Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование
информационных систем

Системное Программирование

Азимов Рустам Шухратуллович

Диагностика синтаксических ошибок в динамически формируемом коде

Бакалаврская работа

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

Рецензент:
программист ООО "ИнтеллиДжей Лабс" Авдюхин Д. А.

Санкт-Петербург
2016

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems

Software Engineering

Rustam Azimov

Syntax errors detection in dynamically generated code

Bachelor's Thesis

Scientific supervisor:
Senior Lecturer Semen Grigorev

Reviewer:
Software Developer at IntelliJ Labs OOO Dmitry Avdyukhin

Saint-Petersburg
2016

Оглавление

Введение **4**

1. Постановка задачи **5**

2. Обзор **6**

2.1. Регулярная аппроксимация динамически формируемого выражения . . 6

2.2. Алгоритм ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения 6

2.3. Понятие синтаксической ошибки в алгоритмах LR-семейства 11

2.4. Проект YaccConstructor 12

3. Понятие синтаксической ошибки **14**

4. Механизм диагностики ошибок **16**

4.1. Компактное представление префиксов внутреннего графа 16

4.2. Алгоритм построения префиксов 17

4.3. Алгоритм диагностики ошибок 19

5. Корректность механизма диагностики ошибок **23**

5.1. Корректность алгоритма построения префиксов 23

5.2. Корректность алгоритма диагностики ошибок 30

6. Экспериментальное исследование **34**

7. Заключение **37**

Список литературы **38**

Введение

При разработке сложных программных систем существует проблема недостатка выразительности и гибкости языков программирования общего назначения. Часто для решения данной проблемы помимо основного языка используют один или несколько других (встроенных) языков. В данном подходе программа, написанная на основном языке, динамически формирует строковое представление кода на встроенном языке, и далее сформированная строка анализируется и выполняется специальными компонентами (базы данных, веб-браузер) во время исполнения основной программы.

Как правило, динамически формируемые выражения, описывающие код программ на встроенных языках, конструируются с использованием конкатенаций строковых литералов в ветках условных операторов, циклах и рекурсивных процедурах. Это приводит к множеству возможных значений встроенного кода, что не позволяет в общем виде использовать статический анализ для проверки корректности формируемого выражения. В результате этого становятся недоступными такие типы функциональности, как информирование о синтаксических ошибках, автодополнение и подсветка синтаксиса. Отсутствие этих возможностей повышает вероятность ошибок, которые обнаруживаются лишь во время выполнения программы, а также усложняет процесс разработки и тестирования.

Существует ряд инструментов, позволяющих проводить анализ динамически формируемых строковых выражений: Java String Analyzer [5, 9], PHP String Analyzer [12], Alvor [8, 2, 1], IntelliLang [7], PHPStorm [14], Varis [13] (анализ этих технологий приведен в работе [21]). Большинство реализаций проводят диагностику синтаксических ошибок, но плохо расширяемы: как в смысле поддержки других языков, так и в смысле решения новых задач. Существует алгоритм, описанный в статье [19] и реализованный как часть проекта YaccConstructor [11], который позволяет провести синтаксический анализ динамически формируемых выражений. В отличие от других инструментов, реализация данного алгоритма хорошо расширяема и строит конечное представление леса разбора относительно входной грамматики, которое может быть использовано в дальнейшем семантическом анализе. Недостаток данного алгоритма заключается в отсутствии механизма диагностики синтаксических ошибок. Устранению этого недостатка и посвящена данная работа.

1. Постановка задачи

Целью данной работы является разработка механизма диагностики ошибок в алгоритме ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения, реализованного в проекте YaccConstructor [11].

Для достижения этой цели были сформулированы следующие задачи.

- Определить понятие синтаксической ошибки в терминах регулярной аппроксимации динамически формируемого выражения.
- Разработать механизм диагностики ошибок в синтаксическом анализе регулярной аппроксимации динамически формируемого выражения.
- Доказать корректность механизма.
- Реализовать предложенный механизм в рамках проекта YaccConstructor.
- Провести экспериментальное исследование.

2. Обзор

Алгоритм, доработке которого посвящена данная работа, основан на алгоритме RNGLR [16] (Right-Nulled Generalized LR). В свою очередь, RNGLR является модификацией алгоритма Generalized LR (GLR) [18], предназначенного для анализа естественных языков. GLR-алгоритм использует управляющие таблицы семейства LR (Left-to-right Rightmost) алгоритмов [6] с возможностью содержать несколько действий в одной ячейке. Таким образом, при описании работы алгоритма ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения необходимо рассмотреть все вышеперечисленные алгоритмы.

В данном разделе дано краткое описание работы алгоритма ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения. Также рассмотрено понятие синтаксической ошибки в алгоритмах LR-семейства и описан проект, в рамках которого велась разработка предложенного алгоритма.

2.1. Регулярная аппроксимация динамически формируемого выражения

Под регулярной аппроксимацией подразумевается аппроксимирование сверху множества значений динамически формируемого выражения некоторым регулярным выражением. В терминах привычных для задач распознавания, проверка корректности выражений из аппроксимирующего множества формулируется как задача проверки включения регулярного языка в другой (как правило, контекстно-свободный язык), которая разрешима во многих важных на практике случаях [3]. Метод построения рассматриваемой аппроксимации, представленный в работе [4], реализован [10] в проекте YaccConstructor.

2.2. Алгоритм ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения

Чтобы понять принцип работы данного алгоритма, сначала рассмотрим алгоритмы LR-семейства, такие как LR(1), SLR(1), LALR(1). Все эти алгоритмы принимают на вход строку и контекстно-свободную грамматику. Если удалось построить детерминированную управляющую таблицу по данной грамматике, то перечисленные алгоритмы позволяют ответить на вопрос о принадлежности произвольной строки языку, порождаемому данной грамматикой. Входная строка читается слева направо и в процессе чтения программа, выполняющая синтаксический анализ, (*парсер*) меняет свои состояния по правилам, установленным в управляющей таблице. Нужная ячейка таб-

лицы определяется текущим состоянием и правым контекстом (одним или несколькими следующими терминалами) входной строки. История состояний программы хранится в виде стека. Кроме перехода из состояния в состояние ячейка управляющей таблицы содержит действия, которые бывают двух видов: *сдвиг* (*push, shift*) — чтение следующей части входной строки и *свертка* (*reduce*) — применение одного из правил входной грамматики. Строка принимается алгоритмом лишь в том случае, если она была полностью обработана, и конечное состояние парсера — одно из заранее определенных *принимающих состояний*.

Иногда построить детерминированную таблицу не удается, и в одной ячейке управляющей таблицы могут быть конфликты следующих видов: сдвиг/свертка (shift/reduce) и свертка/свертка (reduce/reduce). Существует два подхода, применяемых к обработке таких неоднозначностей. Первый из них подразумевает разрешение конфликтов, а второй — анализ всех возможных последовательностей действий, предпринимаемых парсером. Алгоритмы семейства GLR, разработанные для анализа произвольных неоднозначных контекстно-свободных грамматик, используют второй подход. При работе этих алгоритмов порождается множество стеков состояний и деревьев разбора, для эффективного хранения которых используются специализированные структуры данных: структурированный в виде графа стек GSS (Graph Structured Stack) и компактное представление леса разбора SPPF [15] (Shared Packed Parse Forest).

Следует отметить, что оригинальный GLR-алгоритм не способен анализировать все контекстно-свободные грамматики, поэтому в работе [16] был предложен RNGLR-алгоритм, который является его расширением. RNGLR специальным способом обрабатывает *обнуляемые справа правила* входной грамматики (имеющие вид $A \to \alpha\beta$, где β выводит пустую строку ϵ). RNGLR, как и GLR, строит GSS "слоями", т.е. сначала выполняются все возможные операции свертки для текущего терминала, после чего осуществляется операция сдвига к следующему терминалу входной строки. Вершина GSS представляется в виде пары (s, l), где s — состояние парсера, а l — уровень (позиция текущего терминала во входном потоке).

Алгоритм, доработке которого посвящена данная работа, является расширением алгоритма RNGLR, способным производить синтаксический анализ регулярного множества входных строк. При анализе рассматриваемым алгоритмом регулярного множества, состоящего из единственной строки конечной длины, результат будет аналогичен результату анализа алгоритма RNGLR. В расширении вместо строки на вход подается конечный недетерминированный автомат с единственными начальным и конечным состояниями, который порождает регулярную аппроксимацию значений динамически формируемого выражения. Данный автомат представляется в виде ориентированного графа (далее *входного графа*) с вершинами — состояниями автомата и ребрами — переходами автомата. Строится *внутренний граф*, который получается из входного — ассоциацией вершин GSS и некоторых коллекций с вершинами гра-

фа. Основная идея расширения заключается в перемещении по внутреннему графу и последовательном построении GSS. В качестве "слоев" выступают вершины внутреннего графа, таким образом каждая вершина GSS хранит состояние парсера *state* и уровень *level* (который отождествляется с вершиной внутреннего графа). Теперь операция сдвига выполняется не по одному токену, а по множеству токенов, нагруженных на дуги, исходящие из текущей вершины графа.

Для организации порядка обработки вершин внутреннего графа используется глобальная очередь Q. При добавлении новой вершины GSS сначала все *свертки длины 0* (*zero-reductions*) добавляются в очередь операций reduce, после этого выполняется операция сдвига следующих токенов со входа, а соответствующие вершины графа добавляются в очередь Q. Так как добавление нового ребра GSS может порождать новые свертки, то в очередь на обработку Q необходимо добавить вершину внутреннего графа, которой соответствует начальная вершина добавленного ребра (процесс построения GSS описан в алгоритме 3). Операции свертки проводятся вдоль путей в GSS, таким образом, если начальная вершина нового ребра ранее присутствовала в GSS, то необходимо заново вычислить свертки путей, проходящих через эту вершину (функция applyPassingReductions в алгоритме 2).

