

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

На правах рукописи  
УДК 519.686.4

Григорьев Семён Вячеславович

**Синтаксический анализ динамически формируемых  
строковых выражений**

Специальность 05.13.11  
«Математическое и программное обеспечение вычислительных машин,  
комплексов и компьютерных сетей»

Диссертация на соискание учёной степени  
кандидата физико-математических наук

Научный руководитель:  
кандидат физико-математических наук, доцент  
Кознов Дмитрий Владимирович

Санкт-Петербург — 2015

# Содержание

<b>Введение</b>	<b>4</b>
<b>1 Обзор</b>	<b>14</b>
1.1 Языки и грамматики	14
1.2 Конечные автоматы и преобразователи	19
1.3 О применимости статического анализа строковых выражений	22
1.4 Подходы к анализу встроенных языков	24
1.5 Обзор инструментов для работы со встроенными языками	26
1.6 Алгоритмы и структуры данных для обобщённого синтаксического анализа	29
1.6.1 Алгоритм обобщённого LR-анализа	30
1.6.2 Структурированный в виде графа стек	31
1.6.3 Сжатое представление леса разбора	32
1.6.4 Алгоритм RNGLR	33
1.7 Используемые инструменты	37
1.7.1 YaccConstructor	37
1.7.2 ReSharper SDK	40
1.8 Выводы	42
<b>2 Алгоритм синтаксического анализа регулярной аппроксимации</b>	<b>44</b>
2.1 Постановка задачи	44
2.2 Описание алгоритма ослабленного синтаксического анализа регулярной аппроксимации	47
2.2.1 Построение компактного представления леса разбора	51
2.3 Доказательство корректности алгоритма ослабленного синтаксического анализа регулярной аппроксимации	55

<b>3</b>	<b>Инструментальный пакет . . . . .</b>	<b>58</b>
3.1	Архитектура . . . . .	58
3.1.1	Архитектура YS.SEL.SDK . . . . .	60
3.1.2	Архитектура YC.SEL.SDK.ReSharper . . . . .	64
3.2	Области и способы применения YC.SEL.SDK . . . . .	68
3.3	Особенности реализации . . . . .	73
<b>4</b>	<b>Метод реинжиниринга встроенного программного кода . . . . .</b>	<b>75</b>
4.1	Особенности . . . . .	75
4.2	Метод . . . . .	77
<b>5</b>	<b>Эксперименты, ограничения, обсуждение . . . . .</b>	<b>90</b>
5.1	Экспериментальная оценка алгоритма . . . . .	90
5.2	Апробация в промышленном проекте по реинжинирингу . . . . .	92
5.3	Сравнение с инструментом Alvor . . . . .	97
5.4	Разработка расширений для поддержки встроенных языков . . . . .	99
5.5	Ограничения . . . . .	103
<b>6</b>	<b>Сравнение и соотнесение . . . . .</b>	<b>106</b>
	<b>Заключение . . . . .</b>	<b>110</b>
	<b>Литература . . . . .</b>	<b>112</b>
	<b>Список рисунков . . . . .</b>	<b>119</b>
	<b>Список таблиц . . . . .</b>	<b>121</b>

# Введение

## Актуальность работы

Статический анализ исходного кода — получение знаний о программе без её исполнения — является неотъемлемой частью многих процессов, связанных с разработкой программного обеспечения. Он может использоваться, например, в средах разработки для упрощения работы с кодом — подсветка синтаксиса, навигация по коду, контекстные подсказки; для обнаружения проблем на ранних стадиях (до запуска программы) — статический поиск ошибок. Кроме того, статический анализ используется при решении задач трансформации исходного кода и реинжиниринге [1]. Однако многие языки программирования позволяют использовать конструкции, которые существенно затрудняют статический анализ кода.

Например, взаимодействие приложений с базами данных, часто реализуется с помощью встроенных языков: приложение, созданное на одном языке, генерирует код на другом языке и передаёт этот код на выполнение в соответствующее окружение. Примерами могут служить не только динамические SQL-запросы к базам данных в Java-коде, но и формирование HTML-страниц в PHP-приложениях, динамический SQL (Dynamic SQL [2]), фреймворк JSP [3], PHP MySQL interface [4]. Генерируемый код собирается из строк таким образом, чтобы в момент выполнения результирующая строка представляла собой корректное выражение на соответствующем языке. Примеры использования встроенных языков представлены в листингах 0.1, 0.2 и 0.3.

Такой подход весьма гибок, так как позволяет использовать для формирования выражений различные строковые операции (replace, substring и т.д.) и получать части кода из различных источников (например, учитывать текстовый ввод пользователя, что часто используется для задания фильтров при конструировании SQL-запросов). Кроме того, использование динамически формируемых

строковых выражений избавлено от дополнительных накладных расходов, присущих таким технологиям, как ORM<sup>1</sup>, что позволяет достичь высокой производительности. В работе [5] утверждается, что использование динамического SQL растёт.

---

```

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

```

---

Листинг 0.1: Код с использованием динамического SQL

---

```

1 import javax.script.*;
2 public class InvokeScriptFunction {
3     public static void main(String[] args) throws Exception {
4         ScriptEngineManager manager = new ScriptEngineManager();
5         ScriptEngine engine = manager.getEngineByName("JavaScript");
6         // JavaScript code in a String
7         String script =
8             "function hello(name) { print('Hello, ' + name); }";
9         // evaluate script
10        engine.eval(script);
11        // javax.script.Invocable is an optional interface.
12        // Check whether your script engine implements or not!
13        // Note that the JavaScript engine implements
14        // Invocable interface.
15        Invocable inv = (Invocable) engine;
16        // invoke the global function named "hello"
17        inv.invokeFunction("hello", "Scripting!!" );
18    }
19 }

```

---

Листинг 0.2: Вызов JavaScript из Java

Динамически формируемые выражения часто конструируются посредством конкатенации в циклах, ветках условных операторов или рекурсивных процедурах, что приводит к получению множества различных значений для каждого выражения в момент выполнения. При этом фрагменты кода на встроенных языках воспринимаются компилятором исходного языка как простые строки, не

---

<sup>1</sup>Что-то там про ORM!!!

---

```

1 <?php
2     // Embedded SQL
3     $query = 'SELECT * FROM ' . $my_table;
4     $result = mysql_query($query);
5
6     // HTML markup generation
7     echo "<table>\n";
8     while ($line = mysql_fetch_array($result, MYSQL_ASSOC)) {
9         echo "\t<tr>\n";
10        foreach ($line as $col_value) {
11            echo "\t\t<td>$col_value</td>\n";
12        }
13        echo "\t</tr>\n";
14    }
15    echo "</table>\n";
16 ?>

```

---

Листинг 0.3: Использование нескольких встроенных в PHP языков (MySQL, HTML)

подлежащие дополнительному анализу. Невозможность статической проверки корректности формируемого выражения приводит к высокой вероятности возникновения ошибок во время выполнения программы. В худшем случае такая ошибка не приведёт 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. Для реинжиниринга важно иметь возможность изучать систему и модифицировать её, сохраняя функциональность. Однако, например, при наличии в коде приложения встроенного SQL нельзя, не проанализировав все динамически формируемые выражения, точно ответить на вопрос о том, с какими элементами базы данных не взаимодействует система, и удалить их. 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. С другой стороны, распространённой практикой при написании кода является использование интегрированных сред разработки, производящих подсветку синтаксиса и автодополнение, сигнализирующих о синтаксических ошибках, предоставляющих возможность проводить рефакторинг кода. Такая функциональность значительно упрощает процесс разработки и отладки

приложений и полезна не только для основного языка, но и для встроенных языков. Для решения таких задач необходимы инструменты, проводящие анализ множества выражений, которые могут быть получены на этапе исполнения из строковых выражений исходного языка.

Проблема анализа встроенных языков активно исследуется. Большинство работ используют анализ регулярного множества (регулярной аппроксимации), приближающего множество значений динамически формируемого выражения. Как правило, рассматривается вопрос корректности генерируемых выражений или ищутся фрагменты кода, уязвимые для SQL-инъекций. Сильная специализация таких решений не позволяет применять их для других задач. В исследованиях Кюн-Гу Дох (Kyung-Goo Doh) [6–8] предлагается комбинация анализа потока данных и синтаксического анализа на основе LR-алгоритма и поднимается вопрос о семантическом анализе встроенных языков. Предлагается использовать классический для LR-анализа механизм атрибутивных грамматик, однако опускается вопрос ресурсоёмкости данного подхода при сложных видах анализа. В работах А. Бреслава [9, 10] рассматривается подход, основанный на построении регулярной аппроксимации множества возможных значений и последующем анализе с использованием обобщённого LR-алгоритма, что кроме расширения класса поддерживаемых языков даёт дополнительные преимущества при переиспользовании структур данных, характерных для обобщённого анализа. Однако эффективное хранение результатов разбора и оптимизация управления стеком разбора не рассмотрены. Существует также ряд инструментов для работы с динамически формируемыми выражениями: Alvor [11] и IntelliLang [12], предоставляющие поддержку встроенных языков в интегрированных средах разработки, JSA [13] и PHPSA [14], позволяющие искать ошибки в выражениях на встроенных языках, SQLWays [15], поддерживающий трансформацию выражений на встроенных языках, SAFELI [16] — инструмент статического анализа, предназначенный для определения возможности SQL-инъекций в Web-приложениях и некоторые другие. Однако эти инструменты либо не поддерживают часто встречающиеся на практике сложные способы формирования выражений, либо имеют существенные ограничения по функциональности: не поддерживают сложные способы формирования строковых выражений, решают только одну узкую задачу (проверка корректности, поиск уязвимых конструкций) и т.д.

В рамках исследовательского проекта YaccConstructor [17], посвящённого проведению экспериментов в области синтаксического анализа, ведётся работа над платформой для создания инструментов, предназначенных для статического анализа кода на встроенных языках. В данной статье описаны разрабатываемая инфраструктура поддержки встроенных языков и её компоненты: генераторы абстрактных лексических и синтаксических анализаторов. Также уделено внимание поддержке многих языков и приведен пример использования платформы для создания плагина к ReSharper<sup>2</sup> (плагин к Microsoft Visual Studio<sup>3</sup>, расширяющий стандартные средства IDE), позволяющего анализировать динамически формируемые выражения.

### **Цель и задачи работы**

Целью данной работы является создание подхода к статическому анализу динамически формируемых строковых выражений, уменьшающего затраты для создания конечных инструментов, обеспечивающих обработку строковых выражений на различных этапах: в средах разработки (подсветка синтаксиса, статический поиск ошибок); оценка качества кода; трансформация кода, содержащего большое количество динамически формируемых строковых выражений.

### **Методы исследования**

В работе используется алгоритм обобщённого восходящего синтаксического анализа RNGLR [18], созданный Элизабет Скотт (Elizabeth Scott) и Адриан Джонстон (Adrian Johnstone) из университета Royal Holloway (Великобритания). Для компактного хранения леса вывода использовалась структура Shared Packed Parse Forest (SPPF) [19], которую предложил Ян Рекерс (Jan Rekers, University of Amsterdam).

Доказательство завершаемости и корректности предложенного алгоритма проводилось с применением теории формальных языков, теории графов и теории сложности алгоритмов. Приближение множества значений динамически формируемого выражения строилось в виде регулярного множества, описываемого с помощью конечного автомата.

Апробация созданного подхода проводилась в рамках промышленного проекта компании ЗАО “Ланит-Терком” (Россия) по переносу хранимого кода, со-

<sup>2</sup>Сайт проекта ReSharper: <http://www.jetbrains.com/resharper>

<sup>3</sup>Сайт проекта Microsoft Visual Studio: <http://www.visualstudio.com>



держающего большое количество динамического SQL, с MS SQL Server на Oracle Server. Предложенный в работе алгоритм апробирован в рамках инфраструктуры проекта ReSharper компании ООО “ИнтеллиДжей Лабс” (Россия).

### **Научная новизна работы**

На текущий момент существует несколько подходов к анализу динамически формируемых строковых выражений. Некоторые из них, такие как JSA, предназначены только для проверки корректности выражений, основанной на решении задачи о включении одного языка в другой. Выполнение более сложных видов анализа, трансформаций или построения леса разбора не предполагается. В работах А. Бреслава и Кюн-Гу Дох (Kyung-Goo Doh) рассматривается применение механизмов синтаксического анализа для работы с динамически формируемыми выражениями, однако не решается вопрос эффективного представления результатов разбора. Предложенный в диссертации алгоритм предназначен для синтаксического анализа динамически формируемых выражений и построения компактной структуры данных, содержащей для всех корректных значений выражения их дерева вывода. Это позволяет как проверять корректность анализируемых выражений, так и проводить более сложные виды анализов, используя деревья вывода, хранящиеся в построенной структуре данных.

Большинство существующих готовых инструментов для анализа динамически формируемых строковых выражений (JSA, PHPSA, Alvor и т.д.), как правило, предназначены для решения конкретных задач в рамках конкретных языков. Решение новых задач или поддержка других языков с помощью этих инструментов затруднено ввиду ограничений, накладываемых архитектурой и возможностями используемого алгоритма анализа. В рамках работы предложена архитектура, учитывающая возможности предложенного алгоритма и позволяющая упростить создание новых инструментов для анализа динамически формируемых выражений.

### **Практическая значимость работы**

На основе полученных в работе научных результатов был разработан инструментарий (SDK), предназначенный для создания средств статического анализа динамически формируемых выражений. В данный инструментарий входят следующие компоненты: генератор лексических анализаторов, генератор синтаксических анализаторов, библиотеки времени выполнения, реализующие соответ-

ствующие алгоритмы анализа, набор интерфейсов и вспомогательных функций для реализации конечного инструмента.

Набор генераторов позволяет по описанию лексики и синтаксиса языка строить синтаксический и лексический анализатор, обрабатывающий аппроксимацию множества значений динамически формируемого выражения на соответствующем языке, представленную в виде произвольного конечного автомата. Устранение эpsilon-переходов, необходимое для корректной работы синтаксического анализа, происходит на этапе лексического анализа.

Данный инструментарий позволяет автоматизировать создание лексических и синтаксических анализаторов при разработке программных средств, использующих регулярную аппроксимацию для приближения множества значений динамически формируемых выражений. Инструментарий может использоваться для решения задач реинжиниринга — изучения и инвентаризации систем, поиска ошибок в исходном коде, автоматизации трансформации выражений на встроенных языках. Также данный инструментарий может использоваться при реализации поддержки встроенных языков в интегрированных средах разработки.

Разработанная методика обработки динамического SQL основана на использовании инструментария в качестве генератора для создания лексического и синтаксического анализатора для динамически формируемых выражений по соответствующим спецификациям. В случае динамического SQL могут быть переиспользованы ранее разработанные спецификации. Построение регулярной аппроксимации выделяется в отдельный шаг и производится с помощью анализов, реализованных для обработки основного кода. После завершения синтаксического разбора, анализ леса проводится в основном с помощью тех же методов, что и анализ основного кода, что достигается за счёт идентичности структур деревьев. Данная методика может быть переиспользована для работы с произвольными встроенными текстовыми языками.

### **Положения, выносимые на защиту**

1. Разработан алгоритм синтаксического анализа динамически формируемых выражений, позволяющий обрабатывать произвольную регулярную аппроксимацию множества значений выражения в точке выполнения, реализующий эффективное управление стеком и гарантирующий конечность представления леса вывода.

2. Доказана завершаемость и корректность предложенного алгоритма при анализе регулярной аппроксимации, представимой в виде конечного автомата без  $\varepsilon$ -переходов.
3. Создана архитектура инструментария для разработки программных средств статического анализа динамически формируемых строковых выражений.
4. Создана методика обработки динамически формируемых строковых выражений в проектах по реинжинирингу информационных систем.

### **Степень достоверности и апробация работы**

С использованием разработанного инструментария было реализовано расширение к инструменту ReSharper (компания ООО “ИнтеллиДжей Лабс”, Россия), предоставляющее поддержку встроенного T-SQL в проектах на языке программирования C# в среде разработки Microsoft Visual Studio. Была реализована следующая функциональность: статическая проверка корректности выражений и подсветка ошибок, подсветка синтаксиса и подсветка парных элементов. Исходный код разработанного инструментария и расширения доступен в репозитории по адресу <https://github.com/YaccConstructor/YaccConstructor>.

Также была проведена апробация результатов работы на промышленном проекте по переносу хранимого SQL-кода с MS-SQL Server 2005 на Oracle 11gR2 (ЗАО “Ланит-Терком”). Исходная система состояла из 850 хранимых процедур и содержала более 3000 динамических запросов на 2,7 млн. строк хранимого кода. Более половины динамических запросов были сложными и формировались с использованием от 7 до 212 операторов. При этом среднее количество операторов для формирования запроса — 40. Реализованный механизм позволил корректно автоматически обработать примерно 45% запросов и существенно упростил ручную доработку системы [20].

Основные результаты работы были доложены на ряде научно практических конференциях: SECR-2012, SECR-2013, SECR-2014, ТМПА-2014, Parsing@SLE-2013, Рабочий семинар “Наукоемкое программное обеспечение” при конференции PSI-2014. Доклад на SECR-2014 награждён премией Бертрана Мейера за

лучшую исследовательскую работу в области программной инженерии. Разработка инструментальных средств на основе предложенного алгоритма была поддержана Фондом содействия развитию малых форм предприятий в технической сфере (программа УМНИК). Результаты диссертации изложены в 6 научных работах из которых 3 [17, 21, 22] опубликованы в журналах из списка ВАК.

В [17] С. Григорьеву принадлежит реализация ядра платформы YaccConstructor. В работах [21, 22] и [23] С. Григорьеву принадлежит постановка задачи, формулирование требований к разрабатываемым инструментальным средствам. В [24] автору принадлежит идея, описание и реализация анализа встроенных языков на основе RNGLR алгоритма. В [20] С. Григорьеву принадлежит реализация инструментальных средств, проведение замеров, работа над текстом.

### **Структура работы**

Диссертация состоит из семи глав и построена следующим образом. В первой главе проводится обзор области исследования. Рассматриваются подходы к анализу динамически формируемых строковых выражений и соответствующих инструментов. Кроме того, описывается алгоритм обобщённого восходящего синтаксического анализа RNGLR, положенный в основу алгоритма, предложенного в данной работе. Также описываются проекты YaccConstructor и ReSharper SDK, использующиеся в качестве основы разработанного инструментального пакета. Во второй главе формализуется основная задача исследования и излагается алгоритм, её решающий, — алгоритм синтаксического анализа регулярного множества на основе RNGLR, строящий конечную структуру данных, содержащую деревья вывода для всех цепочек анализируемого множества. В третьей главе приводятся доказательства завершаемости и корректности представленного алгоритма, поясняются шаги работы алгоритма на примерах. В четвёртой главе описывается инструментальный пакет YC.SEL.SDK, разработанный в ходе данной работы на основе алгоритма, описанного во второй главе. YC.SEL.SDK предназначен для разработки инструментов анализа динамически формируемых выражений. Описывается архитектура компонентов и особенности их реализации. Также описывается YC.SEL.SDK.ReSharper — обёртка над YC.SEL.SDK, позволяющая создавать расширения к ReSharper для поддержки встроенных языков. Представлена структура конкретного плагина, реализованного на основе и

с использованием YC.SEL.SDK.ReSharper. В пятой главе описывается методика применения YC.SEL.SDK. В шестой главе приводятся результаты экспериментального исследования YC.SEL.SDK. Седьмая глава содержит результаты сравнения и соотнесения реализованного алгоритма с основными существующими аналогами.

# Глава 1

## Обзор

В данной главе введены основные термины и определения, используемые в работе, а также рассмотрены основные подходы к анализу встроенных языков и инструменты для их обработки. Также рассмотрен алгоритм обобщённого восходящего синтаксического анализа RNGLR, лежащий в основе разработанного алгоритма. Кроме того, описаны компоненты, использовавшиеся при разработке инструментального пакета YC.SEL.SDK.

### 1.1 Языки и грамматики

В данном разделе приведён ряд обозначений, понятий и определений из теории формальных языков, которые используются в работе.

**Определение 1.** *Алфавит  $\Sigma$  — это конечное множество символов.*

**Определение 2.** *Цепочкой символов в алфавите  $\Sigma$  называется любая конечная последовательность символов этого алфавита. Цепочка, которая не содержит ни одного символа, называется пустой цепочкой. Для её обозначения будем использовать греческую букву  $\varepsilon$  (не входит в алфавит  $\Sigma$ , а только помогает обозначить пустую последовательность символов).*

**Определение 3.** *Язык  $L$  над алфавитом  $\Sigma$  — это подмножество множества всех цепочек в этом алфавите.*

**Определение 4.** *Грамматика  $G$  — это четвёрка  $\langle T, N, P, S \rangle$ , где*

- $T$  — алфавит терминальных символов или терминалов;
- $N$  — алфавит нетерминальных символов или нетерминалов,  $T \cap N = \emptyset$ ;
- $P$  — конечное подмножество множества  $(T \cup N)^+ \times (T \cup N)^*$ . Элемент  $(a, b) \in P$  называется правилом вывода и записывается в виде  $a \rightarrow b$ , где  $a$  называется левой частью правила,  $b$  — правой частью, и левая часть любого правила из  $P$  обязана содержать хотя бы один нетерминал;
- $S$  — стартовый символ грамматики,  $S \in N$ .

**Определение 5. Вывод цепочки  $\omega$  в грамматике  $G$ .**

Цепочка  $b \in (T \cup N)^*$  непосредственно выводима из цепочки  $a \in (T \cup N)^+$  в грамматике  $G = \langle T, N, P, S \rangle$  (обозначается  $\rightarrow_G$ ), если  $a = x_1 \cdot y \cdot x_2$ ,  $b = x_1 \cdot z \cdot x_2$ , где  $x_1, x_2, y \in (T \cup N)^*$ ,  $z \in (T \cup N)^+$  и правило вывода  $y \rightarrow z$  содержится в  $P$ . Индекс  $G$  в обозначении  $\rightarrow_G$  обычно опускают, если  $G$  понятна из контекста.

Цепочка  $b \in (T \cup N)^*$  выводима из цепочки  $a \in (T \cup N)^+$  в грамматике  $G$  (обозначается  $a \Rightarrow_G b$ ), если существуют цепочки  $z_0, z_1, \dots, z_n (n \geq 0)$ , такие, что  $a = z_0 \rightarrow z_1 \rightarrow \dots \rightarrow z_n = b$ . Последовательность  $z_0, z_1, \dots, z_n$  называется выводом длины  $n$ .

**Определение 6. Язык, порождаемый грамматикой**

$G = \langle T, N, P, S \rangle$  — это множество  $L(G) = \{\omega \in T^* | S \Rightarrow \omega\}$ .

**Определение 7. Левосторонний вывод цепочки  $\omega$  в грамматике  $G = \langle T, N, P, S \rangle$  — это вывод, в котором на каждом шаге заменяется самое левое из всех вхождений нетерминальных символов, то есть каждый шаг вывода имеет вид  $uA\theta \Rightarrow u\beta\theta$ , где  $(A \rightarrow \beta) \in P$ ,  $u \in \Sigma^*$  и  $\theta \in (N \cup \Sigma)^*$ .**

**Определение 8. Правосторонний вывод цепочки  $\omega$  в грамматике  $G = \langle T, N, P, S \rangle$  определяется аналогично левостороннему, то есть на каждом шаге заменяется самое правое вхождение нетерминала.**

**Определение 9. Грамматика  $G$  называется неоднозначной (ambiguous), если существует слово  $\omega \in L(G)$ , которое имеет два или более левосторонних вывода. В противном случае контекстно-свободная грамматика называется однозначной (unambiguous).**

**Определение 10.** Язык  $L_1$  называется *существенно неоднозначным*, если не существует такой грамматики  $G$ , что  $G$  однозначна и  $L_1 = L(G)$ .

**Определение 11.** *Деревом вывода* цепочки  $\omega \in T^*$  в грамматике  $G = \langle T, N, P, S \rangle$  называется упорядоченное дерево со следующими свойствами.

