САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра Системного Программирования

Рагозина Анастасия Константиновна

Сравнение алгоритмов обобщенного восходящего и нисходящего синтаксического анализа

Бакалаврская работа

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

Научный руководитель:
магистр информационных технологий, ст. преп. Григорьев С. В.

Рецензент:
инженер ООО “ИнтеллиДжей Лабс” Беляков А. М.

Санкт-Петербург
2014

SAINT-PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Chair of System Programming

Ragozina Anastasiia

Comparison of generalized ascent and descent parsers

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific supervisor:
master of Information Technology, sen. lector Semyon Grigoriev

Reviewer:
engineer IntelliJ Labs Andrey Belyakov

Saint-Petersburg
2014

Оглавление

Введение **4**

1. Обзор предметной области **6**

2. Постановка задачи **13**

3. Модуль генератора **14**

4. Модуль интерпретатора **16**

4.1. Модификации алгоритма GLL 16

4.2. Тестирование 17

5. Анализ и сравнение **21**

Заключение **22**

Введение

Одной из важных задач, возникающих в процессе автоматического реинжиниринга программного обеспечения, является создание синтаксических анализаторов [1] языков программирования. Синтаксический анализ может использоваться для перевода исходной системы на другой язык программирования, анализа кода и других задач.

Синтаксические анализаторы можно разделить на два класса — нисходящие и восходящие. Нисходящие парсеры привлекательны тем, что их структура полностью соответствует структуре грамматики, что весьма упрощает процесс их написания и отладки. К сожалению, класс грамматик, которые допускают нисходящие анализаторы является весьма ограниченным. На языки, которые могут быть обработаны LL-анализаторами [2] накладываются жёсткие ограничения: любая LL(k)-грамматика должна быть однозначной. Леворекурсивные грамматики не принадлежат классу LL(k) ни для какого k. Иногда удается преобразовать не LL-грамматику в эквивалентную ей LL-грамматику с помощью факторизации и устранения левой рекурсии. Однако проблема существования эквивалентной LL(k)-грамматики для произвольной не LL(k)-грамматики неразрешима[3]. Можно использовать backtracking [4] методы для расширения класса обрабатываемых языков, но даже это не поможет справиться с проблемой левой рекурсии. В свою очередь восходящие LR-анализаторы [2] позволяют обрабатывать более широкий класс грамматик, но не имеют такой тесной связи с грамматикой. Такие анализаторы позволяют работать с леворекурсивными грамматиками, но не могут обрабатывать скрытую левую рекурсию. Кроме того производительность таких анализаторов часто ниже, чем у парсеров, построенных с использованием нисходящих алгоритмов, а размер управляющих таблиц, используемых в процессе работы, может экспоненциально зависеть от размера грамматики [5].

Для того чтобы расширить класс языков, обрабатываемых нисходящими анализаторами, можно использовать обобщенные алгоритмы синтаксического анализа — Generalised LL (GLL) [6] и Generalised LR (GLR). Эти алгоритмы предназначены для разбора по недетерминированным и неоднозначным грамматикам и позволяют обрабатывать все контекстно-свободные грамматики, в том числе и леворекурсивные.

Алгоритм обобщённого восходящего анализа является расширением алгоритма LR-парсера, впервые описан Томитой [11]. Принцип работы похож на принцип работы обычного LR-анализатора, за исключением того, что для конкретной грамматики GLR-парсер обрабатывает все возможные трактовки входной последовательности, используя поиск в ширину. Генераторы GLR-парсеров преобразуют исходную грамматику в таблицы парсера, точно так же, как и генераторы LR-парсеров. Но, тогда как таблицы LR-парсера допускают только один переход состояния (определенное исходным состоянием парсера и входным терминальным символом), таблицы GLR-

парсера допускают множество результирующих состояний. В результате алгоритм допускает конфликты сдвиг/свертка (“shift/reduce”) и свертка/свертка (“reduce/reduce”). Когда возникает конфликт, стек парсера (магазинная память) разветвляется на два или больше параллельных стека, верхние состояния которых соответствуют каждому возможному переходу. После этого следующий входной символ используется, чтобы определить дальнейшие переходы на верхних состояниях каждой ветви стека. При этом опять может возникнуть необходимость разветвления стека. Если же для какого-либо верхнего состояния и входного символа в таблице парсера не существует ни одного перехода, то эта ветвь стека считается ошибочной и отбрасывается. Алгоритм GLR в худшем случае имеет такую же сложность, как алгоритм Кока — Янгера — Касами [13] и алгоритм Эрли [12] — $O(n^3)$.

Принцип работы алгоритма обобщённого нисходящего анализатора схож с принципом работы обобщённого восходящего анализатора: в процессе работы парсера рассматриваются все возможные варианты разбора и в случае конфликта происходит разветвление стека. Такой алгоритм также в худшем случае работает за кубическое время, а для LL-грамматик [2] — за линейное. Алгоритм позволяет создавать парсеры, которые могут быть легко написаны и отлажены вручную из-за своей структуры, близкой к структуре грамматики. Синтаксические анализаторы, построенные с помощью такого алгоритма, позволяют бороться с проблемой левой рекурсии, как скрытой, так и обычной, значительно расширяя класс обрабатываемых нисходящими синтаксическими анализаторами языков.

1. Обзор предметной области

Контекстно-свободная грамматика G состоит из множества нетерминальных символов N, множества терминальных символов T, стартового нетерминала s ∈ N и набора правил вида: a := α, где a ∈ N и α последовательность символов из (T ∪ N)∗.

