САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Григорьев Семён Вячеславович

Синтаксический анализ регулярных множеств

Выпускная квалификационная работа аспиранта

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

Научный руководитель:
к. ф.-м. н., доцент Кознов Д. В.

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

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Chair of Software Engineering

Semyon Grigorev

Parsing of Regular Sets

Graduation Thesis

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

Scientific supervisor:
Ph.D., associate professor Dmitrij Koznov

Reviewer:
Software Developer at IntelliJ Labs Andrey Breslav

Saint-Petersburg
2015

Оглавление

Введение

Организация взаимодействия между разнородными частями информационной системы — задача достаточно сложная. Примерами таких компонент могут служить базы данных, браузеры и элементы, реализующие логику приложения. Один из первых способов взаимодействия между этими компонентами, получивший широкое распространение благодаря своей гибкости и простоте, основан на передаче текстовых команд на языке, поддерживаемом системой, к которой идёт обращение. Части команд обычно хранятся в строковых литералах из которых в процессе работы программы формируется команда, передаваемая на выполнение. Примерами использования динамически формируемых строковых выражений могут служить динамические SQL-запросы к базам данных в java-коде или динамическое формирование HTML-страниц (листинги ??, ??, ??).

Листинг 1 Код с использованием динамического SQL

```
CREATE PROCEDURE [dbo].[MyProc]  @TABLERes   VarChar(30)
AS
    EXECUTE ('INSERT INTO ' + @TABLERes + ' (sText1)' +
             ' SELECT ''Additional condition: '' + sName' +
             ' from #tt where sAction = ''1000000''')
GO
```

Листинг 2 Вызов JavaScript из Java

```
import javax.script.*;
public class InvokeScriptFunction {
    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("JavaScript");
        // JavaScript code in a String
        String script =
            "function hello(name) { print('Hello, ' + name); }";
        // evaluate script
        engine.eval(script);
        // javax.script.Invocable is an optional interface.
        // Check whether your script engine implements or not!
        // Note that the JavaScript engine implements
        // Invocable interface.
        Invocable inv = (Invocable) engine;
        // invoke the global function named "hello"
        inv.invokeFunction("hello", "Scripting!!" );
    }
}
```

Листинг 3 Использование нескольких встроенных в PHP языков (MySQL, HTML)

```
<?php
    // Embedded SQL
    $query = 'SELECT * FROM ' . $my_table;
    $result = mysql_query($query);

    // HTML markup generation
    echo "<table>\n";
    while ($line = mysql_fetch_array($result, MYSQL_ASSOC)) {
        echo "\t<tr>\n";
        foreach ($line as $col_value) {
            echo "\t\t<td>$col_value</td>\n";
        }
        echo "\t</tr>\n";
    }
    echo "</table>\n";
?>
```

Несмотря на развитие метапрограммирования, техник порождающего программирования (generative programming), ORM-технологий, использование в языках программирования динамически формируемых строковых выражений все еще широко распространено. Популярность такого подхода связана также с тем, что выполнение динамически формируемых строк требует существенно меньших накладных расходов, чем применение, например, ORM, что позволяет существенно улучшить производительность системы.

Однако использование динамически формируемых строковых выражений сопряжено с рядом трудностей. Их конструирование часто происходит с применением конкатенации в циклах, ветках условных операторов или рекурсивных процедурах и их композициях, что порождает множество различных значений для каждого выражения в момент выполнения. При этом фрагменты кода на встроенных языках воспринимаются компилятором исходного языка как простые строки, не подлежащие дополнительному анализу. Таким образом, стандартные средства не позволяют проводить даже простой синтаксический анализ динамически формируемых выражений. Невозможность статической проверки корректности формируемого выражения приводит к высокой вероятности возникновения ошибок во время выполнения программы. В худшем случае такая ошибка не приведет к прекращению работы приложения, что указало бы на проблемы, однако целостность данных при этом может оказаться нарушена. Более того, использование динамически формируемых выражений затрудняет как разработку информационных систем, так и реинжиниринг уже созданных. Распространённой практикой при написании кода является использование интегрированных сред разработки, производящих подсветку синтаксиса и автодополнение, сигнализирующих о синтаксических ошибках, предоставляющих возможность прово-

дить рефакторинг кода. Такая функциональность значительно упрощает процесс разработки и отладки приложений и её реализация не только для основного языка, но и для встроенных языков будет полезной для разработчиков. Для задач реинжиниринга важно иметь возможность изучать систему и модифицировать её, сохраняя функциональность. Наличие динамически формируемых выражений затрудняет решение данных задач. Например, при наличии встроенного SQL нельзя точно ответить на вопрос о том, с какими элементами базы данных не взаимодействует система, и удалить их, не проанализировав все динамически формируемые выражения. При переносе такой системы на другую СУБД необходимо гарантировать, что для всех динамически формируемых выражений значение в момент выполнения будет корректным кодом на языке новой СУБД, что требует трансляции встроенных запросов [**?**].

Решение большинства перечисленных выше задач требует проведения синтаксического анализа. Более того, многие задачи требуют не только ответа на вопрос принадлежности некоторому языку всех генерируемых программой предложений, но и построения леса разбора. Так как множество значений выражения может быть бесконечным и сами предложения могут быть бесконечными, то проведение анализа наивным образом с явным построением леса разбора невозможно. Данная проблема затрагивается в исследованиях Kyung-Goo Doh [**?**, **?**, **?**], в которых предлагается комбинация анализа потока данных и синтаксического анализа на основе LR-алгоритма. В работах поднимается вопрос семантического анализа встроенных языков и предлагается использовать классический для LR-анализа механизм атрибутных грамматик, однако, опускается вопрос ресурсоёмкости данного подхода при нетривиальном анализе. Вместе с тем, в работах А. Бреслава [**?**, **?**] рассматривается подход, основанный на построении регулярной аппроксимации множества возможных значений динамически формируемых выражений и последующем анализе с использованием обобщённого LR-алгоритма. Однако этот вопрос изучен не полностью: не рассмотрено хранение результатов разбора с использованием сжатого представления леса вывода и повышение эффективности самого синтаксического анализа при полном использовании структурированного в виде графа стека и механизмов управления им.

В данной работа представлен алгоритм синтаксического анализа произвольного регулярного множества, основанный на механизмах обобщённого синтаксического LR-анализа. В результате анализа строится компактная структура данных, содержащая деревья разбора всех элементов множества, для которых возможен вывод, пригодная для дальнейшего семантического анализа.

1. 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.

1.1. Подходы к анализу встроенных языков

Анализ динамически формируемых выражений актуален как в задачах обеспечения безопасности программного обеспечения (поиск мест в коде, уязвимых для SQL-инъекций [**?**]), так и для разработки, сопровождения и модернизации систем, разработанных с применением встроенных языков. Для решения подобных задач существует ряд различных подходов, основные из которых рассмотрены ниже.

Проверка включения языков. В рамках данного подхода в результате анализа внешнего кода строится язык L_1, являющийся приближением языка L, генерируемого программой. После чего проверяется включение L_1 в язык $L_2(G)$, описанный эталонной грамматикой G. Основной недостаток данного подхода — невозможность получить какую-либо информацию, кроме знания о вхождении или не вхождении одного языка в другой. Как следствие, проведение более сложных видов статического анализа или трансформации невозможно. Можно выделить несколько вариантов данного подхода, различающихся классом языка L_1.

- Регулярная аппроксимация: L_1 является регулярным языком. Однако язык L не обязан быть регулярным, так как программа-генератор может быть реализована на тьюринг-полном языке, что может приводить к существенной потере точности при построении приближения. Достоинством такого подхода является разрешимость задачи проверки включения L_1 в L_2 для регулярных L_1 и L_2, являющегося однозначным контекстно-свободным языком [**?**]. Инструмент, реализующий данный подход, — Java String Analizer [**?**], являющийся анализатором строковых выражений в коде на Java.

