Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»

Кафедра Системного Программирования

Орлов Илья Денисович

Генерация анализатора по грамматике в расширенной БНФ

Бакалаврская работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
ст. преп. Григорьев С. В.

Рецензент:
ст. преп. Полозов В. С.

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering Chair

Ilia Orlov

Analyser generation in grammar in extended BNF

Bachelor's Thesis

Admitted for defence.
Head of the chair:
professor A. N. Terekhov

Scientific supervisor:
sen. lect. S. V. Grigoriev

Reviewer:
sen. lect. V. S. Polozov

Saint-Petersburg
2015

Оглавление

Введение **4**

1. Постановка задачи **6**

2. Обзор **7**

2.1. Алгоритмы 7

2.1.1. LR-алгоритмы 7

2.1.2. GLR-алгоритмы 8

2.2. Технологии 10

3. Алгоритмическое решение **12**

3.1. Алгоритм построения генератора 12

3.1.1. Технологии, используемые для построения генератора 12

3.1.2. Алгоритм построения генератора 14

3.2. Алгоритм работы интерпретатора сгенерированного парсера 15

3.2.1. Технологии, используемые при работе интерпретатора 15

3.2.2. Алгоритм работы интерпретатора 20

4. Детали реализации **32**

5. Апробация **33**

Заключение **35**

Список литературы **36**

Введение

В решении задач синтаксического анализа применяются генераторы синтаксических анализаторов – инструменты, которые по входной грамматике, заданной в определённой форме, порождают парсер, соответствующий данной грамматике. Удобным способом представления грамматики языка программирования является расширенная форма Бэкуса-Наура (EBNF). Правые части продукций грамматики в данной форме, в отличие от обычной формы Бэкуса-Наура (BNF), представляют собой регулярные выражения. Такое представление языка является более естественным и позволяет уменьшить запись продукций и сократить их количество, сделав описание более понятным и простым.

На данный момент многие инструменты синтаксического анализа либо не поддерживают возможности работы с правилами в EBNF форме, либо переводят исходную грамматику в другую форму, добавляя новые нетерминалы и продукции. Это означает, что сгенерированный парсер будет выдавать на выходе информацию по грамматике в преобразованной форме, что неудобно для пользователя, так как будет потеряна связь с исходной записью продукций. Более того, увеличение количества продукций и нетерминалов может отразиться на времени работы сгенерированного парсера, поскольку изменение формы грамматики и добавление новых символов увеличивает размер таблиц состояний и переходов.

Проблема появления новых продукций и дальнейшего их использования при выводе решается при помощи создания связи исходной грамматики с ее трансформированной формой. Однако, такое решение может сказаться на работе парсера в сторону снижения производительности, так как необходимо время на восстановление исходной формы грамматики. Таким образом сохранение EBNF без преобразования может решить как проблему появления новых продукций, так и проблему производительности. В 2014 году на кафедре системного программирования Санкт-Петербургского государственного университета был разработан алгоритм, позволяющий работать с грамматикой в EBNF

напрямую. В данном алгоритме распознавание части стека ведется по специальным меткам, сигнализирующим о конце продукции. Возникает гипотеза о том, что алгоритм имеет свои оптимизации: распознавание части стека может быть реализовано при помощи конечного детерминированного автомата, что может дать выигрыш во времени.

В рамках настоящей работы ставятся задачи разработки алгоритма, позволяющего работать с грамматикой в EBNF напрямую и имеющего оптимизации по работе со стеком, а также реализации генератора синтаксических анализаторов, проведения апробации и сравнения результатов с существующими решениями.

1. Постановка задачи

Целью работы является разработка генератора синтаксических анализаторов, позволяющего работать с грамматикой в EBNF напрямую. Для достижения данной цели были поставлены следующие задачи:

- разработка алгоритма, имеющего оптимизации по работе со стеком;
- реализация алгоритма в рамках проекта YaccConstructor: создание генератора синтаксических анализаторов и интерпретатора парсеров, порождаемых генератором;
- апробация решения: тестирование интепретатора парсеров, сравнение времени работы интепретатора с существующими решениями на файлах разной длины.

2. Обзор

Алгоритмы генерации синтаксических анализаторов имеют широкое прикладное применение в программной инженерии. Кроме того, достаточно большое количество научных исследований направлено на оптимизацию уже существующих алгоритмов.

Обзор состоит из двух частей: теоретические алгоритмы и технологии, в рамках которых данные алгоритмы реализованы.

2.1. Алгоритмы

2.1.1. LR-алгоритмы

В области LR-анализаторов, построенных по грамматике в EBNF напрямую без преобразования, было разработано множество теоретических решений [2]. В частности, алгоритмы, предложенные в статьях [4] и [3], не преобразуют исходную форму грамматики и позволяют работать с грамматикой в EBNF напрямую.

Авторы данных статей предлагают следующую идею. Правые части продукций грамматики преобразуются в эквивалентные конечные недетерминированные автоматы. Пара, состоящая из номера продукции и номера состояния конечного автомата для данной продукции, представляет собой аналог LR-ситуации. Тройки из LR-ситуации и множества символов предпросмотра являются элементами, из которых строятся состояния LR-анализатора.

Проблемой такого подхода является то, что правые части продукций могут порождать цепочки любой длины. Для грамматики в BNF при распознавании входной цепочки количество символов со стека, необходимое для снятия при свёртке, равно длине правой части продукции, по которой осуществляется свёртка. Для грамматики в EBNF, цепочки, порождаемые правыми частями продукций, могут не иметь фиксированной длины, а значит конец одной продукции и начало другой невозможно определить таким способом. Для решения данной проблемы авторы статей предлагают дополнительно складывать на стек но-

мера продукций, по которым происходит порождение входной цепочки.

Идея алгоритма состоит в том, чтобы определить, когда закончится порождаться цепочка по некоторой продукции для того, чтобы свернуть по ней соответствующую часть стека. Элементы LR-анализатора делятся на два множества: kernel и nonkernel, которые строятся посредством операции Closure, как и при обычном LR-анализе. Если переход в LR-анализаторе связан с элементом из множества nonkernel, то в данный момент входная цепочка начала порождаться по новой продукции, а значит необходимо положить на стек номер данной продукции. Если же переход связан с элементом из множества kernel, то входная цепочка продолжает порождаться без перехода к новой продукции, а значит нет необходимости складывать номер продукции на стек. При совершении свёртки по определённой продукции необходимо снимать элементы со стека до тех пор, пока не будет снят номер данной продукции. Таким образом часть стека для совершения свёртки будет распознана.