Кроме состояния анализатора *state* и уровня *level*, в вершине GSS хранится коллекция *проходящих сверток*. Проходящая свертка — это тройка $(startV, N, l)$, соответствующая свертке, чей путь содержит данную вершину GSS, а длина оставшейся части пути равна l. Проходящие свертки сохраняются в каждой вершине пути (кроме первой и последней) во время поиска путей в функции *makeReductions* (алгоритм 2).

В вершинах внутреннего графа хранятся следующие коллекции:

- *processed* — вершины GSS, для которых ранее были вычислены все операции push;
- *unprocessed* — вершины GSS, операции push для которых ещё только предстоит выполнить;
- *reductions* — очередь операций reduce, которые ещё только предстоит выполнить;
- *passingReductionsToHandle* — пары из вершины GSS и ребра GSS, вдоль которых необходимо обновить проходящие свертки.

Algorithm 1 Алгоритм ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения

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

Algorithm 2 Обработка вершины внутреннего графа

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

Algorithm 3 Построение GSS

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

2.3. Понятие синтаксической ошибки в алгоритмах LR-семейства

Конкретная формальная грамматика делит множество всевозможных строк на *принимаемые* (принадлежащие языку, порожденному данной грамматикой) и *непринимаемые*. Несоответствие синтаксическим правилам, определенным грамматикой называется *синтаксической ошибкой*. Таким образом, в строке присутствует синтаксическая ошибка (может быть не одна) тогда и только тогда, когда она не является принимаемой для соответствующей грамматики.

Алгоритмы из LR-семейства обладают свойством *корректного префикса* (*correct-prefix* property) [6], то есть если данные алгоритмы корректно определяют принимаемая ли строка для рассматриваемой грамматики, то они обнаруживают ошибку на первом терминале, который образует из прочтенной части входной строки *некорректный префикс* (ни одна принимаемая строка не начинается на данный префикс).

Если алгоритмы из LR-семейства обнаружили синтаксическую ошибку, то возможен один из двух вариантов:

- парсер обработал всю входную строку, но итоговое состояние не принадлежит множеству принимающих состояний;
- парсер выполнил все операции свертки для текущего терминала, но не имеется ни одной операции сдвига к следующему терминалу входной строки.

Таким образом, данные алгоритмы продолжают свою работу до тех пор, пока обработанная часть строки является корректным префиксом.

2.4. Проект YaccConstructor

В рамках исследовательского проекта YaccConstructor [11] лаборатории языковых инструментов JetBrains на математико-механическом факультете СПбГУ проводятся исследования в области лексического, синтаксического анализа, а также статического анализа встроенных языков. Проект YaccConstructor является модульным инструментом, имеющим собственный язык спецификации грамматик. Также в нем объединены различные алгоритмы лексического и синтаксического анализа. В рамках проекта была создана платформа для статического анализа встроенного кода. На рис. 1 представлена диаграмма последовательности, иллюстрирующая взаимодействие модулей платформы. Выделена компонента, осуществляющая синтаксический анализ множества значений динамически формируемого выражения.

Предыдущая реализация платформы игнорировала синтаксические ошибки, что усложняло процесс разработки и тестирования программных систем, использующих данную платформу. Однако это повлекло необходимость разработки механизма диагностики ошибок в рамках алгоритма синтаксического анализа, чему и посвящена данная работа.

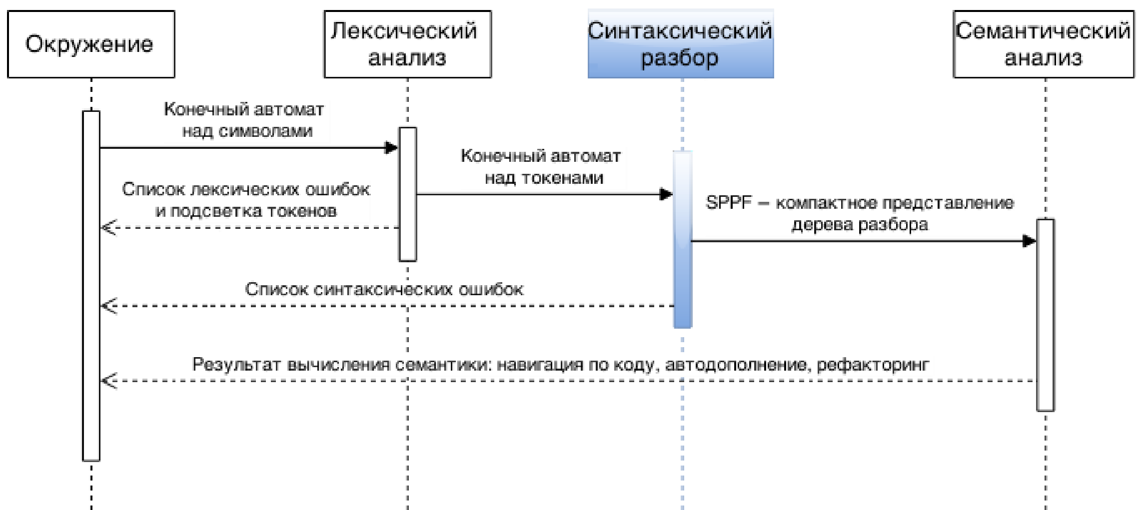

Рис. 1: Диаграмма последовательности: взаимодействие компонентов инструмента YaccConstructor (рисунок взят из работы [20])

3. Понятие синтаксической ошибки

Прежде чем описывать алгоритм диагностики ошибок, необходимо определить понятие синтаксической ошибки для алгоритма ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения. На вход данного алгоритма подается не одна строка, а множество строк, поэтому необходимо обнаруживать ошибки во всех строках из данного множества. Для этого необходимо определить понятие синтаксической ошибки для алгоритма RNGLR. Определим сначала понятие синтаксической ошибки для GLR-алгоритма.

GLR-алгоритм, получивший на вход строку, не принимаемую соответствующей грамматикой, останавливает свою работу в двух случаях:

- синтаксический анализатор обработал всю входную строку, но среди множества итоговых состояний ни одно состояние не принадлежит множеству принимающих состояний;
- синтаксический анализатор выполнил все операции свертки для текущего терминала, но среди множества состояний текущего уровня не имеется ни одного состояния, по которому в управляющей таблице существует операция сдвига к следующему терминалу входной строки.

Таким образом, пока обработанная часть входной строки является корректным префиксом данной грамматики, хотя бы один стек из множества стеков состояний текущего уровня может быть продолжен с использованием операции сдвига к следующему терминалу входной строки. Другими словами, GLR-алгоритм, как и алгоритмы LR-семейства, обладает свойством корректного префикса. *Ошибочным терминалом* входной строки GLR-алгоритма, будем называть первый терминал из необработанной части входной строки, на момент обнаружения синтаксической ошибки (если обработана вся строка, то это специальный терминал, обозначающий конец строки).

Алгоритм RNGLR также обладает свойством корректного префикса, т.к. является расширением GLR-алгоритма со специальной обработкой некоторых правил входной грамматики. Определение *ошибочного терминала* входной строки RNGLR-алгоритма аналогично определению ошибочного терминала GLR алгоритма.

Ввиду того, что алгоритм ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения, в отличие от RNGLR-алгоритма, принимает на вход не строку, а множество строк, то необходимо искать синтаксические ошибки во всем входном множестве строк. Для каждой отдельной строки понятие синтаксической ошибки совпадает с понятием синтаксической ошибки в RNGLR-алгоритме. Множество входных строк выражено конечным недетерминированным автоматом с единственными начальным и конечным состояниями. Данный автомат представляется в виде ориентированного графа (внутреннего графа) с терминалами

на ребрах. В таком случае, обработанная часть входных данных — не префикс строки, а терминалы, ассоциированные с ребрами (то есть нагруженные на ребрах) в пути во внутреннем графе из начальной вершины в вершину, соответствующую текущему уровню. Рассматриваемый путь будем называть *префиксом внутреннего графа*. Если строка p, образованная терминалами на ребрах рассматриваемого пути префикса внутреннего графа P, является корректным префиксом эталонной грамматики, то префикс внутреннего графа P назовем *корректным*, иначе — *некорректным*.

Пусть непустая строка p является корректным префиксом для рассматриваемой грамматики, тогда при обработке данной строки RNGLR-алгоритмом будут прочитаны все её терминалы и на последнем уровне GSS будет хотя бы одна вершина. Пусть $(s_1, .., s_n)$ — все состояния вершин рассматриваемого последнего уровня GSS. Тогда в последней вершине V пути префикса внутреннего графа P $\exists(v_1, .., v_n)$ — GSS-вершины, где $v_i.state = s_i, \forall i \in [1, .., n]$. Префикс внутреннего графа P назовем *корректным для GSS-вершины* v, если $v \in (v_1, .., v_n)$. Если строка p пустая и она является корректным префиксом для рассматриваемой грамматики, то префикс внутреннего графа P состоит из единственной начальной вершины внутреннего графа и назовем его корректным для GSS-вершин $(u_1, .., u_n)$, где $\forall i, u_i$ — либо начальная GSS-вершина, либо существует последовательность сверток длины 0, приводящая парсер из начального состояния в состояние $u_i.state$.

Назовем *ошибочным ребром* ребро внутреннего графа e, которое не нагружено специальным терминалом конца строки EOF, такое, что существует хотя бы один корректный префикс внутреннего графа P, заканчивающийся в вершине, из которой исходит данное ребро, но при добавлении ребра e в конец префикса P образуется некорректный префикс внутреннего графа. Если ребро e нагружено терминалом конца строки EOF, то ребро e является *ошибочным*, если существует хотя бы один корректный префикс внутреннего графа P, заканчивающийся в вершине, из которой исходит данное ребро, но строка, образованная последовательностью терминалов на ребрах пути P не является принимаемой эталонной грамматикой. Аналогом ошибочного терминала во входной строке RNGLR-алгоритма, является ошибочное ребро внутреннего графа.