- Корень помечен  $S$ .
- Если его внутренний узел помечен  $A \in N$  и  $X_1, \dots, X_k \in T \cup N$  — перечисленные слева направо пометки всех сыновей этого узла, то правило  $A \rightarrow X_1 \dots X_k \in P$ .
- Если его внутренний узел помечен  $A \in N$  и  $\varepsilon$  — пометка единственного сына этого внутреннего узла, то правило  $A \rightarrow \varepsilon \in P$ .
- $\omega = a_1 \dots a_m$ , где  $a_1, \dots, a_m \in T \cup \{\varepsilon\}$  перечисленные слева направо пометки всех листьев этого дерева.

**Определение 12.** *Динамически формируемое строковое выражение* — это строковое выражение, значение которого будет известно только в момент выполнения программы.

**Определение 13.** Язык, на котором написана программа, будем называть *внешним языком*.

В случае, когда известно, что значение строкового выражения должно являться кодом на некотором языке, говорят о **встроенных языках** (также называемых встроенными строковыми языками или string-embedded languages [9]). Например, для листинга 1.1 внешним языком является C#. Про переменную `seхes`, основываясь на строках 3–7, можно сделать предположение, что она должна содержать выражение на SQL. Таким образом, в данном примере присутствует SQL, встроенный в C#, и динамически формируемый SQL-запрос. Отметим, что выражение на строке 9 является статическим, а строковое выражение на строке 10 является динамически формируемым, но не является кодом на некотором языке программирования. Обработка таких выражений в общем случае называется анализом строк (string analysis [25]).



---

```

1 public void Example(string tbl, bool cond)
2 {
3     string sExec =
4         "SELECT sOrderDescription, cderitInfo, @sMagicKey FROM ts."
5         + tbl;
6         + (cond ? "WHERE fld = 1 " : "WHERE fld = 2 ");
7
8     db.Execute(sExec);
9
10    Console.WriteLine("Success. Table: " + tbl);
11 }

```

---

Листинг 1.1: Пример кода метода на языке программирования C#, содержащего динамически формируемые строковые выражения

Одним из распространённых способов классификации грамматик является иерархия грамматик по Хомскому [26]. Рассмотрим её более подробно, так как различия между классами играют важную роль в решении задач данной работы.

- **Грамматика типа 0.** Любая грамматика является грамматикой типа 0. На вид правил грамматик этого типа не накладывается никаких дополнительных ограничений. Класс языков типа 0 совпадает с классом рекурсивно перечислимых языков.
- **Грамматикой типа 1** будем называть неукорачивающую грамматику. Грамматика  $G = \langle T, N, P, S \rangle$  называется неукорачивающей, если правая часть каждого правила из  $P$  не короче левой части: для любого правила  $\alpha \rightarrow \beta \in P$  выполняется неравенство  $|\alpha| \leq |\beta|$ . В виде исключения в неукорачивающей грамматике допускается наличие правила  $S \rightarrow \varepsilon$ , при условии, что  $S$  не встречается в правых частях правил. Тип 1 также можно определить с помощью контекстно-зависимых грамматик. Грамматика  $G = \langle T, N, P, S \rangle$  называется контекстно-зависимой (КЗ), если каждое правило из  $P$  имеет вид  $\alpha \rightarrow \beta$ , где  $\alpha = \omega_1 A \omega_2, \beta = \omega_1 \gamma \omega_2, A \in N, \gamma \in (T \cup N)^+, \omega_1, \omega_2 \in (T \cup N)^*$ . В виде исключения в КЗ-грамматике допускается наличие правила с пустой правой частью  $S \rightarrow \varepsilon$ , при условии, что  $S$  не встречается в правых частях правил. Цепочку  $\omega_1$  называют левым контекстом, цепочку  $\omega_2$  называют правым контекстом. Язык, порождаемый контекстно-зависимой грамматикой, называется контекстно-зависимым языком.

- **Грамматикой типа 2** будем называть контекстно-свободную грамматику. Грамматика  $G = \langle T, N, P, S \rangle$  называется контекстно-свободной (КС), если каждое правило из  $P$  имеет вид  $A \rightarrow \beta$ ,  $A \in N, \beta \in (T \cup N)^*$ . Заметим, что в КС-грамматиках допускаются правила с пустыми правыми частями. Язык, порождаемый контекстно-свободной грамматикой, называется контекстно-свободным языком.
- **Грамматикой типа 3** является регулярная грамматика, определение которой приведено ниже.

Грамматика  $G = \langle T, N, P, S \rangle$  называется праволинейной, если каждое правило из  $P$  имеет вид  $A \rightarrow wB$  либо  $A \rightarrow w$ , где  $A, B \in N, w \in T^*$ . Грамматика  $G = \langle T, N, P, S \rangle$  называется леволинейной, если каждое правило из  $P$  имеет вид  $A \rightarrow Bw$  либо  $A \rightarrow w$ , где  $A, B \in N, w \in T^*$ .

При фиксированном языке  $L$  два следующих утверждения эквивалентны:

- существует праволинейная грамматика  $G_1$ , такая что  $L = L(G_1)$ ;
- существует леволинейная грамматика  $G_2$ , такая что  $L = L(G_2)$ .

Из данного утверждения следует, что праволинейные и леволинейные грамматики определяют один и тот же класс языков, который будем называть классом регулярных языков. Право- и леволинейные грамматики будем называть **регулярными грамматиками**.

Существуют различные способы описания языков. Если язык конечен, то его можно описать простым перечислением входящих в него цепочек. Однако формальный язык может быть бесконечным, и в таком случае требуются механизмы, позволяющие конечным образом представлять бесконечное множество цепочек. Можно выделить два основных подхода для такого представления:

1. механизм распознавания, когда описывается процедура, проверяющая принадлежность цепочки описываемому языку;
2. механизм порождения (генерации), когда задаётся механизм, способный построить все цепочки описываемого языка.

Основной способ реализации механизма порождения — использование грамматик, которые как раз и описывают правила построения цепочек некоторого языка. Вместе с этим, можно явным образом описать процедуру-генератор цепочек языка, что также будет являться описанием языка. Например, программа на любом языке программирования, генерирующая некоторый текст, является описанием языка. В данной работе будут рассматриваться такие программы.

## 1.2 Конечные автоматы и преобразователи

Одним из способов задания регулярных языков является описание конечного автомата, который может быть использован и как генератор, и как распознаватель.

**Определение 14.** *Конечный автомат* (*Finite State Automata*, [27]) — это пятёрка  $M = \langle Q, \Sigma, \Delta, I, F \rangle$ , где:

- $\Sigma$  — конечный алфавит;
- $Q$  — конечное множество состояний;
- $I$  — множество начальных состояний,  $I \subseteq Q$ ;
- $F$  — множество заключительных или допускающих состояний,  $F \subseteq Q$ ;
- $\Delta \subseteq Q \times \Sigma^* \times Q$ ; если  $\langle p, x, q \rangle \in \Delta$ , то  $\langle p, x, q \rangle$  называется переходом (transition) из  $p$  в  $q$ , а слово  $x$  — меткой (label) этого перехода; в общем случае автомат является недетерминированным (НКА), то есть позволяющим несколько переходов с одинаковым начальным состоянием и одинаковой меткой.

**Определение 15.** Конечный автомат  $\langle Q, \Sigma, \Delta, I, F \rangle$  называется **детерминированным** (deterministic) (ДКА), если

- множество  $I$  содержит ровно один элемент;
- для каждого перехода  $\langle p, x, q \rangle \in \Delta$  выполняется равенство  $|x| = 1$ ;

- для любого символа  $a \in \Sigma$  и для любого состояния  $p \in Q$  существует не более одного состояния  $q \in Q$  со свойством  $\langle p, a, q \rangle \in \Delta$ .

**Определение 16.** Конечный автомат с  $\varepsilon$ -переходами — это конечный автомат, в котором есть возможность совершать переходы по  $\varepsilon$ .

**Определение 17.**  $\varepsilon$ -НКА  $A$  — это НКА, где  $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$ .

**Определение 18.** Язык, распознаваемый конечным автоматом  $M$ , — это язык  $L(M)$ , состоящий из всех допускаемых данным автоматом цепочек. Также говорят, что автомат  $M$  описывает или задаёт некоторый язык  $L(M)$ .

Класс регулярных языков эквивалентен классу конечных автоматов в том смысле, что для любого регулярного языка  $L_1$  можно построить детерминированный конечный автомат  $M$ , такой  $L(M) = L_1$ . При этом множество языков, допускаемых автоматами с  $\varepsilon$ -переходами, совпадает с множеством языков, допускаемых детерминированными конечными автоматами. Также будет удобно отождествлять регулярный язык и регулярное множество.

Конечные автоматы можно изображать в виде диаграмм переходов (transition diagram). На диаграмме каждому состоянию соответствует вершина графа, а переходу — дуга. Дуга из  $p$  в  $q$ , помеченная словом  $x$ , означает, что  $\langle p, x, q \rangle$  является переходом данного конечного автомата. Вершины, соответствующие начальным и конечным состояниям, отмечаются отдельно: конечные состояния изображаются как двойной круг, начальные отмечаются отдельной входной дугой, не имеющей стартовой вершины. Также в данной работе будет использоваться следующая цветовая нотация: конечные вершины обозначены красным цветом, начальные — зелёным. Таким образом, автомат представим в виде графа и в данной работе к конечным автоматам будет применяться терминология из теории графов.

**Определение 19.** Конечный преобразователь (Finite State Transducer, [28]) — это конечный автомат, который может возвращать конечное число символов для каждого входного символа. Конечный преобразователь может быть задан следующей шестёркой элементов:  $\langle Q, \Sigma, \Delta, q_0, F, E \rangle$ , где:

- $Q$  — множество состояний;

- $\Sigma$  — входной алфавит;
- $\Delta$  — выходной алфавит;
- $q_0 \in Q$  — начальное состояние;
- $F \subseteq Q$  — набор конечных состояний;
- $E \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Delta \cup \{\varepsilon\}) \times Q$  — набор переходов.

Конечные преобразователи находят широкое применение в области обработки естественного языка (Natural Language Processing, [29]). Кроме этого, они используются и при проведении лексического анализа, который является переводом входной цепочки из одного языка в другой: из языка над алфавитом символов в язык над алфавитом терминалов. Большинство генераторов лексических анализаторов строят по описанию лексики языка соответствующий конечный преобразователь.

Важной операцией над конечными преобразователями является операция композиции. **Композиция** конечных преобразователей — это два последовательно взаимодействующих конечных преобразователя, работающих следующим образом: выход первого конечного преобразователя подаётся на вход второму, что позволяет описывать цепочки трансформаций. Ниже дано формальное определение операции композиции над конечными преобразователями, допускающие наличие  $\varepsilon$ -переходов.

**Определение 20.** *Композицией двух конечных преобразователей*

$T_1 = \langle Q_1, \Sigma_1, \Delta_1, q_{0_1}, F_1, E_1 \rangle$  и  $T_2 = \langle Q_2, \Sigma_2, \Delta_2, q_{0_2}, F_2, E_2 \rangle$  является конечный преобразователь  $T = \langle Q_1 \times Q_2, \Sigma_1, \Delta_2, \langle q_{0_1}, q_{0_2} \rangle, F_1 \times F_2, E \cup E_\varepsilon \cup E_{i,\varepsilon} \cup E_{o,\varepsilon} \rangle$ , где:

- $E = \{ \langle \langle p, q \rangle, a, b, \langle p', q' \rangle \rangle \mid \exists c \in \Delta_1 \cap \Sigma_2 : \langle p, a, c, p' \rangle \in E_1 \wedge \langle q, c, b, q' \rangle \in E_2 \}$ ;
- $E_\varepsilon = \{ \langle \langle p, q \rangle, a, b, \langle p', q' \rangle \rangle \mid \langle p, a, \varepsilon, p' \rangle \in E_1 \wedge \langle q, \varepsilon, b, q' \rangle \in E_2 \}$ ;
- $E_{i,\varepsilon} = \{ \langle \langle p, q \rangle, \varepsilon, a, \langle p, q' \rangle \rangle \mid \langle q, \varepsilon, a, q' \rangle \in E_2 \wedge p \in Q_1 \}$ ;
- $E_{o,\varepsilon} = \{ \langle \langle p, q \rangle, a, \varepsilon, \langle p', q \rangle \rangle \mid \langle p, a, \varepsilon, p' \rangle \in E_1 \wedge q \in Q_2 \}$ .

В рамках данной работы конечные преобразователи и их композиция будут использоваться для лексического анализа динамически формируемых строковых выражений.

### 1.3 О применимости статического анализа строковых выражений

Статический анализ динамически формируемых выражений полезен на различных этапах работы с кодом при решении различных задач [30]. Рассмотрим пример и поясним, какие задачи необходимо решать и каким образом анализ строковых выражений помогает решать данные задачи. Мы рассмотрим пример встроенного SQL, однако все рассмотренные задачи актуальны и для других языков.

Одной из широко распространённых задач, является оценка качества кода и его сложности с использованием различных формальных метрик [31]. При вычислении таких метрик важно учитывать, что использование динамически формируемых выражений, сложность их конструирования, количество и содержание возможных значений и многие другие характеристики сказываются на качестве и сложности содержащего их кода. По этой причине необходимо иметь возможность оценивать сложность динамически формируемых выражений с различных точек зрения [32, 33]. С одной стороны, необходимо оценивать сложность формирования выражения. Так в примере, представленном в листинге 1.2, для формирования запроса используется цикл (строки 9–13), что может приводить к потенциально бесконечному множеству различных значений выражения и усложнять процесс сопровождения. С другой стороны, важна сложность возможных значений выражения. В листинге 1.2, в динамически формируемом запросе используется конструкция соединения таблиц (JOIN, строка 17). Большое количество соединений и сложность условий часто становится причиной проблем с производительностью и может служить признаком неудачного дизайна схемы данных [33].

Другой набор задач, связанный с сопровождением и модификацией систем, разработанных с активным использованием динамически формируемых выраже-

---

```

1 public void NewReport(int prodId = 0, int status = 0, int nType = 0)
2 {
3     int nProdIdL = prodId;
4
5     string sMagicKey = "[" + prodId.ToString() + "]";
6
7     string tbl = status == 0 ? "InOrders " : "OutOrders ";
8
9     while (nProdIdL > 0)
10    {
11        sMagicKey = "[" + sMagicKey + "]";
12        nProdIdL = nProdIdL - 1;
13    }
14
15    string sExec =
16        "SELECT sOrderDescription, cderitInfo, " + sMagicKey
17        + " FROM ts." + tbl +
18        "as t1 JOIN tCreData cd (NOLOCK) ON cd.ncredataid "
19        + " = t1.ncredataid";
20
21    string sWhere = nType == 0
22        ? "WHERE nOrderType = 0 AND nStatus > 2 "
23        : "WHERE nStatus > 0 ";
24
25    sExec = "INSERT INTO reports (description, creditInfo, id)"
26        + " VALUES " + sExec + sWhere;
27
28    db.Execute(sExec);
29 }

```

---

Листинг 1.2: Пример кода метода на языке программирования C#, формирующего и выполняющего динамический SQL-запрос

ний — это извлечение знаний о системе [34] и автоматизированный реинжиниринг программного обеспечения [35]. Например, при активном использовании встроенного SQL может возникать задача анализа или восстановления схемы данных [30]. Проанализировав структурное представление динамически формируемого SQL-кода в примере из листинга 1.2, можно сделать вывод о том, что данный код обращается к таблицам InOrders, OutOrders и tCreData на чтение, а к таблице reports на запись. Без анализа строковых выражений эта информация не может быть получена, а она может быть полезна, например, при модификации схемы данных.

Так как встроенные языки активно используются на практике, то важна их поддержка в интегрированных средах разработки, что может быть полезно не

только при непосредственной разработке, но и при автоматизированном и ручном изучении кода, совмещённом с решением перечисленных выше задач. Дополнительная поддержка встроенных языков в средах разработки может включать подсветку синтаксиса и парных элементов, навигацию по коду с учётом динамически формируемого выражения, диагностику и подсветку ошибок, что упрощает работу с кодом. Например, в листинге 1.2 пропущен пробел между блоком WHERE (переменная `sWhere`, строки 20–22) и началом конструкции SELECT (переменная `sExec`, строки 15–18). То есть при выполнении данного метода будет формироваться некорректный запрос, но об этом станет известно только в момент выполнения метода. Однако ошибки такого рода можно обнаруживать без запуска программы и сообщать об этом разработчику.

## 1.4 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 [36]), 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  $G$ . Основным недостатком 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 [37]. Инструмент, реализующий данный подход, — Java String Analyzer [13], 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 [37]. По этой причине при использовании такого приближения будет получено неточное решение, так как потребуются применение эвристик. Данный подход реализован в инструменте RHP SA [14], предназначенном 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 [6–8]. Так как задача проверки включения для двух контекстно-свободных языков неразрешима, то представлено приближённое решение. В работе [8] обоснована возможность семантического анализа на основе классического для LR-анализа механизма: атрибутивных грамматик [38] 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 [9, 10] 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 [10]. Это позволяет более гибко переиспользовать существующие реализации тех или иных шагов и упрощает создание нового инструмента на базе имеющихся. Использование атрибутивных грамматик — классического для LR-анализа способа задания семантики — и построение леса разбора в рамках данного подхода также не обсуждается.

## 1.5 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 [39], SwisSQL [40], SQL Ways [15]. Эти инструменты применяются для трансляции хранимого SQL-кода, однако только SQL Ways обладает воз-

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 1.4.

**Java String Analyzer (JSA, [13,41])** — 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, [14,42])** — инструмент для статического анализа строк в программах на PHP. Расширяет подход инструмента JSA [13]. 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 операции и учитывать их при построении контекстно-свободной грамматики, используется конечный преобразователь. Дальнейший анализ строковых выражений полностью заимствован из инструмента JSA.

**Alvor [9–11]** — плагин к среде разработки Eclipse<sup>1</sup>, предназначенный для статической проверки корректности SQL-выражений, встроенных в Java<sup>2</sup>. Для

<sup>1</sup>Сайт среды разработки Eclipse: <http://www.eclipse.org/ide/> (Посещён 23.06.2015.)

<sup>2</sup>Во время написания данного текста велась работа над поддержкой встроенных языков в PHP.

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 [43]. 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** [12] — плагин к средам разработки PhpStorm [44] и IntelliJ IDEA<sup>3</sup>, 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** [44] — 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** [45] — 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

---

<sup>3</sup>IntelliJ IDEA — среда разработки для JVM-языков. Сайт: <https://www.jetbrains.com/idea/> (Посещён 23.06.2015.)

к объявлению (jump to declaration), построения графа вызовов (call graph) для встроеного JavaScript.

**Абстрактный синтаксический анализ.** Kyung-Goo Doh, Hyunha Kim, David A. Schmidt в серии работ [6–8] описали алгоритм статического анализа динамически формируемых строковых выражений на примере статической проверки корректности динамически генерируемого HTML в PHP-программах. Хотя для данного примера отсутствует 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 [46]). В работе [8] данный подход был расширен вычислением семантики с помощью атрибутивных 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.

## 1.6 Алгоритмы и структуры данных для обобщённого синтаксического анализа

В этом разделе будет описан алгоритм обобщённого восходящего синтаксического анализа RNGLR [18], используемый в данной работе. Кроме этого, будут

описаны основные структуры данных, специфичные для обобщённого синтаксического анализа.

Анализ динамически формируемых выражений подразумевает работу со множеством значений. При синтаксическом анализе множества появится множество стеков и множество деревьев разбора. Среди существующих алгоритмов есть класс алгоритмов обобщённого синтаксического анализа [47], в рамках которого разработаны эффективные методы работы с множеством стеков и деревьев разбора. По этой причине рассмотрим некоторые алгоритмы обобщённого синтаксического анализа и применяемые в них структуры данных более подробно.

### 1.6.1 Алгоритм обобщённого LR-анализа

Один из подходов к синтаксическому анализу — это табличный LR-анализ, при котором строится правосторонний вывод и дерево вывода строится снизу вверх. Механизм анализа основан на применении автомата с магазинной памятью, управляющие таблицы для которого строятся на основе грамматики обрабатываемого языка [48]. Идея состоит в том, что символы входной цепочки переносятся в стек до тех пор, пока на вершине стека не накопится цепочка, совпадающая с правой частью какого-либо из правил (операция *перенос* или *shift*). Далее все символы этой цепочки извлекаются из стека, и на их место помещается нетерминал, соответствующий этому правилу (операция *свёртка* или *reduce*). Входная цепочка допускается автоматом, если после переноса в автомат последнего символа входной цепочки и выполнении необходимого числа свёрток в стеке окажется только стартовый нетерминал грамматики.

Как уже было сказано ранее, при выполнении табличного синтаксического анализа для данной грамматики строятся таблица действий и таблица переходов. Таблица переходов — это вспомогательная таблица, использующаяся при одном из действий и в ячейке может содержать либо состояние анализатора, либо символ ошибки.

Таблица действий определяет дальнейшее действие в текущем состоянии и с текущим символом на входе. Каждая ячейка данной таблицы может содержать одно из следующих значений:



- accept (“успех”) — разбор входной цепочки завершился успешно;
- shift (“перенос”) — на вершину стека переносится состояние, которое соответствует входному символу, читается следующий символ;
- reduce (“свёртка”) — в стеке набрались состояния, которые можно заменить одним, исходя из правил грамматики; значение нового состояния берётся из таблицы переходов;
- error (“ошибка”) — анализатор обнаружил ошибку во входной цепочке.

При работе с неоднозначными грамматиками могут возникнуть ситуации, когда в одну ячейку таблицы необходимо записать несколько действий. Это означает, что в процессе обработки некоторой цепочки при цепочки анализатор не может однозначно решить, какое действие совершить в текущем состоянии. Таким образом возникают конфликты shift/reduce, когда можно либо прочитать очередной символ, либо произвести свёртку, и reduce/reduce, когда можно произвести свёртку по нескольким правилам грамматики.

Для решения данной проблемы Масару Томитой был предложен алгоритм Generalized LR (GLR) [49], изначально предназначенный для анализа естественных языков. GLR-алгоритм был предназначен для работы с неоднозначными контекстно-свободными грамматиками, а значит умел обрабатывать shift/reduce и reduce/reduce конфликты. Используемые в данном алгоритме управляющие таблицы схожи с управляющими таблицами LR-алгоритма, но отличаются тем, что ячейки 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 специальные структуры данных, основанные на графах.

## 1.6.2 Структурированный в виде графа стек

Структурированный в виде графа стек (Graph Structured Stack или GSS) [49] является ориентированным графом, чьи вершины соответствуют элементам отдельных стеков, а ребра связывают последовательные элементы. Вершина может иметь несколько входящих рёбер, что соответствует слиянию нескольких стеков

или несколько исходящих, что соответствует конфликту — ситуации, в которой дальнейший разбор может осуществляться несколькими способами. Объединение стеков происходит в процессе анализа, когда на вершинах различных веток, соответствующих одинаковой позиции во входном потоке, оказывается одинаковое состояние анализатора. За счёт такой организации GSS обеспечивается переиспользование общих участков отдельных стеков. Пример организации GSS приведён на рисунке 1.1: “наивное” решение копировать стеки при возникновении конфликтов приводит к дублированию информации, чего можно избежать при использовании GSS.



Рисунок 1.1: Пример GSS

### 1.6.3 Сжатое представление леса разбора

Для переиспользования общих поддеревьев вывода Ян Рекерс (Jan Rekers, University of Amsterdam) предложил сжатое представление леса разбора (Shared Packed Parse Forest, SPPF) [19], которое позволяет компактно представлять множество деревьев вывода. Важным свойством SPPF является то, что из него можно извлечь только те и только те деревья, которые могли быть получены в результате построения вывода конкретного входа в заданной грамматике. Для обес-



печения этого свойства в SPPF, кроме терминальных и нетерминальных узлов, добавляются дополнительные узлы различных типов. Конкретный набор типов дополнительных узлов может отличаться в зависимости от алгоритма анализа.

