

Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»

Кафедра Системного Программирования

Полубелова Марина Игоревна

Лексический анализ динамически формируемых строковых выражений

Бакалаврская работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

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

Рецензент:
программист ООО «ИнтеллиДжей Лабс» Беляков А. М.

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY

Chair of Software Engineering

Polubelova Marina

Lexical analysis of dynamically generated string expressions

Graduation Thesis

Admitted for defence.

Head of the chair:
professor Terekhov A. N.

Scientific supervisor:
master of Information Technology, senior lecturer Grigorev S. V.

Reviewer:
software developer at “IntelliJ Labs Co.Ltd” Belyakov A. M.

Saint-Petersburg
2015

Оглавление

Введение	4
1. Постановка задачи	6
2. Обзор	7
2.1. Конечные преобразователи	7
2.2. Обзор существующих инструментов	8
2.3. Средства обработки строковых операций	10
2.4. Проект YaccConstructor	11
3. Алгоритм лексического анализа	12
4. Особенности реализации	16
4.1. Архитектура инструмента	16
4.2. Детали реализации	18
4.2.1. Конечные автоматы	18
4.2.2. Лексический анализ	19
5. Апробация	23
Заключение	26
Список литературы	28

Введение

Многие языки программирования позволяют работать со строковыми выражениями. Эти выражения могут быть получены с помощью строковых операций и языковых конструкций, например, условных операторов и циклов, то есть быть динамически формируемыми. Такие строковые выражения можно использовать, например, при генерации нового кода или при формировании запросов к базе данных. В качестве примера использования динамически формируемого строкового выражения рассмотрим SQL-запрос в C#:

```
1 private void Go(bool cond)
2 {
3     string tableName = cond ? "Sold" : "OnSale ";
4     string queryString =
5         "SELECT ProductID, UnitPrice, ProductName "
6         + "FROM dbo.products_" + tableName
7         + "WHERE UnitPrice > 1000 "
8         + "ORDER BY UnitPrice DESC;";
9     Program.ExecuteImmediate(queryString);
10 }
```

Listing 1: Пример встроенного SQL в C#

В этом примере выполняется запрос к базе данных (строка 9), в котором сам запрос получен в результате конкатенации нескольких строк (строки 5 – 8), выполненной с помощью условного оператора (строка 3). В строке 3 при формировании имени таблицы пропущен пробел, что приведет к ошибке выполнения запроса в случае истинности выражения `cond`. Но об ошибках подобного рода мы узнаем только в момент выполнения программы. На этом же примере видно, что в строке выполнения запроса (строка 9) содержится множество значений динамически формируемого строкового выражения `queryString`, компактное представление которого также хотелось бы иметь при разработке таких программ.

Динамически формируемые строковые выражения воспринимаются компилятором как обычные строки, что делает систему, использующую такой подход, ненадежной и уязвимой. Несмотря на то, что сейчас распространены другие технологии разработки программ, использующих язык запросов SQL, например LINQ, обозначенная выше проблема все равно остается актуальной. Во-первых, существует множество уже написанных программ и систем с использованием динамически формируемых строковых выражений, которые необходимо поддерживать. Во-вторых, может потребоваться реинжиниринг этих программ. Например, если выполнять миграцию

системы с языка Transact-SQL на Oracle-SQL, то тогда в приведенном выше примере необходимо формировать запрос уже на другом языке и при этом необходимо сохранить связь преобразованного кода с исходным. Так как в результате миграции могут возникнуть новые имена переменных, функций и другие языковые конструкции, то пользователям системы нужно быть уверенным в том, что логика и объекты системы не изменились. В-третьих, в web-программировании широко используется такой подход, например, динамически генерируемый HTML в PHP-программах. То есть в этом случае необходим статический анализ динамически формируемых строковых выражений.

Таким образом, ставится вопрос корректности программ, получающихся в результате использования динамически формируемых строковых выражений. Один из подходов к анализу таких программ заключается в проверке включения языков [4, 9]. Этот подход только отвечает на вопрос, включается ли язык, который порождает программа, в язык, описанный пользователем. Следующий подход состоит в проведении лексического анализа и синтаксического разбора компактного представления множества динамически формируемых строковых выражений, в качестве которого может быть регулярное выражение [8] или data-flow уравнение [5]. Этот подход позволяет для динамически формируемых строковых выражений разработать такую же функциональность, которую предоставляют интегрированные среды разработки для обычных языков, например, автодополнение, рефакторинг и дополнительные статические проверки.

Существуют инструменты, которые реализуют рассмотренные подходы, например, Java String Analyzer [4], PHP String Analyzer [9] и Alvor [8]. Однако они обладают рядом недостатков, среди которых можно выделить низкую точность проводимого анализа и трудность добавления поддержки нового языка. Последний недостаток можно заменить автоматизированным процессом, используя, например, стандартные генераторы лексических и синтаксических анализаторов. В статье [3] были показаны преимущества проведения лексического анализа отдельно от синтаксического разбора. Одним из преимуществ лексического анализа является то, что он не вносит потери точности проводимого анализа, а также позволяет переиспользовать существующие определения грамматик. В данной работе, выполненной в рамках проекта YaccConstructor [15], будет представлена реализация инструмента для проведения лексического анализа выражений, которые могут быть получены с помощью строковых операций и циклов, с сохранением привязки к исходному коду. В этой работе также будет представлена реализация генератора лексических анализаторов, что позволит сравнительно легко добавить обработку нового языка в реализованный инструмент.

1. Постановка задачи

Целью данной работы является реализация инструмента для проведения лексического анализа динамически формируемых строковых выражений. Для её достижения были поставлены следующие задачи:

- реализовать механизм для лексического анализа выражений, формируемых с помощью циклов и строковых операций, сохраняющий привязку частей динамически формируемого строкового выражения к исходному коду и привязку лексических единиц внутри каждой части;
- реализовать генератор лексических анализаторов, который по заданной спецификации языка строит описание конечного преобразователя. Конечный преобразователь — это конечный автомат, который может выводить конечное число символов для каждого входного символа.

2. Обзор

2.1. Конечные преобразователи

Конечный преобразователь (Finite State Transducer, [6]) — это конечный автомат (Finite State Automata, [11]), который может выводить конечное число символов для каждого входного символа. Конечный преобразователь может быть задан следующей шестеркой элементов: $\langle Q, \Sigma, \Delta, q_0, F, E \rangle$, где

- Q — множество состояний,
- Σ — входной алфавит,
- Δ — выходной алфавит,
- $q_0 \in Q$ — начальное состояние,
- $F \subseteq Q$ — набор конечных состояний,
- $E \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times (\Delta \cup \{\varepsilon\}) \times Q$ — набор переходов.

Конечные преобразователи находят широкое применение в области обработки естественного языка (Natural Language Processing, [10]), также их можно использовать и при проведении лексического анализа. Важной операцией над конечными преобразователями является операция композиции. **Композиция** конечных преобразователей — это два стоящих рядом взаимодействующих конечных преобразователя, работающих таким образом: выход первого конечного преобразователя является входом для второго конечного преобразователя. Ниже дано формальное определение операции композиции над конечными преобразователями, допускающие наличие **eps**-переходов.

