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

Кафедра системного программирования

Вербицкая Екатерина Андреевна

Синтаксический анализ регулярных множеств

Дипломная работа

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

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

Рецензент:
программист ООО "ИнтеллиДжей Лабс",
магистр прикладной математики и информатики Бреслав А. А.

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY

Chair of Software Engineering

Ekaterina Verbitskaia

Parsing of Regular Sets

Graduation Thesis

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

Scientific supervisor:
Senior Lecturer Semen Grigorev

Reviewer:
Software Developer at IntelliJ Labs Andrey Breslav

Saint-Petersburg
2015

Оглавление

Введение

Существует множество программ, которые динамически формируют выражения на некотором языке из строковых литералов с помощью строковых операций, и передают эти выражения специальным компонентам времени исполнения для дальнейшего анализа и выполнения. Такие программы мы называем *генераторами*; динамически формируемые выражения будем называть *встроенным кодом* или *выражениями на встроенном языке*. Множество средств предоставляют возможность использовать встроенный код, например: динамический SQL (Dynamic SQL [1]), фреймворк JSP [11], PHP mySQL interface [14].

Достоинствами использования такой динамической генерации кода являются гибкость, выразительность и высокая производительность полученных программ. Однако этот подход делает поведение программы гораздо менее предсказуемым за счёт невозможности применения стандартных для многих языков программирования видов статического анализа. Например, во время компиляции основной программы стандартными средствами не представляется возможности провести синтаксический анализ формируемых выражений. Данное обстоятельство делает невозможным вычисление семантики, и, как следствие, не позволяет решить многие классические задачи статического анализа программного кода, такие как проверка корректности типов или определение неинициализированных переменных. Помимо влияния на надёжность системы использование встроенных языков значительно усложняет разработку и понимание ранее написанного кода, а также его отладку.

Преодолеть многие проблемы такого подхода позволяет статический анализ множества значений динамически формируемого выражения. Одним из примеров такого анализа является проверка соответствия формируемых выражений некоторой эталонной грамматике. Например, если речь идёт о встраивании SQL-выражений в код на PHP, необходимо проверить, что все порождаемые выражения действительно принадлежат языку SQL. Множество значений динамически формируемого выражения является языком L. Задача проверки корректности строк языка L формализуется как задача проверки его вложенности в эталонный язык L_r. Как правило, язык, на котором написана основная программа, является тьюринг-полным, в следствие чего язык L может быть рекурсивно перечислимым, что делает задачу проверки вложенности языков алгоритмически неразрешимой. Распространённый подход к анализу встроенных языков заключается в анализе множества, аппроксимирующего сверху (over-approximation) множество порождаемых выражений [4, 5, 12, 2]. Например, если аппроксимирующее множество L_a – регулярный язык, а эталонный язык L_r относится к классу детерминированных контекстно-свободных, задача проверки $L_a \subseteq L_r$ разрешима [3]. Данные рассуждения используются во многих существующих подходах к анализу динамически формируемых выражений.

Несмотря на то, что вычисление семантики можно производить одновременно с синтаксическим анализом, часто оказывается удобнее работать со структурным представлением исходного кода. Таким представлением может являться лес деревьев вывода формируемых выражений. Однако непосредственная генерация таких деревьев невозможна по причине необходимости экспоненциального количества ресурсов, а так же заранее неограниченной высоты таких деревьев, например, в случае формирования выражения через конкатенацию в цикле. Таким образом, необходимо некоторое представление леса вывода, имеющее конечный размер. Алгоритмы синтаксического анализа семейства Generalized LR, предназначенные для работы с произвольными, в том числе неоднозначными, контекстно-свободными грамматиками, строят сжатый лес разбора (Shared Packed Parse Forest, SPPF [15]), компактно представляющий все возможные варианты разбора входной цепочки. В случае анализа встроенных языков SPPF может быть использован как компактное представление леса разбора строк аппроксимирующего множества L_a.

1. Постановка задачи

Целью данной работы является разработка алгоритма ослабленного синтаксического анализа для регулярной аппроксимации множества значений динамически формируемого выражения, который для всех корректных относительно некоторой эталонной грамматики цепочек строит конечное представление множества деревьев разбора, при этом цепочки, не принадлежащие эталонному языку, игнорируются.

В рамках данной дипломной работы поставлены следующие задачи.

- Разработать алгоритм синтаксического анализа динамически формируемых выражений, поддерживающий работу с произвольными входными графами.
- Реализовать предложенный алгоритм.
- Доказать корректность алгоритма.
- Провести апробацию.

2. Обзор

Предлагаемый в данной работе алгоритм заимствует распространённые принципы существующих работ данной области. Помимо этого переиспользуется RNGLR-алгоритм синтаксического анализа вместе с соответствующими структурами данных. В данном разделе приведён обзор подходов к анализу встроенных языков, дано краткое описание RNGLR-алгоритма, а также описан проект, в рамках которого велась разработка предложенного алгоритма.

2.1. Подходы к анализу динамически формируемых выражений

Анализ встроенных языков начинается с поиска и выделения в исходном коде основной программы так называемых *точек интереса* или *хотспотов* (hotspot), в которых осуществляется исполнение или передача формируемого выражения выделенной компоненте. Определение точек интереса обычно производится либо с помощью пользователя, посредством аннотаций или явного указания соответствующих строк исходного кода, либо автоматически, если есть некоторые знания об используемом фреймворке. Например, на хотспот могут указывать такие функции как exec динамического SQL. В точке интереса производится вычисление множества значений динамически формируемого выражения, а также его последующая аппроксимация. Следующим шагом, как правило, производится токенизация, то есть по языку над алфавитом символов строится язык над алфавитом терминалов эталонной грамматики. Затем полученный язык может анализироваться (синтаксически и семантически) в зависимости от стоящей задачи. Перечисленные шаги не обязательно производятся последовательно, более того, некоторые из них могут быть опущены.

Под *регулярной аппроксимацией* мы подразумеваем аппроксимирование множества значений динамически формируемого выражения некоторым регулярным выражением. В терминах, привычных для задач распознавания, проверка корректности выражений из аппроксимирующего множества формулируется как задача проверки включения регулярного языка в эталонный, как правило, контекстно-свободный, язык, которая разрешима во многих важных на практике случаях [3]. Многие существующие работы по анализу встроенных языков используют такой подход. В работе [4] для построения аппроксимации используется анализ прямой достижимости (forward reachability analysis), дальнейший анализ основан на распознавании образцов (pattern detection) в аппроксимирующем множестве или генерации некоторого конечного подмножества строк для анализа с помощью отдельных инструментов. В работе [5] построение регулярной аппроксимации осуществляется посредством расширения контекстно-свободной аппроксимации, построенной во время анализа исход-

ного кода основной программы. Предложенный нами подход вдохновлён проектом Alvor [12, 2], который использует технику, основанную на GLR-алгоритме, для синтаксического анализа регулярной аппроксимации. В данном проекте для упрощения задачи синтаксического анализа, регулярный язык над алфавитом символов сначала трансформируется в регулярный язык над алфавитом терминалов эталонной грамматики с помощью специальной процедуры лексического анализа.

В серии работ [7, 8, 9] предложен подход, основанный на явном представлении множества формируемых выражений в виде системы уравнений потока данных (data-flow equations). В качестве базы алгоритма синтаксического анализа выбран традиционный LALR(1)-алгоритм, при этом переиспользуются таблицы управления. Синтаксический анализ производится во время решения уравнений потока данных в домене абстрактных стеков. Проблема возможного бесконечного роста стеков, возникающая в общем случае, разрешается с помощью абстрактной интерпретации (abstract interpretation [6]). Впоследствии данный подход был расширен вычислением семантики с помощью атрибутных грамматик, что позволило анализировать более широкий, чем LALR(1), класс грамматик.