Таким образом, цель диагностики ошибок в рамках синтаксического анализа регулярной аппроксимации множества значений динамически формируемого выражения заключается в обнаружении ошибочных ребер внутреннего графа и выводе корректных префиксов внутреннего графа, заканчивающихся в вершине, из которой исходит рассматриваемое ошибочное ребро, и становящихся некорректными при добавлении в конец этого ребра.

4. Механизм диагностики ошибок

Предлагаемый механизм диагностики ошибок состоит из двух частей:

- алгоритм синтаксического анализа регулярной аппроксимации динамически формируемых выражений (далее основной анализ) модифицируется, позволяя для каждой GSS вершины строить все корректные для нее префиксы внутреннего графа;
- после основного анализа с помощью построенных префиксов обнаруживаются ошибочные ребра внутреннего графа.

Далее в этой главе будут рассмотрены: компактное представление префиксов внутреннего графа, алгоритм построения корректных префиксов внутреннего графа для GSS-вершин и алгоритм диагностики ошибок, проводящий анализ построенных префиксов.

4.1. Компактное представление префиксов внутреннего графа

Так как внутренний граф может иметь циклы, то множество различных префиксов внутреннего графа может быть бесконечным. В качестве компактного представления всех корректных префиксов внутреннего графа для вершины GSS используется ориентированный граф с выделенным множеством начальных вершин (далее *граф префиксов*). Из-за особенностей операций, используемых при построении графов префиксов, начальные вершины представляют не начало, а конец хранимых префиксов внутреннего графа в рассматриваемом графе префиксов. Каждая вершина в графе префиксов ассоциируется с ребром GSS или является специальной вершиной EOP (End Of Prefix). Ребра GSS, в свою очередь, будем делить на три вида:

- *терминальное* ребро — ребро, порожденное операцией сдвига по какому-то терминалу t;
- *нетерминальное* — ребро, порожденное операцией свертки длины l, где $l > 0$;
- *обнуляемое* — ребро, порожденное операцией свертки длины l, где $l = 0$.

Будем говорить, что начальная вершина V графа префиксов GP_1 *соединена* с графом префиксов GP_2, если для любой начальной вершины U графа префиксов GP_2, существует ребро из вершины V в вершину U. Каждая, кроме EOP, начальная вершина графа префиксов соединена с одним графом префиксов. Вершина EOP не имеет исходящих дуг.

С каждым нетерминальным ребром ассоциируется множество путей в GSS, по которым произведена операция свертки для данного нетерминального ребра. Будем

говорить что данное нетерминальное ребро *порождает* каждый путь из рассмотренного множества путей GSS.

Рассмотрим путь $(V_1, .., V_n)$ в графе префиксов GP как последовательность вершин графов префиксов. Удалим вершину EOP, если она присутствует, а также заменим все вершины в данном пути на ребра GSS, с которыми они ассоциируются. Получим последовательность $(e_1, .., e_n)$ ребер GSS. *Раскрытием* данной последовательности будем называть последовательность, получающуюся в результате применения следующих действий:

- все нетерминальные ребра GSS e, заменяются на последовательность ребер, соответствующую одному из порожденных ребром e пути;
- все обнуляемые ребра GSS удаляются из последовательности.

Если после конечного числа раскрытий в последовательности останутся только терминальные ребра GSS, то будем говорить, что изначальный путь в графе префиксов *сводится* к строке, получающейся инвертированием этой последовательности терминальных ребер и их замены на терминалы, с которыми они ассоциированы. Если полученная строка получается заменой ребер в пути префикса внутреннего графа P, на терминалы, которыми нагружены эти ребра, то путь в графе префиксов *сводится* к префиксу внутреннего графа P. Будем говорить, что граф префиксов GP *порождает* префикс внутреннего графа P, если $\exists(V_1, .., V_n, EOP)$ — путь в графе префиксов GP, где V_1 — одна из начальных вершин графа префиксов GP, который сводится к префиксу внутреннего графа P. В графе префиксов также могут быть циклы, что позволяет сводиться к бесконечному множеству префиксов внутреннего графа.

4.2. Алгоритм построения префиксов

Данный алгоритм является модификацией алгоритма ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения. Добавляется ассоциация вершин GSS с коллекцией $prefixes$ — граф префиксов, порождающий все корректные префиксы внутреннего графа для данной GSS вершины. После создания начальной GSS-вершины в ее графе префиксов создается начальная вершина EOP. Добавляется ассоциация нетерминальных ребер GSS с коллекцией $paths$ — множество путей в GSS, порождаемых рассматриваемым нетерминальным ребром. Функция $addEdge$ модифицируется, добавляется дополнительный параметр $pathsToAdd$ — множество путей в GSS, которое необходимо добавить к множеству путей в GSS, порождаемых ребром $edge$ из вершины v_t в вершину v_h. Также создается начальная вершина графа префиксов $v_t.prefixes$, ассоциированная с ребром $edge$ и соединенная с графом префиксов $v_h.prefixes$. Функция $addVertex$ не изменилась.

Algorithm 4 Модификация построения GSS

1: **function** ADDEDGE($v_h, innerGraphV, state_t, isZeroReduction, pathsToAdd$)
2: $(v_t, isNew) \leftarrow$ ADDVERTEX($innerGraphV, state_t$)
3: **if** GSS does not contain edge from v_t to v_h **then**
4: $edge \leftarrow$ create new edge from v_t to v_h
5: $\mathcal{Q}.Enqueue(innerGraphV)$
6: **if** not $isNew$ and $v_t.passingReductions.Count > 0$ **then**
7: add $(v_t, edge)$ in $innerGraphV.passingReductionsToHandle$
8: **if** not $isZeroReduction$ **then**
9: **for all** e in outgoing edges of $innerGraphV$ **do**
10: calculate the set of reductions by v and the token on e
11: and add them in $innerGraphV.reductions$
12: $V \leftarrow$ vertex of the prefix graph,
13: associated with $edge$ and connected to $v_h.prefixes$
14: add V to initial vertexes of $v_t.prefixes$
15: **if** pathsToAdd is not empty **then**
16: $edge \leftarrow$ edge from v_t to v_h
17: add all paths in $pathsToAdd$ to $edge.paths$

Параметр $pathsToAdd$ при вызове функции $addEdge$ для терминальных или обнуляемых ребер GSS является пустым множеством.

Algorithm 5 Модификация операции сдвига

1: **function** PUSH($innerGraphV$)
2: $\mathcal{U} \leftarrow$ 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: $push \leftarrow$ calculate next state by $v_h.state$ and the token on e
7: $pathsToAdd \leftarrow$ empty set
8: ADDEDGE($v_h, e.Head, push, false, pathsToAdd$)
9: add v_h in $innerGraphV.processed$

Algorithm 6 Модификация операции свертки

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

После описанных модификаций алгоритм синтаксического анализа регулярной аппроксимации динамически формируемого выражения для каждой вершины GSS дополнительно конструирует все корректные для нее префиксы внутреннего графа.

4.3. Алгоритм диагностики ошибок

Данный алгоритм получает на вход внутренний граф с построенными в ходе основного синтаксического анализа структурами (в том числе и префиксами внутреннего графа GSS вершин), а также сгенерированные RNGLR-таблицы. Алгоритм делает обход внутреннего графа и для каждого исходящего из вершины внутреннего графа ребра анализирует множества графов префиксов соседних вершин.

Для обхода внутреннего графа используется глобальная очередь Q. Все ребра

внутреннего графа, ведущие не в конечную вершину, обрабатываются в функции $processVertex$. Для ребер, ведущих в конечную вершину внутреннего графа, используется глобальная очередь F, с последующей обработкой в функции $processEOF$.

В результате работы алгоритма все ребра из множества $errors$ являются ошибочными, а ребра из множества $probErrors$ — возможно ошибочными. То есть множество всех ошибочных ребер принадлежит объединению двух данных множеств. Кроме того, с элементами этих множеств ассоциируются множества графов префиксов (элемент e множества $errors$ или множества $probErrors$ ассоциируется со множеством $errors[e]$ или $probErrors[e]$ соответственно), которые порождают корректные префиксы, заканчивающиеся в вершине, из которой исходит рассматриваемое ребро. Данные множества могут быть использованы при создании сообщения пользователю о возможных ошибках динамически формируемого выражения. Все префиксы, порождаемые графами префиксов из множества $errors[e]$, становятся некорректными при добавлении в конец ребра e. Все префиксы, порождаемые графами префиксов из множества $probErrors[e]$, возможно становятся некорректными при добавлении в конец ребра e. Но множество всех корректных префиксов внутреннего графа, заканчивающихся в вершине, из которой исходит ребро e, и становящихся некорректными при добавлении ребра e в конец, порождается хотя бы одним графом префиксов из объединения множеств $errors[e]$ и $probErrors[e]$, где множество $errors[e]$ пусто, если $e \notin errors$, и множество $probErrors[e]$ пусто, если $e \notin probErrors$.

