существует два

варианта: либо после первого применения операции раскрытия в последовательности останется единственное терминальное ребро $e_{term} = (u_{term}, v_{j+1})$, ассоциированное с терминалом t_l, либо при применении первых $i > 0$ операций раскрытия последовательность ребер GSS будет состоять из единственного нетерминального ребра, причем $\forall d \in [1, .., i]$, после применения первых d операций раскрытия последовательность состоит из единственного ребра $S_d = (h_d, g_d)$. Первый вариант означает, что префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_l, V_{l+1})$, является корректным для GSS-вершины u_{term}. А так как в GSS, построенном основным анализом, существует ребро $e_j = (v_j, v_{j+1})$, ассоциированное с нетерминалом N и с GSS-путем, в котором присутствует единственное ребро e_{term}, то префикс внутреннего графа, соответствующий пути P, также является корректным и для GSS-вершины v_j. Рассмотрим второй вариант. Пусть $e_j = S_0 = (h_0, g_0)$, тогда $\forall d \in [1, .., i]$, при применении d-ой операции раскрытия нетерминальное ребро S_{d-1} заменяется на последовательность ребер, соответствующих одному из порожденных ребром S_{d-1} пути, начинающегося в вершине w_d и закачивающегося в вершине g_{d-1}, причем w_d, как и v_j, принадлежит вершине внутреннего графа V_{l+1}. Пусть $w_0 = v_j$. Так как в данных путях ребро S_d является единственным не обнуляемым ребром, то $\forall d \in [1, .., i]$, существует путь из вершины w_d в вершину h_d и существует путь из вершины g_d в вершину g_{d-1}, оба из которых имеют только обнуляемые ребра. При применении $(i + 1)$-ой операции раскрытия нетерминальное ребро S_i заменятся на последовательность ребер, соответствующую пути в GSS из вершины w_{term} в вершину g_i, причем в этом пути единственное не обнуляемое ребро — это терминальное ребро $S_{term} = (h_{term}, g_{term})$, ассоциированное с терминалом t_l. Так как $\forall d \in [1, .., i]$, существует путь из вершины g_d в вершину g_{d-1}, состоящий только из обнуляемых ребер, и

существует путь из вершины g_{term} в вершину g_i, состоящий только из обнуляемых ребер, то также существует путь из вершины g_{term} в вершину $g_0 = v_{j+1}$ и существует путь из вершины g_d в вершину $g_0 = v_{j+1}$, состоящие только из обнуляемых ребер. А так как префикс внутреннего графа, соответствующий пути P_1, является корректным для GSS-вершины v_{j+1}, то префикс внутреннего графа, соответствующий пути P_1, также является корректным и для GSS-вершин из множества $\{g_1, .., g_i, g_{term}\}$. Так как в GSS, построенным основным анализом, существует терминальное ребро $S_{term} = (h_{term}, g_{term})$, ассоциированное с терминалом t_l, то префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_l, V_{l+1})$, является корректным для GSS-вершины h_{term}. А так как существует путь из вершины w_{term} в вершину h_{term}, то префикс внутреннего графа, соответствующий пути P, также является корректным и для GSS-вершины w_{term}. Аналогичным образом доказывается, что $\forall d \in [0, ..i]$, префикс внутреннего графа, соответствующий пути P, является корректным для GSS-вершины w_d. А так как $w_0 = v_j$, то, в частности, префикс внутреннего графа, соответствующий пути P, является корректным для GSS-вершины v_j.

Таким образом мы показали, что префикс внутреннего графа, соответствующий пути P, является корректным для GSS-вершины v_j. А так как существует путь из вершины v_1 в вершину v_j, состоящий из обнуляемых ребер, то из доказательства утверждения леммы при $r = 0$ следует, что префикс внутреннего графа, соответствующий пути P, также является корректным и для GSS-вершины v_1, что и доказывает утверждение леммы при $r = 1$. База индукции доказана.

Индукционный переход. Пусть утверждение леммы доказано для всех строк p_2 длины, меньшей r. Докажем утверждение леммы для строки p_2 длины $r > 1$.

Так как строка p_2 не пустая, то среди ребер в последовательности E существует хотя бы одно не обнуляемое

ребро. Пусть j такое, что $e_j = (v_j, v_{j+1})$ — последнее не обнуляемое ребро последовательности E. Если $j = (m-1)$, то префикс внутреннего графа, соответствующий пути P_1, корректен для GSS-вершины v_{j+1}, так как $v_{j+1} = v_m$. Если $j < (m-1)$, то существует путь в GSS из обнуляемых ребер, начинающийся в вершине v_{j+1} и заканчивающийся в вершине v_m. Последовательность ребер в данном пути сводится к пустой строке. Значит, по индукционному предположению, префикс внутреннего графа, соответствующий пути P_1, является корректным для GSS-вершины v_{j+1}. Теперь рассмотрим не обнуляемое ребро e_j.

Если e_j — терминальное ребро, то оно ассоциировано с терминалом t_l. Тогда в управляющих таблицах существует операция сдвига по терминалу t_l из состояния $v_{j+1}.state$ в состояние $v_j.state$. Значит, RNGLR-алгоритм, обработав строку p_1, может выполнить операцию сдвига из GSS-вершины с состоянием $v_{j+1}.state$ в вершину с состоянием $v_j.state$. Значит, префикс внутреннего графа, соответствующий пути $(V_1, .., V_l, V_{l+1})$, является корректным для GSS-вершины v_j. Так как строка p_2 имеет длину $r > 1$, то $j > 1$. Последовательность ребер $(e_1, .., e_{j-1})$ за конечное число раскрытий сводится к строке $p_j = (t_{l+1}, .., t_{l+r-1})$. Так как длина строки p_j равна $(r-1)$, то по индукционному предположению префикс внутреннего графа, соответствующий пути P, корректен для GSS-вершины v_1. Утверждение леммы доказано.

Если e_j — нетерминальное ребро. Тогда рассмотрим два случая.

Первый случай: $\exists i \in [1, .., (j-1)], e_i$ — не обнуляемое ребро. Тогда ребро e_j в последовательности E после рассматриваемого конечного числа раскрытий сводится к непустой строке $(t_l, .., t_{l+h-1})$ длины $h < r$. А так как префикс внутреннего графа, соответствующий пути P_1, является корректным для GSS-вершины v_{j+1}, то, по

индукционному предположению, префикс внутреннего графа, соответствующий пути $(V_1, .., V_l, V_{l+1}, .., V_{l+h})$, является корректным для GSS-вершины v_j. А так как последовательность ребер $(e_{j-1}, .., e_1)$ за конечное число раскрытий сводится к строке $(t_{l+h}, .., t_{l+r-1})$ длины $(r-h) < r$, то, по индукционному предположению, префикс внутреннего графа, соответствующий пути P, является корректным для GSS-вершины v_1, что и доказывает утверждение леммы.

Второй случай: $\forall i \in [1, .., (j-1)], e_i$ — обнуляемое ребро. Покажем, что префикс внутреннего графа, соответствующий пути P, также является корректным и для GSS-вершины v_j.

В данном случае e_j — единственное не обнуляемое ребро в последовательности E. Рассмотрим ту конечную последовательность раскрытий, примененную к последовательности ребер E, после которой остаются только терминальные ребра, причем строка, образованная инвертированием этой последовательности терминальных ребер и заменой их на терминалы, с которыми они ассоциированы, является строкой $p_2 = (t_l, .., t_{l+r-1})$. Существует два варианта: либо после первого применения операции раскрытия в последовательности будет более одного не обнуляемого ребра, либо при применении первых $i > 0$ операций раскрытия последовательность ребер GSS будет состоять из единственного нетерминального ребра, причем $\forall d \in [1, .., i]$, после применения первых d операций раскрытия последовательность состоит из единственного ребра $S_d = (h_d, g_d)$. Первый вариант означает, что нетерминальное ребро e_j ассоциировано с путем $F = (x_1, .., x_n)$, где GSS-вершина x_1, как и вершина v_j, принадлежит вершине внутреннего графа V_{l+r}, а GSS-вершина $x_n = v_{j+1}$. Причем в пути F имеется более одного не обнуляемого ребра. Найдем минимальное y такое, что ребро $e_{first} = (x_y, x_{y+1})$ является не обнуляемым. Так как в пути F имеется более одного не обнуляемого ребра, то последовательность ребер $F_1 = ((x_1, x_2), .., (x_y, x_{y+1}))$, как

и последовательность ребер $F_2 = ((x_{y+1}, x_{y+2}), .., (x_{n-1}, x_n))$, после применения конечного числа раскрытий сводится к строке длины, меньшей r. Пусть последовательности ребер F_1 и F_2 сводятся за рассматриваемое конечное число раскрытий к строкам $f_1 = (t_{l+c}, .., t_{l+r-1})$ и $f_2 = (t_l, .., t_{l+c-1})$, где $0 < c < r$. Так как $x_n = v_{j+1}$, то по индукционному предположению префикс внутреннего графа, соответствующий пути $(V_1, .., V_{l+c})$, является корректным для GSS-вершины x_{y+1}. Это, в свою очередь, означает, что префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_{l+r})$, является корректным для GSS-вершины x_1. А так как в GSS, построенным основным анализом, существует ребро $e_j = (v_j, v_{j+1})$, ассоциированное с GSS путем F, то префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_{l+r})$, является также корректным и для GSS-вершины v_j. Рассмотрим второй вариант. Аналогично рассуждениям в базе индукции при $r = 1$, конструируем множества $\{S_0, .., S_i\}, \{w_0, .., w_i\}, \{h_0, .., h_i\}, \{g_0, .., g_i\}$. При применении $(i + 1)$-ой операции раскрытия нетерминальное ребро S_i заменятся на последовательность ребер, соответствующую пути R в GSS из вершины r_1 в вершину g_i, причем в пути R существует более одного не обнуляемого ребра. GSS-вершина r_1, как и вершина v_j, принадлежит вершине внутреннего графа V_{l+r}. Применив к пути R, рассуждения, аналогичные рассуждениям при рассмотрении пути F, получим, что префикс внутреннего графа, соответствующий пути P, является корректным для любой GSS-вершины из множества $\{w_0, .., w_i, r_1\}$, а так как $w_0 = v_j$, то, в частности, префикс внутреннего графа, соответствующий пути P, является корректным для GSS-вершины v_j.

Таким образом показали, что префикс внутреннего графа, соответствующий пути P, является корректным для GSS-вершины v_j. А так как существует путь из вершины v_1 в вершину v_j, состоящий из обнуляемых ребер, то по индукционному предположению префикс внутреннего графа,

соответствующий пути P, также является корректным и для GSS-вершины v_1, что и доказывает утверждение леммы. □

Теорема 2. *Каждый префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_l)$ внутреннего графа и порождаемый графом префиксов $v.prefixes$, является корректным для GSS-вершины v.*

Доказательство. Пусть p — строка, образованная из терминалов на ребрах пути P. Длина строки p равна $(l - 1)$. Так как префикс внутреннего графа, соответствующий пути P, порождается графом префиксов $v.prefixes$, то в этом графе префиксов существует путь X из одной из начальных вершин графа префиксов $v.prefixes$ в вершину EOP, который сводится к строке p.

Если путь X состоит из единственной вершины EOP, то она является начальной вершиной графа префиксов $v.prefixes$, а значит v — начальная вершина GSS. А так как путь X в графе префиксов $v.prefixes$ сводится к пустой строке, то строка p — пустая, и путь P состоит из единственной начальной вершины внутреннего графа V_1. Значит, префикс внутреннего графа, соответствующий пути P, корректен для начальной вершины GSS, которой является вершина v.

