Правительство Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

«Санкт-Петербургский государственный университет»

Кафедра системного программирования

Мавчун Екатерина Валерьевна

Сравнение алгоритмов табличного восходящего синтаксического анализа

Дипломная 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» Подкопаев А.В.

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY

Chair of Software Engineering

Ekaterina Mavchun

Comparison of table-based bottom-up parsers

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific supervisor:
master of Information Technology, senior lecturer Semyon Grigoriev

Reviewer:
software developer at «IntelliJ Labs» Anton Podkopaev

Saint-Petersburg
2015

Оглавление

Введение **4**

1. Постановка задачи **7**

2. Обзор **8**

2.1. Восходящий синтаксический анализ 8

2.2. LR(k)-анализаторы 8

2.3. Алгоритмы, выбранные для исследования 10

2.4. Проект YaccConstructor 12

2.5. Проект Facio 14

3. Интеграция YaccConstructor с Facio **15**

3.1. Преобразование внутренних представлений (IL) 16

3.1.1. Внутреннее представление YaccConstructor 16

3.1.2. Внутреннее представление Facio 20

3.1.3. Реализация преобразования IL 21

3.2. Выбор генератора на стороне Facio 22

3.3. Добавление преобразования общего вида 23

4. Эксперименты **24**

4.1. Методика проведения эксперимента 24

4.2. Эксперимент 1 26

4.3. Эксперимент 2 30

4.4. Вывод 33

Заключение **34**

Список литературы **35**

Введение

Синтаксический анализатор (парсер) — это программа, которая проверяет текст на соответствие некоторой грамматике и по линейной последовательности токенов этого текста строит дерево разбора (синтаксическое дерево) [2], которое хорошо подходит для дальнейшей обработки и анализа текста. Синтаксические анализаторы широко используются, например, для создания компиляторов, также они используются в лингвистике, для машинного перевода. Поэтому создание синтаксических анализаторов является важной задачей.

Существует множество различных типов алгоритмов синтаксического анализа, одним из которых является *восходящий анализ* (bottom-up analysis) [7], который поддерживает LR(k)-грамматики [10], рассматривая цепочки токенов слева направо, и строит правый вывод. При этом узлы синтаксического дерева строятся от листьев (терминальных символов) к корню (начальному символу грамматики). Одним из подходов к реализации восходящего синтаксического анализатора является табличный способ: для грамматики строятся таблица действий и таблица переходов. Первая позволяет задавать следующую операцию анализатора в зависимости от входного терминального символа и текущего состояния анализатора. А вторая определяет состояние в зависимости от нетерминального символа и текущего состояния.

Так как создавать анализаторы вручную достаточно трудоёмко, то разработаны и используются специальные генераторы анализаторов, например, Yacc [12], Bison [4], ANTLR [1]. Известно [13], что для любой контекстно-свободной грамматики можно построить анализатор, сложность которого не превышает O(n^3), где n — это длина входного текста. Но анализаторы реально используемых языков программирования

обычно имеют линейную сложность, то есть для заданного языка существует такая грамматика, которая позволяет создать более быстрый анализатор.

В книге [7] доказывается, что известные алгоритмы табличного восходящего анализа — LR(0), LR(1), SLR(1), LALR(1) — имеют линейную сложность. Однако востребовано практическое исследование производительности данных алгоритмов, так как на практике серьёзную роль играют различные константы. То есть на практике производительность анализаторов, соответствующих вышеперечисленным алгоритмам, может оказаться различной. Таким образом, практические и теоретические результаты могут отличаться. Следовательно, для перечисленного набора алгоритмов необходимо на одинаковых входных данных исследовать их производительность. Теоретические сравнительные исследования, в основном, подкрепляются «искусственными» примерами, поэтому важно провести сравнения на более реалистичных грамматиках. При этом целесообразно иметь единую платформу для экспериментов с тем, чтобы иметь возможность запускать различные анализаторы, реализующие соответствующие алгоритмы, и сравнивать время работы на фиксированных входных данных.

На кафедре системного программирования был разработан инструмент для создания синтаксических анализаторов и обработки грамматик YaccConstrutor (YC) [15]. В этом инструменте созданы различные генераторы для создания синтаксических анализаторов: RNGLR, FsYacc и др. [15]. Также существует сторонний инструмент для создания синтаксических и лексических анализаторов — Facio [14], который поддерживает ряд таких алгоритмов восходящего синтаксического анализа, как LR(0), LR(1), SLR(1), LALR(1). Выбор именно этого инстру-

мента обусловлен тем, что он разработан на платформе .NET [9], язык программирования F# аналогично инструменту YC. Также Facio содержит необходимые алгоритмы синтаксического анализа. Таким образом, требуется получить унифицированный инструмент, позволяющий работать с алгоритмами обоих инструментов.

В качестве основы единой платформы, предлагается использовать два инструмента — YC и Facio. Таким образом, чтобы получить единую среду с различными алгоритмами, возникает задача интеграции этих инструментов. Такая платформа может быть использована в учебных целях для изучения и наглядного сравнения различных алгоритмов восходящего синтаксического анализа. Также добавление алгоритмов в YC позволит без лишних усилий создавать необходимые для других задач анализаторы.

1. Постановка задачи

Целью данной работы является сравнение производительности различных алгоритмов табличного восходящего синтаксического анализа, поскольку теоретические и практические результаты могут отличаться. Известно, что рассматриваемые алгоритмы линейны, но важно исследовать их производительность относительно друг друга. Для достижения этой цели необходимо решить следующие задачи:

- реализовать интеграцию YaccConstructor и Facio;
- выполнить сравнение алгоритмов табличного восходящего синтаксического анализа, то есть провести замеры времени работы сгенерированных синтаксических анализаторов в зависимости от размеров входного текста.

Практическую реализацию данной работы можно использовать в учебных целях: есть единая среда с некоторым набором генераторов, можно на практике сравнивать производительность синтаксических анализаторов, сгенерированных с помощью различных алгоритмов. Такое сравнение является корректным, поскольку используется один интерпретатор — программа, производящая пооператорный анализ, обработку и выполнение исходной программы.

К тому же инструмент YC активно развивается и используется при решении различных задач, поэтому добавление новых генераторов в YC необходимо, например, для анализа встроенных языков (абстрактного анализа) [8].