Однако, такой подход позволяет построить LR-анализатор не для всех расширенных LR-грамматик. Причина в том, что переход в LR-анализаторе может быть одновременно связан с элементами из обоих множеств: kernel и nonkernel. Такая ситуация называется конфликтом стека. В статье [3] приведены два алгоритма, первый из которых покрывает не весь класс расширенных LR-грамматик. Второй покрывает класс полностью, но при этом производит дополнительные вычисления: специальные таблицы для каждой продукции и символа входной цепочки, которые помогают разрешить конфликты стека.

2.1.2. GLR-алгоритмы

Большинство алгоритмов, приведённых в вышеперечисленных статьях, имеют существенное ограничение по классу входных грамматик: входная грамматика должна быть однозначна. Проблема неоднозначности грамматики решается при помощи алгоритма обобщённого синтаксического анализа GLR [1]. GLR-алгоритм разрешает конфликты, возникшие в ходе распознавания входной цепочки, путём ветвления стека. Стек в данном случае представляется в виде графа GSS, где вершины

– состояния LR-анализатора, а ребра содержат символы грамматики.

GLR-алгоритм успешно работает с неоднозначными грамматиками, но также не покрывает весь класс контекстно-свободных грамматик. Алгоритм некорректно обрабатывает входные цепочки, грамматика которых содержит вырожденную правую рекурсию.

Проблему такого рода успешно решает модифицированный алгоритм обобщённого синтаксического анализа RNGLR [5]. RNGLR-алгоритм работает корректно со всеми контекстно-свободными грамматиками. Отличие от GLR-алгоритма состоит в том, чтобы производить свёртку на стеке в случае, когда правая часть продукции может породить пустую цепочку.

Алгоритм обобщённого синтаксического анализа RNGLR успешно работает со всеми неоднозначными грамматиками. Однако, грамматики, находящиеся в EBNF, необходимо преобразовать в BNF.

В 2014 году на кафедре системного программирования СПбГУ была предложена модификация алгоритма RNGLR [6], позволяющая работать с грамматиками в EBNF напрямую без преобразования.

Основная идея алгоритма близка к идее, предложенной в статьях [4] и [3], и представляет собой модификацию алгоритма RNGLR. Правые части продукций грамматики представляются в виде конечных недетерминированных автоматов. Модификация состоит в том, чтобы использовать стек, способный обрабатывать конфликтные ситуации, как это происходит в алгоритмах обобщённого синтаксического анализа. При этом данный алгоритм умеет обрабатывать случаи вырождения правой части продукции, аналогично алгоритму RNGLR. Стек представляется в виде графа GSS, на рёбрах которого дополнительно складываются множества, содержащие номера продукций, по которым происходит порождение входной цепочки.

Алгоритм умеет обрабатывать все контекстно-свободные грамматики, но имеет свои недостатки. Размер графа GSS напрямую зависит от количества символов во входной цепочке. В исходных задачах при больших входных данных хранить такую структуру в памяти само по себе достаточно дорого. К тому же использование специальных множеств

на рёбрах графа для распознавания продукций, по которым происходит порождение входной цепочки, ещё больше увеличивает накладные расходы по памяти, что может повлиять на время работы распознавателя.

Таких дополнительных расходов можно избежать, использовав для распознавания части стека реверсивный недетерминированный конечный автомат, полученный из недетерминированного конечного автомата правой части продукции, по которой происходит свёртка. Количество таких автоматов, которое необходимо хранить в памяти, зависит лишь от количества продукций в исходной грамматике, а не от размера входной цепочки, что может сократить расходы по памяти и уменьшить время работы.

2.2. Технологии

Среди генераторов синтаксических анализаторов наиболее известными считаются YACC [15] и последовавшие за ним Ocamlyacc [16] и Bison [11]. Данные генераторы по умолчанию принимают грамматику, описанную в BNF. Версии YACC могут принимать грамматику в EBNF, но при этом производят преобразование исходной грамматики, а не работают с ней напрямую. Генераторы синтаксических анализаторов, такие как Bison [11], Jison [12] и SDF/SGLR [13], умеют обрабатывать неоднозначные грамматики, используя GLR алгоритм. Однако, данные генераторы не предусматривают сохранения исходной формы грамматики.

В 2010 году на кафедре системного программированию СПбГУ была написана работа [10], посвящённая генерации синтаксических анализаторов неоднозначных контекстно-свободных грамматик и грамматике в EBNF без изменения исходной формы. Недостатком данной работы является довольно слабый класс грамматик: LR(0). Данный алгоритм был реализован в рамках проекта YaccConstructor [14].

В 2012 году в рамках данного проекта был реализован алгоритм RNGLR [7], работающий со всеми контекстно-свободными грамматика-

ми, но преобразующий их исходную форму.

В 2014 году в рамках этого же проекта была реализована модификация алгоритма RNGLR, позволяющая работать со всеми контекстно-свободными грамматиками в EBNF без преобразования исходной формы.

3. Алгоритмическое решение

Алгоритм состоит из двух частей:

- алгоритм для построения генератора синтаксических анализаторов;
- алгоритм для построения интерпретатора сгенерированного парсера.

Перед каждой частью дан обзор используемых в алгоритмах технологий.

3.1. Алгоритм построения генератора

3.1.1. Технологии, используемые для построения генератора

На вход алгоритму подаётся грамматика G в расширенной форме Бэкуса-Наура, где правые части продукций грамматики представляют собой регулярные выражения. К каждой правой части продукции грамматики приписана семантика: некоторый набор функций (набор может быть пустым). Семантика к регулярному выражению приписывается следующим образом: к каждой части регулярного выражения приписывается свой набор функций. Под частью регулярного выражения понимается следующее рекурсивное определение:

- символ грамматики (терминал или нетерминал) является частью регулярного выражения;
- последовательность из частей регулярного выражения является частью регулярного выражения;
- выражение вида $A|B$ является частью регулярного выражения, где A и B – части регулярного выражения;
- выражение вида $A?$ является частью регулярного выражения, где A – часть регулярного выражения;

- выражение вида $A*$ является частью регулярного выражения, где A – часть регулярного выражения;
- выражение вида $A+$ является частью регулярного выражения, где A – часть регулярного выражения.

Таким образом правые части продукций грамматики представляют собой регулярные выражения с приписанной семантикой к каждой части данного регулярного выражения.

