Модел. и анализ информ. систем. Т. **0**, № **0** (2000) 5–16

УДК 517.9

Инструментальная поддержка встроенных языков в интегрированных средах разработки

Григорьев С. В.[1]

Ярославский государственный университет им. П. Г. Демидова
150000 Россия, г. Ярославль, ул. Советская, 14

e-mail: ivanov@mail

получена 20 декабря 2012

Ключевые слова: аттрактор, бифуркация

Часто при разработке сложных программных систем используется более, чем один язык программирования. В таком случае принято говорить об основном (или исходном) языке и одном или нескольких встроенных языках. Из строковых выражений основного языка динамически формируются программы на отличном от него языке, которые потом интерпретируются специальными, работающими во время исполнения компонентами, такими как базы данных или веб-браузеры. Большинство языков программирования общего назначения могут играть роль как основного, так и встроенного языка. Самый яркий пример реализации такого подхода — динамический SQL, специфицированный в стандарте ISO SQL и поддерживаемый большинством СУБД.

Автодополнение и подсветка синтаксиса — стандартная для интерактивных сред разработки функциональность — значительно упрощает процесс разработки с использованием встроенных языков. Существует несколько инструментов, предоставляющих функциональность интегрированных сред разработки для встроенных языков, но они в основном поддерживают только одни конкретный встроенный язык и поддержка другого языка требует нетривиального ручного вмешательства. Мы разрабатываем платформу, позволяющую создавать инструменты для статического анализа динамически формируемых выражений. Мы также продемонстрируем плагин к ReSharper, осуществляющий подсветку синтаксиса и обнаружение ошибок для встроенных в C# SQL и JSON, созданный с помощью этой платформы.

Введение

Часто при разработке сложных программных систем используется более, чем один язык программирования. В таком случае принято говорить об основном (или исходном) языке и одном или нескольких встроенных языках. Из строковых выражений основного языка динамически формируются программы на отличном от него языке,

[1]Работа выполнена при финансовой поддержке гранта ...

которые потом интерпретируются специальными, работающими во время исполнения компонентами, такими как базы данных или веб-браузеры. Большинство языков программирования общего назначения могут играть роль как основного, так и встроенного языка. Ниже приведены примеры использования встроенных языков.

Вызов javaScript из Java:

```
import javax.script.*;

public class InvokeScriptFunction {
    public static void main(String[] args) throws Exception {
        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("JavaScript");

        // JavaScript code in a String
        String script = "function hello(name) { print('Hello, ' + name); }";
        // evaluate script
        engine.eval(script);

        // javax.script.Invocable is an optional interface.
        // Check whether your script engine implements or not!
        // Note that the JavaScript engine implements Invocable interface.
        Invocable inv = (Invocable) engine;

        // invoke the global function named "hello"
        inv.invokeFunction("hello", "Scripting!!" );
    }
}
```

Код с использованием динамического SQL:

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

Использование нескольких встроенных в PHP языков (MySQL, HTML)

```
<?php
$query = 'SELECT * FROM '. $my_table; // Embedded SQL
$result = mysql_query($query);
// HTML markup generation
echo "<table>\n";
while ($line = mysql_fetch_array($result, MYSQL_ASSOC)) {
echo "\t<tr>\n";
foreach ($line as $col_value) {
```

```
echo "\t\t<td>$col_value</td>\n";}
echo "\t</tr>\n";}
echo "</table>\n";
?>
```

Встроенные языки позволяют компенсировать недостаток выразительности языков общего назначения в случае использования их в контексте специфичном для предметной области. Однако использование такого подхода сопряжено с рядом трудностей. Динамически формируемые выражения обычно конструируются из строковых констант и выражений основного языка посредством конкатенации в циклах, ветках условных операторов или рекурсивных процедурах, причем эти структуры могут вкладываться друг в друга, что порождает множество различных вариантов. Фрагменты кода на встроенных языках воспринимаются компилятором исходного языка как простые строки, не подлежащие анализу. Таким образом, стандартные средства не позволяют проводить даже простой синтаксический анализ динамически формируемых выражений. Невозможность статической проверки корректности формируемого выражения приводит к высокой вероятности возникновения ошибок во время выполнения программы.