2. Обзор

2.1. Восходящий синтаксический анализ

Восходящие алгоритмы синтаксического анализа строят синтаксическое дерево снизу вверх, то есть от листьев (терминальных символов) к корню (начальному символу грамматики). При этом исходная цепочка «сворачивается» к аксиоме грамматики: подстрока входной цепочки сопоставляется с правой частью правила грамматики, которое заменяется соответствующим этому правилу нетерминалом, то есть левой частью правила. Так как восходящие анализаторы способны анализировать большое количество разнообразных грамматик, то они используются довольно часто, поэтому для данных алгоритмов созданы генераторы — программы, создающие анализаторы.

2.2. LR(k)-анализаторы

Восходящий синтаксический анализ позволяет рассматривать LR(k)-грамматики. Рассмотрим аббревиатуру LR(k) более детально:

- L обозначает, что входная цепочка просматривается слева направо (Left-to-right scan);
- R обозначает, что строится правый вывод цепочки (Rightmost derivation);
- не более k символов используется для принятия решения на каждом шаге.

При LR(k)-анализе применяется метод «перенос»-«свёртка» (shift-reduce) с использованием стекового автомата [7]. Идея метода заключается в том, что символы входной цепочки переносятся в стек до тех пор,

пока на вершине стека не накопится цепочка, совпадающая с правой частью какого-либо из правил (операция «перенос»). Далее все символы этой цепочки извлекаются из стека, и на их место помещается нетерминал, соответствующий этому правилу (операция «свёртка»). Входная цепочка допускается автоматом, если после переноса в автомат последнего символа входной цепочки и выполнения операции «свёртка» в стеке окажется только аксиома грамматики.

Бывают ситуации, когда при анализе некоторой цепочки анализатор не может решить, делать «сдвиг» или «свёртку» (конфликт «сдвиг» /«свёртка»), или не может решить, какую из нескольких «свёрток» применить (конфликт «свёртка»/«свёртка»).

Анализатор состоит из следующих частей: входная цепочка, выход, стек, управляющая программа, таблица действий и таблица переходов. Схема анализатора представлена на рис. 1.

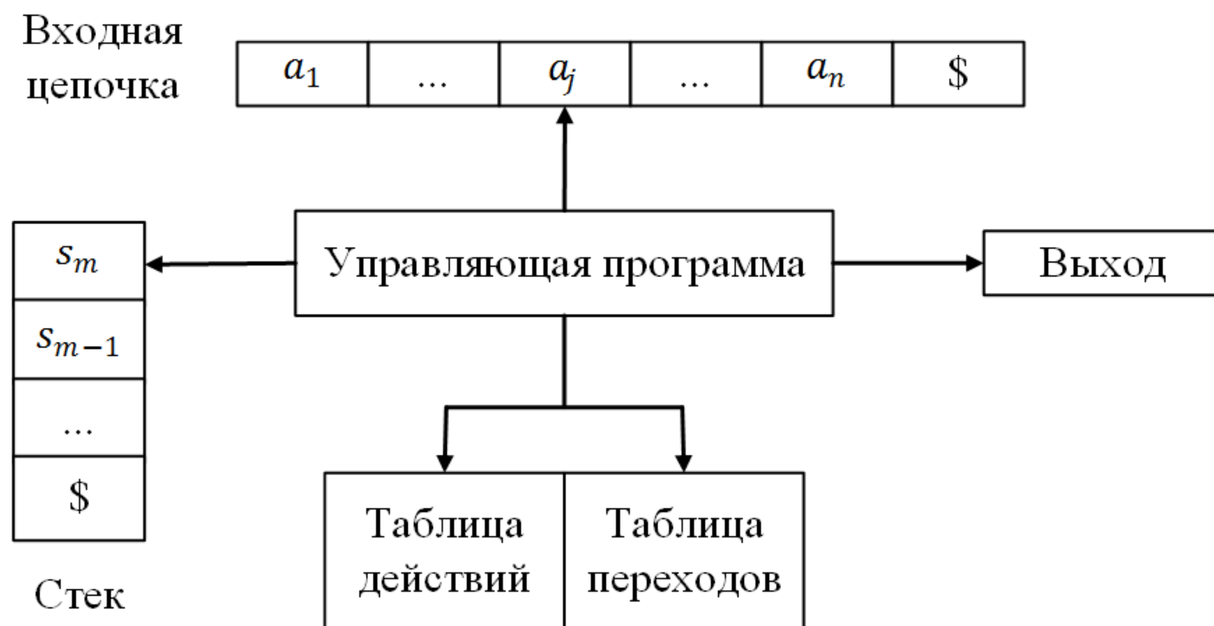

Рис. 1: Схема анализатора, генерируемого алгоритмом табличного восходящего синтаксического анализа [13]

Как уже было описано ранее, при выполнении табличного синтаксического анализа для данной грамматики строятся две таблицы: таблица

действий и таблица переходов. Таблица переходов — это вспомогательная таблица, используется при одном из действий и может содержать следующие значения:

- S — символ состояния;
- error — ошибка.

Таблица действий определяет дальнейшее действие в текущем состоянии и с текущим символом на входе. Каждый элемент таблицы действий может содержать одно из четырёх значений:

- accept («успех») — разбор входной цепочки завершился успешно;
- shift («перенос») — на вершину стека переносится состояние, которое соответствует входному символу, читается следующий символ;
- reduce («свёртка») — в стеке набрались состояния, которые можно заменить одним, исходя из правил грамматики; значение берётся из таблицы переходов;
- error («ошибка») — анализатор обнаружил ошибку во входной цепочке.

2.3. Алгоритмы, выбранные для исследования

Восходящим анализаторам соответствуют LR(k)-грамматики. Иерархия подклассов этих грамматик представлена на рис. 2.

Грамматика является LR(k)-грамматикой, если существует анализатор, написанный для этой грамматики, который читает входные данные слева направо и использует для предпросмотра не более k символов.

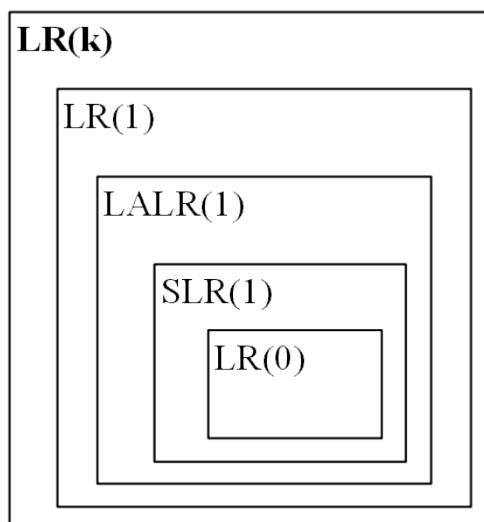

Рис. 2: Иерархия подклассов контекстно-свободных грамматик [3]

Эти грамматики могут быть проанализированы восходящими синтаксическими анализаторами.