Поскольку правые части продукций грамматики представляют собой регулярные выражения, то они могут быть преобразованы в эквивалентные недетерминированные конечные автоматы (НКА) по алгоритму, описанному Хопкрофтом [8].

Для построения НКА с учётом семантики, приписанной к регулярному выражению, необходима небольшая модификация данного алгоритма: при построении НКА любую часть регулярного выражения, к которой приписана семантика, необходимо окружить ϵ-переходами.

Пусть в ходе построения НКА при чтении регулярного выражения началась часть этого выражения с приписанной семантикой, и пусть на данном шаге последним построенным состоянием НКА было q_s. Тогда необходимо создать новое состояние v_i и направить ϵ-ребро из q_s в q_i. При этом на данное ребро необходимо положить, кроме ϵ-символа, специальную метку о том, что началась часть регулярного выражения с приписанной семантикой. Затем данная часть регулярного выражения строится по обычному алгоритму. Пусть на выходе после построения данной части имеется состояние q_j НКА. Тогда необходимо создать новое состояние v_f и направить ϵ-ребро из q_j в q_f. На данное ϵ-ребро, помимо ϵ-символа, необходимо положить набор функций данной семантики, а также отметку о том, что часть регулярного выражения с семантикой закончилась.

Необходимо заметить, что, согласно алгоритму, НКА строится рекурсивно. Каждое состояние НКА имеет свой уникальный номер, полученный в ходе построения. При этом если состояние q_i НКА создано раньше, чем состояние q_j, то $i < j$. Это свойство пригодится для даль-

нейшего построения интерпретатора.

3.1.2. Алгоритм построения генератора

Регулярные выражения правых частей грамматики G могут быть заменены на соответствующие НКА, порождающие тот же язык.

Грамматика G может быть представлена как $G = (V_N, V_T, S, Q, \delta, F, P)$, где V_N – конечное множество нетерминальных символов, V_T – конечное множество терминальных символов, $S \in V_N$ – начальный символ грамматики, Q – конечное множество состояний НКА в правых частях продукций, $\delta : Q \times V \to 2^Q$ – функция переходов для НКА (где $V = V_N \cup V_T$), $F \subset Q$ – конечные состояния НКА, P – множество продукций.

Продукция p может быть представлена как пара (A, q), где $A \in V_N$ – левая часть продукции, а q - начальное состояние НКА в правой части.

LR-анализатор строится аналогично классическому построению LR-анализатора. Алгоритм основан на статьях [4] и [5]. Регулярные правые части грамматики преобразуются в эквивалентные НКА согласно алгоритму, описанному выше в разделе 3.1.1. Пара (p, i) является аналогом LR-ситуации, где p – номер продукции, i – состояние НКА, соответствующего этой продукции. Тройки (p, i, γ) являются аналогами пунктов, которые строятся при классическом построении LR-анализатора. p и i являются парой LR-ситуации, γ – множество символов предпросмотра. Посредством функций CLOSURE и GOTO строятся все состояния LR-анализатора и таблица переходов, которая в свою очередь подвергается модификации согласно построению RN-таблиц из статьи [5].

Для проведения синтаксического анализа также потребуются дополнительные построения:

1. Реверсивные НКА (РДКА), построенные из НКА правых частей продукций. Для построения РДКА достаточно задать функцию переходов и определить множество состояний автоматов, а также множества начальных и конечных состояний. За множество состояний положим множество Q, за множество начальных состояний положим множество F, за множество конечных состоя-

ний положим все начальные состояния НКА или $\{q \in Q \mid \exists p = (A, q), p \in P\}$. Функцию переходов δ_r зададим следующим образом: если $\delta(q_i, v) = Q_j$ (где $q_i \in Q, Q_s \subset Q, v \in V$), то $\forall q_j \in Q_j$ $\delta_r(q_j, v) = q_i$. Специальные метки и наборы функций, добавленные на рёбра НКА для подсчёта семантики, не требуют сохранения в РДКА.

2. Реверсивные детерминированные конечные автоматы (РДКА), полученные из РНКА при помощи алгоритма, описанного в книге [9]. Каждый РДКА будет принимать тот же язык, что и соответствующий РНКА.

Множество РНКА в дальнейшем использоваться не будет. Оно необходимо лишь для построения множества РДКА. В итоге для каждой продукции у нас будет построено два конечных автомата. Зная номер продукции можно будет однозначно определить начальные состояния автоматов, соответствующие данной продукции.

Также необходимо учесть ещё одну особенность при построении RN-таблицы. В классической RN-таблице свёртки $(A ::= \alpha.\gamma, a)$ представляют собой тройку (A, m, f), где A – нетерминал, к которому производится свёртка, $m = |\alpha|$, $f = I(\gamma)$, если $m \neq 0$, $f = I(A)$, если $m = 0$, где I – индексирующая функция для ϵ-SPPF дерева. Когда грамматика находится в EBNF, правые части продукции могут порождать строки без фиксированной длины, а значит невозможно производить свёртку по длине правой части. Для этого m заменяется на пару из q_0 и r_0, где q_0 и r_0 – начальные состояния НКА и РДКА соответственно, которые принадлежат продукции, по которой необходимо произвести свёртку.

3.2. Алгоритм работы интерпретатора сгенерированного парсера

3.2.1. Технологии, используемые при работе интерпретатора

Для представления стека в ходе работы интерпретатора используется специальный граф GSS. Построение и работа с данным графом

полностью совпадает с алгоритмом, описанным в статье [5].

Для представления дерева разбора используется специальная структура SPPF – сжатый лес разбора входной цепочки. Построение и работу с данной структурой также можно найти в статье [5]

Для добавления некоторой цепочки символов в SPPF, необходимо построить дерево специального вида для дальнейшего подсчёта семантики для данной цепочки. Данное дерево и будет добавляться в SPPF. Для построения специального дерева будет использоваться НКА.

Поскольку в общем случае НКА может иметь несколько переходов по некоторому символу, а также переходы по ϵ-рёбрам, то необходимо построить интерпретатор НКА для распознавания некоторой цепочки. Далее описан алгоритм, позволяющий распознать входную цепочку при помощи НКА и посчитать соответствующую семантику для данной цепочки, то есть построить дерево специального вида:

- листья дерева есть символы входной цепочки;
- промежуточные узлы дерева, включая корень, есть наборы функций семантики;
- при прочтении листьев дерева слева направо должна получаться входная цепочка;
- пусть некоторая часть seq входной цепочки соответствует некоторой части регулярного выражения, и пусть к данной части регулярного выражения приписан набор функций семантики s. Спустившись от узла дерева, который помечен s, до листьев и прочитав их слева направо, должна получиться цепочка, равная seq.

