Лексический анализ динамически формируемых строковых выражений

Полубелова Марина Игоревна
Санкт-Петербургский Государственный Университет
Санкт-Петербург, Россия
polubelovam@gmail.com

Григорьев Семён Вячеславович
Санкт-Петербургский Государственный Университет
Санкт-Петербург, Россия
Semen.Grigorev@jetbrains.com

***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 и межсайтовый скриптинг. Статический анализ строковых выражений предназначен для борьбы с недостатками подхода посредством проверки, что все формируемые выражения удовлетворяют некоторым свойствам, без запуска программы. Лексический анализ или токенизация формируемого кода является важным шагом такого статического анализа. В данной статье будет 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, string analysis, lexing, string-embedded language**

I. Введение

Многие языки программирования позволяют работать со строковыми выражениями. Они могут формироваться динамически с использованием строковых операций и языковых конструкций, например, условных операторов и циклов. Такие выражения широко используются в программных интерфейсах ODBC, ADO.NET, JDBC, предназначенных для формирования запросов к базе данных в языках программирования C++, C#, Java, соответственно, а также в web-программировании. Некоторые примеры использования динамически формируемых строковых выражений представлены в листингах 1, 2.

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

```
private void Go(bool cond)
{
  string tableName = cond ? "Sold " : "OnSale ";
  string query =
      "SELECT ProductID, UnitPrice, ProductName"
          + " FROM dbo.products_" + tableName
          + " WHERE UnitPrice > 1000 "
          + " ORDER BY UnitPrice DESC;";
  Program.ExecuteImmediate(query);
}
```

Листинг 1: Пример встроенного SQL в C#

```
<?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 2: Использование нескольких встроенных в PHP языков (MySQL, HTML)

Классическим подходом к статическому анализу является проведение 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 будет описан автоматизированный подход создания лексического анализатора для динамически формируемого кода. В рамках работы были разработаны 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 является модульным инструментом для проведения лексического анализа и синтаксического разбора, а также платформой для поддержки встроенных языков.

II. Обзор

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

A. 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 проверка выражения на соответствие описанию некоторой эталонной грамматики, либо статический анализ программы на уязвимость. Так как генерация всех значений динамически формируемого выражения значительно снижает скорость проведения анализа и возможны ситуации, когда количество 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 [3]. 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 [4], 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 [5], предназначенном для поиска SQL инъекций и межсайтового скриптинга в PHP-программах, применяется техника path pruning, позволяющая проводить анализ только тех значений строкового выражения, которые оно может принять в момент выполнения программы Инструмент Stranger [6] 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 рамках инструмента 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 [7], 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 с ожидаемым в программе.

Инструмент SAFELI [8], также предназначенный для поиска уязвимостей в web-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 выра-

жения был разработан инструмент Alvor [9], который является расширением для среды разработки Eclipse, предназначенным для статической валидации SQL-выражений, встроенных в программы на Java. Одной из возможностей инструмента является 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.

B. Инструмент YaccConstructor

YaccConstructor [2] является модульным инструментом с открытым исходном кодом [10], предназначенным для исследований в области лексического анализа и синтаксического разбора, а также платформой для поддержки встроенных языков [11]. 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 (DAG). 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.

C. 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. Для реализации инструмента для проведения лексического анализа динамически формируемого кода был выбран генератор лексических анализаторов FsLex [1]. Этот выбор обусловлен тем, что реализованный механизм является компонентом инструмента YaccConstructor, основным языком разработки которого является 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 3 .

```
{
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 3: Формат определения спецификации для языка

В описании спецификации выделяется три блока: заголовок (`header`), блок именованных регулярных выражений и блок точек вхождения (`entrypoint`).

Обычно заголовок содержит директивы `open`, необходимые для выполнения действий (`action`), а также вспомогательные функции. Между заголовком и точками вхождений можно определять имена для регулярных выражений. Для каждой точки вхождения генерируется функция на языке F# с таким же именем. Аргументом для таких функций является буфер лексических анализаторов, а возвращают они вычисленное действие. При этом возвращаемое значение должно иметь тип `Parser.token`, который описывает токены.

Буфер лексических анализаторов — это абстрактный тип данных, реализованный в модуле стандартной библиотеки `Lexing` (Microsoft.FSharp.Text.Lexing). Методы `FromString`, `FromBytes`, `FromChars` создают лексические буферы, принимающие данные из строки символов, массива байтов и символов, соответственно. Данный тип позволяет сохранять для лексемы координаты отступов начала и конца в исходном коде и накопленную строку.