Для сравнения были выбраны классические, наиболее часто используемые алгоритмы табличного синтаксического восходящего анализа, соответствующие представителям основных подклассов грамматик: LR(0), LR(1), LALR(1), SLR(1). Перечисленные алгоритмы описаны в книге «Parsing Techniques: A Practical Guide» [7].

Рассмотрим более детально выбранные для исследования алгоритмы табличного восходящего синтаксического анализа.

1. **LR(0)**

   Left-to-right scan Rightmost derivation (0) — генератор синтаксического анализатора для LR(0)-грамматик. Для анализа использует только содержимое стека.

2. **LR(1)**

   Left-to-right scan Rightmost derivation (1) — генератор синтаксического анализатора для LR(1)-грамматик. Для принятия решения использует один символ входной цепочки.

3. **SLR(1)**

   Simple LR(1) — Simple LR(1). Более мощный алгоритм, чем LR(0), но менее мощный, чем LR(1) и LALR(1). Количество состояний, т.е. строк таблицы, такое же как у LR(0), но данный алгоритм автоматически решает некоторые конфликты.

4. **LALR(1)**

   Look-Ahead LR(1) — Look-Ahead LR(1). Упрощённый LR(1)-алгоритм, достаточно мощный для того, чтобы провести синтаксический анализ большинства языков, избегая при этом больших таблиц, в отличие от LR(1)-анализатора. При использовании данного алгоритма можно получить таблицы меньшего размера в результате слияния любых двух состояний, которые совпадают с точностью до символов входной строки, то есть таблицы LALR(1)-анализатора получаются из таблиц LR(1)-анализатора слиянием «эквивалентных» состояний в одно. Количество состояний такое же как у LR(0). Данный алгоритм способен решать большее количество конфликтов автоматически, чем SLR(1).

2.4. Проект YaccConstructor

YC [16] — это инструмент, который используется для создания синтаксических анализаторов и обработки грамматик, позволяет создавать анализаторы с использованием различных алгоритмов синтаксического анализа, таких как RNGLR, FParsec, FsYacc и др. YC разработан на платформе .NET [9], язык программирования — F# [6]. YC имеет модульную структуру, которая изображена на рис. 3. Более детальное описание компонент:

- *Frontend* — языки задания атрибутной грамматики, которые поддерживает YC, например, Yard, FsYacc, AntLR;
- *Intermediate Language* — внутреннее представление анализатора;
- *Conversions* — общие преобразования грамматик, например, преобразование, приводящее грамматику к нормальной форме Хомского (CNF);
- *Backend* — при помощи него на основе внутреннего представления генерируется конечный продукт для пользователя. В том числе это и алгоритмы синтаксического анализа: FParsec, FsYacc, RNGLR, YardPrinter, GLL.

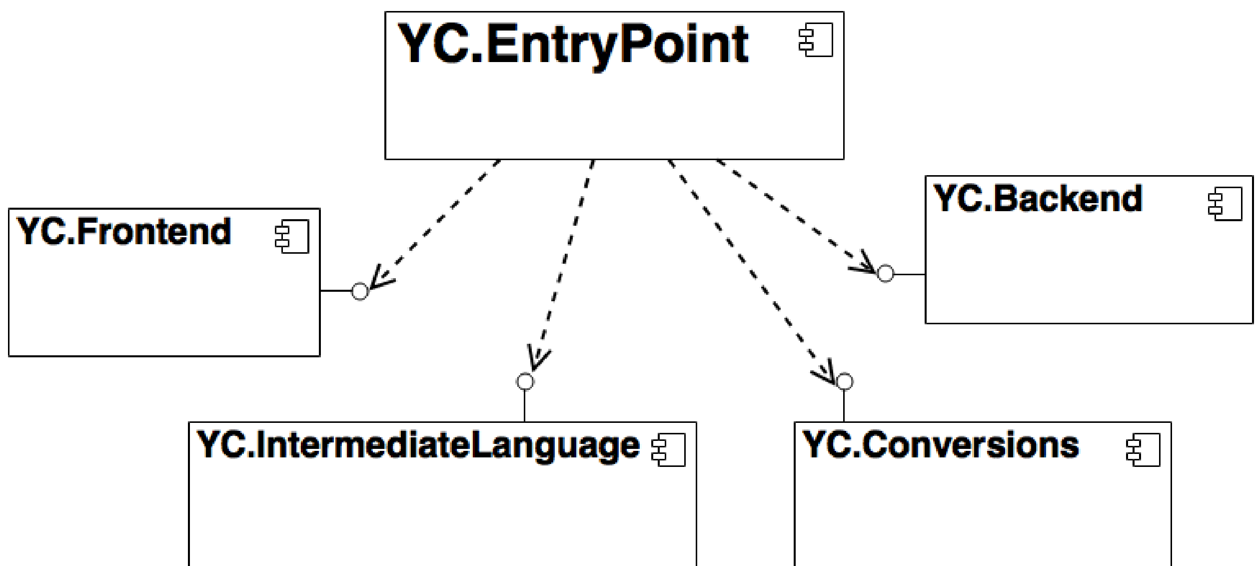

Рис. 3: Архитектура инструмента YC

Входная грамматика, описанная на одном из поддерживаемых языков описания трансляций, преобразуется во внутреннее представление (IL — Intermediate Language), к внутреннему представлению применяются общие преобразования, и далее по внутреннему представлению конструируется таблица анализатора, соответствующая указанному генератору.

2.5. Проект Facio

Facio — это инструмент, предназначен для создания лексических и синтаксических анализаторов, включает в себя библиотеку для создания, обработки и анализа контекстно-свободных грамматик. Facio реализован на платформе .NET [9], язык программирования — F# [6]. Структура инструмента Facio изображена на рис. 4. Facio состоит из следующих компонент:

- *Frontend* — язык задания атрибутной грамматики, например, FsYacc;
- *Intermediate Language* — внутреннее представление анализатора;
- *Backend* — при помощи него на основе внутреннего представления генерируется конечный продукт для пользователя. В том числе это и алгоритмы синтаксического анализа, такие как LR(0), LR(1), LALR(1), SLR(1).