Рекурсивные LL(1)-анализаторы состоят из набора функций, по одной для каждого нетерминала. Альтернатива α из правила для a, по которой будет производиться разбор, выбирается в соответствии с текущим символом во входном потоке, и после этого вызывается функция для разбора α. Для определения того, какое именно правило вызывать, используются таблицы предиктивного анализа [2], построение которых осуществляется с помощью множеств FIRST и FOLLOW. Эти множества строятся по грамматике. В процессе нисходящего синтаксического анализа они позволяют выбрать применяемую продукцию на основании очередного символа из входного потока. Множество FIRST для нетерминала есть множество терминалов, с которых может начинаться данный нетерминал, а множество FOLLOW — множество терминалов, которые могут следовать за данным нетерминалом.

Рассмотрим грамматику H_0:

$$\begin{aligned} \mathrm{s} &:= \mathrm{a} + \mathrm{b} \mid \mathrm{b} \\ \mathrm{a} &:= \mathrm{A} \\ \mathrm{b} &:= \mathrm{B} \end{aligned}$$

Для такой грамматики множества FIRST будут выглядеть следующим образом: FIRST(s) = {A, B}, FIRST(a) = {A}, FIRST(b) = {B}. А множество FOLLOW выглядит так: FOLLOW(s) = {$}, FOLLOW(a) = {+}, FOLLOW(b) = {$}, где $ означает конец входного потока. Далее информация из этих множеств собирается в таблицу предиктивного анализа M, строки которой помечены нетерминалами, а столбцы терминалами или маркером конца входной потока $. Алгоритм построения таблицы основан на следующей идее: если очередной входной символ a находится во множестве FIRST(α), то выбирается продукция a := α. Если же $\alpha = \epsilon$ или $\alpha :=^* \epsilon$, то правило a := α выбирается, если текущий входной символ принадлежит множеству FOLLOW(a) или из входного потока получен $, который при этом входит в FOLLOW(a).

Управляющая таблица для грамматики H_0 изображена на Рис. 1:

	A	B	+	$
s	s := a + b	s := b	error	error
a	a := A	error	error	error
b	error	b := B	error	error

Рис. 1: Управляющая таблица для грамматики H_0.

В ячейках таблицы хранятся продукции, по которым будет происходить разворачивание нетерминала. Возможны ситуации, когда таблица может иметь ячейки с несколькими записями. Например, если грамматика неоднозначная или леворекурсивная, тогда M будет содержать минимум одну запись с несколькими продукциями. Как упоминалось ранее, хотя устранение левой рекурсии и левая факторизация не являются сложными задачами, существуют грамматики, для которых не существует аналогичной LL(1) - грамматики.

Для работы с такими грамматиками используются обобщённые алгоритмы синтаксического анализа, о которых упоминалось ранее. Обобщенный нисходящий анализатор похож на LL(1)-анализатор, но с добавлением механизма для корректной работы с неоднозначностями и недерминизмом. Эта проблема решается за счёт замены вызова функций на независимые операции на стеке вызовов. Для каждого нетерминала создаётся функция, которая помечена соответствующей меткой. В местах возникновения неоднозначностей некоторые инструкции goto могут иметь несколько целевых меток, что соответствует возможности выбора нескольких продукций для текущего входного символа. Для того чтобы обрабатывать подобные ситуации используются дескрипторы, с помощью которых запоминаются все возможные варианты разбора. После этого разбор возобновляется с точки, записанной в следующем дескрипторе. Такой дескриптор содержит в себе информацию, полностью описывающую текущую точку разбора: указатель в грамматике, указатель во входном потоке, вершину стека и текущую вершину дерева, которое строится в процессе синтаксического анализа. Для хранения всех дескрипторов используется очередь. В местах недетерминизма создаются новые дескрипторы с использованием текущего входного символа, текущего слота, вершины стека и дерева. Для некоторых грамматик количество дескрипторов, которые создаются в процессе разбора, может экспоненциально зависеть от размера входа. При этом если для каждого такого дескриптора отдельно хранить стек, то это может привести к чрезмерному потреблению памяти парсером. Для решения этой проблемы стеки комбинируются с использованием структуры graph structured stack (GSS) [15], которая позволяет хранить только вершину стека.

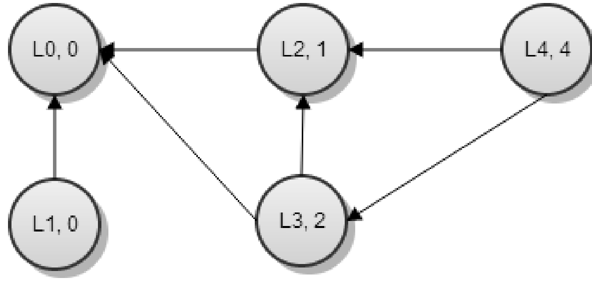

Рис. 2: Стек представленный в виде графа.

Например, граф на Рис. 2 представляет стеки вида: {(L0, 0), (L2, 1), (L1, 2)}, {(L0, 0), (L2, 1), (L1, 2)}, {(L0, 0), (L3, 1), (L1, 2)}, {(L0, 0), (L2, 1),(L3, 1), (L1, 2)}, {(L0, 0), (L3, 1)}.