- Контекстно-свободное приближение: L_1 является контекстно-свободным языком. Достоинством такого приближения является его бóльшая точность, однако проверка включения одного контекстно-свободного языка в другой является неразрешимой в общем случае задачей [**?**]. По этой причине при использовании такого приближения будет получено неточное решение, так как потребуется применение эвристик. Данный подход реализован в инструменте PHPSA [**?**], предназна-

ченном для проверки корректности динамически формируемых программами на PHP выражений.

Синтаксический анализ. Данный подход основан на применении техник синтаксического анализа для работы с динамически формируемыми выражениями. Благодаря этому, кроме проверки корректности выражений, становится возможным решение более сложных задач, требующих знаний о структуре вывода или работы с деревом разбора, таких как семантический анализ или трансформации. Ниже перечислены существующие на текущий момент варианты данного подхода.

- Абстрактный LR-анализ. В исследованиях группы во главе с Kyung-Goo Doh предлагается комбинация анализа потока данных и синтаксического анализа на основе LALR(k) алгоритма, позволяющая строить множество LR-стеков для всех значений строкового выражения [**?**, **?**, **?**]. Так как задача проверки включения для двух контекстно-свободных языков неразрешима, то представлено приближённое решение. В работе [**?**] обоснована возможность семантического анализа на основе классического для LR-анализа механизма: атрибутных грамматик [**?**] и выполнения семантического действия при выполнении свёртки. Однако не до конца исследована эффективность данного подхода при работе с семантическими действиями, требующими больших ресурсов при вычислении.

- Синтаксический анализ регулярного множества. Для языка L строится регулярная аппроксимация. Далее над построенной аппроксимацией решаются задачи лексического и синтаксического анализа. Данный подход рассмотрен в работах [**?**, **?**] и реализован в инструменте Alvor. Данный инструмент является плагином к среде разработки Eclipse, предоставляющим поддержку встроенного SQL в Java: статический поиск ошибок, тестирование запросов в базе данных. Достоинством такого подхода является разделение обработки на независимые шаги: построение аппроксимации, лексический анализ, синтаксический анализ [**?**]. Это позволяет более гибко переиспользовать существующие реализации тех или иных шагов и упрощает создание нового инструмента на базе имеющихся. Использование атрибутных грамматик — классического для LR-анализа способа задания семантики — и построение леса разбора в рамках данного подхода также не обсуждается.

1.2. Обзор инструментов для работы со встроенными языками

Задачи анализа динамически формируемых строковых выражений возникают в различных контекстах и применительно к различным языкам, что приводит к появлению разнообразных программных инструментов.

Среди языков, код на которых динамически формируется в виде строк, одним из наиболее распространённых является SQL с его многочисленными диалектами. При этом часто используется динамический SQL: генерация выражений на SQL в рамках кода на SQL, часто в хранимых процедурах. Одна из актуальных задач, при решении которой необходимо обрабатывать динамический SQL, — это миграция приложений баз данных. Для её решения существует ряд промышленных инструментов. В силу особенностей решаемой задачи нас интересуют инструменты для трансляции хранимого кода приложений баз данных. Самыми известными в данной области являются такие инструменты как PL-SQL Developer [**?**], SwisSQL [**?**], SQL Ways [**?**]. Эти инструменты применяются для трансляции хранимого SQL-кода, однако только SQL Ways обладает возможностью трансформации строковых SQL-запросов в ряде простых случаев. Динамически формируемые запросы со сложной логикой построения не поддерживаются современными промышленными инструментами.

Далее рассмотрим инструменты, которые изначально ориентированы на решение различных задач анализа динамически формируемых выражений. Многие из них предназначены для предоставления поддержки встроенных языков в интегрированных средах разработки. Как правило, эти инструменты реализуют один из основных подходов, описанных в разделе **??**.

Java String Analyzer (JSA, [**?**, **?**]) — инструмент для анализа строк и строковых операций в программах на Java. Основан на проверке включения регулярной аппроксимации встроенного языка в контекстно-свободное описание эталонного. Для каждого строкового выражения строится конечный автомат, представляющий приближенное значение всех значений этого выражения, которые могут быть получены во время выполнения программы. Для того, чтобы получить этот конечный автомат, необходимо из графа потока данных анализируемой программы построить контекстно-свободную грамматику, которая получается в результате замены каждой строковой переменной нетерминалом, а каждой строковой операции — правилом продукции. После этого полученная грамматика аппроксимируется регулярным языком. В качестве результата работы данный инструмент также возвращает строки, которые не входят в описанный пользователем язык, но могут сформироваться во время исполнения программы.

PHP String Analyzer (PHPSA, [**?**, **?**]) — инструмент для статического анализа строк в программах на PHP. Расширяет подход инструмента JSA [**?**]. Использует контекстно-свободную аппроксимацию, что достигается благодаря отсутствию этапа преобразования контекстно-свободной грамматики в регулярную, и это повышает точность проводимого анализа. Для того, чтобы обрабатывать строковые операций и учитывать их при построении контекстно-свободной грамматики, используется конечный преобразователь. Дальнейший анализ строковых выражений полностью заимствован из инструмента JSA.

Alvor [?, ?, ?] — плагин к среде разработки Eclipse[1], предназначенный для статической проверки корректности SQL-выражений, встроенных в Java[2]. Для компактного представления множества динамически формируемого строкового выражения используется понятие абстрактной строки, которая, фактически, является регулярным выражением над используемыми в строке символами. В инструменте Alvor отдельным этапом выделен лексический анализ. Поскольку абстрактную строку можно преобразовать в конечный автомат, то лексический анализ заключается в преобразовании этого конечного автомата в конечный автомат над терминалами при использовании конечного преобразователя, полученного генератором лексических анализаторов JFlex [?]. Несмотря на то, что абстрактная строка позволяет конструировать строковые выражения при участии циклов, плагин в процессе работы выводит сообщение о том, что не может поддержать данные языковые конструкции. Также инструмент Alvor не поддерживает обработку строковых операций, за исключением конкатенации, о чём так же выводится сообщение во время работы.

IntelliLang [?] — плагин к средам разработки PHPStorm [?] и IntelliJ IDEA[3], предоставляющий поддержку встроенных строковых языков, таких как HTML, SQL, XML, JavaScript в указанных средах разработки. Плагин обеспечивает подсветку синтаксиса, автодополнение, статический поиск ошибок. Для среды разработки IntelliJ IDEA расширение IntelliLang также предоставляет отдельный текстовый редактор для работы со встроенным языком. Для использования данного плагина требуется ручная разметка переменных, содержащих выражения на том или ином встроенном языке.

PHPStorm [?] — интегрированная среда разработки для PHP, которая осуществляет подсветку и автодополнение встроенного кода на HTML, CSS, JavaScript, SQL. Однако такая поддержка осуществляется только в случаях, когда строка получена без использования каких-либо строковых операций. Также PHPStorm для каждого встроенного языка предоставляет отдельный текстовый редактор.

Varis [?] — плагин для Eclipse, представленный в 2015 году и предоставляющий поддержку кода на HTML, CSS и JavaScript, встроенного в PHP. В плагине реализованы функции подсветки встроенного кода, автодополнения, перехода к объявлению (jump to declaration), построения графа вызовов (call graph) для встроенного JavaScript.

Абстрактный синтаксический анализ. Kyung-Goo Doh, Hyunha Kim, David A. Schmidt в серии работ [?, ?, ?] описали алгоритм статического анализа динамически формируемых строковых выражений на примере статической проверки корректности динамически генерируемого HTML в PHP-программах. Хотя для данного примера от-

[1]Сайт среды разработки Eclipse: `http://www.eclipse.org/ide/` (Посещён 23.06.2015.)