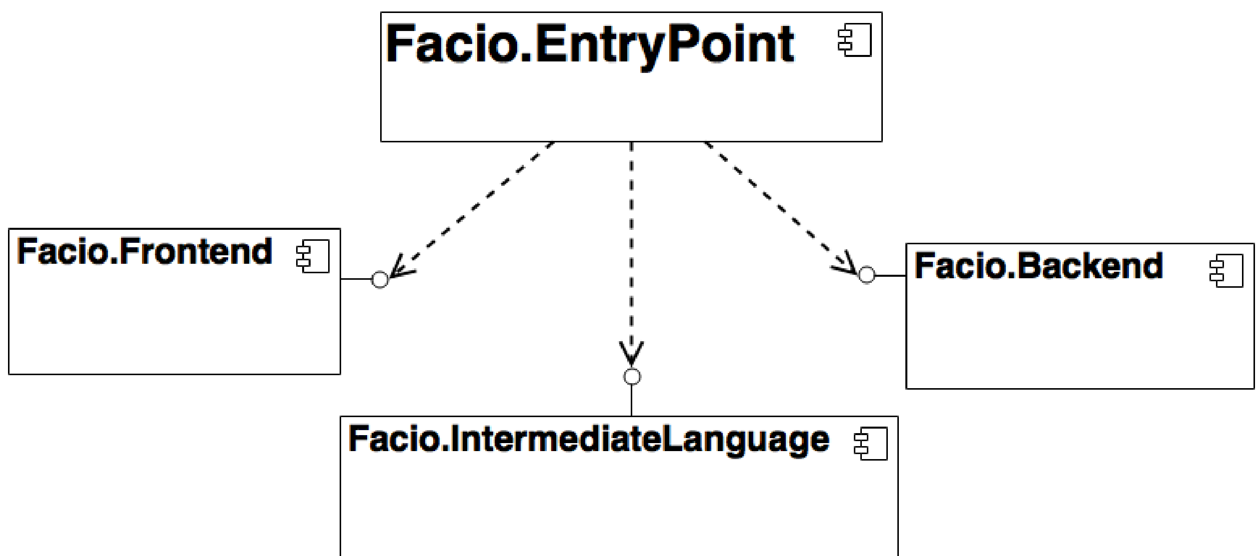

Рис. 4: Архитектура инструмента Facio

3. Интеграция YaccConstructor с Facio

Как уже было описано ранее, Facio позволяет конструировать синтаксические анализаторы с использованием различных алгоритмов, которые не реализованы в YC. Чтобы иметь единую среду, поддерживающую алгоритмы, реализованные и в YC и в Facio, необходимо реализовать интеграцию этих инструментов. Как было указано ранее, оба инструмента реализованы на одном языке и на одной платформе.

Структура инструмента, получившегося в результате интеграции, изображена на рис 5:

- *YC.FacioBackend* — модуль, преобразующий внутреннее представление, оформлен как генератор, который можно указать в аргументах при запуске YC. То есть после того, как входная грамматика была преобразована во внутреннее представление YC, над этим внутренним представлением происходит преобразование форматов, в результате которого получается внутреннее представление Facio. После этого есть возможность использовать любой генератор, реализованный в рамках проекта Facio;

- *YC.NoAlt.Conversion* — реализованное общее преобразование грамматик (удаление верхнеуровневых альтернатив);

- *Facio* — модуль, соответствующий инструменту Facio (рис. 4).

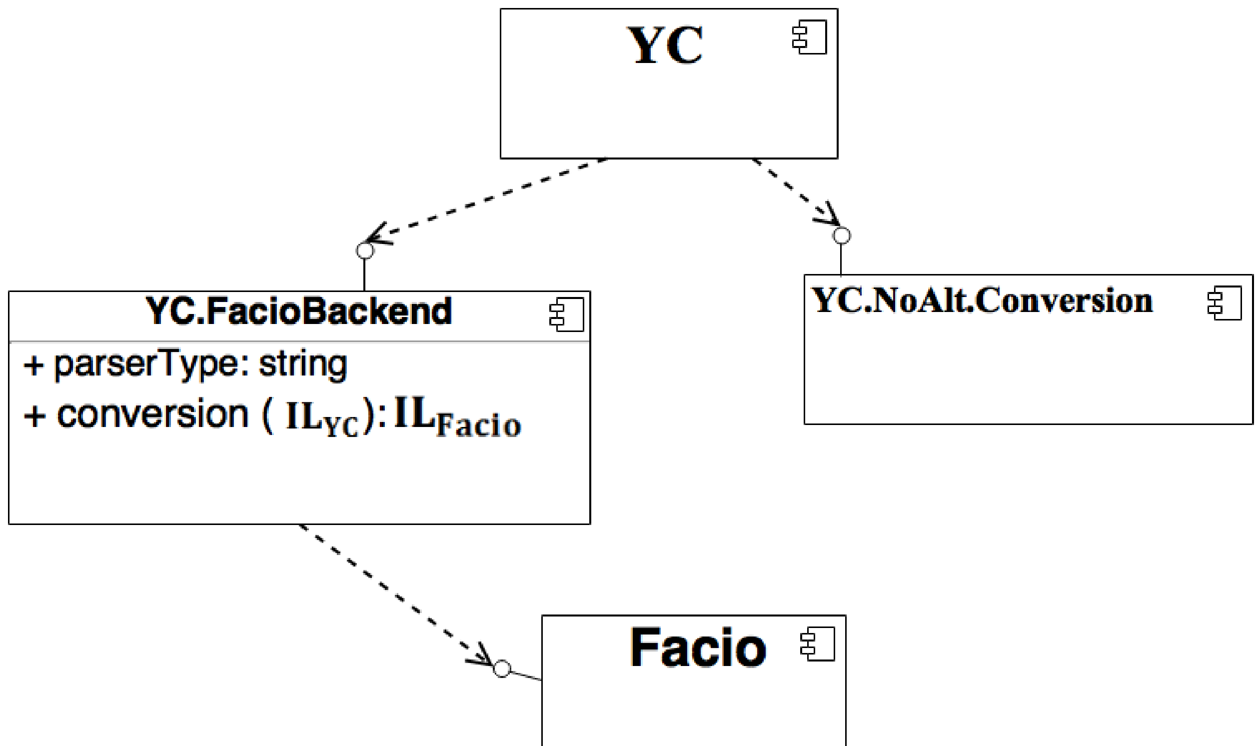

Рис. 5: Интеграция YC с Facio

3.1. Преобразование внутренних представлений (IL)

Каждый из инструментов имеет своё внутреннее представление, то есть в каждом инструменте по-разному хранятся данные (терминалы, нетерминалы, правила и т.д.). Таким образом, необходимо реализовать набор преобразований одного внутреннего представления в другое.