Algorithm 7 Алгоритм диагностики ошибок

```
1: function findErrors(inputGraph, parserSource)
2:     Q.Enqueue(inputGraph.startVertex)
3:     while Q is not empty do
4:         v ← Q.Dequeue()
5:         processVertex(v)
6:     while F is not empty do
7:         e ← F.Dequeue()
8:         processEOF(e)
```

Algorithm 8 Анализ неконечных ребер внутреннего графа

```
1: function processVertex(innerGraphV)
2:     for all e in outgoing edges of innerGraphV do
3:         if token on e is EOF then
4:             F.Enqueue(e)
5:         else
6:             if e.Head was not processed then
7:                 Q.Enqueue(e.Head)
8:             headPrefixes ← all prefixes graphs
9:              from all GSS vertexes of e.Head;
10:            pushedPrefixes ← all prefixes graphs, to which connected
11:             some initial vertex of prefixes graph in headPrefixes,
12:             associated with terminal edge;
13:            withCycle ← all cyclical prefixes graphs in pushedPrefixes
14:            withoutCycle ← all non-cyclical graphs in pushedPrefixes
15:            notPushedPrefixes ← all prefixes graphs
16:             from all GSS vertexes of innerGraphV,
17:             without prefixes graphs from pushedPrefixes;
18:            for all p in notPushedPrefixes do
19:                if prefixes graph p has a cycle then
20:                    add e to probErrors
21:                    add p to probErrors[e]
22:                else
23:                    for all prefixPath in paths of prefixes graph p do
24:                        pathG ← prefixes graph generated by prefixPath
25:                        if withoutCycle does not have any prefixes graph with
26:                             equivalent to prefixPath path  then
27:                            if withCycle is not empty then
28:                                add e to probErrors
29:                                add pathG to probErrors[e]
30:                            else
31:                                add e to errors
32:                                add pathG to errors[e]
```

Algorithm 9 Анализ конечных ребер внутреннего графа

```
1: function processEOF(edge)
2:     acceptedPrefixes ← all prefixes graphs
3:      from all GSS vertexes of edge.Tail with accepted state;
4:     withCycle ← all cyclical prefixes graphs in acceptedPrefixes
5:     withoutCycle ← all non-cyclical graphs in acceptedPrefixes
6:     notAcceptedPrefixes ← all prefixes graphs
7:      from all GSS vertexes of edge.Tail with not accepted state;
8:     for all p in notAcceptedPrefixes do
9:         if prefixes graph p has a cycle then
10:            add edge to probErrors
11:            add p to probErrors[edge]
12:        else
13:            for all prefixPath in paths of prefixes graph p do
14:                pathG ← prefixes graph generated by prefixPath
15:                if withoutCycle does not have any prefixes graph
16:                     with equivalent to prefixPath path  then
17:                    if withCycle is not empty then
18:                        add edge to probErrors
19:                        add pathG to probErrors[edge]
20:                    else
21:                        add edge to errors
22:                        add pathG to errors[edge]
```

5. Корректность механизма диагностики ошибок

5.1. Корректность алгоритма построения префиксов

В данном разделе приведено доказательство того, что построенные графы префиксов порождают только корректные для соответствующей вершины стека префиксы внутреннего графа.

Теорема 1. *Каждый корректный для GSS-вершины v префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_l)$ во внутреннем графе, порождается графом префиксов $v.prefixes$, построенным алгоритмом построения префиксов.*

Доказательство. Обозначим входную эталонную грамматику за G. Пусть p — строка, образованная терминалами на ребрах пути P. Докажем теорему методом математической индукции по l — количеству вершин в пути P.

База $l = 1$. Тогда p — пустая строка. Так как рассматриваемый префикс внутреннего графа корректен для GSS-вершины v, то либо v — начальная вершина GSS v_0, либо существует последовательность сверток длины 0, приводящая парсер из начального состояния $v_0.state$ в состояние $v.state$. В первом случае $v = v_0$, а, значит, среди начальных вершин графа префиксов $v.prefixes$ имеется вершина EOP. Так как путь в графе префиксов $v.prefixes$ состоящий из единственной начальной вершины EOP сводится к пустой строке, то граф префиксов $v.prefixes$ порождает префикс внутреннего графа, соответствующий пути P. Во втором случае в GSS существует путь из вершины v в вершину v_0, состоящий из обнуляемых ребер $(e_1, .., e_a)$. Значит в графе префиксов $v.prefixes$ существует путь $S = (E_1, .., E_a, EOP)$, где E_1 — начальная вершина графа префиксов $v.prefixes$ и $\forall i \in [1, .., a], E_i$ — ассоциируется с обнуляемым ребром e_i. Путь S также сводится к пустой строке, следовательно граф префиксов $v.prefixes$ порождает префикс внутреннего графа, соответствующий пути P. База доказана.

Индукционный переход. Пусть теорема доказана для всех префиксов внутреннего графа, соответствующих путям с количеством вершин не большим k, где $k > 0$. Докажем теорему для префикса внутреннего графа, соответствующего пути $P = (V_1, .., V_{k+1})$.

Так как рассматриваемый префикс внутреннего графа корректен для GSS-вершины v, то непустая строка p — корректный префикс для грамматики G. Значит, получив на вход строку p, алгоритм RNGLR обработает все терминалы этой строки и построит GSS, имеющий вершины $(v_1, .., v_n)$ на последнем уровне. Причем, $\exists i : v_i.state = v.state$. По построению GSS существует хотя бы один путь из вершины v_i в начальную GSS-вершину v_0. Пусть $Q = (w_1, .., w_m)$ — один из таких путей, где $w_1 = v_i$, а $w_m = v_0$. Рассмотрим ребро $e = (w_1, w_2)$. Ребро e может быть терминальным, нетерминальным или обнуляемым.

Если e — терминальное ребро, то пусть оно ассоциировано с терминалом t, нагруженным на ребро (V_k, V_{k+1}) внутреннего графа. Префикс внутреннего графа, соответствующий пути $P' = (V_1, .., V_k)$, корректен в силу корректности префикса, соответсвующего пути P. То есть в вершине внутреннего графа V_{k+1} существуют вершины GSS $(u_1, .., u_b)$, для которых префикс внутреннего графа, соответствующий P', корректен. Тогда $\exists j \in [1, .., b] : u_j.state = w_2.state$. По индукционному предположению префикс внутреннего графа, соответствующий пути P', порождается графом префиксов $u_j.prefixes$. Следовательно существует путь $R = (R_1, .., R_c, EOP)$ графа префиксов $u_j.prefixes$, где R_1 — начальная вершина графа префиксов, такой, что путь R сводится к строке p', где p' — строка p без последнего терминала t. Так как в построенном RNGLR-алгоритмом GSS присутствует ребро $e = (w_1, w_2)$, то в управляющих таблицах, соответствующих грамматике G, существует операция сдвига по терминалу t из состояния $w_2.state$ в состояние $w_1.state$. Значит при обработке основным анализом GSS-вершины u_j будет добавлено ребро $e' = (v, u_j)$, т.к. $u_j.state = w_2.state$ и $v.state = w_1.state$. Значит в граф префиксов $v.prefixes$ будет добавлена начальная вершина E', соответствующая терминальному ребру $e' = (v, u_j)$ и соединенная с графом префиксов $u_j.prefixes$. Поэтому в графе префиксов $v.prefixes$ существует путь $R' = (E', R_1, .., Rc, EOP)$, который сводится к строке p. Таким образом, граф префиксов $v.prefixes$ порождает префикс внутреннего графа, соответствующий пути P.

Если e — нетерминальное ребро, то пусть оно ассоциировано с нетерминалом N и путями GSS $(P_1, .., P_d)$. Пусть в GSS, сконструированным RNGLR-алгоритмом, вершина w_2 была создана после обработки первых $s < k$ терминалов строки p. Обозначим p_1 — строка, составленная из первых s терминалов строки p, а p_2 — строка, составленная из последних $(k-s)$ терминалов строки p. Так как префикс, соответствующий пути $P_s = (V_1, .., V_{s+1})$ — корректен, то в вершине V_{s+1} существует GSS-вершина h такая, что этот префикс корректен для GSS-вершины h и $h.state = w_2.state$. Количество вершин пути P_s равно $(s+1)$, что не превышает k, поэтому по индукционному предположению префикс внутреннего графа, соответствующий пути P_s порождается графом префиксов $h.prefixes$. Значит существует путь $R = (R_1, .., R_c, EOP)$ в графе префиксов $h.prefixes$, где R_1 – начальная вершина графа префиксов, такой, что путь R сводится к строке p_1. В GSS, построенном RNGLR-алгоритмом, имеется нетерминальное ребро e, значит в GSS, построенным основным анализом, имеется нетерминальное ребро $e' = (v, h)$, ассоциированное с нетерминалом N, так как $h.state = w_2.state$ и $v.state = w_1.state$. Значит в граф префиксов $v.prefixes$ будет добавлена вершина E', ассоциированная с нетерминальным ребром e'. Среди путей GSS, ассоциированных с ребром e' будут также присутствовать пути, аналогичные путям $(P_1, .., P_d)$ (последовательности состояний вершин GSS в таких путях совпадают, соответствующие ребра будут иметь одинаковый вид, соответствующие терминальные ребра будут ассоци-

рованы с одинаковыми терминалами, а соответствующие нетерминальные ребра — с одинаковыми нетерминалами). Так как в GSS, построенном RNGLR-алгоритмом, нет циклов (кроме циклов, состоящих только из обнуляемых ребер), а количество ребер конечно, то из нетерминального ребра e с помощью конечного числа раскрытий можно получить последовательность терминальных ребер, соответствующих строке p_2. Значит и в GSS, построенном основным анализом, можно с помощью аналогичного конечного числа раскрытий из ребра e', получить последовательность терминальных ребер, соответствующих строке p_2. Таким образом, в графе префиксов $v.prefixes$ существует путь $(E', R_1, .., R_c, EOP)$, который сводится к строке p. Следовательно граф префиксов $v.prefixes$ порождает префикс внутреннего графа, соответствующий пути P.

