Math-Net.Ru
Общероссийский математический портал

М. И. Полубелова, С. В. Григорьев, Лексический анализ динамически формируемых строковых выражений, *Системы и средства информ.*, 2016, том 26, выпуск 2, 43–62

Использование Общероссийского математического портала Math-Net.Ru подразумевает, что вы прочитали и согласны с пользовательским соглашением
http://www.mathnet.ru/rus/agreement

Параметры загрузки:
IP: 3.145.191.111
16 октября 2024 г., 12:22:16

ЛЕКСИЧЕСКИЙ АНАЛИЗ ДИНАМИЧЕСКИ ФОРМИРУЕМЫХ СТРОКОВЫХ ВЫРАЖЕНИЙ*

М. И. Полубелова[1], *С. В. Григорьев*[2]

Аннотация: Строковые выражения могут использоваться для формирования и последующего исполнения кода во время выполнения основной программы. Такой подход обладает высокой выразительностью, однако затрудняет разработку, отладку и сопровождение, а также является источником таких уязвимостей, как внедрение SQL (Structural Query Language) и межсайтовый скриптинг. Статический анализ строковых выражений предназначен для борьбы с недостатками подхода посредством проверки того, что все формируемые выражения удовлетворяют некоторым свойствам, без запуска программы. Лексический анализ, или токенизация формируемого кода, является важным шагом такого статического анализа. В статье описан автоматизированный подход к созданию лексических анализаторов динамически формируемого кода, который позволит упростить создание инструментов, предназначенных для статического анализа такого кода.

Ключевые слова: анализ строковых выражений; генератор лексических анализаторов; лексический анализ; встроенные языки

DOI: 10.14357/08696527160203

1 Введение

Многие языки программирования позволяют работать со строковыми выражениями. Последние могут формироваться динамически с использованием строковых операций и языковых конструкций, например условных операторов и циклов. Такие выражения широко используются в программных интерфейсах ODBC (Open DataBase Connectivity), ADO.NET (ActiveX Data Objects.NET) и JDBC (Java DataBase Connectivity), предназначенных для формирования запросов к базе данных на языках программирования C++, C# и Java соответственно, а также в web-программировании. Некоторые примеры использования динамически формируемых строковых выражений представлены в листинге 1.

Динамически формируемые строковые выражения воспринимаются компилятором как обычные строки, что усложняет разработку и сопровождение системы. Во-первых, о наличии ошибок, таких как лексические или синтаксические,

*Статья рекомендована к публикации в журнале Программным комитетом конференции «Tools & Methods of Program Analysis» («Инструменты и методы анализа программ», TMPA-2015), Санкт-Петербург, 12–14 ноября 2015 г.

[1]Санкт-Петербургский государственный университет, polubelovam@gmail.com

[2]Санкт-Петербургский государственный университет, Semen.Grigorev@jetbrains.com

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

Листинг 1 Использование нескольких встроенных в PHP языков

в сформированном выражении становится известно только в момент выполнения программы, когда оно начинает выполняться в своем программном окружении. Во-вторых, при ненадлежащей обработке пользовательского ввода система становится уязвимой, например для SQL-инъекций или межсайтового скриптинга. Указанные проблемы можно решить, включив обработку строковых выражений в статический анализ программы.

Классический подход к статическому анализу заключается в проведении лексического анализа и синтаксического разбора исходного кода. Синтаксический анализ строит структурное представление, которое используется в дальнейшем, например, для семантического анализа или трансформаций, проводимых в контексте реинжиниринга. В рамках данного подхода широко распространен автоматизированный способ создания лексических и синтаксических анализаторов такими генераторами, как Lex, Yacc и их потомками. Использование подобных инструментов сокращает затраты на создание программных продуктов, требующих построения структурного представления кода. Однако существующие генераторы лексических и синтаксических анализаторов не применимы для создания инструментов обработки динамически формируемого кода из-за того, что такой код, как правило, не представим в виде линейного потока, принимаемого на вход классическими анализаторами.

Таким образом, есть необходимость в создании генератора лексических анализаторов для динамически формируемого кода, который предоставляет функциональность, аналогичную классическому. Это позволит упростить создание инструментов, которые предназначены для решения задач, возникающих при обработке динамически формируемого кода. Примерами таких задач являются трансформация запросов с одного языка на другой, возникающая в контексте реинжиниринга информационных систем, подсветка синтаксиса и ошибок в интегрированных средах разработки, а также подсчет различных метрик.

В данной статье описан автоматизированный подход к созданию лексического анализатора для динамически формируемого кода. В рамках работы был разработан алгоритм лексического анализа и генератор лексических анализаторов, за основу которого была взята библиотека FsLex[1]. Указанные компоненты были реализованы как часть проекта YaccConstructor[2], который служит модульным инструментом для проведения лексического анализа и синтаксического разбора, а также платформой для поддержки встроенных языков.

2 Обзор

В данном разделе рассматриваются существующие инструменты, предназначенные для работы с динамически формируемыми выражениями, генератор лексических анализаторов FsLex и проект YaccConstrucor, в котором ведется разработка автоматизированного подхода к созданию лексических анализаторов для динамически формируемого кода.

2.1 Обзор существующих инструментов

Для работы с динамически формируемыми строковыми выражениями существует ряд инструментов. Почти все они предназначены для решения какой-то одной конкретной задачи: либо проверки выражения на соответствие описанию некоторой эталонной грамматики, либо статического анализа программы на уязвимость. Так как генерация всех значений динамически формируемого выражения значительно снижает скорость проведения анализа и возможны ситуации, когда число принимаемых выражением значений может быть бесконечным, то целесообразно иметь конечное представление множества значений данного выражения и уже над ним проводить анализ.