Композицией двух конечных преобразователей $T_1 = \langle Q_1, \Sigma_1, \Delta_1, q_{01}, F_1, E_1 \rangle$ и $T_2 = \langle Q_2, \Sigma_2, \Delta_2, q_{02}, F_2, E_2 \rangle$ является конечный преобразователь $T = \langle Q_1 \times Q_2, \Sigma_1, \Delta_2, \langle q_{01}, q_{02} \rangle, F_1 \times F_2, E \cup E_\varepsilon \cup E_{i,\varepsilon} \cup E_{o,\varepsilon} \rangle$, где

- $E = \{ \langle \langle p, q \rangle, a, b, \langle p', q' \rangle \rangle \mid \exists c \in \Delta_1 \cap \Sigma_2 : \langle p, a, c, p' \rangle \in E_1 \wedge \langle q, c, b, q' \rangle \in E_2 \}$
- $E_\varepsilon = \{ \langle \langle p, q \rangle, a, b, \langle p', q' \rangle \rangle \mid \langle p, a, \varepsilon, p' \rangle \in E_1 \wedge \langle q, \varepsilon, b, q' \rangle \in E_2 \}$
- $E_{i,\varepsilon} = \{ \langle \langle p, q \rangle, \varepsilon, a, \langle p, q' \rangle \rangle \mid \langle q, \varepsilon, a, q' \rangle \in E_2 \wedge p \in Q_1 \}$
- $E_{o,\varepsilon} = \{ \langle \langle p, q \rangle, a, \varepsilon, \langle p', q \rangle \rangle \mid \langle p, a, \varepsilon, p' \rangle \in E_1 \wedge q \in Q_2 \}.$

2.2. Обзор существующих инструментов

Для работы с динамически формируемыми строковыми выражениями существует ряд инструментов, которые реализуют два основных подхода. Первый подход заключается в проверке включения языков. Данный подход только отвечает на вопрос, включается ли язык, который порождает программа, в язык, описанный пользователем, например, с помощью грамматики или простым перечислением строковых выражений, которых он ожидает получить в результате выполнения программы. Его реализуют инструменты, представленные ниже.

- **Java String Analyzer (JSA, [4, 20])** — инструмент для анализа формирования строк и строковых операций в программах на Java. Для каждого строкового выражения строится конечный автомат, представляющий приближенное значение всех значений этого выражения, которые могут быть получены во время выполнения программы. Для того, чтобы получить этот конечный автомат, необходимо из flow-графа анализируемой программы построить контекстно-свободную грамматику, которая получается в результате замены каждой строковой переменной нетерминалом, а каждой строковой операции — правилом продукции. После чего полученная грамматика аппроксимируется регулярным языком. В качестве результата работы данный инструмент также выдает строки, которые не входят в описанный пользователем язык, но могут сформироваться во время исполнения программы.
- **PHP String Analyzer (PHPSA, [9, 22])** — инструмент для статического анализа строк в программах на PHP. Расширяет подход предыдущего инструмента JSA. Отсутствует этап преобразования контекстно-свободной грамматики в регулярный язык, что повышает точность проводимого анализа. Для обработки строковых операций используется конечный преобразователь, который позволяет оставаться в рамках контекстно-свободной грамматики. Дальнейший анализ строковых выражений полностью взят из инструмента JSA.

Второй подход заключается в проведении лексического анализа и синтаксического разбора компактного представления множества динамически формируемых выражений. Данный подход реализуют инструменты, представленные ниже.

- **Alvor [8, 17]** — плагин к среде разработки Eclipse, предназначенный для статической валидации SQL-выражений, встроенных в Java. Для компактного представления множества динамически формируемого строкового выражения используется понятие абстрактной строки, которая фактически является регулярным выражением над используемыми в строке символами. В инструменте Alvor отдельным этапом выделен лексический анализ. Поскольку абстрактную строку

можно преобразовать в конечный автомат, то лексический анализ заключается в преобразовании этого конечного автомата в конечный автомат над токенами при использовании конечного преобразователя, полученного генератором JFlex. Несмотря на то, что абстрактная строка позволяет конструировать строковые выражения при участии циклов, плагин сообщает о том, что не может поддерживать такие языковые конструкции (см.рис. 1). Также инструмент Alvor не поддерживает обработку строковых операций, за исключением конкатенации (см. рис. 2).

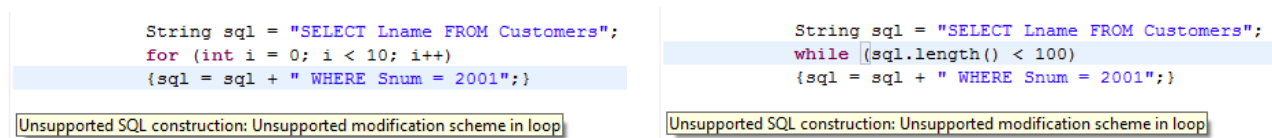


Рис. 1: Формирование строкового выражения с помощью цикла for и while в среде разработки Eclipse с установленным плагином Alvor

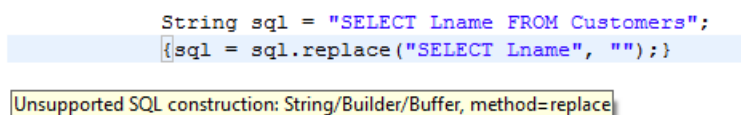


Рис. 2: Формирование строкового выражения с помощью строковой операции replace в среде разработки Eclipse с установленным плагином Alvor

- **Абстрактный синтаксический анализ.** Kyung-Goo Doh, Hyunha Kim, David A. Schmidt в статье [5] описали алгоритм статического анализа динамически формируемых строковых выражений на примере статической валидации динамически генерируемого HTML в PHP-программах. Хотя для данного примера отсутствует этап проведения лексического анализа, в общем случае можно использовать композицию лексического анализа и синтаксического разбора. Для этого достаточно хранить состояние конечного преобразователя, который используется для лексического анализа, внутри состояния синтаксического разбора. Данный алгоритм также предусматривает обработку строковой операции `string-replacement` с использованием конечного преобразователя, который по аналогии с лексическим конечным преобразователем хранит свое состояние внутри состояния синтаксического разбора. На вход абстрактный парсер принимает data-flow уравнения, составленные из исходного кода, и LALR(1) - таблицу, в качестве результата выдает набор абстрактных синтаксических деревьев.

Данный обзор показывает необходимость в реализации инструмента для проведения анализа строковых выражений, формируемых с помощью циклов и строковых

операций, с сохранением привязки частей динамически формируемого строкового выражения к исходному коду и привязки лексических единиц внутри каждой части. Данная привязка может использоваться, например, при формировании сообщений об ошибках пользователю, а также при проведении реинжиниринга, для которого важно сохранить связь с исходным кодом. Для обработки языка, используемого в динамически формируемом строковом выражении, необходимо также реализовать генератор лексических анализаторов, который по спецификации обрабатываемого языка строит описание конечного преобразователя. В данной работе будет представлена реализация инструмента для проведения лексического анализа динамически формируемых строковых выражений. Преимущества проведения лексического анализа отдельно от синтаксического разбора следующие [3]: во-первых, при таком подходе проще обнаружить причину ложного срабатывания проводимого анализа; во-вторых, лексический анализ не вносит потери точности анализа; в-третьих, можно переиспользовать существующие определения грамматик языков.

2.3. Средства обработки строковых операций

Поскольку динамически формируемые строковые выражения могут быть получены и при участии строковых операций, то их обработку также стоит включить в проводимый анализ. При этом некоторые из этих операций могут быть выражены через строковую операцию `replace`. Для реализации обработки этой операции, каждый аргумент которой является конечным автоматом, использовался алгоритм, описанный в статье [1]. Данный алгоритм реализован в инструменте Stranger [13], который предназначен для верификации строковых операций в РНР программах. В этом инструменте динамически формируемые строковые выражения представлены посредством конечного автомата с использованием библиотеки MONA [21], которая написана на языке программирования C и для конечного автомата использует MBDD-представление [2]. По причине того, что для символов алфавита конечного автомата необходимо хранить дополнительную информацию, инструмент Stranger не используется в данной работе.