Если e — обнуляемое ребро, обозначим $e_f = (w_f, w_{f+1})$ — первое не обнуляемое ребро в пути Q. Префикс внутреннего графа, соответствующий пути P, является корректным для GSS-вершины w_f. Граф префиксов $w_f.prefixes$ порождает префикс внутреннего графа, соответствующий пути P, так как в GSS, построенном RNGLR-алгоритмом, существует путь из вершины w_f, который начинается с терминального или нетерминального ребра, а такие случаи уже рассмотрены. Пусть $(F_1, .., F_g, EOP)$ — путь графа префиксов $w_f.prefixes$, который сводится к строке p. Тогда $\exists (Z_1, .., Z_{f-1}, F_1, .., F_g, EOP)$ — путь графа префиксов $w_1.prefixes$, где $\forall j \in [1, .., (f-1)], Z_j$ ассоциируется с обнуляемым ребром (w_j, w_{j+1}). Значит этот путь также сводится к строке p. А так как $w_1.state = v.state$, то $w_1 = v$. Следовательно граф префиксов $v.prefixes$ порождает префикс внутреннего графа, соотвествующий пути P. □

Для доказательства того, что каждый префикс внутреннего графа, порождаемый графом префиксов $v.prefixes$, является корректным для GSS-вершины v нам понадобится следующая лемма.

Лемма. *Пусть GSS-вершина v_m корректна для префикса внутреннего графа, соответствующего пути $P_1 = (V_1, .., V_l)$, терминалы на ребрах которого образуют строку $p_1 = (t_1, .., t_{l-1})$ ($p_1 = \epsilon$, если $l = 1$). Пусть $\exists Q = (v_1, .., v_m)$ — путь в GSS, в котором $E = (e_1, .., e_{m-1})$ — последовательность ребер, сводящаяся за конечное число раскрытий к строке $p_2 = (t_l, .., t_{l+r-1})$ ($p_2 = \epsilon$, если $r = 0$). Тогда префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_l, V_{l+1}, .., V_{l+r})$ корректен для GSS-вершины v_1, и терминалы на ребрах пути P образуют строку $p = p_1 \cdot p_2$.*

Доказательство. Докажем лемму методом математической индукции по длине строки p_2 равной r.

База $r = 0$ или $r = 1$. Так как GSS-вершина v_m корректна для префикса внутреннего графа, соответствующего пути P, то RNGLR-алгоритм, получив на вход строку p_1, обработает все терминалы этой строки и на последнем уровне построенного GSS, создаст вершину w_m такую, что $w_m.state = v_m.state$.

Если $r = 0$, то строка $p_2 = \epsilon$, следовательно $\forall i, e_i$ — обнуляемо. Следовательно $\forall i, v_i$ ассоциировано с V_l. Таким образом, существует последовательность операций сверток длины 0, приводящих анализатор из состояния $v_m.state$ в состояние $v_1.state$. Следовательно в GSS, построенном RNGLR-алгоритмом, в результате применения аналогичной последовательности операций сверток длины 0, на последнем уровне будет существовать путь из обнуляемых ребер, начинающийся в GSS-вершине w_1 и заканчивающийся в вершине w_m, где $w_1.state = v_1.state$. Из существования на последнем уровне GSS, построенного RNGLR-алгоритмом, вершины w_1 следует, что префикс внутреннего графа, соответствующий пути P_1 также является корректным и для GSS-вершины v_1. А так как в данном случае путь P_1 равен пути P, то утверждение леммы при $r = 0$ доказано.

Если $r = 1$, то строка p_2 — не пустая, а значит среди ребер в последовательности E существует ровно одно не обнуляемое ребро. Пусть j такое, что $e_j = (v_j, v_{j+1})$ — не обнуляемое ребро последовательности E. Если $j = (m-1)$, то префикс внутреннего графа, соответствующий пути P_1, корректен для GSS-вершины v_{j+1}, так как $v_{j+1} = v_m$. Если $j < (m-1)$, то существует путь в GSS из обнуляемых ребер, начинающийся в вершине v_{j+1} и заканчивающийся в вершине v_m. Последовательность ребер в данном пути сводится к пустой строке. Так как утверждение леммы при $r = 0$ было доказано, то префикс внутреннего графа, соответствующий пути P_1, является корректным для GSS-вершины v_{j+1}. Теперь рассмотрим не обнуляемое ребро e_j. Если e_j — терминальное ребро, то оно ассоциировано с терминалом t_l. Тогда в управляющих таблицах существует операция сдвига по терминалу t_l из состояния $v_{j+1}.state$ в состояние $v_j.state$. Значит, RNGLR-алгоритм, обработав строку p_1, может выполнить операцию сдвига из GSS-вершины с состоянием $v_{j+1}.state$ в вершину с состоянием $v_j.state$. Значит префикс внутреннего графа, соответствующий пути $(V_1, .., V_l, V_{l+1})$, является корректным для GSS-вершины v_j. Если $j = 1$, то $v_j = v_1$ и утверждение леммы доказано. Пусть $j > 1$. Последовательность ребер $(e_1, .., e_{j-1})$ за конечное число раскрытий сводится к пустой строке. Так как случай при $r = 0$ был рассмотрен, то доказано, что префикс внутреннего графа, соответствующий пути P, корректен для GSS-вершины v_1. Теперь пусть e_j — нетерминальное ребро. Покажем, что префикс внутреннего графа, соответствующий пути P, также является корректным и для GSS-вершины v_j.

Рассмотрим ту конечную последовательность раскрытий, примененную к последовательности ребер E, после которой останется единственное терминальное ребро, ассоциированное с терминалом t_l. Существует два варианта: либо после первого применения операции раскрытия в последовательности останется единственное терминальное ребро $e_{term} = (u_{term}, v_{j+1})$, ассоциированное с терминалом t_l, либо при применении первых $i > 0$ операций раскрытия последовательность ребер GSS будет состоять из единственного нетерминального ребра, причем $\forall d \in [1, .., i]$, после применения

первых d операций раскрытия последовательность состоит из единственного ребра $S_d = (h_d, g_d)$. Первый вариант означает, что префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_l, V_{l+1})$, является корректным для GSS-вершины u_{term}. А так как в GSS, построенным основным анализом, существует ребро $e_j = (v_j, v_{j+1})$, ассоциированное с нетерминалом N и с GSS путем, в котором присутствует единственное ребро e_{term}, то префикс внутреннего графа, соответствующий пути P, также является корректным и для GSS-вершины v_j. Рассмотрим второй вариант. Пусть $e_j = S_0 = (h_0, g_0)$, тогда $\forall d \in [1, .., i]$, при применении d-ой операции раскрытия нетерминальное ребро S_{d-1} заменяется на последовательность ребер, соответствующих одному из порожденных ребром S_{d-1} пути, начинающегося в вершине w_d и закачивающегося в вершине g_{d-1}, причем w_d, как и v_j принадлежит вершине внутреннего графа V_{l+1}. Пусть $w_0 = v_j$. Так как в данных путях ребро S_d является единственным не обнуляемым ребром, то $\forall d \in [1, .., i]$, существует путь из вершины w_d в вершину h_d и существует путь из вершины g_d в вершину g_{d-1}, оба из которых имеют только обнуляемые ребра. При применении $(i + 1)$-ой операции раскрытия нетерминальное ребро S_i заменятся на последовательность ребер, соответствующую пути в GSS из вершины w_{term} в вершину g_i, причем в этом пути единственное не обнуляемое ребро — это терминальное ребро $S_{term} = (h_{term}, g_{term})$, ассоциированное с терминалом t_l. Так как $\forall d \in [1, .., i]$, существует путь из вершины g_d в вершину g_{d-1}, состоящий только из обнуляемых ребер, и существует путь из вершины g_{term} в вершину g_i, состоящий только из обнуляемых ребер, то также существует путь из вершины g_{term} в вершину $g_0 = v_{j+1}$ и существует путь из вершины g_d в вершину $g_0 = v_{j+1}$, состоящие только из обнуляемых ребер. А так как префикс внутреннего графа, соответствующий пути P_1, является корректным для GSS-вершины v_{j+1}, то префикс внутреннего графа, соответствующий пути P_1, также является корректным и для GSS-вершин из множества $\{g_1, .., g_i, g_{term}\}$. Так как в GSS, построенным основным анализом, существует терминальное ребро $S_{term} = (h_{term}, g_{term})$, ассоциированное с терминалом t_l, то префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_l, V_{l+1})$, является корректным для GSS-вершины h_{term}. А так как существует путь из вершины w_{term} в вершину h_{term}, то префикс внутреннего графа, соответствующий пути P, также является корректным и для GSS-вершины w_{term}. Аналогичным образом доказывается, что $\forall d \in [0, ..i]$, префикс внутреннего графа, соответствующий пути P, является корректным для GSS-вершины w_d. А так как $w_0 = v_j$, то, в частности, префикс внутреннего графа, соответствующий пути P, является корректным для GSS-вершины v_j.

Таким образом показали, что префикс внутреннего графа, соответствующий пути P, является корректным для GSS-вершины v_j. А так как существует путь из вершины v_1 в вершину v_j, состоящий из обнуляемых ребер, то из доказательства утверждения леммы при $r = 0$ следует, что префикс внутреннего графа, соответствующий пути P, также является корректным и для GSS-вершины v_1, что и доказывает утверждение

леммы при $r = 1$. База индукции доказана.

Индукционный переход. Пусть утверждение леммы доказано для всех строк p_2 длины меньшей r. Докажем утверждение леммы для строки p_2 длины $r > 1$.

Так как строка p_2 не пустая, то среди ребер в последовательности E существует хотя бы одно не обнуляемое ребро. Пусть j такое, что $e_j = (v_j, v_{j+1})$ — последнее не обнуляемое ребро последовательности E. Если $j = (m-1)$, то префикс внутреннего графа, соответствующий пути P_1, корректен для GSS-вершины v_{j+1}, так как $v_{j+1} = v_m$. Если $j < (m-1)$, то существует путь в GSS из обнуляемых ребер, начинающийся в вершине v_{j+1} и заканчивающийся в вершине v_m. Последовательность ребер в данном пути сводится к пустой строке. Значит, по индукционному предположению префикс внутреннего графа, соответствующий пути P_1, является корректным для GSS-вершины v_{j+1}. Теперь рассмотрим не обнуляемое ребро e_j.