Лексический анализатор читает символы из буфера и сравнивает их с регулярным выражением, которое соответствует правилу, до тех пор, пока префикс не совпадает с одним из регулярных выражений. Если префикс совпадает с несколькими регулярными выражениями, то из них выбирается то, которое первым определено в спецификации. Например, если у нас есть два правила, которые на выражения “*” и “**” должны возвращать токены `MULT` и `POW`, соответственно, то в определении спецификации необходимо сперва указать правило для токена `POW`, а только потом для токена `MULT`. После выбора регулярного выражения, к накопленной строке применяется действие, соответствующее этому регулярному выражению.

Пример спецификации для языка арифметических выражений представлен в листинге 4.

```
{
module Calc.Lexer
open Calc.Parse
open Microsoft.FSharp.Text.Lexing

let lexeme lb = LexBuffer<_>.LexemeString lb
}

let digit = ['0'-'9']
let whitespace = [' ' '\t' '\r' '\n']

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

Листинг 4: Лексическая спецификация для языка арифметических выражений

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 построенного конечного преобразователя. Для того 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.

III. Лексический анализ

Основной задачей лексического анализа является преобразование входного потока символов в поток токенов, соответствующих спецификации обрабатываемого языка, и сохранение привязки лексических единиц к исходному коду. В классическом лексическом анализе входной поток является линейным. Для проведения лексического анализа динамически формируемого строкового выражения необходима структура, которая является конечным представлением множества значений этого выражения. Для построения аппроксимации используется алгоритм, предложенный в статье [12], поэтому такой структурой является 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 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.

A. Конечные преобразователи и конечные автоматы

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, при этом 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, Σ' — входной и выходной алфавиты соответственно, при этом Δ задает на множестве Q структуру ориентированного графа, дуги которого помечены парами $(x : y)$, где $x \in \Sigma$, $y \in \Sigma' \cup \{\varepsilon\}$. q_0 — начальное состояние, $F \subseteq Q$ — множество конечных состояний.

В алгоритме лексического анализа используется операция **композиции** над двумя конечными преобразователями. Композиция конечных преобразователей [13] — это два последовательно взаимодействующих конечных преобразователя: выход первого конечного преобразователя является входом для второго конечного преобразователя. Пусть даны $M_1 = \langle Q_1, \Sigma_1, \Sigma_1', \Delta_1, q_{01}, F_1\rangle$ и $M_2 = \langle Q_2, \Sigma_2, \Sigma_2', \Delta_2, q_{02}, F_2\rangle$, то результатом композицией M_1 и M_2 будет конечный преобразователь $\langle Q_1 \times Q_2, \Sigma_1, \Sigma_2', \Delta, q_{01} \times q_{02}, F_1 \times F_2\rangle$, при этом Δ задает на множестве $Q_1 \times Q_2$ структуру ориентированного графа, дуги которого помечены парами $(x : y)$, где $x \in \Sigma_1$, $y \in \Sigma_2'$, если $\exists z \in \Sigma_1' \cap \Sigma_2$: в M_1 есть дуга, помеченная $(x : z)$, и в M_2 есть дуга, помеченная $(z : y)$.

B. Генератор лексических анализаторов

Для проведения лексического анализа динамически формируемого выражения применяется операция композиции к двум конечными преобразователями, 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 является 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 возвращение 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 композиция, из-за ограничений на выходной алфавит. В данной работе предлагается использование типа `Option`: `Some` обозначает возвращение токена, `None` — его отсутствие.

Использование нескольких точек вхождений 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. В терминах конечного преобразователя это означает, что Δ задает на множестве Q структуру ориентированного графа, дуги которого помечены парами $(x : y)$, где $x \in \Sigma$, $y \in \Sigma' \cup \{\varepsilon\} \cup Q$. То есть конечный преобразователь должен уметь выдавать не только символ из выходного алфавита, но и состояние другого конечного преобразователя, в которое должен перейти анализатор. Обработав этот конечный преобразователь, анализатор должен вернуться в состояние исходного конечного преобразователя, которое было получено при переходе. Определение операции композиции над такими типами конечных преобразователей на данный момент не обнаружено, что создает ограничение на количество используемых точек вхождений: можно использовать только одну точку вхождения.

С введенными выше ограничениями спецификация для языка арифметических выражений будет иметь вид, представленный в листинге 5.

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. Массив действий состоит из функций, задающих отображение из некоторого конечного автомата в тип `Option<'token>`. Для спецификации языка, представленного в листинге 5, массив действий будет иметь следующий вид:

```
{
module Calc.Lexer
open Calc.Parser
}

let digit = ['0'-'9']
let whitespace = [' ' '\t' '\r' '\n']

