Санкт-Петербургский государственный университет

Фундаментальная информатика и информационные технологии
Профиль Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

Рагозина Анастасия Константиновна

Ослабленный синтаксический анализ динамически формируемых выражений на основе алгоритма GLL

Магистерская диссертация

Научный руководитель:
к. ф.-м. н., доц. Булычев Д. Ю.

Рецензент:
ООО "ИнтеллиДжей Лабс"
инженер-программист Шкредов С. Д.

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY

Fundamental Computer Science and Information Technologies
Software of Computers, Complexes and Networks

Ragozina Anastasiya

GLL-based relaxed parsing of dynamically generated code

Master's Thesis

Scientific supervisor:
assistant professor Dmitri Boulytchev

Reviewer:
IntelliJ Labs Co. Ltd
Software Developer Sergey Shkredov

Saint-Petersburg
2016

Оглавление

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 [**?**], 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 [**?**, **?**, **?**]. Изучению данной проблемы посвящена также работа [**?**], 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 анализа 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 — суммарная длина 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 [**?**]: 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 [**?**], RNGLR, BRNGLR [**?**]. 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) [**?**] 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 **??**) и соответствующий анализатор, написанный методом рекурсивного спуска.

$$
\begin{aligned}
s &\rightarrow a\ b \mid b\ C \\
b &\rightarrow A \mid C \\
a &\rightarrow A
\end{aligned}
$$

Листинг 1: Грамматика G_0

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. **??** показан пример объединения нескольких стеков: для стеков S1, S2 и S3 в результате будет получен S1.

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 входном 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 **??**).

Для такой грамматики в процессе разбора будет построен GSS (в

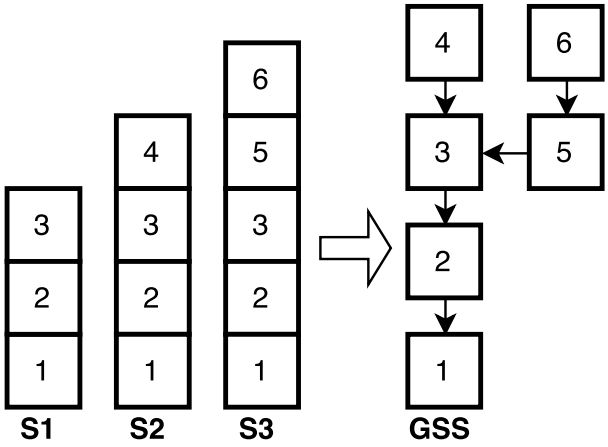

Рис. 1: Стек, структурированный в виде графа

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

Листинг 2: Грамматика G_1

дальнейшем просто стек), показанный на рис. 2(а). На рисунке видно, что рёбра дублируются. Этот же стек может быть представлен как показано на рис. 2(б). Таким образом можно значительно уменьшить размер графа без потери информации.

Данная модификация стека предложена в работе [**?**], 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 [**?**]. В данной работе были использованы

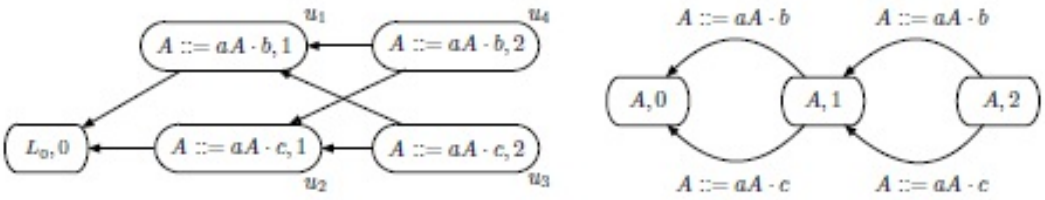

Рис. 2: а — старая версия GSS; б — модифицированный GSS

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 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) [**?**], 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 **??**) и входной цепочки "bbb" может быть построено два дерева, показанных на рис. **??**(а) и рис. **??**(б), которые могут быть сжаты в SPPF так, как показано на рис. **??**(в).

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

Листинг 3: Грамматика G_2

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 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

