# Автоматизированная трансляция динамических SQL-запросов в задачах реинжиниринга

С.В.Григорьев
rsdpisuy@gmail.com

Я.А.Кириленко
jake@math.spbu.ru

Санкт-Петербургский государственный университет
198504, Университетский проспект, 28, Старый Петергоф,
Санкт-Петербург, Россия

В данной работе известный алгоритм абстрактного синтаксического анализа адаптируется для задачи статической трансляции динамических SQL-запросов в задачах реинжиниринга. Описаны необходимые модификации исходного алгоритма, позволяющие учитывать семантику входного языка: изменение процедуры обработки состояний парсера и добавление минимизации результатов синтаксического анализа. Описываются проблемы, возникающие при анализе семантики SQL-предложений, обсуждаются их возможные решения. Представлены результаты апробации описанного алгоритма в промышленном проекте по переносу информационной системы с MS-SQL Server на Oracle.

## Введение

Почти все современные СУБД предоставляют возможность вызова динамических SQL-предложений, которая формально введена в стандарт SQL в 1992 году [11]. Операторы динамического SQL формируются не на этапе компиляции, а позже – на этапе выполнения приложения, как строковые выражения.

Необходимость трансляции [2] динамических запросов очень актуальна в задачах реинжиниринга информационных систем (ИС), при

миграции приложений баз данных и их переносе на другие СУБД [4]. При этом требуется транслировать код хранимых процедур, триггеров, функций с одного диалекта SQL на другой. Если SQL-код содержит динамически формируемые конструкции, то необходимо гарантировать, что после его обработки соответствующие строковые переменные получат корректные значения, из которых в целевой системе будут сформированы корректные SQL-операторы.

Задача трансляции динамических SQL-запросов осложняется тем, что при реинжиниринге информационных систем часто производится переименование таблиц, колонок, процедур и/или их удаление (удаление "мёртвого" кода, удаление неиспользуемых объектов). В этом случае надо гарантировать не только синтаксическую корректность результатов трансляции для динамически формируемых конструкций самих по себе, но и их корректность по отношению к такому переименованию и удалению объектов.

Существующие промышленные инструменты разработки приложений баз данных, такие как PL-SQL Developer [15], SwisSQL [17], SQL Ways [16], предоставляют возможности для трансляции хранимого SQL-кода, но не поддерживают трансляцию динамических запросов. Существует ряд инструментов, например, Java String Analyzer (JSA) [12] или PHP String Analyzer [14], которые предназначены для обработки динамически формируемых строк и выполняют различные проверки этих строк, например, на синтаксическую корректность, но не решают задачу трансляции или других трансформаций динамически формируемых строк.

В рамках данной работы алгоритм абстрактного синтаксического анализа [13] применяется к задаче статической трансляции динамических SQL-запросов. Описаны необходимые модификации исходного алгоритма, позволяющие учитывать семантику входного языка: изменение процедуры обработки состояний парсера и добавление минимизации результатов синтаксического анализа. В статье рассматриваются проблемы, возникающие при работе с семантикой SQL-запросов и предлагаются их возможные решения. В статье также представлены результаты апробации предложенного подхода в промышленном проекте по реинжинирингу информационной системы с MS-SQL Server на Oracle, содержащей более 2 млн. строк хранимого кода и более 3000 динамических SQL-запросов.

# 1 Обзор

## 1.1 Алгоритм абстрактного синтаксического анализа

Для решения задачи трансляции динамических запросов необходимо статически вычислить новые значения для всех переменных, участвующих в формировании динамических запросов. Это должно быть сделано таким образом, чтобы в целевой системе они формировали корректное выражение и не требовали дальнейшей обработки. Для этого необходимо для каждой точки выполнения вычислить все возможные значения, которые может принимать динамически формируемое SQL-предложение, затем провести синтаксический разбор этого множества. В результате разбора получится множество деревьев – лес, над которым нужно провести необходимые преобразования, выполнить трансляцию и на основе полученной информации сформировать новые значения для переменных. Для синтаксического анализа обобщённого представления множества строк, таких как data-flow уравнение и регулярное выражение, существует алгоритм абстрактного синтаксического анализа [13]. Часто удобно считать, что компактное представление анализируемого множества описывается с помощью графа, где на дугах содержаться 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 сформированного запроса может быть представлено графом, показанным на рис. 1.