[2]Во время написания данного текста велась работа над поддержкой встроенных языков в PHP.

[3]IntelliJ IDEA — среда разработки для JVM-языков. Сайт: `https://www.jetbrains.com/idea/` (Посещён 23.06.2015.)

сутствует этап проведения лексического анализа, в общем случае можно использовать композицию лексического анализа и синтаксического разбора. Для этого достаточно хранить состояние конечного преобразователя, который используется для лексического анализа, внутри состояния синтаксического разбора. Данный алгоритм также предусматривает обработку строковой операции `string-replacement` с использованием конечного преобразователя, который по аналогии с лексическим конечным преобразователем хранит своё состояние внутри состояния синтаксического разбора. На вход абстрактный синтаксический анализатор принимает data-flow уравнения, полученные при анализе исходного кода, и LALR(1)-таблицу. Далее производится решение полученных на вход уравнений в домене LR-стеков. Проблема возможного бесконечного роста стеков, возникающая в общем случае, разрешается с помощью абстрактной интерпретации (abstract interpretation [**?**]). В работе [**?**] данный подход был расширен вычислением семантики с помощью атрибутных грамматик, что позволило анализировать более широкий, чем LALR(1), класс грамматик. В качестве результата алгоритм возвращает набор абстрактных синтаксических деревьев. На текущий момент реализацию данного алгоритма в открытом доступе найти не удалось, хотя в работах авторов приводятся результаты апробации. Таким образом, на данный момент не существует доступного инструмента, основанного на данном алгоритме.

1.3. Регулярная аппроксимация множества значений динамически формируемых выражений

Программа, использующая встроенные текстовые языки, является программой-генератором: она может порождать некоторое множество предложений и определяет некоторый язык L. Многие практически значимые задачи, связанные с динамически формируемыми выражениями, среди которых самая активно исследуемая — проверка корректности генерируемых предложений, связаны с проверкой включения языка L в некоторый другой язык L_2. Как правило, язык L_2 является некоторым языком программирования, выражения на котором должна генерировать программа (SQL, HTML). Язык L_2 часто является контекстно-свободным (КС) языком. Однако проверка включения для двух КС языков неразрешима, но проверка включения регулярного языка в некоторые практически значимые подклассы КС языков (LL(k), детерминированные КС) разрешима [**?**]. По этой причине в ряде работ [**?**, **?**, **?**], посвящённых анализу динамически формируемых выражений, используется регулярное приближение языка L. Важно, что можно построить регулярный язык, являющийся приближением сверху для языка L, то есть аппроксимацию сверху (over-approximation): регулярный язык содержит все предложения, генерируемые программой и, возможно, ещё какие-то. Благодаря этому можно говорить о достоверности многих видов статического анализа, например поиска ошибок, в том смысле, что если алгоритм анализа

регулярной аппроксимации не сообщил о наличии ошибок, то все выражения языка L корректны. Однако, могут быть ложные срабатывания: ошибочное предложение может принадлежать аппроксимации, но не принадлежать языку L.

Таким образом, регулярная аппроксимация для множества значений динамически формируемых выражений позволяет решать многие важные задачи. На практике для её представления можно использовать конечные автоматы, так как для любого регулярного языка L можно построить конечный автомат, такой что он принимает те и только те цепочки, которые принадлежат языку L. В данной работе используется регулярная аппроксимация множества значений строковых выражений, представленная в виде конечного автомата. Для её построения используется алгоритм, изложенный в работе [**?**].

1.4. RNGLR-алгоритм

RNGLR-алгоритм (Right-Nulled Generalized LR) является модификацией предложенного Масару Томитой алгоритма Generalized LR (GLR) [**?**], предназначенного для анализа естественных языков. GLR-алгоритм был предназначен для анализа неоднозначных контекстно-свободных грамматик. В процессе работы с такими грамматиками не всегда на основе имеющейся информации можно однозначно принять решение о следующем шаге: выполнить чтение очередного токена или выполнить свёртку (Shift/Reduce конфликт); по какому из правил выполнять свёртку(Reduce/Reduce конфликт). Это приводит к тому, что для одной входной цепочки можно получить несколько вариантов разбора и, соответственно, несколько LR-стеков.

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 $\rightarrow \alpha\beta$, 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 соответствует 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

Листинг 4 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)
```

Листинг 5 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
```

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 **??**.

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, используется компактное представление леса разбора (Shared Packed Parse Forest, SPPF) [**?**], предложенное Rekers []. В общем случае SPPF может быть произвольным связанным графом, однако из него всегда можно восстановить те и только те деревья, которые соответствуют какому либо варианту разбора данной входной цепочки. Это достигается благодаря тому, что в SPPF кроме вершин, соответствующих непосредственно узлам синтаксического дерева, могут присутствовать специальные типы вершин, отвечающие за корректное переиспользование поддеревьев. Структура SPPF описана ниже.

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.

Листинг **??** представляет более детальное описание алгоритма.

1.5. Проект YaccConstructor и платформа для анализа встроенных языков

В рамках проекта YaccConstructor [**?**] 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 [**?**].

Предыдущая реализация платформы работала с грубой аппроксимацией, которая не осуществляла поддержку формирования выражения в циклах и с помощью строковых выражений [**?**]. 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.

2. Описание алгоритма ослабленного синтаксического анализа регулярной аппроксимации

Алгоритм принимает на вход эталонную однозначную контекстно-свободную грамматику $G = \langle T, N, P, S \rangle$ 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, и не содержит 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, см. листинг **??**).

Листинг 6 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 стек в виде GSS подобно тому, как это делается RNGLR-алгоритме. 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 изменяет операции shift и reduce (смотри строку 5 в алгоритме **??** и строки 9 и 21 в алгоритме **??**). Для того, чтобы управлять порядком обработки вершин входного графа, мы используем глобальную очередь $\mathcal{Q}$. Каждый раз, когда добавляется новая вершина в GSS, сначала необходимо произвести все свёртки длины 0, и после этого выполнить сдвиг следующих токенов входного потока. Таким образом необходимо добавить соответствующую вершину графа в очередь на обработку. Добавление нового ребра в 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 листинге **??**. 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 свертки (смотри функцию applyPassingReductions в листинге **??**).

Листинг 7 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 внутренним графом) посредством копирования графа входного автомата и связывания с его вершинами следующих коллекций:

- processed — содержит вершины GSS, для которых ранее были вычислены все операции push; это множество агрегирует все вершины GSS, ассоциированные с вершиной внутреннего графа;
- unprocessed — содержит вершины GSS, операции push для которых ещё толь-

ко предстоит выполнить; это множество аналогично множеству $\mathcal{Q}$ алгоритма RNGLR;

- reductions — это очередь, аналогичная очереди $\mathcal{R}$ RNGLR-алгоритма: все операции reduce, которые ещё только предстоит выполнить;

- passingReductionsToHandle — содеожит пары (a, b), где a — это вершина GSS, а b — это ребро GSS, вдоль которого необходимо осуществлять проходящие свёртки.

Листинг 8 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 также коллекция проходящих свёрток, то есть троек $(startV, N, l)$, соответствующих свёрткам, 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. листинг **??**).

2.1. 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, так же 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 ребер, 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 ребер 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.

Представленный в листинге **??** код динамически формирует выражение **`expr`** в

строке 4.