---

```

1 s ::= m
2 s ::= p
3 p ::= A n
4 m ::= A l
5 l ::= n
6 n ::= B C

```

---

Листинг 1.3: Грамматика  $G_1$

Пример SPPF для грамматики  $G_1$  (листинг 1.3) и входа ABC приведён на рисунке 1.2. Представлены два различных дерева вывода (1.2a и 1.2b) и результат их объединения в SPPF 1.2c. Узлы с именами вида “n <name>” — это нетерминальные узлы, “e <name>” — терминальные, “prod <num>” — дополнительные узлы, показывающие, согласно какой продукции из грамматики производился вывод нетерминала, являющегося предком данного узла.

## 1.6.4 Алгоритм RNGLR

RNGLR-алгоритм (Right-Nullled Generalized LR) [18] является модификацией предложенного Масару Томитой алгоритма, который не был способен обрабатывать все контекстно-свободные грамматики. Чтобы устранить данный недостаток, Элизабет Скотт (Elizabeth Scott) и Адриан Джонстон (Adrian Johnstone) из университета Royal Holloway (Великобритания) предложили RNGLR-алгоритм, который расширяет GLR-алгоритм специальным способом обработки обнуляемых справа правил (right-nullable rules, имеющих вид  $A \rightarrow \alpha\beta$ , где  $\beta$  выводит пустую строку  $\epsilon$ ), позволяя обрабатывать произвольные контекстно-свободные грамматики. Алгоритм синтаксического анализа динамически формируемых выражений, представленный в данной работе, основан на RNGLR-алгоритме. Его подробное описание в виде псевдокода приведено ниже.

Для эффективного представления множества стеков во время синтаксического анализа в алгоритме RNGLR, как и в классическом GLR, используется структурированный в виде графа стек (GSS). Вершина GSS — это пара  $(s, l)$ , где

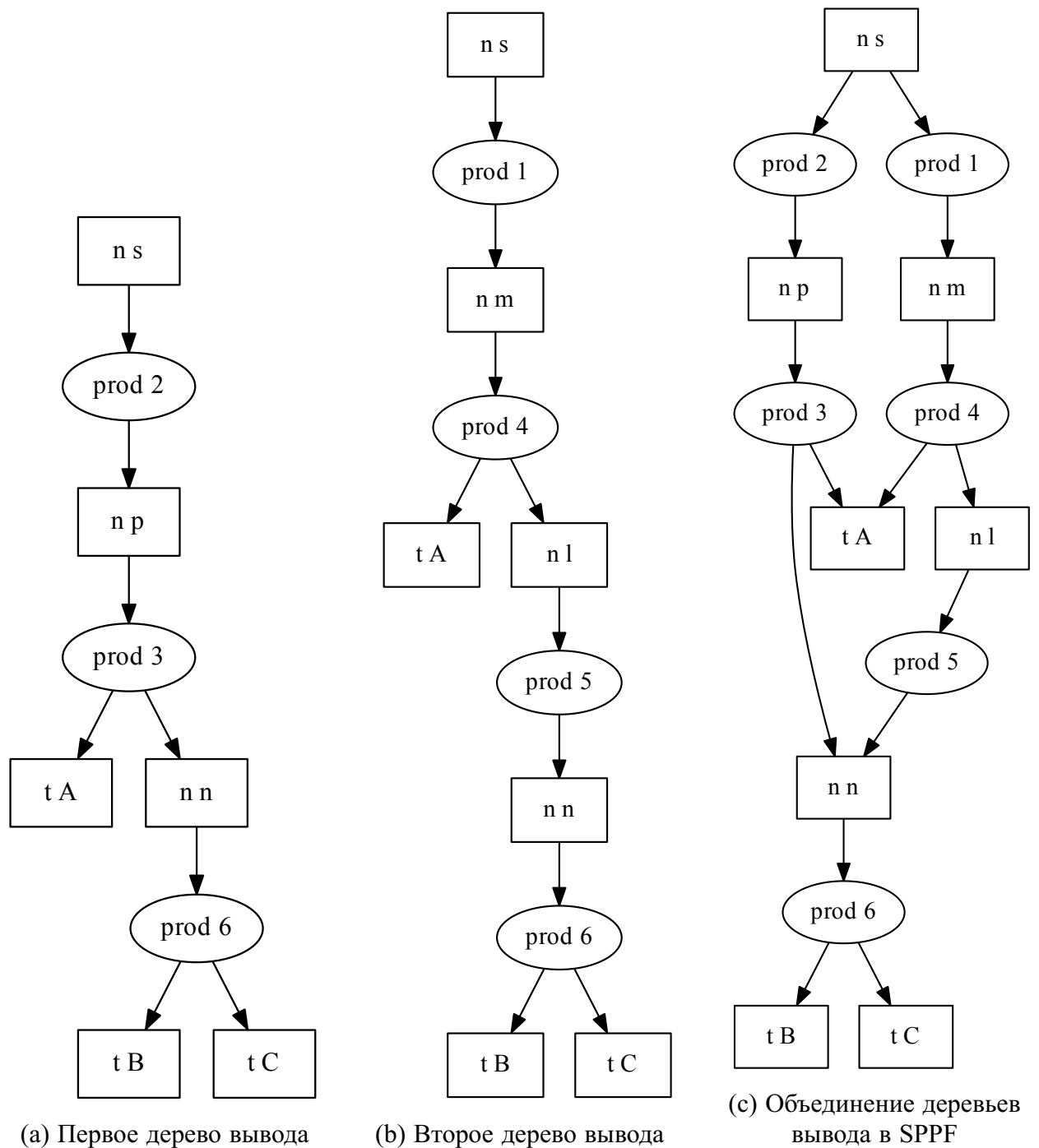


Рисунок 1.2: Пример SPPF для грамматики  $G_1$  и входа **ABC**

$s$  — состояние синтаксического анализатора, 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 в GSS необходимо добавить ребро  $(v_t, v_h)$ . По построению,

---

```

1: function PARSE(grammar, input)
2:    $\mathcal{R} \leftarrow \emptyset$        $\triangleright$  Очередь троек: вершина GSS, нетерминал, длина свёртки
3:    $\mathcal{Q} \leftarrow \emptyset$        $\triangleright$  Коллекция пар: вершина GSS, состояние синтаксического
      анализатора
4:   if input =  $\epsilon$  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  $\mathcal{R}$  is not empty do
17:       (v, N, l)  $\leftarrow \mathcal{R}.Dequeue()$ 
18:       find the set  $\mathcal{X}$  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  $\mathcal{X}$  do
21:          $state_t \leftarrow$  calculate new state by  $state_h$  and nonterminal N
22:         ADDEDGE(i,  $v_h$ , v.level,  $state_t$ , (l = 0))
23:   function PUSH(i)
24:      $\mathcal{Q}' \leftarrow$  copy  $\mathcal{Q}$ 
25:     while  $\mathcal{Q}'$  is not empty do
26:       (v, state)  $\leftarrow \mathcal{Q}.Dequeue()$ 
27:       ADDEDGE(i, v, v.level + 1, state, false)

```

---

Листинг 1.4: Критерии сравнения инструментов анализа динамически формируемых строковых выражений

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 синтаксического анализатора  $s'$  по  $s$  и следующему символу входного потока. Пара  $(v, s')$ , называемая push, добавляется в глобальную коллекцию  $\mathcal{Q}$ . Также при добавлении

---

```

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  $\mathcal{Q}$ 
5:     calculate the set of zero-reductions by  $v$  and the  $input[i + 1]$  and
6:     add them to  $\mathcal{R}$ 
7:   return  $v$ 
8: function ADDEDGE( $i, v_h, level_t, state_t, isZeroReduction$ )
9:    $v_t \leftarrow$  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  $\mathcal{R}$ 

```

---

Листинг 1.5: Построение GSS

новой вершины в GSS вычисляется множество  $\epsilon$ -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- $\epsilon$ ) ребро. Подробное описание работы со структурированным в виде графа стеком GSS представлено в листинге 1.5.

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 SPPF, являющееся ориентированным графом и в данном случае обладающее следующей структурой.

1. *Корень* соответствует стартовому нетерминалу грамматики.
2. *Терминальные* вершины, не имеющие исходящих дуг, соответствуют либо терминалам грамматики, либо деревьям вывода пустой строки  $\epsilon$ .
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 достижимы

мые из них, упорядочены и могут являться либо терминальными, либо нетерминальными вершинами. Количество таких вершин лежит в промежутке  $[l - k \dots l]$ , где  $l$  — это длина правой части продукции, а  $k$  — количество финальных символов, выводющих  $\varepsilon$ . При этом такие символы игнорируются для уменьшения потребления памяти.

SPPF создаётся параллельно с построением GSS. С каждым ребром GSS ассоциирован либо терминальный, либо нетерминальный узел. Когда добавление ребра в GSS происходит во время операции push, новая терминальная вершина создаётся и ассоциируется с ребром. Нетерминальные вершины ассоциируются с рёбрами, добавленными во время операции reduce. Если ребро уже есть 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, в результате чего остаются только корректные деревья разбора для входной строки.

Листинг ?? представляет более детальное описание алгоритма.

## 1.7 Используемые инструменты

В этом разделе описываются основные инструменты, использованные в работе: YaccConstructor [17, 50], выбранный в качестве основы для реализации алгоритма синтаксического анализа динамически формируемых выражений, и ReSharper SDK [51], использованный для разработки сшиений для microsoft Visual Studio IDE.

### 1.7.1 YaccConstructor

Одной из задач данной работы является создание инструментария, упрощающего разработку целевых инструментов статического анализа строковых выражений, которые включают такие этапы, как лексический и синтаксический

анализ. При создании лексических и синтаксических анализаторов широко распространённой практикой является использование генераторов, которые по описанию языка строят соответствующий анализатор. Данный подход должен быть реализован и для создания инструментов анализа встроенных языков. Работа с описаниями языков программирования — грамматикой и лексической спецификацией — в рамках решаемой задачи аналогична работе с ними в стандартных генераторах. По этой причине необходимо было выбрать готовую платформу для работы с грамматиками и создания синтаксических и лексических анализаторов.

В качестве такой платформы был выбран исследовательский проект лаборатории языковых инструментов JetBrains YaccConstructor (YC) [17, 50], который является модульной платформой с открытым исходным кодом для исследований в области лексического и синтаксического анализа и разработки соответствующих инструментов. YC реализован на платформе Microsoft .NET<sup>4</sup>, основной язык разработки — F#<sup>5</sup> [52].

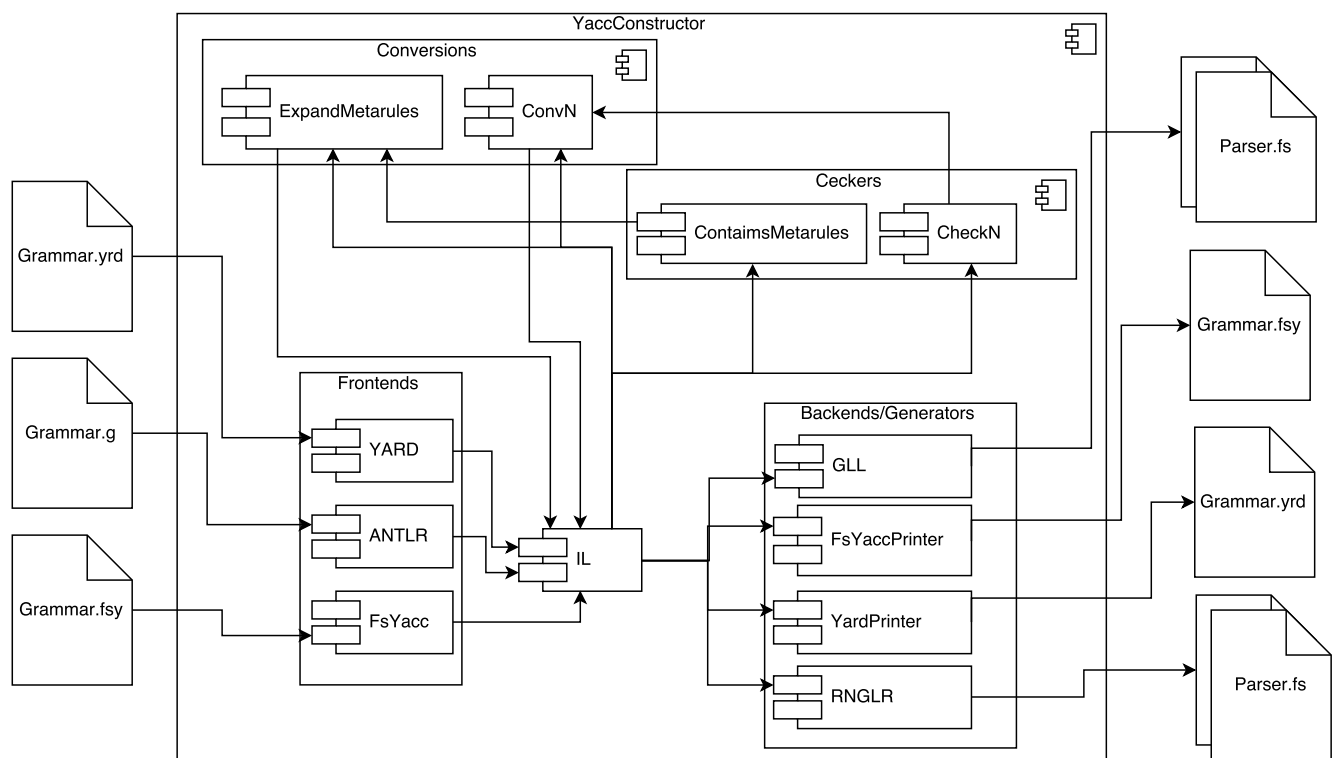


Рисунок 1.3: Архитектура платформы YaccConstructor

<sup>4</sup>Microsoft .NET — платформа для разработки программных продуктов компании Microsoft. Общие сведения о платформе: [https://msdn.microsoft.com/ru-ru/library/zw4w595w\(v=vs.110\).aspx](https://msdn.microsoft.com/ru-ru/library/zw4w595w(v=vs.110).aspx) (Посещено 23.06.2015.)

<sup>5</sup>F# — функциональный язык программирования для платформы .NET. Информация о языке: <http://fsharp.org> (Посещено 23.06.2015.)

Архитектура УС, представленная на рисунке 1.3, позволяет собирать требуемый инструмент из существующих модулей: можно выбрать фронтенд, соответствующий используемому языку спецификации грамматики, задать необходимые преобразования грамматики, указать необходимый генератор. Генераторы (backend) представляют различные инструменты, которые по внутреннему представлению грамматики получают результат, полезный для конечного пользователя. Например, это могут быть генераторы синтаксических анализаторов, основанные на различных алгоритмах или принтеры, генерирующие текст грамматики в определённом формате или на определённом языке. УС является расширяемой платформой: модуль любого типа может быть реализован, в том числе с переиспользованием уже существующих, и подключён к платформе.

В рамках УС разработан выразительный язык спецификации грамматик YARD, поддерживающий атрибутные грамматики, грамматики в EBNF <sup>6</sup> и многое другое. В листинге 1.6 представлена грамматика языка арифметических выражений Calc на языке YARD.

---

```

1  [<Start>]
2  expr: factor [MULT expr]
3  powExpr: NUM | LBR expr RBR
4  factor: powExpr [POW factor]

```

---

Листинг 1.6: Пример грамматики языка арифметических выражений на языке YARD

Выразительный синтаксис языка описания грамматики удобен для разработчика, однако генераторы, как правило, не поддерживают обработку грамматик в таком виде. Часто требуется, чтобы правая часть правила грамматики не содержала регулярных выражений, а состояла бы из цепочки из терминалов и нетерминалов. Это необходимо для работы алгоритма построения таблиц. Для решения этой задачи в УС реализован ряд преобразований грамматик. В результате их применения к грамматике, представленной в листинге 1.6, можно получить грамматику, представленную в листинге 1.7.

Также в рамках платформы УС в качестве одного из модулей ранее был реализован генератор синтаксических анализаторов на основе RNLRL-алгоритма.

---

<sup>6</sup>EBNF — расфигурная форма Бэкуса-Наура. Грамматики в этой форме позволяют использовать регулярные выражения в правых частях правил.!!!!

---

```
1  [<Start>]
2  expr: factor
3  expr: factor MULT expr
4  powExpr: NUM
5  powExpr: LBR expr RBR
6  factor: powExpr
7  factor: powExpr POW factor
8  startRule: expr
```

---

Листинг 1.7: Пример преобразованной грамматики языка арифметических выражений

Это позволяет переиспользовать общие функции и структуры данных при разработке анализатора для встроенных языков. Таким образом, алгоритм анализа встроенных языков и соответствующий генератор может быть реализован в рамках платформы УС в качестве одного из модулей. При этом можно использовать готовый язык описания грамматики и преобразования, а также переиспользовать необходимые элементы генератора анализаторов на основе RNGLR-алгоритма. Таким образом, УС был выбран в качестве основы для реализации благодаря удобной архитектуре и большому количеству готовых решений.

## 1.7.2 ReSharper SDK

Для демонстрации разработанного в рамках данной работы инструмента представляется целесообразным создать на его основе целевое решение. Для этого необходимо окружение, способное обрабатывать внешний язык, поскольку такие операции над внешним языком, как построение дерева разбора, базовый анализ (например, построение графа потока управления или графа потока данных), лежат за пределами рассматриваемой в данной работе задачи, и их выполнение должно осуществляться сторонними инструментами. При этом такие инструменты должны предоставлять необходимую для решения основной задачи функциональность. Кроме того, необходимо иметь удобный способ взаимодействия с пользователем: необходимо получать код для анализа и отображать результаты в виде, удобном для пользователя. Один из самых распространённых способов работы с программным кодом — это работа в интегрированной среде разработки.



Так как реализация работы велась на платформе Microsoft .NET, то соответствующее окружение для обработки внешнего языка и организации взаимодействия с пользователем на данной платформе должно работать на данной платформе. Одной из самых известных сред разработки для платформы .NET, позволяющей создавать расширения на .NET языках, является Microsoft Visual Studio IDE<sup>7</sup>. Она и была выбрана в качестве цели для интеграции инструмента статического анализа строковых выражений. Однако у данной среды разработки достаточно сложный механизм создания расширений, что вызвано его универсальностью. При решении нашей задачи универсальность не требовалась, поэтому можно было бы использовать менее универсальный и более простой механизм.

Такой механизм предоставляется ReSharper SDK [51]. ReSharper [53] — расширение для Microsoft Visual Studio, предоставляющее пользователю широкий набор дополнительных анализов кода. Для разработчиков предоставляется свободно распространяемая SDK, реализующая анализ таких языков, как C#, VB.NET и др. Большая часть функциональности ReSharper выделена в свободно распространяемую SDK, что даёт возможность сторонним разработчикам создавать собственные расширения к ReSharper, переиспользуя его функциональность. В контексте данной работы это позволяет упростить создание плагинов для поддержки различных встроенных языков. Кроме того, ReSharper SDK предоставляет более удобную “обёртку” над многими интерфейсами Microsoft Visual Studio, что упрощает взаимодействие с ней и полезно, например, при предоставлении результатов пользователю.

Следует также отметить, что ReSharper является многоязыковым инструментом, то есть поддерживает большой набор различных языков и спроектирован так, чтобы максимально упростить поддержку новых языков. Это позволяет создавать инструменты, обрабатывающие не только различные встроенные языки, но также и различные внешние.

Выделим следующую функциональность ReSharper SDK, необходимую для реализации поддержки встроенных языков в Microsoft Visual Studio.

- Построение дерева разбора внешнего языка (C#, JavaScript и другие, поддерживаемые ReSharper), узлы которого содержат координаты в исходном

---

<sup>7</sup>Сайт среды разработки Microsoft Visual Studio IDE: <https://www.visualstudio.com/> (Посещён 23.06.2015.)

коде, что позволяет точно связывать текстовое и структурное представление кода.

- Анализ потока данных и анализ потока управления для поддерживаемых языков. На основе существующих анализов можно строить более сложные, необходимые, например, для построения регулярной аппроксимации.
- Вывод сообщений об ошибках и графическое выделение некорректных мест в текстовом редакторе.
- Взаимодействие с редактором кода, позволяющее настраивать подсветку синтаксиса, получать позицию курсора и управлять ею, что нужно для динамической подсветки парных элементов.

Таким образом ReSharper SDK и Microsoft Visual Studio IDE выбраны в качестве основы для разработки целевого инструмента на основе представленных в данной работе результатов.

## 1.8 Выводы

На основе проведённого обзора можно сделать следующие выводы, обосновывающие необходимость проведения исследований в области статического анализа динамически формируемых строковых выражений.

- Проблема анализа строковых выражений актуальна в нескольких областях: поддержка встроенных языков в интегрированных средах разработки, оценка качества кода, содержащего динамически формируемые строковые выражения, реинжиниринг программного обеспечения.
- Большинство реализаций поддерживают конкретный внешний и конкретный встроенный язык и, как правило, решают одну достаточно узкую задачу. При этом, зачастую, плохо расширяемы, как в смысле поддержки других языков, так и в смысле решения новых задач. Полноценные средства разработки инструментов статического анализа динамически формируемых выражений, упрощающие создание решений для новых языков, отсутствуют.

- Для эффективного решения этих задач необходимо структурное представление кода, однако на текущий момент не представлено законченного решения, позволяющего строить деревья вывода для динамически формируемых выражений.

Кроме того, обзор позволяет выявить следующие подходы, технологии и средства.

- В качестве приближения множества значений целесообразно использовать регулярную аппроксимацию [13, 54], так как при работе с ней ряд важных задач является разрешимым в общем случае, что не верно для контекстно-свободной. Более того, работа с регулярной аппроксимацией упрощает решение такой задачи, как лексический анализ встроенных языков.
- Лексический и синтаксический анализы должны быть разделены. Это оправдано как с теоретической, так и с практической точек зрения, так как лексический анализ не приносит потери точности и упрощается переиспользование спецификаций языков и самих анализаторов.
- В качестве основы алгоритма синтаксического анализа динамически формируемых выражений можно выбрать алгоритм обобщённого синтаксического анализа [47], так как в нём реализовано эффективное управление множеством стеков и деревьев разбора, что важно при работе с динамически формируемыми выражениями.

## Глава 2

# Алгоритм синтаксического анализа регулярной аппроксимации

В данной главе формально описана задача синтаксического анализа динамически формируемых выражений, а так же изложен алгоритм её решения. Также сформулированы и доказаны утверждения о корректности данного алгоритма.

### 2.1 Постановка задачи

Анализируемая программа  $\mathcal{P}$  является генератором выражений и таким образом задаёт некоторый язык  $L_1$ . Предположим, что  $\mathcal{P}$  взаимодействует с базой данных с помощью динамически формируемых выражений. В этом случае мы говорим, что  $\mathcal{P}$  использует динамический SQL, подразумевая, что программой генерируются выражения на языке SQL. Однако это не совсем корректно, так как программа может, например, содержать ошибку и некоторые из построенных ей выражений не будут корректными SQL-выражениями. Кроме того, программа  $\mathcal{P}$  может генерировать не все возможные выражения на SQL. Таким образом, в этом случае  $L_1 \neq \text{SQL}$ .

С другой стороны, может быть задан эталонный язык  $L_2$  — язык, которому должны принадлежать все выражения генерируемые программой  $\mathcal{P}$ . Как пра-

вило  $L_2$  описывается с помощью контекстно-свободной грамматики. Например, это может быть грамматика конкретного диалекта SQL, взятая из документации.

В результате мы имеем описание двух языков, и задача проверки корректности выражений, генерируемых  $\mathcal{P}$ , может быть сформулирована как проверка вложенности языка  $L_1$  в язык  $L_2$ . Однако в рамках данной работы нас интересует не просто вложенность языков, а построение деревьев вывода. Как уже говорилось, язык  $L_2$  может быть задан с помощью грамматики. Пусть задана грамматика  $G$ , такая, что  $L_2 = L(G)$ , тогда задача синтаксического анализа динамически формируемых выражений может быть поставлена следующим образом: для всех цепочек  $\omega \in L_1$ , выводимых в грамматике  $G$ , построить соответствующие деревья вывода.