Распространенной практикой при написании кода является использование интегрированных сред разработки, производящих подсветку синтаксиса и автодополнение, сигнализирующих о синтаксических ошибках, предоставляющих возможность проводить рефакторинг кода. Все эти функции значительно упрощают процесс разработки и отладки приложений. Полезными могут оказаться инструменты, проводящие статический анализ множества выражений, которые динамически формируются из строковых выражений основного языка во время выполнения программы. Данный процесс назовем статическим анализом динамически формируемых выражений или абстрактным анализом.

В рамках исследовательского проекта YaccConstructor[1], посвященного проведению экспериментов в области синтаксического анализа, ведется работа над платформой для создания инструментов, предназначенных для статического анализа кода на встроенных языках. В данной статье описаны разрабатываемая инфраструктура поддержки встроенных языков и ее компоненты: генераторы абстрактных лексических и синтаксических анализаторов. Также уделено внимание поддержке многих языков и приведен пример использования платформы для создания плагина к ReSharper[2] (плагин к Microsoft Visual Studio [3], расширяющий стандартные средства IDE), позволяющего анализировать динамически формируемые выражения.

1. Платформа

Многие редакторы и IDE используются для различных языков программирования. Чтобы дать возможность обрабатывать разные встроенные языки, которые на данном этапе не поддерживаются редактором или IDE, необходимо иметь соответствующую функциональность. Для этого используются модули, реализующие поддержку встроенных языков, ниже будем называть это языковыми расширениями. Возможность IDE поддерживать разные встроенные языки может быть полезна не только

пользователям, желающим получить поддержку встроенного языка, но и разработчикам для создания новых языковых расширений.

Существующие инструменты для работы со встроенными языками реализуют поддержку какого-то одного конкретного языка или поддержка нового языка может потребовать нетривиального ручного вмешательства. Чтобы получить поддержку встроенного языка без изменений в исходном коде, необходимо, с одной стороны, предоставить механизм для простой реализации поддержки нового языка и интеграции его в систему, а с другой — дать пользователю возможность указывать в исходном коде строки, соответствующие конкретным встроенным языкам.

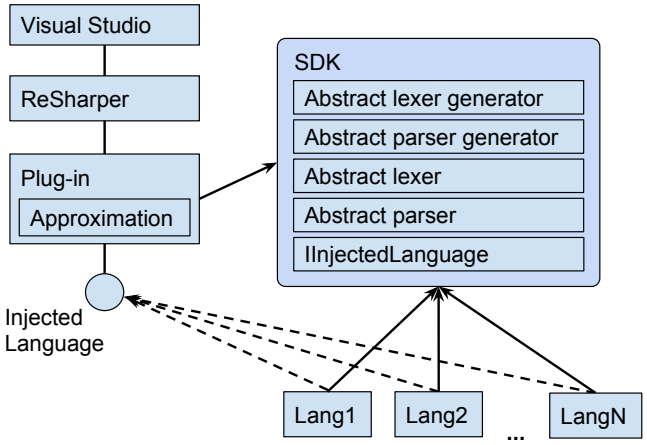

Рис. 1. Структура платформы

На рисунке 1 представлена общая структура платформы, основные модули которой описаны ниже.

Lang1, Lang2, ..., LangN — модули, реализующие поддержку встроенных языков. Каждый из таких модулей реализует общий интерфейс, позволяющий унифицировано подключать поддержку нового языка. Для создания нового языкового расширения разработчику необходимо установить плагин для ReSharper, описать грамматику и лексическую спецификацию языка, по этим файлам генерируется лексер и парсер. В результате разработчик получает модуль, позволяющий поддерживать новый встроенный язык.