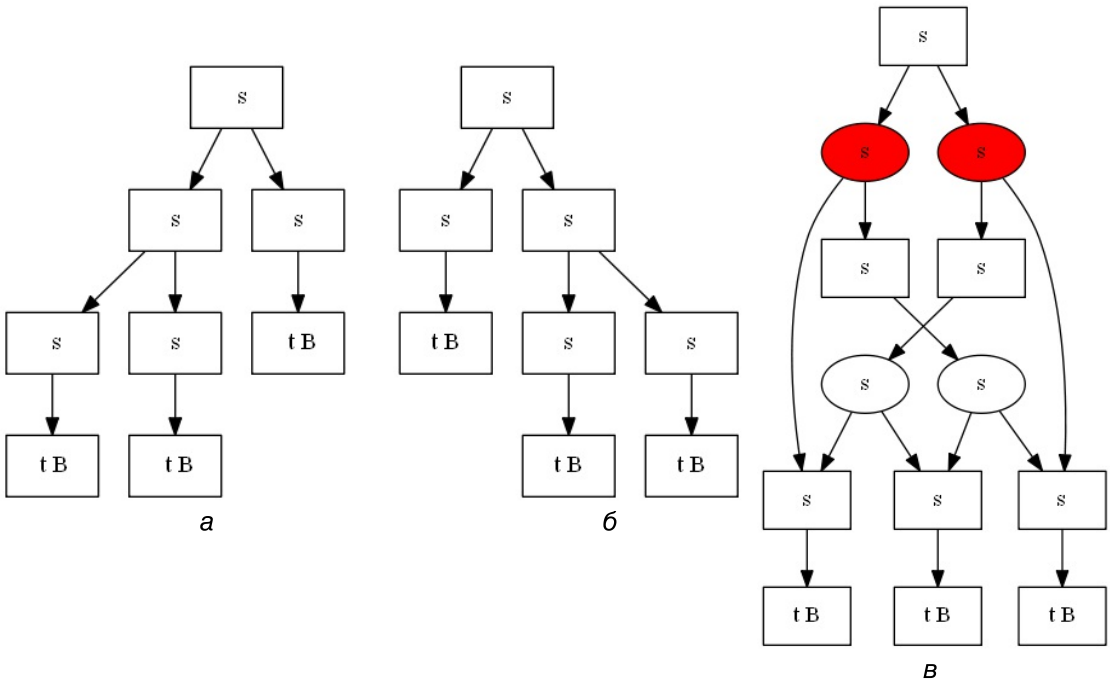

Рис. 3: а — левый вывод; б — правый вывод; в — сжатое представление леса разбора

имеют потомков и являются листьями в SPPF. Нетерминальные и промежуточные узлы могут иметь несколько потомков, упакованные 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. **??** изображен SPPF для грамматики G_2.

На рис. **??** 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.

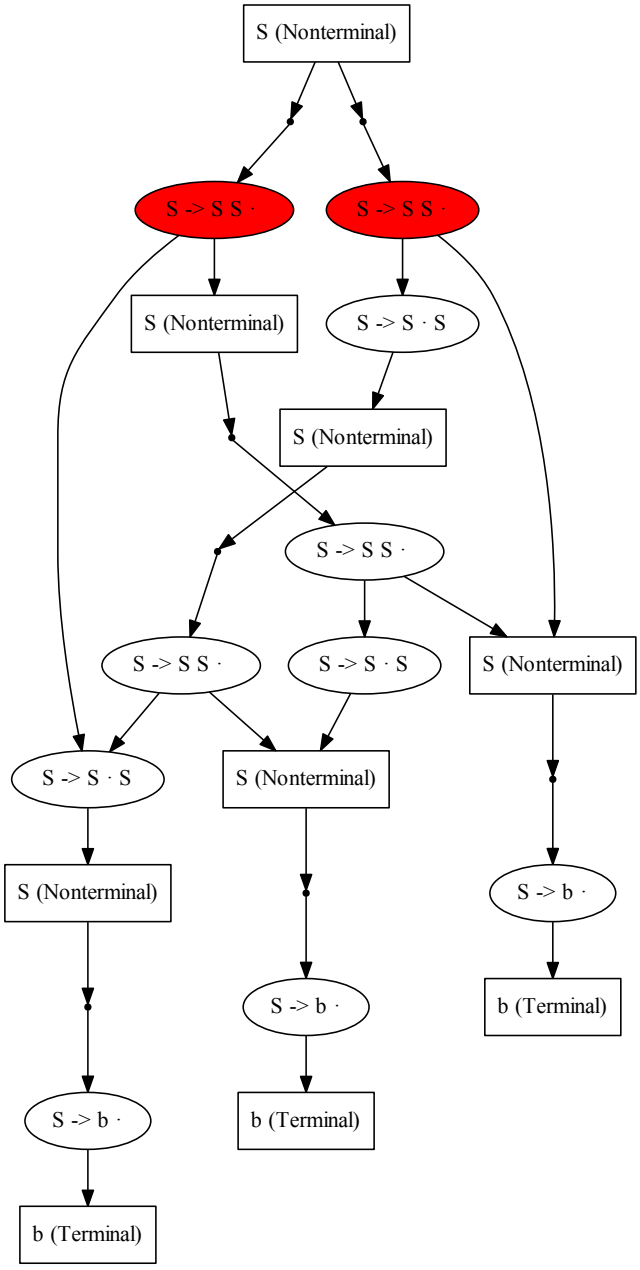

Рис. 4: Бинаризированное представление сжатого леса разбора для грамматики G_2 и входной строки “bbb”

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 останавливается, как только очередь дескрипторов становится пустой.

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

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

Листинг 4: Грамматика G_3

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. Ниже приведён пример GLL-анализатора для грамматики G_3. В приведённом примере функция L_0 содержит основной цикл,