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 5: Лексическая спецификация для языка арифметических выражений

```
[|
  (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 6: Массив действий для спецификации языка, указанной в листинге 5

C. 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 $\Sigma' = \{s \mid (s, \_) \in \Sigma\}$. То есть дуги графа, являющегося представлением конечного преобразователя M, помечены парами `(('символ', привязка) : 'символ')`. Отображение Δ отличается от отображения Δ' только тем, что появился выходной алфавит. Такое преобразование необходимо для выполнения операции композиции.

В конечный преобразователь M необходимо добавить переход по символу `'eof'` от всех конечных состояний в новое состояние, которое теперь будет являться 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. В этой ситуации набор лексических ошибок получается следующим образом: применяется операция разности к двум конечным автоматам, полученных из конечных преобразователей M и M_1 с помощью входной проекции соответствующих конечных преобразователей. После чего происходит лексический анализ подмножества языка, принимаемого лексическим анализатором.

Этап 2. Если конечный преобразователь M_1 получен, то происходит этап его интерпретации, результатом которой будет являться 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-

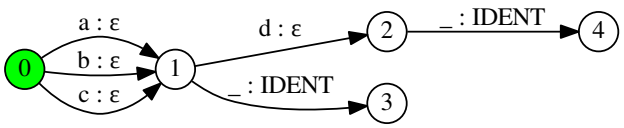


Рис. 1: Вершины 1 и 2 являются *action*-вершинами, но из вершины 1 токен продолжает накапливаться

вершины всегда возвращается один тип токена, но может выходить дуга, помеченная символом $(\_, \varepsilon)$, что означает, что токен продолжает накапливаться. Пример такого конечного преобразователя представлен на рис. 1. Из одной action-вершины могут быть достижимы несколько других action-вершин. Количество достижимых вершин соответствует количеству токенов, которые нужно выделить в конечном преобразователе M_1 для данной action-вершины.

Этот этап состоит из шагов, представленных ниже. В них используется структура `GraphAction` (см. листинг 7). Эта структура 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.

```
struct GraphAction =
    int startV
    array<int> endV
    FSA fsa  //finite state automaton
```

Листинг 7: Структура GraphAction

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

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)
```

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 является 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.

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

```
function BFSINV(vStart, M2, 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 M2 do
                grAct.fsa.AddEdge(e)
                c ← u is action-vertex or init state of M2
                if c then
                    if ∃e = (u, (_, ε), _) in M1 then
                        Q.Enqueue(u)
                else
                    Q.Enqueue(u)
```

D. Пример

Рассмотрим работу алгоритма для лексического анализа на примере. Пусть результатом аппроксимации является конечный автомат A, представленный на рис. 2. Для этого примера сохранение привязки для символов опускаем.

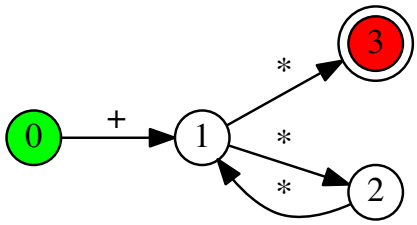


Рис. 2: Результат аппроксимации

Преобразуем этот конечный автомат во входную структуру для алгоритма. Для этого сперва строится детерминированный конечный автомат A' (см. рис. 3), затем — конечный преобразователь M (см. рис. 4).

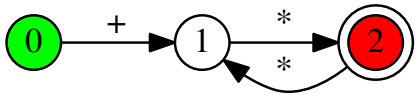


Рис. 3: Детерминированный конечный автомат A'

Для обработки конечного преобразователя M используется спецификация для языка арифметических

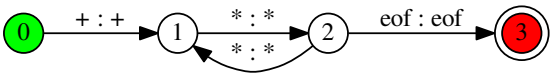


Рис. 4: Конечный преобразователь M для лексического анализа

выражений (см. листинг 5). Результат композиции конечного преобразователя M с конечным преобразователем, построенным генератором лексических анализаторов, показан на рис. 5.

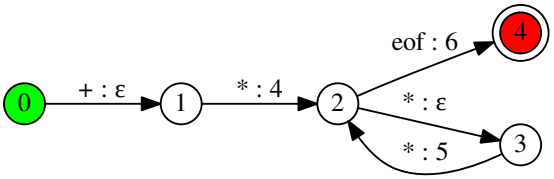
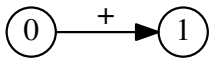
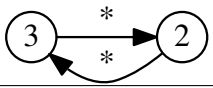


Рис. 5: Результат композиции

На рис. 5 цифры соответствуют индексу в массиве действий, который представлен в листинге 6 (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, 3.