Если e_j — терминальное ребро, то оно ассоциировано с терминалом t_l. Тогда в управляющих таблицах существует операция сдвига по терминалу t_l из состояния $v_{j+1}.state$ в состояние $v_j.state$. Значит, RNGLR-алгоритм, обработав строку p_1, может выполнить операцию сдвига из GSS-вершины с состоянием $v_{j+1}.state$ в вершину с состоянием $v_j.state$. Значит префикс внутреннего графа, соответствующий пути $(V_1, .., V_l, V_{l+1})$, является корректным для GSS-вершины v_j. Так как строка p_2 имеет длину $r > 1$, то $j > 1$. Последовательность ребер $(e_1, .., e_{j-1})$ за конечное число раскрытий сводится к строке $p_j = (t_{l+1}, .., t_{l+r-1})$. Так как длина строки p_j равна $(r-1)$, то по индукционному предположению префикс внутреннего графа, соответствующий пути P, корректен для GSS-вершины v_1. И утверждение леммы доказано.

Если e_j — нетерминальное ребро. Тогда рассмотрим два случая.

Первый случай: $\exists i \in [1, .., (j-1)], e_i$ — не обнуляемое ребро. Тогда ребро e_j в последовательности E после рассматриваемого конечного числа раскрытий сводится к непустой строке $(t_l, .., t_{l+h-1})$, длины $h < r$. А так как префикс внутреннего графа, соответствующий пути P_1, является корректным для GSS-вершины v_{j+1}, то, по индукционному предположению, префикс внутреннего графа, соответствующий пути $(V_1, .., V_l, V_{l+1}, .., V_{l+h})$, является корректным для GSS-вершины v_j. А так как последовательность ребер $(e_{j-1}, .., e_1)$ за конечное число раскрытий сводится к строке $(t_{l+h}, .., t_{l+r-1})$ длины $(r-h) < r$, то, по индукционному предположению, префикс внутреннего графа, соответствующий пути P, является корректным для GSS-вершины v_1, что и доказывает утверждение леммы.

Второй случай: $\forall i \in [1, .., (j-1)], e_i$ — обнуляемое ребро. Покажем, что префикс внутреннего графа, соответствующий пути P, также является корректным и для GSS-вершины v_j.

В данном случае e_j — единственное не обнуляемое ребро в последовательности E. Рассмотрим ту конечную последовательность раскрытий, примененную к последовательности ребер E, после которой остаются только терминальные ребра, причем

строка, образованная инвертированием этой последовательности терминальных ребер и заменой их на терминалы, с которыми они ассоциированы, является строкой $p_2 = (t_l, .., t_{l+r-1})$. Существует два варианта: либо после первого применения операции раскрытия в последовательности будет более одного не обнуляемого ребра, либо при применении первых $i > 0$ операций раскрытия последовательность ребер GSS будет состоять из единственного нетерминального ребра, причем $\forall d \in [1, .., i]$, после применения первых d операций раскрытия последовательность состоит из единственного ребра $S_d = (h_d, g_d)$. Первый вариант означает, что нетерминальное ребро e_j ассоциировано с путем $F = (x_1, .., x_n)$, где GSS-вершина x_1, как и вершина v_j, принадлежит вершине внутреннего графа V_{l+r}, а GSS-вершина $x_n = v_{j+1}$. Причем в пути F имеется более одного не обнуляемого ребра. Найдем минимальное y такое, что ребро $e_{first} = (x_y, x_{y+1})$ является не обнуляемым. Так как в пути F имеется более одного не обнуляемого ребра, то последовательность ребер $F_1 = ((x_1, x_2), .., (x_y, x_{y+1}))$, как и последовательность ребер $F_2 = ((x_{y+1}, x_{y+2}), .., (x_{n-1}, x_n))$ после применения конечного числа раскрытий сводится к строке длины меньшей r. Пусть последовательности ребер F_1 и F_2 сводятся за рассматриваемое конечное число раскрытий к строкам $f_1 = (t_{l+c}, .., t_{l+r-1})$ и $f_2 = (t_l, .., t_{l+c-1})$, где $0 < c < r$. Так как $x_n = v_{j+1}$, то по индукционному предположению префикс внутреннего графа, соответствующий пути $(V_1, .., V_{l+c})$, является корректным для GSS-вершины x_{y+1}. Это, в свою очередь, означает, что префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_{l+r})$, является корректным для GSS-вершины x_1. А так как в GSS, построенным основным анализом, существует ребро $e_j = (v_j, v_{j+1})$, ассоциированное с GSS путем F, то префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_{l+r})$, является также корректным и для GSS-вершины v_j. Рассмотрим второй вариант. Аналогично рассуждениям в базе индукции при $r = 1$, конструируем множества $\{S_0, .., S_i\}, \{w_0, .., w_i\}, \{h_0, .., h_i\}, \{g_0, .., g_i\}$. При применении $(i+1)$-ой операции раскрытия нетерминальное ребро S_i заменятся на последовательность ребер, соответствующую пути R в GSS из вершины r_1 в вершину g_i, причем в пути R существует более одного не обнуляемого ребра. GSS-вершина r_1, как и вершина v_j, принадлежит вершине внутреннего графа V_{l+r}. Применив к пути R, рассуждения, аналогичные рассуждениям при рассмотрении пути F, получим, что префикс внутреннего графа, соответствующий пути P, является корректным для любой GSS-вершины из множества $\{w_0, .., w_i, r_1\}$, а так как $w_0 = v_j$, то, в частности, префикс внутреннего графа, соответствующий пути P, является корректным для GSS-вершины v_j.

Таким образом показали, что префикс внутреннего графа, соответствующий пути P, является корректным для GSS-вершины v_j. А так как существует путь из вершины v_1 в вершину v_j, состоящий из обнуляемых ребер, то по индукционному предположению префикс внутреннего графа, соответствующий пути P, также является корректным и для GSS-вершины v_1, что и доказывает утверждение леммы. □

Теорема 2. *Каждый префикс внутреннего графа, соответствующий пути* $P = (V_1, .., V_l)$ *внутреннего графа и порождаемый графом префиксов* $v.prefixes$*, является корректным для GSS-вершины* v.

Доказательство. Пусть p — строка, образованная из терминалов на ребрах пути P. Длина строки p равна $(l - 1)$. Так как префикс внутреннего графа, соответствующий пути P, порождается графом префиксов $v.prefixes$, то в этом графе префиксов существует путь X из одной из начальных вершин графа префиксов $v.prefixes$ в вершину EOP, который сводится к строке p.

Если путь X состоит из единственной вершины EOP, то она является начальной вершиной графа префиксов $v.prefixes$, а значит v — начальная вершина GSS. А так как путь X в графе префиксов $v.prefixes$ сводится к пустой строке, то строка p — пустая, и путь P состоит из единственной начальной вершины внутреннего графа V_1. Значит префикс внутреннего графа, соответствующий пути P, корректен для начальной вершины GSS, которой является вершина v.

Пусть путь $X = (E_1, .., E_m, EOP)$, где $m > 0$ и E_1 — одна из начальных вершин графа префиксов $v.prefixes$. Из построения графов префиксов следует, что существует путь из GSS-вершины v в начальную GSS-вершину v_0. Причем последовательность ребер в данном пути после конечного числа раскрытий сводится к строке p. А так как префикс внутреннего графа, соответствующий пути $F = (V_1)$, корректен для начальной GSS-вершины v_0 и существует путь из GSS-вершины v в GSS-вершину v_0, в котором последовательность ребер сводится за конечное число раскрытий к строке p, то, по лемме, префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_l)$ является корректным для GSS-вершины v. □

5.2. Корректность алгоритма диагностики ошибок

В данном разделе приведено доказательство того, что любое ребро e внутреннего графа из множества $errors$ является ошибочными, а все ошибочные ребра принадлежат множеству $errors \cup probErrors$. Также будет доказано, что любой префикс внутреннего графа, порождаемый графом префиксов из множества $errors[e]$, является корректным, но при добавлении к данному префиксу в конец ребра e образуется некорректный префикс внутреннего графа, а все такие префиксы порождаются хотя бы одним графом префиксов из множества $errors[e] \cup probErrors[e]$.

Теорема 3. *После работы алгоритма диагностики ошибок любое ребро* e *внутреннего графа из множества* $errors$ *является ошибочными. А любой префикс внутреннего графа, порождаемый графом префиксов из множества* $errors[e]$*, является корректным, но при добавлении к данному префиксу в конец ребра* e *образуется некорректный префикс внутреннего графа.*

Доказательство. Докажем, что ребро e является ошибочным, методом от про-

тивного. Пусть ребро внутреннего графа $e \in errors$ не является ошибочным. Пусть ребро e нагружено терминалом t, исходит из вершины внутреннего графа V_l и входит в вершину V_{l+1}. Обозначим входную эталонную грамматику за G. Рассмотрим два случая.

Если V_{l+1} — не конечная вершина внутреннего графа. Пусть $(e_1, .., e_n)$ — терминальные ребра GSS, ассоциированные с терминалом t, причем $\forall i \in [1, .., n], e_i = (x_i, y_i)$, где x_i принадлежит вершине V_{l+1}, а y_i принадлежит вершине V_l. Пусть GSS-вершина v, принадлежащая вершине внутреннего графа V_l, принадлежит множеству Z, если $\forall i \in [1, .., n], v \neq y_i$. Так как $e \in errors$, то $\exists v \in Z$ — GSS-вершина, такая, что $v.prefixes$ порождает некоторый префикс внутреннего графа, который не порождается графом префиксов $y_i.prefixes, \forall i \in [1, .., n]$. Множество всех таких префиксов внутреннего графа обозначим K. Пусть один из префиксов внутреннего графа множества K соответствует пути $P = (V_1, .., V_l)$. Пусть p — строка, образованная последовательностью терминалов на пути P. Так как $v.prefixes$ порождает префикс внутреннего графа, соответствующий пути P, то, по ТЕОРЕМЕ 2, получаем, что этот префикс является корректным для GSS-вершины v. Значит строка p является корректным префиксом для грамматики G. Пусть Q — множество GSS-вершин, для которых префикс внутреннего графа, соответствующий пути P, является корректным. Так как рассматриваемый префикс внутреннего графа не порождается графом префиксов $y_i.prefixes, \forall i \in [1, .., n]$, то, по ТЕОРЕМЕ 1, данный префикс не является корректным для GSS-вершины $y_i, \forall i \in [1, .., n]$. Значит, $\forall i \in [1, .., n], y_i \notin Q$. Поэтому $\forall q \in Q$, в управляющих таблицах не существует операции сдвига из состояния $q.state$ по терминалу t. Значит, RNGLR-алгоритм, обработав все терминалы строки p и получив следующим терминалом входной строки — терминал t, не сможет сделать ни одной операции сдвига по данному терминалу. Таким образом, префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_l, V_{l+1})$, не является корректным. Значит, ребро e является ошибочным. Получили противоречие.