Техника GLL основывается на идее обхода грамматики с использованием входной строки. На каждом шаге работы алгоритма хранится два указателя: в грамматике (слот) и во входном потоке. Слоты по сути своей похожи на item в LR(0) анализаторах и могут быть описаны следующим образом $x := x1 \ldots xi \bullet xi{+}1 \ldots xq$, где $\bullet$ указывает позицию перед символом $xi{+}1$. На каждом этапе разбора имеется слот вида $x := \alpha \bullet X\beta$ или $x ::= \alpha\bullet$ и позиция во входном потоке i. В процессе разбора возможны следующие ситуации.

- Если текущий символ в грамматике является терминалом X и совпадает с текущим символом во входном потоке, то указатель в грамматике нужно сдвинуть на одну позицию вправо, $x := \alpha X \bullet \beta$, и увеличить указатель во входном потоке на единицу. Никаких дополнительных действий со стеком при этом не производится. Иначе если терминал X не совпадает с текущим символом во входном потоке, то текущая ветка разбора считается ошибочной, отбрасывается и разбор продолжается с использованием следующего дескриптора.

- Если x — это нетерминал a, то необходимо в стек записать слот, по которому продолжить разбор после того, как правило для a будет разобрано. Указатель в грамматике перемещается на $a := \bullet\gamma$, а указатель во входном потоке остаётся без изменений.

- Если указатель в грамматике имеет следующий вид $x := \alpha\bullet$ и стек не пуст, то слот $y := \delta x \bullet \mu$, который хранится на вершине стека, извлекается и становится текущим.

- Если текущий слот имеет вид $s := \tau\bullet$ и весь входной поток рассмотрен, то возвращается дерево, построенное в процессе разбора, иначе разбор заканчивается ошибкой.

Результатом работы синтаксического анализатора является абстрактное синтаксическое дерево, у которого корень помечен стартовым нетерминалом, а листья — терминалами. В алгоритме обобщённого нисходящего анализа для построение абстрактного дерева используется специальная структура shared packed parse forests (SPPF) [14], которая позволяет хранить деревья более компактно. Эта необходимость возникает из-за неоднозначностей в грамматике, которые приводят к тому, что для одной и той же подстроки можно построить несколько деревьев, в которых одни и те же ячейки будут храниться несколько раз. Это может привести к тому, что в результате разбора количество деревьев будет расти экспоненциально. Этого можно избежать за счёт комбинирования всех деревьев в одну структуру — SPPF. В SPPF ячейки, которые ссылаются на одинаковые поддеревья, разделяются, а ячейки, которые соответствуют различному выводу одной и той же строки из одного и того же нетерминала, комбинируются с использованием "packed node". Структура такого дерева отображена на Рис. 3.

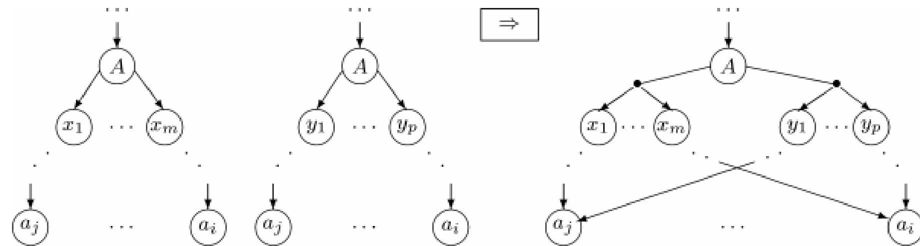

Рис. 3: Shared packed parse forest.

Для автоматического создания синтаксических анализаторов существует несколько подходов: можно полностью генерировать весь код парсера по грамматике, а потом использовать его. Схема работы генератора при использовании данного подхода отображена на Рис. 4.

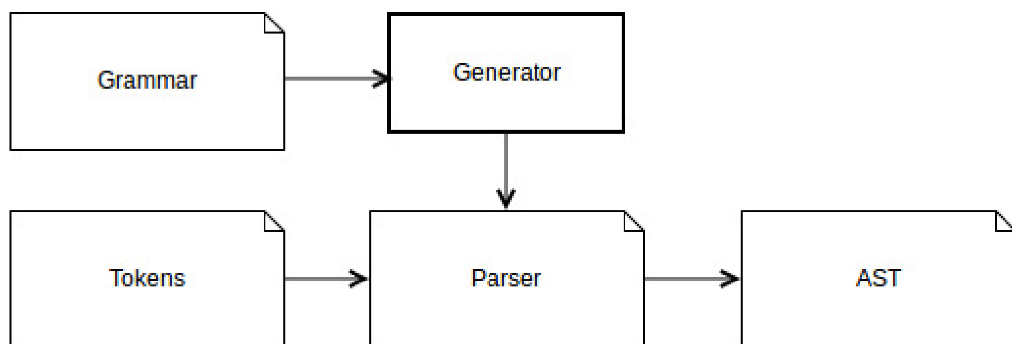

Рис. 4: Схема работы генератора синтаксических анализаторов при генерации всего кода парсера.

Чаще всего такой подход используется при генерации анализаторов, построенных

методом рекурсивного спуска. При этом для каждого правила грамматики генерируются функции, которые последовательно вызываются в процессе работы парсера. Чаще всего такие анализаторы пишутся вручную, что обусловлено их простотой и прямолинейностью, но существуют инструменты для автоматической генерации таких парсеров. Например, инструмент ANTLR [16] — генератор парсеров, позволяющий автоматически создавать парсеры на одном из целевых языков программирования по описанию LL(*)-грамматики на языке, близком к EBNF [2].