2.2. RNGLR-алгоритм

RNGLR-алгоритм (Right-Nulled Generalized LR) является модификацией предложенного Масару Томитой алгоритма Generalized LR (GLR) [18], предназначенного для анализа естественных языков. GLR-алгоритм был предназначен для анализа неоднозначных контекстно-свободных грамматик. Неоднозначности грамматики порождают конфликты сдвиг/свёртка (shift/reduce) и свёртка/свёртка (reduce/reduce). GLR-алгоритм использует управляющие таблицы, схожие с управляющими таблицами LR-алгоритма, ячейки которых могут содержать несколько действий. Основная идея GLR-алгоритма состоит в проведении всех возможных действий во время синтаксического анализа. При этом для эффективного представления множества стеков и деревьев вывода используются особые структуры данных, основанные на графах.

Оригинальный алгоритм, предложенный Томитой, не был способен анализировать все контекстно-свободные грамматики. Элизабет Скотт и Адриан Джонстоун предложили RNGLR-алгоритм, который расширяет GLR-алгоритм специальным способом обработки обнуляемых справа правил (right-nullable rules, имеющих вид $A \to \alpha\beta$, где β выводит пустую строку ϵ).

Для эффективного представления множества стеков во время синтаксического анализа в алгоритме RNGLR используется структурированный в виде графа стек (Graph Structured Stack, GSS), который является ориентированным графом, чьи вершины соответствуют элементам отдельных стеков, а ребра связывают последовательные элементы. Каждая вершина может иметь несколько входящих рёбер, что соот-

Algorithm 1 RNGLR-алгоритм

```
1: function parse(grammar, input)
2:     R ← ∅                    ▷ Очередь троек: вершина GSS, нетерминал, длина свёртки
3:     Q ← ∅                    ▷ Коллекция пар: вершина GSS, состояние парсера
4:     if input = ε then
5:         if grammar accepts empty input then report success
6:         else report failure
7:     else
8:         addVertex(0, 0, startState)
9:         for all i in 0..input.Length − 1 do
10:            reduce(i)
11:            push(i)
12:        if i = input.Length − 1 and there is a vertex in the last level of GSS which state
   is accepting then
13:            report success
14:        else report failure
15: function reduce(i)
16:    while R is not empty do
17:        (v, N, l) ← R.Dequeue()
18:        find the set X of vertices reachable from v along the path of length (l − 1)
19:        or length 0 if l = 0
20:        for all v_h = (level_h, state_h) in X do
21:            state_t ← calculate new state by state_h and nonterminal N
22:            addEdge(i, v_h, v.level, state_tail, (l = 0))
23: function push(i)
24:    Q' ← copy Q
25:    while Q' is not empty do
26:        (v, state) ← Q.Dequeue()
27:        addEdge(i, v, v.level + 1, state, false)
```

Algorithm 2 Построение GSS

```
1: function addVertex(i, level, state)
2:     if GSS does not contain vertex v = (level, state) then
3:         add new vertex v = (level, state) to GSS
4:         calculate the set of shifts by v and the input[i + 1] and add them to Q
5:         calculate the set of zero-reductions by v and the input[i + 1] and
6:         add them to R
7:     return v
8: function addEdge(i, v_h, level_t, state_t, isZeroReduction)
9:     v_t ← addVertex(i, level_t, state_t)
10:    if GSS does not contain edge from v_t to v_h then
11:        add new edge from v_t to v_h to GSS
12:        if not isZeroReduction then
13:            calculate the set of reductions by v and the input[i + 1] and
14:            add them to R
```

ветствует слиянию нескольких стеков, за счёт чего производится переиспользование общих участков отдельных стеков. Вершина GSS — это пара (s, l), где s — состояние парсера, а l — уровень (позиция во входном потоке).

RNGLR-алгоритм последовательно считывает символы входного потока слева направо, по одному за раз, и строит GSS по "слоям": сначала осуществляются все возможные свёртки для данного символа, после чего сдвигается следующий символ со входа. Свёртка или сдвиг модифицируют GSS следующим образом. Предположим, что необходимо добавить ребро (v_t, v_h) в GSS. По построению, конечная вершина добавляемой дуги к такому моменту уже обязательно находится в GSS. Если начальная вершина также содержится в GSS, то в граф добавляется новое ребро (если оно ранее не было добавлено), иначе создаются и добавляются в граф и начальная вершина, и ребро. Каждый раз, когда создаётся новая вершина $v = (s, l)$, алгоритм вычисляет новое состояние парсера s' по s и следующему символу входного потока. Пара (v, s'), называемая push, добавляется в глобальную коллекцию $\mathcal{Q}$. Также при добавлении новой вершины в GSS вычисляется множество ϵ-свёрток, после чего элементы этого множества добавляются в глобальную очередь $\mathcal{R}$. Свёртки длины $l > 0$ вычисляются и добавляются в $\mathcal{R}$ каждый раз, когда создаётся новое (не-ϵ) ребро. Подробное описание работы со структурированным в виде графа стеком GSS содержится в алгоритме 2.

В силу неоднозначности грамматики входная строка может иметь несколько деревьев вывода, как правило, содержащих множество идентичных поддеревьев. Для того, чтобы компактно хранить множество деревьев вывода, создано компактное представление леса разбора (Shared Packed Parse Forest, SPPF) [15]. SPPF является ориентированным графом и обладает следующей структурой.

1. *Корень* (то есть, вершина, не имеющая входящих дуг) соответствует стартовому нетерминалу грамматики.

2. *Терминальные* вершины, не имеющие исходящих дуг, соответствуют либо терминалам грамматики, либо деревьям вывода пустой строки ϵ.

3. *Нетерминальные* вершины являются корнем дерева вывода некоторого нетерминала грамматики; только вершины-продукции могут быть непосредственно достижимы из таких вершин.

4. *Вершины-продукции*, представляющие правую часть правила грамматики для соответствующего нетерминала. Вершины, непосредственно достижимые из них, упорядочены и могут являться либо терминальными, либо нетерминальными вершинами. Количество таких вершин лежит в промежутке $[l - k..l]$, где l — это длина правой части продукции, а k — количество финальных символов, выводящих ϵ (правые обнуляемые символы игнорируются для уменьшения потребления памяти).

SPPF создаётся одновременно с построением GSS. Каждое ребро GSS ассоциировано с либо с терминальным, либо с нетерминальным узлом. Когда добавление ребра в GSS происходит во время операции push, новая терминальная вершина создаётся и ассоциируется с ребром. Нетерминальные вершины ассоциируются с ребрами, добавленными во время операции reduce. Если ребро уже было в GSS, к ассоциированной с ним нетерминальной вершине добавляется новая вершина-продукция. Подграфы, ассоциированные с рёбрами пути, вдоль которого осуществлялась свёртка, добавляются как дети к вершине-продукции. После того, как входной поток прочитан до конца, производится поиск всех вершин, имеющих принимающее состояние анализатора, после чего подграфы, ассоциированные с исходящими из таких вершин рёбрами, объединяются в один граф. Из полученного графа удаляются все недостижимые из корня вершины, что в итоге оставляет только корректные деревья разбора для входной строки.

Алгоритм 1 представляет более детальное описание алгоритма.