Инструмент Stranger подразумевает только получение строк, которые могут быть получены в результате выполнения программы и при этом являться причиной уязвимости этой программы. Для достижения этой цели необходимо выполнить операцию пересечения конечного автомата, представляющий приближенное значение всех значений обрабатываемого выражения, которые могут быть получены во время выполнения, с конечным автоматом, который является шаблоном для поиска уязвимости в программах. То есть реализация инструмента Stranger не подразумевает проведение лексического анализа и синтаксического разбора множества динамически формируемых строковых выражений.

2.4. Проект YaccConstructor

Данная работа выполнена в рамках проекта YaccConstructor [25] лаборатории языковых инструментов JetBrains на математико-механическом факультете. В рамках этого проекта разрабатывается модульный инструмент для проведения лексического анализа и синтаксического разбора, а также платформа для исследования и разработки генераторов лексических и синтаксических анализаторов. Инструмент YaccConstructor также является платформой для поддержки встроенных языков, демонстрация которой может быть представлена в виде плагина к инструменту ReSharper [24]. Диаграмма последовательности, показывающая взаимодействие компонентов инструмента YaccConstructor, представлена на рис. 3. На диаграмме синим цветом выделена та часть, которой посвящена данная работа.

Ранее в проекте YaccConstructor был реализован инструмент для проведения лексического анализа динамически формируемых строковых выражений [14, 16]. Но прежняя реализация обладала грубой аппроксимацией исходного кода, а именно, не осуществляла поддержку циклов и строковых операций. Поэтому было решено реализовать инструмент, который улучшил бы точность проводимого анализа, поддерживая при этом циклы и строковые операции. Чтобы это сделать, необходимо было реализовать библиотеку для выполнения операций над конечными автоматами и конечными преобразователями, изменить алгоритм проведения лексического анализа и на основе него реализовать генератор лексических анализаторов.

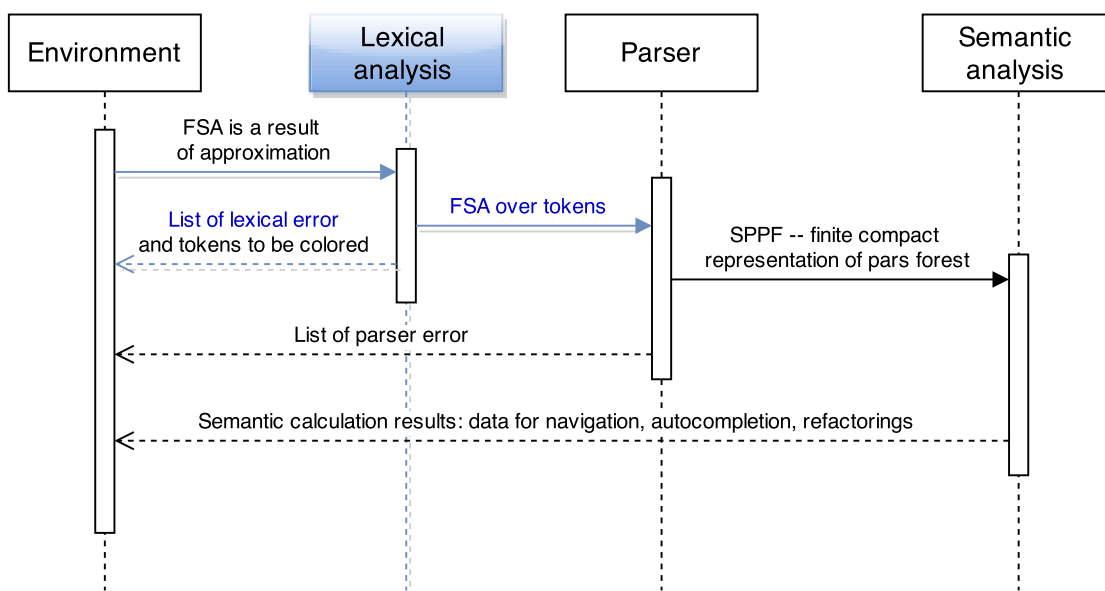


Рис. 3: Диаграмма последовательности, показывающая взаимодействие компонентов инструмента YaccConstructor

3. Алгоритм лексического анализа

Основной задачей лексического анализа является выделение токенов во входном потоке символов, соответствующих спецификации обрабатываемого языка, и сохранение привязки лексических единиц к исходному коду. В классическом лексическом анализе входной поток является линейным, но для проведения лексического анализа динамически формируемого строкового выражения необходима структура, которая являлась бы компактным представлением множества значений этого выражения. Как уже было сказано, такие выражения получаются в результате строковых операций и языковых конструкций языка, с помощью которых эти строки и формируются. В данной работе такой структурой является конечный автомат, который можно представить в виде графа с отмеченными вершинами, соответствующие начальным и конечным состояниям исходного автомата, а ребра этого графа будут содержать строки¹.

В классическом лексическом анализе распространенной практикой является применение генераторов лексических анализаторов, которые по спецификации обрабатываемого языка строят описание конечного преобразователя. На основе него входной поток символов переводятся в поток токенов. Данный подход можно использовать и для лексического анализа динамически формируемых строковых выражений.

Результатом работы лексического анализа динамически формируемого строкового выражения будет являться конечный автомат, каждое ребро которого содержит токен. В классическом лексическом анализе токен можно представить в виде структуры, содержащей идентификатор токена и последовательность символов, выделенной из входного потока. В нашем же случае токен будет представлять структуру, содержащую идентификатор токена и *конечный автомат*, который представляет собой часть множества значений исходного динамически формируемого строкового выражения, которая выделена лексическим анализатором в данный тип токена. При этом для каждого символа хранится информация, из какой строки получен этот символ и координаты его позиций внутри этой строки.

Предлагаемый алгоритм для проведения лексического анализа динамически формируемого строкового выражения состоит из следующих шагов:

Шаг 1. На вход лексическому анализатору подается конечный автомат, являющийся результатом аппроксимации динамически формируемого строкового выражения. Вершины графа соответствуют случаям конкатенации строк, наличие циклов в графе означает, что формирование строк происходило при участии языковых конструкций, например, *for* и *while*, ветвления в графе соответствуют условным операторам.

¹В дальнейшем для конечного автомата и для конечного преобразователя будет использоваться терминология, применимая к графам.

Шаг 2. Входной конечный автомат над строками преобразуется к конечному автомату над символами следующим образом: каждое ребро входного конечного автомата разбивается на последовательность новых ребер, метки которых содержат только по одному символу из строки исходного ребра. На этом этапе происходит сохранение привязки: с каждым символом сохраняются координаты позиций этого символа в исходной строке, а также привязка этой строки к исходному коду. Над полученным конечным автоматом запускаем процедуру построения детерминированного конечного автомата.

Шаг 3. Из конечного автомата, полученного на предыдущем шаге, строим конечный преобразователь следующим образом: для каждого ребра конечного автомата строим новое ребро конечного преобразователя, содержащее пару, первый аргумент которой содержит информацию, хранимую на ребре конечного автомата, то есть символ и его привязку к исходному коду, а второй аргумент — только этот символ.

Шаг 4. Затем происходит применение операции композиции к двум конечным преобразователям, где первый конечный преобразователь получен на предыдущем шаге, а второй — из описания, построенного генератором лексических анализаторов, по спецификации обрабатываемого языка. Результатом этой операции является либо набор лексических ошибок, либо конечный преобразователь, каждое ребро которого содержит пару. Первым элементом этой пары является символ со своей привязкой к исходному коду, а вторым — функция, которая возвращает либо токен, либо ничего.