Если V_{l+1} — конечная вершина внутреннего графа, тогда t является специальным терминалом EOF конца строки. Пусть A_1 — множество GSS-вершин, принадлежащих вершине V_l, и имеющих принимающее состояние, а A_2 — множество GSS-вершин, принадлежащих вершине V_l, и имеющих непринимающее состояние. Так как $e \in errors$, то существует GSS-вершина $v \in A_2$ такая, что граф префиксов $v.prefixes$ порождает некоторый корректный префикс внутреннего графа, который не порождается ни одним графом префиксов GSS-вершин из множества A_1. Множество всех таких префиксов внутреннего графа обозначим L. Пусть один из префиксов внутреннего графа множества L соответствует пути $F = (V_1, .., V_l)$. Пусть f — строка, образованная последовательностью терминалов на пути F. По ТЕОРЕМЕ 2, данный префикс внутреннего графа является корректным для GSS-вершины v, так как он порождается графом префиксов $v.prefixes$. Значит, префикс внутреннего графа, соответствующий пути

F, является корректным. А так как графы префиксов GSS-вершин из множества A_1 не порождают префикс внутреннего графа, соответствующий пути F, то, по ТЕОРЕМЕ 1, рассматриваемый префикс внутреннего графа не является корректным ни для одной GSS-вершины из множества A_1. Значит, RNGLR-алгоритм, обработав все терминалы строки f, на последнем уровне GSS не будет иметь ни одной вершины с принимающим состоянием. Значит, строка f не является принимаемой грамматикой G. Таким образом, ребро e является ошибочным. Получили противоречие.

В случае, когда V_{l+1} не является конечной вершиной внутреннего графа, префикс внутреннего графа, соответствующий пути $P = (V_1, .., V_l)$, взят из множества K произвольным образом. Для этого префикса показано, что он является корректным, но при добавлении к данному префиксу в конец ребра e образуется некорректный префикс внутреннего графа. Из построения множества графов префиксов $errors[e]$ следует, что данные графы префиксов порождают префиксы внутреннего графа из множества K и только их. Таким образом, любой префикс внутреннего графа, порождаемый одним из графов префиксов $errors[e]$, является корректным, но при добавлении к нему в конец ребра e образуется некорректный префикс внутреннего графа. Аналогично, данное утверждение доказывается в случае, если V_{l+1} — конечная вершина внутреннего графа с множеством префиксов внутреннего графа равным L. □

ТЕОРЕМА 4. *После работы алгоритма диагностики ошибок любое ошибочное ребро e внутреннего графа, исходящее из вершины V_l и входящее в вершину V_{l+1}, принадлежит объединению множеств $errors$ и $probErrors$. А любой префикс внутреннего графа, заканчивающийся в вершине V_l и становящийся некорректным при добавлении в конец ребра e, порождается хотя бы одним графом префиксов из объединения множеств $errors[e]$ и $probErrors[e]$.*

ДОКАЗАТЕЛЬСТВО. Пусть ребро e нагружено терминалом t. Пусть L — множество всех префиксов внутреннего графа, заканчивающихся в вершине V_l и становящихся некорректным при добавлении в конец ребра e. Так как ребро e является ошибочным, то множество L не пусто. Рассмотрим префикс внутреннего графа из множества L. Пусть этому префиксу соответствует путь внутреннего графа $P = (V_1, .., V_l)$. Пусть p — строка, получающаяся заменой в последовательности ребер в пути P на терминалы, которыми нагружены соответствующие ребра. Так как данный префикс внутреннего графа является корректным, то существует GSS-вершина v, принадлежащая вершине внутреннего графа V_l, такая, что рассматриваемый префикс является корректным для вершины v. По ТЕОРЕМЕ 1, данный префикс внутреннего графа порождается графом префиксов $v.prefixes$. Обозначим G — входная эталонная грамматика. Рассмотрим два случая.

Если V_{l+1} — не конечная вершина внутреннего графа. Пусть $(e_1, .., e_n)$ — терминальные ребра GSS, ассоциированные с терминалом t, причем $\forall i \in [1, .., n], e_i = (x_i, y_i)$, где x_i принадлежит вершине V_{l+1}, а y_i принадлежит вершине V_l. Пусть GSS-вершина

u, принадлежащая вершине внутреннего графа V_l, принадлежит множеству Z, если $\forall i \in [1,..,n], u \neq y_i$. Так как один из префиксов, порождаемых графом префиксов $v.prefixes$, становится некорректным при добавлении в конец ребра e, то $v \in Z$. Если граф префиксов $v.prefixes$ имеет цикл, то ребро $e \in probErrors$, а префикс внутреннего графа, соответствующий пути P, порождается одним из графов префиксов множества $probErrors[e]$. Пусть граф префиксов $v.prefixes$ не имеет циклов. В виду того, что рассматриваемый префикс внутреннего графа становится некорректным при добавлении в конец ребра e, то $\forall i \in [1,..,n], y_i.prefixes$ не порождает данный префикс. Если $\exists i \in [1,..,n]$ такое, что граф префиксов $y_i.prefixes$ имеет цикл, то $e \in probErrors$, а префикс внутреннего графа, соответствующий пути P, порождается одним из графов префиксов множества $probErrors[e]$. Если такого i не существует, то $e \in errors$, а префикс внутреннего графа, соответствующий пути P, порождается одним из графов префиксов множества $errors[e]$. Таким образом, в данном случае, $e \in errors \cup probErrors$, а префикс внутреннего графа, соответствующий пути P, порождается одним из графов префиксов множества $errors[e] \cup probErrors[e]$.

Если V_{l+1} — конечная вершина внутреннего графа, тогда t является специальным терминалом EOF конца строки. Пусть A_1 — множество GSS-вершин, принадлежащих вершине V_l, и имеющих принимающее состояние, а A_2 — множество GSS-вершин, принадлежащих вершине V_l, и имеющих непринимающее состояние. Так как один из префиксов, порождаемых графом префиксов $v.prefixes$, становится некорректным при добавлении в конец ребра e, то $v \in A_2$. Если граф префиксов $v.prefixes$ имеет цикл, то ребро $e \in probErrors$, а префикс внутреннего графа, соответствующий пути P, порождается одним из графов префиксов множества $probErrors[e]$. Пусть граф префиксов $v.prefixes$ не имеет циклов. Ввиду того, что рассматриваемый префикс внутреннего графа становится некорректным при добавлении в конец ребра e, то $\forall z \in A_1, z.prefixes$ не порождает данный префикс. Если $\exists z \in A_1$ такое, что граф префиксов $z.prefixes$ имеет цикл, то $e \in probErrors$, а префикс внутреннего графа, соответствующий пути P, порождается одним из графов префиксов множества $probErrors[e]$. Если такого z не существует, то $e \in errors$, а префикс внутреннего графа, соответствующий пути P, порождается одним из графов префиксов множества $errors[e]$.

Таким образом, во всех случаях, $e \in errors \cup probErrors$, а префикс внутреннего графа, соответствующий пути P, порождается одним из графов префиксов множества $errors[e] \cup probErrors[e]$. Так как префикс внутреннего графа, соответствующий пути P, выбирался из множества L произвольным образом, то утверждение теоремы доказано. □

6. Экспериментальное исследование

Предложенный механизм диагностики ошибок был реализован на платформе .NET как часть проекта YaccConstructor; основным языком разработки являлся F# [17]. Данная реализация является модификацией ранее реализованного в рамках проекта алгоритма ослабленного синтаксического анализа регулярной аппроксимации динамически формируемого выражения.

Модифицированный алгоритм был протестирован на серии тестов с целью проверки работоспособности. Данные тесты проверяли, что алгоритм строит корректные множества *errors* и *probErros* ребер входного графа. Для каждого теста специфицировалась грамматика на языке YARD и в явном виде задавался граф конечного автомата, ребра которого были промаркированы лексемами входной грамматики. Входные графы для данных тестов содержали как ветвления, так и циклы. На всех тестах модифицированный алгоритм корректно строил множества *errors* и *probErros*. Кроме того, на всех тестах, входные графы которых не содержали циклов, модифицированный алгоритм точно определял все ошибочные ребра входного графа, то есть *probErros*, в данном случае, являлось пустым множеством.

Также на нескольких сериях синтетических тестов была протестирована производительность модифицированного алгоритма. 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 входная грамматика, использованная в данных тестах.

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

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

- *height* — количество ветвлений в базовом блоке;
- *length* — максимальное количество повторений базовых блоков;
- *errorBranches* — количество веток в базовом блоке, содержащих ошибочное ребро (на рис. 2 изображен базовый блок без ошибочных ребер, а на рис. 3 — базовый блок с двумя выделенными ошибочными ребрами).

Замеры времени работы алгоритмов проводились на машине со следующими техническими характеристиками: Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz, RAM: 8.0 GB, процессор x64.