2.3. Проект YaccConstructor и платформа для анализа встроенных языков

В рамках проекта YaccConstructor [13] лаборатории языков инструментов JetBrains на математико-механическом факультете СПбГУ проводятся исследования в области лексического и синтаксического анализа, а также статического анализа встроенных языков. Проект YaccConstructor представляет собой модульный инструмент, имеет собственный язык спецификации грамматик, объединяет различные алгоритмы лексического и синтаксического анализа. В рамках проекта была создана платформа для статического анализа встроенного кода [16]; диаграмма последовательности, иллюстрирующая взаимодействие модулей платформы представлена на рисунке 1. Цветом выделена компонента, осуществляющая синтаксический анализ множества значений динамически формируемого выражения.

Предыдущая реализация платформы работала с грубой аппроксимацией, которая не осуществляла поддержку формирования выражения в циклах и с помощью строковых выражений [10]. Используемая аппроксимация не являлась аппроксимацией сверху, что сказывалось на точности результатов анализа, и поэтому впоследствии она была заменена на регулярную. Однако это повлекло необходимость изменения алгоритма синтаксического анализа, чему и посвящена данная работа.

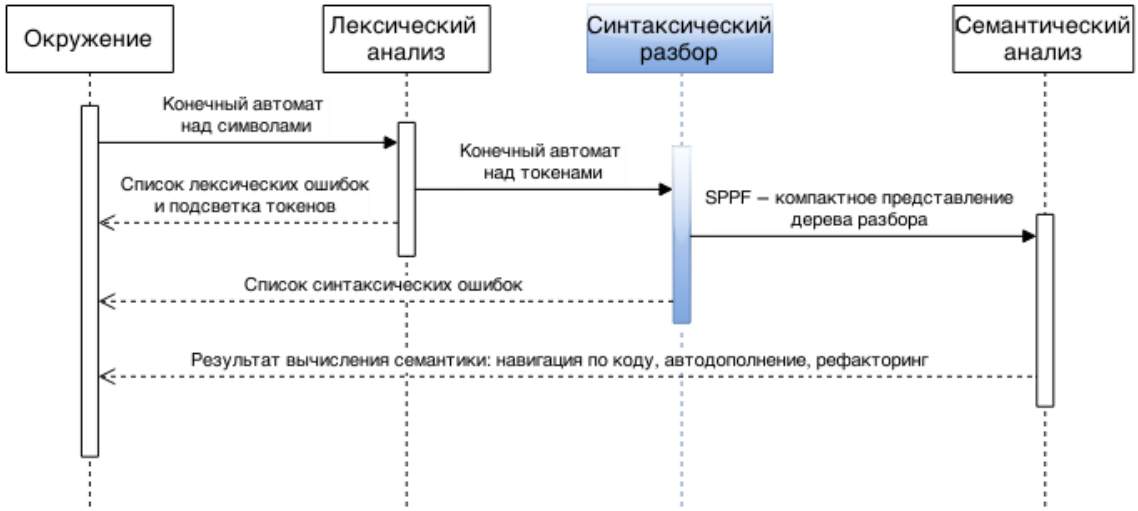

Рис. 1: Диаграмма последовательности: взаимодействие компонентов инструмента YaccConstructor

3. Алгоритм ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения

3.1. Описание алгоритма

Алгоритм принимает на вход эталонную грамматику G над алфавитом терминальных символов T и детерминированный конечный автомат $(Q, \Sigma, \delta, q_0, q_f)$, имеющий одно стартовое состояние q_0, одно конечное состояние q_f, без ϵ-переходов, где $\Sigma \subseteq T$ — алфавит входных символов, Q — множество состояний, δ — отношение перехода. По описанию грамматики генерируются управляющие RNGLR-таблицы и некоторая вспомогательная информация (называемая $parserSource$ в псевдокоде).

Algorithm 3 Алгоритм ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения

```
1:  function parse(grammar, automaton)
2:      inputGraph ← construct inner graph representation of automaton
3:      parserSource ← generate RNGLR parse tables for grammar
4:      if inputGraph contains no edges then
5:          if parserSource accepts empty input then report success
6:          else report failure
7:      else
8:          addVertex(inputGraph.startVertex, startState)
9:          Q.Enqueue(inputGraph.startVertex)
10:         while Q is not empty do
11:             v ← Q.Dequeue()
12:             makeReductions(v)
13:             push(v)
14:             applyPassingReductions(v)
15:         if ∃v_f : v_f.level = q_f and v_f.state is accepting then report success
16:         else report failure
```

Алгоритм производит обход графа входного автомата и последовательно строит GSS тем же способом, как это делает RNGLR-алгоритм. Однако, так как мы имеем дело с графом вместо линейного потока, понятие следующего символа трансформируется во *множество терминальных символов*, лежащих на всех исходящих рёбрах данной вершины, что несколько изменяет операции shift и reduce (смотри строку 5 в алгоритме 4 и строки 9 и 21 в алгоритме 5). Для того, чтобы управлять порядком обработки вершин входного графа, мы используем глобальную очередь $\mathcal{Q}$. Каждый раз, когда добавляется новая вершина GSS, сначала необходимо произвести все свёртки длины 0, после чего выполнить сдвиг следующих токенов со входа. Таким образом необходимо добавить соответствующую вершину графа в очередь на обра-

ботку. Добавление нового ребра GSS может порождать новые свёртки, таким образом в очередь на обработку необходимо добавить вершину входного графа, которой соответствует начальная вершина добавленного ребра. Детальное описание процесса построения GSS приведено в алгоритме 3. Свёртки производятся вдоль путей в GSS, и если было добавлено ребро, начальная вершина которого ранее присутствовала в GSS, необходимо заново вычислить проходящие через эту вершину свёртки (смотри функцию applyPassingReductions в алгоритме 4).

Algorithm 4 Обработка вершины внутреннего графа

```
1:  function push(innerGraphV)
2:      U ← copy innerGraphV.unprocessed
3:      clear innerGraphV.unprocessed
4:      for all v_h in U do
5:          for all e in outgoing edges of innerGraphV do
6:              push ← calculate next state by v_h.state and the token on e
7:              addEdge(v_h, e.Head, push, false)
8:              add v_h in innerGraphV.processed
9:  function makeReductions(innerGraphV)
10:     while innerGraphV.reductions is not empty do
11:         (startV, N, l) ← innerGraphV.reductions.Dequeue()
12:         find the set of vertices X reachable from startV
13:          along the path of length (l − 1), or 0 if l = 0;
14:         add (startV, N, l − i) in v.passingReductions,
15:          where v is an i-th vertex of the path
16:         for all v_h in X do
17:             state_t ← calculate new state by v_h.state and nonterminal N
18:             addEdge(v_h, startV, state_t, (l = 0))
19: function applyPassingReductions(innerGraphV)
20:     for all (v, edge) in innerGraphV.passingReductionsToHandle do
21:         for all (startV, N, l) ← v.passingReductions.Dequeue() do
22:             find the set of vertices X,
23:              reachable from edge along the path of length (l − 1)
24:             for all v_h in X do
25:                 state_t ← calculate new state by v_h.state and nonterminal N
26:                 addEdge(v_h, startV, state_t, false)
```

Так же как и RNGLR, мы ассоциируем вершины GSS с позициями входного графа, однако в нашем случае уровень вершины — это состояние входного автомата. Мы строим внутреннюю структуру данных (в дальнейшем изложении называемую *внут-*

ренним графом) посредством копирования графа входного автомата и ассоциации с его вершинами следующих коллекций.