Шаг 5. На этом шаге происходит интерпретация полученного конечного преобразователя, результатом которой будет являться конечный автомат над токенами.

При поддержке строковых операций особый интерес представляет строковая операция `replace`(M_1, M_2, M_3), где M_1, M_2, M_3 являются детерминированными конечными автоматами, так как поддержка ряда других часто используемых строковых операций может быть реализована на его основе. Конечные автоматы M_1, M_2, M_3 порождают языки $L(M_1), L(M_2), L(M_3)$ соответственно. **Replace** — это операция, которая для любого слова из языка $L(M_1)$ ищет вхождения любого слова из языка $L(M_2)$ и заменяет их на слова из языка $L(M_3)$. Результатом строковой операции `replace` является конечный автомат. При этом есть строковые операции, которые выводят обрабатываемый язык за класс регулярных и контекстно-свободных языков.

Если строковые выражения формировались при участии строковых операций, которые можно выразить с помощью строковой операции `replace`, то для их раскрытия требуется выполнение первых двух шагов описанного алгоритма для каждого участвующего в строковой операции конечного автомата. Далее применяется алгоритм, описанный в статье [1]. Но так как авторы этой статьи используют MBDD-представление для конечного автомата, а текущая реализация инструмента этого не предполагает, то алгоритм был упрощен и стал состоять из следующих шагов

($\#_1, \#_2 \notin \Sigma$, где Σ — входной алфавит конечных автоматов M_1, M_2, M_3):

Шаг 1. Построение конечного автомата M'_1 из M_1 : дублируем состояния конечного автомата M_1 , затем каждое исходное состояние соединяем с дублированным состоянием, соответствующее этому исходному состоянию, переходом по символу $\#_1$, а каждое дублированное состояние соединяем с исходным состоянием — по символу $\#_2$. То есть $L(M'_1) = \{w'|k > 0, w = w_1x_1w_2 \dots w_kx_kw_{k+1} \in L(M_1), w' = w_1\#_1x_1\#_2w_2 \dots w_k\#_1x_k\#_2w_{k+1}\}$.

Шаг 2. Построение конечного автомата M'_2 из M_2 . Для начала построим конечный автомат M_h , который принимает строки, не содержащие в себе любую подстроку из языка $L(M_2)$. То есть $L(M_h)$ является дополнением к множеству $\{w_1xw_2|x \in L(M_2), w_1, w_2 \in \Sigma^*\}$. Затем соединим конечные состояния конечного автомата M_h с начальным состоянием конечного автомата M_2 по символу $\#_1$, а также конечные состояния конечного автомата M_2 с начальным состоянием конечного автомата M_h по символу $\#_2$. То есть $L(M'_2) = \{w'|k > 0, w' = w_1\#_1x_1\#_2w_2 \dots w_k\#_1x_k\#_2w_{k+1}, \forall 1 \leq i \leq k, x_i \in L(M_2), \forall 1 \leq i \leq k+1, w_i \in L(M_h)\}$.

Шаг 3. Построение конечного автомата M' как результат операции пересечения над конечными автоматами M'_1 и M'_2 .

Шаг 4. Результирующий конечный автомат M получается из конечного автомата M' путем замены строк, которые находятся между символами $\#_1$ и $\#_2$, на слова языка $L(M_3)$. То есть $L(M) = \{w|k > 0, w_1\#_1x_1\#_2w_2 \dots w_k\#_1x_k\#_2w_{k+1} \in L(M'_1) \cap L(M'_2), w = w_1c_1w_2 \dots w_kc_kw_{k+1} \forall 1 \leq i \leq k, c_i \in L(M_3)\}$.

После раскрытия операции `replace` можно продолжить выполнение алгоритма для проведения лексического анализа строкового выражения со следующего шага.

Рассмотрим следующий пример кода:

```

1 private void Go(int cond){
2     string columnName = cond > 3 ? "X" : (cond < 0 ? "Y" : "Z");
3     string queryString = "SELECT name" + columnName + " FROM table";
4     Program.ExecuteImmediate(queryString);}

```

Listing 2: Пример кода

Результатом аппроксимации этого кода будет конечный автомат (см рис. 4).

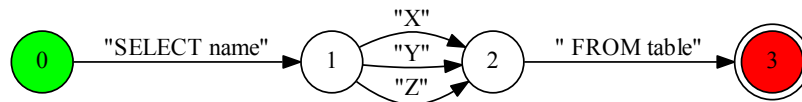


Рис. 4: Результат аппроксимации кода, представленного в листинге 2

Конечный автомат, полученный в результате применения описанного выше алгоритма лексического анализа динамически формируемого строкового выражения к конечному автомату, представленного на рис. 4, показан на рис. 5.



Рис. 5: Результат работы лексера для конечного автомата, представленного на рис. 4

При этом токен `SELECT` содержит в себе конечный автомат, представленный на рис. 6. Конечный автомат первого токена `IDENT` представлен на рис. 7. Второго токена `IDENT` — на рис. 8. Видно, что для каждого символа сохраняются строка, из которой этот символ получен, и координаты этого символа внутри этой строки. Так как токен содержит в себе конечный автомат, а не строку, то получается один идентификатор `IDENT` после `SELECT`, а не три, как это было раньше [14, 16], что упрощает проведение синтаксического разбора в таких случаях.

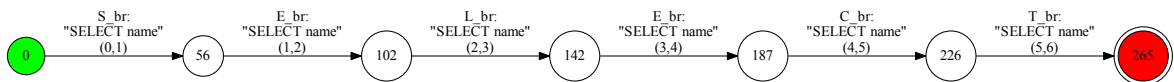


Рис. 6: Конечный автомат, содержащейся в токене `SELECT`

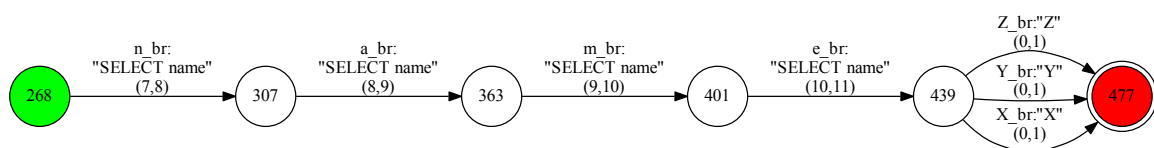


Рис. 7: Конечный автомат, содержащейся в первом токене `IDENT`

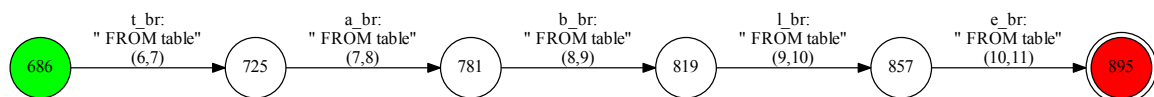


Рис. 8: Конечный автомат, содержащейся во втором токене `IDENT`

4. Особенности реализации

4.1. Архитектура инструмента

В рамках проекта YaccConstructor была создана и реализована архитектура инструмента, реализующая алгоритм для проведения лексического анализа динамически формируемых строковых выражений. Диаграмма компонентов реализованного инструмента представлена на рис. 9.