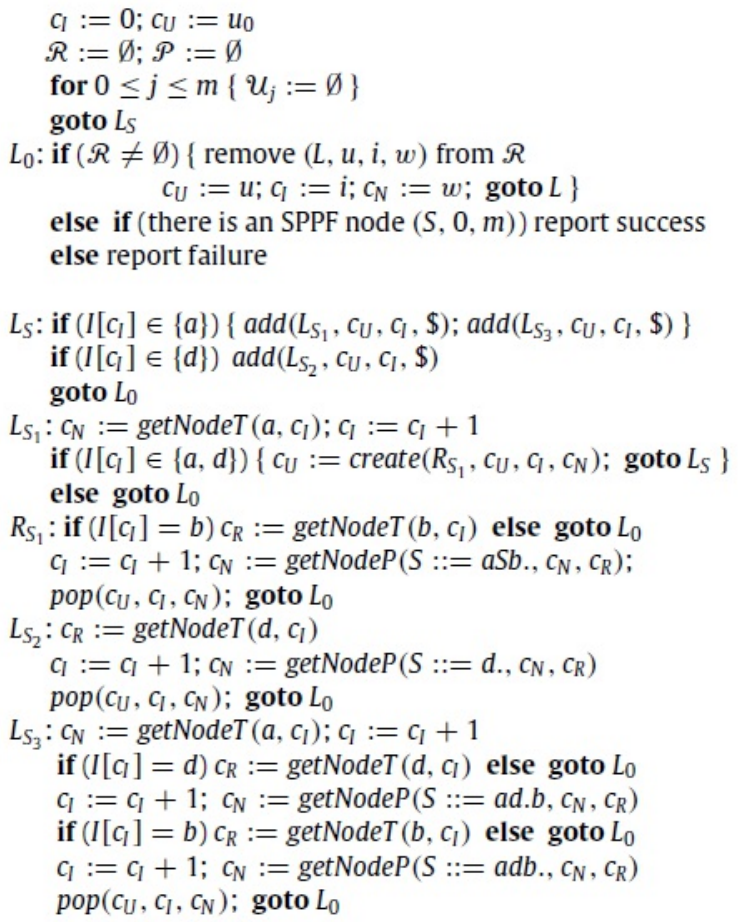

Рис. 5: GLL-анализатор для грамматики G_3

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) [**?**, **?**] — 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.

- PHP String Analyzer (PHPSA) [**?**, **?**] — инструмент для статического анализа строк в программах на PHP. Расширяет подход инстру-

мента JSA. В инструменте уточнена проводимая аппроксимация и это повышает точность проводимого анализа.

- Alvor [?, ?, ?] — 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 [?] — 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 разметка переменных, содержащих выражения на том или ином встроенном языке.

- Varis [?] — 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 [**?**]. Алгоритм абстрактного синтаксического анализа комбинирует анализ потока данных и синтаксический LR-анализа. 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 [**?**], 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 [**?**]. 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 конечный автомат, такой 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 [**?**] — 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# [**?**]. Предоставляемый язык спецификации грамматик YARD, поддерживает атрибутные грамматики, 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 [**?**]. В рамках соответствующих модулей реализованы механизмы лексического анализа, описанный в [**?**], и алго-

ритм синтаксического анализа, описанный в [?]. 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 [?], REAGO [?], ковариационые модели (covariation model, CM) [?], 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 оказывается 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 о 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 [**?**]. 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 [**?**], 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 [**?**]. Наиболее известным инструментом, использующим CM для поиска рРНК является Infernal [**?**]. 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 [**?**] позволяет решать задачу построения леса вывода для динами-

чески формируемого кода. Поэтому идеи, предложенные в этих работах взяты за основу данной диссертации.

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 RNGL-алгоритме. При этом архитектура этого решения позволяет легко заменять алгоритмы синтаксического анализа, оставляя без изменений другие компоненты, такие как лексический анализ, построение аппроксимации [**?**]. 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 **??** будет построен конечный автомат на рис. **??**.

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

Листинг 5: Код на C#, динамически формирующий скобочную последовательность

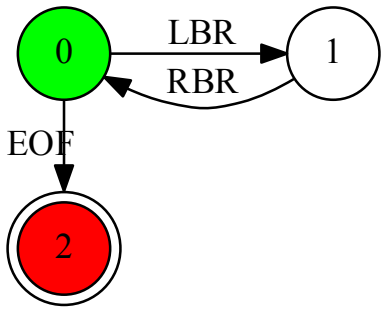

Рис. 6: Конечный автомат, представляющий аппроксимацию встроенного кода для листинга **??**

В алгоритме вместо хранения позиции во входном потоке теперь хранится номер вершины во входном графе. Поскольку вход является

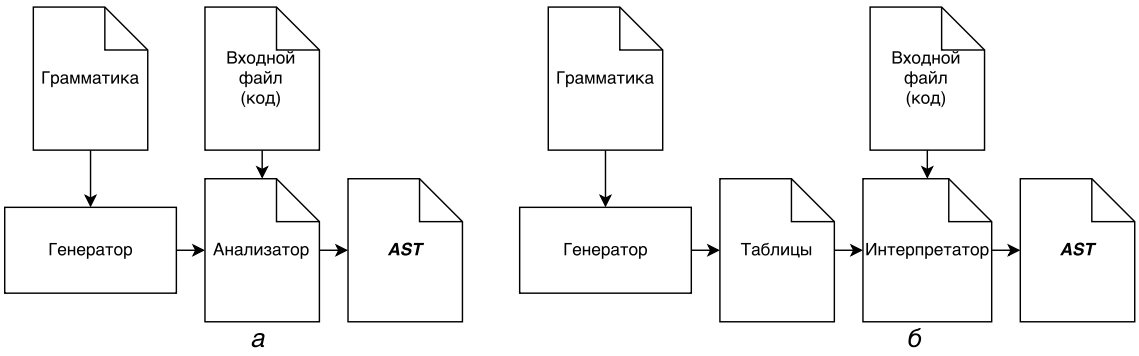

Рис. 7: Подходы к генерации синтаксических анализаторов

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 [**?**] — 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. **??**(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 переис-

пользуется. По грамматике каждый раз генерируется дополнительная информация, которая необходима интерпретатору в процессе работы. Структура такого генератора представлена на рис. **??**(б). Чаще всего в качестве дополнительной информации генерируются таблицы синтаксического анализа, управляющие процессом разбора.