SDK предоставляет разработчику набор инструментов и готовых функций (набор генераторов, функции синтаксического анализа), упрощающих разработку нового языкового расширения. Кроме того, он содержит интерфейс для парсеров IInjectedLanguageModule и атрибут InjectedLanguage, который даёт пользователю возможность указывать в коде программы встроенный язык для строковых выражений. Атрибут InjectedLanguage содержит поле languageName типа String (languageName - название встроенного языка). По значению поля languageName происходит поиск языкового расширения, соответствующего языку, название которого пользователь указал в атрибуте. Если языковое расширение для указанного пользователем языка не было подключено, то пользователь получит сообщение об ошибке.

Данный атрибут может быть применён к объявлениям методов. Если же используется стандартный метод, то необходимо указать его в отдельном конфигура-

ционном файле. Например, если реализуется собственный метод для выполнения Transact SQL запросов, то для того, чтобы проверять корректность принимаемого запроса, он должен быть отмечен атрибутом как показано в примере ниже.

```
[InjectedLanguage (“TSQL”)]
public static void ExecuteImmediate(string query)
{
Logger.log(query);
DB.execute(query);
}
```

Если предположить, что DB.execute — это библиотечный метод и мы не можем отметить его объявление, то необходимо добавить его в конфигурационный файл разрабатываемого инструмента.

Для простоты использования готового инструмента необходимо иметь возможность динамически загружать языковое расширение для поддержки нового встроенного языка. Динамическая загрузка осуществляется с помощью средства для создания расширяемых приложений Mono.Addins.

1.1. Анализ встроенных языков

Для синтаксического анализа обобщённого представления множества строк, таких как data-flow уравнение, регулярное выражение, существует алгоритм абстрактного синтаксического анализа [4]. Часто удобно считать, что компактное представление описывается с помощью графа, где на дугах содержатся терминальные символы (токены), а вершины соответствуют случаям конкатенации строк в процессе формирования запроса. Например, пусть обрабатывается следующий код, формирующий и выполняющий динамический запрос.

```
IF @X = @Y
SET @TABLE = ’#tbl1’
ELSE SET @TABLE = ’tbl2’
SET @S = ’SELECT x FROM ’ + @TABLE
EXECUTE (@S)
```

Переменная @S, содержащая динамически формируемый запрос, может принимать два значения в точке выполнения запроса. После обработки этого кода множество значений переменной @S в точке выполнения может быть представлено графом, показанным на рисунке 2.

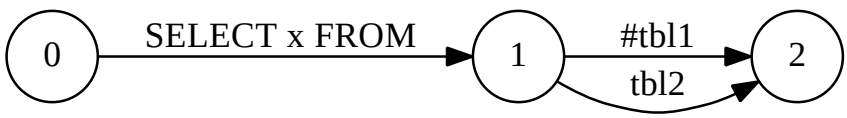

Рис. 2. (2)Входной граф для лексического анализатора, построенный по коду из примера 1

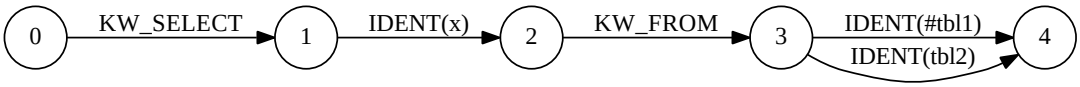

Рис. 3. (3)Граф для динамического запроса из примера

Граф, полученный в результате применения к нему процедуры токенизации или абстрактного лексического анализа, показан на рисунке 3.

Следующий шаг — абстрактный синтаксический анализ, который основан на идее переиспользования управляющих конструкций из классического синтаксического анализа и реализации специального механизма их интерпретации. По описанию синтаксиса анализируемого языка генерируются управляющие таблицы для анализатора. Интерпретатор таблиц (LR-автомат) при этом модифицируется таким образом, чтобы вычислять все возможные состояния синтаксического анализатора для каждой вершины графа [5]. Рассмотрим пример. Пусть задана следующая грамматика:

```
s -> Ae
e -> BD | CD
```

На вход построенному по данной грамматике анализатору подаётся граф, представленный на рисунке 4.

