

Санкт-Петербургский государственный университет

Программная инженерия

Горохов Артем Владимирович

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

Выпускная 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 STATE 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 Dmitry Avduhin

Saint-Petersburg  
2017

# Оглавление

Введение	4
1. Постановка задачи	6
2. Обзор	7
2.1. Расширенные контекстно-свободные грамматики . . . . .	7
2.2. Структурированный в виде графа стек . . . . .	9
2.3. Сжатое представление леса разбора . . . . .	10
2.4. Алгоритм Generalised LL . . . . .	11
2.5. Анализ метагеномных сборок . . . . .	12
2.6. Проект YaccConstructor . . . . .	13
3. Представление ECFG	14
4. Лес разбора по ECFG	16
5. Алгоритм построения леса разбора по ECFG	19
6. Синтаксический анализ регулярных множеств	22
7. Реализация	24
8. Эксперименты	25
Заключение	28
Список литературы	29
Приложение	32

# Введение

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

Существуют различные инструменты для создания синтаксических анализаторов, которые по входной грамматике позволяют создать инструментальное средство для синтаксической обработки текстов, созданных на этом языке. Проблема заключается в том, что эти инструменты сначала преобразуют грамматику к контекстно-свободной форме, и только по ней строят синтаксический анализатор.

Есть работы, которые описывают синтаксический анализ с помощью расширенных контекстно-свободных грамматик (extended context-free grammar (ECFG)). Но нет инструментов, основанных на данных работах. Кроме того, подходы, описанные в данных исследованиях, поддерживают лишь подклассы контекстно-свободных языков.

Алгоритмы обобщённого синтаксического анализа, например Generalised LL [17], способны использовать контекстно-свободные грамматики описывающие произвольные контекстно-свободные языки. Но они так же не работают с грамматиками в форме EBNF без предварительного преобразования к контекстно-свободной форме.

В биоинформатике стоит задача поиска генов и иных последовательностей в геномах. Эти последовательности имеют некоторые общие свойства, которые можно описать контекстно-свободной грамматикой. Есть инструменты типа *infernal* [12], которые используют синтаксический анализ для поиска структур в геномах. Но не всегда генетический материал образует обычную последовательность. Метагеномные сборки - результат считывания генов нескольких организмов, представленный в виде конечного автомата, пути в котором задают гены этих организмов. Подход к синтаксическому анализу конечных автоматов называют синтаксическим анализом регулярных множеств. Было бы здорово

применить полученный GLL алгоритм для синтаксического анализа регулярных множеств, в частности — метагеномных сборок.

На нашей кафедре, в рамках исследовательского проекта YaccConstructor [24], разрабатывается подход поиска структур заданных с помощью контекстно-свободной грамматики в метагеномных сборках, основанный на алгоритме Generalised LL. Предполагается, что синтаксический анализ по ECFG без преобразований даст ощутимый прирост производительности существующего подхода.

# 1. Постановка задачи

Целью данной работы является разработка модификации алгоритма GLL работающей с грамматиками в расширенной форме Бэкуса-Наура и проверка того, как полученный алгоритм влияет на производительность поиска структур заданных с помощью контекстно-свободной грамматики в метагенномных сборках. Для её достижения были поставлены следующие задачи.

- Выбрать или разработать подходящее представление ECFG.
- Спроектировать структуру данных для представления леса разбора по ECFG.
- Разработать алгоритм на основе Generalised LL, строящий лес разбора по ECFG.
- Разработать механизм анализа регулярных множеств в алгоритме.
- Реализовать алгоритм в рамках проекта YaccConstructor.
- Провести эксперименты и сравнение.

## 2. Обзор

### 2.1. Расширенные контекстно-свободные грамматики

Статический анализ программ обычно выполняется над структурным представлением кода. И синтаксический анализ - это классический способ получить такое представление. Генераторы синтаксических анализаторов часто используются для автоматизации создания анализатора: эти инструменты получают синтаксический анализатор по грамматике.

Расширенная форма Бэкуса-Наура (EBNF) [23] является метасинтаксисом для представления контекстно-свободных грамматик. В дополнение к конструкциям используемым в форме Бэкуса-Наура, в ней используются следующие конструкции: альтернатива  $|$ , необязательные символы  $[...]$ , повторение  $\{...\}$  и группировка  $(...)$ .

Эта форма широко используется для спецификации грамматики в технической документации ввиду того, что выразительная сила EBNF делает спецификацию синтаксиса более компактной и удобочитаемой. Поскольку документация является одним из основных источников информации о языке для разработчиков синтаксических анализаторов, было бы полезно иметь генератор анализаторов, который поддерживает грамматики в EBNF. Заметим, что EBNF является лишь стандартизированной формой для *расширенных контекстно-свободных грамматик* [9], которые могут быть определены следующим образом:

**Определение 1.** *Расширенная контекстно-свободная грамматика (ECFG) [9] — это кортеж  $(N, \Sigma, P, S)$ , где  $N$  и  $\Sigma$  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 CFG, и, как результат, генераторы анализаторов требуют преобразования в CFG. Возможно преобразование ECFG в CFG [8], но это преобразование 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 синтаксического анализа.