Алгоритм абстрактного синтаксического анализа основан на идее повторного использования управляющих конструкций путём реализации специального механизма их интерпретации. По спецификации синтаксиса анализируемого языка генерируются управляющие таблицы

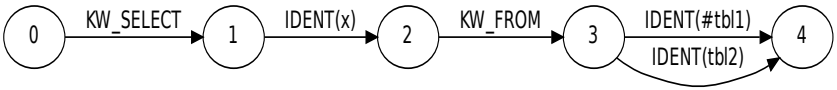


Рис. 1: Пример графа для динамического запроса.

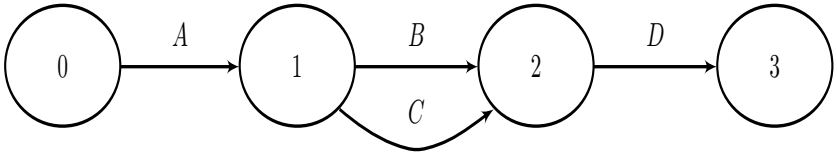


Рис. 2: Пример входного графа для алгоритма абстрактного синтаксического разбора.

для анализатора. На практике для этого можно использовать классические генераторы синтаксических анализаторов, например Yacc [19]. Интерпретатор таблиц (LR-автомат) при этом модифицируется таким образом, чтобы вычислять все возможные состояния синтаксического анализатора для каждой вершины графа [6]. То есть в основе алгоритма лежит обход графа с поиском неподвижной точки. Рассмотрим пример. Пусть задана следующая грамматика:

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

На вход построенному по данной грамматике анализатору подаётся граф, представленный на рис. 2.

Во время синтаксического анализа будет вычислено множество состояния анализатора в каждой вершине графа, как показано на рис. 3.

Один из основных недостатков данного подхода заключается в том, что эта задача в общем случае неразрешима [6]. Поэтому неизбежно

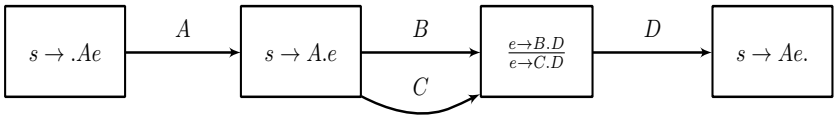


Рис. 3: Состояния парсера для графа, представленного на рис. 2

появляются эвристики, и, как следствие, становится сложно формально гарантировать корректность результата. Кроме этого, при использовании данного подхода в задачах трансляции накладываются дополнительные ограничения на исходный и целевой языки: семантически одинаковые конструкции должны иметь схожую структуру синтаксиса. В противном случае задача не может быть сведена к вычислению новых значений для уже существующих переменных, а необходимость создания новых переменных существенно усложняет решение задачи и дальнейший анализ.

С практической точки зрения важным является тот факт, что данный подход при работе с большими приложениями, имеющими сложные динамические SQL-запросы, очень требователен к вычислительным ресурсам [3], так как в общем случае требует построения всех возможных деревьев разбора, а в сложных системах логика построения запроса может содержать сотни ветвлений, что приводит к экспоненциальному росту размера результирующего леса.

## 1.2 Инструменты для анализа динамически формируемых конструкций

Многие приложения используют при работе базами данных динамически формируемые запросы, что приводит к тому, что для крупных систем затрудняет обеспечение надёжности и безопасности их работы. Поэтому задача статической обработки динамически формируемых строк достаточно актуальна, и одно из её основных направлений – это работа со встроенным SQL: проверка корректности формируемых запросов и защита систем от SQL-инъекций [1] [18]. Также существуют и развиваются инструменты для проверки корректности генерируемого HTML.

Для работы с динамически формируемыми SQL-запросами существует ряд инструментов, которые кратко описаны ниже.

- Java String Analyzer (JSA) [12] – средство для анализа формирования произвольных строк и строковых операций в программах на Java. Для каждого строкового выражения строится конечный автомат, представляющий приближённое значение всех значений этого выражения, которые могут быть получены во время выполнения.