При этом НКА будет построен так, чтобы распознавать цепочку жадным образом. Любой путь в автомате может быть представлен как последовательность состояний автомата, по которым необходимо пройти, чтобы распознать входную цепочку. У каждого состояния есть свой номер, а значит путь может быть представлен как последовательность

номеров данных состояний. Пусть есть два пути длины n и m (для определённости n < m) с последовательностями состояний $p_1..p_n$ и $q_1..q_m$. Тогда более жадный путь будет определяться следующим образом: если $\exists k : 1 \leq k \leq n$, что $q_k < p_k$ и при этом $q_i = p_i$ для $\forall i : 1 \leq i < k$, то путь длины m назовём более жадным. Иначе назовём более жадным путь длины n.

Пусть НКА построен способом, описанным в разделе 3.1.1. На вход НКА подаётся цепочка символов l, принимаемая данным НКА. На выходе получается построенный лес разбора для подсчёта семантики по данной входной цепочке. При этом НКА может распознать цепочку разными способами. Для этого распознавание будет производиться жадным способом, что даст на выходе ровно одно дерево разбора.

Общая идея алгоритма заключается в том, чтобы считывать очередной символ входной цепочки и совершать всевозможные переходы в НКА по данному символу. После чего для каждого получившегося состояния необходимо совершать всевозможные ϵ-переходы до тех пор, пока следующий переход не будет помечен символом, отличным от ϵ. Таким образом получится множество состояний НКА, которое содержит переходы по символам, отличных от ϵ. После чего можно считывать следующий символ из входной цепочки и производить следующую итерацию алгоритма. Для получения начального множества необходимо из начального состояния НКА совершать аналогичным образом всевозможные ϵ-переходы до тех пор, пока следующий переход не будет помечен символом, отличным от ϵ.

Таким образом каждую итерацию алгоритма можно разбить на две стадии: переход по символу, отличному от ϵ, и переход по ϵ-рёбрам для получения следующего множества состояний. При этом первым шагом алгоритма необходимо посчитать начальное множество состояний НКА, из которых есть переходы, отличные от ϵ. После чего можно запускать итерацию по символам входной цепочки.

Для каждого состояния НКА и i, где i – количество символов, которое было прочитано из входной цепочки, будем хранить следующую тройку $(path, tree, eps)$, где:

- $path$ – путь, состоящий из номеров состояний НКА в том порядке, в котором были произведены шаги по данным состояниям;

- $tree$ – указатель на текущую вершину в дереве для данного состояния НКА;

- eps – множество состояний НКА, в которые был осуществлён переход по ϵ-рёбрам, пока не был встречен символ, отличный от ϵ.

Исходно у каждого состояния НКА, кроме начального, такое множество троек пусто. Они будут строиться в ходе распознавания входной цепочки. Начальное же состояние имеет тройку вида $(0, null, \varnothing)$.

Определим общее правило, при котором не нужно осуществлять дальнейший переход в НКА.

Пусть на k-ом шаге при чтении символа c возможен переход из состояния q_i в состояние q_j, и пусть состоянию q_i для k-ого шага соответствует тройка $(path_i, tree_i, ers_i)$, а состоянию q_j для k-ого шага тройка $(path_j, tree_j, ers_j)$ (если у состояния q_j тройки для k-ого шага нет, то переход необходимо осуществить). Переход не нужно осуществлять, если выполняется одно из следующих условий:

- прочтённый символ есть ϵ, и состояние q_j содержится во множестве eps_i – таким образом будут распознаны ϵ-циклы;

- путь $path_j$ является более жадным, чем путь $path_i$.

Во всех остальных случаях переход по символу необходимо осуществить.

Разберём одну итерацию алгоритма.

Пусть на k-ом шаге обработки входной цепочки был прочитан символ c, отличный от ϵ. Пусть множество состояний, в котором находится НКА, есть Q_i. Выберем из Q_i только те состояния, по которым есть переход по символу c, после чего совершим данные переходы. Рассмотрим некоторый переход в отдельности. Пусть по символу c был совершён переход из состояния q_i в состояние q_j НКА, и пусть состояние q_i содержало тройку $(path_i, tree_i, ers_i)$ для k-ого шага. Необходимо посчитать новую тройку $(path_j, tree_j, ers_j)$ для состояния q_j и k-ого шага:

- путь $path_j$ будет получаться из пути $path_i$ присоединением в конец пути $path_i$ состояния v_i;

- к вершине, на которую указывал $tree_i$, необходимо приписать символ c, а $tree_j$ направить на родителя данной вершины;

- множество eps_j будет пустым;

Пусть после совершения переходов будет получено множество Q_j. Для всех состояний из данного множества необходимо совершать ϵ-переходы до тех пор, пока следующее состояние не будет иметь переход по символу, отличному от ϵ или же состояние окажется конечным состоянием НКА. Данные ϵ-переходы могут быть трёх видов: со специальной меткой о начале семантики, со специальной меткой о конце семантики и набором функций и просто ϵ-переход без меток. Разберём каждый такой переход по отдельности.

Пусть есть ϵ-переход с меткой о начале семантики из состояния q_i в состояние q_j, и пусть состояние q_i содержит тройку $(path_i, tree_i, ers_i)$ для k-ого шага. Необходимо посчитать новую тройку $(path_j, tree_j, ers_j)$ для состояния q_j и k-ого шага:

- путь $path_j$ будет получаться из пути $path_i$ присоединением в конец пути $path_i$ состояния v_i;

- у вершины, на которую указывает $tree_i$, необходимо создать ребёнка, и $tree_j$ направить на данного ребёнка;

- множество eps_j будет множеством eps_i с добавленным состоянием q_i.

Пусть есть ϵ-переход с меткой о конце семантики и набором функций из состояния q_i в состояние q_j, и пусть состояние q_i содержит тройку $(path_i, tree_i, ers_i)$ для k-ого шага. Необходимо посчитать новую тройку $(path_j, tree_j, ers_j)$ для состояния q_j и k-ого шага:

- путь $path_j$ будет получаться из пути $path_i$ присоединением в конец пути $path_i$ состояния v_i;

- к вершине, на которую указывает $tree_i$, необходимо приписать набор функций и $tree_j$ направить на родителя данной вершины;
- множество eps_j будет множеством eps_i с добавленным состоянием q_i.

