Санкт-Петербургский Государственный Университет

Программная инженерия

Горохов Артем Владимирович

Поддержка расширенных контекстно-свободных грамматик в алгоритме синтаксического анализа Generalised LL

Выпуская квалификационная работа

Научный руководитель:
к. ф. -м. н., доц. Григорьев С. В.

Рецензент:
СУИ НИУ ИТМО, программист Авдюхин Д.А.

Санкт-Петербург
2017

SAINT PETERSBURG UNIVERSITY

Software engineering

Artem Gorokhov

Support of extended context-free grammars in Generalised LL parsing algorithm

Graduation Thesis

Scientific supervisor:
associate professor Semyon Grigorev

Reviewer:
ITMO University, programmer Dmitrii Avdiukhin

Saint Petersburg
2017

Оглавление

Введение

Общеупотребимый способ описания синтаксиса языков программирования — расширенные контекстно-свободные грамматики. Примером могут служить спецификации языков C, $C++$, $Java$ и т.д. С одной стороны, эта форма проста для понимания людей, с другой, достаточно формальна и допускает автоматизированное создание синтаксических анализаторов.

Существуют различные инструменты, такие как ANTLR [1], Bison [5], позволяющие для грамматики языка создать синтаксический анализатор для текстов, созданных на этом языке. Проблема заключается в том, что эти инструменты сначала преобразуют грамматику к форме Бэкуса-Наура и только затем строят синтаксический анализатор.

Существуют работы, описывающие синтаксический анализ с помощью расширенных контекстно-свободных грамматик (Extended Context-Free Grammar, ECFG) [6], [14], но практические средства, основанные на данных работах, не были созданы. Кроме того, подходы, описанные в данных исследованиях, поддерживают лишь подклассы контекстно-свободных языков.

Алгоритмы обобщённого синтаксического анализа, например Generalised LL [22], способны использовать контекстно-свободные грамматики, описывающие произвольные контекстно-свободные языки. Но они так же не допускают использования ECFG без предварительного преобразования к форме Бэкуса-Наура.

Одно из возможных применений обобщённого синтаксического анализа — поиск генов и иных последовательностей в биологическом материале. Из материала извлекаются геномы содержащихся в нём организмов — последовательности над алфавитом $\{A; C; G; T\}$. Но для того чтобы снизить расходы на хранение этих последовательностей, по ним строится конечный автомат. В геноме существуют различные последовательности, позволяющие классифицировать организм, которые имеют некоторые общие свойства, которые можно описать контекстно-

свободной грамматикой [18], поэтому для их поиска можно применять синтаксический анализ. Синтаксический анализ конечных автоматов называют синтаксическим анализом регулярных множеств. Одной из целей данной работы является применение полученного GLL алгоритма в синтаксическом анализе регулярных множеств, в частности — метагеномных сборок.

На кафедре Системного Программирования СПбГУ, в рамках исследовательского проекта YaccConstructor [32], разрабатывается подход для поиска структур, заданных с помощью контекстно-свободной грамматики в метагеномных сборках, основанный на алгоритме Generalised LL. Предполагается, что синтаксический анализ для расширенных контекстно-свободных грамматик без преобразований даст значительный прирост производительности существующего подхода.

1. Постановка задачи

Целью данной работы является модификация алгоритма Generalised LL, для обработки расширенных контекстно-свободных грамматик (ECFG) и проверка того, как полученный алгоритм влияет на производительность поиска структур, заданных с помощью контекстно-свободной грамматики в метагеномных сборках. Для её достижения были поставлены следующие задачи.

- Выбрать или разработать подходящее представление ECFG для использования в синтаксическом анализе.
- Спроектировать структуру данных для представления леса разбора для ECFG.
- Разработать алгоритм на основе Generalised LL, строящий лес разбора для ECFG.
- Разработать механизм анализа регулярных множеств в алгоритме построения леса разбора для ECFG.
- Реализовать разработанный алгоритм в рамках проекта YaccConstructor.
- Провести экспериментальное сравнение реализованного алгоритма с существующим в проекте YaccConstructor при анализе метагеномной сборки.

2. Обзор

2.1. Расширенные контекстно-свободные грамматики

Расширенная форма Бэкуса-Наура (EBNF) [29] является метасинтаксисом для представления контекстно-свободных грамматик. В дополнение к конструкциям, используемым в форме Бэкуса-Наура, в ней используется следующие конструкции: альтернатива |, необязательные символы [...], повторение {...} и группировка (...).

Форма EBNF широко используется для спецификации грамматики в технической документации ввиду того, что её выразительная сила делает спецификацию синтаксиса более компактной и удобной для чтения. Поскольку документация является одним из основных источников информации о языке для разработчиков синтаксических анализаторов, полезно иметь генератор анализаторов, который поддерживает грамматики в EBNF. Заметим, что EBNF является лишь стандартизированной формой для *расширенных контекстно-свободных грамматик* [12].

***Определение* 1.** *Расширенная контекстно-свободная грамматика (ECFG) [12] — это кортеж* (N, Σ, P, S)*, где* N *и* Σ *конечные множества нетерминалов и терминалов соответственно,* $S \in N$ *является стартовым символом, а* P *(продукция) является отображением из* N *в регулярное выражение над алфавитом* $N \cup \Sigma$*.*

ECFG широко используется в качестве входного формата для генераторов синтаксических анализаторов, но классические алгоритмы синтаксического анализа часто требуют контекстно-свободную форму (CFG), в продукциях которой допускаются лишь последовательности из терминалов и нетерминалов. Таким образом для работы генераторов анализаторов требуется преобразование в CFG. Возможно преобразование ECFG в CFG [11], но оно приводит к увеличению размера грамматики и изменению её структуры: при трансформации добавляются новые нетерминалы. В результате синтаксический анализатор строит

дерево вывода относительно преобразованной грамматики, и разработчику языка сложнее отлаживать грамматику и использовать результат синтаксического анализа. Кроме того, увеличение размера грамматики отрицательно сказывается на производительности анализа.

Существует широкий спектр методов анализа и алгоритмов [4, 7, 9, 10, 11, 12, 15, 17], которые способны обрабатывать ECFG. Детальный обзор результатов и задач в области обработки ECFG представлены в статье “Towards a Taxonomy for ECFG and RRPG Parsing” [12]. Следует отметить, что большинство алгоритмов основано на классических методах LL [10, 4, 8] — нисходящий анализ и LR [17, 15, 7] — восходящий анализ, но они работают только с ограниченными подклассами расширенных контекстно-свободных грамматик — LL(k), LR(k). Таким образом, нет решения для обработки произвольных (в том числе неоднозначных) ECFG.