Отметим, что решение изложенных выше вопросов в общем виде возможно не всегда. Трудности могут быть связаны прежде всего с классами рассматриваемых языков. Эталонный язык  $L_2$ , скорее всего, является некоторым языком программирования и принадлежит к классу контекстно-свободных языков (хотя могут быть и исключения). Генерируемый язык  $L_1$ , вообще говоря, не обязан быть даже контекстно-зависимым. Так как программа-генератор реализована на тьюринг-полном языке, то  $L_1$  в общем случае может быть рекурсивно-перечислимым. Из практических соображений хочется думать, что  $L_1$  всё же является подмножеством некоторого языка программирования и может быть достаточно хорошо приближен некоторым контекстно-свободным языком. Однако проверка включения двух контекстно-свободных языков не разрешима в общем случае [37]. При этом разрешима в общем случае проверка включения регулярного языка в однозначный контекстно-свободный [37]. В однозначные контекстно-свободные языки попадают такие практически значимые классы, как LR(k) и LL(k), что позволяет использовать в качестве  $L_2$  практически значимые языки. При этом для  $L_1$  необходимо строить регулярное приближение  $L_R$ .

Чтобы обеспечить надёжность анализа строковых выражений, необходима аппроксимация сверху, то есть должно выполняться включение  $L_1 \subseteq L_R$ . Это необходимо для того, чтобы не потерять информацию об анализируемом языке. Например, при работе со встроенным SQL безопаснее считать таблицу используемой, чем удалить её как неиспользуемую, если она на самом деле использовалась. При этом необходимо следить за точностью такой аппроксимации, чтобы

избегать слишком большого количества ошибок. В работе [54] показано, как можно строить достаточно точную регулярную аппроксимацию сверху с учётом строковых операций и циклов.

Таким образом, далее мы будем предполагать, что язык  $L_2$  является однозначным контекстно-свободным и задан некоторой грамматикой  $G$ , а для языка  $L_1$  строится регулярный язык  $L_R$  такой, что  $L_1 \subseteq L_R$  и мы работаем далее с  $L_R$ . При этом основной нашей задачей является построение деревьев разбора для всех цепочек из  $L_R$ , корректных относительно грамматики  $G$ . Однако  $L_R$  может быть бесконечным языком и, следовательно, содержать бесконечное множество корректных цепочек. По этой причине явное построение всех деревьев разбора не представляется возможным. Решением, возможно, будет являться структура, которая при конечном объёме будет хранить бесконечное количество деревьев. При этом деревья должны однозначно извлекаться из этой структуры. То есть из построенной структуры можно получить только те и только те деревья, которые соответствуют разбору какой-либо корректной цепочки из  $L_R$  языка в эталонной грамматике  $G$ .

Регулярный язык  $L_R$  может быть представлен в виде конечного автомата. Заметим, что аппроксимация, построенная непосредственно по исходному коду, будет являться конечным автоматом  $M_1 = (Q_1, \Sigma_1, \delta_1, q_{0_1}, q_{f_1})$  над алфавитом символов, что не очень удобно для проведения синтаксического анализа, так как грамматика  $G = \langle T, N, P, S \rangle$  задана, скорее всего, над алфавитом токенов и  $\Sigma_1 \not\subseteq T$ . Чтобы устранить эту проблему, можно воспользоваться конечным преобразователем, который предназначен для трансформации языков. Таким образом, можно получить конечный автомат  $M_2 = (Q_2, \Sigma_2, \delta_2, q_{0_2}, q_{f_2})$  такой, что  $\Sigma_2 \subseteq T$ . Преобразование конечного автомата над алфавитом символов в конечный автомат над алфавитом токенов называется лексическим анализом регулярной аппроксимации или встроенного языка.

Заметим так же, что без потери общности можно считать, что  $M_2$  является детерминированным конечным автоматом без  $\varepsilon$ -переходов, так как любой конечный автомат можно преобразовать к эквивалентному детерминированному без  $\varepsilon$ -переходов.

Таким образом, решаемая в данной работе задача синтаксического анализа динамически формируемых выражений будет формулироваться следую-

щим образом. Для данной однозначной контекстно-свободной грамматики  $G = \langle T, N, P, S \rangle$  и детерминированного конечного автомата без  $\varepsilon$ -переходов  $M = (Q, \Sigma, \delta, q_0, q_f)$  такого, что  $\Sigma \subseteq T$ , необходимо построить конечную структуру данных  $F$ , содержащую деревья вывода в  $G$  всех цепочек  $\omega \in L(M)$ , корректных относительно грамматики  $G$ , и не содержащую других деревьев. Иными словами, необходимо построить алгоритм  $\mathbb{P}$  такой, что

$$(\forall \omega \in L(M))(\omega \in L(G) \Rightarrow (\exists t \in \mathbb{P}(L(M), G))AST(t, \omega, G))$$

$$\wedge (\forall t \in \mathbb{P}(L(M), G))(\exists \omega \in L(M))AST(t, \omega, G).$$

Здесь  $AST(t, \omega, G)$  — это предикат, который истинен, если  $t$  является деревом вывода  $\omega$  в грамматике  $G$ .

Так как  $\mathbb{P}$  игнорирует ошибки, то будем называть его алгоритмом *ослабленного* (relaxed) синтаксического анализа регулярной аппроксимации динамически формируемого выражения.

## 2.2 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  $\varepsilon$ -переходов, где  $\Sigma \subseteq T$  — алфавит входных символов,  $Q$  — множество состояний,  $\delta$  — отношение перехода. По описанию грамматики генерируются управляющие RNGLR-таблицы и некоторая вспомогательная информация (называемая *parserSource* в псевдокоде, см. листинг 2.1).

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

---

```

1: function PARSE(grammar, automaton)
2:   inputGraph  $\leftarrow$  construct inner graph representation of automaton
3:   parserSource  $\leftarrow$  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  $\leftarrow$  Q.Dequeue()
12:      MAKEREDUCTIONS(v)
13:      PUSH(v)
14:      APPLYPASSINGREDUCTIONS(v)
15:      if  $\exists v_f : v_f.level = q_f$  and vf.state is accepting then report success
16:      else report failure

```

---

Листинг 2.1: Алгоритм ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения

shift и reduce (смотри строку 5 в алгоритме 2.2 и строки 9 и 21 в алгоритме 2.3). Для того, чтобы управлять порядком обработки вершин входного графа, мы используем глобальную очередь  $Q$ . 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 2.1. 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 листинге 2.2).

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 изложе-



---

```

1: function PUSH(innerGraphV)
2:    $\mathcal{U} \leftarrow \text{copy } innerGraphV.unprocessed$ 
3:   clear innerGraphV.unprocessed
4:   for all  $v_h$  in  $\mathcal{U}$  do
5:     for all  $e$  in outgoing edges of innerGraphV do
6:        $push \leftarrow \text{calculate next state by } v_h.state \text{ and the token on } e$ 
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:    ( $startV, N, l$ )  $\leftarrow innerGraphV.reductions.Dequeue()$ 
12:    find the set of vertices  $\mathcal{X}$  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  $\mathcal{X}$  do
17:       $state_t \leftarrow \text{calculate new state by } v_h.state \text{ and nonterminal } N$ 
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 ( $startV, N, l$ )  $\leftarrow v.passingReductions.Dequeue()$  do
22:      find the set of vertices  $\mathcal{X}$ ,
23:      reachable from  $edge$  along the path of length  $(l - 1)$ 
24:      for all  $v_h$  in  $\mathcal{X}$  do
25:         $state_t \leftarrow \text{calculate new state by } v_h.state \text{ and nonterminal } N$ 
26:        ADDEDGE( $v_h, startV, state_t, false$ )

```

---

Листинг 2.2: Обработка вершины внутреннего графа

нии 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 только 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.

---

```

1: function ADDVERTEX(innerGraphV, state)
2:    $v \leftarrow$  find a vertex with state = state in
3:     innerGraphV.processed  $\cup$  innerGraphV.unprocessed
4:   if  $v$  is not null then                                      $\triangleright$  Вершина была найдена в GSS
5:     return ( $v$ , false)
6:   else
7:      $v \leftarrow$  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, statet, isZeroReduction)
14:   ( $v_t$ , isNew)  $\leftarrow$  ADDVERTEX(innerGraphV, statet)
15:   if GSS does not contain edge from  $v_t$  to  $v_h$  then
16:      $edge \leftarrow$  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

```

---

Листинг 2.3: Построение GSS

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 2.2).

## 2.2.1 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 2.4 код динамически формирует выражение **expr** в строке 4.

---

```

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

---

Листинг 2.4: Пример кода на языке программирования C#, динамически формирующего скобочную последовательность

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 2.1.



Рисунок 2.1: Конечный автомат, задающий регулярную аппроксимацию выражения **expr**

В результате работы предложенного алгоритма будет получено конечное представление леса разбора SPPF, изображённое на рисунке 2.2.



Рисунок 2.2: Конечное представление леса разбора для выражения **expr**



Рисунок 2.3: Дерево вывода для выражения  $expr = "()"$



Рисунок 2.4: Дерево вывода для выражения  $expr = "()()"$



Рисунок 2.5: Дерево вывода для выражения  $expr = "()()()"$

Из построенного SPPF можно извлечь бесконечное количество деревьев, каждое из которых является деревом вывода некоторой цепочки из регулярной аппроксимации. На серии рисунков 2.3, 2.4, 2.5 представлены извлечённые деревья разбора для различных значений выражения **expr**.

## 2.3 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  $Q$  только тогда, когда происходит добавление нового ребра в GSS. Так как количество ребер в GSS конечно, алгоритм завершает работу для любых входных данных.  $\square$

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  $\varepsilon$ -дерево, либо дерево, состоящее из единственной вершины-терминала. При этом  $\varepsilon$ -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  $N \rightarrow A_0, A_1, \dots, A_n$ , где  $A_0, A_1, \dots, A_n$  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.  $\square$

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.  $\square$

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 [18], 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.  $\square$

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.

## Глава 3

# Инструментальный пакет

В данной главе описан инструментальный пакет (Software Development Kit, SDK) **YC.SEL.SDK**, предназначенный для разработки различных решений по статическому анализу динамически формируемых выражений. Представлена архитектура разработанного SDK, а также архитектура надстройки **YC.SEL.SDK.ReSharper**, позволяющей создавать расширения для ReSharper, предоставляющие поддержку встроенных языков. Изложенный выше алгоритм синтаксического анализа реализован в рамках одной из компонент SDK. YC.SEL.SDK и YC.SEL.SDK.ReSharper являются **платформами** для разработки инструментов статического анализа динамически формируемого кода.

### 3.1 Архитектура

Практически любой язык программирования может использоваться как встроенный. Даже если рассматривать только SQL, то окажется, что у него множество различных диалектов, каждый из которых имеет свои особенности. Внешним языком также может быть любой язык программирования. Трудность заключается в том, что любое из сочетаний внешнего и встроенного языка может встретиться на практике, и задачи, которые необходимо решать в этой ситуации, могут быть различными (поиск ошибок, подсчёт метрик, автоматизация трансформаций и т.д.). Реализовать универсальный инструмент, решающий все задачи для всех языков, не представляется возможным. Более целесообразно создать набор инструментов, упрощающий создание конечных решений для конкретных

языков и конкретных задач. В качестве примера можно рассмотреть инструменты для разработки компиляторов, которые включают в себя генераторы лексических, синтаксических анализаторов и набор библиотек с вспомогательными функциями, и тем самым упрощают создание конкретного компилятора для выбранного языка и целевой платформы.

Требуемый набор инструментов для работы со встроенными языками должен поддерживать весь процесс обработки кода, который может выглядеть так, как представлено на рисунке 3.1. Можно выделить следующие основные шаги.

- Анализ основного кода, который выполняется сторонним инструментом. Результат этого шага — это дерево разбора с информацией, достаточной для выполнения дальнейших шагов.
- Построение аппроксимации множества возможных значений динамически формируемых выражений.
- Лексический анализ построенной на предыдущем шаге аппроксимации.
- Синтаксический анализ, результатом которого является лес разбора, пригодный для дальнейшей обработки.
- Обработка леса разбора, вычисление семантики.

На каждом шаге может быть получена информация, полезная для пользователя, такая как список ошибок, и её необходимо отобразить для него соответствующим образом.

Существующие инструменты для работы со встроенными языками обычно реализуют поддержку какого-то фиксированного набора языков. При этом поддержка нового языка, как правило, требует нетривиальной доработки инструмента. Чтобы получить поддержку встроенного языка без изменений в исходном коде базового инструмента, необходимо предоставить соответствующий механизм.

Для того чтобы упростить процесс создания конечных инструментов, создан SDK, одной из компонент которого является генератор синтаксических анализаторов на основе предложенного в данной работе алгоритма. Также в него входит генератор лексических анализаторов, библиотека построения регулярной ап-

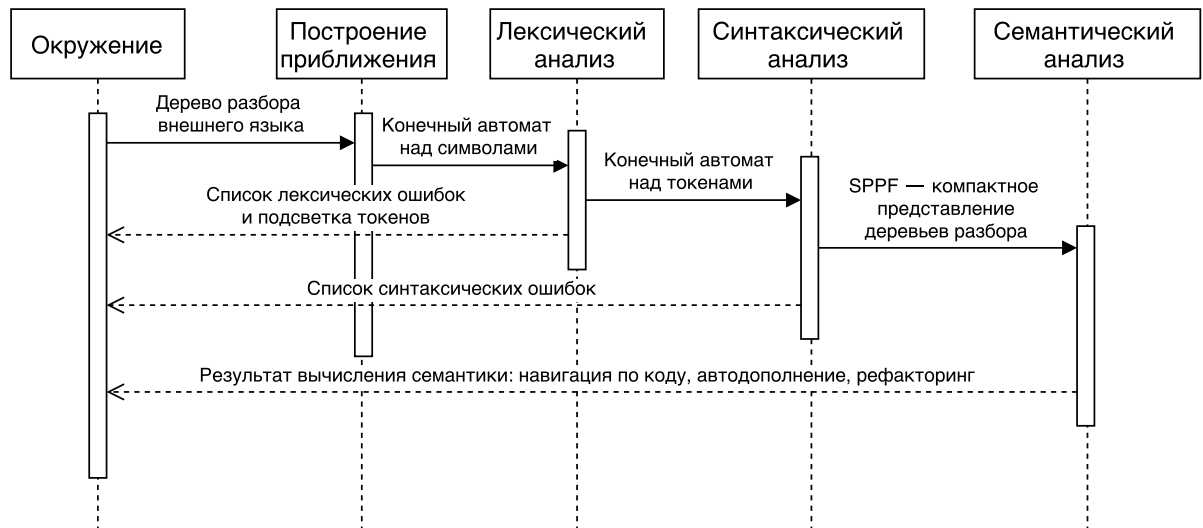


Рисунок 3.1: Диаграмма последовательности обработки встроенных языков

проксимации, набор вспомогательных функций. Подробное описание компонент приведено далее.

Так как анализ внешнего языка является сложной самостоятельной задачей, то он не включён в разработанный SDK. На вход созданному на основе SDK инструменту должно подаваться дерево разбора внешнего языка с информацией, достаточной для решения поставленных в данной работе задач.

### 3.1.1 Архитектура YS.SEL.SDK

Разработанный SDK включает компоненты, необходимые для реализации шагов, представленных на рисунке 3.1 и описанных ранее, за исключением анализа внешнего языка. Архитектура SDK изображена на рисунке 3.2 и включает в себя генераторы анализаторов и различные библиотеки времени исполнения.

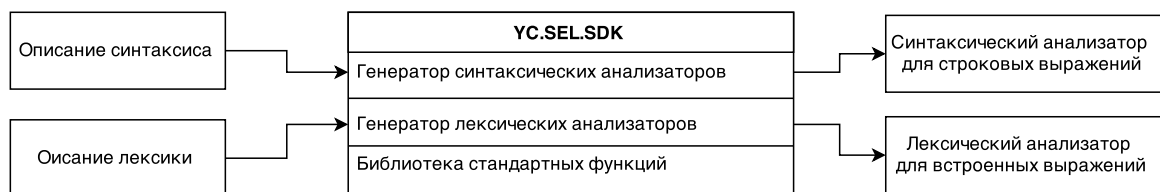


Рисунок 3.2: Архитектура SDK целиком

Так как анализ внешнего языка не входит в задачи разработанного SDK, то первый шаг, выполнение которого необходимо обеспечить, — это построение

аппроксимации. В нашем случае строится регулярное приближение множества значений динамически формируемого выражения.

Построение регулярной аппроксимации основано на алгоритме, изложенном в работе [54], который позволяет строить приближение сверху для множества значений выражений. То есть  $L_R$ , задаваемый регулярным приближением, не меньше, чем  $L_1$ , задаваемый программой (выполняется включение  $L_R \in L_1$ ). Это позволяет говорить о надёжности дальнейших анализов в том смысле, что они не теряют информации о  $L_1$ . Например, это важно при поиске ошибок. Если в  $L_R$  не обнаружено ошибок (то есть  $L_R \in L_2$ , где  $L_2$  — эталонный), значит и в  $L_1$  ошибок нет. При этом могут быть найдены ошибки в  $L_R$ , которых нет в  $L_1$ , то есть будут ложные срабатывания. Однако наличие ложных срабатываний лучше, чем пропущенные ошибки, и их количество может быть уменьшено путём повышения точности аппроксимации.

Для того чтобы сделать построение приближения независимым от внешнего языка, реализовано обобщённое представление графа потока управления (Control Flow Graph, CFG) [38], которое содержит всю необходимую для дальнейшей работы информацию. Таким образом, разработчику необходимо реализовать построение обобщённого представления CFG для конкретного внешнего языка. В результате компонента строит конечный автомат, являющийся приближением множества значений динамически формируемых выражений.

Компонента, отвечающая за лексический анализ, состоит из двух частей: генератора лексических анализаторов, который по описанию лексики обрабатываемого языка строит соответствующий конечный преобразователь, и интерпретатора, который производит анализ входной структуры данных на основе построенного генератором преобразователя. Архитектура компоненты представлена на рисунке 3.3. На вход принимается конечный автомат над символами, результатом работы является конечный автомат над алфавитом токенов анализируемого языка. Входной конечный автомат может быть построен с помощью компоненты построения регулярной аппроксимации. Основные структуры данных — конечный автомат и конечный преобразователь — и функции работы с ними описаны в соответствующей библиотеке.

Лексический анализатор реализован на основе инструмента FsLex, который является стандартным генератором лексических анализаторов для языка F#. При

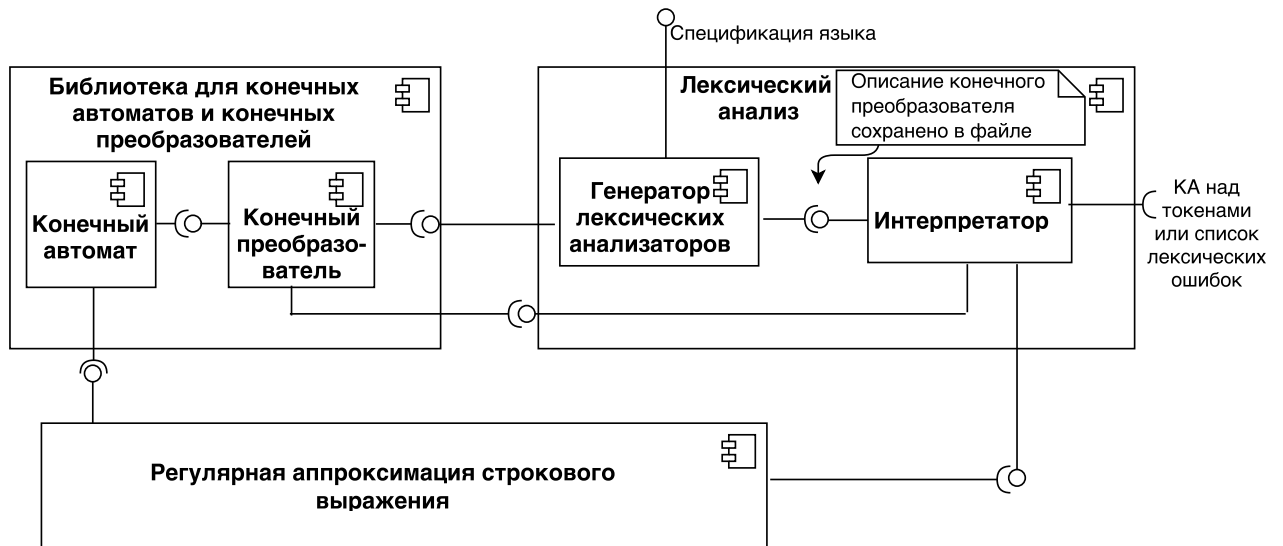


Рисунок 3.3: Архитектура лексического анализатора

реализации был переиспользован язык описания лексики и некоторые структуры данных.

Реализованный генератор лексических анализаторов обладает следующими особенностями.

- Поддерживаются разрывные токены, то есть токены формируемые из нескольких строковых литералов.
- Сохраняется привязка лексических единиц к исходному коду: сохраняется информация о строковом литерале, из которого породился токен и координаты его внутри этой строки. Так как одна лексическая единица может формироваться из нескольких строковых литералов, то привязка сохраняется отдельно для каждой части.
- Поддерживается обработка входных конечных автоматов, содержащих циклы.
- Так как значение токена может формироваться с помощью цикла и, как следствие, быть бесконечным, то каждый токен содержит конечный автомат, порождающий все возможные значения для данного токена, а не единственное значение, как это реализовано в классическом лексическом анализе.

**Генератор синтаксических анализаторов**, названный ARNGLR, реализован на основе алгоритма, описанного в разделе 2.2. Его архитектура представлена на рисунке 3.4.

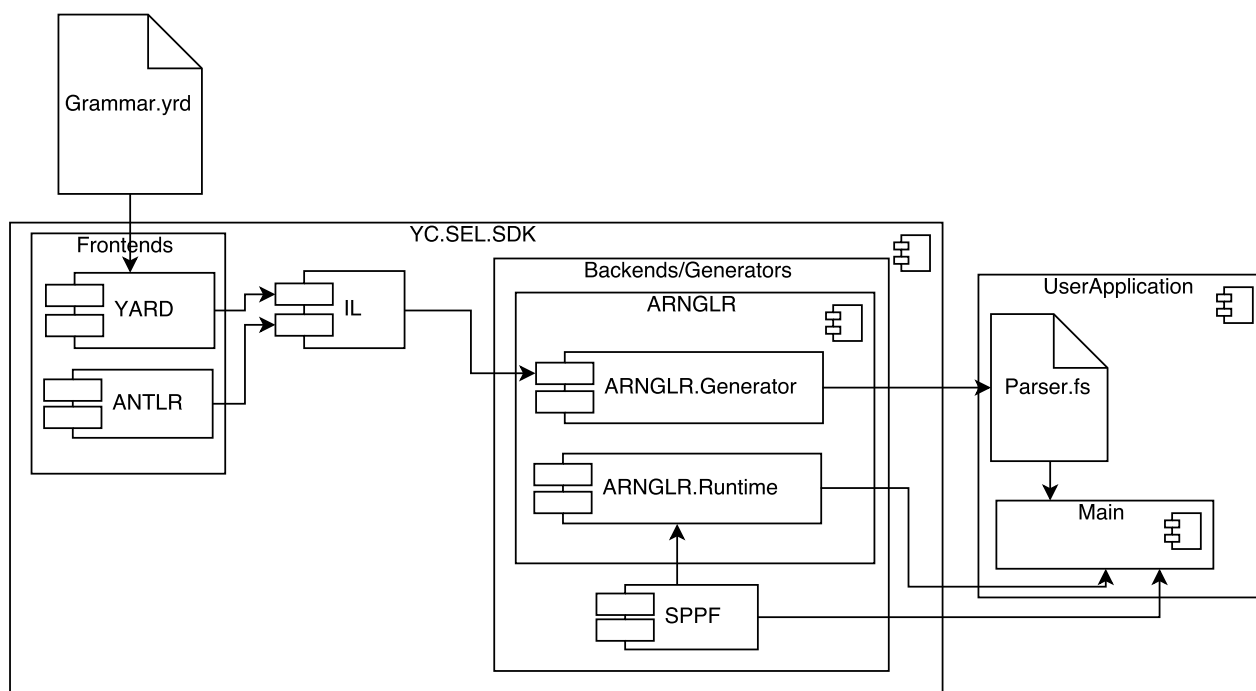


Рисунок 3.4: Архитектура синтаксического анализатора

Генератор реализован как один из модулей YC, а значит может принимать на вход внутреннее представление грамматики (IL), которое может быть получено с помощью различных фронтов, однако в рамках YC.SEL.SDK основным фронтом является YARD, так как он предоставляет наиболее развитые средства для описания грамматик. По грамматике обрабатываемого языка строятся управляющие таблицы анализатора. Построенные таблицы должны быть включены в разрабатываемое приложение. Интерпретатор, предназначенный для синтаксического разбора конечного автомата, полученный после лексического анализа, реализован в виде отдельной библиотеки, которая также должна быть подключена к разрабатываемому приложению. В результате работы интерпретатора будет получен SPPF, который может быть использован для дальнейшей обработки (например, подсчёта метрик). Для упрощения работы с SPPF реализован ряд вспомогательных функций.

### 3.1.2 Архитектура YC.SEL.SDK.ReSharper

ReSharper — это расширение к Microsoft Visual Studio IDE, предоставляющее широкий спектр дополнительной функциональности по анализу и рефакторингу кода. ReSharper поддерживает несколько языков, например C#, Visual Basic .NET, JavaScript, и этот список может быть расширен благодаря наличию свободно распространяемого ReSharper SDK, описание которого было представлено ранее в разделе 1.7.2. ReSharper.SDK позволяет получить деревья разбора для поддерживаемых языков, предоставляет набор готовых анализов и упрощает взаимодействие с Microsoft Visual Studio IDE и её компонентами. Более того, предоставляется возможность разработки собственных расширений для ReSharper на основе ReSharper.SDK.

Microsoft Visual Studio является достаточно распространённой средой разработки, но не поддерживает встроенные языки, поэтому было решено разработать ряд расширений к ReSharper с использованием разработанного инструментария, которые будут устранять данный недостаток. Стоит отметить, что не ставилось задачи поддерживать все встроенные языки, так как встроенным может быть любой язык программирования. Также не было необходимости поддерживать все внешние языки программирования. Необходимо на базе разработанного YC.SEL.SDK создать инфраструктуру, позволяющую реализовывать поддержку новых встроенных языков в Microsoft Visual Studio через расширения к ReSharper и реализовать несколько расширений, демонстрирующих возможности созданной инфраструктуры.

Так как необходимо поддерживать различные языки, то необходимо обеспечить расширяемость по новыми языками. Классический подход к решению такой задачи для интегрированных сред разработки заключается в том, что поддержка нового языка реализуется в виде независимой компоненты. Если пользователь хочет получить поддержку какого-либо языка в своей среде разработки, то он должен установить соответствующий пакет. При этом поддержка различных языков осуществляется независимо, однако часто выделяется общая функциональность, которая может быть оформлена в виде отдельного пакета.

Для предоставления описанных выше возможностей была реализована надстройка над YC.SEL.SDK, упрощающая создание расширений для ReSharper, названная YC.SEL.SDK.ReSharper. В неё включены компоненты, ре-



ализующие функции, которые упрощают взаимодействие YC.SEL.SDK и ReSharper.SDK. Назначены основных из них описаны ниже.

- Общая точка расширения, необходимая для подключения функциональности для различных встроенных языков, которая может быть реализована в различных расширениях, к ReSharper через общий интерфейс. Также общая точка расширения позволяет использовать общую функциональность, необходимую для работы со встроенными языками.
- Отображение в IDE информации, полученной в ходе анализа. Например, подсветка синтаксиса и ошибок. Вывод диагностических сообщений с информацией об ошибках.
- Преобразование данных, из формата, используемого в ReSharper.SDK, в формат для YC.SEL.SDK. Например, преобразование графа потока управления внешнего языка, построенного ReSharper.SDK, в формат, пригодный для построения регулярной аппроксимации средствами YC.SEL.SDK.
- Управление работой анализаторов, необходимое, с одной стороны, для обеспечения своевременной реакции на изменения в коде, совершённые пользователем, а с другой, для прекращения вычислений, результаты которых уже не актуальны. Управление построено на основе общего для ReSharper механизма, обеспечивающего асинхронную работу анализов. При этом вычисления могут быть прерваны, если, например, пользователь внёс в код изменения, делающие анализ или его результаты некорректными.

Как уже говорилось, встроенными могут быть различные языки и учесть заранее все их особенности не представляется возможным. Кроме того, даже при использовании одного встроенного языка могут использоваться различные способы выполнения сформированного запроса. Таким образом, необходимо предоставлять возможность настройки расширений конечным пользователем. Для этого в рамках YC.SEL.SDK.ReSharper была реализована возможность управления следующими основными параметрами расширений.

- Подсветка синтаксиса для каждого языка. Предоставлена возможность указать цвет для каждого типа токена.

- Указание парных элементов. Для каждого языка можно указать, какие лексические единицы считать парными: для каждой пары указывается “левый” (открывающая скобка) и “правый” (закрывающая скобка) элементы. При расположении курсора в тексте рядом с одним из элементов пары будут подсвечены соответствующие элементы. Пример подсветки парных элементов приведён на рисунке 5.9.
- Точки интереса или хотспоты (hotspot) — это места, в которых должно быть сформировано финальное выражение. Необходимо знать, какой хотспот какому языку соответствует. При этом нужно учитывать, что одному языку может соответствовать несколько хотспотов. Например, динамически сформированный SQL-запрос в программе на языке программирования C# может быть выполнен с помощью метода `ExecuteQuery` класса `DataContext`<sup>1</sup> или же текст запроса может быть передан как аргумент конструктора класса `SqlCommand`<sup>2</sup> с последующим выполнением с помощью метода `ExecuteReader`.

Настройка указанных выше параметров хранится в соответствующих конфигурационных файлах в формате XML, которые на данный момент редактируются вручную. Настройка подсветки синтаксиса и парных элементов совмещена в одном файле и для каждого языка создаётся отдельный такой файл. Конфигурационный файл с точками интереса является общим для всех языков и, соответственно, для всех установленных расширений для поддержки встроенных языков.

В листинге 3.1 приведён пример конфигурации подсветки синтаксиса и парных скобок для языка Calc. Для указания цвета используются имена, принятые в ReSharper (например, "CONSTANT\_IDENTIFIER\_ATTRIBUTE"), что должно сделать настройку цветов более единообразной. В xml-тэге `Matched` содержится описание парных элементов. Каждая пара описывается в xml-тэге `Pair` и для одного языка может быть указано более одной такой пары.

<sup>1</sup> Документация метода `ExecuteQuery` (посещено 28.06.2015): [https://msdn.microsoft.com/en-us/library/system.data.linq.datacontext.executequery\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.data.linq.datacontext.executequery(v=vs.110).aspx)

<sup>2</sup> Документация класса `SqlCommand` (посещено 28.06.2015): [https://msdn.microsoft.com/ru-ru/library/sebfsz50\(v=vs.110\).aspx](https://msdn.microsoft.com/ru-ru/library/sebfsz50(v=vs.110).aspx)

---

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <SyntaxDefinition name="CalcHighlighting">
3     <Colors>
4         <Tokens color="CONSTANT_IDENTIFIER_ATTRIBUTE">
5             <Token> DIV </Token>
6             <Token> LBRACE </Token>
7             <Token> MINUS </Token>
8             <Token> MULT </Token>
9             <Token> NUMBER </Token>
10            <Token> PLUS </Token>
11            <Token> POW </Token>
12            <Token> RBRACE </Token>
13        </Tokens>
14    </Colors>
15    <!-- Dynamic highlighting: -->
16    <Matched>
17        <Pair>
18            <Left> LBRACE </Left>
19            <Right> RBRACE </Right>
20        </Pair>
21    <!-- You can specify more then one pair:
22        <Pair>
23            <Left> LEFT_SQUARE_BRACKET </Left>
24            <Right> RIGHT_SQUARE_BRACKET </Right>
25        </Pair>
26        <Pair>
27            <Left> LEFT_FIGURE_BRACKET </Left>
28            <Right> LEFT_FIGURE_BRACKET </Right>
29        </Pair>
30    -->
31    </Matched>
32 </SyntaxDefinition>