Пусть путь $X = (E_1, .., E_m, EOP)$, где $m > 0$ и E_1 — одна из начальных вершин графа префиксов $v.prefixes$. Из построения графов префиксов следует, что существует путь из GSS-вершины v в начальную GSS-вершину v_0. Причем последовательность ребер в данном пути после конечного числа раскрытий сводится к строке p. А так как префикс внутреннего графа, соответствующий пути $F = (V_1)$, корректен для начальной GSS-вершины v_0 и существует путь из GSS-вершины v в GSS-вершину v_0, в котором последовательность ребер сводится за конечное число раскрытий к строке p, то, по лемме, префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_l)$, является корректным для GSS-вершины v. □

5.2 Корректность алгоритма диагностики ошибок

В данном разделе приведено доказательство того, что любое ребро e внутреннего графа из множества $errors$ является ошибочными, а все ошибочные ребра принадлежат множеству $errors \cup probErrors$. Также будет доказано, что любой префикс внутреннего графа, порождаемый графом префиксов из множества $errors[e]$, является корректным, но при добавлении к данному префиксу в конец ребра e образуется некорректный префикс внутреннего графа, а все такие префиксы порождаются хотя бы одним графом префиксов из множества $errors[e] \cup probErrors[e]$.

Теорема 3. *После работы алгоритма диагностики ошибок любое ребро e внутреннего графа из множества $errors$ является ошибочными. А любой префикс внутреннего графа, порождаемый графом префиксов из множества $errors[e]$, является корректным, но при добавлении к данному префиксу в конец ребра e образуется некорректный префикс внутреннего графа.*

Доказательство. Докажем, что ребро e является ошибочным, методом от противного. Пусть ребро внутреннего графа $e \in errors$ не является ошибочным. Пусть ребро e нагружено терминалом t, исходит из вершины внутреннего графа V_l и входит в вершину V_{l+1}. Обозначим входную эталонную грамматику через G. Рассмотрим два случая.

Если V_{l+1} — не конечная вершина внутреннего графа. Пусть $(e_1, .., e_n)$ — терминальные ребра GSS, ассоциированные с терминалом t, причем $\forall i \in [1, .., n], e_i = (x_i, y_i)$, где x_i принадлежит вершине V_{l+1}, а y_i принадлежит вершине V_l. Пусть GSS-вершина v, принадлежащая вершине внутреннего графа V_l, принадлежит множеству Z, если $\forall i \in [1, .., n], v \neq y_i$. Так как $e \in errors$, то $\exists v \in Z$ — GSS-вершина, такая, что $v.prefixes$ порождает некоторый префикс внутреннего графа, который не порождается графом префиксов $y_i.prefixes, \forall i \in [1, .., n]$. Множество всех таких префиксов внутреннего графа обозначим K. Пусть один из префиксов внутреннего графа

множества K соответствует пути $P = (V_1, .., V_l)$. Пусть p — строка, образованная последовательностью терминалов на пути P. Так как $v.prefixes$ порождает префикс внутреннего графа, соответствующий пути P, то, по ТЕОРЕМЕ 2, получаем, что этот префикс является корректным для GSS-вершины v. Значит, строка p является корректным префиксом для грамматики G. Пусть Q — множество GSS-вершин, для которых префикс внутреннего графа, соответствующий пути P, является корректным. Так как рассматриваемый префикс внутреннего графа не порождается графом префиксов $y_i.prefixes, \forall i \in [1, .., n]$, то, по ТЕОРЕМЕ 1, данный префикс не является корректным для GSS-вершины $y_i, \forall i \in [1, .., n]$. Значит, $\forall i \in [1, .., n], y_i \notin Q$. Поэтому $\forall q \in Q$, в управляющих таблицах не существует операции сдвига из состояния $q.state$ по терминалу t. Значит, RNGLR-алгоритм, обработав все терминалы строки p и получив следующим терминалом входной строки терминал t, не сможет сделать ни одной операции сдвига по данному терминалу. Таким образом, префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_l, V_{l+1})$, не является корректным. Значит, ребро e является ошибочным. Получили противоречие.

Если V_{l+1} — конечная вершина внутреннего графа, тогда t является специальным терминалом EOF конца строки. Пусть A_1 — множество GSS-вершин, принадлежащих вершине V_l и имеющих принимающее состояние, а A_2 — множество GSS-вершин, принадлежащих вершине V_l и имеющих непринимающее состояние. Так как $e \in errors$, то существует GSS-вершина $v \in A_2$ такая, что граф префиксов $v.prefixes$ порождает некоторый корректный префикс внутреннего графа, который не порождается ни одним графом префиксов GSS-вершин из множества A_1. Множество всех таких префиксов внутреннего графа обозначим L. Пусть один из префиксов внутреннего графа множества L соответствует пути $F = (V_1, .., V_l)$. Пусть f — строка, образованная последовательностью терминалов

на пути F. По теореме 2, данный префикс внутреннего графа является корректным для GSS-вершины v, так как он порождается графом префиксов $v.prefixes$. Значит, префикс внутреннего графа, соответствующий пути F, является корректным. А так как графы префиксов GSS-вершин из множества A_1 не порождают префикс внутреннего графа, соответствующий пути F, то, по теореме 1, рассматриваемый префикс внутреннего графа не является корректным ни для одной GSS-вершины из множества A_1. Значит, RNGLR-алгоритм, обработав все терминалы строки f, на последнем уровне GSS не будет иметь ни одной вершины с принимающим состоянием. Значит, строка f не является принимаемой грамматикой G. Таким образом, ребро e является ошибочным. Получили противоречие.

В случае, когда V_{l+1} не является конечной вершиной внутреннего графа, префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_l)$, взят из множества K произвольным образом. Для этого префикса показано, что он является корректным, но при добавлении к данному префиксу в конец ребра e образуется некорректный префикс внутреннего графа. Из построения множества графов префиксов $errors[e]$ следует, что данные графы префиксов порождают префиксы внутреннего графа из множества K и только их. Таким образом, любой префикс внутреннего графа, порождаемый одним из графов префиксов $errors[e]$, является корректным, но при добавлении к нему в конец ребра e образуется некорректный префикс внутреннего графа. Аналогично, данное утверждение доказывается в случае, если V_{l+1} — конечная вершина внутреннего графа с множеством префиксов внутреннего графа равным L. □

Теорема 4. *После работы алгоритма диагностики ошибок любое ошибочное ребро e внутреннего графа, исходящее из вершины V_l и входящее в вершину V_{l+1}, принадлежит объединению множеств errors и probErrors. А любой префикс внутреннего графа, заканчивающийся в вершине V_l и стано-*

вящийся некорректным при добавлении в конец ребра e, порождается хотя бы одним графом префиксов из объединения множеств $errors[e]$ и $probErrors[e]$.

Доказательство. Пусть ребро e нагружено терминалом t. Пусть L — множество всех префиксов внутреннего графа, заканчивающихся в вершине V_l и становящихся некорректным при добавлении в конец ребра e. Так как ребро e является ошибочным, то множество L не пусто. Рассмотрим префикс внутреннего графа из множества L. Пусть этому префиксу соответствует путь внутреннего графа $P = (V_1, .., V_l)$. Пусть p — строка, получающаяся заменой в последовательности ребер в пути P на терминалы, которыми нагружены соответствующие ребра. Так как данный префикс внутреннего графа является корректным, то существует GSS-вершина v, принадлежащая вершине внутреннего графа V_l, такая, что рассматриваемый префикс является корректным для вершины v. По теореме 1, данный префикс внутреннего графа порождается графом префиксов $v.prefixes$. Обозначим через G входную эталонную грамматику. Рассмотрим два случая.

Если V_{l+1} — не конечная вершина внутреннего графа. Пусть $(e_1, .., e_n)$ — терминальные ребра GSS, ассоциированные с терминалом t, причем $\forall i \in [1, .., n], e_i = (x_i, y_i)$, где x_i принадлежит вершине V_{l+1}, а y_i принадлежит вершине V_l. Пусть GSS-вершина u, принадлежащая вершине внутреннего графа V_l, принадлежит множеству Z, если $\forall i \in [1, .., n], u \neq y_i$. Так как один из префиксов, порождаемых графом префиксов $v.prefixes$, становится некорректным при добавлении в конец ребра e, то $v \in Z$. Если граф префиксов $v.prefixes$ имеет цикл, то ребро $e \in probErrors$, а префикс внутреннего графа, соответствующий пути P, порождается одним из графов префиксов множества $probErrors[e]$. Пусть граф префиксов $v.prefixes$ не имеет циклов. В виду того, что рассматриваемый префикс внутреннего графа становится некорректным при добавлении в конец ребра e, то $\forall i \in [1, .., n], y_i.prefixes$

не порождает данный префикс. Если $\exists i \in [1, .., n]$ такое, что граф префиксов $y_i.prefixes$ имеет цикл, то $e \in probErrors$, а префикс внутреннего графа, соответствующий пути P, порождается одним из графов префиксов множества $probErrors[e]$. Если такого i не существует, то $e \in errors$, а префикс внутреннего графа, соответствующий пути P, порождается одним из графов префиксов множества $errors[e]$. Таким образом, в данном случае, $e \in errors \cup probErrors$, а префикс внутреннего графа, соответствующий пути P, порождается одним из графов префиксов множества $errors[e] \cup probErrors[e]$.

Если V_{l+1} — конечная вершина внутреннего графа, тогда t является специальным терминалом EOF конца строки. Пусть A_1 — множество GSS-вершин, принадлежащих вершине V_l, и имеющих принимающее состояние, а A_2 — множество GSS-вершин, принадлежащих вершине V_l, и имеющих непринимающее состояние. Так как один из префиксов, порождаемых графом префиксов $v.prefixes$, становится некорректным при добавлении в конец ребра e, то $v \in A_2$. Если граф префиксов $v.prefixes$ имеет цикл, то ребро $e \in probErrors$, а префикс внутреннего графа, соответствующий пути P, порождается одним из графов префиксов множества $probErrors[e]$. Пусть граф префиксов $v.prefixes$ не имеет циклов. Ввиду того, что рассматриваемый префикс внутреннего графа становится некорректным при добавлении в конец ребра e, то $\forall z \in A_1, z.prefixes$ не порождает данный префикс. Если $\exists z \in A_1$ такое, что граф префиксов $z.prefixes$ имеет цикл, то $e \in probErrors$, а префикс внутреннего графа, соответствующий пути P, порождается одним из графов префиксов множества $probErrors[e]$. Если такого z не существует, то $e \in errors$, а префикс внутреннего графа, соответствующий пути P, порождается одним из графов префиксов множества $errors[e]$.

Таким образом, во всех случаях $e \in errors \cup probErrors$, а префикс внутреннего графа, соответствующий

пути P, порождается одним из графов префиксов множества $errors[e] \cup probErrors[e]$. Так как префикс внутреннего графа, соответствующий пути P, выбирался из множества L произвольным образом, то утверждение теоремы доказано. □

6 Экспериментальное исследование

Предложенный механизм диагностики ошибок был реализован на платформе .NET как часть проекта YaccConstructor; основным языком разработки являлся F# [10]. Данная реализация является модификацией ранее реализованного в рамках проекта алгоритма ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения.

Модифицированный алгоритм был протестирован на серии тестов с целью проверки работоспособности. Данные тесты проверяли, что алгоритм строит корректные множества $errors$ и $probErrors$ ребер входного графа. Для каждого теста специфицировалась грамматика на языке YARD и в явном виде задавался граф конечного автомата, ребра которого были промаркированы лексемами входной грамматики. Входные графы для данных тестов содержали как ветвления, так и циклы. На всех тестах модифицированный алгоритм корректно строил множества $errors$ и $probErrors$. Кроме того, на всех тестах, входные графы которых не содержали циклов, модифицированный алгоритм точно определял все ошибочные ребра входного графа, то есть $probErrors$, в данном случае, являлось пустым множеством.

Также на нескольких сериях синтетических тестов была протестирована производительность модифицированного алгоритма. Анализ промышленного проекта по миграции базы данных с MS-SQL Server 2005 на Oracle 11gR2 показал, что запросы часто формируются конкатенацией фрагментов, каждый из которых формируется с помощью ветвлений или циклов. Ниже приведена входная грамматика, использованная

в данных тестах:

$$start\_rule ::= s$$
$$s ::= s \text{ PLUS n}$$
$$n ::= \text{ONE} \mid \text{TWO} \mid \text{THREE} \mid \text{FIVE} \mid \text{SIX} \mid \text{SEVEN}$$

Входные графы представляли собой конкатенацию базовых блоков без циклов. Каждая серия тестов характеризовалась тремя параметрами:

- *height* — количество ветвлений в базовом блоке;
- *length* — максимальное количество повторений базовых блоков;
- *errorBranches* — количество веток в базовом блоке, содержащих ошибочное ребро (на рис. 2 изображен базовый блок без ошибочных ребер, а на рис. 3 — базовый блок с двумя выделенными ошибочными ребрами).