Конечное представление множества значений строкового выражения впервые было использовано в инструменте Java String Analyzer [1]. Этот инструмент предназначен для анализа строк и строковых операций в Java-программе. Результатом этого приближения стал конечный автомат, который используется для проверки включения языков: проверяется включение языка, порождаемого программой, в язык, описанный пользователем. Затем инструмент PHP String Analyzer [2], использовав идею предыдущего инструмента, уточнил проводимую аппроксимацию, результатом которой стала контекстно-свободная грамматика. Этот инструмент применяется для статической валидации HTML-страниц, генерируемых в PHP-программе.

В инструменте Pixy [3], предназначенном для поиска SQL-инъекций и межсайтового скриптинга в PHP-программах, применяется техника path pruning, позволяющая проводить анализ только тех значений строкового выражения,

[1] http://fsprojects.github.io/FsLexYacc/.
[2] https://github.com/YaccConstructor/YaccConstructor.

которые оно может принять в момент выполнения программы. Инструмент Stranger [4] расширяет указанный подход, используя в качестве структурного представления динамически формируемого кода конечный автомат над алфавитом символов обрабатываемого языка. При этом рассматривается общий случай, когда аргументами строковых операций являются конечные автоматы. В инструменте Stranger разработан алгоритм, который вычисляет результат этих операций и возвращает его в виде конечного автомата, что позволяет достичь более высокой точности анализа по сравнению с аналогами.

Разработчики следующего инструмента расширили круг решаемых задач, сформулировав вопросы безопасности, корректности и производительности сформированных запросов с использованием таких программных интерфейсов, как ADO.NET и JDBC. Описание этого инструмента дается в статье [5], в которой также была указана разработанная функциональность: поиск SQL-инъекций, извлечение множества всех значений для строкового выражения, удаление неиспользуемых переменных в формируемом запросе, а также проверка на соответствие типов возвращаемого запросом значения ожидаемым в программе.

Инструмент SAFELI [6], также предназначенный для поиска уязвимостей в веб-приложениях, отличается от рассмотренных тем, что структурным представлением динамически формируемого кода является синтаксическое дерево разбора. Результат получается посредством сопоставления полученного дерева с синтаксическим деревом шаблона уязвимости, параметризованного реальными данными. Однако SAFELI не поддерживает обработку строковых выражений, которые могут быть получены при участии строковых операций.

Для проведения лексического анализа и синтаксического разбора множества значений строкового выражения был разработан инструмент Alvor [7], который является расширением для среды разработки Eclipse, предназначенным для статической валидации SQL-выражений, встроенных в программы на Java. Одна из возможностей инструмента — поиск лексических и синтаксических ошибок, однако поддержка нового языка генерируемого кода является нетривиальной задачей из-за отсутствия генераторов лексических и синтаксических анализаторов. Кроме того, Alvor не поддерживает обработку выражений, полученных с помощью строковых операций (кроме конкатенации) и циклов.

2.2 Инструмент YaccConstructor

YaccConstructor [8] является модульным инструментом с открытым исходном кодом, предназначенным для исследований в области лексического анализа и синтаксического разбора, а также платформой для поддержки встроенных языков [9]. Данный инструмент реализован на платформе Microsoft .NET, основной язык разработки — F#.

Разработанный механизм анализа встроенных языков ранее имел ограничения на структуру динамически формируемого выражения: лексический и синтаксический анализаторы могли обрабатывать только аппроксимацию, представленную

в виде ориентированного ациклического графа. Это не позволяло корректно обрабатывать выражения, полученные с помощью циклов.

В данной работе такое ограничение снимается: лексический анализатор работает с произвольным конечным автоматом над алфавитом символов обрабатываемого языка. Разработанный модуль для лексического анализа, который состоит из генератора лексических анализаторов и интерпретатора, соответствующего предложенному алгоритму лексического анализа, внедрен в инструмент YaccConstructor.

2.3 Генератор лексических анализаторов FsLex

При проведении лексического анализа часто используются генераторы лексических анализаторов, которые по спецификации обрабатываемого языка строят описание конечного преобразователя, на основе которого входной поток символов преобразуется в поток токенов. В качестве инструмента для проведения лексического анализа динамически формируемого кода был выбран генератор лексических анализаторов FsLex. Этот выбор обусловлен тем, что реализованный механизм является компонентом инструмента YaccConstructor, основным языком разработки которого служит язык программирования F#.

Генератор лексических анализаторов на вход принимает файл с расширением .fsl, в котором описана лексическая спецификация языка, формат определения которой представлен в листинге 2.

Результатом работы генератора является файл с расширением .fs с кодом F# для лексического анализатора. Этот файл содержит код, указанный в заголовке спецификации, конечный преобразователь и функции, которые были созданы на каждую точку вхождения, а также вызов функции интерпретатора

```
{
module Lexer
// header: any valid F# code can appear here
open Parser //specifies type of tokens
}
// regex macros
let ident = regexp ...
// rules
rule entrypoint = parse
| regexp { action }
| ...
and entrypoint = parse
...
```

Листинг 2 Формат определения спецификации для языка

построенного конечного преобразователя. Чтобы использовать такой лексический анализатор, полученный файл вместе с описанием типов токенов, которые автоматически строятся по грамматике эталонного языка, необходимо добавить в модуль, предназначенный для лексического анализа.

3 Лексический анализ

Основная задача лексического анализа — преобразование входного потока символов в поток токенов, соответствующих спецификации обрабатываемого языка, и сохранение привязки лексических единиц к исходному коду. В классическом случае входной поток является линейным. Для проведения лексического анализа динамически формируемого выражения необходима структура, которая выступает конечным представлением множества значений этого выражения. Для построения аппроксимации используется алгоритм, предложенный в статье [10], поэтому такой структурой служит конечный автомат над алфавитом символов обрабатываемого языка.