Рассмотрим внутреннее представление для каждого из инструментов.

3.1.1. Внутреннее представление YaccConstructor

Язык описания трансляций Yard представляет грамматику в виде, описанном на листинге 1.

```
grammar: ('{'header'}')? rules ('{'footer'}')?
```

Listing 1: Грамматика, описанная на Yard

Здесь: *header* — текст перед описанием грамматики, например, объявление импорта, начинающееся с ключевого слова open; *footer* — текст после описания грамматики; *rules* — правила грамматики.

Входная грамматика преобразуется во внутреннее представление. В YC в роли внутреннего представления анализатора выступает модуль *Definition* (листинг 2), который содержит следующие элементы:

- *info* — информация (например, происхождение) об этом описании грамматики;
- *head* — текст перед описанием грамматики;
- *grammar* — само описание грамматики;
- *foot* — текст после описания грамматики;
- *tokens* — токены с указанием типа;
- *'patt* — тип атрибутов (аргументов);
- *'expr* — тип выражения в семантическом действии (*action code*).

```
module Definition =
    type info = { fileName : string }
    type t<'patt,'expr
        when 'patt : comparison and 'expr : comparison> = {
        info     : info;
        head     :'expr option;
        grammar  : Grammar.t<'patt,'expr>;
        foot     :'expr option;
        options  : Map<string, string>
        tokens   : Map<string, string option>
    }
```

Listing 2: Внутреннее представление синтаксического анализатора в инструменте YC

В свою очередь, *Grammar* (листинг 3) — это список элементов типа *Module*:

- тип *Module* — список правил;
- *allPublic* — указывает все ли правила являются публичными, то есть видны из других модулей.

```
module Grammar =
    type Module<'patt,'expr> = {
        rules     : Rule.t<'patt,'expr> list
        openings  : Source.t list
        name      : Source.t option
        allPublic : bool
    }
    type t<'patt,'expr> = Module<'patt,'expr> list
```

Listing 3: Внутреннее представление грамматики в инструменте YC

Каждое правило имеет следующую структуру (листинг 4):

- *name* — имя правила, используется для того, чтобы ссылаться на данное правило из других правил;
- *args* — наследуемые аргументы правила;
- *body* — тело правила (продукция);
- *isStart* — указывает, является ли это правило стартовым, если является, то перед правилом пишется «[<Start>]»;
- *isPublic* — указывает является ли правило публичным (перед правилом указано «public»);
- *metaArgs* — список мета-аргументов (имён правил, параметризующих данное правило).

```
module Rule =
    type t<'patt,'expr> = {
        name     : Source.t
        args     :'patt list
        body     : (Production.t<'patt,'expr>)
        isStart  : bool
        isPublic : bool
        metaArgs :'patt list
    }
```

Listing 4: Внутреннее представление правил в инструменте YC

Продукция имеет структуру, указанную в листинге 5.

- *omit* — указывает, нужно ли включать правило в абстрактное синтаксическое дерево (AST);
- *rule* — правило, соответствующее данной продукции;
- *binding* — связывание, позволяет писать, например, f:F или f:=F;
- *checker* — условие в продукции;
- *t* — тип узла продукции в дереве вывода, может принимать следующие значения:
  - *PAlt* — альтернатива: (e1 | e2);
  - *PSeq* — последовательность атрибутов;
  - *PToken* — токены, конечные элементы синтаксического анализа;
  - *PRef* — ссылка на другой нетерминал внутри продукции (вместе с дополнительным списком аргументов).

    Также *t* может принимать значения *PMany*, *PMetaRef*, *PLiteral*, *PRepet*, *PPerm*, *PSome*, *POpt*.

```
module Production =
    type IRuleType
    type DLabel = {
        label  : string;
        weight : float option
    }
    type elem<'patt,'expr> = {
        omit    : bool;
        rule    : (t<'patt,'expr>);
        binding :'patt option;
        checker :'expr option
    }
    and t<'patt,'expr> =
        |PAlt     of (t<'patt,'expr>) * (t<'patt,'expr>)
        |PSeq     of (elem<'patt,'expr>) list * 'expr option * DLabel option
        |PToken   of Source.t
        |PRef     of Source.t * 'expr option
        ...
```

Listing 5: Внутреннее представление продукции в инструменте YC

3.1.2. Внутреннее представление Facio

В инструменте Facio спецификация синтаксического анализатора (листинг 6) для грамматики содержит следующие элементы:

1. *Header* и *Footer* — текст перед и после описания грамматики соответственно;

2. *NonterminalDeclarations* — ассоциативный массив для стартовых нетерминалов: ключ — тип нетерминалов, значение — список нетерминалов, имеющих этот тип.
   StartingProductions — список стартовых нетерминалов грамматики;

3. *TerminalDeclarations* — ассоциативный массив: ключ — тип тер-

миналов, значение — список терминалов, имеющих этот тип;

4. *Associativities* — явное объявление ассоциативности терминалов;

5. *Productions* — правила, представляющие собой список пар *(нетерминал, список продукций)*, где каждый элемент списка продукций содержит:

   - символы (список терминалов или нетерминалов, соответствующих текущей продукции);

   - семантическое действие, которое будет выполнено в результате применении данного правила.

```
type Specification = {
    Header : CodeFragment option;
    Footer : CodeFragment option;
    NonterminalDeclarations : (DeclaredType * NonterminalIdentifier) list;
    TerminalDeclarations : (DeclaredType option * TerminalIdentifier list) list;
    StartingProductions : NonterminalIdentifier list;
    Associativities : (Associativity * TerminalIdentifier list) list;
    Productions : (NonterminalIdentifier * ProductionRule list) list;
}
```

Listing 6: Внутреннее представление синтаксического анализатора в инструменте Facio

3.1.3. Реализация преобразования IL

В рамках задачи интеграции на стороне YC был добавлен генератор со своим преобразованием, которое позволяет перевести внутреннее представление YC во внутреннее представление Facio.