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 , 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 следующий 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 рас-

смотрен, 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 в отличие от алгоритма, основанного на RNGLR , в который потребовалось внести существенные изменения для поддержки циклов [**?**]. 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 так же 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 алгоритма, функции управляющей процессом разбора и функций для построения дерева и стека.

```
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 with curSmb
                    cR ← getNodeT(curEdge)
                    cI ← curEdge.TargetVertex
                    cL ← label(cL.rule, cL.position + 1)
                    cN ← getNodeP(cL, cN, cR)
                    condition ← false
                end if
            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, $)
                    end for
                end for
            end if
        else
            pop(cU, cI, cN)
        end if
    end if
end function
```

Листинг 6: Функция, содержащая в себе основную логику алгоритма

```
function CONTROL()
    condition ← true
    while not stopr do
        if condition then dispatcher()
        elseprocessing()
        end if
    end while
end function
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
    end if
end function
```

Листинг 7: Функции, управляющие процессом разбора

На листинге **??** 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 очереди дескриптор, присваивает значения переменным. Функция `parsing()` на листинге **??** содержит в себе основную логику алгоритма.

3.1. Пример работы алгоритма

Рассмотрим следующий пример. В качестве входных данных будем испоьзовать конечный автомат M, представленный на рис. **??**, который генерирует произвольные скобочные последовательности. Необходимо построить лес разбора для всех цепочек, порождаемых автоматом M, выводимых в грамматике G_4 (листинг **??**), описывающей язык правиль-

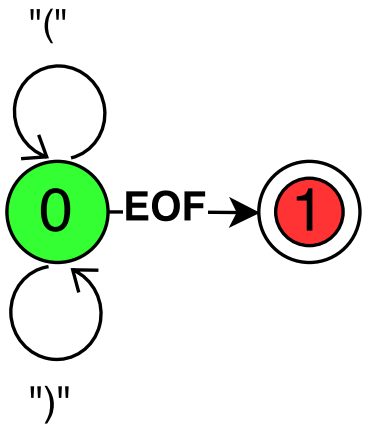

Рис. 8: Конечный автомат, подаваемый на вход анализатору

ных скобочных последовательностей.

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

Листинг 8: Грамматика G_4

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

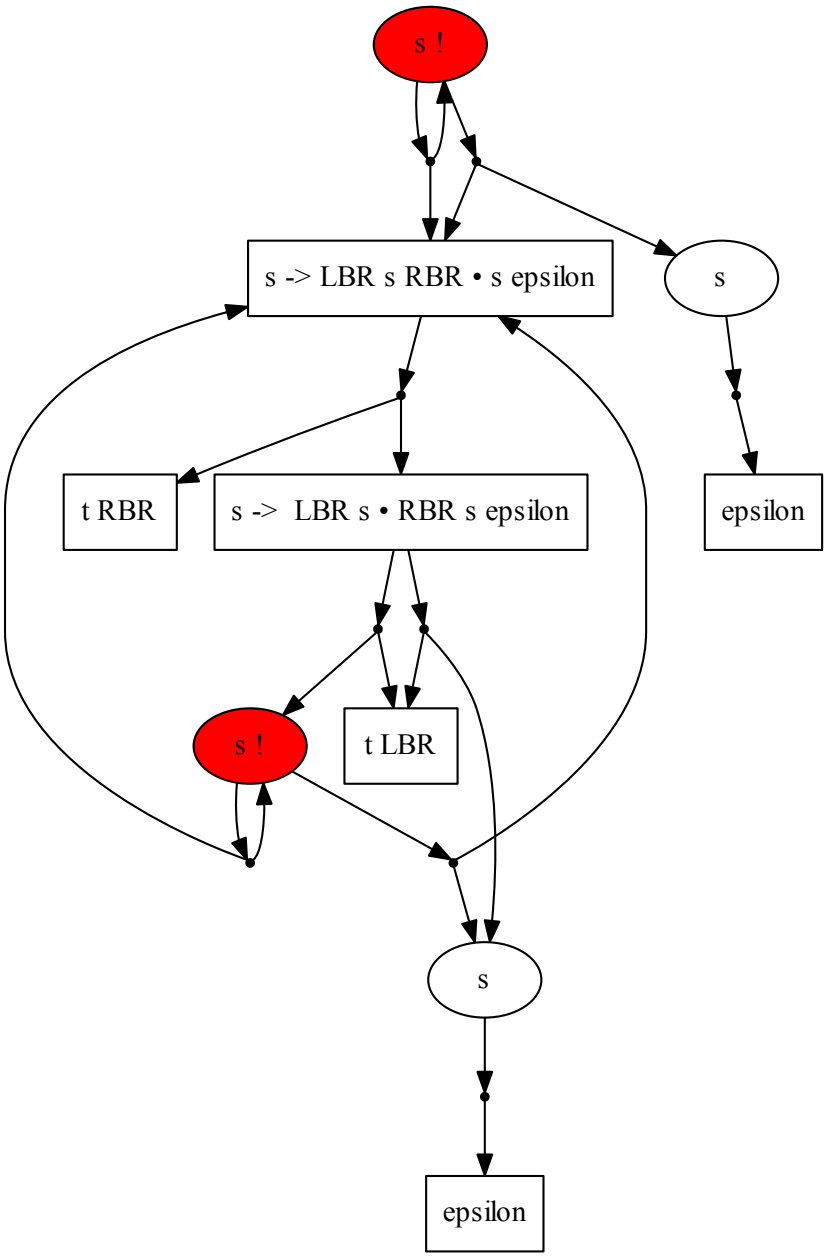

Рис. 9: Сжатое представление леса разбора для грамматики G_4 и конечного автомата на рис. **??**

дескрипторов конечно.