Алгоритмы синтаксического анализа на основе LL проще для восприятия, чем основанные на LR, и могут обеспечить лучшую диагностику ошибок. В настоящее время LL(1) представляется наиболее практичным алгоритмом. К сожалению, некоторые языки не являются LL(k) (для любого k) и не могут быть использованы в LL(k) анализаторах, другие проблемы для инструментов на основе LL — леворекурсивные грамматики и неоднозначности в грамматике, которые, вместе с предыдущим недостатком, усложняют создание синтаксических анализаторов. Алгоритм Generalised LL, предложенный в [22], решает все эти проблемы: он обрабатывает произвольные CFG, в том числе неоднозначные и леворекурсивные. В худшем случае временная и пространственная сложность GLL зависит кубически от размера входа. А для LL(1) грамматик, он демонстрирует линейную временную и пространственную сложность.

Чтобы увеличить производительность Generalised LL-алгорима, была предложена поддержка лево-факторизованных грамматик в нём [24]. Из описания GLL-алгоритма ясно, что для уменьшения времени анализа и количества используемой памяти можно снизить количество дескрипторов для обработки. Один из путей для достижения этого —

$S ::= a\ a\ b\ c\ d \mid a\ a\ c\ d \mid a\ a\ c\ e \mid a\ a$

(a) Исходная грамматика G_0

$S ::= a\ a\ (b\ c\ d \mid c\ (d \mid e) \mid \varepsilon\ )$

(b) Факторизованная грамматика G'_0

Рис. 1: Пример факторизации грамматики

уменьшение размера грамматики (снижение количества различных позиций в ней). Этого можно достичь факторизацией грамматики. Пример факторизации показан на рис. 1: из грамматики G_0 в процессе факторизации получается грамматика G'_0. Этот пример рассмотрен в работе [24], и доказано, что для некоторых грамматик факторизация существенно увеличивает производительность алгоритма GLL. В данной работе эта идея развита в поддержку расширенных контекстно-свободных грамматик.

2.2. Стек, структурированный в виде графа

В процессе синтаксического анализа используется стек, позволяющий отслеживать историю разбора нетерминалов. Но, при встречи неоднозначности в грамматике, для каждого варианта разбора должен быть создан новый стек, основанный на текущем. Такое представление неэффективно, так как эти стеки имеют общие узлы, которые могут быть переиспользованы. Поэтому был предложен стек, структурированный в виде графа (Graph Structured Stack, GSS). GSS комбинирует в себе все варианты стеков.

В работе [19] используется стек, структура которого определена в [2]. В его узлах хранятся нетерминалы и начальные позиции их разбора во входе. На рёбрах, исходящих из узла с меткой (A, i), хранятся позиции в грамматике с которых нужно продолжать разбор после разбора нетерминала A, а также корень построенного SPPF до начала разбора нетерминала A.

2.3. Сжатое представление леса разбора

Результатом работы синтаксического анализатора является структурное представление входа: дерево разбора. Если возможно несколь-

ко выводов входа, строится несколько деревьев: для каждого варианта разбора. Например, для грамматики G_0 (рис. 2) и входа ccc будут построены 2 дерева, показанные на рис. 3a.

Для некоторых грамматик количество деревьев может экспоненциально зависеть от размера входа. Чтобы снизить расходы для хранения и обработки всех деревьев, используется структура данных Shared Packed Parse Forest (SPPF). Будем использовать бинаризованную версию SPPF, предложенную в [25], для уменьшения потребления памяти и достижения кубической наихудшей временной и пространственной сложности. Бинаризованный SPPF может использоваться в GLL [23] и содержит следующие типы узлов (i и j — это начало и конец выведенной подстроки для данного узла).

- Упакованные узлы вида (M, k), где M — позиция в грамматике, k — начало выведенной подстроки правого ребёнка. У упакованных узлов обязательно существует правый ребёнок — символьный узел, и опциональный левый — символьный или промежуточный узел.

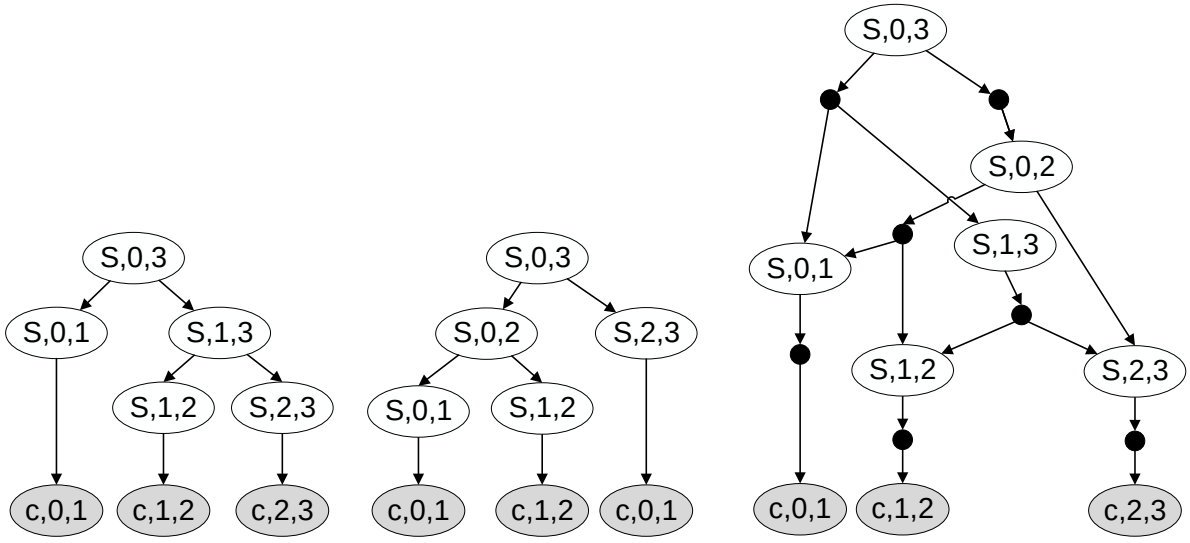
- Символьный узел помечен (X, i, j) где X — терминал или нетерминал. Терминальные символьные узлы — листья. Нетерминальные символьные узлы могут иметь несколько упакованных детей.

- Промежуточные узлы помечены (M, i, j), где M — позиция в грамматике, могут иметь несколько упакованных детей, каждый из которых представляет различные варианты разбора.

Дети символьных и промежуточных узлов — упакованные. Различные упакованные дети — различные варианты поддеревьев. Если у узла или его потомков более одного упакованного ребёнка, то для него было построено несколько вариантов разбора для строки. Промежуточные и упакованные узлы необходимы для бинаризации SPPF. Так, деревья, представленные на рис. 3a, объединяются в SPPF показанный на рис. 3b.

$$S ::= S\ S \mid c$$

Рис. 2: Грамматика G_0



(a) Возможные деревья вывода (b) SPPF

Рис. 3: Пример для входа *ccc* и грамматики G_0

2.4. Алгоритм Generalised LL

Цель обобщенных алгоритмов синтаксического анализа — обеспечить создание синтаксических анализаторов для произвольных контекстно-свободных грамматик. Алгоритм Generalised LL (GLL) [22] включает в себя свойства классических LL алгоритмов: он проще в понимании и обеспечивает более хорошую диагностику ошибок, чем обобщенные LR алгоритмы. Кроме того, опыт показывает, что решения на основе GLR более сложны, чем основанные на GLL, что согласуется с наблюдением в [11], что синтаксические анализаторы ECFG на основе LR очень сложны. Таким образом, в качестве основы для решения был выбран GLL алгоритм.