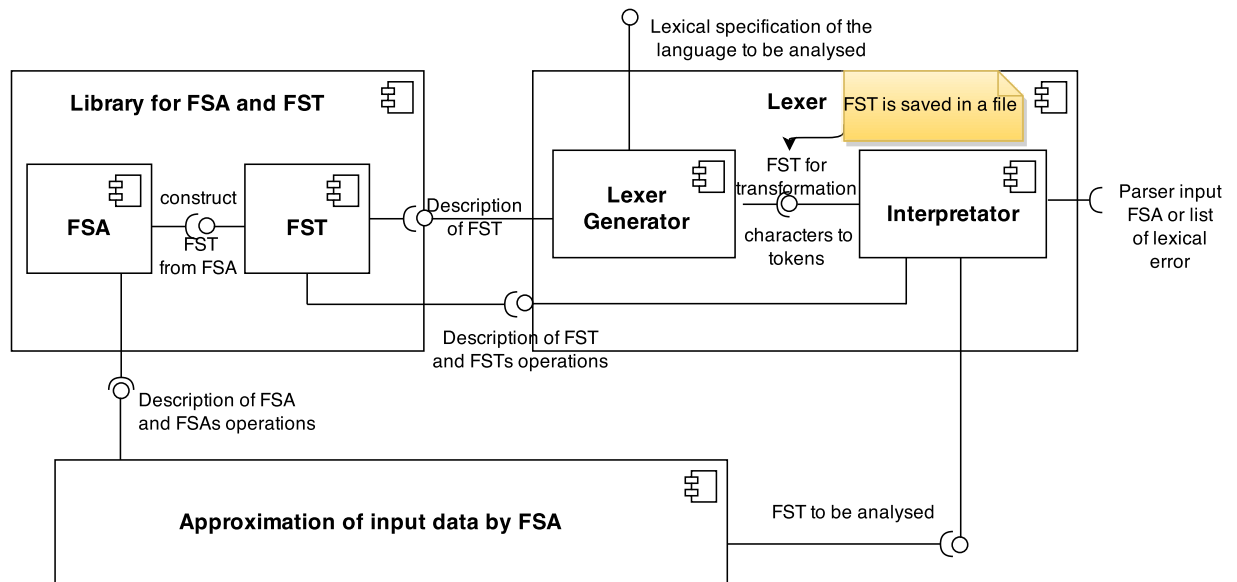


Рис. 9: Диаграмма компонентов реализованного инструмента

Ниже представлено описание каждого компонента.

Компонент **Approximation of input data by FSA** отвечает за аппроксимацию входного динамически формируемого строкового выражения, результатом которой будет конечный автомат. Для реализации этого компонента нужно знать описание конечного автомата и иметь доступ к операциям над ними. Например, при раскрытии операции **Replace** потребуется вызов соответствующей функции в библиотеке (компонент **Library for FSA and FST**).

Компонент **Library for FSA and FST** состоит из двух компонентов: **FSA** (Finite State Automata) и **FST** (Finite State Transducer), которые являются библиотеками для конечного автомата и конечного преобразователя соответственно. Библиотека для конечного автомата используется для построения аппроксимации входных данных, а библиотека для конечных преобразователей — для проведения лексического анализа динамически формируемых строковых выражений. Диаграмма классов этой библиотеки представлена на рис. 11. Для конечного автомата реализовано ряд операций таких как: пересечение, объединение, дополнение, конкатенация, **Replace**, построе-

ние детерминированного конечного автомата из недетерминированного. Для конечного преобразователя реализованы операции композиция, объединение, пересечение и построение конечного преобразователя из конечного автомата с помощью функции преобразования. Язык реализации библиотеки — F#. Для представления конечного автомата и конечного преобразователя в виде графа, ребра которого соответствуют переходам, а вершины — состояниям, использовалась библиотека QuickGraph [23]. Для их визуализации — инструмент Graphviz [19].

Компонент **Lexer** состоит из двух компонентов: **Lexer Generator** и **Interpretator**, которые являются генератором лексических анализаторов и лексическим анализатором соответственно. За основу генератора лексических анализаторов для динамически формируемых строковых выражений взят инструмент FsLex, который по спецификации обрабатываемого языка строит описание конечного преобразователя. Генератор лексических анализаторов FsLex был изменен так, чтобы он строил описание конечного преобразователя, соответствующее реализованной библиотеке (компонент **Library for FSA and FST**). При этом это описание сохраняется в отдельном файле, что позволяет многократно использовать описание конечного преобразователя для обработки одного языка. Лексический анализатор на вход принимает два конечных преобразователя. Первый преобразователь получен в результате преобразования конечного автомата, являющейся результатом аппроксимации обрабатываемого входного динамически формируемого строкового выражения (компонент **Approximation of input data by FSA**), а второй преобразователь взят из описания, построенного генератором лексических анализаторов. Результатом работы лексического анализатора будет являться либо конечный автомат над токенами, либо список лексических ошибок. Диаграмма классов этого компонента представлена на рис. 10. Язык реализации лексера — F#.

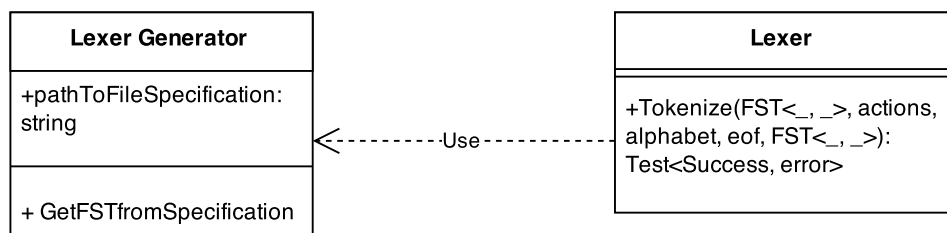


Рис. 10: Диаграмма классов компоненты **Lexer**

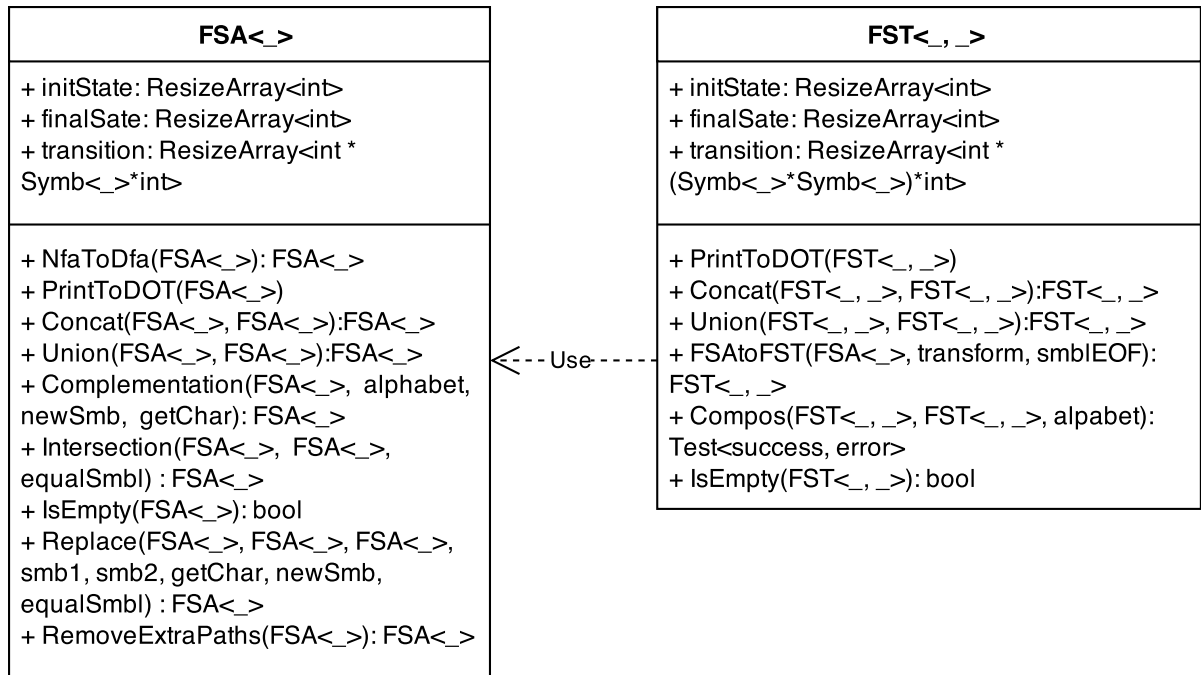


Рис. 11: Диаграмма классов компоненты **Library for FSA and FST**

4.2. Детали реализации

4.2.1. Конечные автоматы

Результатом некоторых операций над конечными автоматами, например, пересечение, может быть пустой конечный автомат. *Пустым конечным автоматом* будем называть граф, у которого не существует пути от начального состояния до какого-либо конечного состояния. Полагаем, что конечный автомат, не содержащий переходов и имеющий только одно состояние, в котором начальное состояние совпадает с конечным, принимает *пустой* язык, но при этом этот автомат не является пустым, так как существует путь от начального состояния до конечного состояния длины 0.