```

---

Листинг 3.1: Пример конфигурационного файла для настройки подсветки синтаксиса

Листинг 3.2 содержит пример описания точек интереса. Для каждой точки интереса должна быть указана следующая информация.

- Какому встроенному языку соответствует точка. Тэг `Language`.

- Полное имя метода, являющегося точкой интереса. Тэг Method.
- Порядковый номер аргумента данного метода, являющегося выражением на встроенном языке. Нумерация начинается с нуля. Тэг ArgumentPosition.
- Возвращаемый тип метода. Тэг ReturnType.

---

```

1 <?xml version="1.0" encoding="utf-8"?>
2 <!-- comment about body -->
3 <Body>
4     <!-- comment about hotspot -->
5     <Hotspot>
6         <!-- comment about tsql -->
7         <Language> TSQL </Language>
8         <!-- comment about fullName -->
9         <Method>Program.ExecuteImmediate</Method>
10        <!-- zero-based -->
11        <ArgumentPosition> 0 </ArgumentPosition>
12        <!-- comment about return type -->
13        <ReturnType> void </ReturnType>
14    </Hotspot>
15    <Hotspot>
16        <Language> Calc </Language>
17        <Method>Program.Eval</Method>
18        <ArgumentPosition> 0 </ArgumentPosition>
19        <ReturnType> int </ReturnType>
20    </Hotspot>
21 </Body>

```

---

Листинг 3.2: Пример конфигурационного файла для настройки точек интереса

## 3.2 Области и способы применения YC.SEL.SDK

Разработанный SDK предназначен для создания инструментов статического анализа динамически формируемых строковых выражений. Решения, созданные с его помощью, могут применяться для работы с проектами, активно использую-

щими динамически формируемые строковые выражения. Необходимость работать с такими проектами может возникнуть, например, в следующих областях.

- Реинжиниринг программного обеспечения.
- Поддержка встроенных языков в средах разработки.
- Оценка качества и сложности кода.

Общим для всех этих областей является то, что для решения многих задач необходимо структурное представление динамически формируемого кода. При этом анализируемые языки могут быть различными и процесс их анализа часто тесно связан с анализом внешнего языка.

Отметим, что встроенные языки используются всё менее активно в молодых проектах и системах. На смену им приходят более надёжные способы композиции языков и метапрограммирования. Например LINQ или ORM-технологии. Однако это не всегда так. Использование строковых выражений для взаимодействия с базами данных и генерации WEB-страниц в приложениях на PHP всё ещё широко распространено [5]. Это необходимо учитывать при поддержке встроенных языков в средах разработки. Для каких-то языков на первый план выходят возможности по изучению и модификации уже созданного кода, а для каких-то — возможность быстро и удобно создавать новый код. Во втором случае могут возникнуть дополнительные требования к скорости работы инструмента, так как подразумевается выполнение некоторых операций “на лету”, что может послужить ограничением на использование SDK, так как многие механизмы, реализованные в нём, не предусматривают возможности уменьшения точности в пользу увеличения быстродействия. Оценка качества и сложности кода часто может выполняться в рамках комплекса задач по реинжинирингу системы, однако может быть и самостоятельной задачей, например, при оценке сложности работ по поддержке и сопровождению информационной системы.

Детали применения SDK могут варьироваться в зависимости от решаемых задач и контекста использования. Например, механизм построения регулярной аппроксимации может быть реализован независимо в рамках внешнего инструмента. Однако основной сценарий использования аналогичен использованию инструментариев для разработки компиляторов. Последовательность ша-

гов, представленная ниже, может быть изменена в зависимости от особенностей задачи.

- Создание грамматики обрабатываемого языка. Грамматика может быть создана на основе документации соответствующего языка или переиспользована готовая, что оправданно, например, при создании анализатора для динамического SQL, когда внешний и встроенный языки совпадают.
- Генерация синтаксического анализатора по созданной грамматике. Для этого используется генератор синтаксических анализаторов, присутствующий в SDK. Результатом работы генератора является файл с исходным кодом на языке программирования F#, который должен быть включён в разрабатываемый код. Файл содержит описание типов для лексических единиц, управляющие таблицы анализатора и функцию, которая по конечному автомату над алфавитом токенов анализируемого языка построит SPPF, содержащий деревья вывода всех корректных цепочек.
- Создание лексической спецификации обрабатываемого языка. Спецификация может быть извлечена из документации или заимствована из других проектов. При обработке динамически формируемого SQL возможно переиспользовать спецификацию, созданную для основного языка, которым также является SQL. При этом необходимо обратить внимание на то, что типы лексических единиц определяются на основе созданного на предыдущих шагах синтаксического анализатора.
- Генерация лексера по созданной спецификации. Для этого применяется генератор лексических анализаторов, входящий в состав SDK. В результате его применения получается файл с исходным кодом на языке F#, который должен быть подключён к разрабатываемому решению.
- Реализация механизма построения регулярной аппроксимации, результатом которого является функция, строящая конечный автомат над алфавитом символов. Данный механизм может быть реализован либо на основе предоставляемого в рамках SDK, либо независимо. В первом случае от разработчика требуется построить обобщённый CFG для внешнего языка. Во

втором случае необходимо только гарантировать правильность возвращаемого конечного автомата. Второй подход может быть использован, например, при наличии реализованного механизма протягивания констант для внешнего языка. Это позволит создать возможно менее точное, но, скорее всего, более быстрое построение аппроксимации. Такой подход применим при автоматизированном реинжиниринге, когда ручная доработка кода является обязательным шагом и абсолютная точность автоматической обработки не требуется. Ещё одна возможная область применения второго подхода — это поддержка встроенных языков в средах разработки. Здесь также часто не требуется высокая точность для подсказок пользователю, однако производительность крайне важна. Поэтому иногда приходится жертвовать точностью анализа для достижения нужной скорости работы.

- Реализация работы с SPPF. Синтаксический анализатор возвращает SPPF — конечное представление леса разбора всех корректных цепочек из аппроксимации. Дальнейшая работа с ним может строиться по двум основным сценариям.
  - Непосредственная обработка SPPF. В этом случае все вычисления происходят над SPPF без извлечения отдельных деревьев. Это позволит ускорить обработку результатов разбора, так как количество деревьев может быть бесконечным, а SPPF является конечной структурой данных. Однако существует несколько проблем, связанных с таким подходом. Во-первых, требуется создание новых процедур обработки, так как классические, как правило, ориентированы на работу с деревьями. Во-вторых, могут возникнуть трудности при выполнении некоторых анализов, вызванные тем, что в SPPF хранятся “бесконечные” деревья. Например, необходимо вычислить максимальную глубину вложенности конструкции `if`, являющуюся одной из стандартных метрик сложности кода. SPPF может содержать циклы и может оказаться так, что конструкция `if` встречается в цикле таким образом, что потенциальная глубина вложенности может быть бесконечной. Такая ситуация не является стандартной при обработке деревьев разбора и её надо отслеживать отдельно.

- Извлечение отдельных деревьев из SPPF и их обработка. Данный подход может оказаться удобным, если уже существуют процедуры обработки синтаксических деревьев для языка, который оказался встроенным. Это помогает избежать затрат на создание новой функциональности. Такое может произойти при работе с динамическим SQL. В этом случае для работы с деревом разбора внешнего языка и деревьями, извлечёнными из SPPF, можно использовать одни и те же процедуры, так как языки идентичны.

Недостатком данного подхода является то, что конечность числа деревьев не гарантирована. Это значит, что не удастся обработать все деревья. Стоит отметить, что даже в случае конечности числа деревьев, перебор и обработка всех деревьев разбора может потребовать значительных ресурсов.

- Реализация механизмов сбора, обработки и отображения информации, такой как сообщения об ошибках или любой другой, полученной в процессе анализа. Необходимо для предоставления пользователю информации, ожидаемой в рамках решаемой задачи.

На рисунке 3.5 изображён один из возможных сценариев использования SDK. Особенностью является цикличность процесса, характерная, например для реинжиниринга программного обеспечения.

Встраивание анализа строковых выражений в последовательность обработки кода всей системы зависит от решаемых задач. Первыми шагами идут действия, необходимые для того, чтобы получить входные данные для анализа. Для этого необходимо провести лексический и синтаксический анализ внешнего языка, построить граф потока управления. После этого возможно построение аппроксимации и дальнейший анализ встроенных языков. Параллельно с этим может проводиться дальнейшая обработка внешнего языка. Степень параллельности зависит от независимости решаемых задач. Например, некоторые метрики сложности для основного кода и для динамически формируемого можно считать независимо и выводить отдельно. С другой стороны, может возникнуть необходимость вычислить некую комплексную метрику, учитывающую параметры и внешнего



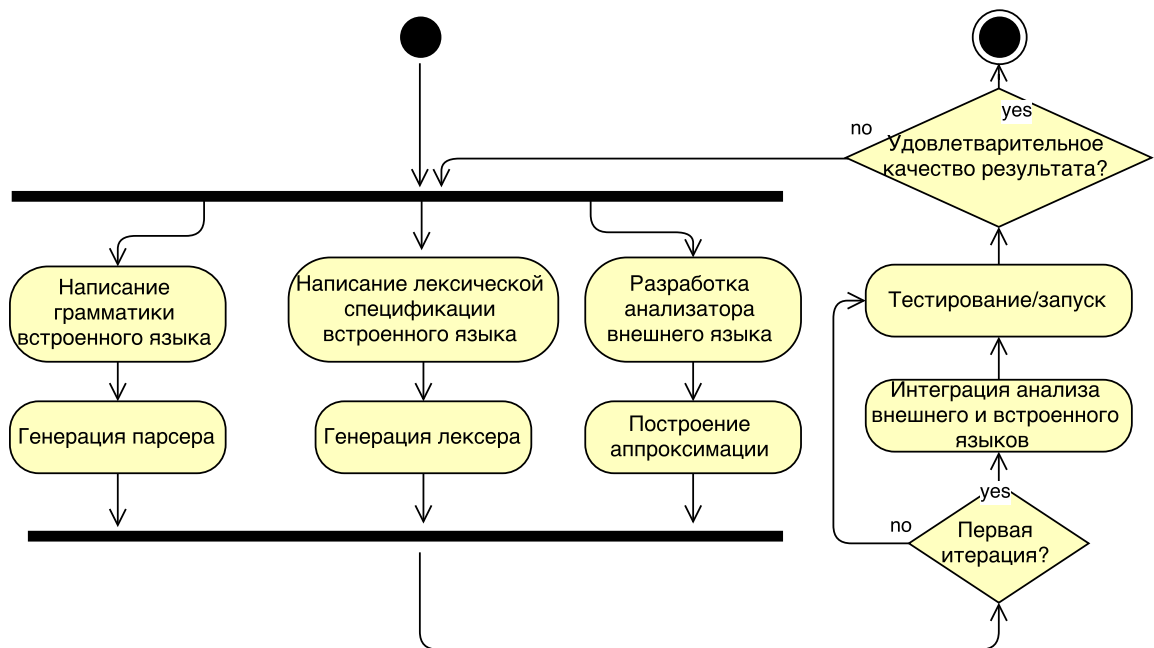


Рисунок 3.5: Один из возможных вариантов использования SDK в проектах по реинжинирингу

и динамически формируемого кода, что приведёт к необходимости синхронизации.

### 3.3 Особенности реализации

Разработка инструментального пакета с описанной выше архитектурой и плагинов для ReSharper велась в рамках исследовательского проекта YaccConstructor (YC), описанного в разделе 1.7.1. Разработка велась на платформе .NET и основным языком реализации — F# [52]. Весь исходный код опубликован на GitHub: <https://github.com/YaccConstructor>. Большинство компонент опубликовано под “открытой” лицензией Apache License Version 2.0 <sup>3</sup>.

За основу алгоритма синтаксического анализа динамически формируемых выражений был взят реализованный в YC алгоритм синтаксического анализа RNGLR. Генератор управляющих таблиц был использован практически без модификаций, а интерпретатор был реализован отдельный. Кроме этого, общими

<sup>3</sup>Одна из так называемых “открытых” лицензий. Сайт с текстом лицензии и сопроводительными материалами (посещено 28.06.2015): <http://www.apache.org/licenses/LICENSE-2.0>

являются некоторые структуры данных и вспомогательные функции, такие как представление леса разбора и его печать в формате DOT <sup>4</sup>, представление GSS.

Лексический анализ реализован на основе инструмента FsLex, который потребовал значительных доработок для того, чтобы обеспечить обработку конечного автомата, а не линейного входа. Все остальные компоненты, необходимые для статического анализа динамически формируемых выражений, такие как построение аппроксимации, вспомогательные функции для упрощения построения целевых инструментов были реализованы “с нуля” в рамках проекта YC.

Бинарные пакеты, содержащие основную функциональность, опубликованы в сети интернет на NuGet <sup>5</sup>.

---

<sup>4</sup>DOT — текстовый язык описания графов. Описание языка (посещено 28.06.2015): <http://www.graphviz.org/doc/info/lang.html>

<sup>5</sup>NuGet — менеджер пакетов для платформы .NET и одноимённый ресурс для их публикации. Позволяет публиковать и устанавливать пакеты, автоматически отслеживать зависимости между ними. Домашняя страница: <https://www.nuget.org/>

## Глава 4

# Метод реинжиниринга встроенного программного кода

В данной главе изложен метод проведения реинжиниринга информационных систем, использующих строковые встроенные языки (string-embedded languages [9]). Рассмотрены основные типы задач, решаемых при реинжиниринге, указаны особенности обработки встроенных языков при их решении. Основной акцент сделан на обработку динамически формируемого SQL-кода, однако изложенный метод может быть адаптирован к обработке систем, использующих другие встроенные языки.

### 4.1 Особенности

Реинжиниринг информационных систем как правило является комплексным мероприятием, которое может в себя включать различные шаги, такие как анализ кода и его изучение, его рефакторинг, трансформацию, перенос на другие программно-аппаратные платформы [1]. При этом часто проводится комплексный анализ многокомпонентной системы целиком, что приводит к необходимости учитывать различные особенности системы и работать с разнородными артефактами: документацией, исходным кодом, конфигурационными скриптами, скриптами баз данных и т.д. В связи с высокой сложностью такого анализа,

реинжиниринг, как правило, является автоматизированным, а не полностью автоматическим процесс, что означает возможность (и необходимость) активного участия человека. Это позволяет использовать инструменты, которые не приводят сразу к получению конечного результата, однако минимизируют ресурсы, необходимые для решения задачи. Например, может оказаться, что в системе присутствуют файлы особого формата и для их автоматической обработки потребуется создание уникального инструмента, что потребует значительных ресурсов. Однако объём этих файлов мал и их ручной анализ прост. В ситуации, когда затраты на создание инструмента значительно выше затрат на ручной анализ с тем же качеством результата, часто оказывается выгоднее отказаться от создания инструмента.

При этом, при реинжиниринге информационных систем необходимо учитывать и встроенный код. С одной стороны, динамически формируемый код может содержать важную информацию о системе. Например, в динамически формируемых SQL-запросах могут содержаться сведения о схеме данных и особенностями работы с базой данных, не извлекаемые из внешнего кода. С другой стороны, при активном использовании динамически формируемый код является сущностью, требующей отдельного внимания при обработке системы и, следовательно, должен быть учтён, например, при оценке сложности системы, часто проводимой на начальных этапах при определении сроков и стоимости работ.

При большом количестве встроенного кода, процесс его обработки необходимо автоматизировать. Однако, системы используют строковые встроенные языки с разной интенсивностью и разными способами. В зависимости от характера использования встроенных языков меняются требования к автоматизации их обработки: необходима ли инструментальная поддержка или ручная обработка потребует меньше ресурсов, чем создание или освоение инструмента, какими именно инструментами пользоваться в тех или иных случаях, на какие аспекты необходимо обратить внимание при принятии решений.