Результатом работы лексического анализа динамически формируемого строкового выражения является конечный автомат над алфавитом токенов эталонной грамматики языка. В классическом лексическом анализе токен можно представить в виде структуры, содержащей идентификатор токена и последовательность символов, выделенных из входного потока. В контексте данной статьи токен представляет собой структуру, содержащую идентификатор токена и *конечный автомат*, описывающий все возможные последовательности символов для данного токена в данной позиции. При этом для каждого символа хранится информация: из какой строки получен этот символ и координаты его позиций внутри этой строки. Это необходимо для того, чтобы сохранить информацию о происхождении токена, так как он мог быть сформирован из различных строковых переменных в исходном коде.

Таким образом, **основная задача лексического анализа динамически формируемого строкового выражения** заключается в переводе конечного автомата над алфавитом символов обрабатываемого языка в конечный автомат над алфавитом токенов эталонной грамматики языка с сохранением привязки лексических единиц к исходному коду.

В данной статье для конечных автоматов и конечных преобразователей используются определения, представленные ниже.

Конечным автоматом называется кортеж $A = \langle Q, \Sigma, \Delta, q_0, F \rangle$, где Q — конечное множество состояний; Σ — входной алфавит; Δ задает на множестве Q структуру ориентированного графа, дуги которого помечены символами (x), где $x \in \Sigma \cup \{\varepsilon\}$; q_0 — начальное состояние; $F \subseteq Q$ — множество конечных состояний.

Конечным преобразователем называется кортеж $M = \langle Q, \Sigma, \Sigma', \Delta, q_0, F \rangle$, где Q — конечное множество состояний; Σ и Σ' — входной и выходной алфавиты соответственно; Δ задает на множестве Q структуру ориентированного графа,

дуги которого помечены парами $(x : y)$, где $x \in \Sigma$ и $y \in \Sigma' \cup \{\varepsilon\}$; q_0 — начальное состояние; $F \subseteq Q$ — множество конечных состояний.

В алгоритме лексического анализа используется операция **композиции** над двумя конечными преобразователями. Композиция конечных преобразователей [11] — это два последовательно взаимодействующих конечных преобразователя: выход первого конечного преобразователя служит входом для второго конечного преобразователя.

3.1 Генератор лексических анализаторов

Для проведения лексического анализа динамически формируемого выражения к двум конечным преобразователям применяется операция композиции, которая использует явное представление этих преобразователей, что порождает ограничения на формат определения лексической спецификации для языка.

Генератор лексических анализаторов FsLex строит конечный преобразователь, в котором входным алфавитом являются символы, имеющие кодировку ASCII или Unicode, выходным алфавитом — функции, тип возвращаемых значений которых соответствует типу `Token`. Эти функции соответствуют действиям (`action`), которые определены в спецификации для языка. Однако бывают ситуации, когда нужно исключить некоторые выражения, например пробелы и комментарии, из результата. Обычно это происходит на этапе проведения лексического анализа: в соответствующем действии не происходит возвращения токена, однако возвращаемое значение должно иметь тип `Token`. В классическом случае происходит вызов функции, соответствующей точке вхождения, от измененного состояния буфера лексического анализатора. Такой способ не применим для лексического анализа динамически формируемого кода, когда используется операция композиции, из-за ограничений на выходной алфавит. В данной работе предлагается использование следующих типов `Option`: `Some` обозначает возвращение токена; `None` — его отсутствие.

Использование нескольких точек вхождения в определении спецификации означает рекурсивное определение функций, которые создаются на каждую точку вхождения: в соответствующем действии происходит вызов одной из этих функций. Обычно такой подход используют для обработки вложенных конструкций, например комментариев. Данный случай не учитывается в реализованном инструменте, что создает ограничение на число используемых точек вхождения: можно использовать только одну точку вхождения.

Чтобы показать введенные ограничения, в листингах 3 и 4 указана спецификация для языка арифметических выражений, которую принимают на вход генератор FsLex и разработанный инструмент соответственно.

В результате своей работы генератор создает файл с расширением .fs с кодом на языке F# для лексического анализатора. В этом файле содержится код, указанный в заголовке спецификации, конечный преобразователь, массив действий и функция `tokenize`. Функция `tokenize` осуществляет лексический разбор.

```
rule token = parse
| whitespace { token lexbuf }
| ['-']? digit+ ('.'digit+)?
  (['e' 'E'] digit+)?
       { NUMBER(lexbuf) }
| '-'  { MINUS(lexbuf) }
| '/'  { DIV(lexbuf) }
| '+'  { PLUS(lexbuf) }
| "**" { POW(lexbuf) }
| '*'  { MULT(lexbuf) }
```

Листинг 3 FsLex

```
rule token = parse
| whitespace { None }
| ['-']? digit+ ('.'digit+)?
  (['e' 'E'] digit+)?
       { Some(NUMBER(gr)) }
| '-'  { Some(MINUS(gr)) }
| '/'  { Some(DIV(gr)) }
| '+'  { Some(PLUS(gr)) }
| "**" { Some(POW(gr)) }
| '*'  { Some(MULT(gr)) }
```

Листинг 4 YaccConstructor

```
[| (fun (gr:FSA<_>) -> None );
   (fun (gr:FSA<_>) -> Some(NUMBER(gr)));
   (fun (gr:FSA<_>) -> Some(MINUS(gr)));
   (fun (gr:FSA<_>) -> Some(DIV(gr)));
   (fun (gr:FSA<_>) -> Some(PLUS(gr)));
   (fun (gr:FSA<_>) -> Some(POW(gr)));
   (fun (gr:FSA<_>) -> Some(MULT(gr))); |]
```

Листинг 5 Массив действий для спецификации языка, указанного в листинге 4

Массив действий состоит из функций, задающих отображение из некоторого конечного автомата в тип `Option<'token>`. Для спецификации языка из листинга 4 массив действий представлен в листинге 5.

3.2 Алгоритм лексического анализа

На этапе построения аппроксимации множества значений динамически формируемого строкового выражения происходит сохранение привязки: с каждым символом сохраняются координаты позиций этого символа в исходной строке, а также привязка этой строки к исходному коду. Результатом этого этапа является конечный автомат A, в котором входной алфавит состоит из элементов вида `('символ', привязка)`.