Листинг 9 Пример кода на языке программирования C#, динамически формирующего скобочную последовательность

```
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 — открывающая скобка, а RBR — закрывающая. Граф конечного автомата, задающего такую аппроксимацию, изображён на рисунке **??**.

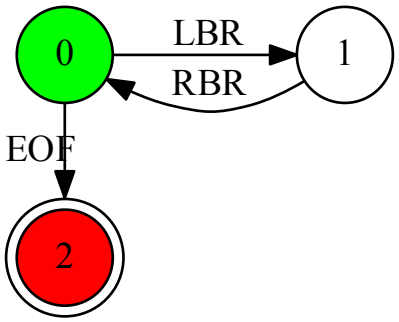

Рис. 1: Конечный автомат, задающий регулярную аппроксимацию выражения **expr**

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

Из построенного SPPF можно извлечь бесконечное количество деревьев, каждое из которых является деревом вывода некоторой цепочки из регулярной аппроксимации. На серии рисунков **??**, **??**, **??** представлены извлечённые деревья разбора для различных значений выражения **expr**.

2.2. Доказательство корректности алгоритма ослабленного синтаксического анализа регулярной аппроксимации

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 — это количество состояний синтаксического анализатора. Таким образом, количество вершин в графе GSS ограничено сверху числом $N \times n$, где n — это количество вершин графа входного автомата. Так как в GSS нет кратных ребер, количество его ребер — $O((N \times n)^2)$. На каждой итерации основного цикла алгоритм извлекает из очереди Q и обрабатывает одну вершину внутреннего графа. Вершины добавляются в очередь Q

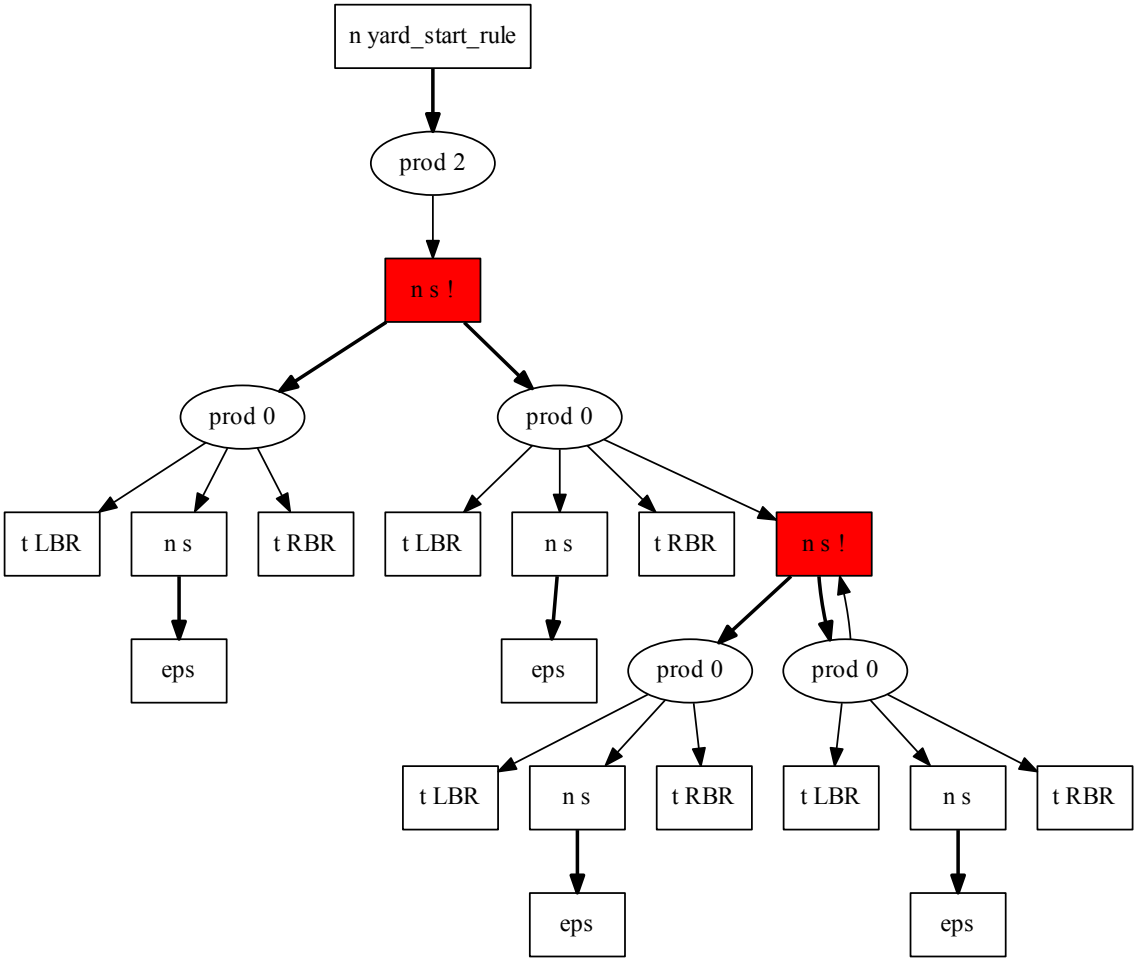

Рис. 2: Конечное представление леса разбора для выражения **`expr`**

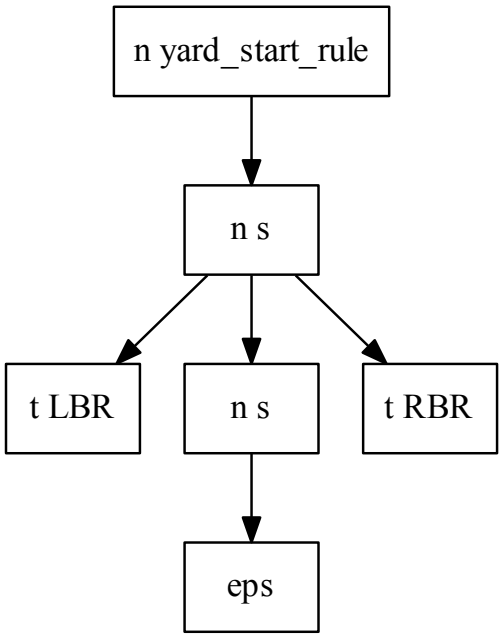

Рис. 3: Дерево вывода для выражения $expr =$ "()"

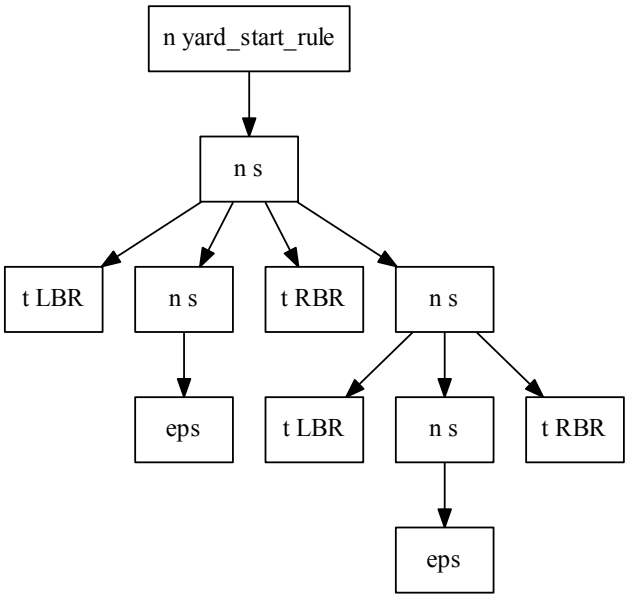

Рис. 4: Дерево вывода для выражения $expr =$ "()()"

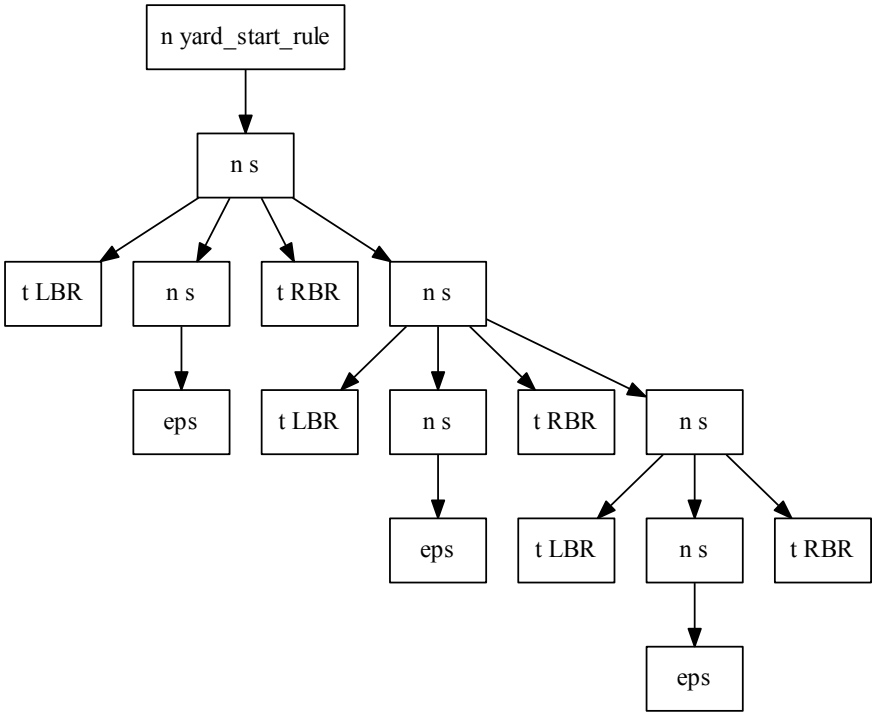

Рис. 5: Дерево вывода для выражения $expr =$ "()()()"

только тогда, когда происходит добавление нового ребра в GSS. Так как количество ребер 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, во-вторых, что для каждой корректной относительно эталонной грамматики цепочки существует корректное дерево вывода в SPPF.

Лемма. Пусть обрабатывается внутренний граф $\mathcal{G} = (V, E)$. Тогда для каждого ребра GSS (v_t, v_h) такого, что $v_t \in V_t.processed$, $v_h \in V_h.processed$, где $V_t \in V$ и $V_h \in V$, терминалы ассоциированного поддерева соответствуют некоторому пути из вершины V_h в V_t в графе $\mathcal{G}$.

Доказательство. Будем строить доказательство при помощи индукции по высоте дерева вывода. База индукции — это минимальное дерево: либо ε-дерево, либо дерево, состоящее из единственной вершины-терминала. При этом 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 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 ассоциированно 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. □

Так как SPPF является сжатым представлением леса разбора, то для получения конкретного дерева его необходимо извлечь из SPPF.

Теорема 2. Любое дерево, извлечённое из SPPF, является корректным.

Доказательство. Рассмотрим произвольное дерево, извлечённое из SPPF, 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 из GSS, начало которых лежит на последнем уровне стека и помечено принимающим состоянием, а конец — в вершинах на уровне 0. □

Теорема 3. Для каждой строки, соответствующей пути p во входном графе, и выводимой в эталонной грамматике G, из SPPF может быть извлечено корректное дерево t. То есть t будет являться деревом вывода цепочки, соответствующей пути p, в грамматике G.