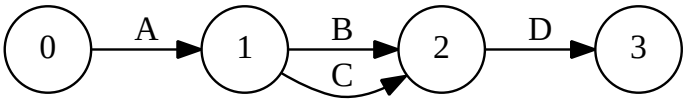

Рис. 4. (4) Пример входного графа

Во время синтаксического анализа будет вычислено множество состояний анализатора в каждой вершине графа (Fig.5).

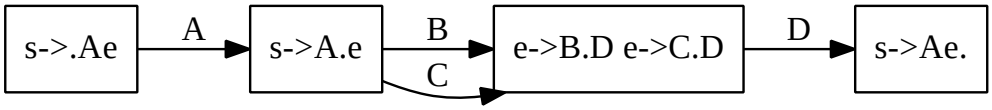

Рис. 5. (5) Состояния парсера для графа, представленного на Fig. 4

Несмотря на то, что уже существуют реализации инструментов для работы с динамически формируемыми выражениями в конкретных языках, хорошо показавшие себя на практике (например, Java String Analyzer [6], Alvor [7] или PHP String Analyzer [8]), отдельного внимания заслуживает вопрос обобщения лексического анализа и синтаксического разбора, используемых в таких инструментах, с целью создания генератора абстрактных анализаторов. Кроме того, при промышленной

разработке инструментов лексического и синтаксического анализа для языков программирования уже стали традиционными инструменты, позволяющие автоматически генерировать соответствующие анализаторы по спецификациям обрабатываемого языка. В области автоматизации разработки подобных инструментов для абстрактного анализа также получены результаты [11], но они требуют доработки.

1.2. Генератор лексических анализаторов

Генератор лексических анализаторов — это инструмент для получения лексического анализатора по лексической спецификации обрабатываемого языка. Абстрактный лексический анализатор основан на конечном преобразователе (finite-state transducer [9]). Это конечный автомат, который может выводить конечное число символов для каждого входного символа. В качестве основы нашего инструмента мы используем генератор лексических анализаторов для F# — FsLex. По сгенерированному описанию автомата преобразователь переводит входной граф в граф, содержащий соответствующие спецификации токены.

В отличие от классического анализа в абстрактном лексическом анализе бывают ситуации с “рваными” лексическими единицами, то есть случаи, когда токен находился на двух и более ребрах входного графа. Назовем такие токены разрывными литералами. В качестве примера рассмотрим код на C#, который содержит динамически формируемый запрос, написанный на T-SQL. Разрывный литерал был сконструирован при помощи тернарного оператора.

```
private void Go(bool cond)
{
string name_table = cond ? "1" : "2";
Program.ExecuteImmediate("select fld from tbl_" + name_table);
}
```

Результатом аппроксимации динамического запроса является граф, который подается на вход лексическому анализатору и который представлен на рис. Л

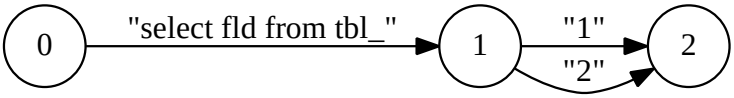

Рис. 6. (6)Результат аппроксимации динамического запроса в примере 1

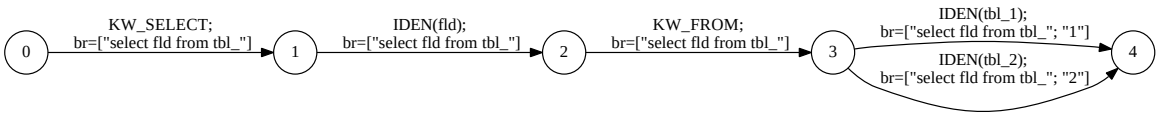

Рис. 7. (7) Результат лексического анализа для графа, представленного на рисунке 6

Результат работы токенизатора показан на рис. В. На этом рисунке также можно увидеть структуру backreference(или br). Эта структура хранит для каждого токена строки исходного кода из которых он образован, а также координаты начала и конца токена внутри этих строк. Сохраненная привязка может понадобится при дальнейшем разборе полученного графа и при выдаче сообщений о лексических ошибках.