Идея алгоритма GLL основана на обработке так называемых дескрипторов, которые могут однозначно определить состояние процесса синтаксического анализа. Дескриптор представляет собой кортеж (L, i, T, S), где:

- L указатель на позицию в грамматике вида $(S \to \alpha \cdot \beta)$;

- i — позиция во входе;
- T — корень построенного леса разбора;
- S — текущий узел стека (GSS) [2].

GLL двигается одновременно по входу и грамматике, создавая множество дескрипторов в случае неоднозначности и использует очередь для управления обработкой дескрипторов. В начальном состоянии существует только один дескриптор, который состоит из начальной позиции в грамматике ($S \to \cdot\beta$), во входе (i = 0), фиктивного узла дерева (\$) и дна стека. На каждом шаге алгоритм извлекает дескриптор из очереди и действует в зависимости от грамматики и входа. Если имеется неоднозначность, то алгоритм помещает в очередь дескрипторы для всех возможных случаев, чтобы обработать их позже. Для достижения кубической временной сложности важно помещать в очередь только дескрипторы, которые не создавались ранее. Для того чтобы решить добавлять дескриптор или нет используется глобальное хранилище всех созданных дескрипторов. Существует подход на основе таблиц [19] для реализации GLL, который генерирует только таблицы для данной грамматики вместо полного кода синтаксического анализатора. Эта идея похожа на алгоритм в оригинальной статье и использует те же техники построения леса разбора и обработки стека. Псевдокод, иллюстрирующий этот подход, можно найти в приложении. Обратите внимание, что в приложении и далее в псевдокод не включена проверка для множеств first/follow.

2.5. Анализ метагеномных сборок

В практических задачах часто могут возникать ситуации, когда необходимо проводить синтаксический анализ сразу нескольких строк. Существует подход, при котором для строк подлежащим анализу строится конечный автомат, порождающий эти строки. И тогда задача сводится к анализу регулярного языка относительно грамматики.

Такой подход называют синтаксическим анализом регулярных множеств. Одно из применений этого подхода — биоинформатика.

Биоинформатика включает в себя множество задач, решения которых необходимы в биологических исследованиях. Одна из них — задача идентификации организмов в биологическом материале. В результате оцифровывания материала получают геномы организмов — строки над алфавитом $\{A; C; G; T\}$. Извлечённые из материала последовательности объединяются в метагеномную сборку — конечный автомат, пути в котором задают полученные гены.

Существует множество подходов к анализу и идентификации образцов. Один из них — поиск и сравнение участков таких структур как 16s рРНК, тРНК, так как по ним можно достаточно точно классифицировать организм, которому они принадлежат. Существуют такие инструменты, как REAGO [20] и HMMER [28], использующие скрытые цепи Маркова для поиска, Infernal [16], использующий ковариационные модели. Но они работают лишь с линейными цепочками генома — не объединёнными в конечный автомат, такое представление требует большого количества памяти и неэффективно. С другой стороны, инструмент Xander [30] использует композицию скрытых моделей Маркова и метагеномной сборки, представленной в виде автомата. Изъян данного инструмента в существенно более низкой точности результата в сравнения с инструментами, использующими ковариационные модели.

Другой подход разрабатывается на кафедре Системного Программирования СПбГУ. Поиск производится непосредственно в метагеномной сборке по таким структурам как тРНК, 16s рРНК. Эти структуры имеют некоторые общие свойства в строении, которые могут быть описаны контекстно-свободной грамматикой [18]. Таким образом, можно использовать синтаксический анализ регулярных множеств для поиска структур. Этот подход описан в работе [19], основан на алгоритме синтаксического анализа Generalised LL и был реализован в рамках проекта YaccConstructor.

2.6. Проект YaccConstructor

YaccConstructor [32] — это исследовательский проект кафедры Системного Программирования СПбГУ и лаборатории языковых инструментов JetBrains. Проект направлен на изучение алгоритмов синтаксического и лексического анализа и занимается разработкой инструмента YaccConstructor, предоставляющего платформу для создания и изучения новых алгоритмов. Инструмент имеет модульную архитектуру и включает в себя язык описания грамматик Yard, который поддерживает расширенные контекстно-свободные грамматики. Кроме того, в инструменте реализован генератор синтаксических анализаторов на основе Generalised LL алгоритма. Инструмент разработан на платформе $.NET$, на языке программирования $F\#$.

3. Представление ECFG

Чтобы облегчить задание грамматики в форме ECFG для синтаксического анализатора будем использовать рекурсивный автомат (Recursive Automaton (RA) [26] для представления ECFG. Будем использовать следующее определение RA.

Определение 2. *Рекурсивный автомат R это кортеж $(\Sigma, Q, S, F, \delta)$, где Σ — конечное множество терминалов, Q - конечное множество состояний, $S \in Q$ — начальное состояние, $F \subseteq Q$ — множество конечных состояний, $\delta : Q \times (\Sigma \cup Q) \to Q$ — функция перехода.*

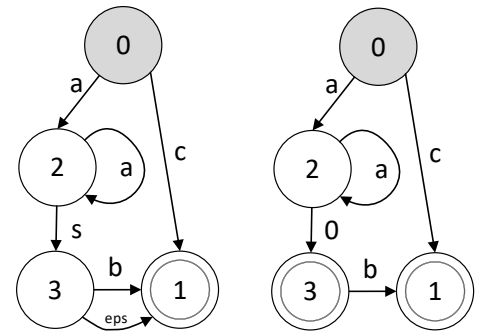
В рамках этой работы единственное различие между рекурсивным автоматом и общеизвестным конечным автоматом (FSA) состоит в том, что переходы в RA обозначаются либо терминалом (Σ), либо состоянием автомата (Q). Далее в этой работе будем называть переходы по элементам из Q *нетерминальными переходами*, а по терминалам — *терминальными переходами*. Переход по нетерминалу в состояние q подразумевают построение вывода для некоторой подстроки начиная с текущей позиции вывода по этому нетерминалу и последующий разбор оставшейся подстроки начиная с состояния q.

Заметим, что позиции грамматики эквивалентны состояниям автомата, которые строятся из правых частей продукций. Правые части продукций ECFG являются регулярными выражениями над объединенным алфавитом терминалов и нетерминалов. Итак, наша цель — построить RA с минимальным числом состояний для заданной ECFG, что можно сделать следующими шагами.

- Построить конечный автомат, используя метод Томпсона [27] для правых частей продукций.
- Создать карту из каждого нетерминала в соответствующее начальное состояние автомата. Эта карта должна оставаться консистентной на протяжение всех следующих шагов.
- Преобразовать автоматы из предыдущего шага в детерминированные без ε-переходов используя алгоритм, описанный в [3].