Существует широкий спектр методов анализа и алгоритмов [3, 4, 6, 7, 8, 9, 11, 13], которые способны обрабатывать грамматику в ECFG. Детальный обзор результатов и задач в области обработки ECFG представлены в статье “Towards a Taxonomy for ECFG and RRPg Parsing” [9]. Заметим только, что большинство алгоритмов основаны на классических методах LL [7, 3, 5] и LR [13, 11, 4], но они работают только с ограниченными подклассами ECFG. Таким образом, нет решения для обработки произвольных (в том числе неоднозначных) ECFG.

Алгоритмы синтаксического анализа на основе LL более интуитивны, 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 леворекурсивные грамматики — проблема для инструментов на основе LL. Другим ограничением для LL анализаторов являются неоднозначности в грамматике, которые, вместе с предыдущими недостатками, усложняют создание синтаксических анализаторов. Алгоритм Generalised LL, предложенный в [17], 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 алгоритма, его



$$\begin{array}{ll}
S ::= a a b c d \mid a a c d \mid a a c e \mid a a & S ::= a a (b c d \mid c (d \mid e) \mid \varepsilon) \\
\text{(a) Исходная грамматика } G_0 & \text{(b) Факторизованная грамматика } G'_0
\end{array}$$

Рис. 1: Пример факторизации грамматики

авторы Elizabeth Scott и Adrian Johnstone предложили поддержку лево-факторизованных грамматик в этом алгоритме [19]. 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 — 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 [19], и показано, что для некоторых грамматик факторизация существенно увеличивает производительность алгоритма GLL. В данной работе эта идея развита в поддержку расширенных контекстно-свободных грамматик.

## 2.2. Структурированный в виде графа стек

В процессе синтаксического анализа используется стек, позволяющий отслеживать историю разбора нетерминалов. Но грамматика может быть неоднозначной и для каждого варианта разбора плодится новый стек, который в дальнейшем поддерживается. Но такие стеки не эффективно хранить и использовать, так как они имеют довольно много одинаковых узлов. Поэтому был предложен структурированный в виде графа стек (Graph Structured Stack(GSS)). GSS комбинирует в себе все варианты стеков.

Анастасия Рагозина в своей работе использует стек предложенный в работе [1]. 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 так же 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 несколько выводов входа, строится несколько деревьев: для каждого варианта разбора. Например для грамматики  $G_0$  2 и входа будут построены 2 дерева, показанные на рис. 3а.

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 [20], для уменьшения потребления памяти и достижения кубической наихудшей временной и пространственной сложности. Бинаризованный SPPF может использоваться в GLL [18] и содержит следующие типы узлов. ( $i$  и  $j$  — 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 есть 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. То есть если у узла или его потомков есть более одного упакованного ребёнка, то для него есть несколько вариантов разбора для строки. Промежуточные и упакованные узлы необходимы для бинаризации SPPF. Так, де-

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

Рис. 2: Грамматика  $G_0$

ревя, представленные на рис. 3а, объединяются в SPPF показанный на рис. 3б.