Метода по реинжинирингу информационных систем, учитывающего встроенный код, обнаружено не было. Далее представлен метод реинжиниринга информационных систем, использующих встроенный код, результатом которого являются рекомендации по выбору разработке конкретного инструмента для его обработки. Непосредственно обработка системы сильно зависит от особенно-

стей конкретного инструмента и решаемых задач и во многом должна быть изложена в инструкции по применению инструмента. По этой причине данный вопрос рассматриваться не будет.

## 4.2 Метод

Как было сказано ранее, встроенным языком может оказаться любой язык, однако на практике достаточно широко распространено использование динамически формируемого SQL-кода и генерация HTML-страниц. Использование динамически формируемого кода на других языках встречается реже. По этим причинам далее описан метод реинжиниринга систем, использующих встроенный SQL, однако он может быть адаптирован и к обработке систем, использующих другие встроенные языки. Общая последовательность шагов метода представлена на рисунке 4.1.

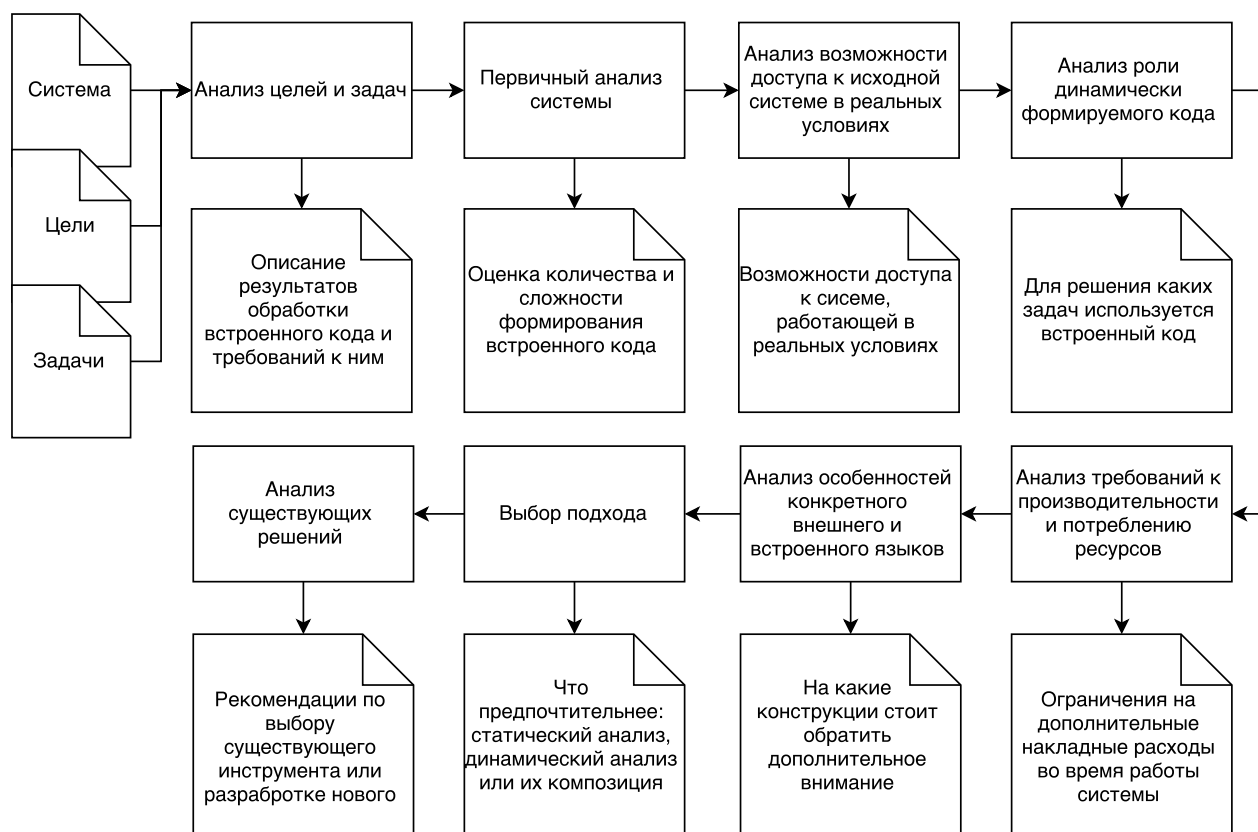


Рисунок 4.1: Основные шаги метода обработки встроенных языков и их результаты

В таблице 4.1 представлено подробное описание шагов с указанием вопросов, на которые необходимо ответить на каждом шаге и информации, необходимой для принятия решения. Обсуждение этих шагов с описанием некоторых деталей и примеров приведено далее.

№	Шаг	Вопросы	Ресурсы	Результат
1	Анализ целей и задач проекта.	Зачем  Что нужно сделать	Цели  Задачи	Цели реинжиниринга, для достижения которых необходима обработка встроенного кода. Список задач, при решении которых необходимо обрабатывать строковые выражения.
2	Первичный анализ системы.	Сколько в системе строковых выражений, требующих обработки. На сколько сложные строковые выражения используются.	Исходный код системы.  Исходный код системы.	Количество точек интереса.  Количество операторов для каждой точки интереса, участвующих в формировании соответствующего запроса.
3	Доступ к исходной системе в реальных условиях.	Важность наличия доступа.  Что доступно на чтение. Какие изменения возможно вносить в работающую систему.	Список задач, требующих обработки строковых выражений. Результаты шага 2: наличие динамически формируемых выражений. Заказчик.  Заказчик.	Есть ли задачи, решение которых упрощается при наличии доступа к работающей системе. Можно ли их решать при его отсутствии.  Какие данные и в каком объёме можно получать от работающей системы. Какого рода изменения и в какие модули можно вносить.

№	Шаг	Вопросы	Ресурсы	Результат
4	Роль встроенного кода в обрабатываемой системе.	<p>А надо ли</p> <p>Для чего используется встроенный код.</p> <p>На сколько активно используется встроенный код.</p>	<p>Список задач, требующих обработки строковых выражений.</p> <p>Код, документация, заказчик.</p> <p>Код, документация, заказчик</p>	<p>Есть ли задачи, отражающиеся на рантайме</p> <p>Функциональная и архитектурная роли встроенного кода: является ли его использование основой дизайна системы, в каких компонентах используется.</p> <p>Количество точек интереса на компоненту/модуль/процедуру, на сколько интенсивно выполняется, на сколько производительность данной компоненты критична.</p>
5	Анализ производительности и вычислительных ресурсов	<p>Необходимо ли проводить данный анализ.</p> <p>На сколько критичен встроенный код.</p> <p>Какие существуют возможности для оценки использования и доступности вычислительных ресурсов.</p> <p>Есть ли запас вычислительных ресурсов.</p>	<p>Список задач, требующих обработки строковых выражений.</p> <p>Результаты шага 4.</p> <p>Заказчик, результаты шага 3.</p> <p>Заказчик.</p>	<p>Есть ли задачи, решение которых требует дополнительных накладных расходов во время работы системы. Можно ли их решать иными способами.</p> <p>На сколько производительность компонент, использующих встроенный код, критична для системы и на сколько она зависит от производительности встроенного кода.</p> <p>Есть ли возможность практически измерить потребление и доступность ресурсов.</p> <p>Есть ли запас вычислительных ресурсов, который можно использовать, в условиях работы целевой системы.</p>
6	Анализ языковых особенностей.	<p>Выделение особенностей, потенциально влияющих на решение поставленных задач.</p> <p>Проверка наличия выделенных особенностей в системе.</p>	<p>Список задач, требующих обработки строковых выражений, информация о используемых языках и средствах работы со встроенным кодом.</p> <p>Исходный код системы.</p>	<p>Список особенностей, потенциально сложных мест, обработка которых может потребовать дополнительных затрат.</p> <p>Список особенностей, присутствующих в обрабатываемой системе, их классификация и описание.</p>

№	Шаг	Вопросы	Ресурсы	Результат
7	Выбор подхода.	<p>Требуется ли изменение встроенного кода и сопровождение результатов.</p> <p>Наличие доступа к системе, работающей в реальных условиях.</p> <p>Доступность дополнительных вычислительных ресурсов.</p> <p>Требуемая точность при решении поставленных задач.</p> <p>Анализ особенностей.</p> <p>Анализ поставленных задач.</p>	<p>Цели реинжиниринга, список задач, требующих обработки строковых выражений.</p> <p>Результаты шага 3, список задач, требующих обработки строковых выражений.</p> <p>Результаты шага 5.</p> <p>Список задач, требующих обработки строковых выражений.</p> <p>Результаты шага 6.</p> <p>Список задач, требующих обработки строковых выражений.</p>	<p>На сколько естественным для разработчика должен выглядеть результат изменения встроенного кода.</p> <p>Возможности по работе с системой в реальных условиях, упрощающие решение поставленных задач.</p> <p>Есть ли возможность использовать дополнительные вычислительные ресурсы для динамической обработки встроенного кода.</p> <p>Сравнение точности при решении поставленных задач с использованием динамического и статического анализа и её соотнесение с требуемой.</p> <p>Есть ли в обрабатываемой системе особенности, влияющие на выбор подхода.</p> <p>Есть ли задачи, требующие строго определённого подхода к решению.</p>
8	Составление рекомендаций по выбору инструмента.	<p>Требуется ли структурное представление встроенного кода.</p> <p>Как обрабатывать сложные случаи использования встроенного кода.</p> <p>Какой подход предпочтительнее: динамический или статический.</p> <p>Какая точность необходима при решении поставленных задач.</p>	<p>Список задач, требующих обработки строковых выражений.</p> <p>Результаты шага 6.</p> <p>Результаты шага 7.</p> <p>Цели, список задач, требующих обработки строковых выражений, заказчик.</p>	<p>Список задач, для решения которых требуется структурное представление встроенного кода.</p> <p>Дополнительные ограничения и требования, соблюдение которых необходимо для обработки сложных случаев, присутствующих в системе, и их реализуемость в рамках подходов и инструментов.</p> <p>Преимущества и недостатки подходов в контексте решаемых задач и ограничений и их соотнесение с возможностями инструментов.</p> <p>Оценка требуемой точности при решении поставленных задач и её сравнение с возможностями подходов и инструментов.</p>

Таблица 4.1: Основные шаги по подготовке к реинжинирингу системы, содержащей строковые выражения



Рассмотрим представленные в таблице 4.1 шаги более подробно.

### 1. Анализ целей и задач реинжиниринга.

На данном шаге необходимо проанализировать цели и задачи, стоящие в рамках проекта по реинжинирингу конкретной информационной системы. От того, какие задачи необходимо решать, напрямую зависит выбор средств и инструментов. Основные классы задач реинжиниринга информационных систем, затрагивающих динамически формируемый код, которые можно выделить на данном этапе, приведены ниже.

- Оценка качества кода и его сложности, что часто сводится к подсчёту некоторых формальных метрик кода [31,33]. Необходимо определить, требуется ли построение структурного представления кода для вычисления требуемых метрик с необходимой точностью.
- Анализ надёжности системы, поиск различного рода уязвимостей и ошибок. Требуется изучение типов ошибок, которые предполагается обнаруживать: обнаружены некоторые типы ошибок требуют построения структурного представления кода (семантические ошибки), в то время, как для других это не требуется (лексические, синтаксические ошибки).
- Извлечение или восстановлению утраченных знаний о системе и её изучение. В зависимости от деталей задачи, может потребоваться динамический анализ или структурное представление встроеного кода.
- Трансформация исходной системы: рефакторинг, перенос на новые программно-аппаратные системы. Для решения подобных задач как правило требуется структурное представление кода.

При этом необходимо проанализировать конечные цели реинжиниринга. Особенно важен данный шаг при необходимости решать задачи трансформации динамически формируемого кода. Например, если целью является как можно более быстрое получение работающей системы, дальнейшее развитие которой не планируется, то можно пожертвовать качеством кода результирующей системы, что может существенно упростить его обработку. Однако, если после выполнения трансформаций планируется активная

разработка или поддержка полученной системы, то необходимо получить результирующий код как можно более привычный для разработчиков системы. Это упростит дальнейшую работу с ним, позволит уменьшить затраты на обучение команды, поиск новых специалистов.

Например, при трансляции хранимого SQL-кода, активно использующего динамический SQL, с одного диалекта на другой, важно сохранить однородность в том смысле, что и основной код и встроенный должны быть написаны на одном и том же диалекте. Это особенно важно если планируется дальнейшее развитие системы. Обусловлено это тем, что различные диалекты SQL содержат большое количество особенностей и разработчики на SQL часто оказываются специалистами достаточно узкого профиля. Таким образом, наличие двух различных диалектов в месте одного может усложнить набор команды. Более того, в процессе разработки необходимость переключаться между несколькими диалектами так же может вызвать трудности.

Таким образом, один из основных вопросов, на которые необходимо ответить при анализе целей: планируется ли активное изменение системы после её реинжиниринга, если он включает трансформации.

2. **Первичный анализ проекта.** На данном этапе необходимо проанализировать исходный код системы и выяснить характеристики встроенного кода. Для этого, как правило, можно использовать стандартные средства анализа программного кода. Однако, скорее всего они будут требовать модификации, по этому необходимо иметь доступ к исходному коду. Необходимо оценить следующие характеристики кода.

- Количество точек интереса — тех точек, где сформированное строковое выражение передаётся на обработку соответствующей компоненте. Это позволит оценить интенсивность использования встроенного кода. Как правило, за выполнение выражений на встроенном языке отвечают характерные языковые конструкции, например EXECUTE<sup>1</sup> в языке T-SQL. следовательно, необходимо оценить количество та-

---

<sup>1</sup>Конструкция языка T-SQL, позволяющая выполнить динамически формируемый запрос. Документация: <https://msdn.microsoft.com/en-us/library/ms188332.aspx>. Посещён 29.07.2015.

ких конструкций. Для грубой оценки можно использовать простой текстовый поиск. Он может оказаться полезным, так как если такой поиск говорит о полном отсутствии соответствующих конструкций, то дальнейший анализ можно не проводить. Для более точной оценки необходимо использовать структурное представление кода, чтобы, например, не учитывать код, находящийся в комментариях. Получить структурное представление и доступ к нему можно с помощью библиотек синтаксического анализа соответствующего языка.

- Сложность формирования строковых выражений. Прежде всего необходимо определить наличие динамически формируемого кода и оценить сложность его формирования, так как в случае использования только константных строковых литералов не потребуются специальных инструментов для анализа встроенного кода.

Для данной оценки можно использовать протягивание констант [38], являющееся одним из стандартных шагов при статическом анализе кода. Однако его использование потребует модификаций. Необходимо чтобы отдельно обрабатывались выражения, отвечающие за формирование кода, и собиралась следующая информация о процессе формирования кода как для каждой точки выполнения, так и для всей системы в целом.

- Количество конкатенаций.
- Количество операторов ветвления: `if-then-else`, `switch-case` и т.д.
- Количество строковых функций: `replace`, `substring` и т.д.
- Количество циклов, как “явных” (`while`, `for`), так и организованных с помощью рекурсии.
- Количество переменных, значение для которых нельзя полностью вычислить статически (например, они получают значение из пользовательского входа).
- Факт формирования кода в телах более чем одного метода/процедуры. Для получения данной информации необходима поддержка межпроцедурного анализа в используемом инструменте.

**3. Возможности доступа к исходной системе в реальных условиях.** Возможность изучения системы в реальных условиях может дать большое количество полезной информации, такой как интенсивность выполнения встроеного кода, которую трудно или даже невозможно получить другими способами, так как даже тесты часто не воспроизводят реальное функционирование системы. Однако он часто оказывается невозможным или крайне затруднённым, что вызывается, с одной стороны вопросами безопасности, с другой стороны надёжностью, так как гарантировать, что внесённые изменения не отразятся на работоспособности системы часто затруднительно.

- Есть ли доступ к системе, работающей в реальных условиях? Что именно доступно: чтение определённых журнальных файлов, перехват сообщений на каком-либо уровне, доступ к реальной базе данных.
- Необходимо выяснить, модификации какого рода разрешены, какую информацию можно собирать, а какую нельзя. Это влияет на то, какие задачи можно решать при помощи динамического анализа.
- Возможен ли запуск модифицированной системы в реальных условиях? Если есть возможность запустить модифицированную систему, то важно убедиться, что будет возможность получить обратно собранную информацию. Наличие такой возможности может позволить, например, собрать информацию о том, с какими объектами работают с использованием динамически формируемых запросов при выполнении конкретных сценариев.

**4. Роль динамически формируемого кода в системе.** На данном шаге необходимо определить, какую роль в функционировании системы играют динамически формируемые запросы. Возможны следующие существенно различные ситуации.

- Использование динамически формируемого кода является один из основных элементов архитектуры системы. Данная ситуация, как правило, выделяется большим количеством точек исполнения и наличием

сложно формируемого кода. Важно определить, какие именно компоненты системы построены таким образом, так как использование встроенных языков может быть весьма неоднородным. Средние значения по всей системе могут быть не велики, при этом ряд ключевых компонент построены с активным использованием динамически формируемого кода, а в остальных он отсутствует. При этом может оказаться, что ключевые компоненты являются ещё и самыми критическими по производительности участками системы.

- Динамически формируемый код используется как вспомогательное средство. Часто явным признаком такой ситуации является его малое количество или полное отсутствие, но это не всегда так. Возможна ситуация, когда динамически формируемый код активно используется для реализации служебной и вспомогательной функциональности, например сбора статистики, администрирования и диагностики системы. Тот факт, что данная функциональность не является основной для системы, может позволить обращать меньше внимания, например, на производительность результатов трансформации соответствующего кода. Это даёт больше возможностей для использования динамического подхода, так как накладные расходы на дополнительные вычисления во время работы системы не окажут существенного воздействия на производительность основных её функций.

## **5. Анализ требований к производительности и потреблению ресурсов.**

Необходимо определить, позволительны ли дополнительные накладные расходы на обработку встроенного кода при работе реальной системы. Ответ на данный вопрос особенно важен, например, при выборе между статическим и динамическим подходом к трансляции динамически формируемого кода. Выполнение динамически формируемых SQL-запросов само по себе требует дополнительных ресурсов. Если на предыдущих шагах было выяснено, что динамически формируемый код активно используется в критическом по производительности месте, а вычислительные ресурсы ограничены, то применение динамического подхода не применимо. Это

связано с тем, что дополнительные вычисления для обработки кода существенно ухудшат быстродействие критических для системы участков.

## 6. Анализ особенностей конкретного внешнего и встроенного языков.

Многие задачи требуют комплексной обработки внешнего и встроенного кода. Например, при статическом поиске ошибок, включающем ошибки использования типов переменных, необходимо проводить анализ, проверяющий, что тип переменной во внешнем коде соответствует типу, возвращаемому запросом. Пример несовпадения типов приведён в листинге 4.1: переменная `result` имеет тип `List<string>`, однако запрос возвращает коллекцию двухэлементных кортежей.

---

```

1 public List<string> NewReport
2   (int prodId = 0, int status = 0, int nType = 0)
3 {
4     int nProdIdL = prodId;
5
6     string sMagicKey = "[" + prodId.ToString() + "]";
7
8     string tbl = status == 0 ? "InOrders " : "OutOrders ";
9
10    while (nProdIdL > 0)
11    {
12        sMagicKey = "[" + sMagicKey + "]";
13        nProdIdL = nProdIdL - 1;
14    }
15
16    string sExec =
17        "SELECT sOrderDescription, " + sMagicKey
18        + " FROM ts." + tbl;
19
20    List<string> result = db.Execute(sExec);
21    return result;
22 }
```

---

Листинг 4.1: Пример кода метода на языке программирования C#, в котором ожидаемый и реальный тип результата запроса не совпадают: переменная `result` имеет тип `List<string>`, однако запрос возвращает коллекцию двухэлементных кортежей

Наличие некоторых связей внешнего и встроенного кода делает невозможным полностью динамический подход, так как без статического анализа не будет получена информация, необходимая для корректного преобразования внешнего кода. Примером такой ситуации может служить

трансформация кода хранимой процедуры с динамическими запросами, формирующими запрос `SELECT` из T-SQL в PL-SQL. При этом результат данного запроса должен быть результатом процедуры. В T-SQL вне зависимости от типа сформированного запроса его выполнение производится с помощью команды `EXECUTE` и результат выполнения автоматически возвращается в качестве результата процедуры, а в PL-SQL для того, чтобы получить результат выполнения запроса и вернуть его наружу из процедуры необходимо использовать конструкцию `EXECUTE IMMEDIATE ... INTO out_var`<sup>2</sup>, при этом `out_var` должна быть объявлена аргументом процедуры с модификатором `OUT`<sup>3</sup>. То есть в данном случае без статического анализа динамически формируемого кода невозможно корректно преобразовать внешний код.

Отдельно нужно проверить наличие конструкций, которые заведомо усложняют автоматическую обработку системы. Примером такой конструкции может служить оператор `MERGE`<sup>4</sup> в PL-SQL, проекция которого в другие языки может потребовать существенных усилий.

7. **Выбор подхода.** Основной вопрос, который необходимо решить на данном шаге — это выбор между статическим и динамическим подходом или их композицией. На предыдущих шагах могли быть выявлены особенности, накладывающие ограничения на выбор. Может оказаться, что необходим не чистый статический [20] или динамический [55] подходы, а их смесь. Такая ситуация возникает, например, когда решение задачи статически крайне затруднено или невозможно, но динамический подход требует неприемлемых накладных расходов. В данном случае может быть применён смешанный подход. Примером может послужить обработка конструкции `MERGE`, статическая проекция которой сложна, в случае активного использования соответствующего кода. В данном случае на этапе статическо-

---

<sup>2</sup>Языковая конструкция, которая позволяет выполнить запрос и сохранить результат его выполнения в переменной. Описание в документации по языку PL-SQL: [http://docs.oracle.com/cd/B12037\\_01/appdev.101/b10807/13\\_elems017.htm](http://docs.oracle.com/cd/B12037_01/appdev.101/b10807/13_elems017.htm). Посещён 29.07.2015.

<sup>3</sup>Модификатор `OUT` для аргумента подпрограммы означает его передачу по ссылке, что позволяет возвращать его значение в вызывающую подпрограмму. Описание в документации по языку PL-SQL: [http://docs.oracle.com/cd/A97630\\_01/appdev.920/a96624/08\\_subs.htm#752](http://docs.oracle.com/cd/A97630_01/appdev.920/a96624/08_subs.htm#752). Посещён 29.07.2015.

<sup>4</sup>Операция позволяющая обновременно добавлять новые записи и обновлять существующие. Описание в документации по языку PL-SQL: [http://docs.oracle.com/cd/B10500\\_01/appdev.920/a96624/13\\_elems30.htm](http://docs.oracle.com/cd/B10500_01/appdev.920/a96624/13_elems30.htm). Посещён 29.07.2015.

го анализа вычисляется вся информация, необходимая для трансформаций во время выполнения, которая может быть вычислена, и сохраняется, например, в специальных переменных. Далее, во время выполнения происходит окончательная трансформация кода в нужный вид и наличие заранее вычисленной информации сокращает накладные расходы.

**8. Рекомендации по выбору инструмента.** Результатом проделанного анализа должно стать решение о том, какими именно средствами проводить анализ встроенного кода. С одной стороны, необходимо выбрать инструмент. Обзор некоторых из них приведён в разделе 1.5 данной работы. Приведём примеры ситуаций, в которых можно использовать некоторые из них.

- Статический поиск синтаксических ошибок может быть автоматически проведён с помощью таких инструментов как JSA [41] или PHPSA [42] для сложно формируемых выражений. Для детальной диагностики может потребоваться применение инструментов типа Alvor [11].
- Статическая трансформация встроенного кода, каждое выражение которого полностью содержится в константном литерале, может быть автоматически осуществлена SQL Ways [15].
- Статическая трансформация встроенного кода со сложной логикой формирования может быть автоматизирована с помощью инструментов, разработанных на основе платформы, аналогичной представленной в данной работе.

С другой стороны необходимо провести границы автоматизации процесса. Провести чёткое разделение крайне сложно, так как возможны различные ситуации в диапазоне от наличия готового инструмента, решающего необходимые задачи и специалистов, умеющих с ним работать, до отсутствия инструмента и специалистов, способных его создать. В такой ситуации характеристики системы, оцененные на предыдущих шагах, не являются решающим фактором в принятии решения о ручной обработке или автоматизации и её границах, однако являются необходимым условием для при-



нения правильного решения. Кроме того, после изучения существующих инструментов, может быть принято решение о необходимости разработки собственного. В таком случае, проведённый анализ позволит сформулировать требования к нему.