Пусть есть ϵ-переход без специальных меток из состояния q_i в состояние q_j, и пусть состояние q_i содержит тройку $(path_i, tree_i, ers_i)$ для k-ого шага. Необходимо посчитать новую тройку $(path_j, tree_j, ers_j)$ для состояния q_j и k-ого шага:

- путь $path_j$ будет получаться из пути $path_i$ присоединением в конец пути $path_i$ состояния v_i;
- $tree_j$ положить равным $tree_i$;
- множество eps_j будет множеством eps_i с добавленным состоянием q_i.

Для вычисления начального множества состояний необходимо совершать аналогичные ϵ-переходы от начального состояния НКА.

После прочтения всех символов входной цепочки в конечном состоянии НКА окажется тройка, в которой вторым элементом будет указатель на корень дерева для подсчёта семантики, построенного по входной цепочке.

3.2.2. Алгоритм работы интерпретатора

На вход интерпретатору подаётся строка терминальных символов. На выходе происходит принятие строки и построение дерева разбора (SPPF) или же сообщение об ошибке. В отличие от грамматики в BNF, правые части продукций грамматики в EBNF могут порождать строки без фиксированной длины. В классическом LR-анализе свёртка на стеке происходит по длине правой части продукции. Таким образом когда следующий переход LR-анализатора связан с тем, что необходимо произвести свёртку по некоторой продукции, то со стека снимается столько

символов, сколько стоит в правой части данной продукции. Когда грамматика находится в EBNF, произвести свёртку таким способом невозможно. Для данного действия будет использоваться множество РДКА автоматов, построенное ранее.

Основные множества для построения графа GSS и SPPF, а именно $\mathcal{Q}, \mathcal{R}$ и $\mathcal{N}$, также будут строиться аналогично. Только теперь в $\mathcal{R}$ будут лежать не пятёрки вида (v, X, m, f, y), где v – вершина графа GSS, X, m и f соответствуют тройке при свёртке в RN-таблице, а y – узел в SPPF, который соответствует первому ребру в графе при произведении свёртки, а шестёрки вида (v, X, q_0, r_0, f, y), где вместо m, как и в случае с построением модифицированной RN-таблицы, будут лежать начальные состояния конечных автоматов. Также отличие будет составлять функция REDUCER(i) [3], которая отвечает за выполнение свёртки на стеке на i шаге обработки входной строки, так как теперь свёртка будет происходить не по длине правой части продукции. Отличия будут и в функции AddChildren [5], так как теперь функция будет добавлять дерево для подсчёта семантики для входной цепочки в SPPF.

Пусть на i-ом шаге обработки входной цепочки необходимо произвести свёртку по продукции. Пусть данной свёртке соответствует шестёрка (v, X, q_0, r_0, f, y) в $\mathcal{R}$. Для распознавания части стека, которая будет заменена на X, используется РДКА, у которого начальным состоянием и является r_0. Из вершины стека v осуществляется возврат по рёбрам графа, которые достижимы из текущей вершины стека и помечены символами из V. Пусть есть обратное ребро из вершины v в вершину u с символом c, тогда необходимо вычислить следующее состояние РДКА, соответствующее переходу из r_0 по символу c, то есть вычислить значение $\delta_{dr}(r_0, c)$. Аналогичным образом осуществляется возврат из следующих вершин графа вместе с вычислением состояний РДКА. На каком-то шаге возврата по рёбрам стека РДКА окажется в конечном состоянии. Таким образом часть стека для свёртки будет распознана. Далее, как и в классическом RNGLR алгоритме, создаётся новая вершина (если её не было) в графе GSS и выходящее из неё ребро в ту вершину, на которой закончилось распознавание, а распознанная це-

почка добавляется в SPPF.

Проблемой такого подхода является возможность распознания лишь части цепочки со стека, которая подлежит свёртке. Рассмотрим следующую грамматику:

```
s : X*
```

Тогда РДКА будет состоять из одного состояния, которое будет являться начальным и конечным одновременно. Функция переходов будет иметь одно правило: $\delta_{rd}(0,\text{X}) = \text{X}$. Если входная цепочка будет иметь вид XXX, то после применения свёртки на любом шаге работы алгоритма будет заменяться лишь один X, а не вся цепочка, что никогда не приведёт к корректному распознаванию.

Если же пытаться распознавать часть стека до тех пор, пока РДКА не будет иметь переход по следующему символу, то в таком случае распознанная цепочка может оказаться длиннее, чем это необходимо для корректного завершения анализа.

Примером данной ситуации может служить следующая грамматика:

```
s : a        (1)
a : A b D    (2)
b : A* X Y   (3)
```

Если входная цепочка будет иметь вид AXYD, то после прочтения первых трёх символов, будет вызвана свёртка по продукции (3), после чего входная цепочка будет иметь вид bD. Распознавание закончится с ошибкой, хотя входная цепочка и принадлежит грамматике.

Для решения данной проблемы необходимо рассматривать все возможные варианты и складывать их на стек. Если РДКА принимает строку, то применяется свёртка, как это происходит в классическом RNGLR алгоритме, и создание новых вершин на стеке при необходимости. Однако движение по стеку в обратном направлении не прекращается. Дальнейшее движение осуществляется для того, чтобы определить все оставшиеся цепочки, которые может принять РДКА. На каждую принятую цепочку будут осуществляться классическая свёртка. Работа по совершению свёртки, соответствующая шестёрке (v, X, q_0, r_0, f, y),

заканчивается тогда, когда дальнейшее движение по стеку невозможно, то есть РДКА не может распознать следующий символ при движении по графу GSS в обратном направлении. Таким образом на стеке будут лежать всевозможные варианты свёрток, которые могли бы быть произведены в ходе анализа.

Множество $\mathcal{X}$ используется для накопления всех возможных путей в графе GSS, по которым будет произведена свёртка. Множество $\mathcal{S}$ содержит пары вида $(v, path)$, где v – вершина в GSS графе, $path$ – путь в GSS графе, по которому была достигнута вершина v. Если текущая вершина стека v_c, а путь к ней $path_c$, то происходит поиск пары $(v_c, path_c)$ во множестве $\mathcal{S}$. Если данной пары во множестве нет, то это означает, что такой вариант ещё не рассматривался, а значит происходит дальнейшее движение по стеку при возможности. Если же такая пара во множестве есть, это означает, что такой вариант уже был рассмотрен, а значит не следует продолжать движение по стеку.