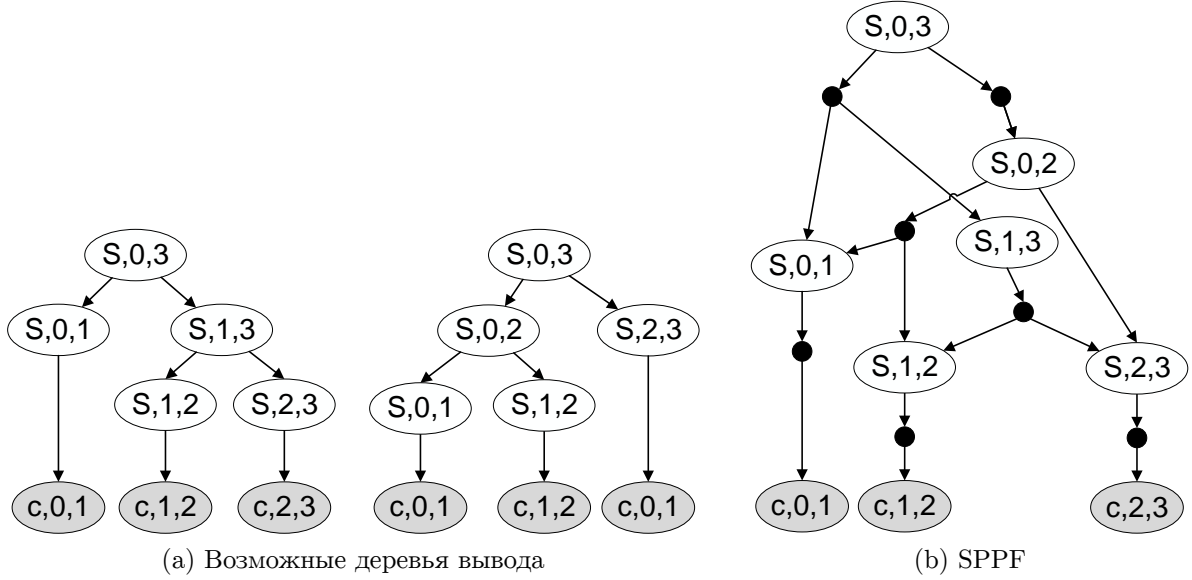


Рис. 3: Пример для входа  $ccc$  и грамматики  $G_0$

## 2.4. Алгоритм Generalised LL

Цель обобщенных алгоритмов синтаксического анализа - обеспечить создание синтаксических анализаторов по произвольным контекстно-свободным грамматикам. Алгоритм Generalised LL (GLL) [17] включает в себя свойства классических LL алгоритмов: он более интуитивен 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 процес-

са синтаксического анализа. Дескриптор представляет собой кортеж  $(L, i, T, S)$ , где:

- $L$  — указатель на позицию в грамматике вида  $(S \rightarrow \alpha \cdot \beta)$ ;
- $i$  — позиция во входе;
- $T$  — корень построенного леса разбора;
- $S$  — текущий узел стека (GSS) [1].

GLL движется одновременно по входу и грамматике, создавая множество дескрипторов в случае неоднозначности и использует очередь для управления обработкой дескрипторов. В начальном состоянии есть только один дескриптор, который состоит из начальной позиции в грамматике  $(S \rightarrow \cdot \beta)$ , 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 есть 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 [15] 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 поиска организмов в биологическом материале. В результате эксперимента получают последовательности генов — строки над алфавитом  $\{A; C; G; T\}$ . Извлечённые из материала последовательности объединяются в конечный автомат, пути в котором задают полученные гены. Этот автомат и есть метагеномная сборка.

Известный подход для поиска организмов в метагеномной сборке это выделение участков соответствующих структурам тРНК, 16s рРНК и др. У этих структур есть некоторые общие свойства, которые могут быть описаны контекстно-свободной грамматикой [14]. Таким образом, можно использовать синтаксический анализ регулярных множеств для поиска структур. В работе [15] был предложен подход к анализу метагеномных сборок с помощью алгоритма GLL. Этот подход был реализован в рамках проекта YaccConstructor.

## 2.6. Проект YaccConstructor

YaccConstructor — исследовательский проект кафедры системного программирования 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) [21] для представления ECFG. Будем использовать следующее определение RA.

**Определение 2.** Рекурсивный автомат  $R$  это кортеж  $(\Sigma, Q, S, F, \delta)$ , где  $\Sigma$  — конечное множество терминалов,  $Q$  — конечное множество состояний,  $S \in Q$  — начальное состояние,  $F \subseteq Q$  — множество конечных состояний,  $\delta : Q \times (\Sigma \cup Q) \rightarrow Q$  — 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 ( $\Sigma$ ), 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 [22] для каждой правой части продукций.
- Создать карту из каждого нетерминала в соответствующее начальное состояние автомата. Эта карта должна оставаться консистентной на протяжении всех следующих шагов.
- Преобразовать автоматы из предыдущего шага в детерминированные без  $\varepsilon$ -переходов используя алгоритм, описанный в [2].

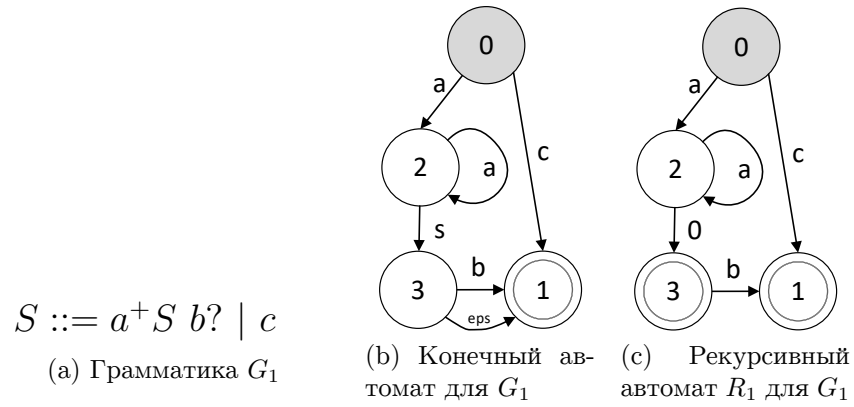


Рис. 4: Преобразование грамматики в рекурсивный автомат

- Минимизировать детерминированный автомат, используя, например, алгоритм Джона Хопкрофта [10].
- 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  $\Delta : Q \rightarrow N$  где  $N$  — имя нетерминала.