Над конечным автоматом A запускается процедура построения детерминированного конечного автомата. При этом конечный автомат детерминирован так, чтобы на дугах из одной вершины не было двух одинаковых символов с одинаковой привязкой. Из полученного конечного автомата $A' = \langle Q, \Sigma, \Delta', q_0, F\rangle$ строится конечный преобразователь $M = \langle Q, \Sigma, \Sigma', \Delta, q_0, F\rangle$ для лексического анализатора, в котором выходной алфавит $\Sigma' = \{s \mid (s, \_) \in \Sigma\}$, т. е. дуги графа, являющегося представлением конечного преобразователя M, помечены парами `(('символ', привязка) : 'символ')`. Отображение Δ отличается от отобра-

жения Δ' только тем, что появился выходной алфавит. Такое преобразование необходимо для выполнения операции композиции.

В конечный преобразователь M необходимо добавить переход по символу `'eof'` из всех конечных состояний в новое состояние, которое теперь станет конечным. Этот символ означает окончание строки и необходим для корректной работы лексического анализатора, так как вычисление действия (`action`) к накопленной строке происходит при чтении следующего символа.

На вход лексический анализатор принимает два конечных преобразователя, один из которых получен в результате построения аппроксимации, а второй — из описания, построенного генератором лексических анализаторов. Предлагаемый алгоритм для проведения лексического анализа динамически формируемого выражения состоит из двух этапов.

Этап 1. Выполнение операции композиции над двумя входными конечными преобразователями. Результатом этой операции является либо набор лексических ошибок, либо конечный преобразователь $M_1 = \langle Q_1, \Sigma_1, \Sigma_1', \Delta_1, q_{01}, F_1 \rangle$. Наличие лексических ошибок возможно в двух случаях: либо конечный преобразователь M содержит символы, которых нет в лексической спецификации, либо конечный преобразователь M порождает такой язык, который не принимает на вход лексический анализатор. Возможна также ситуация, когда конечный преобразователь M порождает язык, в котором есть слова, не принимаемые на вход лексическим анализатором.

Этап 2. Если конечный преобразователь M_1 получен, то происходит этап его интерпретации, результатом которой станет конечный автомат A_{token} над алфавитом токенов. В конечном преобразователе M_1 выделяются *action-вершины* — это вершины, из которых выходит хотя бы одна дуга, содержащая функцию, возвращающую `Some(token)`. После action-вершины всегда возвращается один тип токена, но может выходить дуга, помеченная символом $(\_, \varepsilon)$ и означающая, что токен продолжает накапливаться. Пример такого конечного преобразователя представлен на рис. 1. Из одной action-вершины могут быть достижимы несколько других action-вершин. Число достижимых вершин соответствует числу токенов, которые нужно выделить в конечном преобразователе M_1 для данной action-вершины.

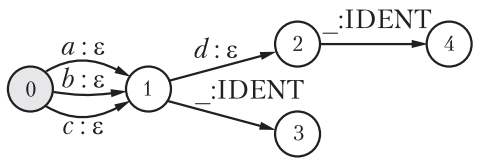


Рис. 1 Вершины 1 и 2 являются *action*-вершинами, но из вершины 1 токен продолжает накапливаться

Этот этап состоит из шагов, представленных ниже. На них используется структура `GraphAction`, которая предназначена для выделения токенов в конечном преобразователе M_1 и содержит поля для стартовой вершины, набора конечных вершин и конечного автомата. Стартовой вершиной может быть только action-вершина или начальное состояние, конечной вершиной — action-вершина или конечное состояние.

В описании алгоритма для дуги конечного преобразователя M_1 используется нотация $e = (u, (a, b), v)$, где u — начальная вершина дуги; v — конечная вершина; (a, b) — метка дуги; $a \in \Sigma_1$, $y \in \Sigma_1' \cup \{\varepsilon\}$. Добавление дуги в конечный автомат, который содержится в структуре `GraphAction`, осуществляется с помощью функции `graphAction.fsa.AddEdge(e)`, которая добавляет дугу из вершины u в вершину v, помеченную символом a, в соответствующий конечный автомат. В алгоритме используются очередь Q, для того чтобы контролировать порядок обхода конечного преобразователя, и структура `GraphAction grAct` для выделения конечного автомата, который содержит токен.

Шаг 1. Для конечного преобразователя M_1 строится набор вершин `actV`, которые являются action-вершинами.

Шаг 2. Для начального состояния конечного преобразователя M_1 и вершин из набора `actV` запускается обход, который накапливает для токена конечный автомат. Порядок обхода представлен в алгоритме 1. Результатом обхода

Алгоритм 1 Порядок обхода конечного преобразователя для сохранения привязки лексических единиц к исходному коду

```
function BFS(vStart, M1)
  grAct.startV ← vStart
  Q.Enqueue(vStart)
  while Q is not empty do
   v ← Q.Dequeue()
   if v is not visited then
    VISIT(v)
    for all e = (v, _, u) in M1 do
      if e = (v, (_, ε), u) then
       c ← v is init state of M1
       if v ≠ vStart or c then
        grAct.fsa.AddEdge(e)
        if u = vStart then
          grAct.endV.Add(u)
        Q.Enqueue(u)
      else
      if v = vStart then
        grAct.fsa.AddEdge(e)
        if u is final state in M1 then
         grAct.endV.Add(u)
         else
         Q.Enqueue(u)
      else
        grAct.endV.Add(v)
```

Алгоритм 2 Порядок обхода инвертированного конечного преобразователя для сохранения привязки лексических единиц к исходному коду

```
function BFSINV (vStart, M_2, M_1)
  grAct.startV ← vStart
  Q.Enqueue(vStart)
  while Q is not empty do
   v ← Q.Dequeue()
   if v is not visited then
     VISIT(v)
     for all e = (v, _, u) in M_2 do
       grAct.fsa.AddEdge(e)
       c ← u is action-vertex or init state of M_2
       if c then
         if ∃e = (u, (_, ε), _) in M_1 then
           Q.Enqueue(u)
       else
       Q.Enqueue(u)
```

для всех вершин является коллекция `tokenAct`, состоящая из элементов типа `GraphAction`.