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 де-

скриптор. 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 доказали 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 Г 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 дескрипторов добавляются дескрипторы для всех альтернатив стартового правила, соответствующие терминалу во входном потоке. Аналогичная ситуация происходит, как только в грамматике встречается нетерминал. Рассматриваются все альтерна-

тивы 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 [**?**].

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. Нужно лишь ответить на вопрос: порождает ли входной автомат данную подстроку или нет и вернуть коорди-

наты 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 сказывается 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. **??**, где изображена архитектура инструмента YaccConstructor и цветом выделен реализованный модуль.

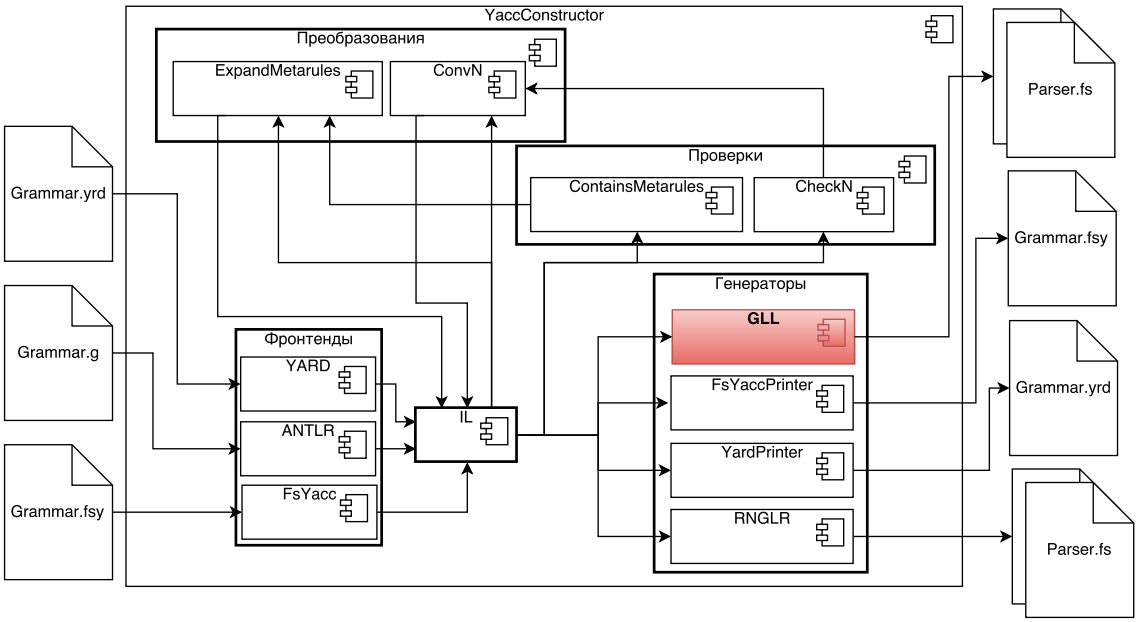

Рис. 10: Архитектура инструмента YaccConstructor (рисунок взят из работы [**?**])

Внутреннее устройство этого модуля показано на рис. **??**. 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, метапрвила), 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.

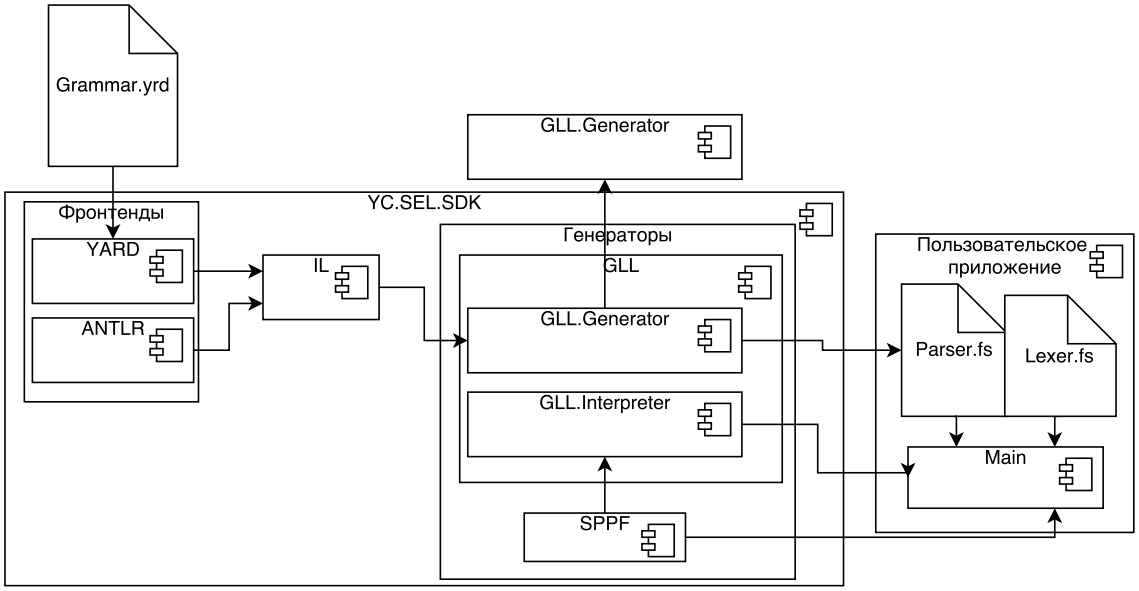

Рис. 11: Принцип работы реализованного модуля (рисунок взят и модифицирован из работы [**?**])