$S ::= a^+ S\ b? \mid c$

(a) Грамматика G_1



(b) Конечный автомат для G_1

(c) Рекурсивный автомат R_1 для G_1

Рис. 4: Преобразование грамматики в рекурсивный автомат

- Минимизировать детерминированный автомат, используя, например, алгоритм Джона Хопкрофта [13].

- Заменить нетерминальные переходы переходами по, стартовым состояниям автоматов, соответствующим данным нетерминалам, используя карту M. Результат этого шага — искомый рекурсивный автомат. Также используем карту M для определения функции $\Delta : Q \to N$ где N — имя нетерминала.

Пример преобразования ECFG в RA представлен на рис. 4, где состояние 0 — начальное состояние результирующего RA.

4. Лес разбора для ECFG

Результатом процесса синтаксического анализа является структурное представление входа — дерево или лес разбора в случае нескольких вариантов деревьев. Для начала, определим дерево вывода для рекурсивного автомата: это дерево, корень которого помечен начальным состоянием, листовые узлы помечены терминалом или ε, а внутренние узлы помечены нетерминалами N и их дети образуют последовательность меток в пути в автомате, который начинается в состоянии q_i, где $\Delta(q_i) = N$. Более формально:

***Определение* 3.** *Дерево вывода последовательности α для рекурсивного автомата $R = (\Sigma, Q, S, F, \delta)$ это дерево со следующими свойствами:*

- *корень помечен $\Delta(S)$;*
- *листья —теминалы $a \in (\Sigma \cup \varepsilon)$;*
- *остальные узлы — нетерминалы $A \in \Delta(Q)$;*
- *у узла с меткой $N_i = \Delta(q_i)$ существует:*
  - *дети $l_0 \dots l_n (l_i \in \Sigma \cup \Delta(Q))$ тогда и только тогда, когда существует путь p в R, $p = q_i \xrightarrow{l_0} q_{i+1} \xrightarrow{l_1} \dots \xrightarrow{l_n} q_m$, где $q_m \in F$, $l_i = \begin{cases} k_i, & \text{if } k_i \in \Sigma, \\ \Delta(k_i), & \text{if } k_i \in Q, \end{cases}$*
  - *только один ребенок помеченный ε тогда и только тогда, когда $q_i \in F$.*

Для произвольных грамматик RA может быть неоднозначным с точки зрения допустимых путей, и, как результат, можно получить несколько деревьев разбора для одной входной строки. Shared Packed Parse Forest (SPPF) [21] может использоваться как компактное представление всех возможных деревьев разбора. Будем использовать бинаризованную версию SPPF, предложенную в [25], для уменьшения потребления памяти и достижения кубической наихудшей временной и

пространственной сложности. Бинаризованный SPPF может использоваться в GLL [23] и содержит следующие типы узлов (здесь i и j называют правый и левый extent — начало и конец выведенной подстроки во входной строке):

- упакованные узлы вида (S, k), где S состояние автомата, k — начало выведенной подстроки правого ребёнка; у упакованных узлов обязательно существует правый ребёнок — символьный узел, и опциональный левый — символьный или промежуточный узел;
- символьный узел помечен (X, i, j) где $X \in \Sigma \cup \Delta(Q) \cup \{\varepsilon\}$; терминальные символьные узлы ($X \in \Sigma \cup \{\varepsilon\}$) — листья; нетерминальные символьные узлы ($X \in \Delta(Q)$) могут иметь несколько упакованных детей;
- промежуточные узлы помечены (S, i, j), где S состояние в автомате, могут иметь несколько упакованных детей.

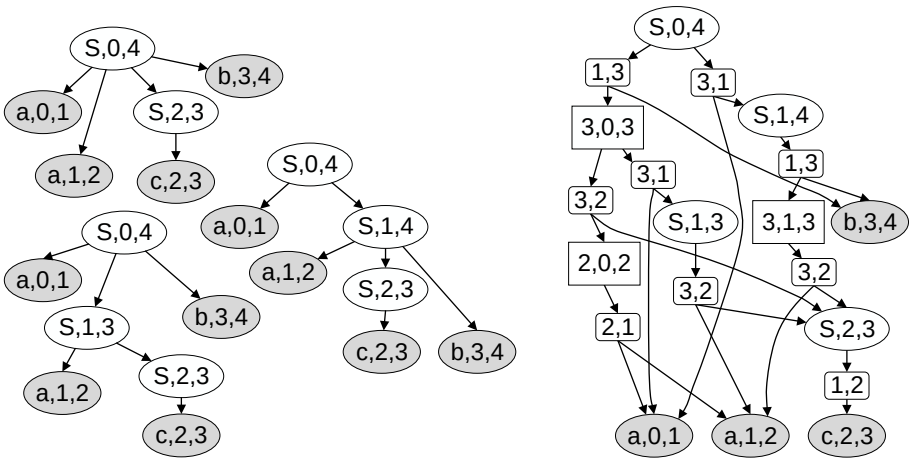
Опишем модификации исходных функций построения SPPF. Функция **getNodeT**(x, i), которая создает терминальные узлы, повторно используется без каких-либо модификаций из базового алгоритма. Чтобы обрабатывать недетерминизм в состояниях, определим функцию **getNodes**, которая проверяет, является ли следующее состояние RA финальным и в этом случае строит нетерминальный узел в дополнение к промежуточному. Она использует изменённую функцию **getNodeP**: вместо позиции в грамматики он принимает в качестве входных данных отдельно состояние RA и символ для нового узла SPPF: текущий нетерминал или следующее состояние RA.

function getNodes(S, A, w, z)
  if (S is final state) **then**
    $x \leftarrow$ **getNodeP**(S, A, w, z)
  else $x \leftarrow \$$
  if ($w = \$$)& not (z is nonterminal node and its extents are equal) **then**

```
        y ← z
    else y ← getNodeP(S, S, w, z)
    return (y, x)

function getNodeP(S, L, w, z)
    (_, k, i) ← z
    if (w ≠ $) then
        (_, j, k) ← w
        y ← find or create SPPF node labelled (L, j, i)
        if (∄ child of y labelled (S, k)) then
            y′ ← new packedNode(S, k)
            y′.addLeftChild(w)
            y′.addRightChild(z)
            y.addChild(y′)
    else
        y ← find or create SPPF node labelled (L, k, i)
        if (∄ child of y labelled (S, k)) then
            y′ ← new packedNode(S, k)
            y′.addRightChild(z)
            y.addChild(y′)
    return y
```

Рассмотрим пример SPPF для ECFG G_1, показанные на рис. 4a. Эта грамматика содержит конструкции (условное вхождение (?) и повторение (+)), которые должны быть преобразованы с использованием дополнительных нетерминалов для создания обычного GLL-анализатора. Предложенный генератор строит рекурсивный автомат R_1 (рис. 4c) и анализатор для него. Возможные деревья ввода последовательности *aacb* показаны на рис. 5a. SPPF, созданный синтаксическим анализатором (рис. 5b), содержит в себе все три дерева.