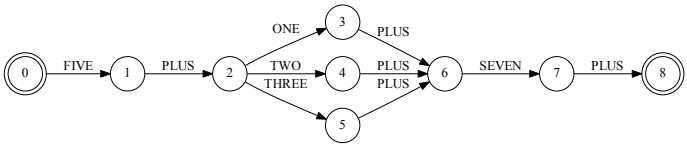


Рис. 2: Базовый блок при $height = 3, errorBranches = 0$

Замеры времени работы алгоритмов проводились на машине со следующими техническими характеристиками: Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz, RAM: 8.0 GB, процессор x64.

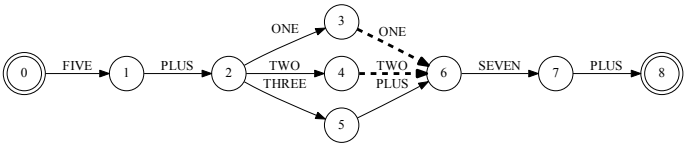


Рис. 3: Базовый блок при $height = 3, errorBranches = 2$

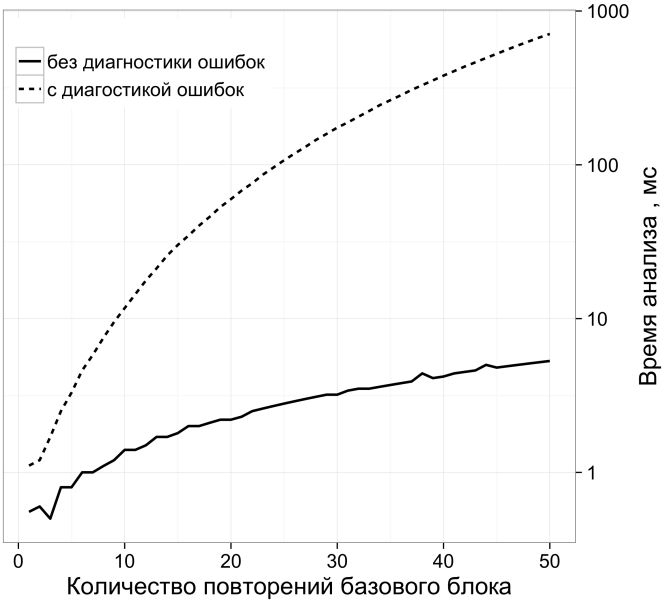


Рис. 4: Сравнение времени работы алгоритма синтаксического анализа до и после модификаций, при $height = 4, errors = 0$

Каждая серия объединяет набор из 50 тестов, каждый из которых содержит одинаковое количество ветвлений в базовом блоке, при этом количество повторений блока совпадает с порядковым номером теста ($length = i$, для теста с номером i). Для каждого теста измерялось время, затраченное на синтаксический анализ. Измерения проводились 10 раз, после чего усреднялись. График, представленный на рис. 4, иллюстрирует сравнение времени работы алгоритма синтаксического анализа до и после модификаций. Можно заметить, что выполненная модификация существенно увеличивает продолжительность анализа. Причина этого в том, что выполненная модификация является прототипом, а в будущем планируется улучшить производительность путем улучшения реализации. График на рис. 5 демонстрирует зависимость времени работы модифицированного алгоритма от количества повторений базового блока и количества веток, содержащих ошибочное ребро, в каждом из них. Наблюдается уменьшение времени работы модифицированного алгоритма при увеличении количества ошибочных ребер. Одной из причин этого является уменьшение количества корректных префиксов внутреннего графа при увеличении количества ошибочных ребер.

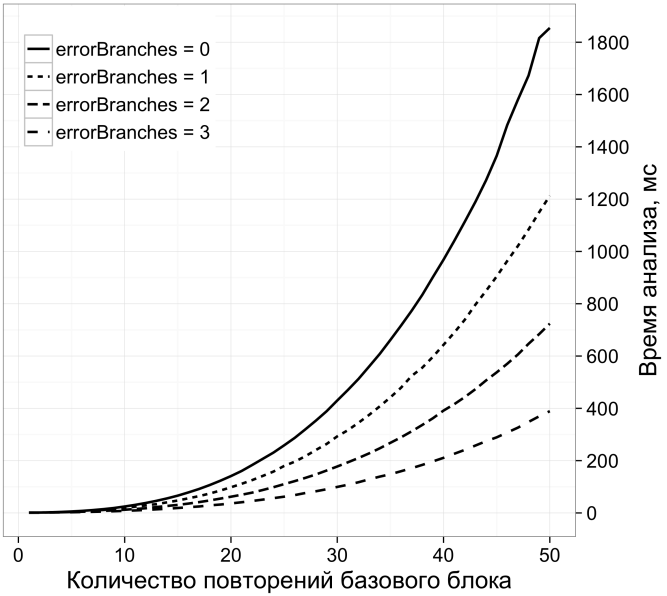


Рис. 5: Зависимость времени работы модифицированного алгоритма от размера входного графа и количества ошибочных ребер при $height = 6$

7 Заключение

В ходе данной работы получены следующие результаты.

- Определено понятие синтаксической ошибки в терминах регулярной аппроксимации динамически формируемого выражения.
- Разработан механизм диагностики ошибок в синтаксическом анализе регулярной аппроксимации динамически формируемого выражения.
- Доказана корректность предложенного механизма.

– Предложенный механизм реализован на языке программирования F# в рамках проекта YaccConstructor.
– Проведено экспериментальное исследование: тестирование работоспособности и тестирование производительности.
– Исходный код проекта YaccConstructor можно найти на сайте https://github.com/YaccConstructor/YaccConstructor, автор принимал участие под учётной записью rustam-azimov.

В дальнейшем планируется использовать результат работы модифицированного алгоритма синтаксического анализа для формирования сообщений об обнаруженных ошибках, удобных для пользователя. Кроме того, необходимо произвести теоретическую оценку сложности модифицированного алгоритма.

Список литературы

1. Christensen A. S., Møller A., Schwartzbach M. I. Precise Analysis of String Expressions. Proc. 10th International Static Analysis Symposium (SAS). — Vol. 2694 of LNCS. — Springer-Verlag, 2003. — P. 1–18.
2. Java String Analyzer. http://www.brics.dk/JSA/ . urldate = "18.05.2016"
3. Minamide Y. Static Approximation of Dynamically Generated Web Pages. Proceedings of the 14th International Conference on World Wide Web. — WWW'05. — New York, NY, USA : ACM, 2005. — P. 432–441.
4. A. Annamaa, A. Breslav, J. Kabanov, V. Vene. An Interactive Tool for Analyzing Embedded SQL Queries. Proceedings of the 8th Asian Conference on Programming Languages and Systems. — APLAS'10. — Berlin, Heidelberg : Springer-Verlag, 2010. — P. 131–138.
5. Annamaa A., Breslav A., Vene V. Using Abstract Lexical Analysis and Parsing to Detect Errors in String-Embedded DSL Statements. Proceedings of the 22nd Nordic Workshop on Programming Theory. — 2010. — P. 20–22.

6. Alvor. https://bitbucket.org/plas/alvor urldate = "18.05.2016"
7. IntelliLang. https://www.jetbrains.com/idea/help/intellilang.html urldate = "18.05.2016"
8. PHPStorm IDE. https://www.jetbrains.com/phpstorm/ . urldate = "18.05.2016"
9. Nguyen H. V., Kästner C. and Nguyen T. N. Varis: IDE Support for Embedded Client Code in PHP Web Applications. Proceedings of the 37th International Conference on Software Engineering (ICSE). — New York, NY : ACM Press, 2015. — Formal Demonstration paper.
10. Syme, Don and Granicz, Adam and Cisternino, Antonio. Expert F# (Expert's Voice in .Net). ISBN: 1590598504, 9781590598504.
11. Kirilenko I., Grigorev S., Avdiukhin D. Syntax Analyzers Development in Automated Reengineering of Informational System. St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications and Control Systems. — 2013. — Vol. 174, no. 3. — P. 94–98.
12. Verbitskaia E., Grigorev S., Avdyukhin D. Relaxed Parsing of Regular Approximations of String-Embedded Languages. Preliminary Proceedings of the PSI 2015: 10th International Andrei Ershov Memorial Conference. — PSI'15. — 2015. — P. 1–12.
13. Khabibullin M., Ivanov A., Grigorev S. On Development of Static Analysis Tools for String-Embedded Languages. In Proceedings of the 11th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '15). ACM, New York, NY, USA, Article 5, 10 pages
14. Asveld P. R. J., Nijholt A. The Inclusion Problem for Some Subclasses of Contextfree Languages. Vol. 230. — Essex, UK : Elsevier Science Publishers Ltd., 1999. — December. — P. 247–256.
15. Tomita M. An Efficient Context-free Parsing Algorithm for Natural Languages. Proceedings of the 9th International Joint Conference on Artificial Intelligence – Volume 2. — IJCAI'85. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1985. — P. 756–764.
16. Rekers J. Parser Generation for Interactive Environments. 1992.
17. Scott E., Johnstone A. Right Nulled GLR Parsers. ACM Trans. Program. Lang. Syst. — 2006. — Vol. 28, no. 4. — P. 577–618.
18. Григорьев С. В. Синтаксический анализ динамически формируемых программ : Дисс... кандидата наук. Санкт-Петербургский государственный университет. — 2015

19. Вербицкая Е. А. Синтаксический анализ регулярных множеств. Санкт-Петербургский государственный университет. — 2015.
20. Dick G., Ceriel H. Parsing Techniques: A Practical Guide. Upper Saddle River, NJ, USA : Ellis Horwood, 1990. — ISBN: 0-13-651431-6.
21. F. Yu, M. Alkhalaf, T. Bultan, O. H. Ibarra. Automata-based Symbolic String Analysis for Vulnerability Detection. Form. Methods Syst. Des. — 2014. — Vol. 44, no. 1. — P. 44–70.

Поддержка конъюнктивных грамматик в алгоритме GLL

Горохов Артем Владимирович

Санкт-Петербургский государственный университет
gorohov.art@gmail.com

Аннотация Большинство синтаксических анализаторов созданы для распознавания контекстно-свободных языков, но для решения некоторых задач их выразительности недостаточно. Так, для решения такой задачи биоинформатики, как поиск цепочек ДНК в метагеномных сборках необходимо использовать конъюнктивные грамматики для описания искомых структур. В работе представлена модификация реализации алгоритма синтаксического анализа GLL, расширяющая класс распознаваемых анализатором языков до конъюнктивных.

Введение

Синтаксический анализ, как правило, используется для построения структурного представления кода с использованием грамматики, описывающей разбираемый язык. Абстрактное синтаксическое дерево, являющееся результатом работы синтаксического анализатора, в дальнейшем используется для проведения статического анализа кода или же в каких-то других целях. Как правило, на вход синтаксическому анализатору подаётся линейная последовательность токенов, представляющая код программы. Однако могут возникать ситуации, когда вход не может быть представлен линейно. Такие ситуации могут

возникать, например, при автоматической генерации кода. Генерация может происходить в циклах, с использованием условных операторов или строковых операций. Поэтому для описания генерируемых цепочек можно использовать конечный автомат, порождающий цепочки, который уже не будет являться линейным. Такую задачу будем называть синтаксическим анализом регулярных множеств.

Кроме этого, ещё одной областью, где может быть применим синтаксический анализ регулярных множеств, является бионформатика. Одной из часто возникающих задач в биоинформатике является классификация организмов, находящихся в образцах, полученных из окружающей среды [11]. По образцам строится метагеномная сборка, которая содержит в себе смесь из ДНК всех содержащихся в сборке организмов. В свою очередь ДНК является последовательностью символов в алфавите {A, C, G, T}. ДНК организмов, которые относятся к одному и тому же виду, содержат одинаковые подцепочки, которые и необходимо выделить, чтобы классифицировать организм. Как правило, эти подцепочки — это последовательности РНК. РНК может быть описана с помощью грамматики. Метагеномная сборка, в свою очередь, может быть представлена в виде графа с цепочками на рёбрах. Таким образом, в таком графе необходимо найти цепочки, выводимые в грамматике, описывающей РНК.