Доказательство. Рассмотрим произвольное корректное дерево и докажем, что оно может быть извлечено из SPPF. Доказательство повторяет доказательство корректности для RNGLR-алгоритма [**?**], за исключением следующей ситуации. 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, так как во входном графе могут быть циклы и нет возможности упорядочить обработку его вершин. Это, в свою очередь, может приводить к появлению новых путей для свёрток, которые уже были ранее обработаны. Единственный возможный способ образования 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, то достаточно продолжить свёртки, проходящие через вершину v_t. Таким образом гарантируется выполнение всех возможных свёрток и построение корректного дерева вывода. □

Замечание. Построение леса разбора осуществляется одновременно с построением GSS, при этом дерево вывода нетерминала связывается с ребром GSS каждый раз при обработке соответствующей свёртки, вне зависимости от того, было ли ребро в графе до этого или 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.

3. Реализация и тестирование

Предложенный алгоритм был реализован на платформе .NET как часть проекта YaccConstructor; основным языком разработки являлся F# [**?**]. 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.

3.1. Экспериментальная оценка алгоритма

Алгоритм синтаксического анализа динамически формируемых выражений, описанный выше, 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 G_t, использованная в этих тестах.

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

Входные данные представляли собой конечные автоматы над алфавитом терминальных символов грамматики G_t, построенные с помощью конкатенации базовых блоков. Предполагается, что такие графы могут быть получены в результате построения регулярной аппроксимации по некоторой программе и выполнения её лексического анализа. В данном случае базовый блок — это шаблонный конечный автомат, который используется для построения тестовых конечных автоматов. 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 **??**, если истина — то изображённые на рисунке **??**).

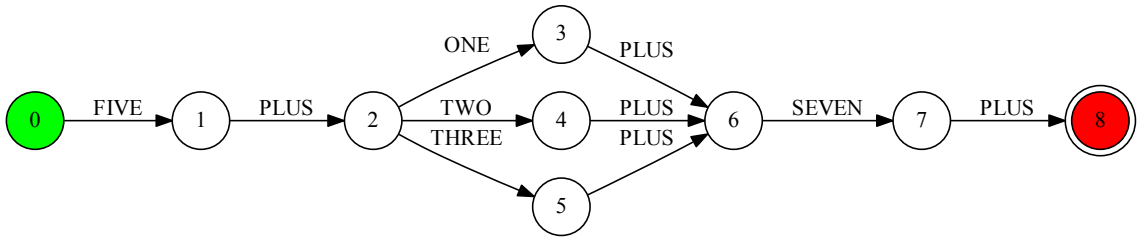

Рис. 6: Базовый блок без циклов при $height = 3$

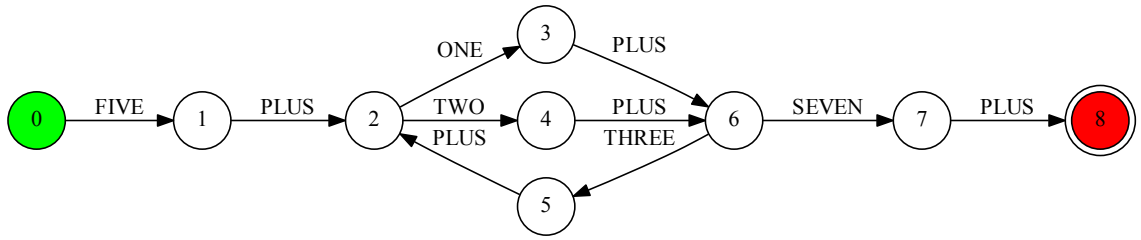

Рис. 7: Базовый блок, содержащий цикл, при $height = 3$

Замеры проводились на вычислительной станции со следующими характеристиками.

- Операционная система: Microsoft Windows 8.1 Pro

- Тип системы: x64-based PC
- Процессор: Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz, 3601 Mhz, 4 Core(s), 8 Logical Processor(s)
- Объём оперативной памяти: 16.0 GB

Чтобы выявить зависимость времени от размера входных данных, тесты проводились сериями. 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, то есть $length = i$ для i-того теста. Для каждого теста измерялось время, затраченное на синтаксический анализ. Измерения проводились 10 раз, после чего усреднялись. График, представленный на рисунке **??**, иллюстрирует зависимость времени, затрачиваемого на синтаксический анализ, от количества повторения базового блока и количества ветвлений в каждом из них. Можно заметить, что время, затрачиваемое на анализ, растёт линейно, в зависимости от размера входного графа. График на рисунке **??** показывает, что наличие циклов в графе, при одинаковом значении параметра $height$, увеличивает продолжительность анализа, при этом зависимость времени от размера графа остаётся линейной.