SPPF строится аналогично классическому RNGLR алгоритму с модификацией для подсчёта семантики. Аналогично определяется и ϵ-SPPF для цепочки γ при условии, что существует вывод $\gamma \overset{*}{\Rightarrow} \epsilon$ продукции вида $A ::= \alpha\gamma$ при $|\gamma| > 1$, где $\alpha \neq \epsilon$. Такие цепочки γ называются нулевой частью. Если $A ::= \gamma$, то ϵ-SPPF определяется для A. I – является индексирующей функцией, которая назначает номера, начиная с 1, все нетерминалам X таким, что $X \overset{*}{\Rightarrow} \epsilon$ и нулевым частям γ. Функция $I(\epsilon) = 0$. Узел ϵ-SPPF дерева для цепочки ω будет помечаться как $x_{I(\omega)}$, для ϵ узел будет помечен как x_0.

При добавлении некоторой цепочки в SPPF дерево, по данной цепочке при помощи НКА строится дерево для подсчёта семантики, после чего данное дерево добавляется в SPPF. Таким образом SPPF подвергается определённой модификации. Пусть есть цепочка, которая является порождением некоторой продукции с нетерминалом A в правой части. По данной цепочке при помощи НКА можно построить дерево для подсчёта семантики, корнем которого является узел $root$ с своим набором функций. Тогда в SPPF к узлу с символом A будет подвешено дерево для подсчёта семантики за узел $root$.

Ниже представлен псевдокод для алгоритма по работе интерпретатора парсеров.

Вход: RN-таблица $\mathcal{T}$, входная цепочка $a_1..a_d$, ϵ-SPPF деревья для нетерминалов, из которых выводится пустая цепочка, и для нулевых частей ω, корневые узлы для ϵ-SPPF деревьев, помеченные как $x_{I(\omega)}$

Algorithm 1 Parser algorithm

1: **function** Parser(i)
2:     **if** d = 0 **then**
3:         **if** $accept \in \mathcal{T}(0, \$)$ **then**
4:             report $Success$
5:         **else**
6:             report $Fail$
7:         **end if**
8:     **else**
9:         create a node v_0 with label 0, start state of LR-analyzer
10:         let $U_0 \leftarrow \{v_0\}$
11:         let $\mathcal{R}, \mathcal{Q} \leftarrow \varnothing$
12:         let $a_{d+1} \leftarrow \$$
13:         let $U_1, .., U_d \leftarrow \varnothing$
14:         **if** $pk \in \mathcal{T}(0, a_1)$ **then**
15:             $\mathcal{Q}.Enqueue(v_0, k)$
16:         **end if**
17:         **for all** $r(X, q_0, r_0, f) \in \mathcal{T}(0, a_1)$ **do**
18:             $\mathcal{R}.Enqueue(v_0, X, q_0, r_0, f, \epsilon)$
19:         **end for**
20:         **for** $i = 0$ to d **do**
21:             **while** $U_i \neq \varnothing$ **do**
22:                 $\mathcal{N} \leftarrow \varnothing$
23:                 **while** $\mathcal{R} \neq \varnothing$ **do**
24:                     Reducer(i)
25:                 **end while**
26:                 Shifter(i)
27:             **end while**
28:         **end for**
29:         **if** LR-analyzer accepting state l labels an element t $\in U_d$ **then**
30:             let root-node be the SPPF node that labels the edge $(t, v0)$ in the GSS
31:             remove nodes in the SPPF not reachable from root-node
32:             report $Success$
33:         **else**
34:             report $Fail$
35:         **end if**
36:     **end if**
37: **end function**

Algorithm 2 Get paths for reduce

```
1: function GetPaths(v, r, path, S, X)
2:     let D ← pairs (node, label) which can be reached from v of length 1
3:     for all p ∈ D do
4:         if ∃rn = δ_dr(r, p.label.symbol) then
5:             path ← path + p.node
6:             if rn ∈ F_dr then
7:                 X.Enqueue(path)
8:             end if
9:             if (p.node, path) ∉ S then
10:                S.Enqueue(p.node, path)
11:                GetPaths(p.node, rn, path, S, X)
12:            end if
13:        end if
14:    end for
15:    if r ∈ F_dr then X.Enqueue(ϵ)
16:    end if
17: end function
```

Algorithm 3 Reduce algorithm

```
1: function Reducer(i)
2:     R.Dequeue(v, X, q, r, f, y)
3:     let 𝒳, 𝒮 = ∅
4:     GetPaths(v, r, 𝒮, 𝒳)
5:     if r ∉ F_dr then
6:         y_m ← y
7:     end if
8:     for all path ∈ 𝒳 do
9:         let y_{m−1}, .., y_1 be the edge labels on the path
10:        let u be the final node on the path
11:        let k ← label of u
12:        let pl ← 𝒯(k, X)
13:        if r ∈ F_dr then
14:            z ← x_f
15:        else
16:            suppose z ∈ U_c
17:            if ∄z ∈ 𝒩 with label (X, c) then
18:                create z with label (X, c)
19:                𝒩.Enqueue(z)
20:            end if
21:        end if
22:        if ∃w ∈ U_i with label l then
23:            if there is no edge from w to u then
24:                create an edge from w to u with label z
25:                if r ∉ F_dr then
26:                    for all r(B, q_0, r_0, f) ∈ 𝒯(l, a_{i+1}) where r_0 ∉ F_dr do
27:                        R.Enqueue(u, B, q_0, r_0, f, z)
28:                    end for
29:                end if
30:            end if
31:        else
32:            create a node w ∈ U_i with label l
33:            create an edge(w, u) with label z
34:            if ph ∈ 𝒯(l, a_{i+1}) then
35:                (Q).Enqueue(w, h)
36:            end if
37:            for all r(B, q_0, r_0, f) ∈ 𝒯(l, a_{i+1}) where r_0 ∈ F_dr do
38:                R.Enqueue(u, B, q_0, r_0, f, ϵ)
39:            end for
40:            if r ∉ F_dr then
41:                for all r(B, r_0, f) ∈ 𝒯(l, a_{i+1}) where r_0 ∉ F_dr do
42:                    R.Enqueue(u, B, q_0, r_0, f, z)
43:                end for
44:            end if
45:        end if
46:        if r ∉ F_dr then
47:            AddChildren(q_0, z, y_1, .., y_m, f)
48:        end if
49:    end for
50: end function
```