Грамматики, описывающие структуру РНК, являются неоднозначными. Грамматика называется неоднозначной, если одна и та же цепочка может быть выведена несколькими способами. Такие алгоритмы синтаксического анализа как LR и LL не позволяют обрабатывать неоднозначные грамматики. Для работы с неоднозначными грамматиками существуют алгоритмы обобщённого синтаксического анализа GLR [12], GLL [4]. В рамках проекта YaccConstructor [5, 6] был реализован алгоритм GLL, кроме того, была предложена его модификация для обработки нелинейных входных

данных — графов. Реализованная модификация позволяет находить цепочки транспортной РНК (тРНК) в небольших метагеномных сборках, возвращая координаты начала и конца найденной цепочки. Проблема заключается в том, что грамматика для описания тРНК является сильно неоднозначной, что сказывается на производительности и точности полученных результатов. Для повышения точности можно применять конъюнктивные грамматики [3], в которых для описания продукций используется операция конъюнкции. Такие грамматики расширяют класс контекстно-свободных языков и позволяют точнее описать структуру тРНК. Данная работа посвящена описанию модификаций решения на основе алгоритма GLL для работы с конъюнктивными грамматиками.

1 Постановка задачи

Целью данной работы является добавление поддержки конъюнктивных грамматик в YaccConstructor. Для её достижения были поставлены следующие задачи:

- реализовать поддержку конъюнктивных грамматик в языке спецификации грамматик YARD;
- реализовать поддержку конъюнктивных грамматик в генераторе GLL-анализаторов;
- провести экспериментальные исследования работы алгоритма.

2 Обзор предметной области

Одной из задач, часто возникающих в биоинформатике, является классификация организмов в образцах, полученных из окружающей среды. Из образцов извлекается смесь ДНК всех организмов, которая представляется в виде метагеномноnй сборки. Метагеномная сборка, в свою очередь,

может быть представлена в виде конечного автомата, порождающего геномы всех организмов из образца. О видовой принадлежности организма можно судить по его РНК. Для поиска РНК в метагеномных сборках существуют различные подходы. Некоторые из них используют скрытые модели Маркова [13] для поиска, например, инструмент REAGO [8]. Недостатком инструмента является то, что он не обрабатывает метагеномные сборки, представленные в виде графа, а представление сборки в другом виде требует слишком больших объёмов памяти. Инструмент Xander [9] позволяет работать со сборками, представленными в виде графов, но в основе лежит механизм, обладающий низкой точностью. Синтаксический анализ также применяется для анализа метагеномных сборок, например, в инструменте Infernal [10], который не предназначен для работы с графами.

Грамматика, описывающая РНК, является сильно неоднозначной и её не всегда можно привести к однозначной форме. Для работы с неоднозначными грамматиками используются алгоритмы обобщённого синтаксического анализа. Принцип работы таких алгоритмов заключается в том, что они просматривают все возможные пути вывода входной цепочки и строят все деревья вывода этой цепочки. Существует инструмент SBP [2], основанный на алгоритме GLR, позволяющий работать с конъюнтивными грамматиками. В данной работе используется алгоритм GLL, так как в среднем он работает быстрее. Алгоритм обобщённого анализа GLL основан на нисходящем анализе и отличается высокой скоростью работы и простотой. В алгоритме для хранения всех деревьев вывода используется структура данных SPPF (Shared Packed Parse Forest) [1]. Эта структура данных позволяет переиспользовать узлы с одинаковыми поддеревьями под ними. На рис. 1 показано, как объединяются разные выводы нетерминала S. Создаются дополнительные узлы, соответствующие каждому из выводов. Выделяются одинаковые поддеревья и остаётся только один

экземпляр каждого, на который в дальнейшем ссылаются предки.

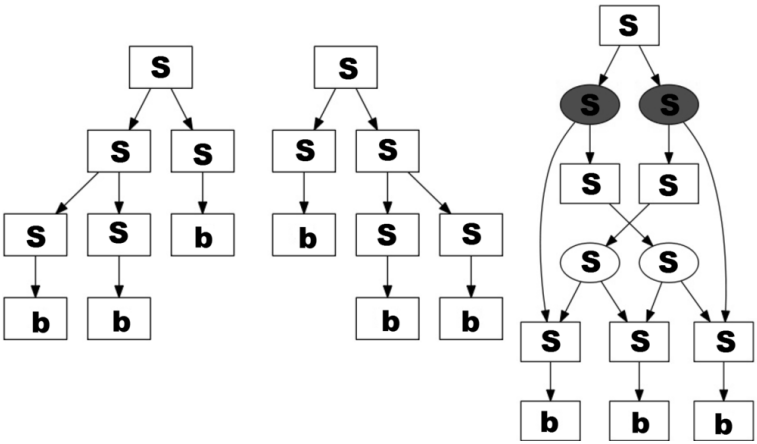


Рис. 1: Преобразование деревьев разбора к SPPF

Алгоритм GLL позволяет работать с любыми КС-грамматиками, в том числе и сильно неоднозначными. Однако наличие сильной неоднозначности в грамматике сказывается на точности получаемых результатов и производительности. Конъюнктивные грамматики при описании правил вывода используют операцию конъюнкции. Цепочка принадлежит языку, задаваемому такой грамматикой, если существует вывод по обоим конъюнктам. На рис. 2 изображена грамматика для языка, не являющегося контекстно-свободным, описанная с помощью операции конъюнкции. Такие грамматики дают возможность точно описать структуру тРНК, что позволяет снизить количество ошибок при разборе.

```
S ::= A B & D C
A ::= a A |ϵ
B ::= b B c |ϵ
C ::= c C |ϵ
D ::= a D b |ϵ
```

Рис. 2: Грамматика для контекстно-зависимого языка $\{a^n b^n c^n, n \geq 0\}$

3 Основная часть

В данном разделе рассмотреннно расширение языка YARD и модификация алгоритма синтаксического анализа GLL.

3.1 Архитектура

Ввиду модульной структуры проекта YaccConstructor, задействованную структуру проекта можно разделить на части, показанные на рис. 3.

В проекте используется язык описания грамматик YARD [7], который поддерживает различные конструкции, упрощающие разработку грамматик: повторения x*[1..10], дизъюнкции A|B, условное вхождние и подобные. Перед подачей генератору, дерево разбора грамматики проходит через множество преобразований, в результате которых грамматика приводится к форме Бэкуса-Наура [14].

3.2 Расширение YARD

Для работы с конъюнктивными грамматиками в язык YARD была добавлена новая конструкция: конъюнкция, приоритет которой между дизъюнкцией и последовательностью. Язык YARD поддерживает различные конструкции вроде повторений (x*[1..10]), которые преобразуются перед подачей генератору. Так как в язык

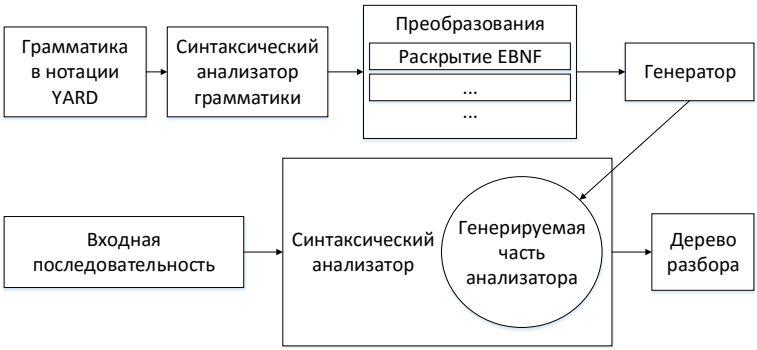


Рис. 3: Структура существующего решения

была добавлена новая конструкция, нужно было обеспечить её поддержку во всех существующих преобразованиях.

Кроме того, добавленную конструкцию также необходимо преобразовывать к виду, принимаемому генератором. Предлагается следующее решение: грамматика преобразовывается к контекстно-свободной, с дополнительной информацией о существовании конъюнкции. Для каждой конъюнкции в грамматике создаётся три новых правила: по одному на каждый конъюнкт и одно на дизъюнкцию конъюнктов, в исходном правиле конъюнкция заменяется ссылкой на правило с дизъюнкцией конъюнктов. На рис. 4 и 5 показан пример начальной грамматики (G_0) и результат её преобразования (G_1). В правиле S содержится конъюнкция, которая заменяется на ссылку на сгенерированное правило Conj0. Для каждого из конъюнктов также генерируется правило и продукция правила Conj0 представляет собой выбор между ними.

При этом имена правил генерируются так, что в дальнейшем правила, заменившие конъюнкцию, можно определять по имени.

```
S ::= A D & B
A ::= A a |ϵ
D ::= A d |ϵ
B ::= A b D |ϵ
```

Рис. 4: Грамматика G_0

```
     S ::= Conj0
 Conj0 ::= Conj0_0 | Conj0_1
Conj0_0 ::= A D
Conj0_1 ::= B
     A ::= A a |ϵ
     D ::= A d |ϵ
     B ::= A b D |ϵ
```

Рис. 5: Грамматика G_1

3.3 Модификация алгоритма GLL

После построения SPPF синтаксическим анализатором нужно убедиться в том, что у правил с именем, соответствующим конъюнкции, есть вывод по обоим продукциям, иначе ветвь вывода нужно исключить из результатов разбора. Заметим, что невозможно проверять это в процессе разбора, т.к. нет возможности отследить, когда построятся все возможные выводы нетерминала.

Данная задача решается с помощью рекурсивного обхода дерева в глубину. Для каждого узла проверяется корректность вывода его потомков. Затем, если узел является нетерминальным, проверяется имя нетерминала, которому он соответствует, если оно является сгенерированным именем правила конъюнкции, то, при наличии двух выводов нетерминала, поддерево, начиная с текущего нетерминала, считается корректным.

Таким образом время работы алгоритма возрастает на время, необходимое для обхода дерева разбора. Так как размер дерева не превышает кубического от длины входных данных, то сложность алгоритма (в худшем случае) остаётся прежней $O(n^3)$.

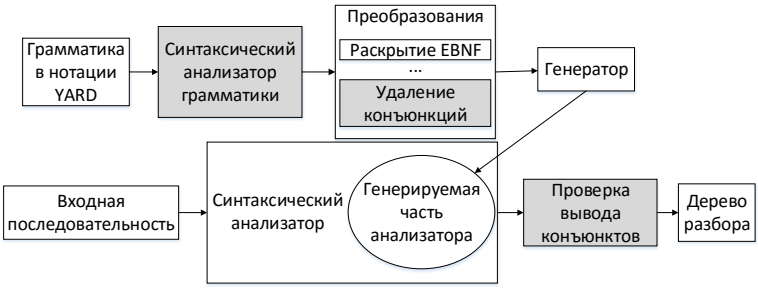


Рис. 6: Структура решения с учётом внесённых изменений

4 Эксперименты

Были проведены экспериментальные исследования, целью которых являлась проверка того, что конъюнктивные грамматики позволяют задавать структуру тРНК так, что синтаксический анализатор находит меньше некорректных цепочек.

На рис. 7 и 9 представлены грамматки, описывающие структуру тРНК. Грамматика G_2 является контекстно-свободной, а грамматика G_3 — конъюнктивной. По данным грамматикам были сгенерированы соответствующие синтаксические анализаторы.

На вход построенным синтаксическим анализаторам подавались сгенерированные цепочки ДНК длиной

```
[<Start>]
folded: stem<(any*[1..3]
            stem<any*[7..10]>
            any*[1..3]
            stem<any*[5..8]>
            any*[3..5]
            stem<any*[5..8]>
            )>

stem<s>:
     A stem<s> U
   | U stem<s> A
   | C stem<s> G
   | G stem<s> C
   | G stem<s> U
   | U stem<s> G
   | s

any: A | U | G | C
```

Рис. 7: КС-грамматика вторичной структуры тРНК

от 100 до 1000 симовлов. Эти цепочки содержали в себе последовательности тРНК, а также другие последовательности, которые можно ложно признать за тРНК. Напимер, цепочка на рис. 8, хоть и не является тРНК, распознаётся грамматикой G_2, но не распознаётся грамматикой G_3.