В конечном автомате могут существовать “ненужные” состояния, то есть состояния, которые не лежат на пути от начального состояния до какого-либо конечного состояния. Алгоритм для удаления таких состояний состоит из следующих шагов:

Шаг 1. Считаем, что у конечного автомата только одно начальное состояние. Если конечных состояний несколько, то добавляем одно новое состояние и переходы по **Eps** к этому состоянию от каждого конечного состояния. Считаем добавленное состояние конечным, то есть получаем одно конечное состояние.

Шаг 2. Запускаем обход в глубину (Depth-first search, dfs) от начальной вершины над конечным автоматом, полученным на предыдущем шаге. Удаляем не посещенные вершины из множества всех состояний и из конечных состояний (исходного конечного

автомата).

Шаг 3. Создаем новый конечный автомат, который отличается от текущего конечного автомата тем, что ребра имеют другое направление. То есть получаем инвертированный конечный автомат.

Шаг 4. Запускаем обход в глубину от конечной вершины над конечным автоматом, полученным на предыдущем шаге. Удаляем не посещенные вершины из множества всех состояний и из начального состояния (исходного конечного автомата).

Шаг 5. Удаляем состояние, которое было добавлено на Шаге 1.

Данный алгоритм можно использовать для входного конечного автомата, прежде чем над ним будут проводить какие-либо операции, что позволит сократить количество состояний в этом конечном автомате.

Для того чтобы узнать, является ли конечный автомат пустым, достаточно будет запустить обход в глубину от стартовой вершины и проверить, все ли конечные состояния лежат в множестве не посещенных состояний.

Если в операции $\text{Replace}(M_1, M_2, M_3)$ над конечными автоматами хотя бы один конечный автомат является пустым, то результатом будет являться конечный автомат M_1 . Результат операции Replace , в которой хотя бы один аргумент является конечный автомат с одним состоянием, являющимся и конечным и начальным, и без переходов, представлен ниже.

Пусть M_1 — есть конечный автомат с одним состоянием, являющимся и начальным и конечным, и без переходов. Если M_2 — конечный автомат, который содержит пустое слово, то $\text{Replace}(M_1, M_2, M_3) = M_3$, иначе $\text{Replace}(M_1, M_2, M_3) = M_1$.

Если M_2 — есть конечный автомат с одним состоянием, являющимся и начальным и конечным, и без переходов, то во все состояния конечного автомата M_1 вставляем конечный автомат M_3 .

4.2.2. Лексический анализ

В алгоритме лексического анализа строковых выражений присутствует этап интерпретации конечного преобразователя (**Шаг 5**), полученного в результате операции композиции над двумя конечными преобразователями. Рассмотрим этот процесс более подробно.

Пусть дан входной конечный автомат, представленный на рис. 12.

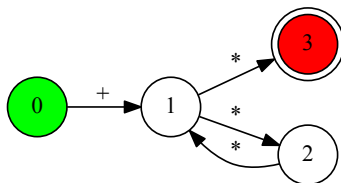


Рис. 12: Входной конечный автомат

Результат композиции конечного преобразователя (преобразованный конечный автомат, **Шаг 3**) с конечным преобразователем, полученным из описания, построенным генератором лексического анализатора по спецификации языка для калькулятора (см. листинг 3), представлен на рис. 13. Где 6 — номер функции, которую возвращает токен PLUS, 7 — номер функции, которую возвращает токен POW, 8 — номер функции, которую возвращает токен MULT.

```

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

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

```

Listing 3: Спецификация языка для калькулятора

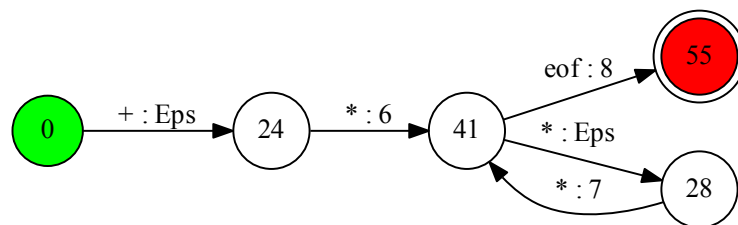


Рис. 13: Результат операции композиция

Для каждого токена сохраняется конечный автомат, который представляет собой часть множества значений исходного динамически формируемого строкового выражения, которая выделена лексическим анализатором в данный тип токена. Чтобы это сделать, необходима структура **GraphAction**, которая хранит одну стартовую action-вершину и несколько конечных action-вершин, а также конечный автомат. *Стартовая action-вершина* — это вершина, из которой выходит ребро, содержащее номер функции. *Конечные action-вершины* — это вершины, в которых заканчивается обход в ширину (Breadth-first search, bfs) из стартовой action-вершины. Условием окончания

является встреча action-вершины или вершины, принадлежащей множеству конечных или стартовых состояний. Конечный автомат хранит результат графа, посещенный обходом в ширину. Алгоритм получения конечного автомата для каждого токена представлен ниже.

Шаг 1. Для конечного преобразователя получаем набор вершин, которые являются стартовыми action-вершинами. Считаем также, что начальное состояние конечного автомата входит в стартовую action-вершину. Запускаем обход в ширину для каждой стартовой action-вершины, который ищет конечные action-вершины. Полученный результат сохраняем в структуре `GraphAction`.

Шаг 2. Создаем новый конечный преобразователь, который получается путем инвертирования ребер исходного конечного преобразователя. Для этого преобразователя также получаем набор вершин, которые являются стартовыми action-вершинами. Конечное состояние также входит в множество стартовых action-вершин. Запускаем обход в ширину для каждой стартовой action-вершины, который ищет конечные action-вершины. Полученный результат сохраняем в структуре `GraphAction`.

Шаг 3. Для каждой структуры `GraphAction`, полученной на Шаге 1, и для каждой конечной action-вершины этой структуры ищем соответствующую структуру `GraphAction`, полученную на Шаге 2, в которой стартовая action-вершина совпадает с текущей конечной action-вершиной. Учитывая, что во второй структуре ребра являются инвертированными, относительно исходного конечного преобразователя, ищем пересечение двух конечных преобразователей. Результатом будет являться токен, идентификатор которого есть номер функции, которую возвращает конечная action-вершина, а также конечный автомат, полученный в результате пересечения (оставив только первые аргументы результата пересечения двух конечных преобразователей), в котором начальное состояние совпадает со стартовой action-вершиной, а конечное — с множеством конечных action-вершин. При этом в конечный автомат, являющийся результатом лексического анализа, добавляется ребро, содержащее этот токен, исходная вершина которого равна стартовой action-вершины, а целевая вершина — текущей конечной action-вершины.