Рассмотрим основные методы, реализованные в генераторе, позволяющие из внутреннего представления YC получить внутреннее представление Facio:

1. Поля *header* и *footer* берутся без изменений.

2. Получение стартовых нетерминалов осуществляется путём обхода всех правил грамматики и проверки значения поля *isStart.* Далее в качестве типа добавляется символ «_» для автоматического вывода типа.

3. Множество всех терминалов получается путём обхода всех правил грамматики и сравнения типа продукции с PToken. Также есть функция, возвращающая ассоциативный массив, который сопоставляет каждому используемому типу список терминалов этого типа. Терминалы и их типы извлекаются из поля tokens модуля *Definition*, если для какого-то токена тип не указан, то берётся тип по умолчанию, указанный в аргументах.

4. Ассоциативность не используется в YC.

5. Чтобы получить продукции в нужном формате, необходимо обойти все правила грамматики, извлечь информацию об аргументах и семантическом действии. Аргументы и семантическое действие для правила, берутся из аргументов *PSeq.*

3.2. Выбор генератора на стороне Facio

Если используется ранее реализованный в YC генератор, то необходимо просто указать этот генератор в аргументах YC. Для случая, когда используется генератор, реализованный в Facio, добавлен ключ *parserType.* То есть на стороне YC необходимо выбрать генератор, осуществляющий преобразование внутренних представлений, но также необходимо предоставить возможность выбирать генератор на стороне Facio.

Ключ *parserType* также указывается в качестве аргумента при запуске YC.

Кроме того добавлен механизм, позволяющий в зависимости от указанного значения ключа *parserType* вызывать соответствующий алгоритм генерации синтаксического анализатора.

3.3. Добавление преобразования общего вида

Иногда грамматикам предъявляют некоторые требования, например, из-за особенностей конкретного генератора синтаксических анализаторов. В YC такие требования реализованы в виде ограничений (*Constraints*). Ограничение проверят конкретное свойство грамматики. Также в YC имеются преобразования грамматик общего вида (*Conversions*). Каждому ограничению соответствует некоторое преобразование, позволяющее привести грамматику к нужному виду. Например, есть проверка на то, что грамматика приведена к нормальной форме Хомского.

В ходе работы выяснилось, что ни один из ранее реализованных в YC алгоритмов синтаксического анализа не выдвигал к грамматике требования отсутствия верхнеуровневых альтернатив, поэтому данное преобразование было реализовано.

4. Эксперименты

4.1. Методика проведения эксперимента

Определим цель эксперимента как сравнение производительности набора алгоритмов табличного восходящего синтаксического анализа, а именно: LR(0), LR(1), SLR(1), LALR(1). Также необходимо выяснить, корректно ли выполнена интеграция инструментов YC и Facio. Следуя [11] сформируем следующие вопросы, на которые эксперимент должен ответить.

1. Сравнить быстродействие исследуемых алгоритмов.
2. Определить, какой из алгоритмов является самым быстрым / медленным по времени выполнения синтаксического анализа.

Для ответов на эти вопросы была использована следующая метрика: измерение времени работы синтаксических анализаторов, сгенерированных с помощью различных алгоритмов, на одинаковых входных данных.

Параметры компьютера, на котором проводились эксперименты были следующими:

- операционная система: Windows 8.1;
- процессор: AMD FX(tm)-6100 Six-Core Processor 3.30 GHz;
- ОЗУ: 16 ГБ.

Все рассматриваемые алгоритмы реализованы на языке программирования F# [6], то есть выбранные алгоритмы сравниваются в одинаковых условиях.

Схема эксперимента изображена на рис.6 и заключается в следующем. Есть грамматика некоторого языка и лексическая спецификация, также есть последовательность входных цепочек разной длины. Для грамматики генерируются синтаксические анализаторы, соответствующие исследуемым алгоритмам. По лексической спецификации генерируется лексический анализатор. Интерпретатор используется один, независимо от выбранного алгоритма, поэтому можно проводить сравнение алгоритмов.

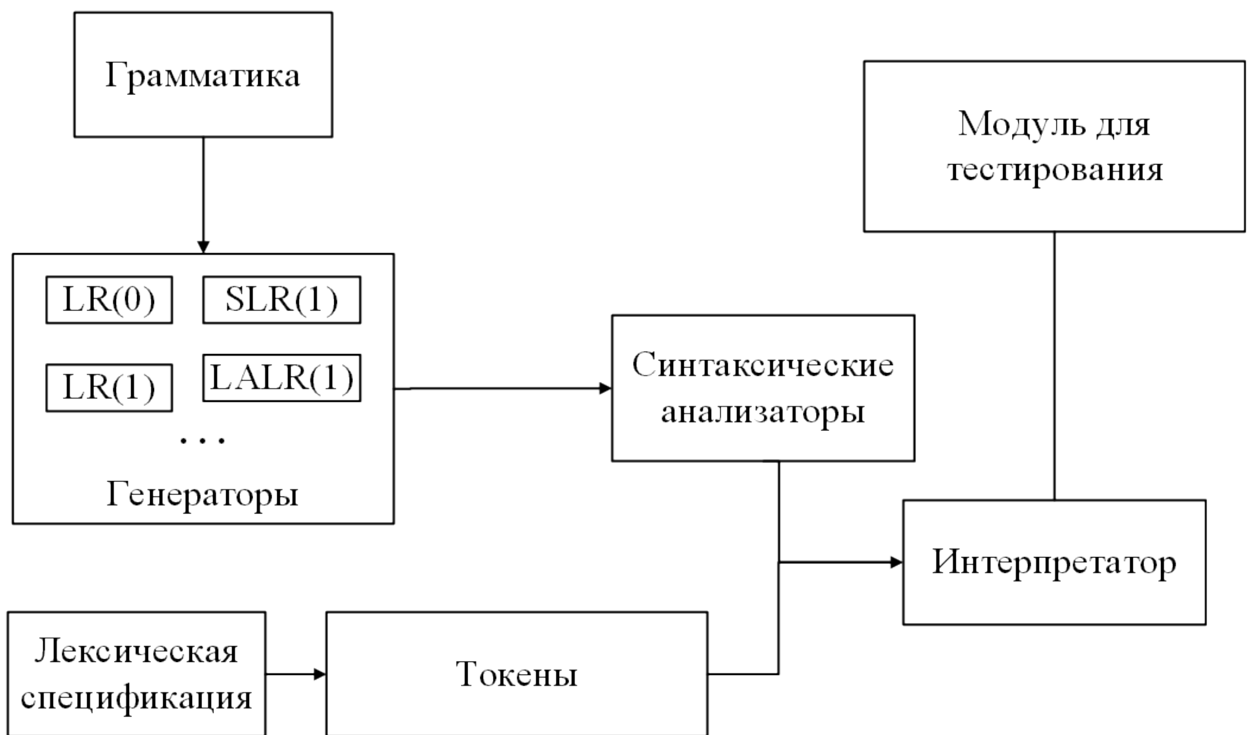

Рис. 6: Схема проведения эксперимента

После запуска интерпретатора на выходе в модуле для замеров получаем время работы синтаксического анализатора в зависимости от длины входной цепочки. По данным времени работы строится график зависимости времени работы от длины входных данных, и таким образом можно определить, какой из алгоритмов работает быстрее или медленнее.