- *processed*: вершины GSS, для которых ранее были вычислены все операции push. Это множество агрегирует все вершины GSS, ассоциированные с вершиной внутреннего графа.
- *unprocessed*: вершины GSS, операции push для которых ещё только предстоит выполнить. Это множество аналогично множеству $\mathcal{Q}$ алгоритма RNGLR.
- *reductions*: очередь, аналогичная очереди $\mathcal{R}$ RNGLR-алгоритма: все операции reduce, которые ещё только предстоит выполнить.
- *passingReductionsToHandle*: пары из вершины GSS и ребра GSS, вдоль которых необходимо осуществлять проходящие свёртки.

Algorithm 5 Построение GSS

```
1:  function addVertex(innerGraphV, state)
2:      v ← find a vertex with state = state in
3:       innerGraphV.processed ∪ innerGraphV.unprocessed
4:      if v is not null then                    ▷ Вершина была найдена в GSS
5:          return (v, false)
6:      else
7:          v ← create new vertex for innerGraphV with state state
8:          add v in innerGraphV.unprocessed
9:          for all e in outgoing edges of innerGraphV do
10:             calculate the set of zero-reductions by v
11:              and the token on e and add them in innerGraphV.reductions
12:         return (v, true)
13: function addEdge(v_h, innerGraphV, state_t, isZeroReduction)
14:     (v_t, isNew) ← addVertex(innerGraphV, state_t)
15:     if GSS does not contain edge from v_t to v_h then
16:         edge ← create new edge from v_t to v_h
17:         Q.Enqueue(innerGraphV)
18:         if not isNew and v_t.passingReductions.Count > 0 then
19:             add (v_t, edge) in innerGraphV.passingReductionsToHandle
20:         if not isZeroReduction then
21:             for all e in outgoing edges of innerGraphV do
22:                 calculate the set of reductions by v
23:                  and the token on e and add them in innerGraphV.reductions
```

Помимо состояния анализатора *state* и уровня *level* (который совпадает с состоянием входного автомата), в вершине GSS хранится коллекция *проходящих свёрток*. Проходящая свёртка — это тройка $(startV, N, l)$, соответствующая свёртке, чей путь содержит данную вершину GSS. Аналогичная тройка используется в RNGLR-алгоритме для описания свёртки, но в данном случае l обозначает длину оставшейся части пути. Проходящие свёртки сохраняются в каждой вершине пути (кроме первой и последней) во время поиска путей в функции *makeReductions* (см. алгоритм 4).

3.2. Построение компактного представления леса разбора

В качестве компактного представления леса разбора всех корректных выражений из множества значений динамически формируемого выражения используется граф SPPF. Построение компактного представления осуществляется одновременно с синтаксическим разбором во время построения графа стеков GSS, также как и в алгоритме RNGLR.

С каждым ребром GSS ассоциируется список лесов разбора фрагмента выражения. В графе GSS нет кратных рёбер, поэтому если во время работы функции *addEdge* в нем было найдено добавляемое ребро, то с ним ассоциируется новый лес разбора, при этом в очередь на обработку не добавляется никаких вершин входного графа.

При добавлении в GSS ребра, соответствующего считанной со входа лексеме, создаётся (и ассоциируется с ним) граф из одной терминальной вершины. Так как входной автомат является детерминированным, с ребром GSS ассоциируется не более одного такого графа.

При обработке свёртки алгоритм осуществляет поиск всех путей в графе GSS заданной длины, после чего происходит добавление в GSS новых рёбер, соответствующих данной свёртке. С каждым таким ребром ассоциируется лес, имеющий в качестве корня (вершины, у которой нет входных рёбер) вершину, соответствующую нетерминалу, к которому осуществлялась свёртка. Ребра каждого из найденных путей, перечисленные в обратном порядке, образуют правую часть некоторого правила грамматики, по которому осуществляется свёртка. Для каждого пути создаётся вершина, помеченная номером такого правила, и добавляется в лес как непосредственно достижимая из корня. Каждое ребро пути ассоциировано со списком лесов вывода символа из правой части правила. Непосредственно достижимыми вершинами вершины-правила становятся ссылки на такие списки, за счёт чего осуществляется переиспользование фрагментов леса.

В алгоритме RNGLR наличие нескольких путей, вдоль которых осуществляется свёртка к нетерминалу, означает существование более чем одного варианта вывода нетерминала. В нашем случае данная ситуация соответствует различным фрагментам нескольких выражений из входного регулярного множества, которые сворачиваются

к одному нетерминалу.

В конце работы алгоритма осуществляется поиск рёбер GSS, для каждого из которых верно, что конечная вершина имеет уровень, равный финальному состоянию входного автомата, и принимающее состояние (accepting state). Результирующее представление леса разбора получается путём удаления недостижимых вершин из графа, созданного объединением лесов разбора, ассоциированных с найденными рёбрами GSS.

Рассмотрим следующий фрагмент кода, динамически формирующий выражение *expr* в строке 4.

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

Множество значений выражения *expr* аппроксимируется регулярным выражением (LBR RBR)$*$, где LBR соответствует открывающейся скобке, а RBR — закрывающейся. Граф конечного автомата, задающего такую аппроксимацию, изображён на рисунке 2.

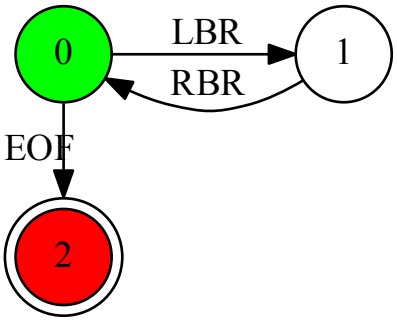

Рис. 2: Конечный автомат, задающий регулярную аппроксимацию выражения *expr*

В результате работы предложенного алгоритма будет получено конечное представление леса разбора SPPF, изображённое на рисунке 3.

Из SPPF можно извлечь бесконечное количество деревьев, каждое из которых является деревом вывода некоторого выражения из регулярной аппроксимации. Рисунок 4 демонстрирует одно из таких деревьев разбора.