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

## 4. Лес разбора по 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  $\varepsilon$ , 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  $\alpha$  для рекурсивного автомата  $R = (\Sigma, Q, S, F, \delta)$  это дерево со следующими свойствами:*

- Корень помечен  $\Delta(S)$ ;
- Листья — терминалы  $a \in (\Sigma \cup \varepsilon)$ ;
- Остальные узлы — нетерминалы  $A \in \Delta(Q)$ ;
- У узла с меткой  $N_i = \Delta(q_i)$  есть:
  - 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  $\varepsilon$  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) [16] может использоваться как компактное представление всех возможных деревьев разбора. Будем использовать биаризованную версию SPPF, предложенную в [20], для уменьшения потребления памяти и достижения кубической наихудшей временной и



пространственной сложности. Бинаризованный SPPF может использоваться в GLL [18] 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 есть 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 **getNodeP**, 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 грамматике 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

```

```

     $y \leftarrow z$ 
else  $y \leftarrow \text{getNodeP}(S, S, w, z)$ 
return  $(y, x)$ 

function GETNODEP( $S, L, w, z$ )
     $(\_, k, i) \leftarrow z$ 
    if ( $w \neq \$$ ) then
         $(\_, j, k) \leftarrow w$ 
         $y \leftarrow$  find or create SPPF node labelled  $(L, j, i)$ 
        if ( $\nexists$  child of  $y$  labelled  $(S, k)$ ) then
             $y' \leftarrow \text{new packedNode}(S, k)$ 
             $y'.addLeftChild(w)$ 
             $y'.addRightChild(z)$ 
             $y.addChild(y')$ 
        else
             $y \leftarrow$  find or create SPPF node labelled  $(L, k, i)$ 
            if ( $\nexists$  child of  $y$  labelled  $(S, k)$ ) then
                 $y' \leftarrow \text{new packedNode}(S, k)$ 
                 $y'.addRightChild(z)$ 
                 $y.addChild(y')$ 
    return  $y$ 

```

Рассмотрим пример SPPF для ECFG  $G_1$  (рис. 4а). 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. 4с) и анализатор по нему. Возможные деревья ввода последовательности  $aacb$  показаны на рис. 5а. SPPF, созданный синтаксическим анализатором (рис. 5b), содержит в себе все три дерева.

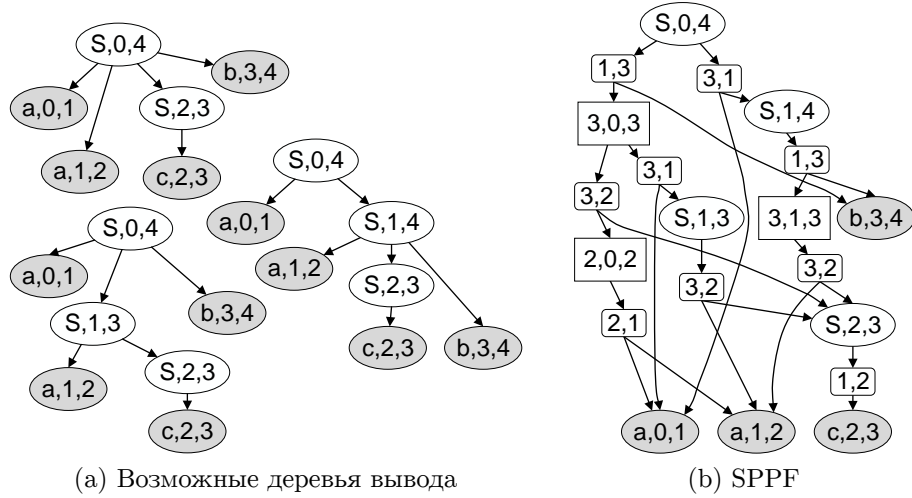


Рис. 5: Пример для входа  $aacb$  и автомата  $R_1$

## 5. Алгоритм построения леса разбора по 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  $\omega$ ,  $C_N$  – 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  $(\varepsilon, i, i)$  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 **getNode** чтобы построить родителя для  $C_N$  и  $t$ . Функция **getNode** возвращает кортеж  $(y, N)$ , где  $N$  — опциональный нетерминальный узел. Создать дескриптор  $(q, C_U, i + 1, y)$  и, если в  $q$  есть 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  $i + 1$ .

- **Есть 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( $S_{call}, S_{next}, u, i, w$ )
   $A \leftarrow \Delta(S_{call})$ 
  if ( $\exists$  GSS node labeled ( $A, i$ )) then
     $v \leftarrow$  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 ( $((v, z) \in \mathcal{P})$  do
         $(y, N) \leftarrow$  getNodes( $S_{next}, u.nonterm, w, z$ )
         $(\_, \_, h) \leftarrow y$ 
        add( $S_{next}, u, h, y$ )
        if  $N \neq \$$  then
           $(\_, \_, h) \leftarrow N$ ; pop( $u, h, N$ )
      else
         $v \leftarrow$  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 ( $((u, z) \notin \mathcal{P})$  then
     $\mathcal{P}.add(u, z)$ 