[1] Способ взлома сайтов и программ, работающих с базами данных, основанный на внедрении в запрос произвольного SQL-кода. Данная тематика очень актуальна, в частности, при обеспечении безопасности информационных систем [18].

- Alvor [4, 6] – плагин для среды разработки Eclipse, предназначенный для статической проверки SQL-выражений, встроенных в Java-код. Этот плагин может использоваться как в режиме разового запуска на всём исходном коде, так и виде инкрементального анализатора, который работает в процессе разработки кода. Найденные SQL-запросы проверяются на соответствие SQL-грамматике, также они могут проверяться посредством исполнения в реальной базе данных.

- PHP string analyzer (Phasa) [14] – статический анализатор для строк, порождаемых программами на PHP. Значения таких строк аппроксимируются некоторой контекстно-свободной грамматикой. Инструмент может использоваться, например, для проверки Web-страниц, динамически генерируемых программой на PHP.

- SAFELI [18] – инструмент статического анализа, предназначенный для определения наличия SQL-инъекций в Web-приложениях на этапе компиляции. SAFELI работает с MSIL[2] байт-кодом ASP.NET[3] приложений.

- A Static Analysis Framework for Database Applications [8] – инструмент предназначен для анализа откомпилированого .NET-кода, использующего технологию ADO.NET[4] для доступа к данным. Может применяться для определения мест, где могут появиться SQL-инъекции, а также для определения «узких мест» в производительности SQL-запросов и проверки ограничений на данные.

Необходимо отметить, что ни один из этих инструментов не решает задачу трансляции динамически формируемых строк.

# 2 Модификации алгоритма абстрактного синтаксического анализа

## 2.1 Терминология

В нашем случае структурой данных, которая поступает на вход абстрактному синтаксическому анализатору, будет граф, представляющий результат протягивания констант. Таким образом, каждому пути

[2]Промежуточный язык, разработанный компанией Microsoft для платформы .NET Framework.

[3]Технология создания веб-приложений и веб-сервисов от компании Microsoft.

[4] Часть платформы .NET, предоставляющая доступ к данным для приложений, основанных на Microsoft .NET.

в графе соответствует некоторое возможное значение динамического запроса. Будем говорить, что данный путь порождает соответствующее значение.

Будем говорить также, что запрос содержит лексические или синтаксические ошибки, если в графе, соответствующем этому запросу, существует хотя бы один путь, порождающий значение с лексической или синтаксической ошибкой соответственно.

В рамках решаемой нами задачи основной структурой данных для лексического и синтаксического анализа является граф. Его можно воспринимать как аналог входного потока символов для лексического анализатора и потока токенов для синтаксического. Поэтому, мы будем говорить о токенизации или лексическом анализе графа, подразумевая под этим некоторый процесс, который переводит граф, содержащий строки, в граф, содержащий токены. Аналогично, будем говорить о синтаксическом анализе (разборе) графа, подразумевая процесс, на выходе которого получается некоторое множество синтаксических деревьев или лес. Каждое дерево соответствует некоторому значению запроса, порождённому некоторым путём во входном графе.

## 2.2 Терминология

Ниже изложен ряд принятых нами ограничений на входные данные предлагаемого нами алгоритма, которые далее считаются верными, если не оговорено иное.

- Обрабатываемый граф является DAG[5]-графом. Данное упрощение сделано на основе следующего практического наблюдения: при раскрытии цикла в единственное повторение не нарушается синтаксическая корректность и учитываются все переменные, участвующие в построении запроса. Этого достаточно для того, чтобы решить задачу трансляции. Также это позволяет упростить задачу и вместо поиска неподвижной точки обойти все вершины один раз в порядке N-нумерации[6].

- У DAG-графа одна стартовая и одна конечная вершина. Для конечной вершины верно, что из неё не выходит ни одна дуга. Это

[5]Directed Acyclic Graph (ориентированный ациклический граф) — ориентированный граф, в котором отсутствуют циклы.

[6]Для данной $M$-нумерации (нумерация вершин в порядке их обхода при поиске в глубину) такая нумерация вершин, что для любых вершин $a$ и $b$ неравенство $N(a) < N(b)$ выполняется тогда и только тогда, когда либо вершина $b$ $M$-достижима из a либо $M(b) < M(a)$ и вершина a не является $M$-достижимой из $b$.