1.3. Генератор синтаксических анализаторов

В рамках проекта YaccConstructor реализован генератор синтаксических анализаторов [1], основанный на RNGLR-алгоритме[12], который является модификацией классического GLR-алгоритма, поддерживающей работу с произвольными КС-грамматиками. Важной особенностью данной реализации является построение легковесного внутреннего представления результатов, над которым после завершения разбора выполняется вычисление пользовательской семантики. Это позволяет не производить лишних вычислений. Например, после получения промежуточного результата пользователь может выбрать только одно дерево и вычислять семантику только для него. Кроме этого, часть вычислений может оказаться ненужными, если они производились в ветке стека, которая впоследствии окажется ошибочной. Поэтому использование данной структуры позволяет ускорить абстрактный анализ.

При обработке графов в RNGLR-алгоритме в момент разветвления возникает ситуация, аналогичная классическим конфликтам Shift/Reduce и Reduce/Reduce, когда необходимо выполнять анализ по нескольким путям. В данном случае необходимость ветвления вызвана тем, что существует несколько вариантов для операции Shift. Ситуация не является классическим конфликтом, но по аналогии мы будем называть такие ситуации Shift/Shift конфликтами. В алгоритм анализа внесена возможность обработки таких ситуаций.

1.3.1. Обнаружение ошибок и восстановление после них

В случае классического синтаксического анализа с линейным входом, если не производится восстановление после ошибок, результатом анализа всегда является либо корректное дерево разбора, либо ошибка. В абстрактном синтаксическом анализе это не так: анализируется целое множество входных выражений, каждое из которых представляет собой некоторый путь во входном графе, и необходимо сообщать не только о первой найденной на каком-либо пути ошибке, но о первой ошибке на каждом пути. Для выполнения этого требования необходимо возвращать целый список ошибок. С другой стороны, не все выражения из входного множества обязательно содержат ошибку. Поэтому наряду со списком ошибок, если они были обнаружены в результате анализа, необходимо возвращать успешный результат разбора. Алгоритм абстрактного синтаксического анализа был изменен так, чтобы возвращать и компактное представление леса разбора, и список ошибок. Стоит отметить, что вне зависимости от количества корректных выражений во входном множестве, всегда будет возвращено лишь одно SPPF.

GLR-алгоритмы предназначены для анализа неоднозначных грамматик и оперируют со структурированным в виде графа стеком, который разветвляется при

возникновении конфликтов. Набор верхушек стека, из которых возможно продолжить анализ, называют фронтом. Если в процессе работы классического GLR-анализатора для некоторого состояния, лежащего на верхушке стека, невозможно сделать ни одного действия, то соответствующую ветку стека считают ошибочной и исключают из фронта. В классическом GLR-алгоритме анализа ошибка диагностируется в момент, когда во фронте не остается ни одной верхушки стека. Для абстрактного синтаксического анализатора такой подход верен не всегда. Ветви стека в абстрактном анализе соответствуют не только неоднозначностям грамматики, но и ветвлениям во входном графе, поэтому данная ситуация может сигнализировать об ошибке в одном из входных выражений. В случае обнаружения ошибки на некотором токене необходимо прекратить разбор тех выражений, которые содержали этот токен, чтобы не порождать ложных сообщений об ошибках. Алгоритм абстрактного синтаксического анализа обрабатывает входной граф, представленный списком вершин, отсортированным в топологическом порядке[GLR-based absract parsing]. Эта особенность немного усложняет механизм игнорирования ошибочных путей. В тот момент, когда алгоритм обнаружил ошибку во входе, происходит вычисление игнорируемого интервала вершин. Если есть отложенные операции push, то следующей вершиной, из которой должен быть продолжен анализ, является вершина с номером равным минимуму из номеров отложенных операций push. Если же отложенных операций push нет, то рассматриваем все вершины стеков, сравниваем номера уровней, на которые последует переход, берем минимальное из этих значений и продолжаем анализ из вершины с вычисленным номером. Такой подход позволяет пропустить только те части путей в графе, которые гарантированно не будут переиспользованы ни в одном корректном выражении.