```

```

for all GSS edges  $(u, S, w, v)$  do
     $(y, N) \leftarrow \text{getNodes}(S, v.\text{nonterm}, w, z)$ 
    add $(S, v, i, y)$ 
    if  $N \neq \$$  then pop $(v, i, N)$ 

function PARSE
     $GSSroot \leftarrow \text{newGSSnode}(\text{StartNonterminal}, 0)$ 
     $R.\text{enqueue}(\text{StartState}, GSSroot, 0, \$)$ 
    while  $R \neq \emptyset$  do
         $(C_S, C_U, i, C_N) \leftarrow R.\text{dequeue}()$ 
        if  $(C_N = \$)$  and  $(C_S \text{ is final state})$  then
             $\text{eps} \leftarrow \text{getNodeT}(\varepsilon, i)$ 
             $(\_, N) \leftarrow \text{getNodes}(C_S, C_U.\text{nonterm}, \$, \text{eps})$ 
            pop $(C_U, i, N)$ 
        for each  $\text{transition}(C_S, \text{label}, S_{\text{next}})$  do
            switch  $\text{label}$  do
                case  $\text{Terminal}(x)$  where  $(x = \text{input}[i])$ 
                     $T \leftarrow \text{getNodeT}(x, i)$ 
                     $(y, N) \leftarrow \text{getNodes}(S_{\text{next}}, C_U.\text{nonterm}, C_N, T)$ 
                    if  $N \neq \$$  then pop $(C_U, i + 1, N)$ 
                    if  $S_{\text{next}}$  have multiple ingoing transitions then
                        add $(S_{\text{next}}, C_U, i + 1, y)$ 
                    else
                         $R.\text{enqueue}(S_{\text{next}}, C_U, i + 1, y)$ 
                case  $\text{Nonterminal}(S_{\text{call}})$ 
                    create $(S_{\text{call}}, S_{\text{next}}, C_U, i, C_N)$ 
    if SPPF node  $(\text{StartNonterminal}, 0, \text{input.length})$  exists then
        return this node
    else report failure

```

## 6. Синтаксический анализ регулярных множеств

Для работы с метагенными 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, основания функция приобретает следующий вид:

**function** PARSEREGULARSET

$GSSroot \leftarrow newGSSnode(StartNonterminal, input.StartState)$

$R.enqueue(StartState, GSSroot, input.StartState, \$)$

**while**  $R \neq \emptyset$  **do**

$(C_S, C_U, i, C_N) \leftarrow R.dequeue()$

**if**  $(C_N = \$)$  and  $(C_S \text{ is final state})$  **then**

$eps \leftarrow \text{getNodeT}(\varepsilon, i)$

$(\_, N) \leftarrow \text{getNodes}(C_S, C_U.nonterm, \$, eps)$

**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**

$T \leftarrow \text{getNodeT}(x, i)$

$(y, N) \leftarrow \text{getNodes}(S_{next}, C_U.nonterm, C_N, T)$

**if**  $N \neq \$$  **then** **pop** $(C_U, k, N)$

**if**  $S_{next}$  have multiple ingoing transitions **then**

**add** $(S_{next}, C_U, k, y)$

**else**

$R.enqueue(S_{next}, C_U, k, y)$

**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.

Проблема в том, что алгоритм не поддерживает циклы во входных автоматах в полной мере. Например, для грамматики  $S ::= a^*$  и входного автомата на рис. 6 Алгоритм просто не остановится, так как будет постоянно обрабатывать один и тот же дескриптор. Этого можно избежать, если для обработки терминалов не просто добавлять дескрипторы в очередь, а вызывать функцию **add**. Это не изменит теоретическую сложность алгоритма, но понятно, что на деле может сказаться на производительности в худшую сторону. Поэтому этот подход можно применять лишь только в случае присутствия циклов во ходе.



Рис. 6: Пример входа для грамматики  $S ::= a^*$  на котором алгоритм не остановится.

## 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.

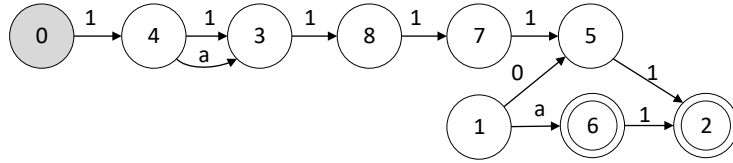


## 8. Эксперименты

Реализация описанного алгоритма GLL была сравнена с реализованным анализатором для факторизованных грамматик. Были сравнены собственная реализация анализатора для факторизованных грамматик и реализация алгоритма описанного в этой работе. Была использована грамматика  $G_2$  (рис. 7а), так как она содержит длинные последовательности в альтернативах, которые не сливаются при факторизации. Рекурсивный автомат построенный для этой грамматике показан на рис. 7б.