Результаты экспериментов приведены в таблице 1. Из них ясно, что грамматика G_2 не распознаёт ложные цепочки, распознаваемые грамматикой G_3. Время работы синтаксических анализаторов показано на графике, изображённом на рис. 10. По графику видно, что время работы синтаксического анализатора, построенного по

ACACCCCCCCUCACCCCCUCCCACCCCCUU

Рис. 8: Пример цепочки нуклеотидов, сгенеированной для экспериментов

```
[<Start>]
folded: stem<subseq> & (any*[7..9] subseq any*[7..9])

subseq: any*[1..3]
        stem<any*[7..10]> & (any*[4..6] any*[7..10] any*[4..6])
        any*[1..3]
        stem<any*[5..8]> & (any*[6] any*[5..8] any*[6])
        any*[3..5]
        stem<any*[5..8]> & (any*[4..5] any*[5..8] any*[4..5])

stem<s>:
      A stem<s> U
    | U stem<s> A
    | C stem<s> G
    | G stem<s> C
    | G stem<s> U
    | U stem<s> G
    | s

any: A | U | G | C
```

Рис. 9: Конъюнктивная грамматика структуры тРНК

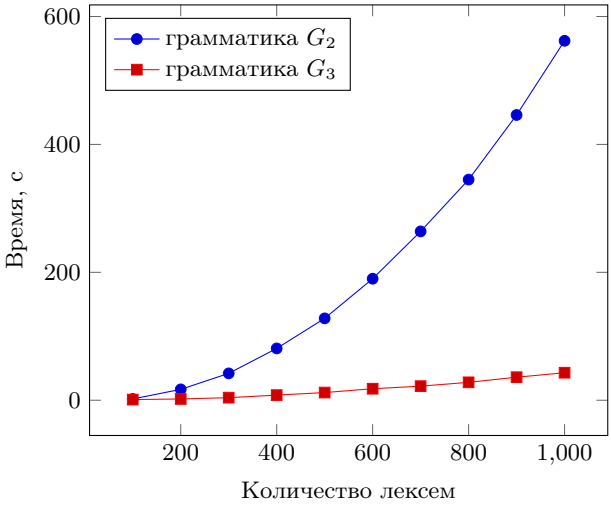


Рис. 10: Среднее время работы алгоритма на конъюнктивной и контекстно-свободной грамматиках тРНК

	КС-грамматика	Конъюнктивная грамматика
Тест 1.	15	0
Тест 2.	5	0
Тест 3.	11	0

Таблица 1: Количество некорректных цепочек, распознанных синтаксическим анализатором

грамматике G_3, значительно превышает время работы другого.

Таким образом, конъюнктивная грамматика позволяет отсеивать цепочки, ложно распознаваемые КС-грамматикой, но за время, значительно большее, чем время работы анализатора по КС-грамматике.

Заключение

В ходе работы получены следующие результаты:

- реализована поддержка конъюнктивных грамматик в языке спецификации грамматик YARD;
- реализована поддержка конъюнктивных грамматик в генераторе GLL-анализаторов;
- результаты экспериментально проверены на небольших метагеномных сборках.

Дальнейшее направление работы В первую очередь, необходимо снизить время работы алгоритма. Полученная реализация предполагает полный обход дерева разбора, что, безусловно, влияет на производительность алгоритма. Кроме того, можно исследовать возможность расширения класса распознаваемых языков до булевых, на основе полученных результатов.

Список литературы

1. Rekers J. Parser Generation for Interactive Environments. 1992.
2. Megacz Adam. Scannerless boolean parsing. Electronic Notes in Theoretical Computer Science. — 2006. — Vol. 164, no. 2. — P. 97–102.
3. Okhotin Alexander. Conjunctive grammars. Journal of Automata, Languages and Combinatorics. — 2001. — Vol. 6, no. 4. — P. 519–535.
4. Scott Elizabeth, Johnstone Adrian. GLL parsing. Electronic Notes in Theoretical Computer Science. — 2010. — Vol. 253, no. 7. — P. 177–189.
5. Кириленко Я.А., Григорьев С.В., Авдюхин Д.А. Разработка синтаксических анализаторов в проектах по автоматизированному реинжинирингу информационных систем Научно-технические ведомости СПбГПУ: информатика, телекоммуникации, управление. — 2013. — no. 174. — P. 94–98.
6. YaccConstructor project. http://yaccconstructor.github.io urldate = "24.05.2016"
7. Grammar specification language YARD. http://yaccconstructor.github.io/YaccConstructor/yard.html urldate = "24.05.2016"
8. Cheng Yuan, Jikai Lei, James Cole, Yanni Sun. Reconstructing 16S rRNA genes in metagenomic data. Bioinformatics. — 2015. — Vol. 31, no. 12. — P. i35–i43.
9. Qiong Wang, Jordan A Fish, Mariah Gilman et al. Xander: employing a novel method for efficient gene-targeted metagenomic assembly. Microbiome. — 2015. — Vol. 3, no. 1. — P. 1.
10. Nawrocki Eric P, Eddy Sean R. Infernal 1.1: 100-fold faster RNA homology searches. Bioinformatics. — 2013. — Vol. 29, no. 22. — P. 2933–2935.
11. James WJ Anderson, Ádám Novák, Zsuzsanna Sükösd et al. Quantifying variances in comparative RNA secondary structure prediction BMC bioinformatics. — 2013. — Vol. 14, no. 1. — P. 149.
12. Tomita M. An Efficient Context-free Parsing Algorithm for Natural Languages. Proceedings of the 9th International Joint Conference on Artificial Intelligence – Volume 2. — IJCAI'85. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1985. — P. 756–764.

13. Stamp Mark. A revealing introduction to hidden Markov models. Department of Computer Science San Jose State University.
14. Crocker David, Overell Paul. Augmented BNF for syntax specifications: ABNF. 2005.

Оптимизация алгоритма лексического анализа динамически формируемого кода

Байгельдин Александр Юрьевич

Санкт-Петербургский государственный университет
a.baygeldin@gmail.com

Аннотация При лексическом анализе динамически формируемого кода необходимо обрабатывать нелинейный входной поток. Для этого можно использовать композицию конечных преобразователей, один из которых представляет входной поток, а второй является лексическим анализатором целевого языка. Такой подход реализован в проекте YaccConstructor, однако обладает недостаточной производительностью. Целью данной работы является исследование возможности улучшения производительности лексического анализа за счет использования более оптимального алгоритма композиции преобразователей.

Введение

Динамически формируемый код — это код, который может быть получен и использован внутри другого кода при помощи строковых операций, таких как конкатенация, циклы, замена подстроки. Яркий пример такого кода — запросы SQL, которые составляются динамически в языках более общего назначения (C#, PHP и др.). На рис. 1 представлен пример кода, составленного с помощью условного выражения и конкатенаций.

```
1  private void Go(bool cond)
2  {
3    string tableName = cond ? "Sold" : "OnSale ";
4    string queryString =
5        "SELECT ProductID, UnitPrice, ProductName "
6            + "FROM dbo.products_" + tableName
7            + "WHERE UnitPrice > 1000 "
8            + "ORDER BY UnitPrice DESC;";
9    Program.ExecuteImmediate(queryString);
10 }
```

Рис. 1: Пример динамически формируемого кода

Однако в третьей строке примера при формировании имени таблицы был пропущен пробел, и эта ошибка будет выявлена только во время выполнения программы. Таким образом, при разработке и реинжиниринге систем, использующих динамически формируемый код, можно было бы избежать множество проблем, если бы в IDE существовала поддержка статического анализа подобного кода [1]. Лексический анализ является важным шагом такого статического анализа.

Задача лексического анализа — выделение лексем во входном потоке и сохранение привязки, т.е. позиции в тексте, к исходному коду. Часто при лексическом анализе используются инструменты для генерации лексических анализаторов по спецификации языка. Лексический анализатор переводит поток символов в поток лексем и может быть представлен в виде конечного преобразователя. **Конечный преобразователь** (Finite State Transducer) — это математическая модель устройства, похожая на

конечный автомат, с тем лишь дополнением, что каждому переходу сопоставляется дополнительное значение, которое выводится в выходной поток. На рис. 2 продемонстрирован пример лексического анализатора языка арифметических выражений (ради простоты, единственной операций является операция сложения), представленного в виде конечного преобразователя.

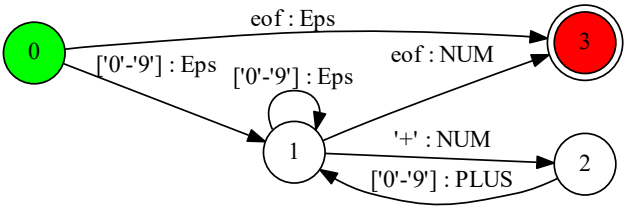


Рис. 2: Пример лексического анализатора, представленного в виде FST

Однако большинство инструментов для генерации лексических анализаторов могут работать лишь с линейным входом, что делает невозможным их непосредственное применение в нашем случае, т.к. поток символов формируется динамически. Одно из возможных решений [7] заключается в построении регулярной аппроксимации множества значений динамически формируемого выражения и последующем применении операции композиции [3] к двум конечным преобразователям: один их них построен по регулярной аппроксимации преобразованием автомата над строками в автомат над символами, а второй является классическим

лексическим анализатором для языка, на котором написан динамически формируемый код. На рис. 4 изображен пример регулярной аппроксимации динамически формируемого кода, представленной в виде конечного автомата, построенного на основе кода на рис. 3.

```
1 private void Go(int cond){
2   string columnName = cond > 3 ? "X" : (cond < 0 ? "Y" : "Z");
3   string queryString = "SELECT name" + columnName + " FROM table";
4   Program.ExecuteImmediate(queryString);}
```

Рис. 3: Пример динамически формируемого кода

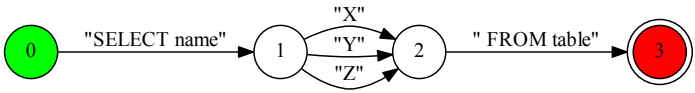


Рис. 4: Пример регулярной аппроксимации динамически формируемого кода

Результатом операции композиции конечных преобразователей является конечный преобразователь, который обладает тем свойством, что результат его работы совпадает с результатом последовательного применения участников композиции на том же входе. В нашем случае, первый конечный преобразователь переводит поток символов с их привязкой к исходному коду в поток символов, а второй (лексический анализатор) переводит поток символов в поток лексем. Таким образом, результирующий конечный

преобразователь переводит поток символов с их привязкой к исходному коду в поток лексем, что и является целью лексического анализа. Результатом применения операции композиции к конечному преобразователю на рис. 5 и лексическому анализатору на рис. 2 является конечный преобразователь изображенный на рис. 6.

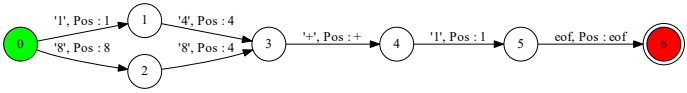


Рис. 5: Конечный преобразователь, участник операции композиции FST

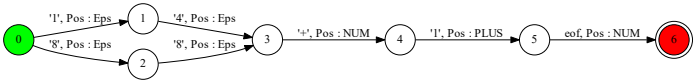


Рис. 6: Результат операции композиции FST

В существующей реализации операция композиции является основной операцией в лексическом анализе динамически формируемых языков. Однако в проекте YaccConstructor [4, 5], в рамках которого было реализовано такое решение [7], ее производительность оказалась неудовлетворительной. Таким образом, целью данной работы являлось исследование возможности улучшения производительности лексического анализа

за счет оптимизации алгоритма композиции конечных преобразователей.

1 Постановка задачи

Целью данной работы являлось исследование возможности улучшения производительности лексического анализа за счет оптимизации алгоритма композиции конечных преобразователей. Для достижения этой цели были поставлены следующие задачи:

- исследовать алгоритмы композиции конечных преобразователей;
- реализовать и интегрировать в проект YaccConstructor более оптимальный алгоритм композиции;
- сравнить производительность реализаций текущего и выбранного алгоритмов.