Для рассмотренного примера стартовыми action-вершинами являются следующие вершины: 0, 24, 28, 41. Построим для каждой из них структуру `GraphAction` (см. таблицу 1).

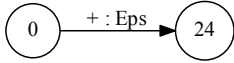

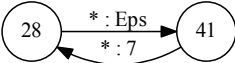
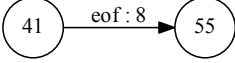
0	24	28	41
			
24	41, 28	28, 41	55

Таблица 1: Структура `GraphAction` для Шага 1

Результат выполнения Шага 2 представлен в таблице 2. При этом конечный автомат показан инвертированным. Также для этого шага не важно, какие конечные action-вершины получаются.

24	28	41	55

Таблица 2: Структура GraphAction для Шага 2

Результат пересечения двух конечных преобразователей, один из которых взят из таблицы 1, а второй — из таблицы 2, при условии что стартовая action-вершина второго конечного преобразователя совпадает с текущей конечной action-вершиной, представлен в таблице 3.

Таблица 3: Результат пересечения конечных преобразователей из таблиц 1 и 2

Результатом лексического анализа конечного автомата, представленного на рис. 12, показан на рис. 14.

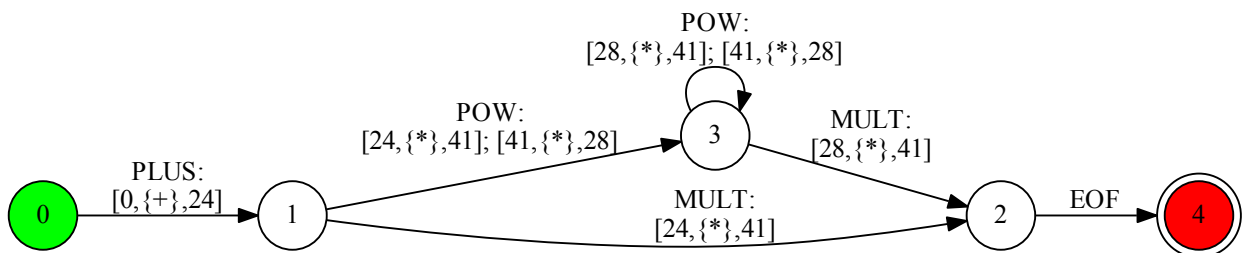


Рис. 14: Результат лексического анализа

5. Апробация

Для данной апробации были подготовлены тесты, на которых реализованный инструмент выполняет поставленную перед ним задачу: обрабатывает случаи, когда строковые выражения были сформированы при участии циклов и строковых операций. Опишем основные этапы сценария тестирования:

Этап 1. Описать спецификацию обрабатываемого языка в файле с расширением `.fsl`. Генератор лексических анализаторов по этой спецификации построит описание конечного преобразователя.

Этап 2. Создать проект, в который добавляем файл с описанием конечного преобразователя. Необходимо также будет описать тип `Token`.

Этап 3. Для тестирования не важно, каким способом получены входные конечные автоматы. Главное, чтобы они удовлетворяли внутреннему описанию конечного автомата.

Этап 4. Когда входные конечные автоматы получены, можно выбрать проводимую операцию над ними. Это может быть, например, обработка операции `replace`, результатом которой является также конечный автомат. Если все необходимые операции уже выполнены, то можно проводить лексический анализ. Для этого необходимо преобразовать конечный автомат в конечный преобразователь. Затем вызвать функцию `tokenize` из сгенерированного файла. В случае успешного проведения лексического анализа, результат запишется в файл с расширением `.dot`, в котором будет описание конечного автомата над токенами. При этом для каждого токена можно распечатать хранимый им конечный автомат.

Рассмотрим примеры, которые показывают важность включения обработки строковых операций и циклов в проводимый анализ.

Пример 1. Рассмотрим следующий пример кода:

```
1 private void Go(){
2     String s = "SELECT nameX FROM tableY";
3     s = s.Replace("SELECT nameX", "b");
4     Program.ExecuteImmediate(s);
5 }
```

Listing 4: Пример кода со строковой операцией `Replace`

В результате выполнения метода `Replace` в переменной `s` будет содержаться строка `"b FROM tableY"`, что приведет к ошибке во время выполнения запроса.

Результат лексического анализа с поддержкой строковой операцией `Replace` представлен на рис. 15. Без поддержки — на рис. 16.

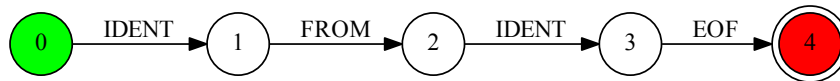


Рис. 15: Результат лексического анализа с поддержкой строковой операции Replace

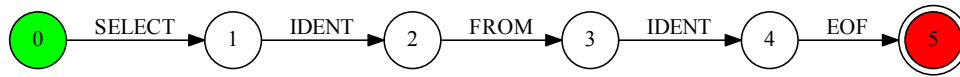


Рис. 16: Результат лексического анализа без поддержки строковой операций Replace

Пример 2. Рассмотрим следующий пример кода:

```

1 private void Go(int number){
2     String s = "SELECT nameX FROM tableY WHERE x < ";
3     while(s.Length < number){ s += "+ 1 ";}
4     Program.ExecuteImmediate(s);
5 }
  
```

Listing 5: Пример кода с циклом while

Результатом аппроксимации кода будет конечный автомат, представленный на рис. 17. Результат лексического анализа представлен на рис. 18.

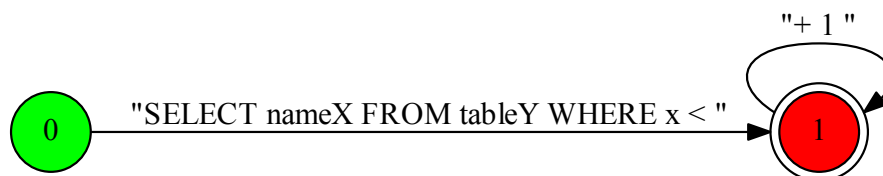


Рис. 17: Результат аппроксимации кода

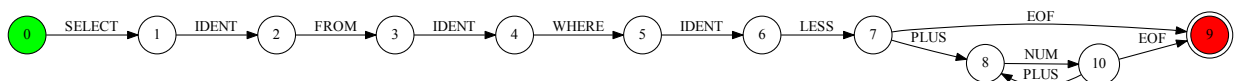


Рис. 18: Результат лексического анализа с поддержкой циклов

Пример 3. Рассмотрим следующий пример кода, в котором конечный автомат токена содержит цикл:

```

1 String s = "SELECT name";
2 for(int i = 0; i < 10; i++){ s += "X";}
3 s+= " FROM tableY";
4 Program.ExecuteImmediate(s);

```

Listing 6: Пример кода, в котором конечный автомат токена содержит цикл

Результат лексического анализа будет конечный автомат, представленный на рис. 19. Конечный автомат первого токена IDENT, содержащий цикл, представлен на рис. 20. На этом же рисунке показана сохраненная привязка символов к исходному коду.