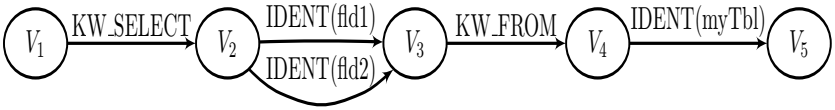


Рис. 4: Граф с потенциально возможным объединением состояний.

достигается явным добавлением дуги с токеном, обозначающим конец ввода (EOF).

- При описании синтаксического анализа и трансляции считаем, что лексический анализ проведён корректно и входной граф не содержит ошибок.

## 2.3 Обработка семантики входного языка

Серьёзным отличием от решаемых ранее задач является необходимость трансляции. При валидации динамически формируемых строк производится только синтаксический анализ, что позволяет объединять состояния парсера как это происходит в алгоритмах GLR-анализа[7]. При трансляции это не возможно, так как неизбежно появляются семантические вычисления и понятие состояния существенно усложняется. В общем случае необходимо вычислить все семантические действия для всех значений динамической строки. Это означает, что если при синтаксическом анализе достаточно учитывать только тип токена, то для трансляции важны и значения токенов. Соответственно, если в вершине $V_3$ на рис. 4 синтаксический анализатор будет находиться в одинаковых состояниях и их можно объединить, то с учётом семантики это два разных состояния, так как значения идентификаторов различаются.

Отметим, что логика формирования динамического запроса может быть очень сложной и содержать порядка сотен операций. При этом система, для которой производится реинжиниринг, может взаимодействовать с другими системами, недоступными в процессе трансляции, которые содержат другие части динамических запросов. Примером такой системы может служить клиентское приложение, которое присылает условия для фильтров (условия where в запросах) в виде частей запросов. Как следствие, не все значения могут быть вычислены на

[7]Generalized Left-to-right Rightmost derivation — расширенный алгоритм LR-анализа, предназначенный для разбора по неоднозначным грамматикам [9].

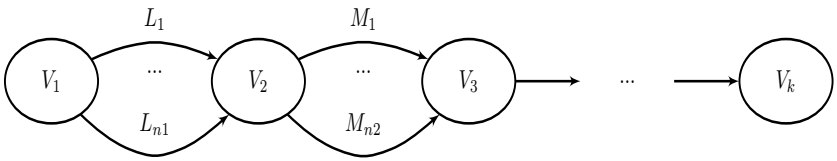


Рис. 5: Стандартный граф, требующий больших ресурсов для обработки.

этапе трансляции. Более того, в общем случае задача статического вычисления всех значений переменных неразрешима, так как, например, множество порождаемых строк может быть бесконечным (такая ситуация может возникнуть при наличии конкатенации в цикле).

Таким образом, серьёзной проблемой являются большие запросы, формируемые с использованием множества условий, которые могут в общем случае порождать при статическом анализе очень много вариантов, что является частой ситуацией. Как следствие, размер леса растёт как показательная функция от числа ветвлений. Условно такая ситуация называется «экспоненциальным взрывом» и подлежит внимательному изучению. В некоторых случаях удаётся эквивалентно переформулировать исходный код, а иногда «вручную» добавить уточнения о заведомо ложных или «несовместных» ветках.

На практике такая ситуация встречается достаточно часто: граф не обязательно должен быть очень сложным, чтобы его обработка потребовала много ресурсов. Для примера рассмотрим граф на рис. 5.

В данном случае для вершины $V_3$ будет получено $N = n_1 \quad n_2$ состояний. Предположим, что $N$ оказалось большим, но все состояния были вычислены. Пусть, также «хвост» графа от вершины $V_3$ до $V_k$ является линейным участком и $k$ достаточно велико. Графы такого вида часто встречаются на практике. При обработке такого графа, при переходе по каждой дуге, начиная с исходящей из $V_3$, необходимо обработать N состояний. В результате требуется обработать $N \quad (k \quad 3)$ состояний, что потребует больших ресурсов.

Таким образом, задача уменьшения количества обрабатываемых состояний является актуальной для систем с большим количеством запросов со сложной логикой построения.