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

(а) Грамматика  $G_2$



(б) РА для грамматики  $G_2$

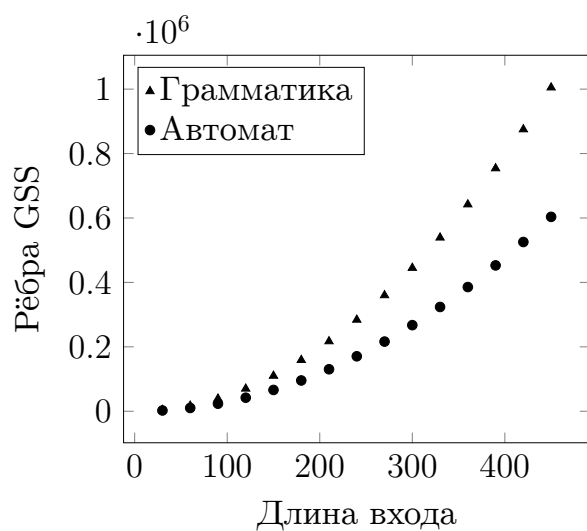
Рис. 7: Грамматика  $G_2$  и РА для неё

Для этой грамматика синтаксический анализатор построенный по рекурсивному автомату создаёт меньше узлов GSS, так как цепочки нетерминалов  $K$  представлены единственным путём в РА. Эта особенность ведёт к снижению количества узлов SPPF и дескрипторов.

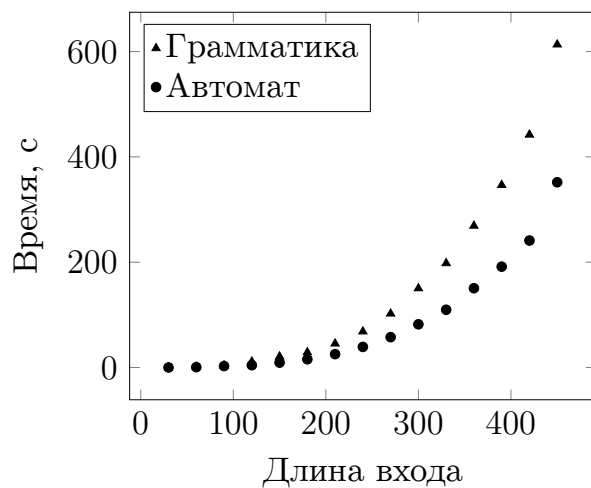
Эксперименты проводились на входах различной длины, результаты приведены на рис. 8. Точные данные для входа  $a^{450}$  показаны в таблице 1.

Тесты проводились на ПК со следующими характеристиками:

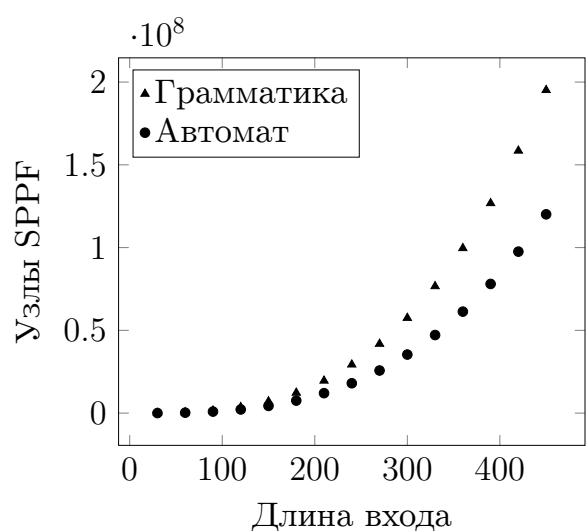
- OS: Microsoft Windows 10 Pro x64
- CPU: Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, 3601 Mhz, 4 Cores, 4 Logical Processors
- RAM: 16 GB



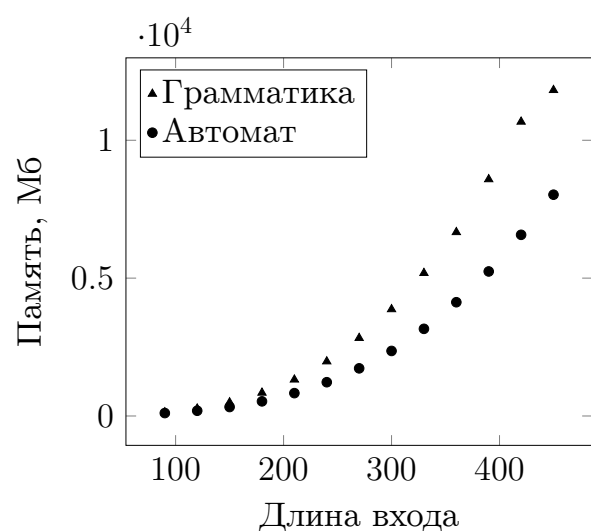
(a) Количество рёбер GSS.