При другом подходе к автоматическому созданию парсеров генерируется только дополнительная необходимая для работы синтаксического анализатора информация, которая используется интерпретатором, содержащим в себе основную логику алгоритма. Чаще всего в качестве дополнительной информации генерируются управляющие таблицы, описанные выше, и информация о грамматике. Схема работы такого генератора отображена на Рис. 5.

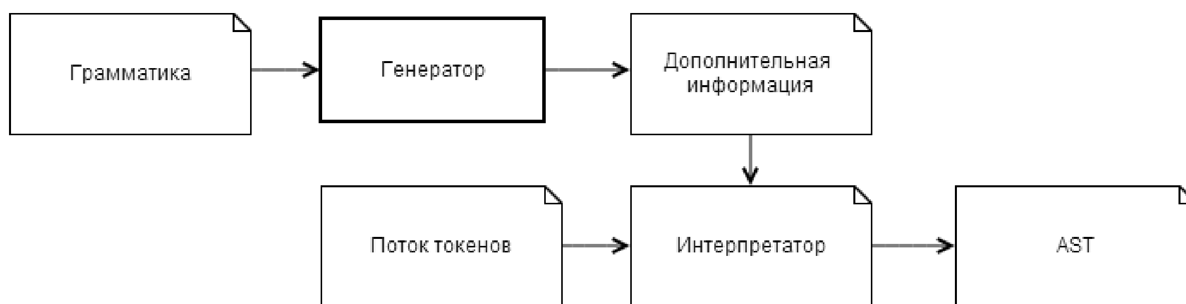

Рис. 5: Схема работы генератора синтаксических анализаторов при генерации только дополнительной информации и использовании интерпретатора.

Как упоминалось ранее, в реинжиниринге программного обеспечения важную роль играет синтаксический анализ. Современные инструменты позволяют генерировать синтаксические анализаторы по спецификации трансляции, но проекты, направленные на создание инструментов для автоматизации реинжиниринга программного обеспечения, выдвигают особые требования к генераторам синтаксических анализаторов [7]. Часто системы, подлежащие реинжинирингу, написаны на языках программирования, которые существовали ещё на заре развития теории синтаксически управляемой трансляции. По этой причине создание синтаксических анализаторов таких языков с помощью современных инструментов практически невозможно. Кроме того генераторы синтаксических анализаторов для реинжиниринга должны обрабатывать широкий класс языков, позволять разрешать неоднозначности в грамматике и порождать парсеры с высокой скоростью работы и хорошим механизмом восстановления после ошибок. Ещё одним важным свойством, значительно влияющим на процесс разработки синтаксического анализатора языка, является простота и удобство входного языка генератора. Важно, чтобы грамматики, написанные на этом языке, были

просты в сопровождении и позволяли создавать новые парсеры быстро и удобно.

Этими свойствами обладает генератор синтаксических анализаторов для нужд автоматического реинжииринга программного обеспечения YaccConstructor (YC) [8], разрабатываемый на кафедре системного программирования Санкт-Петербургского государственного университета. Модульность инструмента позволяет создавать парсеры, используя различные языки спецификации трансляции и алгоритмы синтаксического анализа. Также инструмент поддерживает свой собственный язык описания трансляций — Yard, который позволяет использовать расширенную форму Бэкуса-Наура [2], которая отличается от форм Бэкуса-Наура (EBNF) более «ёмкими» конструкциями, позволяющими при той же выразительной способности упростить и сократить в объёме описание грамматики. Кроме того язык Yard позволяет создавать правила с параметрами и использовать специальные конструкции для разрешения неоднозначностей.

В процессе реинжиниринга грамматика часто подвергается изменениям, которые могут делать её неоднозначной и приводить к конфликтам, обработать которые возможно, используя восходящий алгоритм синтаксического анализа. Для обеспечения возможности работы с неоднозначными грамматиками в рамках проекта реализован GLR-генератор, порождающий восходящие парсеры с использованием алгоритма RNGLR [6], работающий со всеми контекстно-свободными грамматиками. Так же реализован алгоритм восстановления после ошибок, предоставляющий информацию необходимую для диагностики ошибок, и механизм, предоставляющий информацию о конфликтах.

Кроме того в рамках проекта ведётся активная работа над абстрактным анализом встроенных языков. В отличие от классического анализа в абстрактном на вход анализатору подаётся не линейная последовательность символов, а некая структура, например граф. Данный подход позволяет производить статический анализ встроенных запросов. Например, если в коде встречается SQL-запрос, который собирается из строки, то обнаружить ошибку в таком запросе можно только в момент выполнения программы, что значительно замедляет процесс разработки и усложняет отладку. Также было бы удобно иметь возможность автодополнения и подсветки в таких запросах. Для работы с такими запросами и используется абстрактный анализ.

Для создания абстрактных анализаторов чаще всего используется табличный подход. В статье [17] описан пример для создание табличного абстрактного анализатора на основе LR-таблиц. При этом не накладываются ограничения, из-за которых невозможно использовать LL-таблицы. В статье [18] описано решение, позволяющее создать абстрактный анализатор на основе обобщённого алгоритма RNGLR. В алгоритмах RNGLR [9] и GLL [18] процесс построения GSS и SPPF очень похож. По этой причине получение абстрактного анализатора на основе алгоритма GLL представляется возможным. Его использование обладает следующими преимуществами:

- высокая скорость работы для леворекурсивных грамматик;
- возможность более качественной и простой диагностики и восстановления от ошибок.