Шаг 3. Для точного определения конечного автомата, сохраняющего связь между токеном и исходным кодом, необходимо пройти конечный преобразователь M_1 в обратном направлении. Такая необходимость возникает в случае, если конечный преобразователь M_1 содержит циклы. На этом шаге также запускается обход для вершин из набора `actV` и начального состояния конечного преобразователя M_2, который является инвертированным конечным преобразователем M_1. Порядок обхода представлен в алгоритме 2. Результатом обхода для всех вершин является коллекция `tokenActInv`, состоящая из элементов типа `GraphAction`.

Шаг 4. Ищутся пересечения конечных автоматов, полученных на двух предыдущих шагах, при условии, что `tokenAct.endV.current` = = `tokenActInv.startV`. Токен определяется функцией, лежащей на дуге, исходящей из вершины `tokenAct.endV.current`, в него записывается результат пересечения. При этом в конечный автомат A_{token} добавляется переход по этому токену из вершины `tokenAct.startV` в вершину `tokenAct.endV.current`.

Так как символ `'eof'` является вспомогательным символом для лексического анализа, то для всех вершин, из которых выходит дуга, помеченная `'eof'`, добавляется переход в новое состояние по токену `EOF` в конечный автомат A_{token}, которое теперь становится конечным состоянием A_{token}. Над полученным конечным автоматом запускается процедура построения детерминированного конечного автомата. При этом автомат детерминирован так, чтобы на дугах из одной вершины не было двух токенов с одинаковым идентификатором и конечным автоматом.

3.3 Пример

Рассмотрим работу алгоритма лексического анализа на примере. Пусть результатом аппроксимации является конечный автомат A, представленный на рис. 2, *а*. Преобразуем этот конечный автомат во входную структуру для алгоритма. Для этого сперва строится детерминированный конечный автомат A' (рис. 2, *б*), затем конечный преобразователь M (рис. 2, *в*).

Для обработки конечного преобразователя M используется спецификация для языка арифметических выражений (см. листинг 4). Результат композиции конечного преобразователя M с конечным преобразователем, построенным генератором лексических анализаторов, показан на рис. 3.

На рис. 3 цифры соответствуют индексам в массиве действий, которые представлены в листинге 5 (нумерация элементов в массиве начинается с 0). Индекс 4 соответствует функции, которая возвращает токен `Some(PLUS)`, 5 — токен `Some(POW)`, 6 — токен `Some(MULT)`. Так как лексических ошибок получено не было, то происходит этап интерпретации полученного конечного преобразователя M_1, который состоит из 4 шагов. Ниже представлены результаты выполнения каждого шага.

Шаг 1. В набор `actV` добавляются action-вершины 1, 2 и 3.

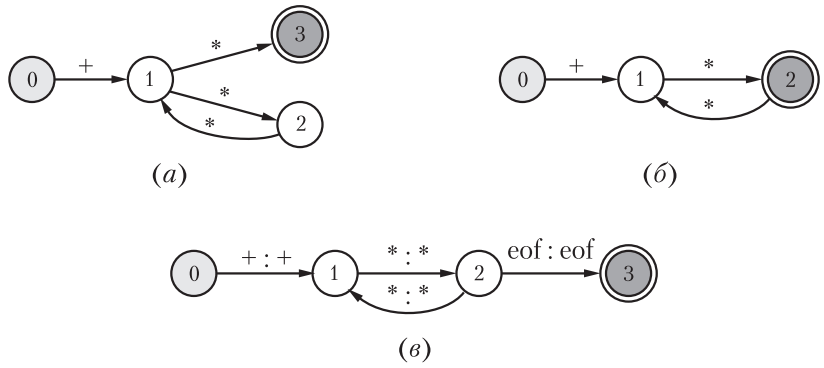


Рис. 2 Получение конечного преобразователя для лексического анализа: (*а*) конечный автомат A; (*б*) конечный автомат A'; (*в*) конечный преобразователь M

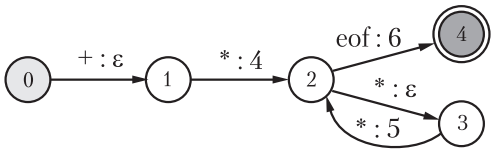
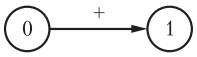
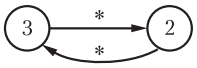
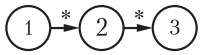
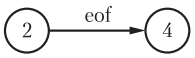


Рис. 3 Результат композиции

Таблица 1 Коллекция tokenAct для шага 2

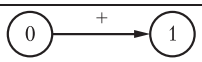
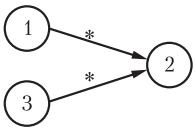
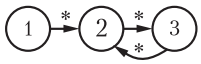
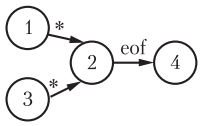
startV	FSA	endV
0		1
1		2,3
2		2,3
3		4

Шаг 2. Запускается обход, представленный в алгоритме 1, для вершин из набора `actV` и начального состояния конечного преобразователя M_1 (вершина 0). Результатом обхода для всех вершин является коллекция `tokenAct`, которая представлена в виде табл. 1.

Шаг 3. Запускается обход, представленный в алгоритме 2, для вершин из набора `actV` и начального состояния конечного преобразователя M_2 (вершина 49). Результатом обхода для всех вершин является коллекция `tokenActInv`, которая представлена в виде табл. 2.