(a) Возможные деревья вывода (b) SPPF

Рис. 5: Пример для входа $aacb$ и автомата R_1

5. Алгоритм построения леса разбора для ECFG

В этом разделе описываются изменения в управляющих функциях базового алгоритма Generalised LL, необходимые для обработки ECFG. Основной цикл аналогичен базовому GLL: на каждом шаге основная функция **parse** извлекает из очереди дескриптор R, подлежащий обработке. Пусть текущий дескриптор – кортеж (C_S, C_U, i, C_N), где C_S — состояние RA, C_U — узел GSS, i — позицию во входной строке ω, C_N — узел SPPF. В ходе обработки дескриптора могут возникнуть следующие не исключающие друг друга ситуации.

- C_S — финальное состояние. Это возможно только если C_S — стартовое состояние текущего нетерминала. Следует построить нетерминальный узел с ребёнком (ε, i, i) и вызвать функцию **pop**, так как разбор нетерминала окончен.

- Существует терминальный переход $C_S \xrightarrow{\omega.[i]} q$. Во-первых, построить терминальный узел $t = (\omega.[i], i, i+1)$, далее вызвать функцию **getNodes** чтобы построить родителя для C_N и t. Функция **getNodes** возвращает кортеж (y, N), где N — опциональный нетерминальный узел. Создать дескриптор $(q, C_U, i+1, y)$ и, если в q ведёт несколько переходов, вызвать

функцию **add** для этого дескриптора. Иначе поместить его в очередь вне зависимости от того был ли он создан до этого. Если $N \neq \$$, вызвать функцию **pop** для этого узла, состояния q и позиции во входе $i + 1$.

- Существуют нетерминальные переходы из C_S. Это значит, что следует начать разбор нового нетерминала, поэтому должен быть создан новый узел GSS, если такового ещё нет. Для этого нужно вызвать функцию **create** для каждого такого перехода. Она осуществляет необходимые операции с GSS и проверяет наличие узла GSS для текущих нетерминала и позиции во входе.

Псевдокод для необходимых функций представлен ниже:

Функция **add** помещает в очередь дескриптор, если он не был создан до этого; эта функция не изменилась.

```
function CREATE(S_call, S_next, u, i, w)
    A ← Δ(S_call)
    if (∃ GSS node labeled (A, i)) then
        v ← GSS node labeled (A, i)
        if (there is no GSS edge from v to u labeled (S_next, w)) then
            add GSS edge from v to u labeled (S_next, w)
            for ((v, z) ∈ P) do
                (y, N) ← getNodes(S_next, u.nonterm, w, z)
                (_, _, h) ← y
                add(S_next, u, h, y)
                if N ≠ $ then
                    (_, _, h) ← N; pop(u, h, N)
    else
        v ← new GSS node labeled (A, i)
        create GSS edge from v to u labeled (S_next, w)
        add(S_call, v, i, $)
    return v

function POP(u, i, z)
    if ((u, z) ∉ P) then
```

```
        P.add(u, z)
        for all GSS edges (u, S, w, v) do
            (y, N) ← getNodes(S, v.nonterm, w, z)
            add(S, v, i, y)
            if N ≠ $ then pop(v, i, N)
function PARSE
    GSSroot ← newGSSnode(StartNonterminal, 0)
    R.enqueue(StartState, GSSroot, 0, $)
    while R ≠ ∅ do
        (C_S, C_U, i, C_N) ← R.dequeue()
        if (C_N = $) and (C_S is final state) then
            eps ← getNodeT(ε, i)
            (_, N) ← getNodes(C_S, C_U.nonterm, $, eps)
            pop(C_U, i, N)
        for each transition(C_S, label, S_next) do
            switch label do
                case Terminal(x) where (x = input[i])
                    T ← getNodeT(x, i)
                    (y, N) ← getNodes(S_next, C_U.nonterm, C_N, T)
                    if N ≠ $ then pop(C_U, i + 1, N)
                    if S_next have multiple ingoing transitions then
                        add(S_next, C_U, i + 1, y)
                    else
                        R.enqueue(S_next, C_U, i + 1, y)
                case Nonterminal(S_call)
                    create(S_call, S_next, C_U, i, C_N)
    if SPPF node (StartNonterminal, 0, input.length) exists then
        return this node
    else report failure
```

6. Синтаксический анализ регулярных множеств

Для работы с метагеномными сборками необходимо осуществить поддержку синтаксического анализа регулярных множеств. При работе с автоматом в качестве входных данных необходимо обрабатывать все переходы из текущей позиции (состояния) в автомате. Так, основная функция приобретает следующий вид:

function parseRegularSet
  $GSSroot \leftarrow newGSSnode(StartNonterminal, input.StartState)$
  $R.enqueue(StartState, GSSroot, input.StartState, \$)$
  while $R \neq \varnothing$ **do**
    $(C_S, C_U, i, C_N) \leftarrow R.dequeue()$
    if $(C_N = \$)$ and (C_S is final state) **then**
      $eps \leftarrow$ **getNodeT**(ε, i)
      $(\_, N) \leftarrow$ **getNodes**$(C_S, C_U.nonterm, \$, eps)$
      pop(C_U, i, N)
    for each $transition(C_S, label, S_{next})$ **do**
      switch $label$ **do**
        case $Terminal(x)$
          for each (input[i] $\xrightarrow{x}$ input[k]) **do**
            $T \leftarrow$ **getNodeT**(x, i)
            $(y, N) \leftarrow$ **getNodes**$(S_{next}, C_U.nonterm, C_N, T)$
            if $N \neq \$$ **then** **pop**(C_U, k, N)
            add(S_{next}, C_U, k, y)
        case $Nonterminal(S_{call})$
          create$(S_{call}, S_{next}, C_U, i, C_N)$
  if SPPF node $(StartNonterminal, input.StartState, \_)$ exists **then**
    return this node
  else report failure

Позициями во входе для автомата становятся номера состояний и обрабатываются все исходящие переходы во входе. Кроме того, Функция

add вызывается при обработке терминального перехода всегда, чтобы поддержать возможные циклы во входе. Например, для грамматики $S ::= a*$ и входного автомата на рис. 6 дескриптор будет создаваться бесконечно, если не добавить его в множество созданных, и алгоритм не остановится. Данное изменение не меняет теоретическую сложность алгоритма, но может сказаться на производительности в худшую сторону. Поэтому этот подход можно применять лишь только в случае присутствия циклов во ходе, иначе вызывать функцию **add** только при наличии нескольких входящих переходов в текущее состояние.

Рис. 6: Пример входа для грамматики $S ::= a^*$.