Именно это является главной причиной, по которой необходимо создать табличный анализатор с использованием алгоритма обобщённого нисходящего синтаксического анализа.

2. Постановка задачи

Целью данной работы является сравнение алгоритмов обобщенного восходящего и нисходящего синтаксического анализа. Для её осуществления были поставлены следующие задачи:

- реализация модуля генератора парсеров в среде YC, необходимо использовать подход, позволяющий генерировать только таблицы для того, чтобы в дальнейшем можно было получить абстрактный анализатор;
- реализация модуля интерпретатора с использованием алгоритма GLL, результатом работы которого является структура, содержащая все деревья разбора входной строки (в среде YC);
- сравнение производительности обобщенного нисходящего GLL и восходящего RNGLR анализа .

3. Модуль генератора

Ранее были описаны два основных подхода, используемые для автоматической генерации синтаксических анализаторов. Основными плюсами первого подхода, при котором генерируется весь кода парсера, является простота и удобство отладки таких парсеров. Несмотря на это в рамках проекта был создан генератор, использующий второй подход, - генерация дополнительной информации (функции для работы с грамматикой и управляющие таблицы). Выбор обусловлен необходимостью получения абстрактного анализатора в дальнейшем, который в процессе работы использует таблицы предиктивного анализа.

На вход генератору подаётся грамматика, после чего она подвергается преобразованиям и представляется в компактном виде. Каждому символу сопоставляется число, считаются -правила, множества FIRST и FOLLOW и функции для работы с грамматикой. Для этой первичной обработки грамматики и её хранения переиспользуются модули из GLR-генератора. Выделен общий модуль для обоих генераторов и структура до и после выделения выглядит следующим образом:

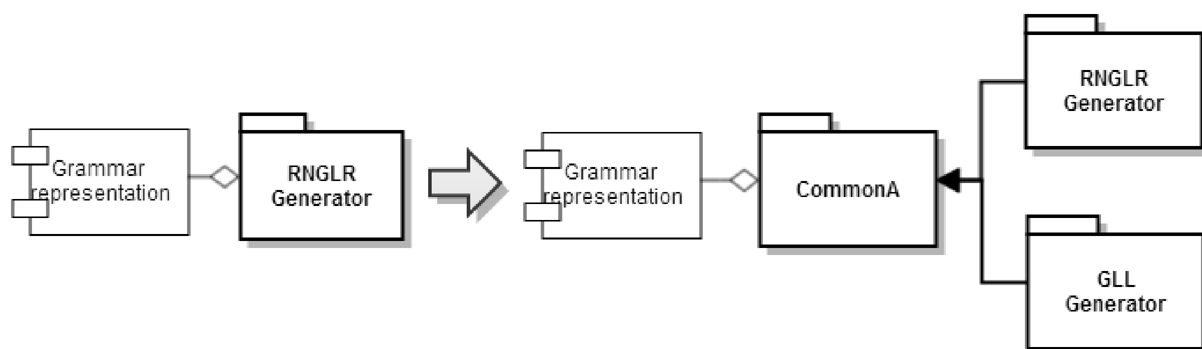

Рис. 6: Переиспользование в модуле генератора.

После этого уже по преобразованной грамматике строится таблица, в ячейках которой хранятся номера продукций, по которым можно произвести разбор для текущего нетерминала и входного символа. В каждой ячейке такой таблицы может храниться несколько номеров продукций, что обусловлено возможным наличием неоднозначностей в грамматике. Данная таблица строится по аналогии с обычной LL-таблицей, с учётом того, что в грамматике могут быть неоднозначности.

Например, рассмотрим неоднозначную грамматику H_1:

$$s := A\ d\ B\ |\ A\ D\ B\ d := D$$

Управляющая таблица T изображена на Рис. 7.

	A	B	D	$
s	s -> A d B s -> A D B	error	error	error
d	error	error	d -> D	error

Рис. 7: Управляющая таблица для грамматики H_1.

В ячейке T[s, A] хранится две продукции, по которым может быть осуществлён разбор. Это происходит потому что такая контекстно-свободная грамматика не принадлежит классу LL(1)-грамматик, так как в ней пересечение множеств FIRST для этих продукций не пусто. Классические LL-таблицы не позволяют хранить более одной продукции, что обусловлено невозможностью обрабатывать не LL-грамматики. Таблица для обобщённого анализа в таких случаях в ячейке хранит все допустимые продукции.

Эта таблица, грамматика и функции для работы с ней в дальнейшем используются интерпретатором, который содержит в себе основную логику алгоритма GLL.

4. Модуль интерпретатора

4.1. Модификации алгоритма GLL

В статье, описывающей алгоритм [6], который взят за основу, по грамматике полностью генерируется синтаксический анализатор. Он состоит из набора функций для нетерминалов грамматики и управляющей функции. Управляющая функция контролирует процесс работы анализатора: извлекает новые дескрипторы (описатель текущего состояния) из очереди, присваивает необходимые значения и вызывает соответствующую функцию разбора.

Передача управления между функциями осуществляется с использованием команды goto. Поскольку в рамках работы необходимо реализовать табличный анализатор, процесс работы которого отличается от описанного в статьях [10], в алгоритм были внесены некоторые изменения.