2 Обзор

Конечный преобразователь может быть задан следующей шестеркой элементов: $\langle Q, \Sigma, \Delta, q_0, F, E\rangle$, где

- Q — множество состояний,
- Σ — входной алфавит,
- Δ — выходной алфавит,
- $q_0 \in Q$ — начальное состояние,
- $F \subseteq Q$ — набор конечных состояний,
- $E \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Delta \cup \{\varepsilon\}) \times Q$ — набор переходов.

Композицией двух конечных преобразователей $T_1 = \langle Q_1, \Sigma_1, \Delta_1, q_{0_1}, F_1, E_1\rangle$ и $T_2 = \langle Q_2, \Sigma_2, \Delta_2, q_{0_2}, F_2, E_2\rangle$ является конечный преобразователь $T = \langle Q_1 \times Q_2, \Sigma_1, \Delta_2, \langle q_{0_1}, q_{0_2}\rangle, F_1 \times F_2, E \cup E_\varepsilon \cup E_{i,\varepsilon} \cup E_{o,\varepsilon}\rangle$, где

- $E = \{\langle\langle p, q\rangle, a, b, \langle p', q'\rangle\rangle \mid \exists c \in \Delta_1 \cap \Sigma_2 : \langle p, a, c, p'\rangle \in E_1 \wedge \langle q, c, b, q'\rangle \in E_2\}$
- $E_\varepsilon = \{\langle\langle p, q\rangle, a, b, \langle p', q'\rangle\rangle \mid \langle p, a, \varepsilon, p'\rangle \in E_1 \wedge \langle q, \varepsilon, b, q'\rangle \in E_2\}$
- $E_{i,\varepsilon} = \{\langle\langle p, q\rangle, \varepsilon, a, \langle p, q'\rangle\rangle \mid \langle q, \varepsilon, a, q'\rangle \in E_2 \wedge p \in Q_1\}$
- $E_{o,\varepsilon} = \{\langle\langle p, q\rangle, a, \varepsilon, \langle p', q\rangle\rangle \mid \langle p, a, \varepsilon, p'\rangle \in E_1 \wedge q \in Q_2\}$.

Текущая реализация алгоритма композиции FST в проекте YaccConstructor работает по определению, т.е. перебирает всевозможные комбинации переходов FST участников композиции и добавляет в результирующий FST ребра, удовлетворяющие формальным описаниям из определения композиции. Поэтому временная сложность текущего алгоритма представляется как

$$O(E_1 * E_2)$$

где E — число ребер соответствующего FST.

Для удобства эту временную сложность можно представить в следующем виде:

$$O(V_1 * V_2 * D_1 * D_2)$$

где V — число вершин, а D — максимальное число исходящих ребер соответствующего FST.

Важно заметить, что в результате работы алгоритма, несмотря на его корректность, в результирующем FST образуется множество недостижимых вершин, которые в дальнейшем с целью улучшения производительности, приходится удалять.

На замену текущему алгоритму был предложен алгоритм, представленный в статье [3], потому, что в результате его работы не образуется недостижимых вершин и он обладает лучшей асимптотической сложностью

$$O(V_1 * V_2 * D_1 * (log(D_2) + M_2))$$

где M — степень недетерминированности (максимальное количество исходящих из вершины ребер по одной метке).

Такая сложность достигается за счет поддержания очереди из пар вершин, в которых первая и вторая часть пары — вершины первого и второго FST соответственно. На каждой итерации алгоритма из очереди снимается одна пара вершин и перебираются комбинации ребер, смежных этим вершинам. Далее, если метка выходного потока на ребре первого FST совпадает с меткой входного потока на ребре второго FST, то пара из вершин, в которые входят эти ребра, добавляется в очередь, а новое правило перехода добавляется в результирующий FST. Псевдокод алгоритма представлен в листинге (11).

Algorithm 11 Композиция FST

```
1: function Compose(I_1, I_2)
2:     Q ← I_1 × I_2
3:     K ← I_1 × I_2
4:     while K ≠ ∅ do
5:         q = (q_1, q_2) ← Head(K)
6:         Dequeue(K)
7:         if q ∈ I_1 × I_2 then
8:             I ← I ⋃ {q}
9:         if q ∈ F_1 × F_2 then
10:            F ← F ⋃ {q}
11:        for ∀(e_1, e_2) ∈ E[q_1] × E[q_2] : output[e_1] = input[e_2] do
12:            q' = (target[e_1], target[e_2])
13:            if q' ∉ Q then
14:                Q ← Q ⋃ {q'}
15:                Enqueue(K, q')
16:            E ← E ⋃ {(q, input[e_1], output[e_2], q')}
17:    return T
```

Алгоритм возвращает новый FST T, множеством начальных вершин в котором является I, множеством конечных вершин — F, а множеством правил перехода —

E. За основную очередь алгоритма обозначено K, а за Q — множество посещенных вершин, которое помогает предотвратить зацикливание алгоритма на входах с циклами.

3 Реализация

3.1 Реализация алгоритма

Алгоритм был реализован на языке F# семейства .NET в библиотеке для работы с конечными преобразователями YC.FST [7]. Можно выделить следующие особенности реализации:

- Класс конечных преобразователей в этой библиотеке поддерживает только целые числа в качестве состояний, но не кортежи из целых чисел, как это представляется в псевдокоде алгоритма. Поэтому было принято решение отображать сжатые представления кортежей из целых чисел в целые числа.
- Для проверки корректности работы алгоритма были написаны модульные тесты, в которых производится композиция и сравнение FST, составленных вручную.

3.2 Интеграция

В процессе работы была произведена реорганизация проекта YaccConstructor, одной из целей которой являлось получение возможности сравнения производительности текущего и выбранного алгоритмов. В рамках интеграции были выполнены следующие задачи.

- Библиотека YC.FST была интегрирована в библиотеку для работы с графами QuickGraph [9].
- Был произведен рефакторинг, заключавшийся в подмене сторонней библиотеки QuickGraph в YaccConstructor на использование собственной сборки этой библиотеки.

– Библиотека для работы с позициями в тексте YC.Utils.SourceText [8] была интегрирована в проект YaccConstructor с целью ее дальнейшего использования при оптимизации алгоритма лексического анализа.

4 Экспериментальное исследование

Сравнение производительности было произведено на заранее построенных регулярных аппроксимациях, построенных по реальному коду, в котором запросы T-SQL (Transact-SQL) формируются динамически, и лексическом анализаторе T-SQL. Результаты измерений в таблице 1 показывают, что реализация выбранного алгоритма дает существенный прирост в производительности как на небольших синтетических примерах, так и на реальном коде, регулярные аппроксимации которого содержат большое количество ребер и вершин.

Кол-во вершин в регулярной аппроксимации	Кол-во ребер в регулярной аппроксимации	Время работы текущего алгоритма (мс)	Время работы выбранного алгоритма (мс)
2	1	74	4
8	34	133	7
40	140	676	39
215	895	4045	248
310	687	7184	394
250	738	16526	1133
711	1766	30285	2068

Таблица 1: Сравнение производительности алгоритмов композиции FST, построенных по регулярным аппроксимациям динамически формируемых выражений на языке T-SQL

На выборке из 600 примеров математическое ожидание ускорения (в количестве раз) выбранного алгоритма по сравнению с текущим равно 18.7, а среднее квадратичное отклонение равно 3.23. Однако стоит заметить, что в специфике нашей задачи выбранный алгоритм должен давать константный выигрыш в производительности для определенного языка по сравнению с текущим вне зависимости от входных данных. Но задачей данной работы являлось сравнение именно реализаций алгоритмов, а они отличаются техническими оптимизациями и необходимостью производить удаление недостижимых вершин после работы текущего алгоритма, в результате чего выигрыш в производительности перестает быть константным.

Заключение

В ходе выполнения данной работы были получены следующие результаты:

- Исследованы два различных алгоритма композиции FST.
- Алгоритм, обладающий лучшей производительностью, реализован и интегрирован в проект YaccConstructor.
- Произведено сравнение производительности реализаций алгоритмов композиции FST.
- Результаты работы представлены на конференции «Современные технологии в теории и практике программирования». Тезисы данной работы были опубликованы в сборнике материалов конференции.

Код, написанный в ходе выполнения данной работы, можно найти в репозитории проекта YaccConstructor [6]. В указанном репозитории автор принимал участие под учетной записью baygeldin.

В дальнейшем планируется исследование структур данных и алгоритмов для работы конечными преобразователями и

позициями в тексте с целью улучшения производительности алгоритма лексического анализа. Также планируется полная поддержка и интеграция с инструментом для генерации лексических анализаторов FsLex [7].

Список литературы

1. Semen Grigorev, Ekaterina Verbitskaia, Andrei Ivanov et al. String-embedded language support in integrated development environment. In Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR '14). — 2014.
2. Полубелова Марина. Лексический анализ динамически формируемых строковых выражений. Дипломная работа кафедры системного программирования СПбГУ. — 2015.
3. Mohri Mehryar. Handbook of Weighted Automata. Monographs in Theoretical Computer Science. Springer, 2009. — P. 213–254.
4. Я.А. Кириленко, С.В. Григорьев, Д.А. Авдюхин. Разработка синтаксических анализаторов в проектах по автоматизированному реинжинирингу информационных систем. 2013.
5. Сайт проекта YaccConstructor. http://yaccconstructor.github.io
6. Репозиторий проекта YaccConstructor https://github.com/YaccConstructor/YaccConstructor
7. Сайт проекта FsLex http://fsprojects.github.io/FsLexYacc/
8. Репозиторий проекта YC.Utils.SourceText https://github.com/YaccConstructor/YC.Utils.SourceText
9. Сайт проекта QuickGraph http://yaccconstructor.github.io/QuickGraph/

Использование символьных конечных преобразователей для лексического анализа динамически формируемого кода

Гумин Егор Дмитриевич

Санкт-Петербургский государственный университет
gumin.egor@gmail.com

Аннотация При статическом анализе динамически формируемого кода возникает необходимость применения лексического анализа к нелинейному входному потоку. Для решения этой задачи в проекте YaccConstructor используется композиция конечных преобразователей, представляющих входной поток и лексер целевого языка, однако существующая реализация обладает недостаточной производительностью. Целью данной работы является исследование возможности увеличения производительности лексического анализа динамического кода за счет применения библиотеки Microsoft.Automata, специализированной для работы с преобразователями.

Введение

Существует подход к написанию программного кода, при котором код на одном языке программирования формируется программой на другом языке с помощью строковых операций, циклов и условных операторов. Такой код называется динамически формируемым. В качестве примера можно рассмотреть генерацию html-страниц в языке JavaScript или

формирования SQL-запросов в C# с помощью строковых операций. На листинге 15 представлен пример динамически формируемого кода.

Listing 15 Пример динамически формируемого кода

```
1 private void Go(int cond){
2   string columnName = cond > 3 ? "X" : (cond < 0 ? "Y" : "Z");
3   string queryString =
4       "SELECT name" + columnName + " FROM table";
5   Program.ExecuteImmediate(queryString);}
```

Этот подход был достаточно распространен, поэтому существует большое множество программ, использующих его. Кроме того, иногда возникает необходимость написания такого кода (например, в случаях, когда ограничения по производительности не позволяют использовать ORM). Статический анализ такого кода, например, подсветка синтаксиса и диагностика ошибок, позволил бы существенно упростить его написание и поддержку.

Существует ряд инструментов [13–15] (подробно рассмотрены в работе [7]), позволяющих проводить статический анализ динамически формируемого кода. Однако эти инструменты обладают такими недостатками, как слабая расширяемость, ограниченная функциональность, применимость только к конкретным языкам. Для решения этих проблем [11] в рамках проекта YaccConstructor [2, 8], который посвящен исследованиям в области лексического и синтаксического анализа, реализована платформа для работы со встроенными языками. Модульность платформы позволяет легко добавлять дополнительную функциональность и поддержку новых языков. Статический анализ динамического кода производится в несколько шагов:

1. построение регулярной аппроксимации сверху множества значений динамически формируемого выражения;
2. лексический анализ;
3. синтаксический анализ;
4. семантический анализ.