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 [**?**]. 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 [**?**]. При реализации описанного алгоритма подобные проблемы так же возникли: 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 [**?**], 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 [?]. Подобные задачи являются интересными с инженерной точки зрения и часто возникают на практике.

Важной задачей так же является представление метагеномной сборки и её обработка, так как, в отличие от графа, являющегося аппроксимацией встроенных языков, граф, представляющего 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 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

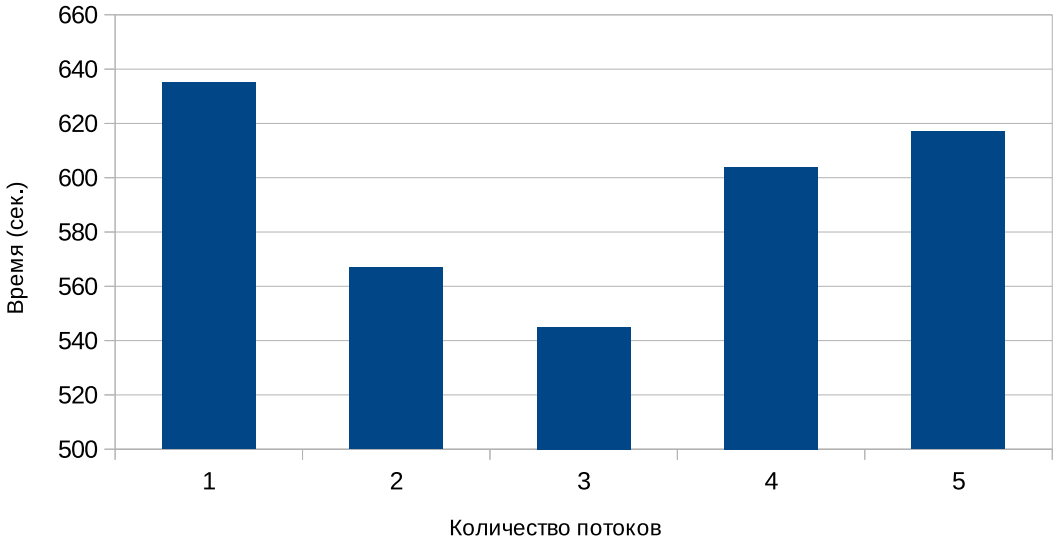

Рис. 12: Сравнение производительности предложенного решения при запуске на нескольких потоках

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. **??**.

Замеры, результаты которых представлены на рис. **??** и рис. **??** 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

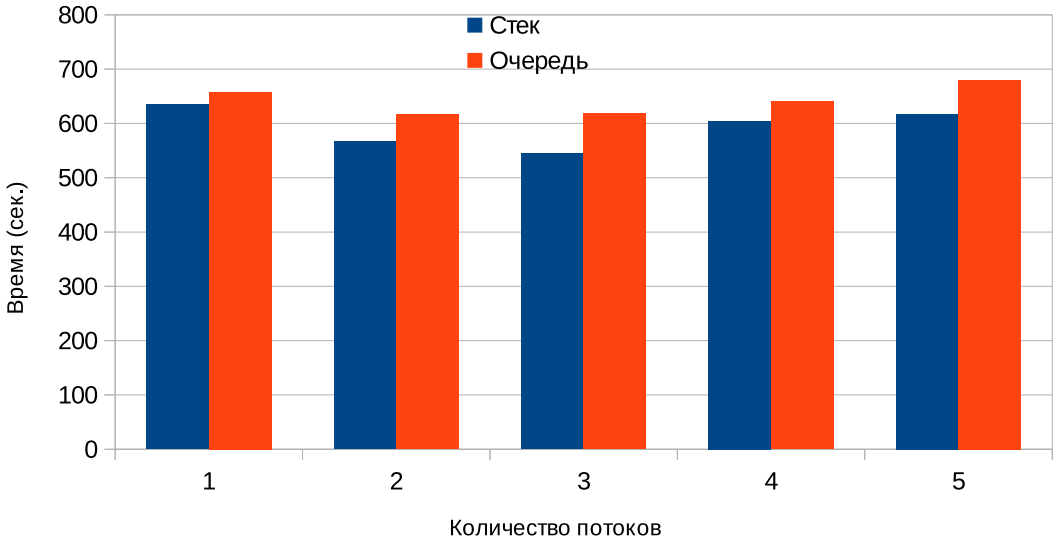

Рис. 13: Сравнение производительности предложенного решения замене стека на очередь для хранения дескрипторов

Из рис. **??** видно, что максимальная производительность наблюдается при использовании двух потоков. Однако 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. **??**. Максимальная производительность достигается при использовании трёх потоков и прирост производительность составляет 14.2%. На рис. **??** представлено сравнение производительности до и после модификации.

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.