Изложенный метод позволяет принять решение о способе обработки встроеного кода к системам с несколькими встроенными языками, так как их обработка может проводиться независимо. В зависимости от конкретных целей и задач проекта по реинжинирингу, выполнение некоторых описанных шагов может быть совмещено с анализом всей системы.

## Глава 5

# Эксперименты, ограничения, обсуждение

В рамках экспериментального исследования и апробации проводилось, в основном, изучение предложенного в данной работе алгоритма синтаксического анализа, архитектуры реализованного инструментария и предложенного метода, так как это является основными результатами данной работы. Так как многие свойства алгоритма либо формально доказаны в рамках работы, либо непосредственно следуют из доказанных свойств и описания (например, завершаемость и корректность на определённых классах входных данных), то интерес при экспериментальном исследовании представляет сравнение с инструментами, реализующими аналогичный алгоритм и оценка производительности.

### 5.1 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.

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

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 лексического анализа регулярной аппроксимации, построенной по некоторой программе. В данном случае базовый блок — это шаблонный конечный автомат, который используется для построения тестовых конечных автоматов. Каждый пакет тестов характеризовался 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 5.1, если истина — то изображённые на рисунке 5.2).

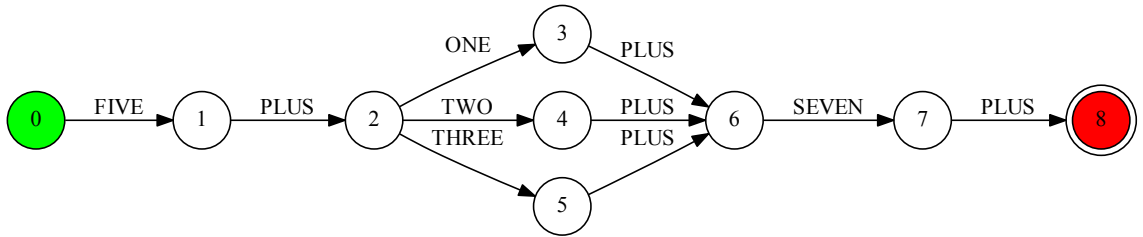


Рисунок 5.1: Базовый блок без циклов при  $height = 3$

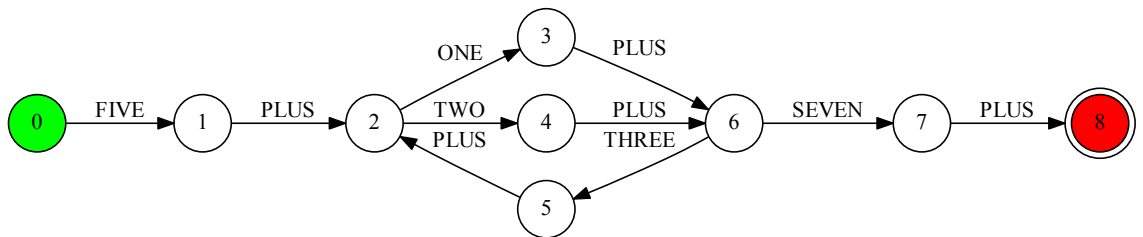


Рисунок 5.2: Базовый блок, содержащий цикл, при  $height = 3$

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 5.3, 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 5.4 показывает, что наличие циклов в графе, при одинаковом значении параметра *height*, увеличивает продолжительность анализа, при этом зависимость времени от размера графа остаётся линейной.

## 5.2 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.

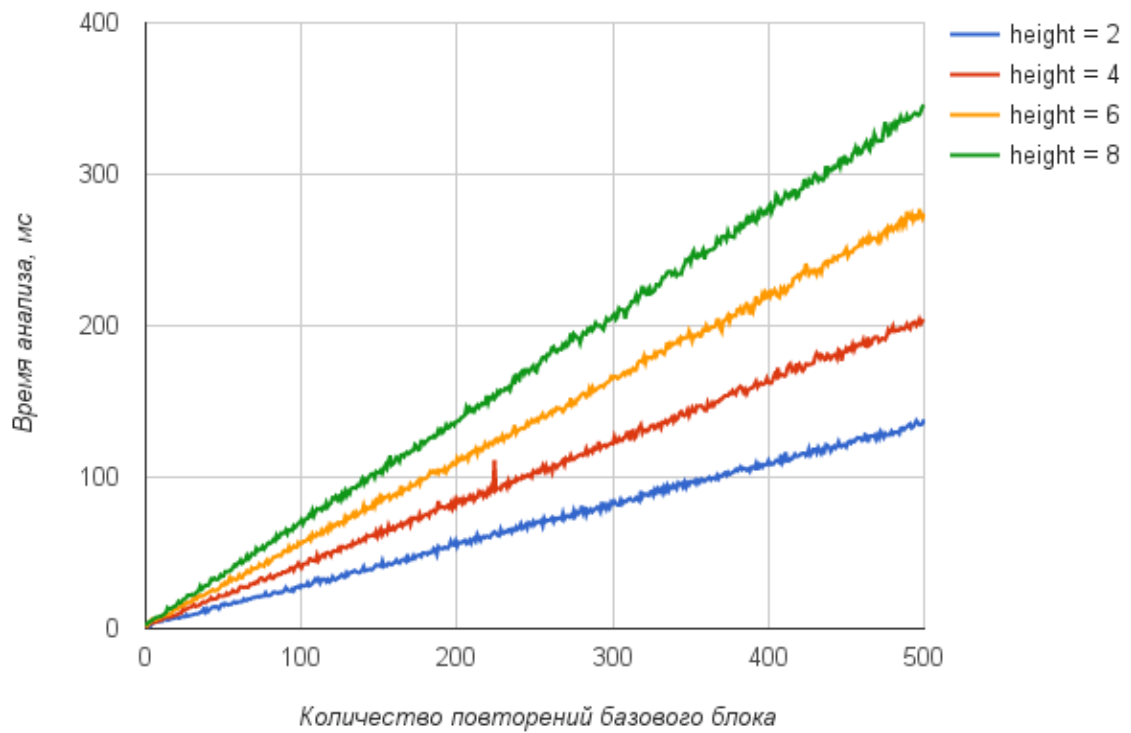


Рисунок 5.3: Зависимость времени работы алгоритма от размера входного графа при  $isCycle = false$

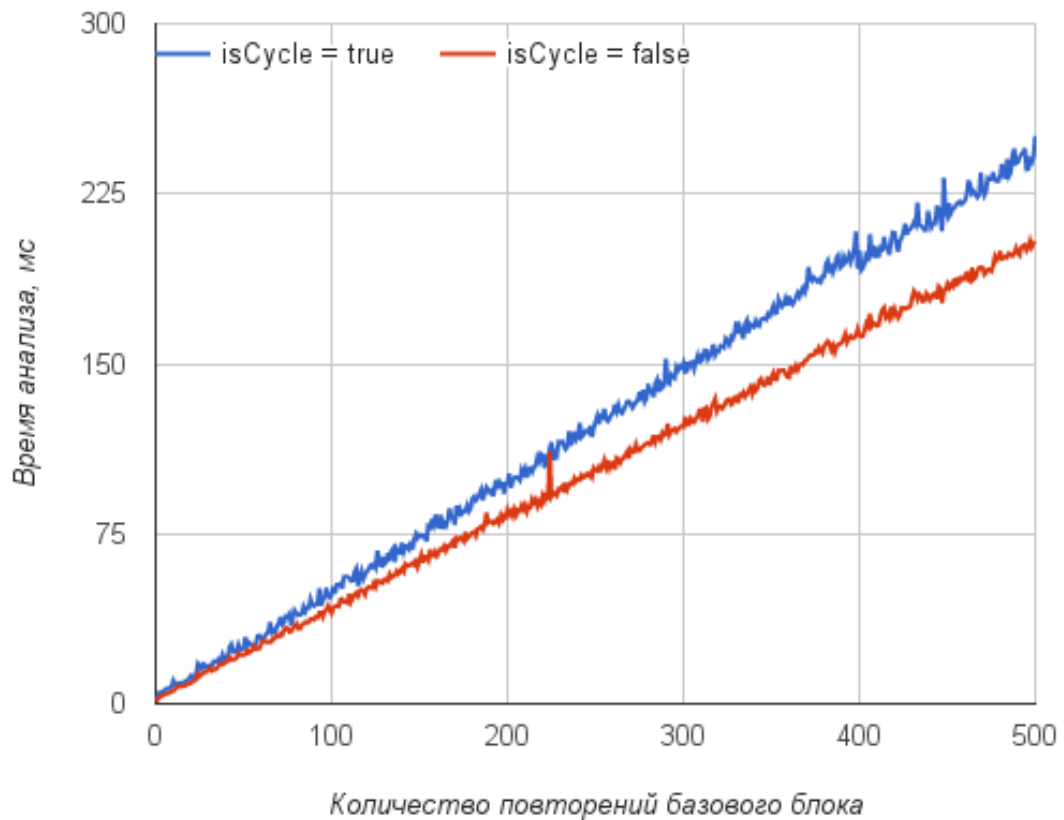


Рисунок 5.4: Зависимость времени работы алгоритма от размера входного графа и наличия в нем циклов при  $height = 4$

При этом среднее количество операторов для формирования запроса равнялось 40 [20].

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 5.2

Тесты проводились на вычислительном устройстве с параметрами, эквивалентными указанным в разделе 5.1. 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.

Результаты измерений времени работы представлены в таблице 5.1. 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

Таблица 5.1: Распределение динамически формируемых SQL-запросов по времени обработки

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

конструкций, не поддерживаемых 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 5.5.

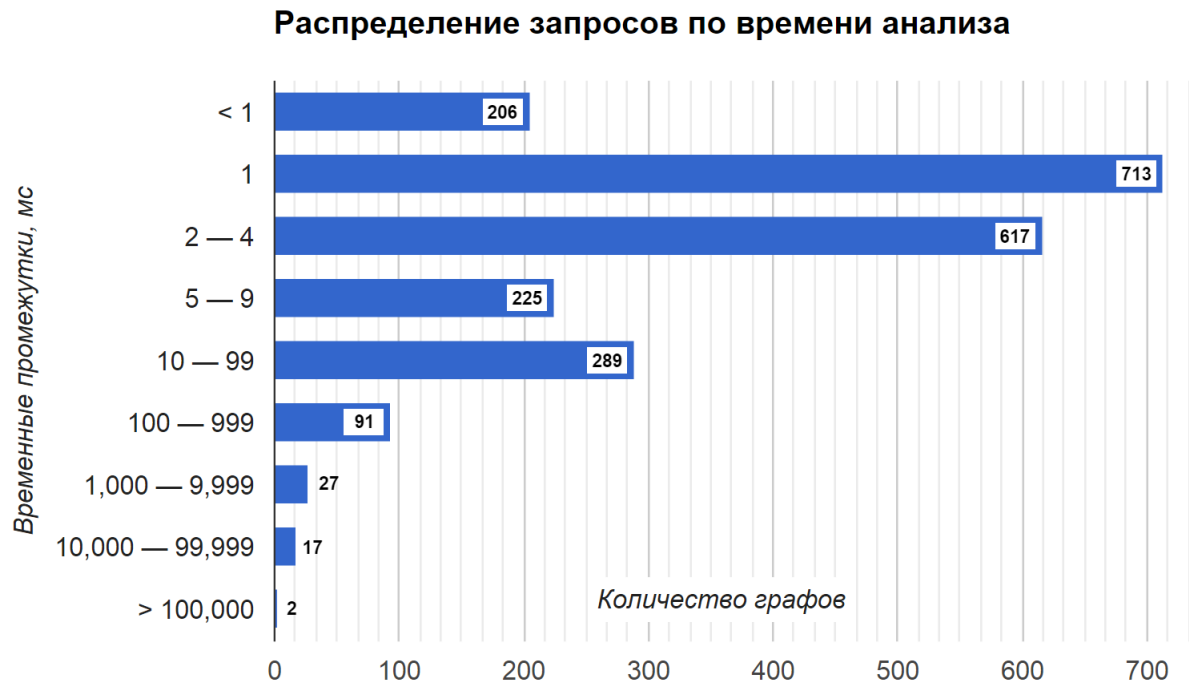


Рисунок 5.5: Распределение запросов по времени анализа

Таблица 5.2: 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

Тестирование на реальных данных показало, что предложенный в работе алгоритм синтаксического анализа применим для синтаксического анализа регулярной аппроксимации множества значений динамически формируемых выра-



жений, 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”.

### 5.3 Сравнение с инструментом Alvor

Единственным доступным инструментом, производящим синтаксический анализ динамически формируемого кода, является Alvor [9, 10]. 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 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 5.1.

Так как Alvor на вход принимает регулярное выражение, называемое *абстрактной строкой* [10], а анализатор, созданный на основе YC.SEL.SDK — конечный автомат, то был реализован генератор, который по входным параметрам создает файл с абстрактной строкой и с описанием соответствующего автомата в формате DOT. Синтаксис описания абстрактной строки приведён в листинге 5.1. 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.

---

```

1 absStr = "str"
2     | '{'absStr(',' absStr)+'}' //alternatives
3     | absStr absStr           //concatenation
4     | absStr '*'              //closure

```

---

Листинг 5.1: Синтаксис описания абстрактной строки

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 5.6 и листинге 5.2 соответственно.

---

```

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

```

---

Листинг 5.2: Пример абстрактных строк для  $height = 2$  одного и двух повторений базового блока

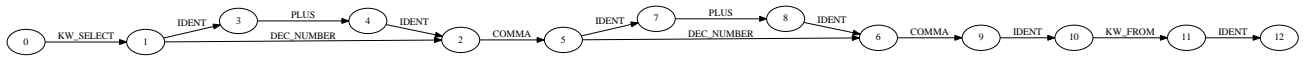


Рисунок 5.6: Входной граф для синтаксического анализатора на базе YC.SEL.SDK при  $height = 2$  и двух повторениях базового блока

Результаты измерений представлены в таблице 5.3 и на графике 5.6. В легенде 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.

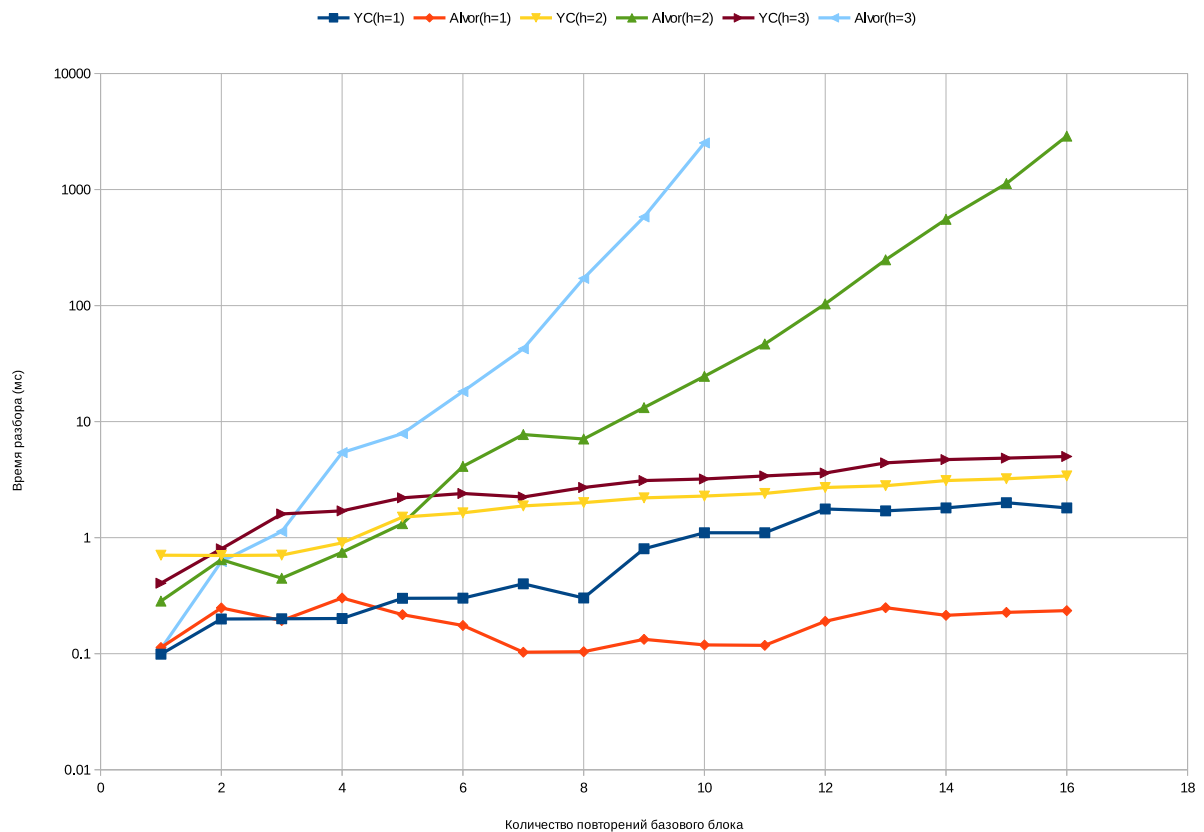


Рисунок 5.7: Сравнение производительности Alvor и синтаксического анализатора на базе YC.SEL.SDK

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 отличие от алгоритма, реализованного в данной работе.

## 5.4 Разработка расширений для поддержки встроенных языков

На основе YC.SEL.SDK и YC.SEL.SDK.ReSharper, которые были представлены ранее, в рамках апробации были реализованы расширения к ReSharper, которые предоставляют поддержку для двух встроенных языков: подмножества T-SQL и Calc. Реализация данных расширений также являлась апробацией предложенной архитектуры разработанного инструментального пакета.

Таблица 5.3: 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	

В рамках данного шага апробации были созданы лексические спецификации и грамматики соответствующих языков. Исходный код описаний опубликован в открытом доступе: <https://github.com/YaccConstructor/YC.GrammarZOO>. Далее, с помощью генераторов из разработанного инструментария, по этим грамматикам построены синтаксические анализаторы, а по лексическим спецификациям — лексические анализаторы.

При создании расширений был также апробирован механизм построения регулярной аппроксимации. Для построения графа потока управления внешнего ввода использовалась функциональность ReSharper SDK. Затем полученный граф переводился в обобщённое представление, по которому строилась регулярная аппроксимация средствами разработанного инструмента.

После того, как отдельные части были готовы, они были объединены в готовый плагин на основе YC.SEL.SDK.ReSharper. В результате было получено два расширения, предоставляющие поддержку соответствующих языков и ядро, содержащее общую, независимую от языков, функциональность, связанную прежде всего с обеспечением взаимодействия между ReSharper и реализованными расширениями.

Расширения предоставляют следующую функциональность: подсветка синтаксиса (рисунок 5.8), подсветка парных (рисунок 5.9) элементов. Для языка Calc также реализована статическая диагностика семантических ошибок, а именно поиск использования необъявленных переменных, что показывает, с одной стороны, возможность непосредственного использования SPPF для проведения анализа динамически формируемого кода, а с другой — возможность реализации дополнительной функциональности, не являющейся общей функциональностью SDK.

Статический поиск использования необъявленных переменных показан на рисунке 5.10. В данном примере переменная *x* объявляется в одной из веток условного оператора и не объявляется в другой, что может привести к ошибке в точке использования, о чём и сообщено пользователю.

```

33 public static void Execute()
34 {
35     Program.Eval("(1 + 2) * 3");
36     Program.ExecuteImmediate("insert into y (x,v) values (1,2)");
37 }
```

Рисунок 5.8: Пример подсветки синтаксиса для нескольких встроенных языков: SQL и Calc

<pre> 26 public static void Insert(bool cond) 27 { 28     var insertQuery = "insert into y(x"; 29     if (cond) 30         insertQuery += ", v"; 31     else 32         insertQuery = insertQuery + ", u"; 33     insertQuery += " values (1, 2)"; 34     Program.ExecuteImmediate(insertQuery); 35 }</pre>	<pre> 26 public static void Insert(bool cond) 27 { 28     var insertQuery = "insert into y(x"; 29     if (cond) 30         insertQuery += ", v"; 31     else 32         insertQuery = insertQuery + ", u"; 33     insertQuery += " values (1, 2)"; 34     Program.ExecuteImmediate(insertQuery); 35 }</pre>
<p>(a) Одной открывающей скобке соответствует несколько закрывающих</p>	<p>(b) Одной закрывающей скобке соответствует одна открывающая</p>

Рисунок 5.9: Пример подсветки парных скобок

```

17 public static void Execute(bool cond)
18 {
19     string query = "varX = 1;";
20     if (cond)
21         query += "varY = 2;";
22     query += "varZ = varX + varY;";
23     Program.ExtEval(query);
24 }

```

Рисунок 5.10: Пример статического обнаружения семантических ошибок для языка Calc

```

43 static int Calculate(bool cond)
44 {
45     Program.ExecuteImmediate("insert into y (x,v) values (1,2)");
46     var expr = "(10";
47     for (int i = 0; i < 10; ++i)
48     {
49         expr += " + 1";
50         if (cond)
51             expr += TrueCaseStr();
52     }
53     return Program.Eval(expr + ") /2");
54 }
55
56 static string TrueCaseStr()
57 {
58     return "*3";
59 }

```

Рисунок 5.11: Пример межпроцедурной обработки встроенных языков

Расширения опубликованы в виде готовых к использованию бинарных пакетов. Функциональность, отвечающая за поддержку каждого языка, распространяется в виде самостоятельного бинарного пакета и может быть независимо подключена или отключена. Структура пакетов представлена на рисунке 5.12.

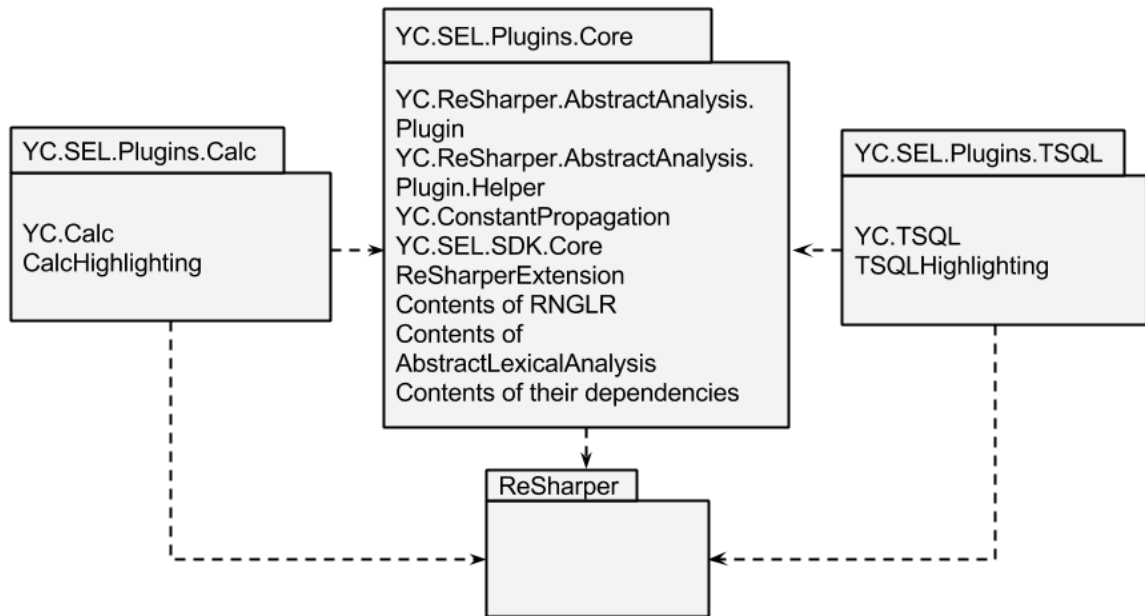


Рисунок 5.12: Структура пакетов расширений для ReSharper, предоставляющих поддержку встроенных T-SQL и Calc

Дополнительно в результате разработки расширений было подтверждено, что анализаторы языков могут быть использованы независимо. Например, анализаторы, разработанные в рамках расширений для ReSharper, используются в тестах не связанных с ReSharper. Это показывает, что независимость шагов обработки динамически формируемых выражений позволяет гибко переиспользовать компоненты, реализующие эти шаги, что является плюсом реализованного решения.