Вместо нескольких функций, соответствующих нетерминалам, используется пара функций: управляющая, которая выполняет ту же роль, что и раньше, и обрабатывающая. Для корректной работы обрабатывающей функции в зависимости от текущего входного символа и символа в грамматике были выделены следующие ситуации.

- Если текущий рассматриваемый символ x = A, где A — терминал, то необходимо перейти к рассмотрению следующего символа в правиле, а указатель во входном потоке увеличить на единицу.
- Если x — это нетерминальный символ, то в стек записывается текущее правило и запоминается позиция в нём. С использованием этой информации будет осуществляться обработка после того, как нетерминал x будет обработан до конца. А рассматриваемым становится правило, по которому раскрывается x в зависимости от текущего символа во входном потоке. Указатель во входном потоке остается без изменений.
- Если какое-то правило рассмотрено до конца и текущий стек не пуст, то извлекаем дескриптор с вершины стека и продолжаем работу с этими данными.

Таким образом обрабатывающая функция просто выполняет разные действия в зависимости от ситуации.

Для работы анализатора необходимы структуры, описывающие деревья вывода и стек. Как упоминалось ранее, процесс работы с такими структурами в алгоритмах GLL и RNGLR похож. Поэтому в ходе разработки GLL-модуля эти структуры переиспользовались без изменений. Модули до и после выделения общей части выглядят следующим образом показаны на Рис. 8.

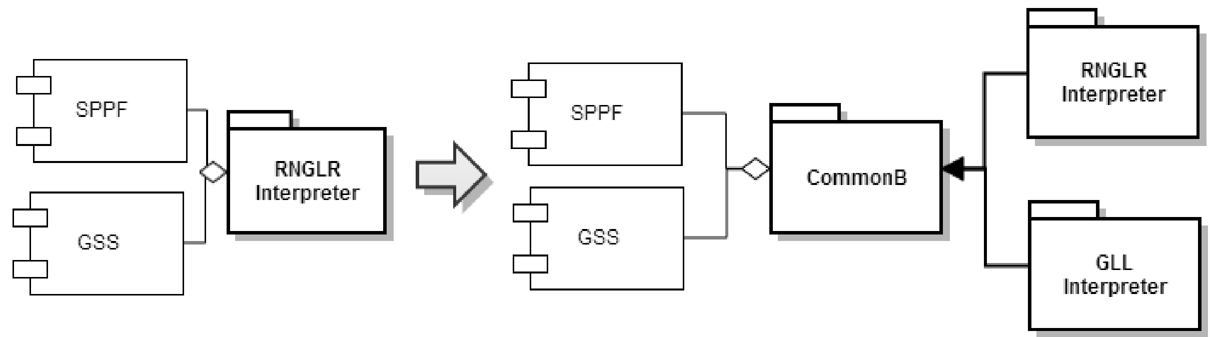

Рис. 8: Переиспользование в модуле интерпретатора.

4.2. Тестирование

Результатом работы интерпретатора является абстрактное синтаксическое дерево, для представления которого используется SPPF. Например, рассмотрим простую грамматику H_2 вида:

s := A b S S | b
b := B

И вход ABSS для такой грамматики. В результате будет получено дерево, изображённое на Рис. 9.

Для грамматики H_3 вида:

s := A d B | A D B
d := D

Для такой грамматики и входа ADB дерево отображено на Рис. 10.

В этом дереве помечен нетерминал, разбор которого может быть произведён двумя разными способами: с помощью продукции s := A d B и s := A D B. При этом терминальные ячейки, которые являются листьями, хранятся в единственном экземпляре в данном случае, т.е. каждая такая ячейка для данной позиции во входном потоке создаётся лишь однажды, а потом переиспользуется.

Рассмотрим леворекурсивную грамматику H_4:

s := B | s s

Для такой грамматики и входа BBB дерево будет иметь вид как на Рис. 11.

Далее рассмотрим грамматику H_5 со скрытой левой рекурсией:

s := B | f
f := s s

Для входа BBB и грамматики H_5 дерево вывода изображено на Рис. 12.

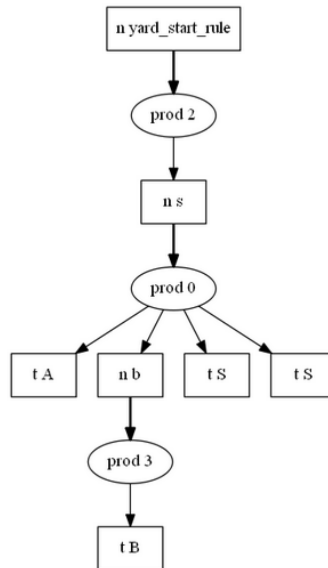

Рис. 9: Дерево вывода для грамматики H_2 и входа ABSS.

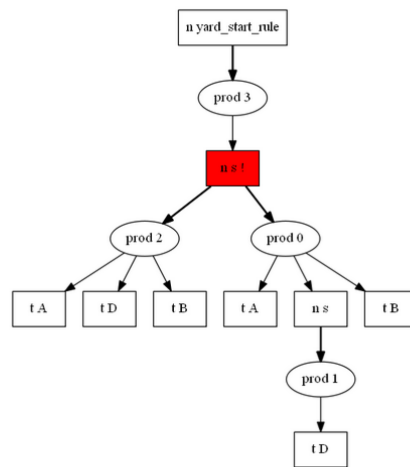

Рис. 10: Дерево вывода для грамматики H_3 и входа ADB.