7. Реализация

Описанный алгоритм реализован в проекте YaccConstructor. На вход генератору поступает структурное представление грамматики, на основе которого генератор создаёт управляющие таблицы. Далее они и входные данные поступают на вход синтаксическому анализатору, который строит SPPF.

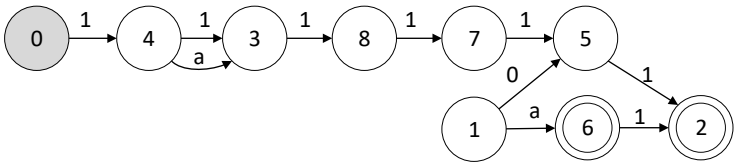
В проекте уже был реализован генератор анализаторов и интерпретатор на основе алгоритма GLL, они были заменены предложенными в этой работе с сохранением остальной инфраструктуры.

8. Эксперименты

В работе [24] было проведено экспериментальное сравнение алгоритма для факторизованных грамматик (Factorised GLL, FGLL) и базового GLL-алгоритма, продемонстрировавшее, что FGLL показывает бо̀льшую производительность для грамматик в форме Бэкуса-Наура, которые могут быть факторизованы. Предполагается, что предложенная в данной работе версия алгоритма продемонстрирует бо̀льшую производительность, чем FGLL, для грамматик, имеющих эквивалентные позиции для алгоритма минимизации автомата, но различные после факторизации. Автомат, построенный для грамматики, в которой есть эквивалентные позиции, для которых алгоритм создаёт большое количество дескрипторов, объединит данные позиции, сократив те самым множество создаваемых дескрипторов, что в свою очередь увеличит производительность. Примером такой ситуации может служить грамматика G_2 (рис. 7a), так как она содержит длинные последовательности в альтернативах, которые не сливаются при факторизации, но эквивалентны для алгоритма минимизации автомата. Рекурсивный автомат построенный для этой грамматики показан на рис. 7b.

$$\begin{aligned} S &::= \ K\ (K\ K\ K\ K\ K \mid a\ K\ K\ K\ K) \\ K &::= \ S\ K \mid a\ K \mid a \end{aligned}$$

(a) Грамматика G_2

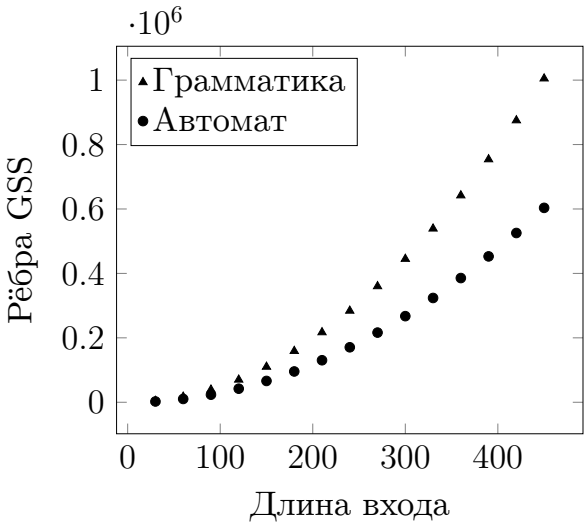


(b) RA для грамматики G_2

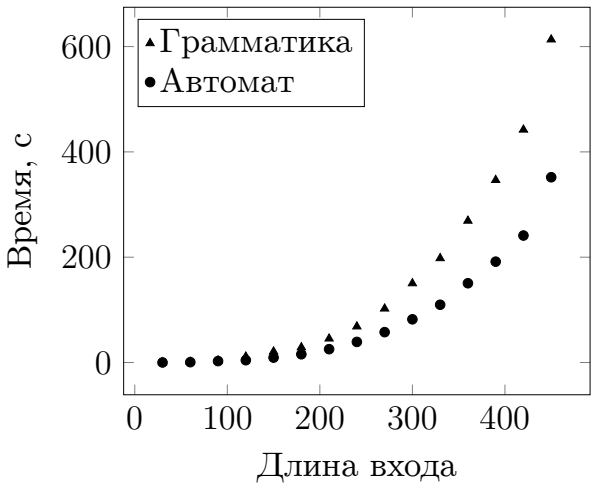
Рис. 7: Грамматика G_2 и RA для неё

Эксперименты проводились на входах различной длины, результаты приведены на рис. 8. Точные данные для входа a^{450} показаны в таблице 1.

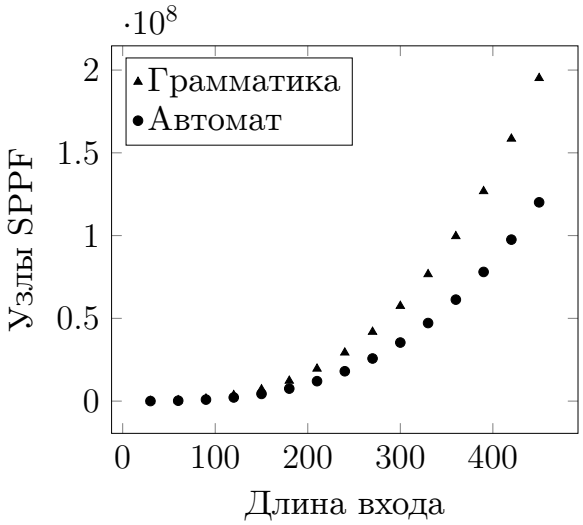
Тесты проводились на ПК со следующими характеристиками: Microsoft Windows 10 Pro x64, Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, 3601 Mhz, 4 Cores, 4 Logical Processors, 16 GB.



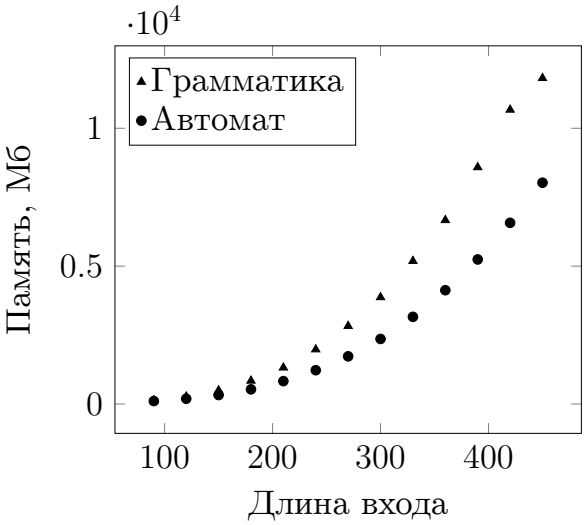
(a) Количество рёбер GSS.