Algorithm 4 Shift algorithm

```
1: function Shifter(i)
2:     if i ≠ d then
3:         let Q'
4:         create a new SPPF node z with label (a_{i+1}, i)
5:         while Q ≠ ∅ do
6:             Q.Dequeue(v, k)
7:             if w ∈ U_{i+1} with label k then
8:                 create an edge from w to v with label z
9:                 for all r(B, q_0, r_0, f) ∈ T(k, a_{i+2}) where r_0 ∉ F_dr do
10:                    R.Enqueue(v, B, q_0, r_0, f, z)
11:                end for
12:            else
13:                create a node w ∈ U_{i+1} with label k
14:                create an edge (w, v) with label z
15:                if ph ∈ T(k, a_{i+2}) then
16:                    Q'.Enqueue(w, h)
17:                end if
18:                for all r(B, q_0, r_0, f) ∈ T(k, a_{i+2}) where r_0 ∉ F_dr do
19:                    R.Enqueue(v, B, q_0, r_0, f, z)
20:                end for
21:                for all r(B, q_0, r_0, f) ∈ T(k, a_{i+2}) where r_0 ∈ F_dr do
22:                    R.Enqueue(w, B, q_0, r_0, f, ϵ)
23:                end for
24:            end if
25:        end while
26:        copy Q' into Q
27:    end if
28: end function
```

Algorithm 5 Add children

```
1: function AddChildren(q_0, y, y1, .., y_m, f)
2:     if f = 0 then
3:         let Λ ← (y_1, .., y_m)
4:     else
5:         let Λ ← (y_1, .., y_m, x_f)
6:     end if
7:     let root ← GetSemanticTree(q_0, Λ)
8:      create a new edge from y to root
9: end function
```

Algorithm 6 Path greedy

```
1: function Greedy(path_u, path_q)
2:     let u_1, .., u_n ← states in path_u
3:     let q_1, .., q_m ← states in path_q
4:     let w ← the less of n and m
5:     for i = 1 to w do
6:         if u_i < q_i then return true
7:         else
8:             if u_i > q_i then return false
9:             end if
10:        end if
11:    end for
12:    return true
13: end function
```

Algorithm 7 ϵ-closure

```
1: function EpsilonClosure(Q, k, l)
2:     let Q_next ← ∅
3:     while Q ≠ ∅ do
4:         letq ← Q.Dequeue()
5:         if q ∈ F and k ≠ Λ.length then
6:             continue
7:         end if
8:         let E ← all edges from q
9:         if ∃e ∈ E where e.label ≠ ϵ then
10:            Q_next.Enqueue(q)
11:        else
12:            for all e ∈ E do
13:                let u ← e.distanation
14:                if ∃(path_u, tree_u, eps_u) for k then
15:                    if u ∈ eps_q or Greedy(path_u, path_q) then
16:                        break
17:                    end if
18:                end if
19:                Q.Enqueue(u)
20:                let path_u ← path_q + u for k
21:                let tree_u ← tree_q for k
22:                let eps_u ← eps_q ∪ {q} for k
23:                if e has label with start semantic then
24:                    create child of tree_q
25:                    let tree_u ← pointer to this child for k
26:                end if
27:                if e has label with end semantic and set of functions s then
28:                    label tree_q with s
29:                    let tree_u ← pointer to the parent of tree_q for k
30:                end if
31:            end for
32:        end if
33:    end while
34:    return Q_next
35: end function
```

Algorithm 8 Step not ϵ

```
1: function Step(Q, k, c)
2:     let Q_next ← ∅
3:     for all q ∈ Q do
4:         let E ← all edges from q
5:         for all e ∈ E where e.label == c do
6:             let u ← e.distanation
7:             Q_next.Enqueue(u)
8:             path_u ← path_q + u for k
9:             label tree_q with c
10:            let tree_U ← pointer to the parent of tree_q for k
11:            let eps_u ← ∅ for k
12:        end for
13:    end for
14:    return Q_next
15: end function
```

Algorithm 9 NFA interpreter

```
1: function GetSemanticTree(q_0, Λ)
2:     let l ← Λ.length
3:     let Q_e ← EpsilonClosure({q_0}, 0, l)
4:     for i = 1 to Λ.length do
5:         let Q_s ← Step(Q_s, i, Λ[i])
6:         let Q_e ← EpsilonClosure(Q_s, i, l)
7:     end for
8:     let f ← final state of NFA
9:     return tree_f for l
10: end function
```

4. Детали реализации

Генератор синтаксических анализаторов и интерпретатор были реализованы в рамках проекта YaccConstructor. YaccConstructor является проектом по разработке инструмента для генерации лексических и синтаксических анализаторов и обработки грамматик. Данный инструмент разработан на языке программирования F#. Как следствие, представленный алгоритм также был разработан на языке программирования F#.

В 2014 году в проекте YaccConstructor был разработан и реализован алгоритм RNGLR.EBNF [6]. Реализация алгоритма, представленного в данной работе, является модификацией алгоритма RNGLR.EBNF.

Модификации подверглись:

- генератор анализаторов: добавлено построение НКА и РДКА, изменена RN-таблицы переходов;
- интерпретатор парсеров: алгоритм работы со стеком.

5. Апробация

Разработанный алгоритм был протестирован на ряде синтетических грамматик.

Сравнение генераторов синтаксических анализаторов проведено по критерию времени работы интерпретатора парсеров на фиксированных входных строках исходной грамматики.

Для сравнения показателей времени работы были взяты алгоритмы, реализованные в рамках проекта YaccConstructor [7] (далее RNGLR) и [6] (далее RNGLR.EBNF), поскольку алгоритм (далее RNGLR.EBNF.DFA), описанный в данной работе, является модификацией как алгоритма RNGLR, так и алгоритма RNGLR.EBNF.

Алгоритм RNGLR позволяет работать с грамматиками в EBNF, но с преобразованием исходной формы грамматики в BNF. Целесообразно сравнить время работы интерпретатора, работающего с парсером, построенным по преобразованной грамматике, со временем работы интерпретатора, работающего с парсером, построенным по исходной грамматике. Данное сравнение позволяет выяснить, как меняется время работы интерпретатора без использования преобразований исходной грамматики.

Алгоритм RNGLR.EBNF работает с грамматиками в EBNF напрямую, но использует другое решение при работе со стеком GSS. В данном случае интерес представляет сравнение времени работы интерпретаторов, использующих парсеры, построенные по исходной грамматике без преобразований, но имеющих разные технологии при распознавании входной цепочки. Данное сравнение позволяет выяснить, какой из алгоритмов имеет лучшее время работы.