Стоит отметить, что область диагностики ошибок и восстановления после них в GLR-алгоритмах синтаксического анализа плохо изучена[ссыль]. Конфликты, возникающие при разборе, усложняют задачу точной диагностики ошибок. Добавив конфликт Shift/Shift, мы еще больше усложнили эту задачу: появилась необходимость различать ошибки, полученные в результате выбора неправильной альтернативы во время классических конфликтов и ошибки, полученные в результате Shift/Shift конфликта. Ситуация окажется намного сложнее при наличии разнородных конфликтов: если грамматика анализируемого языка неоднозначна, и входной граф при этом содержит множество ветвлений. В данной ситуации вероятно обнаружение большого количества ошибок, при этом не ясно, какие из срабатываний действительно соответствуют ошибкам во входных выражениях, а какие являются ложными и возникли вследствие неоднозначности грамматики.

В рамках платформы был реализован механизм восстановления после ошибок для классического RNGLR-алгоритма [10]. Однако вопрос о необходимости восстановления после ошибок в абстрактном анализе необходимо исследовать отдельно, так как он может порождать слишком большое количество ложных ошибок для сложных выражений. Также необходимо оценить уменьшение производительности и выявить проблемы, связанные с переходом к абстрактному анализу.

1.3.2. Особенности вычисления семантики

Для обеспечения такой функциональности, как подсветка и автодополнение, не достаточно только лишь информации о синтаксической корректности выражений.

Необходима также работа с семантическими значениями языковых конструкций. При этом далеко не всегда нужно рассматривать все деревья разбора. При подсветке синтаксиса, например, достаточно выбрать из леса разбора подмножество корректных деревьев таким образом, чтобы множество всех токенов (листьев) было покрыто. То есть, чтобы для каждого токена существовало дерево из выбранного подмножества, которое содержит этот токен. Действительно, если у двух разных деревьев листья совпадают, то при подсветке синтаксиса не так важно, какому именно дереву принадлежит тот или иной лист. Важно, что это дерево синтаксически корректно.

Рассмотрим пример:

```
expr: (expr PLUS expr) | NUM
```

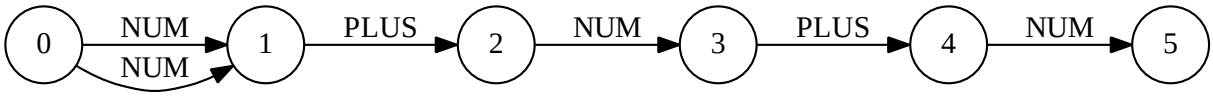

Рис. 8. (8) Граф, содержащий Shift/Reduce конфликт

Для данного графа в результате синтаксического разбора будет построено четыре дерева разбора: для каждого пути будет построено по два дерева из-за неоднозначности грамматики. Однако порождённые конфликтом Shift/Reduce или Reduce/Reduce деревья соответствуют одному и тому же пути в графе, а значит, множества их листьев совпадают. Поэтому при поддержке подсветки синтаксиса для каждого пути в графе достаточно рассмотреть только одно из таких деревьев. Наиболее сложной для реализации поддержки языков является ситуация, соответствующая ветвлению во входном графе, так как в худшем случае придётся рассматривать все возможные деревья разбора.