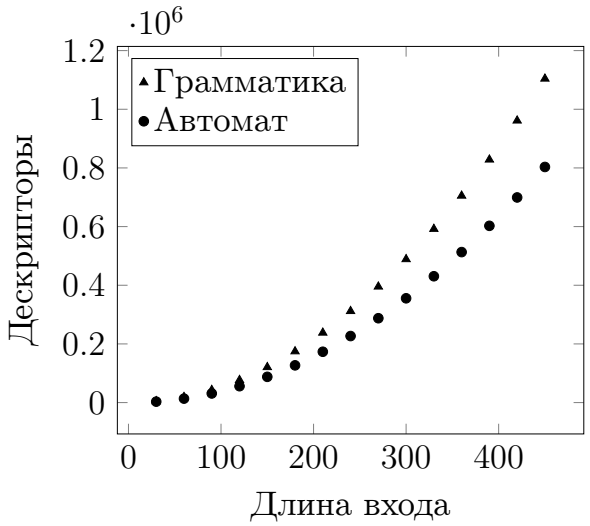
(b) Время работы.



(c) Количество узлов SPPF.



(d) Использование памяти



(e) Количество дескрипторов.

Рис. 8: Результаты экспериментов с грамматикой G_2.

	Время	Дескрипторы	Рёбра GSS	Узлы GSS	Узлы SPPF	Память, Мб
Фактор-ая грамматика	10 мин. 13 с.	1104116	1004882	902	195 млн.	11818
RA	5 мин. 51 с.	803281	603472	902	120 млн.	8026
Ratio	43%	28%	40 %	0 %	39 %	33 %

Таблица 1: Результаты экспериментов для входа a^{450}

Результаты данных экспериментов поддерживают предположение о том, что на некоторых грамматиках предложенный подход показывает результаты лучше, нежели анализатор, построенный для факторизованных грамматик. Для этого рекурсивного автомата анализатор создаёт меньше дескрипторов, чем для грамматики, так как цепочки нетерминалов K в альтернативах представлены единственным путём в автомате. Эта особенность ведёт к снижению количества узлов SPPF и размера GSS. В среднем, с грамматикой G_2 версия с минимизированными автоматами работает на 43% быстрее, использует на 28% меньше дескрипторов, на 40% меньше рёбер GSS, создаёт на 39% меньше узлов SPPF и использует на 33% меньше памяти.

Было проведено экспериментальное сравнение реализации разработанного алгоритма GLL с существующим в проекте YaccConstructor (основан на оригинальном алгоритме Generalised LL). Кроме того, было проведено сравнение производительности алгоритма GLL для факторизованных грамматики, реализованного в проекте YaccConstructor и описанного в данной работе в задаче поиска 16s рРНК в метагеномной сборке. Длинные рёбра сборки были предварительно отфильтрованы с помощью инструмента Infernal. В результате фильтрации сборка разбивается на компоненты, которые можно анализировать независимо друг от друга. Тем не менее, предложенный ранее алгоритм не позволяет обработать некоторые компоненты, поэтому сравнение проводилось на остальных: 10 компонент с 400-100 состояний и переходов и 1118 ком-

понент с менее чем 100 состояний и переходов. Результаты сравнения приведены в таблице 2 и показывают, что при работе с метагеномными сборками новый алгоритм, в среднем, использует на 65% меньше памяти и работает на 45% быстрее базового GLL. Сравнение с анализатором для факторизованной грамматики показывает на 4% меньшее использование памяти новым алгоритмом и на 10% меньшее время работы.

		GSS			
	Деск-ры	Рёбра	Узлы	Память	Время
GLL	802млн	414млн	339млн	20Гб	52 мин. 43 с.
Фактор-ая грамматика	382млн	187млн	134млн	7Гб	29 мин. 27 с.
RA	362млн	190млн	134млн	6,8Гб	26 мин. 34 с.

Таблица 2: Результаты экспериментов с метагеномной сборкой

Заключение

В рамках данной работы была разработана и реализована модификация алгоритма GLL, работающая с расширенными контекстно-свободными грамматиками. Показано, что полученный алгоритм повышает производительность поиска структур, заданных с помощью контекстно-свободной грамматики в метагеномных сборках. Более детально, были получены следующие результаты.

- В качестве подходящего представления ECFG выбраны рекурсивные конечные автоматы.
- Спроектирована структура данных для представления леса разбора для ECFG на основе сжатого леса разбора (SPPF).
- Разработан алгоритм на основе Generalised LL, строящий лес разбора для ECFG.
- Алгоритм реализован в рамках проекта YaccConstructor. Исходный код доступен в репозитории YaccConstructor [31], автор работал под учётной записью “gorohovart”.
- Проведено экспериментальное сравнение реализованного алгоритма с существующим в проекте YaccConstructor, показавшее двухкратный прирост на использованных данных.
- Результаты работы успешно представлены на международной конференции “Tools and Methods of Program Analysis” (Москва, 2017г.) в докладе “Extended Context-Free Grammars Parsing with Generalized LL”

Список литературы

[1] ANTLR Project website. — http://www.antlr.org/.

[2] Afroozeh Ali, Izmaylova Anastasia. Faster, Practical GLL Parsing // International Conference on Compiler Construction / Springer. — 2015. — P. 89–108.

[3] Aho Alfred V, Hopcroft John E. The design and analysis of computer algorithms. — Pearson Education India, 1974.

[4] Alblas Henk, Schaap-Kruseman Joos. An attributed ELL (1)-parser generator // International Workshop on Compiler Construction / Springer. — 1990. — P. 208–209.

[5] Bison Project website. — https://www.gnu.org/software/bison/.

[6] Breveglieri Luca, Crespi Reghizzi Stefano, Morzenti Angelo. Shift-Reduce Parsers for Transition Networks // Language and Automata Theory and Applications: 8th International Conference, LATA 2014, Madrid, Spain, March 10-14, 2014. Proceedings / Ed. by Adrian-Horia Dediu, Carlos Martín-Vide, José-Luis Sierra-Rodríguez, Bianca Truthe. — Cham : Springer International Publishing, 2014. — P. 222–235. — ISBN: 978-3-319-04921-2. — Access mode: http://dx.doi.org/10.1007/978-3-319-04921-2_18.

[7] Breveglieri Luca, Reghizzi Stefano Crespi, Morzenti Angelo. Shift-reduce parsers for transition networks // International Conference on Language and Automata Theory and Applications / Springer. — 2014. — P. 222–235.

[8] Brüggemann-Klein Anne, Wood Derick. On predictive parsing and extended context-free grammars // International Conference on Implementation and Application of Automata / Springer. — 2002. — P. 239–247.

[9] Bruggemann-Klein Anne, Wood Derick. The parsing of extended context-free grammars. — 2002.

[10] Heckmann Reinhold. An efficient ELL (1)-parser generator // Acta Informatica. — 1986. — Vol. 23, no. 2. — P. 127–148.

[11] Heilbrunner Stephan. On the definition of ELR (k) and ELL (k) grammars // Acta Informatica. — 1979. — Vol. 11, no. 2. — P. 169–176.

[12] Hemerik Kees. Towards a Taxonomy for ECFG and RRPG Parsing // International Conference on Language and Automata Theory and Applications / Springer. — 2009. — P. 410–421.

[13] An n log n algorithm for minimizing states in a finite automaton : Rep. / DTIC Document ; Executor: John Hopcroft : 1971.