Для сравнения была взята грамматика вида:

```
s : X* X X?
```

Входные цепочки представляют собой файлы с фиксированным количеством символов от 50000 до 1000000.

Алгоритм RNGLR.EBNF при заданном диапазоне количества входных символов работает некорректно, поскольку имеет исключение вида

переполнение стека.

Ниже приведён график результатов сравнения алгоритмов RNGLR и RNGLR.EBNF.DFA.

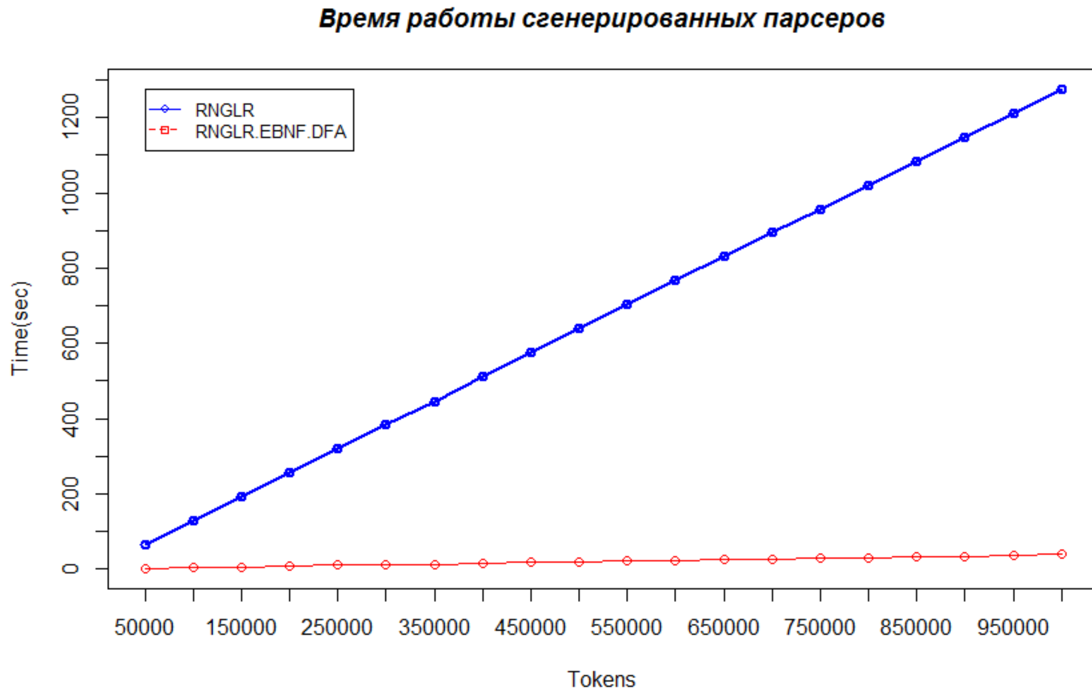

Рис. 1

Данные результаты сравнения показывают, что разработанный алгоритм даёт лучшую производительность, чем существующие решения. При этом алгоритм выигрывает по времени у решения, преобразующего исходную форму грамматики, и не имеет исключения по памяти в сравнении с алгоритмом, работающим с грамматикой в EBNF напрямую.

Заключение

В ходе данной работы были получены следующие результаты:

- разработан алгоритм; за основу были взяты алгоритмы [5] и [4]; была проведена модификация RN-таблиц синтаксического анализа и работы стека интерпретатора;
- алгоритм был реализован в рамках проекта YaccConstructor;
- проведена апробация; сравнение времени работы интерпретатора, построенного на основе разработанного алгоритма, с временем работы интерпретатора, работающего с парсерами, построенными по преобразованной грамматики, показало, что разработанный алгоритм может существенно уменьшить время работы интерпретатора парсеров; при этом интерпретатор, построенный на разработанном алгоритме, имеет меньшие ограничения по количеству символов входной цепочки в сравнении с интерпретатором с иной реализацией работы стека, работающим с парсерами, построенными по исходной грамматике.

Список литературы

[1] Economopoulos Giorgios Robert. Generalised LR parsing algorithms. — 2006.

[2] K. Hemeric. Towards a Taxonomy for ECFG and RRPG Parsing // LNCS. — Vol. 5457. — P. 410 – 421.

[3] Morimoto Shin-ichi, Sassa Masataka. Yet another Generation of LALR Parsers for Regular Right Part Grammar // Acta Informatica. — 2001. — Vol. 37, no. 15. — P. 671 – 697.

[4] Purdom Paul Walton, Brown Cynthia A. Parsing Extended $LR(k)$ Grammars // Acta Informatica. — 1981. — no. 15. — P. 115 – 127.

[5] Scott Elizabeth, Johnstone Adrian. Right Null GLR Parsers // ACM Transactions on Programming Languages and Systems. — 2006. — Vol. 28, no. 4. — P. 577 – 618.

[6] А.А. Алефиров. Непосредственная поддержка грамматик в расширенной форме Бэкуса-Наура в генераторах синтаксических анализаторов // Дипломная работа. СПбГУ. — 2014.

[7] Д.А. Авдюхин. Создание генератора GLR трансляторов для .NET // Курсовая. СПбГУ. — 2012.

[8] Дж.Э. Хопкрофт, Р. Мотвани, Дж.Д. Ульман. Введение в теорию автоматов, языков и вычислений. — Издательский дом "Вильямс", 2002.

[9] Компиляторы: принципы, технологии и инструментарий / Ахо А.В., Лам М.С., Сети Р., Ульман Д.Д. — Издательский дом "Вильямс", 2008.

[10] С.В. Григорьев. Генератор синтаксических анализаторов для неоднозначных контексто-свободных грамматик // Дипломная работа. СПбГУ. — 2010.

[11] Сайт проекта Bison. — URL: http://www.gnu.org/software/bison/.

[12] Сайт проекта Jison. — URL: http://zaach.github.io/jison/.

[13] Сайт проекта Sglr. — URL: http://www.meta-environment.org/.

[14] Сайт проекта YaccConstructor. — URL: https://github.com/YaccConstructor/YaccConstructor.

[15] Статья в сети интернет про проект Yacc. — URL: http://dinosaur.compilertools.net/yacc/.

[16] Страница в сети интернет, посвящённая Ocamlyacc. — URL: http://caml.inria.fr/pub/docs/manual-ocaml-400/manual026.html.