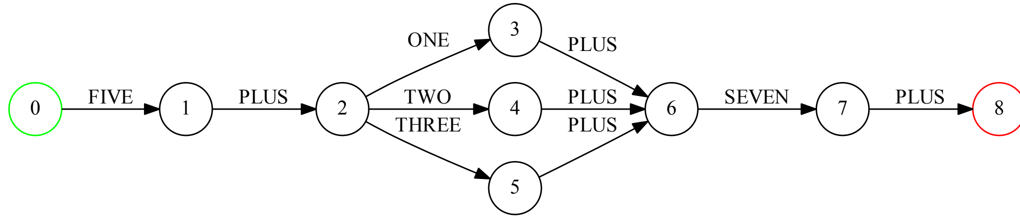

Рис. 2: Базовый блок при $height = 3, errorBranches = 0$

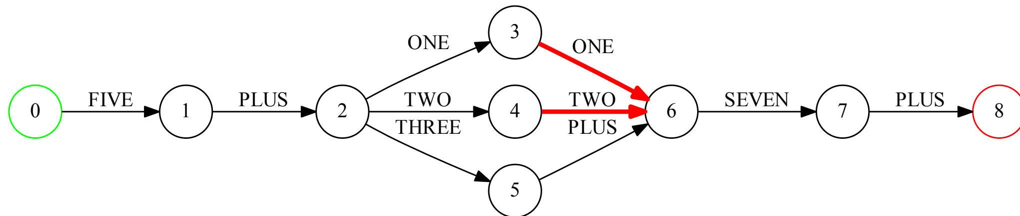

Рис. 3: Базовый блок при $height = 3, errorBranches = 2$

Каждая серия объединяет набор из 50 тестов, каждый из которых содержит одинаковое количество ветвлений в базовом блоке, при этом количество повторений блока совпадает с порядковым номером теста ($length = i$, для теста с номером i). Для каждого теста измерялось время, затраченное на синтаксический анализ. Измерения проводились 10 раз, после чего усреднялись. График, представленный на рис. 4, иллюстрирует сравнение времени работы алгоритма синтаксического анализа до и после модификаций. Можно заметить, что выполненная модификация существенно увеличивает продолжительность анализа. Причина этого в том, что выполненная модификация является прототипом, а в будущем планируется улучшить производительность путем улучшения реализации. График на рис. 5 демонстрирует зависимость времени работы модифицированного алгоритма от количества повторений базового блока и количества веток, содержащих ошибочное ребро, в каждом из них. Наблюдается уменьшение времени работы модифицированного алгоритма при увеличении количества ошибочных ребер. Одной из причин этого является уменьшение количества корректных префиксов внутреннего графа при увеличении количества ошибочных ребер.

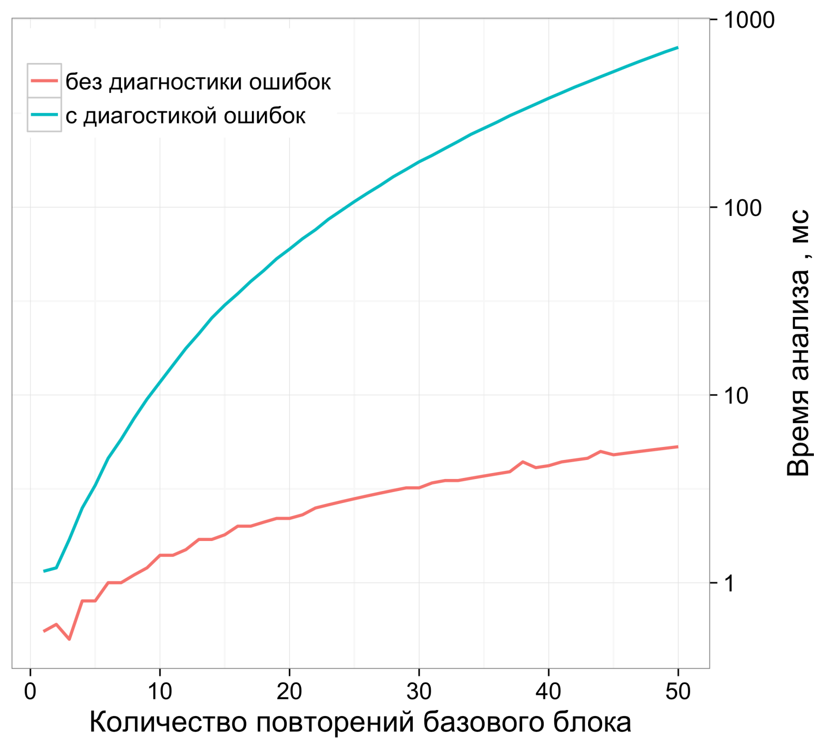

Рис. 4: Сравнение времени работы алгоритма синтаксического анализа до и после модификаций, при $height = 4, errors = 0$

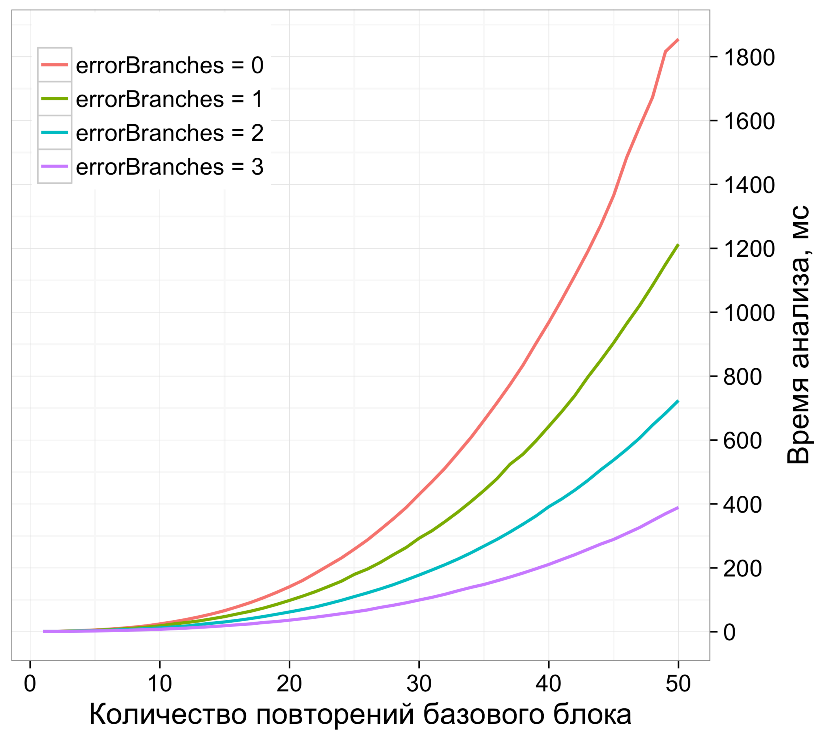

Рис. 5: Зависимость времени работы модифицированного алгоритма от размера входного графа и количества ошибочных ребер при $height = 6$

7. Заключение

В ходе данной работы получены следующие результаты.

- Определено понятие синтаксической ошибки в терминах регулярной аппроксимации динамически формируемого выражения.
- Разработан механизм диагностики ошибок в синтаксическом анализе регулярной аппроксимации динамически формируемого выражения.
- Доказана корректность предложенного механизма.
- Предложенный механизм реализован на языке программирования F# в рамках проекта YaccConstructor.
- Проведено экспериментальное исследование: тестирование работоспособности и тестирование производительности.
- Исходный код проекта YaccConstructor можно найти на сайте `https://github.com/YaccConstructor/YaccConstructor`, автор принимал участие под учётной записью rustam-azimov.

В дальнейшем планируется использовать результат работы модифицированного алгоритма синтаксического анализа для формирования сообщений об обнаруженных ошибках, удобных для пользователя. Кроме того, необходимо произвести теоретическую оценку сложности модифицированного алгоритма.

Список литературы

[1] Alvor [Электронный ресурс]. — URL: `https://bitbucket.org/plas/alvor` (дата обращения: 18.05.2016).

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

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

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

[5] Christensen A. S., Møller A., Schwartzbach M. I. Precise Analysis of String Expressions // Proc. 10th International Static Analysis Symposium (SAS). — Vol. 2694 of LNCS. — Springer-Verlag, 2003. — P. 1–18.

[6] Dick G., Ceriel H. Parsing Techniques: A Practical Guide. — Upper Saddle River, NJ, USA : Ellis Horwood, 1990. — ISBN: 0-13-651431-6.

[7] IntelliLang [Электронный ресурс]. — URL: `https://www.jetbrains.com/idea/help/intellilang.html` (дата обращения: 18.05.2016).

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

[9] Java String Analyzer [Электронный ресурс]. — URL: `http://www.brics.dk/JSA/` (дата обращения: 18.05.2016).

[10] Khabibullin M., Ivanov A., Grigorev S. On Development of Static Analysis Tools for String-Embedded Languages. — 2015. — URL: `https://github.com/YaccConstructor/articles/blob/master/2015/SECR/paper/Main.pdf` (online; accessed: 18.05.2016).

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

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

[13] Nguyen H. V., Kästner C., Nguyen T. 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.

[14] PHPStorm IDE [Электронный ресурс]. — URL: `https://www.jetbrains.com/phpstorm/` (дата обращения: 18.05.2016).

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

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

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

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

[19] Verbitskaia E., Grigorev S., Avdyukhin D. Relaxed Parsing of Regular Approximations of String-Embedded Languages // Preliminary Proceedings of the PSI 2015: 10th International Andrei Ershov Memorial Conference. — PSI'15. — 2015. — P. 1–12.

[20] Вербицкая Е. А. Синтаксический анализ регулярных множеств : Квалификационная работа магистра / Е. А. Вербицкая ; Санкт-Петербургский государственный университет. — 2015.

[21] Григорьев С. В. Синтаксический анализ динамически формируемых программ : Дисс... кандидата наук / С. В. Григорьев ; Санкт-Петербургский государственный университет. — 2015.