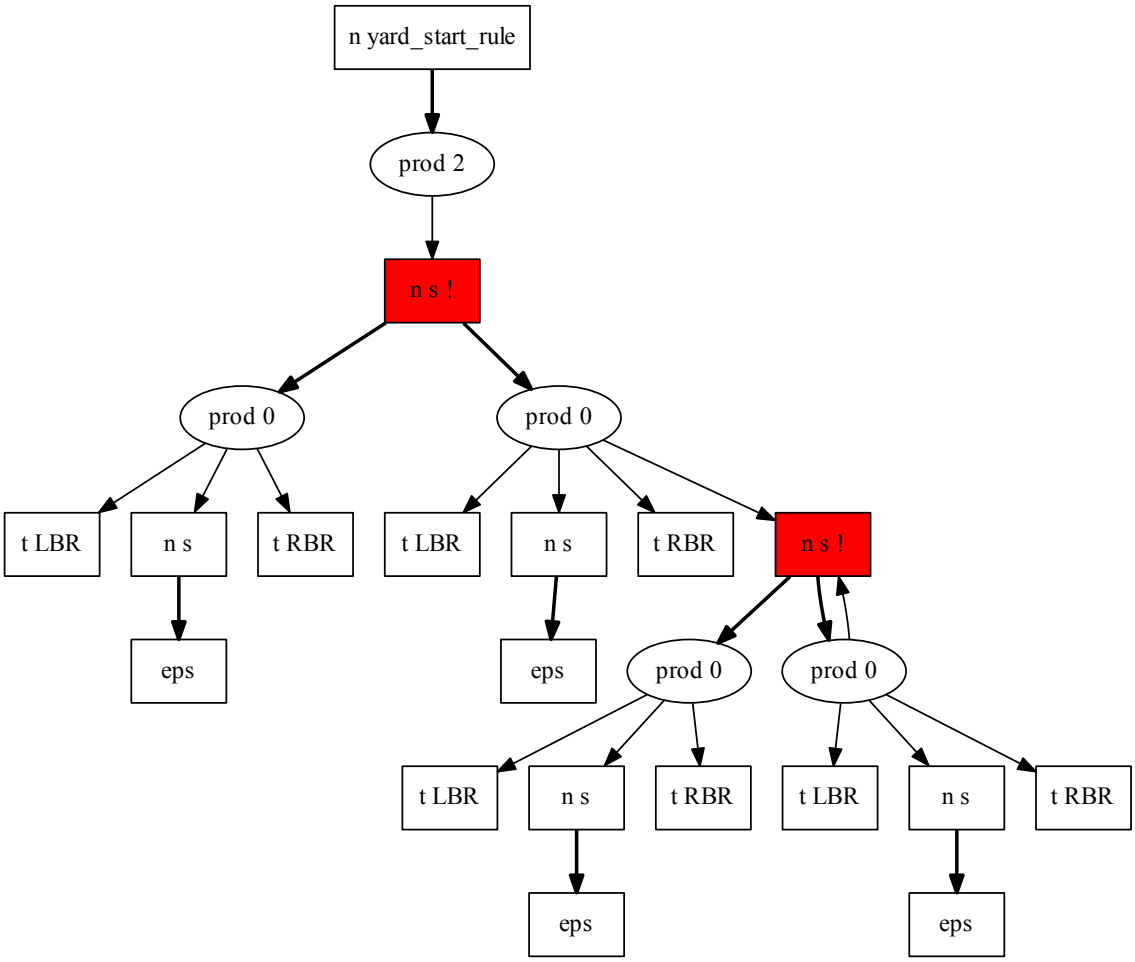

Рис. 3: Конечное представление леса разбора для выражения *expr*

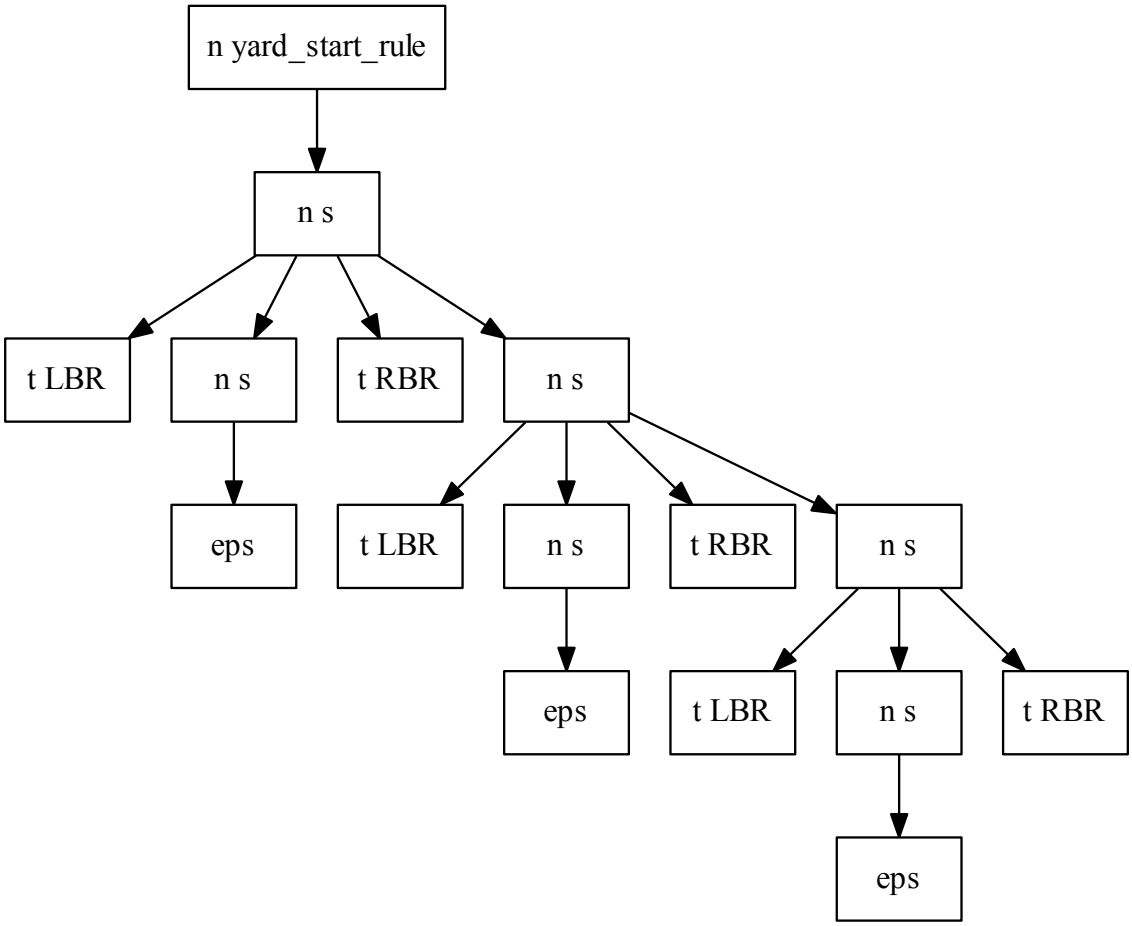

Рис. 4: Дерево вывода для выражения $expr =$ "()()()"

4. Доказательство корректности алгоритма

Теорема 1. *Алгоритм завершает работу для любых входных данных.*

Доказательство. С каждой вершиной внутреннего представления графа входного конечного автомата ассоциировано не более N вершин графа GSS, где N — количество состояний синтаксического анализатора. Таким образом, количество вершин в графе GSS ограничено сверху числом $N \times n$, где n — количество вершин графа входного автомата. Так как в GSS нет кратных рёбер, количество его рёбер — $O((N \times n)^2)$. На каждой итерации основного цикла алгоритм извлекает из очереди Q и обрабатывает одну вершину внутреннего графа. Вершины добавляются в очередь Q только, когда происходит добавление нового ребра в GSS. Так как количество рёбер в GSS конечно, алгоритм завершает работу для любых входных данных. □

Для того, чтобы доказать корректность построения конечного представления леса разбора, нам потребуется следующее определение.

Определение 1. *Корректное дерево* — это упорядоченное дерево со следующими свойствами.

1. Корень дерева соответствует стартовому нетерминалу грамматики G.
2. Листья соответствуют терминалам грамматики G. Упорядоченная последовательность листьев соответствует некоторому пути во входном графе.
3. Внутренние узлы соответствуют нетерминалам грамматики G. Дети внутреннего узла (для нетерминала N) соответствуют символам правой части некоторой продукции для N в грамматике G.

Неформально корректное дерево — это дерево вывода некоторой цепочки из регулярного множества в эталонной грамматике. Далее нам необходимо доказать, во-первых, что конечное представление леса разбора SPPF содержит только корректные деревья, и во-вторых, что для каждой корректной относительно эталонной грамматики цепочки существует корректное дерево вывода в SPPF.

Лемма. *Для каждого ребра GSS (v_t, v_h) такого, что $v_t \in V_t.processed$, $v_h \in V_h.processed$, терминалы ассоциированного поддерева соответствуют некоторому пути во входном графе из вершины V_h в V_t.*

Доказательство. Будем строить доказательство при помощи индукции по высоте дерева вывода. База: либо ϵ-дерево, либо дерево, состоящее из единственной вершины-терминала. ϵ-дерево соответствует пути длины 0; начальная и конечная вершины ребра, соответствующего такому дереву, совпадают, поэтому утверждение верно. Дерево, состоящее из одной вершины, соответствует терминалу, считанному с некоторого ребра (V_h, V_t) внутреннего графа, поэтому утверждение верно.