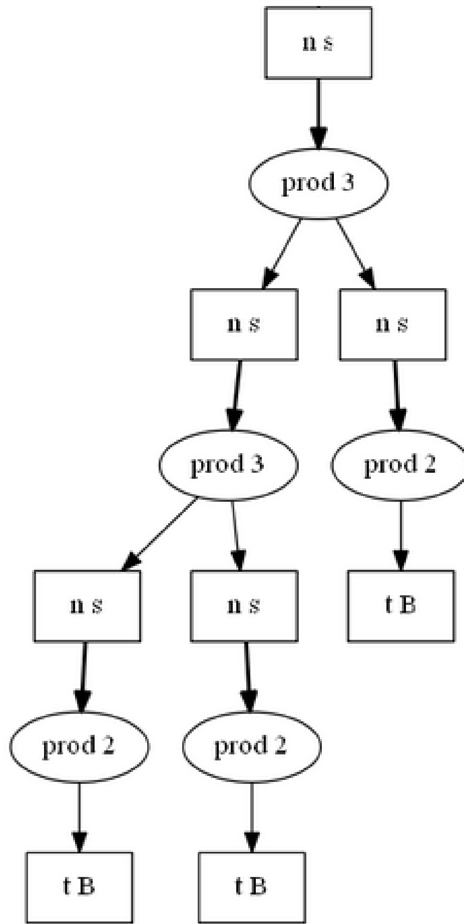

Рис. 11: Дерево вывода для грамматики H_4 и входа BBB.

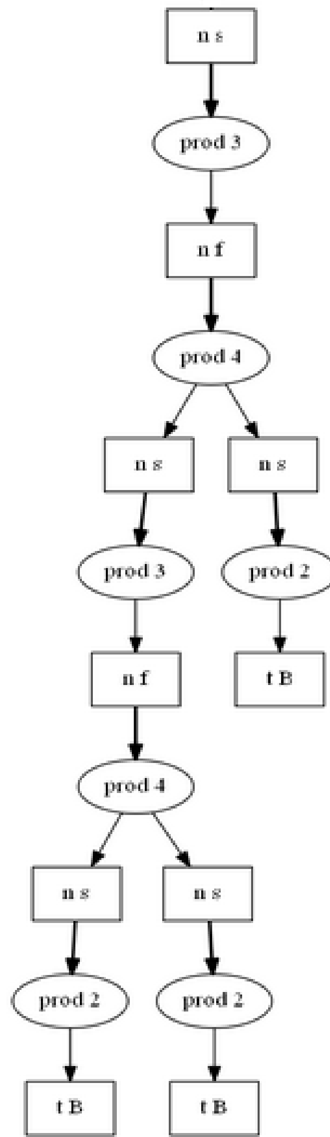

Рис. 12: Дерево вывода для грамматики H_5 и входа BBB.

5. Анализ и сравнение

Основным достоинством GLL является обработка левой и скрытой левой рекурсии. Для демонстрации преимуществ GLL были выбраны соответствующие грамматики и проведено сравнение времени работы с уже существующей реализацией RNGLR.

Для экспериментов использовалась леворекурсивная грамматика H_4 следующего вида:

s := B | s s

На вход подавались последовательности из токенов B с шагом 10. В ходе экспериментов были получены следующие результаты:

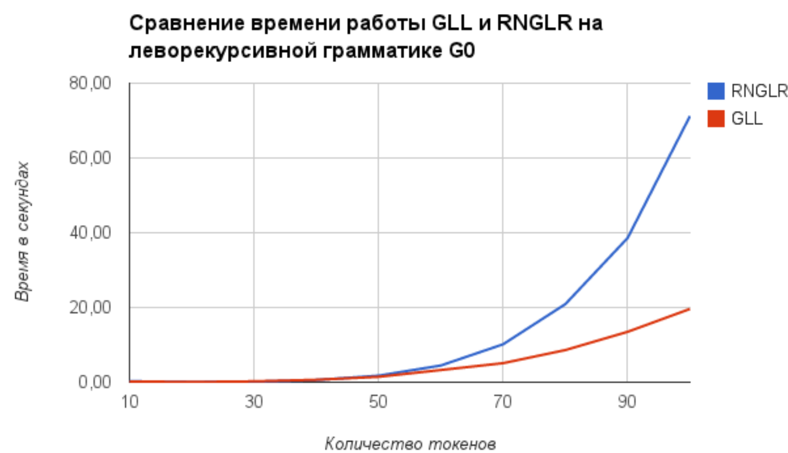

Рис. 13: Результаты сравнения для леворекурсивной грамматики H_4.

Также были проведены эксперименты для грамматики H_5 со скрытой левой рекурсией:

s := B | (f A)
f := s s

Для замеров использовались входные цепочки из токенов B с шагом 10 и токеном A на конце, то есть цепочки вида $B^(n*10)$A. Были получены следующие результаты:

Из экспериментов видно, что GLL грамматики с левой и скрытой левой рекурсией быстрее RNGLR.