4.2. Эксперимент 1

Проведём сравнение алгоритмов, используя грамматику *EasyCalc*, представленную на листинге 7.

```
[<Start>]
s: expr EOF { $1 }
expr : expr MULT numb { $1 * $3 }
     | expr PLUS numb { $1 + $3 }
     | numb { $1 }
numb : ZERO { 0 } | ONE { 1 }
```

Listing 7: Грамматика *EasyCalc*

Результаты сравнения

На рис. 7, 8, 9, 10 изображена зависимость среднего времени работы синтаксического анализатора по 10-ти запускам от длины входной цепочки для LR(0), LR(1), SLR(1), LALR(1) алгоритмов соответственно. Входная цепочка представляла собой последовательность из «0» и «1» со знаками «*» и «+» между ними. Длина входной цепочки изменяется от 2 000 до 10 000 000 токенов.

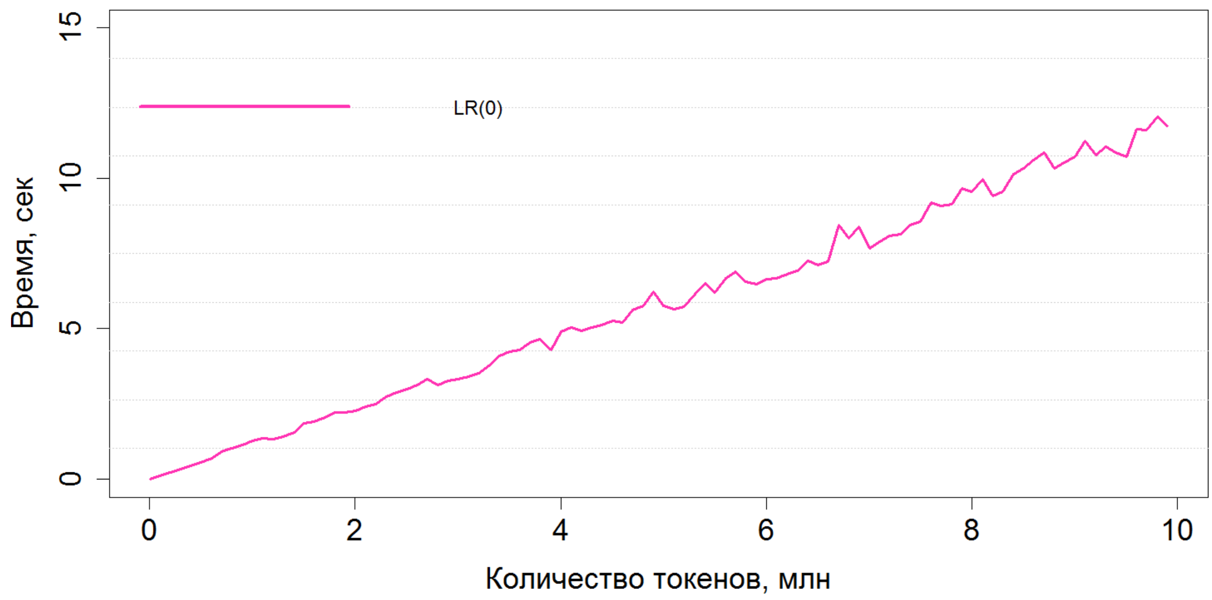

Рис. 7: Время работы LR(0)-анализатора для грамматики *EasyCalc*

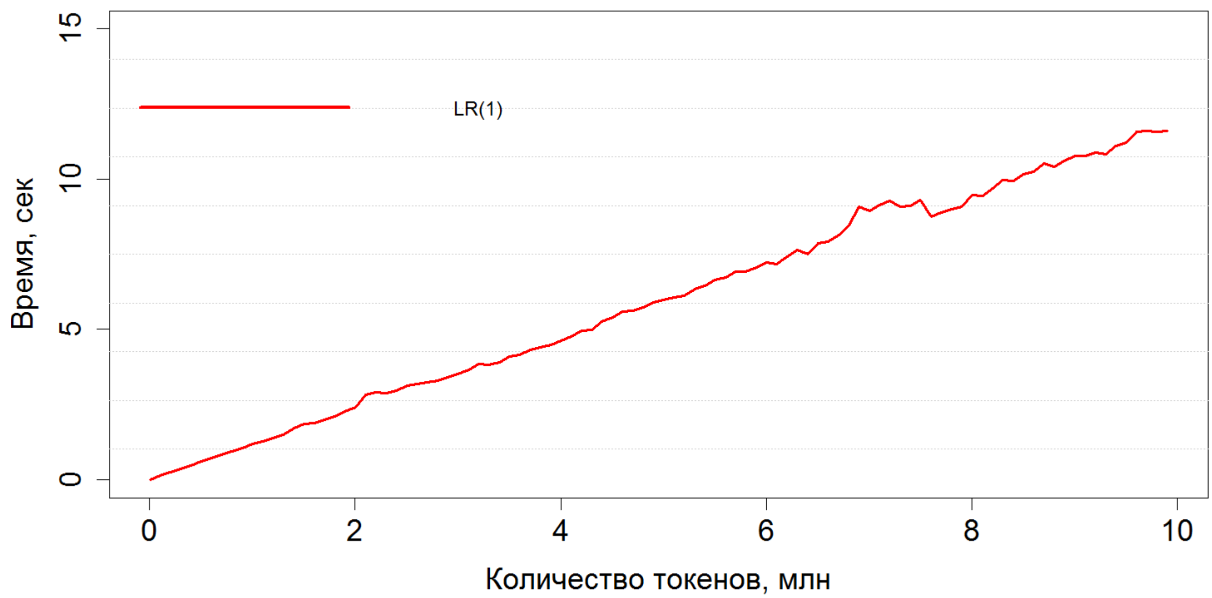

Рис. 8: Время работы LR(1)-анализатора для грамматики *EasyCalc*