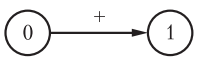
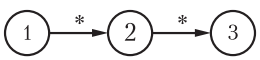
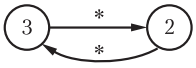
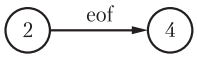
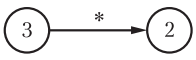
Шаг 4. Выполняются операции пересечения конечных автоматов, результаты которых представлены в табл. 3.

Таблица 2 Коллекция tokenActInv для шага 3

startV	FSA
1	
2	
3	
4	

Результатом лексического анализа выступает конечный автомат, представленный на рис. 4. Каждая дуга графа содержит токен, который хранит в себе конечный автомат. Например, дуга от вершины 1 к вершине 3 содержит

Таблица 3 Результат пересечения конечных автоматов из табл. 1 и 2

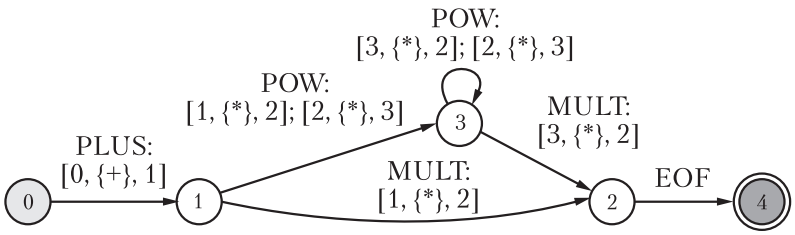


Рис. 4 Результат лексического анализа

токен `POW`, у которого хранится конечный автомат с переходами от состояния 1 к состоянию 2 по символу '*', от 2 к 3 — по символу '*', полученный в результате пересечения конечных автоматов.

4 Архитектура модуля лексического анализа

Архитектура инструмента, реализующего рассмотренный механизм, представлена на рис. 5.

Компонент **Лексический анализатор** состоит из **Генератора лексических анализаторов** и **Интерпретатора**. Генератор лексических анализаторов строит конечный преобразователь, описание которого взято из **Библиотеки для конечных автоматов и конечных преобразователей**, и сохраняет результат в отдельный файл. Такой подход позволяет многократно использовать данный конечный преобразователь для обработки кода, написанного на одном языке. **Интерпретатор** принимает на вход два конечных преобразователя, один из кото-

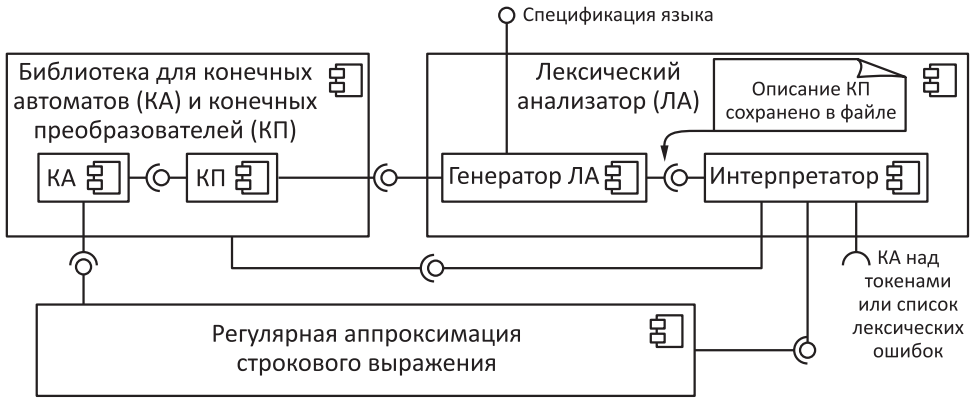


Рис. 5 Архитектура модуля для лексического анализа

рых получен в результате построения аппроксимации (за это отвечает компонент **Регулярная аппроксимация строкового выражения**), а второй построен генератором лексических анализаторов. Результатом работы **Интерпретатора** является либо конечный автомат над алфавитом токенов эталонной грамматики языка, либо список обнаруженных лексических ошибок.

Библиотека для конечных автоматов и конечных преобразователей предоставляет ряд операций, которые необходимы для построения аппроксимации и проведения лексического анализа. Для конечных автоматов используются операции: конкатенация, replace, дополнение, пересечение, а для конечных преобразователей — композиция.

5 Апробация

В данном разделе описан механизм использования реализованного инструмента, а также рассмотрены примеры, демонстрирующие разработанную функциональность.

Для осуществления лексического разбора необходимо выполнить следующие шаги.

Шаг 1. Запустить генератор, указав путь к файлу с расширением .fsl, в котором написана спецификация. В результате создается файл с расширением .fs, содержащий конечный преобразователь и вспомогательные функции.

Шаг 2. Необходимо в отдельном файле с расширением .fs указать описание типов токенов, которые автоматически строятся по грамматике эталонного языка. Полученные файлы подключить к модулю, предназначенному для получения результата лексического разбора.

Шаг 3. Получить конечный автомат, аппроксимирующий множество значений строкового выражения и удовлетворяющий описанию используемой библиотеки для конечных автоматов и конечных преобразователей. Преобразовать этот конечный автомат в конечный преобразователь (см. подразд. 3.2).

Шаг 4. Вызвать функцию `tokenize` из сгенерированного файла. Результатом действия этой функции будет либо конечный автомат над алфавитом токенов эталонной грамматики языка, либо список лексических ошибок.

Рассмотрим примеры, которые показывают преимущества реализованного решения, а именно: возможность сохранения конечного автомата внутри структуры токена и обработки циклов во входном конечном автомате.

Пример 1. Рассмотрим пример кода (листинг 6).