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

Листинг 9: Код для запуска предложенного решения в кластере

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 и всё, что необходимо для реализации обработки массива графов о кластере, это "завернуть" её вызов в окружение `cloud` следующим образом.

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

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

5. Эксперименты

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

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 **??**), 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. **??**.

$$s \rightarrow s\ s\ s \mid s\ s \mid B$$

Листинг 11: Грамматика G_5

Следующей рассматриваемой грамматикой является неоднозначная грамматика правильных скобочных последовательностей G_6 (листинг

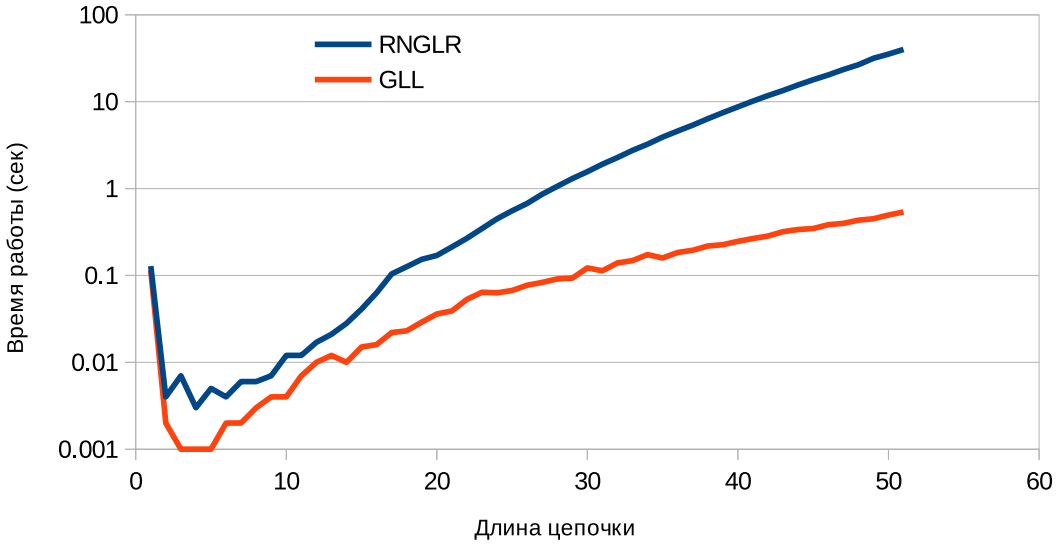

Рис. 14: Сравнение времени работы решений на основе алгоритмов GLL и RNGLR для грамматики G_5

??). Данная грамматика представляет интерес с практический точки зрения, так как она близка к шаблонам для поиска, используемым в задачах биоинформатики.

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

Листинг 12: Грамматика G_6

Результаты измерений представлены на рис. **??**. 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 [**?**] . Входные графы строились с помощью последовательной конкатенации блоков с параллельными путями, которые соответствуют использованию операторов ветвления при построении выражения. Пример входного графа приведён на рис. **??**.

Результаты измерений приведены на рис. **??** 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 оптимизировал-

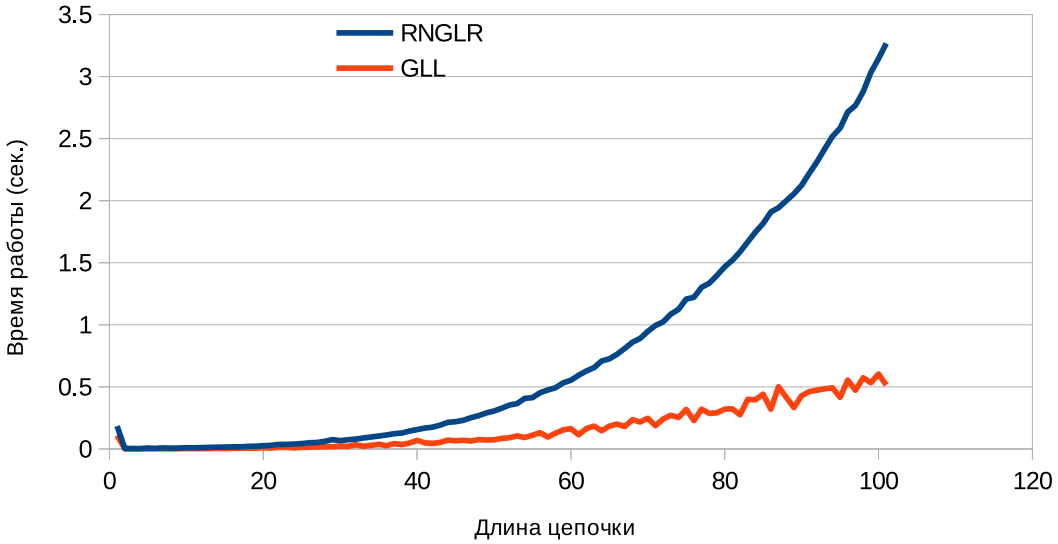

Рис. 15: Сравнение времени работы решений на основе алгоритмов GLL и RNGLR для грамматики G_6