Шаг 2. Запускается обход, представленный в алгоритме 1, для вершин из набора $actV$ и вершины 0, которая является начальным состоянием конечного преобразователя M_1. Результатом обхода для всех вершин является коллекция $tokenAct$, которая представлена в виде таблицы I.

startV	0	1
FSA		
endV	1	2,3

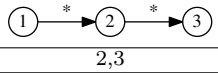
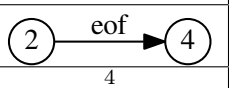
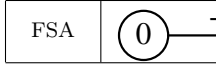
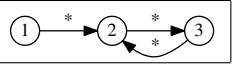
startV	2	3
FSA		
endV	2,3	4

Таблица I: Коллекция $tokenAct$ для Шага 2

Шаг 3. Запускается обход, представленный в алгоритме 2, для вершин из набора $actV$ и вершины 49, которая является начальным состоянием конечного преобразователя M_2. Результатом обхода для всех вершин является коллекция $tokenActInv$, которая представлена в виде таблицы II.

startV	1	3
FSA		

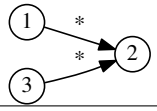
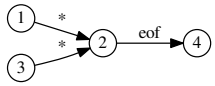
startV	2	4
FSA		

Таблица II: Коллекция $tokenActInv$ для Шага 3

Шаг 4. Выполняются операции пересечения конечных автоматов, результаты которых представлен в таблице III.

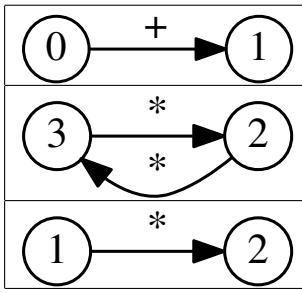
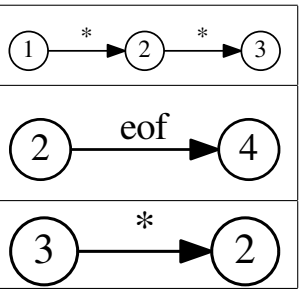
	

Таблица III: Результат пересечения конечных автоматов из таблиц I и II

Результатом лексического анализа будет конечный автомат, представленный на рис. 6. Каждая дуга графа содержит токен, который хранит в себе конечный автомат. Например, дуга от вершины 1 к вершине 3 содержит 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.

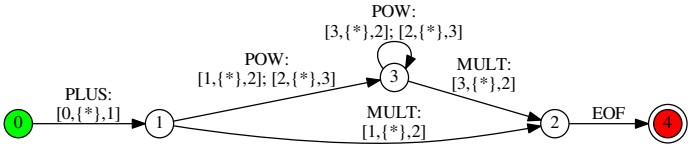


Рис. 6: Результат лексического анализа

IV. Архитектура модуля для лексического анализа

Архитектура инструмента, реализующая рассмотренный механизм, представлена на рис. 7.

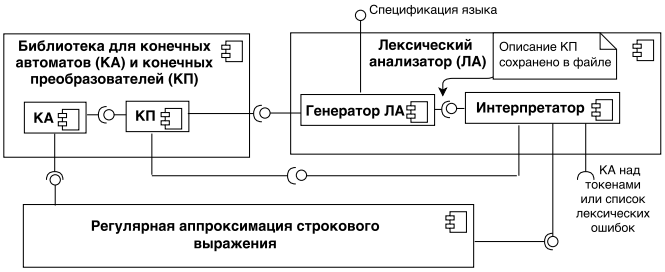


Рис. 7: Архитектура модуля для лексического анализа

Компонента **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 которых из которых 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.

V. 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 в модуль, предназначенный 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. Алгоритм лексического анализа).

Шаг 4. Вызвать функцию `tokenize` из сгенерированного файла. Результатом 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 следующий пример кода:

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

Листинг 8: Пример формирования выражения в цикле

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

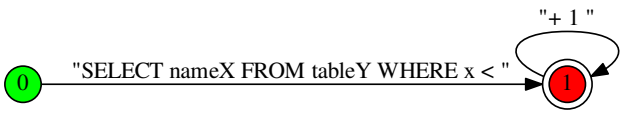


Рис. 8: Результат аппроксимации

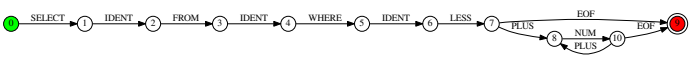


Рис. 9: Результат лексического анализа

Пример 2. Рассмотрим следующий пример кода, в котором конечный автомат токена содержит цикл:

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

Листинг 9: Пример кода, в котором конечный автомат токена содержит цикл

Результатом лексического анализа будет конечный автомат, представленный на рис. 10. Конечный автомат первого токена IDENT, содержащий цикл, представлен на рис. 11. 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.