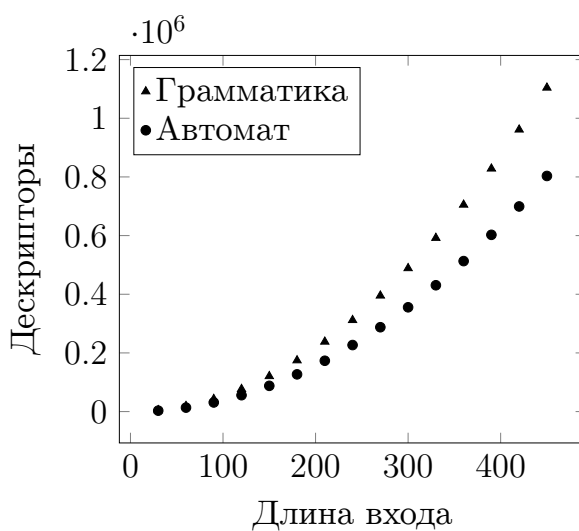
(b) Время работы.



(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 SPFF	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 описанный подход показывает результаты лучше, нежели анализатор построенный по факторизованным грамматикам. 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 SPFF и использует на 33% меньше памяти.

Кроме того было проведено сравнение производительности базового алгоритма GLL и описанного в данной работе в задаче поиска 16s pPHK в метагеномной сборке. Решение уже было предложено в рамках проекта YaccConstrutor. Результаты сравнения для части сборки приведены в таблице 2 и показывают, что при работе с метагеномными сборками новый алгоритм, в среднем, использует на 60% меньше памяти и работает в два раза быстрее.

	Использование памяти			Время, мин
	Дескрипторы	Рёбра GSS	Узлы GSS	
Грамматика	21,134,080	7,482,789	2,731,529	02.26
RA	9,153,352	2,792,330	839,148	01.25
Ratio	57%	63%	69 %	45 %

Таблица 2: Результаты экспериментов с метагеномной сборкой

## Заключение

В рамках данной работы разработана и реализована модификация алгоритма GLL, работающая с расширенными контекстно-свободными грамматиками и 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 по ECFG на основе сжатого леса разбора(SPPF).
- Разработан алгоритм на основе Generalised LL, строящий лес разбора по ECFG.
- Алгоритм реализован в рамках проекта YaccConstructor.
- Проведены эксперименты, показавшие двухкратный прирост производительности на имеющихся метагеномных сборках по сравнению с существующим решением.
- Результаты работы успешно представлены на международной конференции “Tools and Methods of Program Analysis”(Москва, 2017г.)

## Список литературы

- [1] Afroozeh Ali, Izmaylova Anastasia. Faster, Practical GLL Parsing // International Conference on Compiler Construction / Springer. — 2015. — P. 89–108.
- [2] Aho Alfred V, Hopcroft John E. The design and analysis of computer algorithms. — Pearson Education India, 1974.
- [3] Alblas Henk, Schaap-Kruseman Joos. An attributed ELL (1)-parser generator // International Workshop on Compiler Construction / Springer. — 1990. — P. 208–209.
- [4] 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.
- [5] 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.
- [6] Brüggemann-Klein Anne, Wood Derick. The parsing of extended context-free grammars. — 2002.
- [7] Heckmann Reinhold. An efficient ELL (1)-parser generator // Acta Informatica. — 1986. — Vol. 23, no. 2. — P. 127–148.
- [8] Heilbrunner Stephan. On the definition of ELR (k) and ELL (k) grammars // Acta Informatica. — 1979. — Vol. 11, no. 2. — P. 169–176.
- [9] Hemerik Kees. Towards a Taxonomy for ECFG and RRPg Parsing // International Conference on Language and Automata Theory and Applications / Springer. — 2009. — P. 410–421.
- [10] An  $n \log n$  algorithm for minimizing states in a finite automaton : Rep. / DTIC Document ; Executor: John Hopcroft : 1971.

- [11] 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.
- [12] Nawrocki Eric P, Eddy Sean R. Infernal 1.1: 100-fold faster RNA homology searches // *Bioinformatics*. — 2013. — Vol. 29, no. 22. — P. 2933–2935.
- [13] Purdom Jr Paul Walton, Brown Cynthia A. Parsing extended LR (k) grammars // *Acta Informatica*. — 1981. — Vol. 15, no. 2. — P. 115–127.
- [14] 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.
- [15] Ragozina Anastasiya. GLL-based relaxed parsing of dynamically generated code : Master’s Thesis / Anastasiya Ragozina ; SpBU. — 2016.
- [16] Rekers Joan Gerard. Parser generation for interactive environments : Ph.D. thesis / Joan Gerard Rekers ; Universiteit van Amsterdam. — 1992.
- [17] Scott Elizabeth, Johnstone Adrian. GLL parsing // *Electronic Notes in Theoretical Computer Science*. — 2010. — Vol. 253, no. 7. — P. 177–189.
- [18] Scott Elizabeth, Johnstone Adrian. GLL parse-tree generation // *Science of Computer Programming*. — 2013. — Vol. 78, no. 10. — P. 1828–1844.
- [19] Scott Elizabeth, Johnstone Adrian. Structuring the GLL parsing algorithm for performance // *Science of Computer Programming*. — 2016. — Vol. 125. — P. 1–22.
- [20] Scott Elizabeth, Johnstone Adrian, Economopoulos Rob. BRNGLR: a cubic Tomita-style GLR parsing algorithm // *Acta informatica*. — 2007. — Vol. 44, no. 6. — P. 427–461.