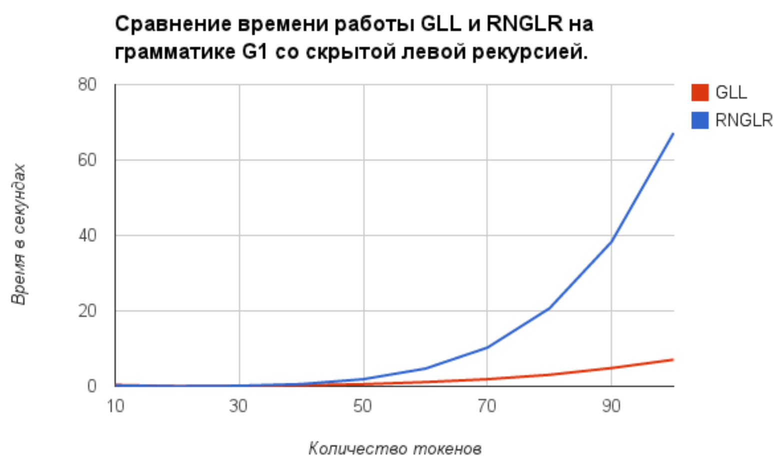

Рис. 14: Результаты сравнения для леворекурсивной грамматики H_5.

Заключение

В ходе данной работы получены следующие результаты.

- В проекте YC реализован генератор табличных GLL анализаторов.
- В проекте YC реализован алгоритм табличного GLL анализа. В отличии от ранее описанного GLL анализа, использующего явную генерацию в код, табличный подход предоставляет возможность использовать данный алгоритм для абстрактного анализа.
- Продемонстрирована работоспособность реализованного алгоритма для основных сценариев: неоднозначная грамматика, грамматика с левой рекурсией и грамматика со скрытой левой рекурсией.
- Проведено сравнение производительности реализованного GLL-анализатора с уже существовавшим и использующимся в промышленных проектах RNGLR-анализатором.

В процессе разработки происходила активная интеграция с уже существующим GLR-модулем, о котором упоминалось выше, и многие структуры переиспользовались.

Также было принято участие во Всероссийской научной конференции по проблемам информатики “СПИСОК” и Software Engineering Colloquium SYRCoSE.

Исходный код можно посмотреть в репозитории проекта — https://code.google.com/p/recursive-ascent/source/checkout. Ветка, в которой разрабатывался GLL — GLLBranch, автор — ragozina.anastasiya.

В ходе работы был получен табличный GLL-анализатор. Преобразование из явной генерации кода в табличный анализ потребовало обобщения и существенной модифи-

кации алгоритма, описанного в существующих работах. В дальнейшем необходимо провести более полное тестирование и создать подробное описание реализованных обобщённых функций. Отдельной большой задачей является реализация абстрактного GLL-анализа — механизма анализа для встроенных языков, для чего и потребовалась табличная версия алгоритма. Для этого, что будет полезно и само по себе, необходимо расширить алгоритм механизмом восстановления после ошибок и реализовать непосредственную поддержку EBNF.

Список литературы

[1] Alfred V. Aho and Ullman, *The Theory of Parsing, Translation and Compiling*, Parsing of Series in Automatic Computation. Prentice-Hall, 1972. P. 33-45.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, Compilers: Principles, Techniques and Tools. Pearson Education, Inc, 2006.

[3] D.J. Rosenkrantz and R.E. Stearns. Proceeding STOC '69 Proceedings of the first annual ACM symposium on Theory of computing. ACM, 1969. P. 165-180.

[4] Dick Grune and J. H. Ceriel Jacobs. Parsing Techniques: A Practical Guide (Second Edition). Springer, 2008.

[5] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen G. Langendoen. Modern Compiler Design (Second Edition). John Wiley & Sons, 2010.

[6] Elizabeth Scott and Adrian Johnstone GLL Parsing. Electronic Notes in Theoretical Computer Science 253, 2010. P. 177-189.

[7] Y.A. Kirilenko, S.V. Grigoriev, D.A. Avdyukhin. Syntax analyzers development in automated reengineering of informational system. Scientific and technical statements SPbSPU Issue 3, 2013.

[8] YaccConstructor home page https://code.google.com/p/recursive-ascent/wiki/YaccConstructor

[9] Elizabeth Scott and Adrian Johnstone. Right Nulled GLR Parsers.

[10] Elizabeth Scott and Adrian Johnstone. Modelling GLL Parser Implementations. Engineering Lecture Notes in Computer Science Volume 6563, 2011. P. 42-61.

[11] Masaru Tomita. Generalized LR parsing. Kluwer academic publishers, 1991.

[12] J. Aycock and N. Horspool. Practical Earley Parsing. The Computer Journal, P. 620–630, 2002.

[13] J. Earley. An Efficient Context-Free Parsing Algorithm. Commun. ACM, P. 94–102, 1970.

[14] REKERS, J. G. 1992. Parser generation for interactive environments. Ph.D. thesis, Universty of Amsterdam.

[15] Masaru Tomita. Eficient parsing for natural language: a fast algorithm for practical systems. Kluwer Academic Publishers, Boston, 1986.

[16] Официальный сайт ANTLR http://www.antlr.org/.

[17] Kyung-Goo Doh, Hyunha Kim, and David A. Schmidt: Abstract parsing: Static analysis of dynamically generated string output using lr-parsing technology. In Proceedings of the 16th International Symposium on Static Analysis, SAS '09. Springer-Verlag: Berlin; Heidelberg, 2009. P. 256–272.

[18] Grigoriev Semyon, Kirilenko Iakov. GLR-based abstract parsing // CEE-SECR'13 Proceedings of the 9th Central Eastern European Software Engineering Conference in Russia.

[19] Elizabeth Scott, Adrian Johnstone. GLL parse-tree generation. Science of Computer Programming, Volume 78, Issue 10, 2013, P. 1828–1844.