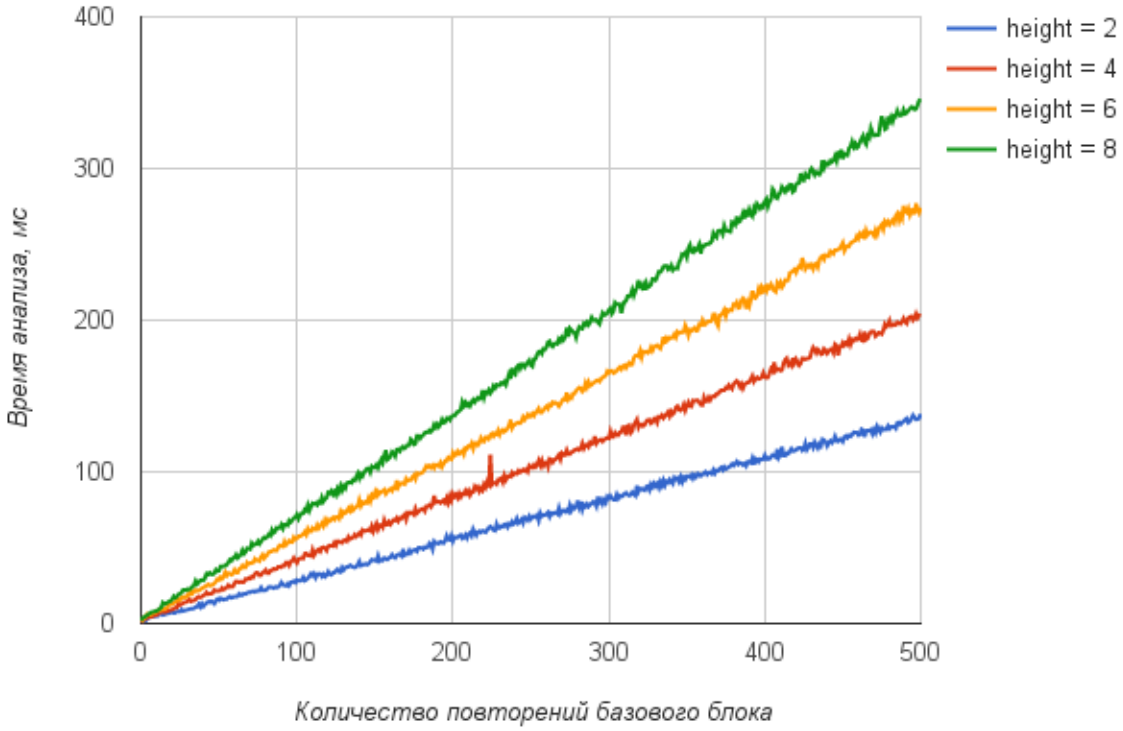

Рис. 8: Зависимость времени работы алгоритма от размера входного графа при $isCycle = false$

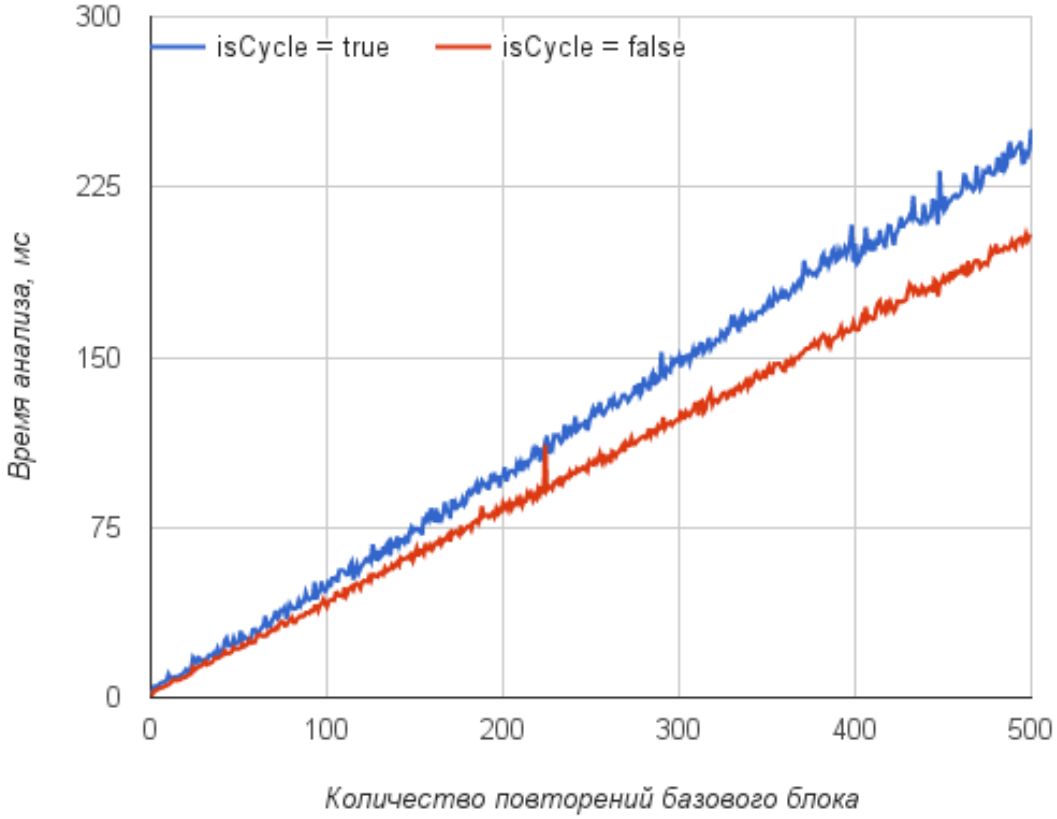

Рис. 9: Зависимость времени работы алгоритма от размера входного графа и наличия в нем циклов при $height = 4$

3.2. Апробация в промышленном проекте по реинжинирингу

Реализованный инструментарий был апробирован в рамках промышленного проекта по миграции базы данных с MS-SQL Server 2005 на Oracle 11gR2, что позволило апробировать как предложенную архитектуру, так и протестировать некоторые части инструментария на реальных данных.

Обрабатываемая система состояла из 850 хранимых процедур и содержала около 2,6 миллионов строк кода. В ней присутствовало 2430 точек выполнения динамических запросов, из которых больше 75% могли принимать более одного значения и при их формировании использовалось от 7 до 212 операторов. При этом среднее количество операторов для формирования запроса равнялось 40 [?].

Так как анализатор T-SQL был разработан ранее в рамках проекта, в котором происходило внедрение, то для создания анализатора встроенного SQL была использована готовая грамматика и по ней построен синтаксический анализатор. Построение регулярной аппроксимации и лексический анализ также были реализованы ранее в рамках основного проекта и были переиспользованы. Возможность использования компонент, созданных не в рамках YC.SEL.SDK, показало преимущества разделения шагов анализа.

Далее были реализованы функции вычисления метрик и вывода результата, после

чего полученная функциональность была встроена в существующую цепочку обработки основного кода. В результате работы реализованных функций формировался отчёт, пример которого приведён в таблице **??**

Тесты проводились на вычислительном устройстве с параметрами, эквивалентными указанным в разделе **??**. В ходе экспериментов измерялись следующие характеристики для каждой точки выполнения динамически формируемого запроса.

- Время обработки t в миллисекундах. Проводилось 10 запусков, время анализа усреднялось.
- Размер входного конечного автомата: количество состояний #Q и количество переходов #T.
- Размер построенного SPPF: количество вершин #V и количество рёбер #E.
- Результат анализа: + — завершился успешно, − — завершился с ошибкой, T — завершён по таймауту.

Таблица 1: Распределение динамически формируемых SQL-запросов по времени обработки

Категория	Количество запросов	Время обработки (минуты)
Содержат корректные выражения	2188	14
Не содержат ни одного корректного выражения	240	9
Обработка завершена по таймауту	1	4
Всего	**2430**	**27**

Результаты измерений времени работы представлены в таблице **??**. 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 лексического анализатора, либо наличие в выражениях конструкций, не поддержанных в существующей грамматике. Так как лексический анализатор и грамматика были полностью заимствованы из оригинального проекта, то наличие этих ошибок не является недоработками алгоритма синтаксического анализа. Общее время синтаксического анализа составило 27 минут, из них 13 минут было затрачено на разбор графов, не содержащих ни одного корректного выражения. Из них 256 секунд — обработка одного графа (5747 рёбер и 3897 вершин), прерванная по таймауту. Дальнейшие значения приводятся только для графов, которые удалось проанализировать. 604 из этих