Рис. 19: Результат лексического анализа

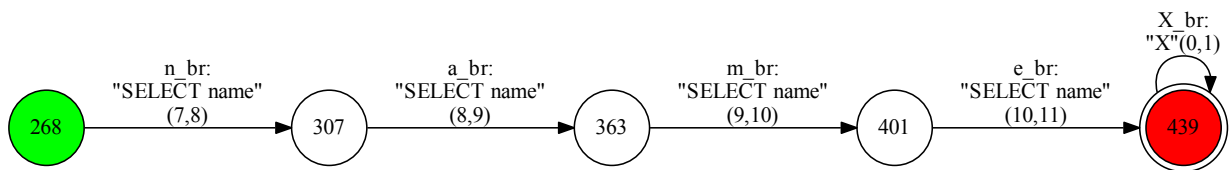


Рис. 20: Конечный автомат первого токена IDENT, содержащий цикл

Таким образом, рассмотренные примеры показывают функциональность реализованного инструмента, которая улучшает точность проводимого анализа строковых выражений. В данной работе не производились замеры производительности реализованного инструмента, потому что целью было создать работоспособный алгоритм, решающий поставленные задачи, который переиспользует существующие решения (алгоритм обработки строковой операции Replace, операции над конечными автоматами и конечными преобразователями, генератор лексических анализаторов FsLex). Оптимизация полученного инструмента за счет подбора структур данных и алгоритмов для работ с конечными автоматами [7] является отдельной задачей и будет решаться в дальнейшем.

Заключение

В рамках выполнения данной работы были получены следующие результаты.

- Предложен алгоритм для лексического анализа строковых выражений, формируемых с помощью циклов и строковых операций, сохраняющий привязку частей динамически формируемого строкового выражения к исходному коду и привязку лексических единиц внутри каждой части.
- Создана и реализована в рамках проекта YaccConstructor архитектура инструмента, реализующая алгоритм. В состав инструмента вошли следующие компоненты:
 - генератор лексических анализаторов на основе библиотеки FsLex, которая по заданной спецификации языка строит описание конечного преобразователя;
 - библиотека для выполнения следующих операций над конечными автоматами: пересечение, дополнение, объединение, конкатенация, replace;
 - библиотека для выполнения следующих операций над конечными преобразователями: объединение, конкатенация и композиция.
- Проведена апробация полученного инструмента.
- Результаты работы вошли в статью “Инструментальная поддержка встроенных языков в интегрированных средах разработки” (Моделирование и анализ информационных систем, Том 21, Номер 6, 2014, ВАК).
- Результаты работы вошли в статью “String-embedded language support in integrated development environment” (CEE-SECR '14 Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia, Article No. 21, ACM New York, NY, USA, 2014)

Код реализованной библиотеки для выполнения операций над конечными автоматами и конечными преобразователями можно найти на сайте <https://github.com/YaccConstructor/YC.FST>. Код инструмента, реализующий алгоритм, можно найти на сайте <https://github.com/YaccConstructor/YaccConstructor>. В указанных репозиториях автор принимал участие под учетной записью polubelova.

В дальнейшем планируется реализовать полную поддержку автоматов, которые могут быть получены при использовании генератора лексического анализатора FsLex [18]. В качестве примера можно привести рекурсивный автомат [12], который в текущей реализации не поддерживается.

Поскольку в проведении лексического анализа активно используются конечные автоматы и конечные преобразователи, то отдельной задачей может стать исследование структур данных и алгоритмов для работ с ними [7] с целью оптимизации времени работы реализованного инструмента.

Более глобальной задачей является реализация инструмента для проведения миграции с одной языковой платформы на другую, поскольку динамически формируемые строковые выражения написаны на каком-то языке, а значит, может возникнуть задача перевода этих строк с одного языка на другой.

Список литературы

- [1] Automata-based symbolic string analysis for vulnerability detection / Fang Yu, Muath Alkhalaf, Tefvik Bultan, Oscar H Ibarra. — 2014. — P. 44–70.
- [2] Biehl Morten, Klarlund Nils, Rauhe Theis. Algorithms for guided tree automata. — 1997. — P. 6–25.
- [3] Breslav Andrey, Annamaa Aivar, Vene Varmo. Using abstract lexical analysis and parsing to detect errors in string-embedded DSL statements / Proceedings of the 22nd Nordic Workshop on Programming Theory. — TUCS General Publication, 2010. — P. 20–22.
- [4] Christensen Aske Simon, Møller Anders, I.Schwartzbach Michael. Precise Analysis of String Expressions / Proc. 10th International Static Analysis Symposium (SAS). — Springer-Verlag: Berlin, 2003. — June. — P. 1–18.
- [5] Doh Kyung-Goo, Kim Hyunha, Schmidt David A. Abstract LR-Parsing // Formal Modeling: Actors, Open Systems, Biological Systems 2011. — 2011. — P. 90–109.
- [6] Hanneforth Thomas. Finite-state Machines: Theory and Applications Unweighted Finite-state Automata / Institut für Linguistik Universität Potsdam.
- [7] Hooimeijer Pieter, Veanes Margus. An Evaluation of Automata Algorithms for String Analysis. — 2011.
- [8] An Interactive Tool for Analyzing Embedded SQL Queries / Aivar Annamaa, Andrey Breslav, Jevgeni Kabanov, Varmo Vene / Programming Languages and Systems. — Springer: Berlin, 2010. — P. 131–138.
- [9] Minamide Yasuhiko. Static approximation of dynamically generated web pages / In Proceedings of the 14th International Conference on World Wide Web, WWW '05. — ACM, 2005. — P. 432–441.
- [10] Mohri Mehryar. Finite-State Transducers in Language and Speech Processing // Computational Linguistics. — 1997. — Vol. 23. — P. 269–311.
- [11] Pitts Andrew M. Lecture Notes on Regular Languages and Finite Automata for Part IA of the Computer Science Tripos. — 2010.
- [12] Tellier Isabelle. Learning Recursive Automata from Positive Examples // New methods in machine learning. — Publications in conferences and workshops, 2006. — P. 775–804.

- [13] Yu Fang, Alkhalaf Muath, Bultan Tevfik. Stranger: An automata-based string analysis tool for php. — 2010. — P. 154–157.
- [14] Вербицкая Екатерина. Абстрактный лексический анализ // Курсовая работа кафедры системного программирования СПбГУ. — 2013. — URL: <http://se.math.spbu.ru/SE/YearlyProjects/2013/YearlyProjects/2013/344/344-Verbitskaya-report.pdf>.
- [15] Кириленко Я.А, Григорьев С.В., Д.А. Авдюхин. Разработка синтаксических анализаторов в проектах по автоматизированному реинжинирингу информационных систем // Научно-технические ведомости Санкт-Петербургского государственного политехнического университета. Информатика. Телекоммуникации. Управление. — 2013. — no. 174. — P. 94 – 98. — URL: http://ntv.spbstu.ru/telecom/article/T3.174.2013_11/.
- [16] Полубелова Марина. Генератор абстрактных лексических анализаторов // Курсовая работа кафедры системного программирования СПбГУ. — 2014. — URL: <http://se.math.spbu.ru/SE/YearlyProjects/2014/YearlyProjects/2014/344/344-Polubelova-report.pdf>.
- [17] Сайт проекта Alvor. — URL: <http://code.google.com/p/alvor/>.
- [18] Сайт проекта FsLex. — URL: <http://fsprojects.github.io/FsLexYacc/>.
- [19] Сайт проекта Graphviz. — URL: <http://www.graphviz.org/>.
- [20] Сайт проекта Java String Analyzer. — URL: <http://www.brics.dk/JSA/>.
- [21] Сайт проекта MONA. — URL: <http://www.brics.dk/mona/>.
- [22] Сайт проекта PHP String Analyzer. — URL: <http://www.score.cs.tsukuba.ac.jp/~minamide/phpsa/>.
- [23] Сайт проекта QuickGraph. — URL: <https://quickgraph.codeplex.com/>.
- [24] Сайт проекта ReSharper. — URL: <https://www.jetbrains.com/resharper/>.
- [25] Сайт проекта YaccConstructor. — URL: <https://github.com/YaccConstructor/>.