Но механизм лексического анализа [7] в проекте YaccConstructor, использующий композицию конечных преобразователей, обладает недостаточной производительностью. Вероятная причина этой проблемы — разрастание конечных преобразователей на больших алфавитах, так как обработка большого количества дуг занимает много времени. Возможное решение — применение символьных конечных преобразователей [1], с помощью которых можно представить структуры данных более компактно. На данный момент единственной библиотекой для работы с символьными конечными преобразователями в рамках платформы .NET является библиотека Microsoft.Automata [3], которая активно поддерживается сообществом Microsoft Research, именно поэтому она и является предметом изучения. В рамках данной работы исследован вопрос о возможности увеличения производительности лексического анализа с помощью использования символьных конечных преобразователей из библиотеки Microsoft.Automata.

1 Постановка задачи

Целью данной работы является исследование применимости символьных конечных преобразователей для лексического анализа. Для ее достижения поставлены следующие задачи:

– Изучить возможности библиотеки Microsoft.Automata.
– Провести сравнение производительности алгоритма операции композиции над символьными конечными

преобразователями в библиотеке Microsoft.Automata с производительностью операции композиции над конечными преобразователями в исследовательском проекте YaccConstructor.
– На основании полученных результатов сделать выводы о применимости библиотеки Microsoft.Automata для лексического анализа в проекте YaccConstructor.

2 Обзор

В данной главе определены основные понятия и проведен обзор основных применяемых формализмов, инструментов и технологий.

2.1 Конечные преобразователи

Конечный преобразователь (Finite State Transducer) [10] — формализм, напоминающий конечный автомат, который при переходе из состояния в состояние пишет символ в выходной поток. Конечный преобразователь можно задать шестеркой элементов: $\langle Q, \Sigma, \Delta, q_0, F, E \rangle$, где

– Q — множество состояний,
– Σ — входной алфавит,
– Δ — выходной алфавит,
– $q_0 \in Q$ — начальное состояние,
– $F \subseteq Q$ — набор конечных состояний,
– $E \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Delta \cup \{\varepsilon\}) \times Q$ — набор переходов.

Важнейшей для лексического анализа операцией над конечными преобразователями является операция композиции. Результат композиции двух конечных преобразователей — конечный преобразователь, работающий так, будто входной поток первого переправили во входной поток второго. Формальное определение дано ниже.

Композицией конечных преобразователей $T_1 = \langle Q_1, \Sigma_1, \Delta_1, q_{0_1}, F_1, E_1\rangle$ и $T_2 = \langle Q_2, \Sigma_2, \Delta_2, q_{0_2}, F_2, E_2\rangle$ является конечный преобразователь $T = \langle Q_1 \times Q_2, \Sigma_1, \Delta_2, \langle q_{0_1}, q_{0_2}\rangle, F_1 \times F_2, E \cup E_\varepsilon \cup E_{i,\varepsilon} \cup E_{o,\varepsilon}\rangle$, где

- $E = \{\langle\langle p, q\rangle, a, b, \langle p', q'\rangle\rangle \mid \exists c \in \Delta_1 \cap \Sigma_2 : \langle p, a, c, p'\rangle \in E_1 \wedge \langle q, c, b, q'\rangle \in E_2\}$
- $E_\varepsilon = \{\langle\langle p, q\rangle, a, b, \langle p', q'\rangle\rangle \mid \langle p, a, \varepsilon, p'\rangle \in E_1 \wedge \langle q, \varepsilon, b, q'\rangle \in E_2\}$
- $E_{i,\varepsilon} = \{\langle\langle p, q\rangle, \varepsilon, a, \langle p, q'\rangle\rangle \mid \langle q, \varepsilon, a, q'\rangle \in E_2 \wedge p \in Q_1\}$
- $E_{o,\varepsilon} = \{\langle\langle p, q\rangle, a, \varepsilon, \langle p', q\rangle\rangle \mid \langle p, a, \varepsilon, p'\rangle \in E_1 \wedge q \in Q_2\}$.

На рис. 1 — 3 представлен пример композиции конечных преобразователей. Результатом композиции конечного преобразователя T_1 с рис. 1 с конечным преобразователем T_2 с рис. 2 будет конечный преобразователь с рис. 3.

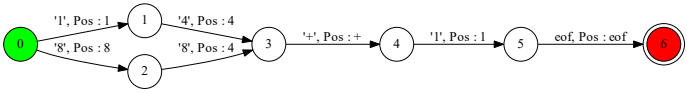


Рис. 1: Конечный преобразователь T_1

2.2 Символьные конечные преобразователи

При разработке лексических анализаторов часто возникает ситуация, когда из одного состояния в другое необходимы переходы сразу по нескольким символам. Преобразователи часто реализовывают как некоторые графы, и каждый переход представляется как дуга в этом графе. Соответственно, при разработке лексических анализаторов часто приходится добавлять много дублирующих ребер,

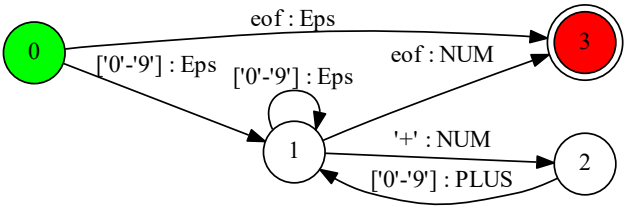


Рис. 2: Конечный преобразователь T_2

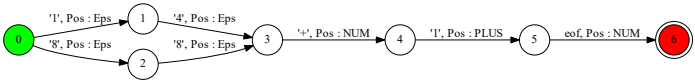


Рис. 3: Результат композиции конечных преобразователей T_1 и T_2

например, если необходимо по любой цифре перейти из состояния A в состояние B, нужно добавить 10 дуг (рис. 4). Чем больше ребер в графе, тем больше расход памяти и ниже производительность.

Для борьбы с этой проблемой был предложен символьный конечный преобразователь (Symbolic Transducer) — конечный преобразователь, каждому переходу которого можно сопоставить не один символ, а формулу (например, регулярное выражение). Таким образом, в приведенном выше примере вместо десяти дуг будет одна дуга, которой соответствует регулярное выражение (рис. 5), описывающее цифры. Такие преобразователи получаются гораздо более компактными.

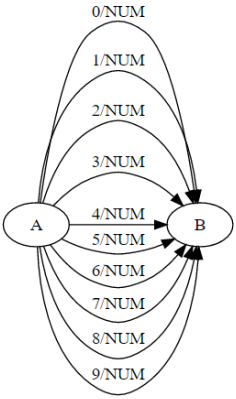


Рис. 4: Пример конечного преобразователя

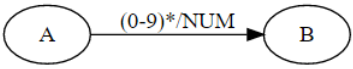


Рис. 5: Символьный конечный преобразователь, эквивалентный конечному преобразователю на рис. 4

Символьный конечный преобразователь можно задать шестеркой элементов: $\langle Q, \Sigma, \Delta, q_0, F, E \rangle$, где

- Q — множество состояний,
- Σ — входной алфавит,
- Δ — выходной алфавит,
- $q_0 \in Q$ — начальное состояние,
- $F \subseteq Q$ — набор конечных состояний,
- $E \subseteq Q \times (R(\Sigma) \cup \{\varepsilon\}) \times (\Delta \cup \{\varepsilon\}) \times Q$ — набор переходов, $R(X)$ — некоторая формула (в нашем случае — регулярное выражение) над множеством X.

2.3 Лексический анализ

Основной задачей лексического анализа является выделение лексем во входном потоке с сохранением их привязки к исходному коду. При классическом лексическом анализе поток символов линеен, но в случае анализа встроенного кода он является нелинейным. Поэтому применяется следующий подход:

1. В некоторой точке программы по множеству значений строкового выражения строится конечный автомат M, аппроксимирующий его сверху.
2. По автомату M строится конечный преобразователь.
3. Производится композиция конечного преобразователя M' с конечным преобразователем, являющимся лексическим анализатором языка, на котором написано выражение.
4. По результирующему преобразователю строится конечный автомат над лексемами, который и является токенизированным входом.

Сложность операции композиции зависит от количества ребер, символьные преобразователи обычно содержат меньшее их количество, чем эквивалентные конечные преобразователи, что позволяет увеличить производительность композиции.

2.4 Библиотека Microsoft.Automata

Microsoft.Automata — .NET-библиотека, содержащая реализации конечных автоматов, конечных преобразователей, операции над ними и средства их анализа. Библиотека состоит из следующих модулей.

– Automata — основной модуль с реализацией автоматов и преобразователей, содержащий в себе такие элементы, как:
  - синтаксические анализаторы грамматик для построения по ним автоматов;

- синтаксические анализаторы регулярных выражений, используемых при построении преобразователей;
- различные структуры данных, например, BDD;
- символьные автоматы и операции над ними;
- символьные преобразователи и операции над ними [12].

– Automata.Z3 — модуль, обеспечивающий интеграцию реализованных формализмов с SMT-решателем Z3 [4, 9].

– Bek — модуль, позволяющий работать с преобразователями и автоматами, используя язык Bek Z3 [5, 6]. Язык Bek — предметно-ориентированный язык для написания и анализа конечных преобразователей, работающих со строками, который является аналогом регулярных выражений для автоматов. Программы на Bek позволяют ответить на вопросы «выводят ли две программы одну и ту же строку», «может ли эта программа вывести некоторую строку?», «что будет, если выполнить композицию программ?».

На листинге 16 приведен пример программы на языке Bek. Программа проходит по строке, используя переменную c, чтобы итерироваться по символам и булеву переменную b, чтобы отслеживать, был ли предыдущий символ экранирующим и добавляет экранирующий символ перед одинарными или двойными кавычками, если он пропущен.

Listing 16 Пример программы на языке Bek

```
program sanitize(t);
  string s;
  s := iter(c in t){b := false;}{
       case (!(b) && ((c == '\'') || (c == '\"'))) :
           b := false;
           yield ('\\');
           yield (c);
       case (c == '\\') :
           b := !(b);
           yield (c);
       case (true) :
           b := false;
           yield (c);
       };
  return s;
```

3 Окружение для тестирования и экспериментальное исследование

В данной главе описывается реализация окружения для тестирования производительности операции композиции на символьных конечных преобразователях в библиотеке Microsoft.Automata и конечных преобразователях в YaccConstructor. Также приведено описание процесса тестирования и сделаны выводы о применимости символьных преобразователей из библиотеки Microsoft.Automata в исследовательском проекте YaccConstructor.

При изучении возможностей библиотеки, проведенном в рамках обзора, было установлено, что полная интеграция проекта YaccConstructor и библиотеки Microsoft.Automata затруднительна. По этой причине для оценки производительности библиотеки Microsoft.Automata было необходимо создать окружение для тестирования,

обеспечивающее возможность генерации тестов по одинаковым данным для разнородных сред: инструмента YaccConstructor и библиотеки Microsoft.Automata.

3.1 Окружение для тестирования

Предложенная архитектура среды для тестирования изображена на рис. 6, цветом обозначены модули, добавленные или модифицированные в рамках данной работы.

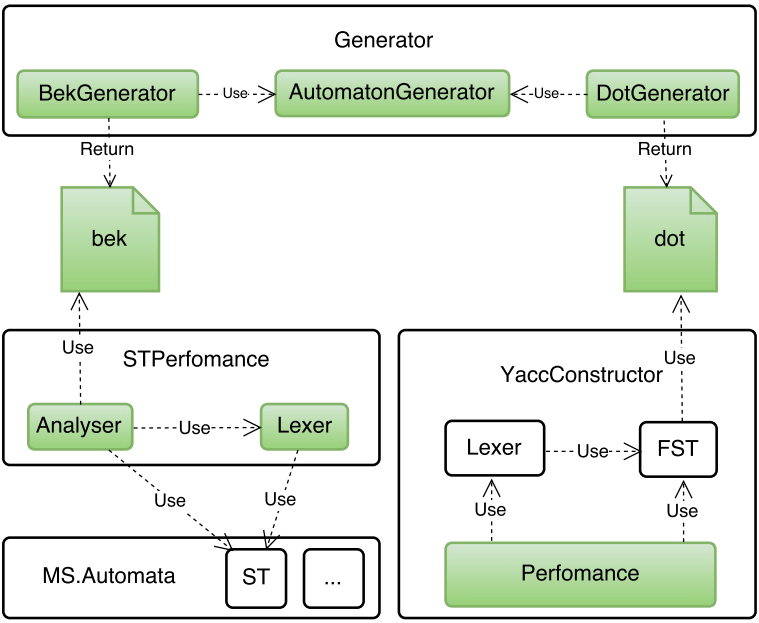


Рис. 6: Архитектура среды для тестирования

Окружение состоит из:

- генератора конечных автоматов для создания на их основе эквивалентных преобразователей для инструмента YaccConstructor и библиотеки Microsoft.Automata;
- генератора конечных преобразователей;
- генератора символьных конечных преобразователей;
- лексического анализатора в форме конечного преобразователя;
- лексического анализатора в форме символьного конечного преобразователя;
- модуля для анализа производительности операции композиции в проекте YaccConstructor;
- модуля для анализа производительности операции композиции в библиотеке Microsoft.Automata.