В рамках проекта создан генератор, который создаёт семантику для подсветки синтаксиса языка. Также он генерирует конфигурационный файл, в котором пользователь может указать, в какой цвет окрашивать ту или иную языковую конструкцию.

2. Дальнейшее развитие

В дальнейшем планируется как развитие платформы, так 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[link to SYRCOSE, master work] или для миграции на новые технологии, например, LINQ. Эта задача связана с двумя проблемами: возможностью проведения нетривиальных трансформаций (могут потребоваться некоторые advanced техники[ter,db]) 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.

Список литературы

1. *Кириленко Я.А., Григорьев С. В.,Авдюхин Д. А.* Разработка синтаксических анализаторов в проектах по автоматизированному реинжинирингу информационных систем.//Научно-технические ведомости СПбГПУ информатика, телекоммуникации, управление. – 2013. – №174. – С. 94-98

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

3. *Aivar Annamaa, Andrey Breslav, and Varmo Vene* Using abstract lexical analysis and parsing to detect errors in string-embedded dsl statements.// In Marina Walden and Luigia Petre, editors, Proceedings of the 22nd Nordic Workshop on Programming Theory.

4. *Mohri Mehryar* Finite-State Transducers in Language and Speech Processing. // Association for Computational Linguistics. – 1997. – http://www.cs.nyu.edu/mohri/pub/cl1.pdf

5. *Иванов А.В.* Восстановление после ошибок в GLR-алгоритме.// Курсовая работа. – СПбГУ. – 2013.

6. *Вербицкая Е.А., Григорьев С.В.* Абстрактный лексический анализ. // СПИСОК-2013: Материалы всероссийской научной конференции по проблемам информатики. – 23–26 апр. 2013 г., Санкт-Петербург. —- СПб.: Издательство ВВМ. – 2013. – С. 792

7. *Elizabeth Scott and Adrian Johnstone* Right nulled GLR parsers.// ACM Trans. Program. Lang. – Syst. 28, 4 (July 2006) – P. 577-618.

8. *Semen Grigorev and Iakov Kirilenko* GLR-based abstract parsing.// In Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia (CEE-SECR '13). – York, NY, USA. – Article 5. – 9 pages.

9. *Giorgios Robert Economopoulos* Generalised LR parsing algorithms. – 2006.

10. *Andrey Terekhov* Good technology makes the difficult task easy.// In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013). – ACM, New York, NY, USA. – P.683–686.

11. *D. Yu. Boulychev, D. V. Koznov, and Andrey A. Terekhov* On Project-Specific Languages and Their Application in Reengineering.// In Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR '02). – IEEE Computer Society, Washington, DC, USA. – p. 177–185.

English Title

Grigorev S. V.

P.G. Demidov Yaroslavl State University, Sovetskaya str., 14, Yaroslavl, 150000, Russia

Keywords: attractor, bifurcation

Complex information systems are often implemented using more than one programming language. Sometimes this variety takes form of one host and one or few string-embedded languages. Textual representation of clauses in a string-embedded language is built at run time by a host program and then analyzed, compiled or interpreted by a dedicated runtime component (database, web browser etc.) Most general-purpose programming languages may play role of the host; one of the most evident examples of string-embedded language is dynamic SQL which was specified in ISO SQL standard and is supported by the majority of DBMS. Standard IDE functionality such as code completion or syntax highlighting can really helps developers who use this technique. There are several tools providing this functionality, but they all process only one concrete string-embedded language and cannot be easily extended for supporting other language. We present a platform which allows to easily create tools for string-embedded language processing. We also demonstrate a plug-in for ReSharper based on this platform which provides code highlighting and static errors checking for string-embedded languages in C# (TSQL, JSON).

Сведения об авторе:
Иванов Иван Иванович,
Ярославский государственный университет им. П.Г. Демидова,
канд. физ.-мат. наук, доцент