Результатом аппроксимации выражения `query` является конечный автомат, представленный на рис. 6. Результат лексического анализа представлен на рис. 7. В результирующем конечном автомате учитываются две ситуации: цикл не выполняется (путь $0 \to \cdots \to 7 \to 9$) и выполняется (путь $0 \to \cdots \to 7 \to \to 8 \to 10 \to 8 \to \cdots \to 10 \to 9$), что позволяет уточнить проводимый анализ в целом.

```
private void Go(int number){
   String query =  "SELECT nameX FROM tableY WHERE x < ";
   while(query.Length < number){
      query += "+ 1 ";
   }
   Program.ExecuteImmediate(query);
}
```

Листинг 6 Пример формирования выражения в цикле

Рис. 6 Результат аппроксимации

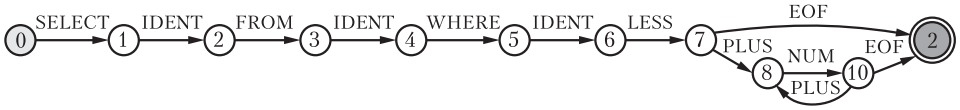


Рис. 7 Результат лексического анализа

Пример 2. Рассмотрим пример кода, в котором конечный автомат токена содержит цикл (листинг 7).

Результатом лексического анализа будет конечный автомат, представленный на рис. 8. Конечный автомат первого токена `IDENT`, содержащий цикл, представлен на рис. 9. На этом же рисунке показана сохраненная привязка символов к исходному коду. Таким образом, если цикл содержится только в конечном автомате токена, то в результирующем конечном автомате циклы отсутствуют, что позволяет упростить структуру входных данных для дальнейшего анализа.

На практике основным сценарием для динамически формируемого выражения является ситуация, представленная на рис. 10, где строки $x_1, x_2, \ldots, x_n$

```
String query = "SELECT name";
for(int i = 0; i < 10; i++){
   query  += "X";
}
query += " FROM tableY";
Program.ExecuteImmediate(query);
```

Листинг 7 Пример кода, в котором конечный автомат токена содержит цикл

Рис. 8 Результат лексического анализа

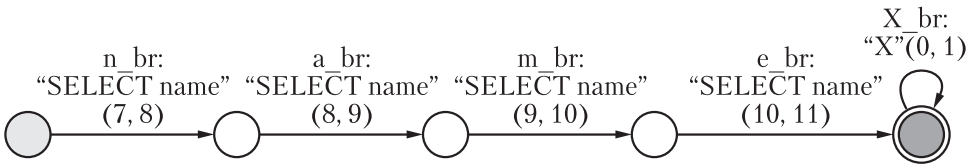


Рис. 9 Конечный автомат первого токена IDENT, содержащий цикл

возвращают одинаковый идентификатор токена. Разработанный механизм лексического анализа вернет один токен при переходе из состояния 1 в состояние 2, что значительно упростит входные данные для синтаксического разбора.

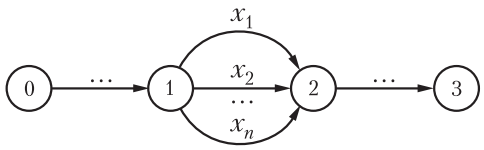


Рис. 10 Конечный автомат

6 Заключение

В данной работе описан алгоритм лексического анализа и основанный на нем генератор лексических анализаторов для динамически формируемого кода. Данный алгоритм был реализован в проекте YaccConstructor на языке программирования F#. Разработанный алгоритм позволяет при проведении лексического анализа для токена сохранять конечный автомат, что значительно упрощает входные данные для синтаксического разбора и привязку лексических единиц к исходному коду, которая необходима для позиционирования места ошибок или навигации по коду. Разработанный генератор лексических анализаторов позволяет получать по спецификации языка соответствующий анализатор, использующий описанный алгоритм. Таким образом, был разработан автоматизированный подход к созданию лексических анализаторов для динамически формируемого кода.

В дальнейшем планируется снять ограничения на формат написания спецификации, по которому генератор лексических анализаторов строит соответствующий анализатор, а также оптимизировать полученный инструмент за счет подбора структур данных и алгоритмов для работы с конечными автоматами [12].

Литература

1. *Christensen A. S., Møller A., Schwartzbach M. I.* Precise analysis of string expressions // 10th Conference (International) on Static Analysis Proceedings. — Berlin–Heidelberg: Springer-Verlag, 2003. P. 1–18.
2. *Minamide Y.* Static approximation of dynamically generated web pages // 14th Conference (International) on World Wide Web Proceedings. — New York, NY, USA: ACM, 2005. P. 432–441.
3. *Jovanovic N., Kruegel C., Kirda E.* Pixy: A static analysis tool for detecting web application vulnerabilities // Symposium on Security and Privacy Proceedings. — Berkeley / Oakland, CA, USA: IEEE, 2006. P. 263–269.
4. *Yu F., Alkhalaf M., Bultan T.* Stranger: An automata-based string analysis tool for PHP // Tools and algorithms for the construction and analysis of systems / Eds. J. Esparza, R. Majumdar. — Lecture notes in computer science ser. — Berlin–Heidelberg: Springer, 2010. Vol. 6015. P. 154–157.
5. *Dasgupta A., Narasayya V., Syamala M.* A static analysis framework for database applications // Conference on Computer Software and Applications Proceedings. — IEEE, 2007. P. 87–96.
6. *Fu X., Qian K.* Safeli: Sql injection scanner using symbolic execution // Workshop on Testing, Analysis, and Verification of Web Services and Applications Proceedings. — New York, NY, USA: ACM, 2008. P. 34–39.
7. *Annamaa A., Breslav A., Kabanov J., Vene V.* An interactive tool for analyzing embedded SQL queries // Programming languages and systems / Ed. K. Veda. — Lecture notes in computer ser. — Berlin–Heidelberg: Springer, 2010. Vol. 6461. P. 131–138.
8. *Кириленко Я. А., Григорьев С. В., Авдюхин Д. А.* Разработка синтаксических анализаторов в проектах по автоматизированному реинжинирингу информационных систем // Научно-технические ведомости СПбГПУ. Информатика. Телекоммуникации. Управление, 2013. Вып. 3(174). С. 94–98.
9. *Grigorev S., Verbitskaia E., Ivanov A., Polubelova M., Mavchun E.* String-embedded language support in integrated development environment // 10th Central and Eastern European Software Engineering Conference in Russia Proceedings. — ACM, 2014. P. 21:1–21:11.
10. *Yu F., Alkhalaf M., Bultan T., Ibarra O. H.* Automata-based symbolic string analysis for vulnerability detection // Form. Method. Syst. Des., 2014. Vol. 44. No. 1. P. 44–70.
11. *Hanneforth T.* Finite-state machines: Theory and applications. Unweighted finite-state automata. — Universität Potsdam, 2008. 99 p. http://tagh.de/tom/wp-content/uploads/fsm_unweigtedautomata.pdf.
12. *Hooimeijer P., Veanes M.* An evaluation of automata algorithms for string analysis // 12th Conference (International) on Verification, Model Checking, and Abstract Interpretation Proceedings. — Berlin–Heidelberg: Springer-Verlag, 2011. P. 248–262.