[14] Lee Gyung-Ok, Kim Do-Hyung. Characterization of extended LR (k) grammars // Information processing letters. — 1997. — Vol. 64, no. 2. — P. 75–82.

[15] Morimoto Shin-ichi, Sassa Masataka. Yet another generation of LALR parsers for regular right part grammars // Acta informatica. — 2001. — Vol. 37, no. 9. — P. 671–697.

[16] Nawrocki Eric P, Eddy Sean R. Infernal 1.1: 100-fold faster RNA homology searches // Bioinformatics. — 2013. — Vol. 29, no. 22. — P. 2933–2935.

[17] Purdom Jr Paul Walton, Brown Cynthia A. Parsing extended LR (k) grammars // Acta Informatica. — 1981. — Vol. 15, no. 2. — P. 115–127.

[18] Quantifying variances in comparative RNA secondary structure prediction / James WJ Anderson, Ádám Novák, Zsuzsanna Sükösd et al. // BMC Bioinformatics. — 2013. — Vol. 14, no. 1. — P. 149.

[19] Ragozina Anastasiya. GLL-based relaxed parsing of dynamically generated code : Master's Thesis / Anastasiya Ragozina ; SpBU. — 2016.

[20] Reconstructing 16S rRNA genes in metagenomic data / Cheng Yuan, Jikai Lei, James Cole, Yanni Sun // Bioinformatics. — 2015. — Vol. 31, no. 12. — P. i35–i43.

[21] Rekers Joan Gerard. Parser generation for interactive environments : Ph. D. thesis / Joan Gerard Rekers ; Universiteit van Amsterdam. — 1992.

[22] Scott Elizabeth, Johnstone Adrian. GLL parsing // Electronic Notes in Theoretical Computer Science. — 2010. — Vol. 253, no. 7. — P. 177–189.

[23] Scott Elizabeth, Johnstone Adrian. GLL parse-tree generation // Science of Computer Programming. — 2013. — Vol. 78, no. 10. — P. 1828–1844.

[24] Scott Elizabeth, Johnstone Adrian. Structuring the GLL parsing algorithm for performance // Science of Computer Programming. — 2016. — Vol. 125. — P. 1–22.

[25] Scott Elizabeth, Johnstone Adrian, Economopoulos Rob. BRNGLR: a cubic Tomita-style GLR parsing algorithm // Acta informatica. — 2007. — Vol. 44, no. 6. — P. 427–461.

[26] Tellier Isabelle. Learning recursive automata from positive examples // Revue des Sciences et Technologies de l'Information-Série RIA: Revue d'Intelligence Artificielle. — 2006. — Vol. 20, no. 6. — P. 775–804.

[27] Thompson Ken. Programming Techniques: Regular Expression Search Algorithm // Commun. ACM. — 1968. — Jun. — Vol. 11, no. 6. — P. 419–422. — Access mode: http://doi.acm.org/10.1145/363347.363387.

[28] Wheeler Travis J, Eddy Sean R. nhmmer: DNA homology search with profile HMMs // Bioinformatics. — 2013. — P. btt403.

[29] Wirth Niklaus. Extended Backus-Naur Form (EBNF) // ISO/IEC. — 1996. — Vol. 14977. — P. 2996.

[30] Xander: employing a novel method for efficient gene-targeted metagenomic assembly / Qiong Wang, Jordan A. Fish, Mariah Gilman et al. // Microbiome. — 2015. — Vol. 3, no. 1. — P. 32. — Access mode: http://dx.doi.org/10.1186/s40168-015-0093-6.

[31] YaccConstructor [Электронный ресурс]. — Режим доступа: https://github.com/YaccConstructor/ (дата обращения: 11.05.2015).

[32] YaccConstructor Project repository. — https://github.com/YaccConstructor/YaccConstructor.

Приложение. Псевдокод Generalized LL алгоритма

```
function ADD(L, u, i, w)
    if (L, u, i, w) ∉ U then
        U.add(L, u, i, w)
        R.enqueue(L, u, i, w)

function CREATE(L, u, i, w)
    (X ::= αA · β) ← L
    if (∃ GSS node labeled (A, i)) then
        v ← GSS node labeled (A, i)
        if (there is no GSS edge from v to u labeled (L, w)) then
            add GSS edge from v to u labeled (L, w)
            for ((v, z) ∈ P) do
                y ← getNodeP(L, w, z)
                add(L, u, h, y) where h is the right extent of y
    else
        v ← new GSS node labeled (A, i)
        create GSS edge from v to u labeled (L, w)
        for each alternative α_k of A do
            add(α_k, v, i, $)
    return v

function POP(u, i, z)
    if ((u, z) ∉ P) then
        P.add(u, z)
        for all GSS edges (u, L, w, v) do
            y ← getNodeP(L, w, z)
            add(L, v, i, y)

function GETNODET(x, i)
    if (x = ε) then  h ← i
    else h ← i + 1
    y ← find or create SPPF node labelled (x, i, h)
```

```
    return y
function getNodeP(X ::= α · β, w, z)
    if (α is a terminal or a non-nullable nontermial) & (β ≠ ε) then
        return z
    else
        if (β = ε) then  L ← X
        else L ← (X ::= α · β)
        (_, k, i) ← z
        if (w ≠ $) then
            (_, j, k) ← w
            y ← find or create SPPF node labelled (L, j, i)
            if (∄ child of y labelled (X ::= α · β, k)) then
                y′ ← new packedNode(X ::= α · β, k)
                y′.addLeftChild(w)
                y′.addRightChild(z)
                y.addChild(y′)
        else
            y ← find or create SPPF node labelled (L, k, i)
            if (∄ child of y labelled (X ::= α · β, k)) then
                y′ ← new packedNode(X ::= α · β, k)
                y′.addRightChild(z)
                y.addChild(y′)
        return y
function dispatcher
    if R ≠ ∅ then
        (C_L, C_u, i, C_N) ← R.dequeue()
        C_R ← $
        dispatch ← false
    else stop ← true
function processing
    dispatch ← true
    switch C_L do
```

case $(X \to \alpha \cdot x\beta)$ where $(x = input[i] \mid x = \varepsilon)$

$C_R \leftarrow$ **getNodeT**(x, i)

if $x \neq \varepsilon$ **then** $i \leftarrow i + 1$

$C_L \leftarrow (X \to \alpha x \cdot \beta)$

$C_N \leftarrow$ **getNodeP**(C_L, C_N, C_R)

$dispatch \leftarrow false$

case $(X \to \alpha \cdot A\beta)$ where A is nonterminal

create$((X \to \alpha A \cdot \beta), C_u, i, C_N)$

case $(X \to \alpha \cdot)$

pop(C_u, i, C_N)

function PARSE

while not *stop* **do**

if *dispatch* **then** **dispatcher**()

else processing()

if SPPF node $(StartNonterminal, 0, input.length)$ exists **then**

return this node

else report failure