## 2.4 Объединение состояний

Мы предлагаем следующую идею для борьбы с «экспоненциальным взрывом». Результатом трансляции должны стать новые значения

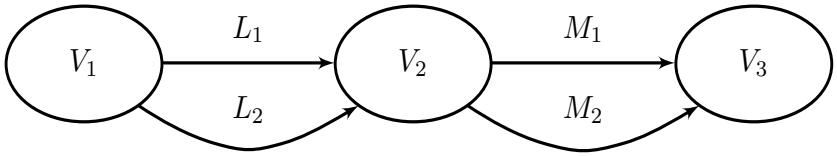


Рис. 6: Граф, требующий решения задачи о выборе корректных путей.

для переменных, участвующих в формировании динамического запроса. Поэтому можно строить не все возможные деревья, а минимальный корректный лес, содержащий все переменные, участвующие в формировании запроса. То есть можно рассматривать не все возможные пути в графе, а только минимальное множество путей, покрывающих все дуги.

Нужно понимать, что нельзя вычислить этот набор путей статически для всего графа, полученного после лексического анализа. Основная проблема заключается в том, что при статическом вычислении заранее не известно, сможет ли данный путь быть корректно разобранным парсером и породить дерево: если путь окажется некорректным, то будет потеряна часть информации. Рассмотрим следующий пример. Пусть абстрактный синтаксический анализ должен обработать следующий граф (рис. 6).

Одно из возможных множеств путей для такого графа при статическом вычислении может быть таким: $(\ _1\quad _1)\ (\ _2\quad _2)$ . Однако, они могут оказаться синтаксически не корректными, и в результате синтаксического анализа не будет получено ни одного дерева. Корректными могут оказаться другие пути. Например, пути $(\ _1\quad _2)\ (\ _2\quad _1)$ . Таким образом, при разборе первого множества не будет получено ни одного дерева, а при разборе второго – два.

Поэтому вычисление набора путей происходит итеративно. Во время синтаксического анализа для каждой вершины, в которую входит более чем одна дуга, выполняется фильтрация состояний. Происходит это следующим образом. Изначально результирующее множество состояний пусто, потом в него постепенно добавляются новые состояния. При этом рассматривается подграф, образованный предками рассматриваемой вершины и множество корректных состояний для текущей вершины. Состояние должно быть добавлено, если истинно одно из следующих условий:

- новому состоянию соответствует путь, который покрывает хотя бы одну дугу, ещё не покрытую путями, соответствующими со-

стояниям, уже добавленным в результирующее множество;

- новое состояние соответствует состоянию парсера, отсутствующему в результирующем множестве, при этом оно может не добавлять новых дуг в покрытие.

Этот алгоритм представлен ниже в виде псевдокода.

```
/*
V - список вершин входного графа в порядке N-нумерации.
v_s - стартовая вершина входного графа.
*/

let filterStates v =
    /* Группируются состояния, соответствующие одинаковым
    состояниям синтаксического анализатора. */
    let groupedByParserState =
        v.States.GroupBy (fun state -> state.Item)

    v.States = Set.empty

    for group in groupedByParserState do
        /* Каждому состоянию соответствует путь от v_s до v.
         Множество путей задаёт множество дуг из исходного
         графа E_s. Нужно построить минимальное множество
         путей, содержащее все дуги из исходного E_s.
         Можно решить эту задачу тривиальным жадным алгоритмом:
         отсортировать пути по убыванию длины и добавлять
         очередной путь в результирующее множество только если
         он содержит дуги, которых ещё нет в результирующем
         множестве. */
        let ordered =
            group.OrderBy (fun s -> -1 * s.Path.Lenght)
        for s in ordered do
            if (s.Path содержит дуги,
                которых нет в ни в одном пути в res)
            then v.States.Add s

for v in V do
    v.States <- ... /*выполнить шаг синтаксического анализа*/
    /*Если в вершину входит более одной дуги,
      то пытаемся отфильтровать состояния.*/
    if v.InEdges.Count > 1 then filterStates v
```

Таким образом получается результирующее множество, которое по мощности не более, чем исходное, содержит состояния, задающие такое же множество состояний парсера, как и исходное множество и соответствующие пути покрывают все дуги в соответствующем подграфе.