Корнем дерева высоты k является некий нетерминал N. По третьему пункту определения корректного дерева существует некоторое правило эталонной грамматики

$N \to A_0, A_1, \ldots, A_n$, где $A_0, A_1, \ldots, A_n$ являются детьми корневого узла. Поддерево A_i ассоциировано с ребром (v_t^i, v_h^i) графа GSS и, так как его высота равна $k-1$, то по индукционному предположению существует путь во внутреннем графе из вершины V_h^i в вершину V_t^i. Вершина $V_t^i = V_h^{i+1}$, так как $v_t^i = v_h^{i+1}$, поэтому во внутреннем графе существует путь из вершины V_h^0 в вершину V_t^n, соответствующий рассматриваемому корректному дереву. □

Теорема 2. *Любое дерево, сгенерированное из SPPF, является корректным.*

Доказательство. Рассмотрим произвольное сгенерированное из SPPF дерево и докажем, что оно удовлетворяет определению 1. Первый и третий пункт определения корректного дерева следует из определения SPPF. Второй пункт определения следует из леммы, если применить её к рёбрам GSS, начало которых лежит на последнем уровне стека и помечено принимающим состоянием, а конец — в вершинах на уровне 0. □

Теорема 3. *Для каждой корректной относительно эталонной грамматики строки, соответствующей пути p во входном графе, из SPPF может быть порождено корректное дерево, соответствующее пути p.*

Доказательство. Рассмотрим произвольное корректное дерево и докажем, что оно может быть порождено из SPPF. Доказательство повторяет доказательство корректности для RNGLR-алгоритма, за исключением следующего момента. RNGLR-алгоритм строит граф GSS по слоям: гарантируется, что $\forall j \in [0..i-1], j-$ый уровень GSS будет зафиксирован на момент построения $i-$ого уровня. В нашем случае это свойство не верно, что может приводить к возможному порождению новых путей для свёрток, которые уже были ранее обработаны. Единственный возможный способ добавления такого нового пути — это добавление ребра (v_t, v_h), где вершина v_t ранее присутствовала в GSS и имела входящие ребра. Так как алгоритм сохраняет информацию о том, какие свёртки проходили через вершины GSS, достаточно продолжить свёртки, проходящие через вершину v_t, и это ровно то, что делает функция *applyPassingReductions*. □

Замечание. Построение леса разбора осуществляется одновременно с построением GSS, при этом дерево вывода нетерминала ассоциируется с ребром GSS каждый раз при обработке соответствующей свёртки, вне зависимости от того, было ли ребро в графе до этого, либо добавлено на данном шаге. Это обстоятельство позволяет утверждать, что если все возможные редукции были выполнены, то и лес разбора содержит все деревья для всех корректных цепочек из аппроксимации.

5. Реализация и тестирование

Предложенный алгоритм был реализован на платформе .NET как часть проекта YaccConstructor; основным языком разработки являлся F# [17]. Ранее в рамках проекта был реализован RNGLR-алгоритм и генератор управляющих таблиц анализа для него. Управляющие таблицы RNGLR-алгоритма переиспользуются, поэтому внесения изменений в генератор не потребовалось. Также были переиспользованы структуры данных для структурированного в виде графа стека GSS и компактного представления разбора SPPF.

Алгоритм был протестирован на различных наборах тестов. Для каждого теста специфицировалась грамматика на языке YARD и в явном задавался граф конечного автомата, ребра которого были промаркированы лексемами эталонной грамматики. Полученный в результате работы алгоритма лес разбора печатался в файл и проверялся на корректность. Тесты можно разделить на две категории.

- *Регрессионные тесты* проверяющие, что предложенный алгоритм выдаёт те же результаты, что и RNGLR-алгоритм, на линейном входе (конечном автомате, принимающем единственную строку). В данный набор вошли все тесты, ранее использованные для тестирования работоспособности реализации RNGLR-алгоритма в проекте YaccConstructor.

- *Тесты на работоспособность*, проверяющие, что алгоритм строит корректное представление леса разбора всех корректных выражений из входного регулярного множества. Входные графы для данного набора тестов содержали как ветвления, так и циклы. Отдельно были рассмотрены случаи вложенных ветвлений и вложенных циклов. На всех тестах алгоритм генерировал корректный лес разбора, игнорируя некорректные относительно эталонной грамматики цепочки.

Замеры времени работы алгоритма проводились на машине со следующими техническими характеристиками: Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, RAM: 16.0 GB, процессор x64.

5.1. Тестирование производительности

Алгоритм был протестирован на нескольких сериях синтетических тестов, цель которых — убедиться в приемлемой производительности алгоритма на практически значимых входных данных. Анализ промышленного проекта по миграции базы данных с MS-SQL Server 2005 на Oracle 11gR2 показал, что запросы часто формируются конкатенацией фрагментов, каждый из которых формируется с помощью ветвлений

или циклов. Ниже приведена эталонная грамматика, использованная в этих тестах.

$$
\begin{aligned}
start\_rule &::= s \\
s &::= s \texttt{ PLUS n} \\
n &::= \texttt{ONE | TWO | THREE | FOUR | FIVE | SIX | SEVEN}
\end{aligned}
$$

Входные графы представляли собой конкатенацию базовых блоков. Каждая серия тестов характеризовалась тремя параметрами:

- *height* — количество ветвлений в базовом блоке;
- *length* — максимальное количество повторений базовых блоков;
- *isCycle* — наличие в базовом блоке циклов (если ложь, то используются базовые блоки, изображённые на рисунке 5, если истина — то изображённые на рисунке 6).

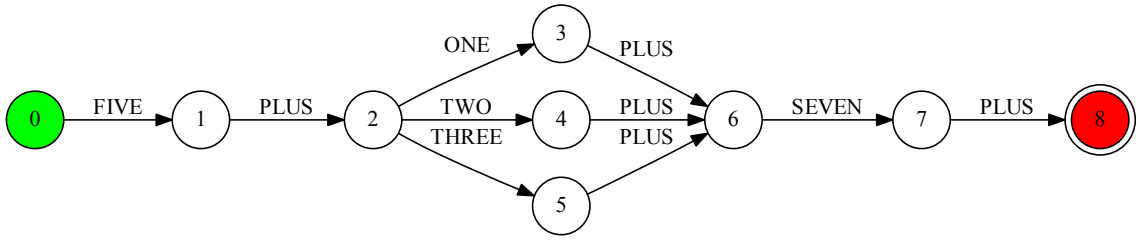

Рис. 5: Базовый блок без циклов при $length = 3$

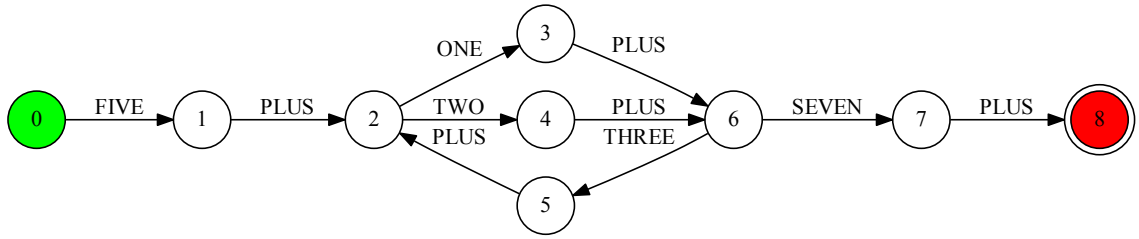

Рис. 6: Базовый блок, содержащий цикл, при $length = 3$

Каждая серия объединяет набор из 500 тестов, каждый из которых содержит одинаковое количество ветвлений в базовом блоке, при этом количество повторений блока совпадает с порядковым номером теста ($length = i$ для i-того теста). Для каждого теста измерялось время, затраченное на синтаксический анализ. Измерения проводились 10 раз, после чего усреднялись, при этом выбросы не учитывались. График, представленный на рисунке 7, иллюстрирует зависимость времени, затрачиваемого на синтаксический анализ, от количества повторения базового блока и количества ветвлений в каждом из них. Можно заметить, что продолжительность анализа растёт линейно, в зависимости от размера входного графа. График на рисунке 8 демонстрирует, что наличие циклов в графе увеличивает продолжительность анализа, при этом зависимость времени от размера графа остаётся линейной.