Поступила в редакцию 26.02.16

LEXICAL ANALYSIS OF DYNAMICALLY GENERATED STRING EXPRESSIONS

M. I. Polubelova and S. V. Grigorev

Saint Petersburg State University, 7/9 Universitetskaya Nab., St. Petersburg 199034, Russian Federation

Abstract: There is a class of applications which utilizes the idea of string embedding of one language into another. In this approach, a host program generates string representation of clauses in some external language, which are then passed to a dedicated runtime component for analysis and execution. Despite providing better expressiveness and flexibility, this technique makes the behavior of the system less predictable, complicates maintenance, and is a source of such vulnerabilities as SQL injections and cross-site scripting. Static analysis of strings is intended to minimize the drawbacks of the approach by checking well-formedness of a set of all dynamically-generated clauses at compile-time. Lexical analysis, or tokenization, is an important step of static analysis. The paper presents an automated approach to lexical analyzers construction which simplifies implementation of static analyzers of dynamically generated code.

Keywords: string analysis; lexing; string-embedded language; lexer generator

DOI: 10.14357/08696527160203

References

1. Christensen, A. S., A. Møller, and M. I. Schwartzbach. 2003. Precise analysis of string expressions. *10th Conference (International) on Static Analysis Proceedings*. Berlin-Heidelberg: Springer-Verlag. 1–18.
2. Minamide, Y. 2005. Static approximation of dynamically generated web pages. *14th Conference (International) on World Wide Web Proceedings*. New York, NY: ACM. 432–441.
3. Jovanovic, N., C. Kruegel, and E. Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. *Symposium on Security and Privacy Proceedings*. Berkeley/Oakland, CA: IEEE. 263–269.
4. Yu, F., M. Alkhalaf, and T. Bultan. 2010. Stranger: An automata-based string analysis tool for PHP. *Tools and algorithms for the construction and analysis of systems*. Eds. J. Esparza, and R. Mayumdar. Lecture notes in computer scievce ser. Berlin-Heidelberg: Springer. 6015:154–157.
5. Dasgupta, A., V. Narasayya, and M. Syamala. 2007. A static analysis framework for database applications. *Conference on Computer Software and Applications Proceedings*. IEEE. 87–96.
6. Fu, X., and K. Qian. 2008. Safeli: Sql injection scanner using symbolic execution. *Workshop on Testing, Analysis, and Verification of Web Services and Applications Proceedings*. New York, NY: ACM. 34–39.

7. Annamaa, A., A. Breslav, J. Kabanov, and V. Vene. 2010. An interactive tool for analyzing embedded SQL queries. *Programming languages and systems*. Ed. K. Veda. Lecture notes in computer ser. Berlin–Heidelberg: Springer. 6461:131–138.
8. Kirilenko, I., S. Grigorev, and D. Avdiukhin. 2013. Razrabotka sintaksicheskikh analizatorov v proektakh po avtomatizirovannomu reinzhiniringu informatsionnykh sistem [Syntax analyzers development in automated reengineering of informational systems]. *Nauchno-tekhnicheskie vedomosti SPbGPU. Informatika. Telekommunikatsii. Upravlenie* [St. Petersburg State Polytechnical University J. Computer Science. Telecommunications and Control Systems] 3(174):94–98.
9. Grigorev, S., E. Verbitskaia, A. Ivanov, M. Polubelova, and E. Mavchun. 2014. String embedded language support in integrated development environment. *10th Central and Eastern European Software Engineering Conference in Russia Proceedings*. Moscow. 21:1–21:11.
10. Yu, F., M. Alkhalaf, T. Bultan, and O. H. Ibarra. 2014. Automata-based symbolic string analysis for vulnerability detection. *Form. Method. Syst. Des*. 44(1):44–70.
11. Hanneforth, T. 2008. Finite-state machines: Theory and applications. Unweighted finite-state automata. 99 p. http://tagh.de/tom/wp-content/uploads/fsm_unweigtedautomata.pdf (accessed April 22, 2016).
12. Hooimeijer, P., and M. Veanes. 2011. An evaluation of automata algorithms for string analysis. *12th Conference (International) on Verification, Model Checking, and Abstract Interpretation Proceedings*. Berlin–Heidelberg: Springer-Verlag. 248–262.

Received February 26, 2016

Contributors

Polubelova Marina I. (b. 1993) — student, Saint Petersburg State University, 7/9 Universitetskaya Nab., St. Petersburg 199034, Russian Federation; polubelovam@gmail.com

Grigorev Semen V. (b. 1989) — PhD student, Saint Petersburg State University, 7/9 Universitetskaya Nab., St. Petersburg 199034, Russian Federation; Semen.Grigorev@jetbrains.com