Применение такой фильтрации позволяет достичь существенного уменьшения количества деревьев, что, как следствие, позволяет улучшить производительность.

# 3 Апробация

Алгоритм абстрактного синтаксического анализа с описанными модификациями был реализован и применён на практике для перевода приложения, созданного на основе MS-SQL Server 2005, на Oracle 11gR2. Исходная система состояла из 850 хранимых процедур и содержала более 3000 динамических запросов, а в общей сложности 2,7 млн. строк хранимого кода. Более половины динамических запросов были сложными, при их формировании использовалось от 7 до 212 операторов. При этом, среднее количество операторов для формирование запроса равнялось 40.

Первая реализация не учитывала возможность наличия сложных запросов и при запуске на компьютере с 16 Гб оперативной памяти алгоритм «зависал». Для того, чтобы избежать таких ситуаций, был введён таймаут на обработку одного запроса (64 секунды). На практике было выяснено, что при увеличении таймаута количество обработанных запросов не увеличивается. Запросы, обработка которых прерывалась по таймауту, попадали в категорию «экспоненциальный рост размера леса».

В таблице 1 приведена статистика по количеству обработанных динамических SQL-запросов для исходного алгоритма с таймаутом и алгоритма с объединением состояний.

| Описание ситуации | Исходный алгоритм | Алгоритм с объединением состояний |
|---|---|---|
| Всего | 3122 | 3122 |
| Успешно | 2181 | 2253 |
| | | |
| **Частично успешно** | 408 | 522 |
| С лексическими ошибками | 283 | 289 |
| С ошибками парсера | 354 | 468 |
| | | |
| **Не разобраны** | 533 | 347 |
| С лексическими ошибками | 140 | 134 |
| С ошибками парсера | 280 | 305 |
| Экспоненциальный рост размера леса | 253 | 41 |
| | | |
| Процент успешных | 69.86% | 72.17% |
| Процент частично успешных | 13.07% | 16.72% |
| Процент с не пустым лесом | 82.93% | 88.89% |

Таблица 1. Сравнение качества работы исходного алгоритма и алгоритма со слиянием состояний.

К частично успешным запросам относятся запросы, при разборе которых были получены ошибки, но при этом лес оказался не пустым. Это самая сложная для анализа категория, так как ошибка парсера в данном случае может быть «ложной тревогой», поскольку при выполнении такое значение могло никогда не порождаться.

В результате применения предложенных в работе оптимизаций удалось сократить количество необработанных из-за таймаута запросов более чем в 6 раз.

# Заключение

Серьёзной проблемой оказались запросы, при анализе попавшие в категорию частично успешных. Основной вопрос связан с наличием синтаксических ошибок в формируемых запросах в исходной системе. С одной стороны, можно сделать предположение, что все порождаемые значения корректны, так как транслируемая система применяется на практике и её текущее поведение можно считать корректным. В такой ситуации ошибка означает ошибку в нашем инструменте или алгоритме. С другой стороны, это предположение не всегда верно по

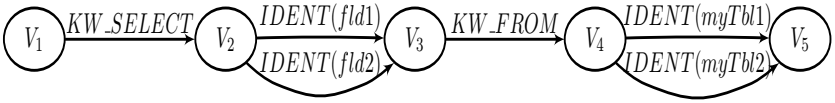


Рис. 7: Пример графа, в котором все пути корректны синтаксически, но не семантически.

двум причинам. Первое – знание семантики разработчиком позволяет ему утверждать, что при выполнении кода некоторые значения динамических запросов никогда не будут получены, однако при статическом анализе они получаются и оказываются некорректными. Второе – в долго живущих системах очень много «мёртвого» кода, который может быть принципиально некорректным. В случае с базами данных ситуация осложняется тем, что они могут хранить данных для тесовых процедур, которые были верны на момент тестирования, но на момент трансляции устарели. Часть таких вопросов может быть снята после удаления «мёртвого» кода. Однако, остаётся большое количество запросов, вопрос о корректности которых необходимо решать «вручную». По этой причине необходимо более подробное изучение вопроса о возможности применения различных методов восстановления после ошибок и автоматической коррекции ошибок [1].