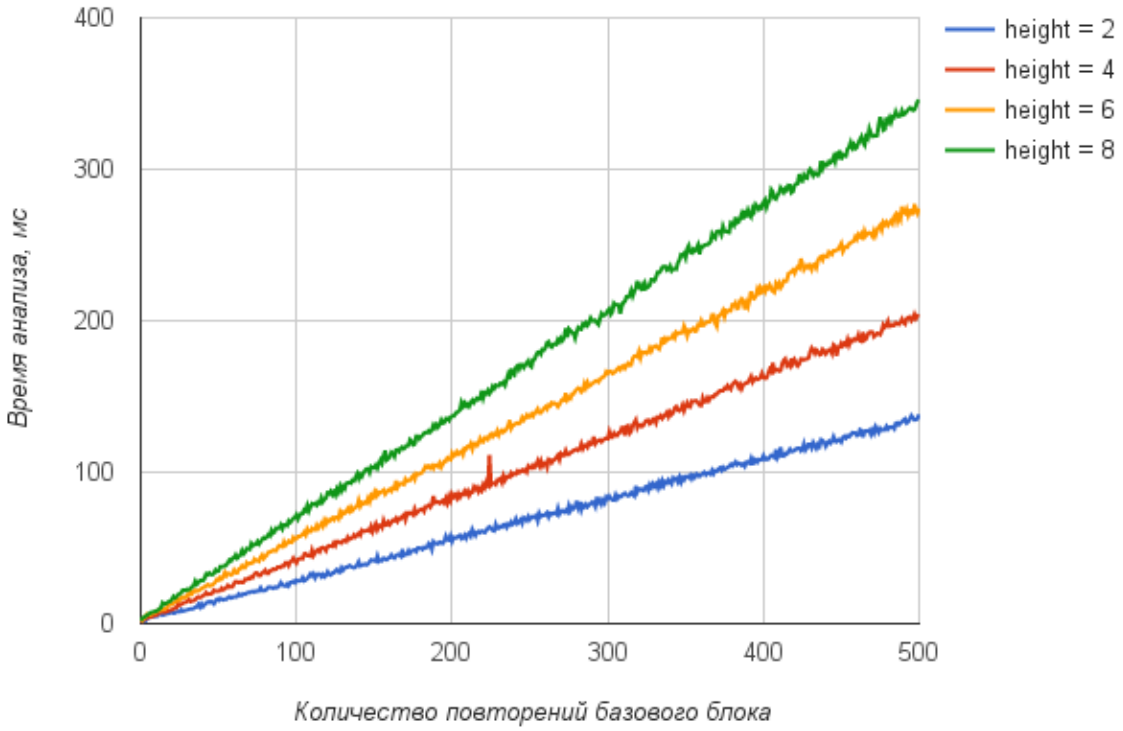

Рис. 7: Зависимость времени работы алгоритма от размера входного графа при $isCycle = false$

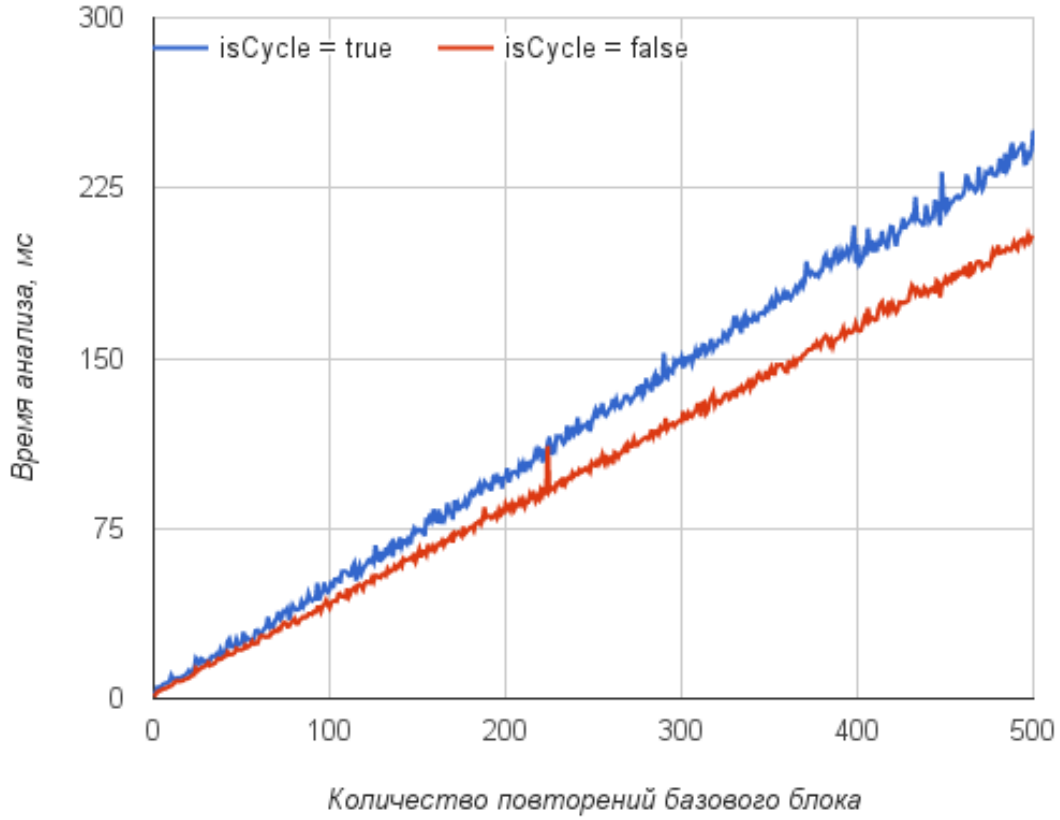

Рис. 8: Зависимость времени работы алгоритма от размера входного графа и наличия в нем циклов при $height = 4$

5.2. Тестирование на реальных данных

Алгоритм был протестирован на наборе данных, взятых из промышленного проекта по миграции базы данных с MS-SQL Server 2005 на Oracle 11gR2. Система содержала около 2,6 миллионов строк кода, 2430 динамических запросов, из которых больше 75% могут принимать более одного значения. Реализация алгоритма была внедрена в проект по миграции, заменив ранее используемую версию алгоритма синтаксического анализа. Алгоритм запускался на данной системе 10 раз, время анализа усреднялось.