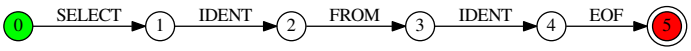


Рис. 10: Результат лексического анализа

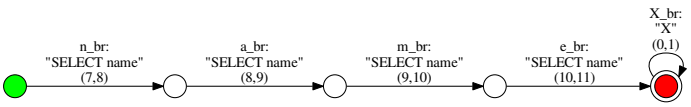


Рис. 11: Конечный автомат первого токена IDENT, содержащий цикл

На практике основным сценарием для динамически формируемого выражения является следующая ситуация, где строки x1, x2, . . . , xn возвращают одинаковый идентификатор токена:

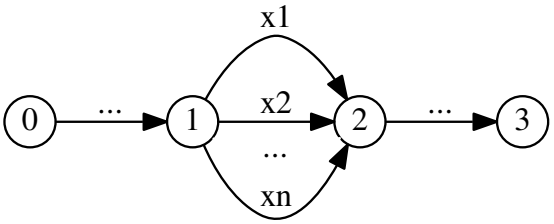


Рис. 12: Конечный автомат

Разработанный механизм лексического анализа вернет один токен при переходе от состояния 1 к состоянию 2, что значительно упрощает входные данные для синтаксического разбора.

VI. 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. Также алгоритм позволяет хранить 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 создания 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 [14].

Список литературы

[1] Сайт проекта fslex. [Online]. Available: http://fsprojects.github.io/FsLexYacc/

[2] I. Kirilenko, S. Grigorev, and D. Avdiukhin, “Syntax analyzers development in automated reengineering of informational system,” *St. Petersburg State Polytechnical University Journal. Computer Science. Telecommunications and Control Systems*, no. 174, pp. 94 – 98, 2013.

[3] A. S. Christensen, A. Møller, and M. I.Schwartzbach, “Precise analysis of string expressions,” in *Proceedings of the 10th International Conference on Static Analysis*. Berlin, Heidelberg: Springer-Verlag, Jun. 2003, pp. 1–18.

[4] Y. Minamide, “Static approximation of dynamically generated web pages,” in *Proceedings of the 14th International Conference on World Wide Web*. New York, NY, USA: ACM, 2005, pp. 432–441.

[5] N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: A static analysis tool for detecting web application vulnerabilities,” in *Security and Privacy, 2006 IEEE Symposium on*. Berkeley/Oakland, CA: IEEE, 2006, pp. 263–269.

[6] F. Yu, M. Alkhalaf, and T. Bultan, “Stranger: An automata-based string analysis tool for php,” in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2010, pp. 154–157.

[7] A. Dasgupta, V. Narasayya, and M. Syamala, “A static analysis framework for database applications,” in *Computer Software and Applications Conference*. IEEE, 2007, pp. 87–96.

[8] X. Fu and K. Qian, “Safeli: Sql injection scanner using symbolic execution,” in *Proceedings of the 2008 Workshop on Testing, Analysis, and Verification of Web Services and Applications*. New York, NY, USA: ACM, 2008, pp. 34–39.

[9] A. Annamaa, A. Breslav, J. Kabanov, and V. Vene, “An interactive tool for analyzing embedded sql queries,” in *Programming Languages and Systems*. Springer Berlin Heidelberg, 2010, pp. 131–138.

[10] Репозиторий проекта yaccconstructor. [Online]. Available: https://github.com/YaccConstructor/YaccConstructor

[11] S. Grigorev, E. Verbitskaia, A. Ivanov, M. Polubelova, and E. Mavchun, “String-embedded language support in integrated development environment,” in *Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia*. ACM, 2014, pp. 21:1–21:11.

[12] F. Yu, M. Alkhalaf, T. Bultan, and O. H. Ibarra, “Automata-based symbolic string analysis for vulnerability detection,” in *Formal Methods in System Design*. Springer US, 2014, pp. 44–70.

[13] T. Hanneforth. Finite-state machines: Theory and applications unweighted finite-state automata. Lecture Notes. [Online]. Available: http://tagh.de/tom/wp-content/uploads/fsm_unweigtedautomata.pdf

[14] P. Hooimeijer and M. Veanes, "An evaluation of automata algorithms for string analysis," in *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer-Verlag, 2011, pp. 248–262.