При необходимости учитывать семантику обрабатываемового языка возникает ряд трудностей, основная из которых – отсутствие возможности гарантировать семантическую корректность результата в процессе синтаксического анализа. Можно получить синтаксически корректные деревья, которые не являются корректными с точки зрения семантики. Например, для графа (см. рис. 7) можно выбрать два пути из четырёх. Оба синтаксически корректны и будут содержать все переменные. Однако, у таблицы myTbl1 может не быть поля fld1, а у таблицы myTbl2 – поля fld2.

Так же возникают проблемы, связанные с особенностями синтаксиса входного языка и его спецификации в грамматике. Например, такие подвыражения конструкции Select, как group by, order by могут идти в любом порядке, но не более одного раза. При их описании в грамматике часто делается допущения, избавляющие от необходимости перечислять все варианты перестановок. Однако такая грамматика допускает ещё и некорректные цепочки из нескольких повторений одинаковых подвыражений, например group_by. В хранимом коде такие ситуации невозможны, так как являются некорректными, а при обработке графов возможны ситуации, когда у select-запроса оказывается

несколько group_by, что является ошибкой. Эта проблема решается либо введением конструкции перестановки в язык описания трансляций, что является более общим и предпочтительным решением, либо «ручной» проверкой результирующего леса, что является более трудоёмким и менее предпочтительным.

## Список литературы


[1] *Ефимов А.А., Кириленко Я.А.* Построение ослабленного LALR-транслятора на основе анализа грамматики на избыточность // Системное программирование. Т. 4, вып. 1, 2009. С. 79–103.

[2] *Мартыненко Б.К.* Языки и трансляции. Издательство Санкт-Петербургского университета, 2008. 257 с.

[3] *Мосиенко М.А., Тиунова А.Е.* Интеграция программной логики с пользовательским интерфейсом при реинжиниринге приложений // Системное программирование. Т. 1, вып. 1, 2004. С. 199–224.

[4] *Трошин С.Л.* Преобразование сетевых баз данных в реляционные: задачи и подходы // Системное программирование. Т. 1, вып. 1. 2004. С. 282–310.

[5] *Annamaa A., Breslav A., Kabanov J. e.a.* An interactive tool for analyzing embedded sql queries. Programming Languages and Systems. LNCS, vol. 6461. Springer: Berlin; Heidelberg. 2010. P. 131–138.

[6] *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. Marina Walden and Luigia Petre, editors. 2010. P. 20-22.

[7] Alvor. URL: http://code.google.com/p/alvor/

[8] *Aske Simon Christensen, Mller A., Michael I. Schwartzbach.* Precise analysis of string expressions // Proc. 10th International Static Analysis Symposium (SAS), Vol. 2694 of LNCS. Springer-Verlag: Berlin; Heidelberg, June, 2003. P. 1–18.

[9] *Grune D., Ceriel J. H. Jacobs.* Parsing techniques: a practical guide. Ellis Horwood, Upper Saddle River, NJ, USA, 1990. P. 322.

[10] *Costantini G., Ferrara P., Cortesi F.* Static analysis of string values // Proceedings of the 13th international conference on Formal methods and software engineering, ICFEM'11. Springer-Verlag: Berlin; Heidelberg, 2011. P. 505–521.

[11] ISO. ISO/IEC 9075:1992: Title: Information technology — Database languages — SQL. 1992. P. 668.

[12] Java String Analyzer. URL: http://www.brics.dk/JSA/

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

[14] PHP String Analyzer. URL: http://www.score.is.tsukuba.ac.jp/ minamide/phps

[15] PL/SQL Developer. URL: http://www.allroundautomations.com/plsqldev.html

[16] SQL Ways. URL: http://www.ispirer.com/products

[17] SwissSQL. URL: http://www.swissql.com/

[18] *Xiang Fu, Xin Lu, Peltsverger B. e.a.* A static analysis framework for detecting sql injection vulnerabilities // Proceedings of the 31st Annual International Computer Software and Applications Conference. Vol. 01, COMPSAC'07, Washington, DC, USA, IEEE Computer Society, 2007. P. 87–96.

[19] Yacc. URL: http://dinosaur.compilertools.net/