- [21] 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.
- [22] 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>.
- [23] Wirth Niklaus. Extended Backus-Naur Form (EBNF) // ISO/IEC. — 1996. — Vol. 14977. — P. 2996.
- [24] YaccConstructor Project repository. — <https://github.com/YaccConstructor/YaccConstructor>.

# Приложение

## Псевдокод Generalized LL алгоритма

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

function CREATE( $L, u, i, w$ )
  ( $X ::= \alpha A \cdot \beta$ )  $\leftarrow L$ 
  if ( $\exists$  GSS node labeled ( $A, i$ )) then
     $v \leftarrow$  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 ( $(v, z) \in \mathcal{P}$ ) do
         $y \leftarrow \text{getNodeP}(L, w, z)$ 
        add( $L, u, h, y$ ) where  $h$  is the right extent of  $y$ 
    else
       $v \leftarrow$  new GSS node labeled ( $A, i$ )
      create GSS edge from  $v$  to  $u$  labeled ( $L, w$ )
      for each alternative  $\alpha_k$  of  $A$  do
        add( $\alpha_k, v, i, \$$ )
  return  $v$ 

function POP( $u, i, z$ )
  if ( $(u, z) \notin \mathcal{P}$ ) then
     $\mathcal{P}.add(u, z)$ 
    for all GSS edges ( $u, L, w, v$ ) do
       $y \leftarrow \text{getNodeP}(L, w, z)$ 
      add( $L, v, i, y$ )

function GETNODET( $x, i$ )
  if ( $x = \varepsilon$ ) then  $h \leftarrow i$ 
  else  $h \leftarrow i + 1$ 
   $y \leftarrow$  find or create SPPF node labelled ( $x, i, h$ )
```



```

    return  $y$ 

function GETNODEP( $X ::= \alpha \cdot \beta, w, z$ )
    if ( $\alpha$  is a terminal or a non-nullable nontermial) & ( $\beta \neq \varepsilon$ ) then
        return  $z$ 
    else
        if ( $\beta = \varepsilon$ ) then  $L \leftarrow X$ 
        else  $L \leftarrow (X ::= \alpha \cdot \beta)$ 
         $(\_, k, i) \leftarrow z$ 
        if ( $w \neq \$$ ) then
             $(\_, j, k) \leftarrow w$ 
             $y \leftarrow$  find or create SPPF node labelled  $(L, j, i)$ 
            if ( $\nexists$  child of  $y$  labelled  $(X ::= \alpha \cdot \beta, k)$ ) then
                 $y' \leftarrow$  new packedNode( $X ::= \alpha \cdot \beta, k$ )
                 $y'.addLeftChild(w)$ 
                 $y'.addRightChild(z)$ 
                 $y.addChild(y')$ 
            else
                 $y \leftarrow$  find or create SPPF node labelled  $(L, k, i)$ 
                if ( $\nexists$  child of  $y$  labelled  $(X ::= \alpha \cdot \beta, k)$ ) then
                     $y' \leftarrow$  new packedNode( $X ::= \alpha \cdot \beta, k$ )
                     $y'.addRightChild(z)$ 
                     $y.addChild(y')$ 
        return  $y$ 

function DISPATCHER
    if  $R \neq \emptyset$  then
         $(C_L, C_u, i, C_N) \leftarrow R.dequeue()$ 
         $C_R \leftarrow \$$ 
         $dispatch \leftarrow false$ 
    else  $stop \leftarrow true$ 

function PROCESSING
     $dispatch \leftarrow true$ 
    switch  $C_L$  do

```

```

case  $(X \rightarrow \alpha \cdot x\beta)$  where  $(x = input[i] \mid x = \varepsilon)$ 
     $C_R \leftarrow \mathbf{getNodeT}(x, i)$ 
    if  $x \neq \varepsilon$  then  $i \leftarrow i + 1$ 
     $C_L \leftarrow (X \rightarrow \alpha x \cdot \beta)$ 
     $C_N \leftarrow \mathbf{getNodeP}(C_L, C_N, C_R)$ 
     $dispatch \leftarrow false$ 

case  $(X \rightarrow \alpha \cdot A\beta)$  where  $A$  is nonterminal
    create $((X \rightarrow \alpha A \cdot \beta), C_u, i, C_N)$ 

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

```