графов прождали ровно одно значение и анализировалось не более 1 миллисекунды. 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, затраченных на анализ, приведено на графике на рисунке **??**.

Рис. 10: Распределение запросов по времени анализа

Тестирование на реальных данных показало, что предложенный в работе алгоритм синтаксического анализа применим для синтаксического анализа регулярной аппроксимации множества значений динамически формируемых выражений, используемых в коде промышленных информационных систем. Также данная апробация показала, что предложенная архитектура SDK позволяет использовать отдельные компоненты независимо и комбинировать их. Результаты успешно внедрены в проект компании ЗАО “Ланит-Терком”.

3.3. Сравнение с инструментом Alvor

Единственным доступным инструментом, производящим синтаксический анализ динамически формируемого кода, является Alvor [**?**, **?**]. Данный инструмент реализует близкий к представленному в работе подход: независимые шаги анализа, что позволяет легко выделить синтаксический анализ, который основан на GLR-алгоритме. Существенным отличием от разработанного алгоритма является то, что Alvor не строит деревья вывода. Важным для успешного проведения измерений является то, что

Таблица 2: Пример отчёта по результатам запуска синтаксического анализа на реальной системе

№	t(мс)	r	DFA		SPPF	
			#Q	#T	#V	#E
1	6	+	75	76	1074	1075
2	73	+	104	806	18786	26377
3	64	+	79	750	17237	23954
4	15982	+	817	32281	920618	1885112
5	3	+	108	107	996	995
6	256000	T	3897	5747	0	0
7	28360	+	924	41408	1315491	2794517
8	17	+	236	506	4608	5165
9	207	+	928	2249	38709	57352
10	110	+	390	942	14757	21805
11	111	+	377	967	14812	21902
12	262	+	764	1907	33332	49955
13	3	+	117	116	1093	1092
14	3	+	92	92	1391	1391

исходный код Alvor опубликован, что позволяет модифицировать его таким образом, чтобы измерять параметры выполнения конкретных методов.

Так как Alvor не предоставляет платформы для простой реализации поддержки новых языков, то для сравнения было выбрано подмножество языка SQL, общее для Alvor и реализованного в рамках апробации инструмента. Отсутствие возможности быстро построить новый анализатор на основе Alvor помешало сравнению на реальных данных, так как даже только спецификация грамматики T-SQL является задачей, требующей большого количества времени. По этой причине сравнение производилось на синтетических тестах, которые строились по принципу, аналогичному изложенному в разделе **??**.

Так как Alvor на вход принимает регулярное выражение, называемое абстрактной строкой [**?**], а анализатор, созданный на основе YC.SEL.SDK — конечный автомат, то был реализован генератор, который по входным параметрам создают файл с абстрактной строкой и с описанием соответствующего автомата в формате DOT. Синтаксис описания абстрактной строки приведён в листинге **??**. При этом, абстрактная строка подвергалась последовательно лексическому и синтаксическому анализу и замерялось время работы последнего, а конечный автомат строился сразу над алфавитом токенов и подвергался синтаксическому анализу.

Примеры тестовых входов для одинаковых входных параметров (однократного и двукратного повторения базовых блоков и $height = 2$) для инструмента на YC.SEL.SDK и Alvor представлены на рисунке **??** и листинге **??** соответственно.

Результаты измерений представлены в таблице **??** и на графике **??**. В легенде

Листинг 10 Синтаксис описания абстрактной строки