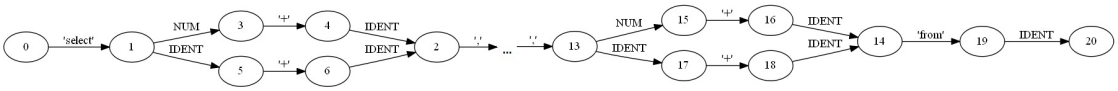

Рис. 16: Структура графа для экспериментов на грамматике T-SQL

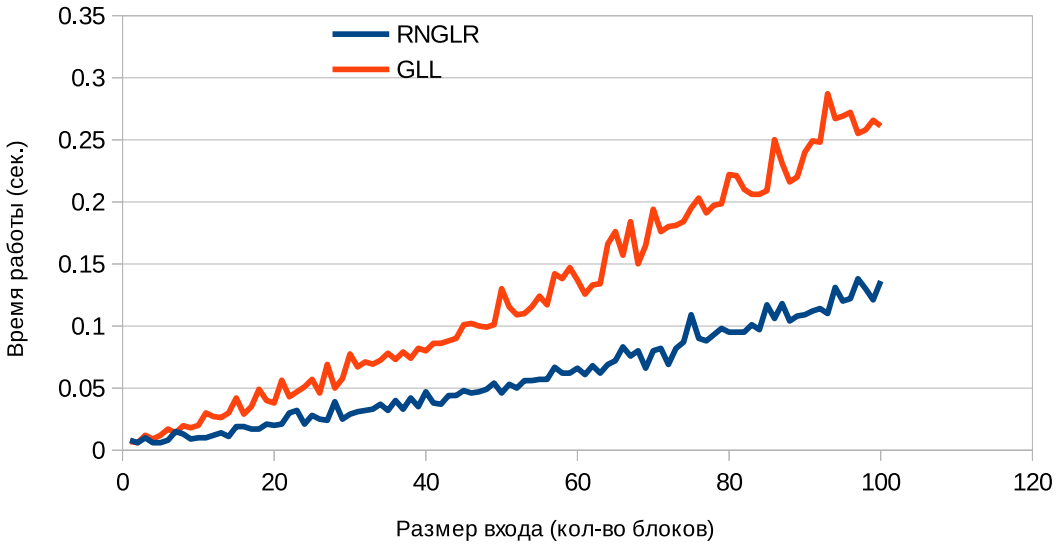

Рис. 17: Сравнение времени работы решений на основе алгоритмов GLL и RNGLR для грамматики T-SQL

ся. Техническая доработка GLL может позволить улучшить его производительность.

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 **??**. Данная грамматика является сильно неоднозначной. В качестве входа была выбрана последовательность из тестов инструмента Infernal [**?**] 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.

```
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?

a_5_8 : any*[5..8]
a_1_3 : any*[1..3]

folded: stem<(a_1_3 stem<any*[7..10]>
              a_1_3 stem<a_5_8>
              any*[3..6]
              stem<a_5_8>)>
```

Листинг 13: Пример грамматики для описания транспортной РНК

График зависимости времени работы от длины цепочки приведён на рис. **??**. Для RNGLR приведены первые 10 замеров, так как дальнейшие измерения для алгоритма на основе RNGLR было решено прекратить

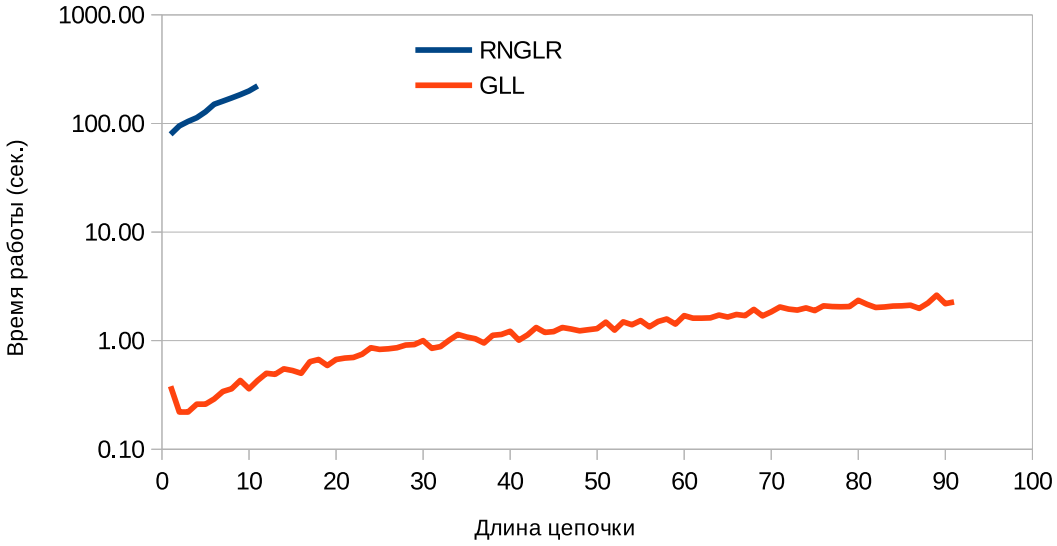

Рис. 18: Сравнение времени работы решений на основе алгоритмов GLL и RNGLR для грамматики на листинге **??**

из-за его низкой производительности. Приведённые результаты показывают, что реализованный в рамках данной работы алгоритм более 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 доступен в репозитории YaccConstructor [**?**], 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, необходи-

мо 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 **??**: правило `stem_3_5<s>` описывает `stem` высотой от 3 до 5 пар.

```
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
stem_3_5<s>: stem <s> & (any*[3..5] s any*[3..5])
```

Листинг 14: Пример конъюнктивной грамматики для описания stem-ов фиксированной высоты

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