## 5.5 Ограничения

Используемые подходы и алгоритмы накладывают ограничения на платформу и разрабатываемые на её основе инструменты. Обсуждению этих ограничений посвящён данный раздел.

Множество, являющееся аппроксимацией множества значений динамически формируемого выражения, принимаемое на вход алгоритмом синтаксического анализа должно быть регулярным множеством. То есть аппроксимация задаёт регулярный язык, в то время как генерируемый программой может быть рекурсивно-перечислимым. Это означает, что на этапе построения аппроксимации будет происходить потеря точности. Например, нехвостовая рекурсия точно не выразима в терминах регулярных множеств. Точность построения кон-

кретной регулярной аппроксимации зависит от конкретного алгоритма, используемого для этого. алгоритм может реализовывать или не реализовывать межпроцедурный анализ, поддерживать или не поддерживать строковые операции, разными способами обрабатывать пользовательский ввод и другие ситуации, когда значение выражения вычислить невозможно. В рамках рассматриваемой платформы реализован алгоритм позволяющий поддерживать межпроцедурный анализ и строковые операции.

Эталонный язык должен быть описан детерминированной контекстно-свободной грамматикой. Большинство языков программирования могут быть описаны такой грамматикой. Однако, с одной стороны, бывают исключения, например C++. С другой стороны, в документации языка программирования может быть приведена недетеминированная грамматика, а её приведение к детерминированной может потребовать значительных усилий.

Вопросы быстродействия инструментов, созданных на основе зависят от контекста их использования. Если при реинжиниринге допустимо проведение длительных анализов, то для многих инструментов, используемых в средах разработки, время отклика критично. Особенно это важно для функциональности, работающей в режиме “на лету”: подсветка синтаксиса, автодополнение, подсказки. Так как платформа создавалась с ориентацией на создание инструментов для реинжиниринга, то некоторые компоненты направлены на увеличение точности анализа, возможно, в ущерб производительности. Примером такой компоненты может служить компонента построения регулярной аппроксимации, которая реализовывает алгоритм, который гарантирует построение приближения сверху, учитывает циклы и строковые функции [54]. Это повышает точность анализа, однако производительность может оказаться слишком низкой для использования в IDE. С другой стороны, при создании инструмента для IDE возможно заменить построение аппроксимации на более легковесное, но менее точное. Например в инструменте Alvor [10], предназначенном прежде всего для интерактивной работы в среде разработки, предлагается такой алгоритм. При этом важно, что возможности платформы позволяют комбинировать различные реализации компонент, так как они независимы. То есть можно использовать один и тот же синтаксический анализатор с разными вариантами лексического для получения требуемых характеристик результирующего инструмента. С другой



стороны, текущая реализация содержит возможности для различного рода оптимизаций: некоторые алгоритмы могут быть ускорены с помощью распараллеливания, выбор оптимальных структур данных, например для конечных автоматов, активно использующихся в рамках платформы, является темой отдельного исследования [56].

Систематическое исследование работы с SPPF находится на начальной стадии даже в контексте обобщённого синтаксического анализа [57]. В рамках же анализа динамически формируемых выражений исследований в этом направлении не обнаружено. По этой причине в рамках данной работы реализован только прототип библиотеки, позволяющей решать некоторые задачи, например, поиск необъявленных переменных, над SPPF в общем виде. Теоретическое исследование данного вопроса является отдельной задачей. С другой стороны, было доказано, что из построенного SPPF извлекаемы деревья вывода для любых цепочек из аппроксимации, а с деревьями можно работать с помощью стандартных методов.

## Глава 6

# Сравнение и соотнесение

В данной главе представлено сравнение разработанного решения с основными существующими решениями в области анализа динамически формируемых строковых выражений. Описание существующих решений представлено в разделе 1.5 данной работы, поэтому далее приводится только сравнение.

В качестве инструментов, с которыми производилось сравнение выбраны следующие: Alvor, JSA, PHPSA, IntelliLang, Varis. Так же проводилось сравнение с инструментом, названным нами условно AbsPars, реализованным авторами работ по абстрактному синтаксическому анализу [6–8]. Несмотря на то что в свободном доступе реализации алгоритма, изложенного в указанных статьях не обнаружено, самими авторами приводятся достаточно подробные результаты апробации реализации алгоритма, что позволяет сделать некоторые выводы о его основных возможностях. Кроме того, стоит отметить, что Varis является очень молодым инструментом: впервые он был представлен в 2015 году на конференции ICSE<sup>1</sup>. По этой причине детальная оценка его возможностей затруднена.

Для сравнения инструментов были выбраны критерии, представленные в таблице 6.1. Критерии отражают основные функциональные возможности инструментов, важные для практического решения задач в различных областях. Сравнение по более детальным критериям не является осмысленным, так как оно становится связанным с особенностями решаемых задач, реализацией, используемыми подходами и часто является их следствием, а не свойством самого инструмента.

---

<sup>1</sup>Международная конференция по разработке программного обеспечения (International Conference on Software Engineering). Сайт конференции: <http://2015.icse-conferences.org/>, (посещён 29.07.2015)

Таблица 6.1: Критерии сравнения инструментов анализа динамически формируемых строковых выражений

Критерий	Название колонки в таблице с результатами сравнения 6.2	Описание
Платформа для разработки	Платформа	Предоставляется ли в явном виде платформа для создания новых инструментов статического анализа динамически формируемых выражений.
Построение леса разбора	Лес разбора	Реализована ли в инструменте функциональность по построению леса разбора динамически формируемого кода.
Поиск синтаксических ошибок	Синт. ошибки	Обнаруживает ли инструмент синтаксические ошибки в динамически формируемом коде.
Поиск семантических ошибок	Сем. ошибки	Обнаруживает ли инструмент семантические ошибки в динамически формируемом коде.
Подсветка синтаксиса	Подсветка	Обеспечивает ли инструмент подсветку синтаксиса динамически формируемого кода.
Модульность обработки	Модульность	Выделены ли отдельные независимые шаги обработки или же анализ является монолитным. Реализованы ли соответствующие выделенным шагам независимые компоненты.

В таблице 6.2 приведены основные результаты сравнения инструментов статического анализа динамически формируемых строковых выражений на основе критериев, представленных ранее. Используются следующие обозначения.

- ‘+’ — функциональность, соответствующая критерию, полностью реализована.
- ‘–’ — функциональность, соответствующая критерию, полностью не реализована.
- ‘+–’ — соответствующая функциональность реализована частично.

Таблица 6.2: Сравнение инструментов анализа динамически формируемых строковых выражений

Инструмент	Платформа	Лес разбора	Синт. ошибки	Сем. ошибки	Подсветка	Модульность
AbsPars	—	+— <sup>a</sup>	+	+	—	—
Alvor	—	—	+	—	—	+
JSA	—	—	+	—	—	—
PHPSA	—	—	+	—	—	—
IntelliLang	+— <sup>b</sup>	—	+	+	+	+
Varis	—	+ <sup>c</sup>	+	—	+	—
YC.SEL.SDK	+	+	— <sup>d</sup>	+	+	+

<sup>a</sup> В работе [8] утверждается, что поддерживаются атрибутные грамматики для описания языка. Это даёт возможность описать семантику построения леса. При этом, в работе обсуждаются проблемы с падением точности анализа при использовании семантики, а построение леса не обсуждается. Возможность построения корректного конечного представления леса разбора требует отдельного исследования.

<sup>b</sup> IntelliLang использует возможности платформы IntelliJ IDEA (<https://www.jetbrains.com/idea/>, посещён 29.07.2015), по этому расширение некоторых возможностей унифицировано, однако самостоятельной платформы для анализа встроенных языков не предоставляется.

<sup>c</sup> Такие свойства конструируемой структуры данных, как конечность, а так же возможности её дальнейшей обработки в общем виде не обсуждаются.

<sup>d</sup> Возможность диагностики синтаксических ошибок не реализована на данный момент в рамках платформы. Однако данная задача может быть решена другими инструментами, так как обработка производится по шагам и возможно добавить новый без существенных затрат.

Проведённое сравнение позволяет выявить несколько аспектов.

- Многие инструменты могут быть расширены, однако ранее не предоставлялось полноценного самостоятельного инструментария специально предназначенного для создания новых инструментов для обработки динамически формируемых выражений.
- Основная цель большинства инструментов — это проверка корректности динамически формируемого кода. Прежде всего осуществляется поиск синтаксических ошибок.
- Инструменты реализованы на основе разных подходов и предназначены для решения разных задач, поэтому детальное сравнение их возможностей, производительности и других аспектов не представляется оправданным. Например, инструмент PHPSA изначально создавался для решения задачи проверки корректности динамически формируемых выражений и отсутствие подсветки синтаксиса не является его недостатком.
- Существование различных инструментов для решения различных задач и создание новых говорит о том, что обработка встроенных языков является актуальной задачей.

В результате можно утверждать, что YC.SEL.SDK является единственной полноценной платформой для создания различных инструментов статического анализа динамически формируемых выражений, применимых в разных областях и обладающих широкими функциональными возможностями.

# Заключение

В ходе выполнения исследования получены следующие результаты.

- Разработан алгоритм синтаксического анализа динамически формируемых выражений, позволяющий обрабатывать произвольную регулярную аппроксимацию множества значений выражения в точке выполнения, реализующий эффективное управление стеком и гарантирующий конечность представления леса вывода. Доказана завершаемость и корректность предложенного алгоритма при анализе регулярной аппроксимации, представимой в виде произвольного конечного автомата без эпсилон-переходов.
- Создана архитектура инструментария для разработки программных средств синтаксического анализа динамически формируемых строковых выражений.
- Реализован инструментальный пакет для разработки средств статического анализа динамически формируемых выражений. На его основе реализован плагин для ReSharper. Код опубликован на сервисе GitHub:  
<https://github.com/YaccConstructor/YaccConstructor> под лицензией Apache License Version 2.0, автор работал под учётной записью с именем gsvgit
- Разработана методика анализа динамически формируемых строковых выражений в проектах по реинжинирингу информационных систем. Данная методика применена в проекте компании ЗАО “Ланит-Терком” по переносу информационной системы с MS-SQL Server на Oracle Server, для чего реализованы соответствующие программные компоненты.

Кроме того, планируется развитие платформы и плагина. На уровне платформы необходимо реализовать механизмы, требующиеся для трансформаций кода

на встроенных языках. Механизмы трансформации встроенных языков требуются для проведения миграции с одной СУБД на другую [20] или для миграции на новые технологии, например, LINQ. Эта задача связана с двумя проблемами: возможностью проведения нетривиальных трансформаций и доказательство корректности трансформаций. Планируется реализация проверки корректности типов. Для SQL это должна быть как проверка типов внутри запроса, так и проверка того, что тип возвращаемого запросом результата соответствует типу хост-переменной, выделенной для сохранения результата в основном коде.

Более общей проблемой, подлежащей дальнейшему исследованию, является возможность выполнения семантических действий непосредственно над SPPF. Это необходимо для рефакторинга, улучшения качества трансляции, автоматизации перехода на более надёжные средства метапрограммирования [58, 59].

Важной задачей является теоретическая оценка сложности предложенного алгоритма синтаксического анализа. В известных работах не приводятся строгих оценок подобных алгоритмов, поэтому данная задача является самостоятельным исследованием.

С целью обобщения предложенного подхода к синтаксическому анализу, а также для получения лучшей производительности и возможностей для более качественной диагностики ошибок, планируется переход на алгоритм обобщённого LL-анализа (GLL) [22, 60]. Планируется исследовать возможность улучшения предложенного алгоритма при переходе на другие алгоритмы обобщённого LR-анализа [47], например, такие как BRNGLR [61] и RIGLR [62].

Кроме того, важной задачей является реализация диагностики ошибок, решение которой для обобщённого восходящего анализа активно исследуется [63–66]. Адаптация предложенных решений для применения в представленном алгоритме требует отдельной работы.

# Литература

1. *Терехов А.Н., Терехов А.А.* Автоматизированный реинжиниринг программ. — СПб: Издательство С.-Петербургского университета, 2000.
2. *9075:1992 ISO. ISO/IEC.* Information technology — Database languages — SQL. — 1992.
3. *Houglund Damon, Tavistock Aaron.* Core JSP. — Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.
- 4.
5. *Cleve Anthony, Mens Tom, Hainaut Jean-Luc.* Data-Intensive System Evolution // *IEEE Computer.* — 2010. — Vol. 43, no. 8. — Pp. 110–112.
6. *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. — Pp. 256–272.
7. *Doh Kyung-Goo, Kim Hyunha, Schmidt David A.* Formal Modeling / Ed. by Gul Agha, José Meseguer, Olivier Danvy. — Berlin, Heidelberg: Springer-Verlag, 2011. — Pp. 90–109.
8. *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.* — Pp. 194–214.
9. 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. — Pp. 131–138.

10. *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. — Pp. 20–22.
- 11.
- 12.
13. *Christensen Aske Simon, Møller Anders, Schwartzbach Michael I*. Precise Analysis of String Expressions // Proc. 10th International Static Analysis Symposium (SAS). — Vol. 2694 of *LNCS*. — Springer-Verlag, 2003. — June. — Pp. 1–18. — Available from <http://www.brics.dk/JSA/>.
14. *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. — Pp. 432–441.
- 15.
16. *Fu Xiang, Qian Kai*. SAFELI: SQL Injection Scanner Using Symbolic Execution // Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications. — TAV-WEB '08. — New York, NY, USA: ACM, 2008. — Pp. 34–39.
17. *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. — Pp. 94–98.
18. *Scott Elizabeth, Johnstone Adrian*. Right Nulled GLR Parsers // *ACM Trans. Program. Lang. Syst.* — 2006. — Vol. 28, no. 4. — Pp. 577–618.
19. *Rekers Jan*. Parser Generation for Interactive Environments. — 1992.

20. *Grigorev Semen, Kirilenko Iakov*. From Abstract Parsing to Abstract Translation // Preliminary Proceedings of the 8th Spring/Summer Young Researchers Colloquium on Software Engineering. — 2013. — Pp. 135–139.
21. *Kirilenko Iakov, Grigorev Semen, Avdiukhin Dmitriy*. Инструментальная поддержка встроенных языков в интегрированных средах разработки // *Моделирование и анализ информационных систем*. — 2014. — Vol. 21, no. 6. — Pp. 131–143.
22. *Grigorev Semen, Ragozina Anastasiya*. Generalized Table-Based LL-Parsing // *Systems and Means of Informatics*. — 2014. — Vol. 25, no. 1. — Pp. 89–107.
23. 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. — Pp. 21:1–21:11.
24. *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. — Pp. 5:1–5:9.
25. *Yu Fang, Cova Marco*. String Analysis.
26. *Chomsky Noam*. Three models for the description of language // *IRE Transactions on Information Theory*. — 1956. — Pp. 113–124.
27. *Pitts Andrew M*. Lecture Notes on Regular Languages and Finite Automata for Part IA of the Computer Science Tripos. — 2010.
28. *Hanneforth Thomas*. Finite-state Machines: Theory and Applications Unweighted Finite-state Automata / Institut fur Linguistik Universitat Potsdam. — 2010.
29. *Mohri Mehryar*. Finite-state Transducers in Language and Speech Processing // *Comput. Linguist.* — 1997. — Vol. 23, no. 2. — Pp. 269–311.
30. *Smith Zachary*. Development of Tools to Manage Embedded SQL // Proceedings of the 49th Annual Southeast Regional Conference. — ACM-SE '11. — New York, NY, USA: ACM, 2011. — Pp. 358–359.

31. *Fenton Norman, Pfleeger Shari Lawrence*. Software Metrics (2Nd Ed.): A Rigorous and Practical Approach. — Boston, MA, USA: PWS Publishing Co., 1997.
32. *Brink Huib van den, Leek Rob van der, Visser Joost*. Quality Assessment for Embedded SQL // Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation. — SCAM '07. — Washington, DC, USA: IEEE Computer Society, 2007. — Pp. 163–170.
33. *van den Brink Huib J.* A Framework to Distil SQL Queries Out of Host Languages in Order to Apply Quality Metrics. — 2007.
34. *Cleve Anthony, Meurisse Jean-Roch, Hainaut Jean-Luc*. Journal on Data Semantics XV / Ed. by Stefano Spaccapietra. — Berlin, Heidelberg: Springer-Verlag, 2011. — Pp. 130–157.
35. *Cleve Anthony, Hainaut Jean-Luc*. Dynamic Analysis of SQL Statements for Data-Intensive Applications Reverse Engineering // Proceedings of the 2008 15th Working Conference on Reverse Engineering. — WCRE '08. — Washington, DC, USA: IEEE Computer Society, 2008. — Pp. 192–196.
36. A Static Analysis Framework For Detecting SQL Injection Vulnerabilities / Xiang Fu, Xin Lu, Boris Peltzberger et al. // Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01. — COMPSAC '07. — Washington, DC, USA: IEEE Computer Society, 2007. — Pp. 87–96.
37. *Asveld Peter R. J., Nijholt Anton*. The Inclusion Problem for Some Subclasses of Context-free Languages // *Theor. Comput. Sci.* — 1999. — Vol. 230, no. 1-2. — Pp. 247–256.
38. *Aho Alfred V., Sethi Ravi, Ullman Jeffrey D.* Compilers: Principles, Techniques, and Tools. — Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.
- 39.
- 40.

- 41.
- 42.
- 43.
- 44.
45. *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.
46. *Cousot Patrick, Cousot Radhia.* Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints // Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. — POPL '77. — New York, NY, USA: ACM, 1977. — Pp. 238–252.
47. *Economopoulos Giorgios Robert.* Generalised LR parsing algorithms. — 2006.
48. *Grune Dick, Jacobs Criel J. H.* Parsing Techniques: A Practical Guide. — Upper Saddle River, NJ, USA: Ellis Horwood, 1990.
49. *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. — Pp. 756–764.
- 50.
- 51.
52. *Syme Don, Granicz Adam, Cisternino Antonio.* Expert F# (Expert's Voice in .Net).
- 53.
54. Automata-based Symbolic String Analysis for Vulnerability Detection / Fang Yu, Muath Alkhalaf, Tefvik Bultan, Oscar H. Ibarra // *Form. Methods Syst. Des.* — 2014. — Vol. 44, no. 1. — Pp. 44–70.

55. *Шанов М.Д, Попов Э.В.* Реинжиниринг баз данных // *Открытые системы.* — 2004. — no. 4. — Pp. 110–112.
56. *Hooimeijer Pieter, Veanes Margus.* An Evaluation of Automata Algorithms for String Analysis // *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation.* — VMCAI'11. — Berlin, Heidelberg: Springer-Verlag, 2011. — Pp. 248–262.
57. *Zaytsev Vadim.* Coupled Transformations of Shared Packed Parse Forests // *Sixth International Workshop on Graph Computation Models (GCM)* / Ed. by Detlef Plump. — Vol. 1403 of *CEUR Workshop Proceedings.* — CEUR-WS.org, 2015. — Pp. 2–17.
58. *Lester Martin Mariusz.* Position Paper: The Science of Boxing // *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security.* — PLAS '13. — New York, NY, USA: ACM, 2013. — Pp. 83–88.
59. *Lester Martin, Ong Luke, Schäfer Max.* Information Flow Analysis for a Dynamically Typed Language with Staged Metaprogramming // *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26–28, 2013.* — 2013. — Pp. 209–223.
60. *Scott Elizabeth, Johnstone Adrian.* GLL Parsing // *Electron. Notes Theor. Comput. Sci.* — 2010. — Vol. 253, no. 7. — Pp. 177–189.
61. *Scott Elizabeth, Johnstone Adrian, Economopoulos Rob.* BRNGLR: A Cubic Tomita-style GLR Parsing Algorithm // *Acta Inf.* — 2007. — Vol. 44, no. 6. — Pp. 427–461.
62. *Scott Elizabeth, Johnstone Adrian.* Generalized Bottom Up Parsers With Reduced Stack Activity // *Comput. J.* — 2005. — Vol. 48, no. 5. — Pp. 565–587.
63. *Valkering Ron.* Syntax error handling in scannerless generalized LR parsers // *Group, University of Amsterdam.* — 2007.
64. Natural and Flexible Error Recovery for Generated Modular Language Environments / Maartje de Jonge, Lennart C. L. Kats, Eelco Visser, Emma Söderberg // *ACM Trans. Program. Lang. Syst.* — 2012. — Vol. 34, no. 4. — Pp. 15:1–15:50.

65. Providing Rapid Feedback in Generated Modular Language Environments: Adding Error Recovery to Scannerless generalized-LR Parsing / Lennart C.L. Kats, Maartje de Jonge, Emma Nilsson-Nyman, Eelco Visser // *SIGPLAN Not.* — 2009. — Vol. 44, no. 10. — Pp. 445–464.
66. *Malone S., Felshin S.* GLR Parsing for Erroneous Input // Generalized LR Parsing / Ed. by M. Tomita. — Boston: Kluwer, 1991. — Pp. 129–140.

# Список рисунков

1.1	Пример GSS . . . . .	32
1.2	Пример SPPF для грамматики $G_1$ и входа <b>ABC</b> . . . . .	34
1.3	Архитектура платформы YaccConstructor . . . . .	38
2.1	Конечный автомат, задающий регулярную аппроксимацию выражения <b>expr</b> . . . . .	52
2.2	Конечное представление леса разбора для выражения <b>expr</b> . . . . .	53
2.3	Дерево вывода для выражения $expr = "()"$ . . . . .	53
2.4	Дерево вывода для выражения $expr = "()()"$ . . . . .	54
2.5	Дерево вывода для выражения $expr = "()()()"$ . . . . .	54
3.1	Диаграмма последовательности обработки встроенных языков . . . . .	60
3.2	Архитектура SDK целиком . . . . .	60
3.3	Архитектура лексического анализатора . . . . .	62
3.4	Архитектура синтаксического анализатора . . . . .	63
3.5	Один из возможных вариантов использования SDK в проектах по реинжинирингу . . . . .	73
4.1	Основные шаги метода обработки встроенных языков и их результаты . . . . .	77
5.1	Базовый блок без циклов при $height = 3$ . . . . .	91
5.2	Базовый блок, содержащий цикл, при $height = 3$ . . . . .	91
5.3	Зависимость времени работы алгоритма от размера входного графа при $isCycle = false$ . . . . .	93
5.4	Зависимость времени работы алгоритма от размера входного графа и наличия в нем циклов при $height = 4$ . . . . .	93
5.5	Распределение запросов по времени анализа . . . . .	96

5.6	Входной граф для синтаксического анализатора на базе YC.SEL.SDK при $height = 2$ и двух повторениях базового блока . . . . .	98
5.7	Сравнение производительности Alvor и синтаксического анализатора на базе YC.SEL.SDK . . . . .	99
5.8	Пример подсветки синтаксиса для нескольких встроенных языков: SQL и Calc . . . . .	101
5.9	Пример подсветки парных скобок . . . . .	101
5.10	Пример статического обнаружения семантических ошибок для языка Calc . . . . .	102
5.11	Пример межпроцедурной обработки встроенных языков . . . . .	102
5.12	Структура пакетов расширений для ReSharper, предоставляющих поддержку встроенных T-SQL и Calc . . . . .	103



# Список таблиц

4.1	Основные шаги по подготовке к реинжинирингу системы, содержащей строковые выражения . . . . .	80
5.1	Распределение динамически формируемых SQL-запросов по времени обработки . . . . .	95
5.2	Пример отчёта по результатам запуска синтаксического анализа на реальной системе . . . . .	96
5.3	Результаты сравнения производительности Alvor и синтаксического анализатора на базе YC.SEL.SDK . . . . .	100
6.1	Критерии сравнения инструментов анализа динамически формируемых строковых выражений . . . . .	107
6.2	Сравнение инструментов анализа динамически формируемых строковых выражений . . . . .	108