```
absStr = "str"
       | '{'absStr(',' absStr)+'}' //alternatives
       | absStr absStr             //concatenation
       | absStr '*'                //closure
```

Листинг 11 Пример абстрактных строк для $height = 2$ одного и двух повторений базового блока

```
"select "{"X3 + Y4","1"}",d from tbl"
"select "{"X3 + Y4","1"}","{"X7 + Y8","5"}",d from tbl"
```

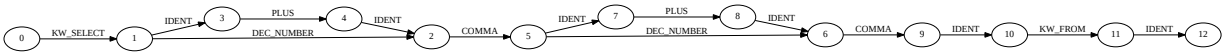

Рис. 11: Входной граф для синтаксического анализатора на базе YC.SEL.SDK при $height = 2$ и двух повторениях базового блока

и в заголовке таблицы указан инструмент (YC или Alvor) и значение параметра *height* (например, h=2). При более чем шестнадцатикратном повторении блоков с $height = 2$ не удалось получить результат от инструмента Alvor за разумное время. Аналогичная ситуация возникает и при более чем десятикратном повторении блоков с $height = 3$. Таким образом, измерения показывают, что время работы анализатора Alvor экспоненциально относительно количества повторений базового блока при $height > 1$. Анализатор созданный на основе YC.SEL.SDK в таких случаях имеет лучшую производительность. При этом на линейном входе Alvor показывает лучшую производительность. Однако, асимптотика YC.SEL.SDK на входных данных подобной структуры такая же как у оригинального RNGLR, что показано в предыдущих экспериментах. При этом реализация алгоритма может быть оптимизирована. Таким образом, производительность на линейном входе может быть улучшена.

В результате измерений выяснено, что производительность реализованного алгоритма синтаксического анализа лучше чем производительность аналогичного алгоритма, реализованного в инструменте Alvor на входных данных, содержащих большое количество ветвлений. При этом, Alvor не строит деревья разбора, в отличии от алгоритма, реализованного в данной работе.

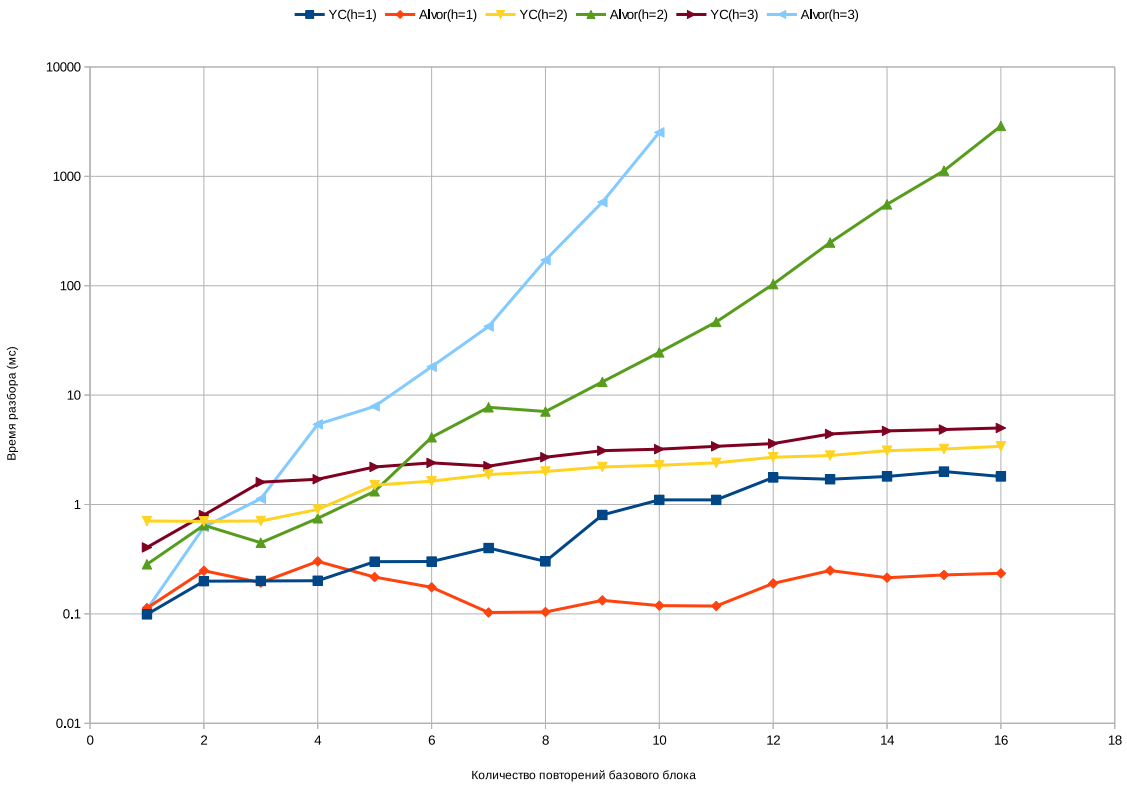

Рис. 12: Сравнение производительности Alvor и синтаксического анализатора на базе YC.SEL.SDK

Таблица 3: Результаты сравнения производительности Alvor и синтаксического анализатора на базе YC.SEL.SDK

№	YC(h=1)	Alvor(h=1)	YC(h=2)	Alvor(h=2)	YC(h=3)	Alvor(h=3)
1	0.099	0.113	0.706	0.284	0.405	0.11
2	0.199	0.248	0.702	0.646	0.801	0.622
3	0.2	0.193	0.707	0.447	1.601	1.129
4	0.201	0.302	0.901	0.748	1.701	5.403
5	0.3	0.217	1.502	1.32	2.203	7.89
6	0.301	0.175	1.635	4.114	2.402	18.187
7	0.4	0.103	1.877	7.734	2.24	42.447
8	0.302	0.104	2.002	7.076	2.704	171.529
9	0.802	0.133	2.202	13.204	3.104	580.545
10	1.102	0.119	2.282	24.578	3.204	2521.318
11	1.102	0.118	2.404	46.662	3.403	
12	1.766	0.19	2.704	103.417	3.605	
13	1.701	0.249	2.803	248.107	4.408	
14	1.803	0.214	3.103	554.314	4.706	
15	2.001	0.227	3.217	1125.976	4.843	
16	1.803	0.235	3.403	2886.261	5.006	

Заключение

В статье описан алгоритм синтаксического анализа регулярного множества. Данный алгоритм может быть применён, например, для синтаксического анализа динамически формируемых выражений, в случае, когда множество возможных значений выражения приближается регулярным множеством. Подобные задачи могут возникать при поддержке встроенных текстовых языков в интегрированных средах разработки, при анализе и модификации программного обеспечения в процессе реинжиниринга.

Представленный алгоритм был реализован на языке F# [**?**] в качестве модуля проекта YaccConstructor с переиспользованием ранее реализованного генератора парсеров на основе RNGLR-алгоритма. На основе нового модуля реализован плагин [**?**] к ReSharper, который предоставляет поддержку встроенного T-SQL в C#. Выполняется подсветка синтаксиса, подсветка парных элементов, статический поиск ошибок и их подсветка в редакторе. Исходный код опубликован в открытом доступе и доступен на сайте `https://github.com/YaccConstructor/YaccConstructor`.

В дальнейшем необходимо выполнить теоретическую оценку сложности алгоритма и требуемого объёма памяти. С практической точки зрения требуется оптимизация алгоритма и улучшение качества диагностики ошибок. Отдельных исследований требует работа с семантикой и возможность трансляции и трансформаций динамически формируемых выражений.

Список литературы

[1] Aho Alfred V., Sethi Ravi, Ullman Jeffrey D. Compilers: Principles, Techniques, and Tools. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1986. — ISBN: 0-201-10088-6.

[2] Alvor plug-in. — URL: `http://code.google.com/p/alvor/` (online; accessed: 11.06.2015).

[3] 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.

[4] 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.

[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 Parsing: Static Analysis of Dynamically Generated String Output Using LR-Parsing Technology // Proceedings of the 16th International Symposium on Static Analysis. — SAS '09. — Berlin, Heidelberg : Springer-Verlag, 2009. — P. 256–272.

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

[10] Doh Kyung-Goo, Kim Hyunha, Schmidt David A. Formal Modeling / Ed. by Gul Agha, José Meseguer, Olivier Danvy. — Berlin, Heidelberg : Springer-Verlag, 2011. — P. 90–109.

[11] 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.

[12] 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.

[13] Grigorev Semen, Kirilenko Iakov. From Abstract Parsing to Abstract Translation // Preliminary Proceedings of the 8th Spring/Summer Young Researchers Colloquium on Software Engineering. — 2013. — P. 135–139.

[14] 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.

[15] IntelliLang plug-in. — URL: `https://www.jetbrains.com/idea/help/intellilang.html` (online; accessed: 11.06.2015).

[16] An Interactive Tool for Analyzing Embedded SQL Queries / Aivar Annamaa, Andrey Breslav, Jevgeni Kabanov, Varmo Vene // Proceedings of the 8th Asian Conference on Programming Languages and Systems. — APLAS'10. — Berlin, Heidelberg : Springer-Verlag, 2010. — P. 131–138.

[17] JFlex. — URL: `http://jflex.de/` (online; accessed: 11.06.2015).

[18] Java String Analyzer. — URL: `http://www.brics.dk/JSA/` (online; accessed: 11.06.2015).

[19] 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.

[20] Minamide Yasuhiko. Static Approximation of Dynamically Generated Web Pages // Proceedings of the 14th International Conference on World Wide Web. — WWW '05. — New York, NY, USA : ACM, 2005. — P. 432–441.

[21] Nguyen Hung Viet, Kästner Christian, Nguyen Tien N. Varis: IDE Support for Embedded Client Code in PHP Web Applications // Proceedings of the 37th International Conference on Software Engineering (ICSE). — New York, NY : ACM Press, 2015. — Formal Demonstration paper.

[22] PHP String Analyzer. — URL: http://www.score.cs.tsukuba.ac.jp/~minamide/phpsa/ (online; accessed: 11.06.2015).

[23] PHPStorm IDE. — URL: https://www.jetbrains.com/phpstorm/ (online; accessed: 11.06.2015).

[24] PL/SQL Developer. — URL: http://www.allroundautomations.com/plsqldev.html (online; accessed: 11.06.2015).

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

[26] SQL Ways. — URL: http://www.ispirer.com/products (online; accessed: 11.06.2015).

[27] Scott Elizabeth, Johnstone Adrian. Right Nulled GLR Parsers // ACM Trans. Program. Lang. Syst. — 2006. — Vol. 28, no. 4. — P. 577–618.

[28] A Static Analysis Framework For Detecting SQL Injection Vulnerabilities / Xiang Fu, Xin Lu, Boris Peltsverger et al. // Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01. — COMPSAC '07. — Washington, DC, USA : IEEE Computer Society, 2007. — P. 87–96.

[29] 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.

[30] SwissSQL. — URL: http://www.swissql.com/ (online; accessed: 11.06.2015).

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

[32] 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.