Подробное описание компонентов приведено далее.

3.2 Лексический анализатор

Для сравнения времени работы операции композиции на символьных конечных преобразователях и конечных преобразователях было решено реализовать лексический анализатор арифметических выражений, аналогичный анализатору, который уже был интегрирован в проект YaccConstructor, так как язык арифметических выражений является достаточно показательным примером и включается практически в любой язык программирования. Лексический анализатор может быть представлен в виде символьного конечного преобразователя и используется для анализа производительности библиотеки Microsoft.Automata. Реализация данного лексического анализатора выполнена на языке Bek.

3.3 Генераторы преобразователей

Генератор конечных автоматов (обозначен как AutomatonGenerator на рис. 6) генерирует случайные

конечные автоматы с указанным количеством дуг и вершин. Результат его работы — файл с промежуточным представлением конечного автомата, который далее интерпретируется генераторами конечных преобразователей (обозначены как BekGenerator и DotGenerator на рис. 6), которые на его основе строят программы, представляющие эквивалентные преобразователи, записанные на языках Bek и DOT соответственно.

3.4 Тестирование производительности композиции в проекте YaccConstructor на конечных преобразователях

За тестирование производительности операции композиции на конечных преобразователях в проекте YaccConstructor отвечает модуль AbstractLexer.Interpreter.Perfomance (обозначен как Perfomance на рис. 6). По dot-файлам лексического анализатора арифметических выражений и dot-файлам, сгенерированными модулем DotGenerator строятся конечные преобразователи. Затем производится композиция преобразователя, построенного по лексическому анализатору с каждым из тестовых преобразователей. Тест многократно повторяется и для каждого теста берется среднее значение затраченного времени.

3.5 Тестирование производительности композиции в библиотеке Microsoft.Automata на символьных конечных преобразователях

За тестирование производительности операции композиции на символьных конечных преобразователях в библиотеке Microsoft.Automata отвечает модуль PerfomanceTest (обозначен как STPerfomance на рис. 6). По bek-файлам лексического анализатора арифметических выражений и bek-файлам, сгенерированными модулем BekGenerator

строятся символьные конечные преобразователи. Затем производится композиция символьного преобразователя, построенного по лексическому анализатору с каждым из тестовых преобразователей. Тест многократно повторяется и для каждого теста берется среднее значение затраченного времени.

3.6 Экспериментальное исследование

С помощью описанного окружения был поставлен следующий эксперимент:

1. Сгенерированы пять конечных автоматов, характеристики которых указаны в таблице 1.
2. По автоматам сгенерированы программы на языке DOT.
3. По автоматам сгенерированы программы на языке Bek.
4. Проведена композиция сгенерированных конечных преобразователей из dot-файлов с конечным преобразователем, представляющим лексический анализатор языка арифметических выражений из проекта YaccConstructor (операция повторялась 100 раз, в качестве результирующего значения взято среднее).
5. Проведена композиция сгенерированных символьных конечных преобразователей из bek-файлов с символьным конечным преобразователем, представляющим лексический анализатор языка арифметических выражений, аналогичный анализатору из YaccConstructor (операция повторялась 100 раз, в качестве результирующего значения взято среднее).

В результате эксперимента получены данные, которые представлены на рис. 7.

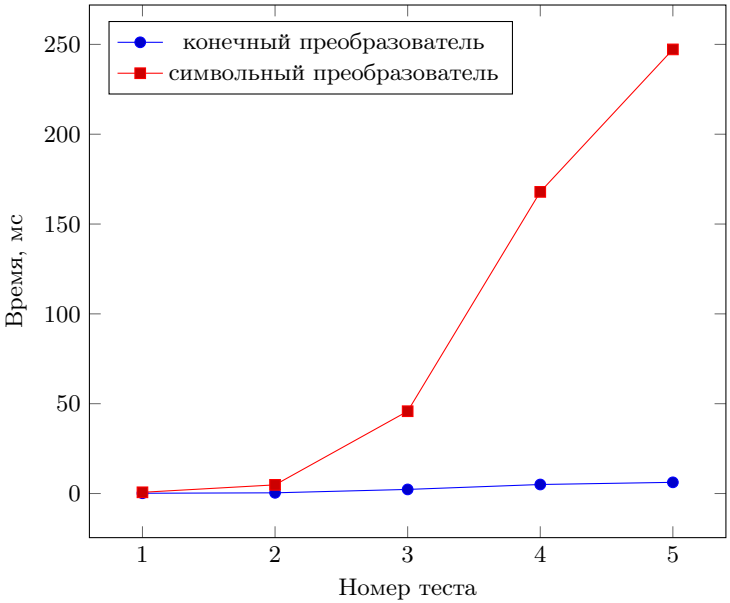


Рис. 7: Результаты измерения скорости работы операции композиции на символьных преобразователях и конечных преобразователях

Тест	Ребра	Вершины
1	1	2
2	6	5
3	25	20
4	50	5
5	60	50

Таблица 1: Размеры сгенерированных автоматов

Эксперимент показал, что операция композиции над символьными конечными преобразователями в библиотеке Microsoft.Automata работает гораздо медленнее, чем операция композиции над конечными преобразователями в проекте YaccConstructor. В связи с этим было принято решение проанализировать производительность самой библиотеки.

Анализ горячих точек показал, что при выполнении задач лексического анализа большую часть времени библиотека тратит на обращения к ядру библиотеки Z3. Поскольку в наших задачах отсутствует необходимость в использовании Z3, было решено в качестве эксперимента отключить некоторые затратные проверки условий, чтобы сократить количество обращений к ядру. Эксперимент показал, что таким образом можно как минимум увеличить производительность операции композиции на 20% на некоторых тестах (время работы операции композиции на модифицированной библиотеке отражено на рис. 8), что показывает, что адаптация библиотеки к нашей задаче может способствовать увеличению производительности, но этого недостаточно для получения оптимальных результатов.

В результате экспериментов были сделаны выводы, что библиотека Microsoft.Automata, разработанная для решения задач из области формальной верификации, недостаточно производительна для лексического анализа. Большое количество обращений к ядру Z3 перекрывает все достоинства, приобретаемые от использования оптимального формализма и структур данных. Таким образом, применение библиотеки Microsoft.Automata в инструменте YaccConstructor для лексического анализа в настоящий момент неоптимально. Использование же символьных конечный преобразователей для лексического анализа все еще выглядит перспективным и подлежит дальнейшему исследованию.

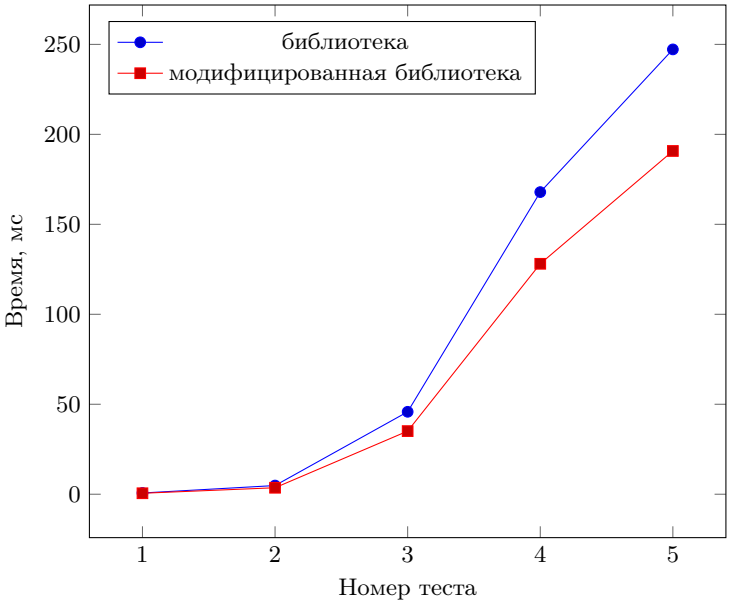


Рис. 8: Результаты измерения скорости работы операции композиции на символьных преобразователях с модифицированной и обычной версиями библиотеки

Заключение

В рамках выполнения данной работы были получены следующие результаты:

- Исследованы возможности библиотеки Microsoft.Automata.
- Проведено сравнение производительности алгоритма операции композиции над символьными конечными преобразователями в библиотеке с производительностью

операции композиции над конечными преобразователями в проекте YaccConstructor.
– На основании полученных результатов сделаны выводы о необходимости написания собственной реализации символьного конечного преобразователя.
– Результаты работы представлены на конференции «Современные технологии в теории и практике программирования», тезисы опубликованы в сборнике материалов конференции.

Код генераторов автоматов и преобразователей, а также код, тестирующий производительность символьных преобразователей в библиотеке Microsoft.Automata можно найти на сайте https://github.com/GuminEgor/Automata. Код, тестирующий производительность конечных преобразователей в проекте YaccConstructor можно найти на сайте https://github.com/GuminEgor/YaccConstructor. В указанных репозиториях автор принимал участие под учетной записью GuminEgor.

В дальнейшем необходима реализация собственной версии символьного конечного преобразователя, адаптированного под задачи лексического анализа, тестирование его производительности и, в случае удовлетворительных результатов, его интеграция в проект YaccConstructor.

Список литературы

1. Nikolaj Bjørner, Margus Veanes. Symbolic transducers: Technical Report MSR-TR-2011-3. Microsoft Research, 2011.
2. Сайт проекта YaccConstructor. https://github.com/YaccConstructor/
3. Сайт проекта Mircosoft.Automata. http://research.microsoft.com/en-us/projects/automata/
4. Сайт проекта Z3. https://github.com/Z3Prover/z3/
5. Сайт проекта Bek. http://research.microsoft.com/en-us/projects/bek/
6. Pieter Hooimeijer, Benjamin Livshits, David Molnar et al. Fast and precise sanitizer analysis with BEK. Proceedings of the 20th USENIX conference on Security, USENIX Association. 2011.

7. Полубелова М.И. Лексический анализ динамически формируемых строковых выражений. http://se.math.spbu.ru/SE/diploma/2015/bmo/444-Polubelova-report.pdf
8. Я.А. Кириленко, С.В. Григорьев, Д.А. Авдюхин. Разработка синтаксических анализаторов в проектах по автоматизированному реинжинирингу информационных систем. 2013.
9. De Moura Leonardo, Bjørner Nikolaj. Z3: An efficient SMT solver. Tools and Algorithms for the Construction and Analysis of Systems. — Springer, 2008. — P. 337–340.
10. Hanneforth Thomas. Finite-state Machines: Theory and Applications Unweighted Finite-state Automata. Institut fur Linguistik Universitat Potsdam.
11. Григорьев С.В. Синтаксический анализ динамически формируемых программ. 2016.
12. Margus Veanes, Pieter Hooimeijer, Benjamin Livshits et al. Symbolic finite state transducers: Algorithms and applications. ACM SIGPLAN Notices. — 2012. — Vol. 47, no. 1. — P.137–150.
13. Aivar Annamaa, Andrey Breslav, Jevgeni Kabanov, Varmo Vene. An Interactive Tool for Analyzing Embedded SQL Queries. Programming Languages and Systems. — Springer: Berlin, 2010. — P. 131–138.
14. Christensen Aske Simon, Møller Anders, I.Schwartzbach Michael. Precise Analysis of String Expressions. Proc. 10th International Static Analysis Symposium (SAS). — Springer-Verlag: Berlin, 2003. — June. — P. 1–18.
15. Minamide Yasuhiko. Static approximation of dynamically generated web pages. In Proceedings of the 14th International Conference on World Wide Web, WWW '05. — ACM, 2005. — P. 432–441.