Алгоритм успешно завершил работу на 2188 входных графах, аппроксимирующих множества значений запросов. Ручная проверка входных графов, на которых алгоритм завершался с ошибкой, показала, что они действительно не содержали ни одного корректного в эталонном языке выражения. Причиной этого стала либо некорректная работа токенизатора, либо наличие в выражениях конструкций, не поддержанных в существующей грамматике. Дальнейшие значения приводятся только для графов, которые удалось проанализировать. 604 из этих графов содержали ровно один путь и анализировалось не более 1 миллисекунды. Общее время синтаксического анализа составило порядка 27 минут, из них 13 минут было затрачено на разбор графов, не содержащих ни одного корректного выражения. В среднем один такой граф анализировался 386 миллисекунд. На разбор 1790 графов ушло не более 10 миллисекунд. На анализ двух графов было затрачено более 2 минут: 152,215 и 151,793 секунд соответственно. Первый граф содержал 2454 вершин и 54335 рёбер, второй — 2212 вершин и 106020 рёбер. Распределение входных графов по промежуткам времени, затраченным на анализ, приведено на графике на рисунке 9.

Тестирование на реальных данных показало, что алгоритм применим для синтаксического анализа регулярной аппроксимации множества значений динамически формируемого выражения.

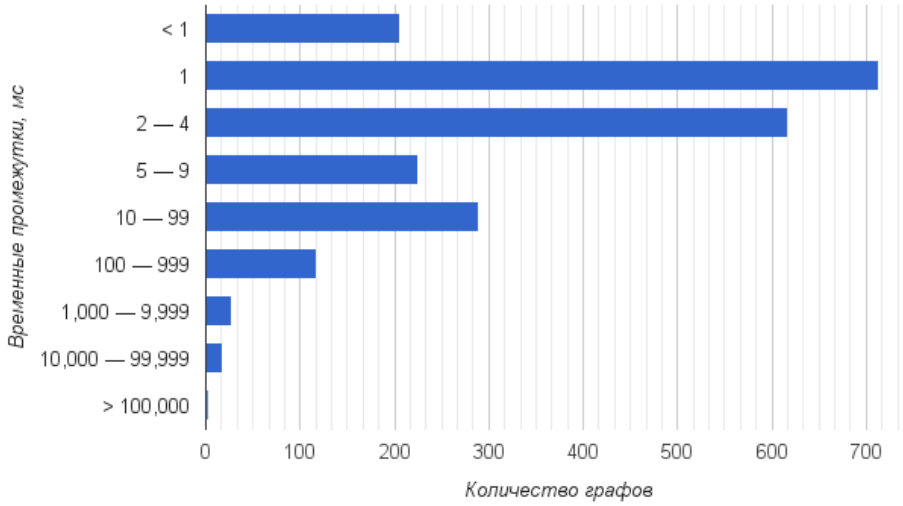

Рис. 9: Распределение запросов по времени анализа

Заключение

В ходе данной работы получены следующие результаты.

- Изучена предметная область: методы обработки встроенных языков и алгоритм обобщённого синтаксического анализа RNGLR.
- Разработан алгоритм синтаксического анализа динамически формируемых выражений, поддерживающий работу с произвольными входными графами.
- Доказана корректность алгоритма:
  - алгоритм завершит работу для любых входных данных;
  - для любой цепочки из входного множества, выводимой в эталонной грамматике G, в SPPF содержится её дерево вывода в G; при этом никакие другие деревья не содержатся в SPPF.
- Выполнена реализация алгоритма на языке программирования F# в рамках исследовательского проекта YaccConstructor.
- Проведена апробация: регрессионное тестирование, тестирование производительности и тестирование на реальных данных.
- Исходный код проекта YaccConstructor можно найти на сайте `https://github.com/YaccConstructor/YaccConstructor`, автор принимал участие под учётной записью kajigor.

В дальнейшем планируется изменить алгоритм таким образом, чтобы помимо построения леса разбора всех корректных выражений осуществлялся бы также поиск ошибочных выражений и сообщение о них. Также необходимо произвести теоретическую оценку сложности алгоритма. Предложенный алгоритм планируется внедрить в инструмент по реинжинирингу информационных систем.

Список литературы

[1] 9075:1992 ISO. ISO/IEC. Information technology — Database languages — SQL. — 1992.

[2] Annamaa Aivar, Breslav Andrey, Vene Varmo. Using Abstract Lexical Analysis and Parsing to Detect Errors in String-Embedded DSL Statements // Proceedings of the 22nd Nordic Workshop on Programming Theory. — 2010. — P. 20–22.

[3] Asveld Peter R. J., Nijholt Anton. The Inclusion Problem for Some Subclasses of Context-free Languages. — Vol. 230. — Essex, UK : Elsevier Science Publishers Ltd., 1999. — December. — P. 247–256.

[4] Automata-based Symbolic String Analysis for Vulnerability Detection / Fang Yu, Muath Alkhalaf, Tevfik Bultan, Oscar H. Ibarra // Form. Methods Syst. Des. — 2014. — Vol. 44, no. 1. — P. 44–70.

[5] Christensen Aske Simon, Møller Anders, Schwartzbach Michael I. Precise Analysis of String Expressions // Proceedings of the 10th International Conference on Static Analysis. — SAS'03. — Berlin, Heidelberg : Springer-Verlag, 2003. — P. 1–18.

[6] Cousot Patrick, Cousot Radhia. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints // Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — Los Angeles, California : ACM Press, New York, NY, 1977. — P. 238–252.

[7] Doh Kyung-Goo, Kim Hyunha, Schmidt David A. Abstract Parsing: Static Analysis of Dynamically Generated String Output Using LR-Parsing Technology // Static Analysis. — Springer Berlin Heidelberg, 2009. — Vol. 5673 of Lecture Notes in Computer Science. — P. 256–272.

[8] Doh Kyung-Goo, Kim Hyunha, Schmidt David A. Abstract LR-parsing. — Berlin, Heidelberg : Springer-Verlag, 2011. — P. 90–109.

[9] Doh Kyung-Goo, Kim Hyunha, Schmidt David A. Static Validation of Dynamically Generated HTML Documents Based on Abstract Parsing and Semantic Processing // Static Analysis. — Springer Berlin Heidelberg, 2013. — Vol. 7935 of Lecture Notes in Computer Science. — P. 194–214.

[10] Grigorev Semen, Kirilenko Iakov. GLR-based Abstract Parsing // Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia. — CEE-SECR '13. — New York, NY, USA : ACM, 2013. — P. 5:1–5:9.

[11] Houglan Damon, Tavistock Aaron. Core JSP. — Prentice Hall PTR, 2000. — October.

[12] An Interactive Tool for Analyzing Embedded SQL Queries / Aivar Annamaa, Andrey Breslav, Jevgeni Kabanov, Varmo Vene // Programming Languages and Systems. — 2010. — P. 131–138.

[13] Kirilenko Iakov, Grigorev Semen, Avdiukhin Dmitriy. Syntax Analyzers Development in Automated Reengineering of Informational System // St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications and Control Systems. — 2013. — Vol. 174, no. 3. — P. 94–98.

[14] PHP mySQL interface. — URL: `http://php.net/manual/en/mysqli.query.php` (online; accessed: 11.06.2015).

[15] Rekers Jan. Parser Generation for Interactive Environments. — 1992.

[16] String-embedded Language Support in Integrated Development Environment / Semen Grigorev, Ekaterina Verbitskaia, Andrei Ivanov et al. // Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia. — CEE-SECR '14. — New York, NY, USA : ACM, 2014. — P. 21:1–21:11.

[17] Syme Don, Granicz Adam, Cisternino Antonio. Expert F# (Expert's Voice in .Net). — ISBN: 1590598504, 9781590598504.

[18] Tomita Masaru. An Efficient Context-free Parsing Algorithm for Natural Languages // Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2. — IJCAI'85. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1985. — P. 756–764.