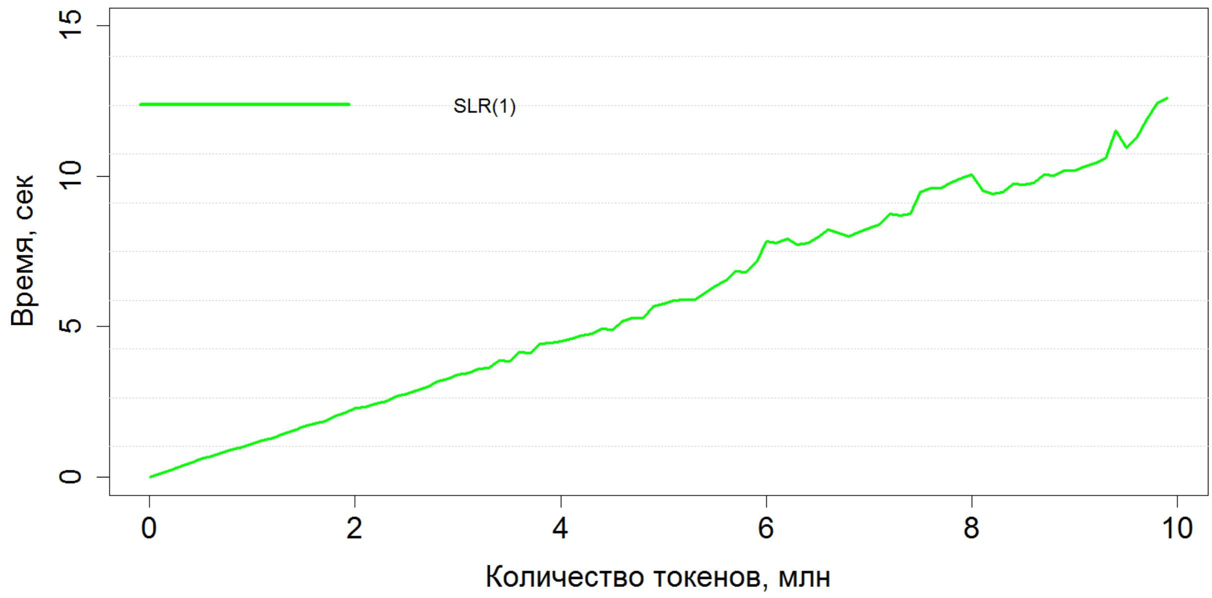

Рис. 9: Время работы SLR(1)-анализатора для грамматики *EasyCalc*

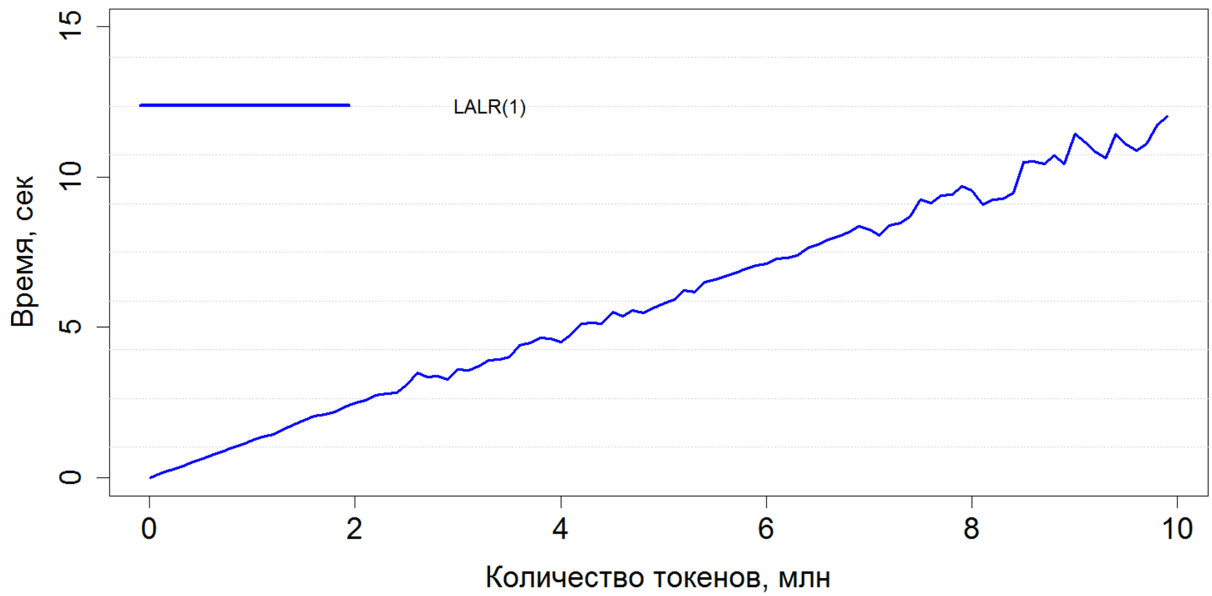

Рис. 10: Время работы LALR(1)-анализатора для грамматики *EasyCalc*

В табл. 1 представлены некоторые результаты измерения среднего времени работы анализатора и среднеквадратического отклонения для грамматики *EasyCalc*.

	Кол-во токенов	LR(0)	LR(1)	SLR(1)	LALR(1)
Среднее время, сек	2 000	0.002	0.002	0.002	0.002
	2 250 000	2.5	2.9	2.4	2.7
	4 500 000	5.2	5.4	4.9	5.4
	6 000 000	6.6	7.2	7.8	7.1
	10 000 000	11.7	11.6	12.5	12
Средне-квадратич. откл., сек	2 000	0.0002	0.0001	0.0001	0.0002
	2 250 000	0.023	0.087	0.304	0.154
	4 500 000	0.261	0.109	0.668	0.182
	6 000 000	0.055	0.167	0.145	0.718
	10 000 000	0.155	0.153	0.201	0.445

Таблица 1: Результаты замеров для грамматики *EasyCalc*

На рис. 11 представлены результаты замеров времени работы для грамматики *EasyCalc*.

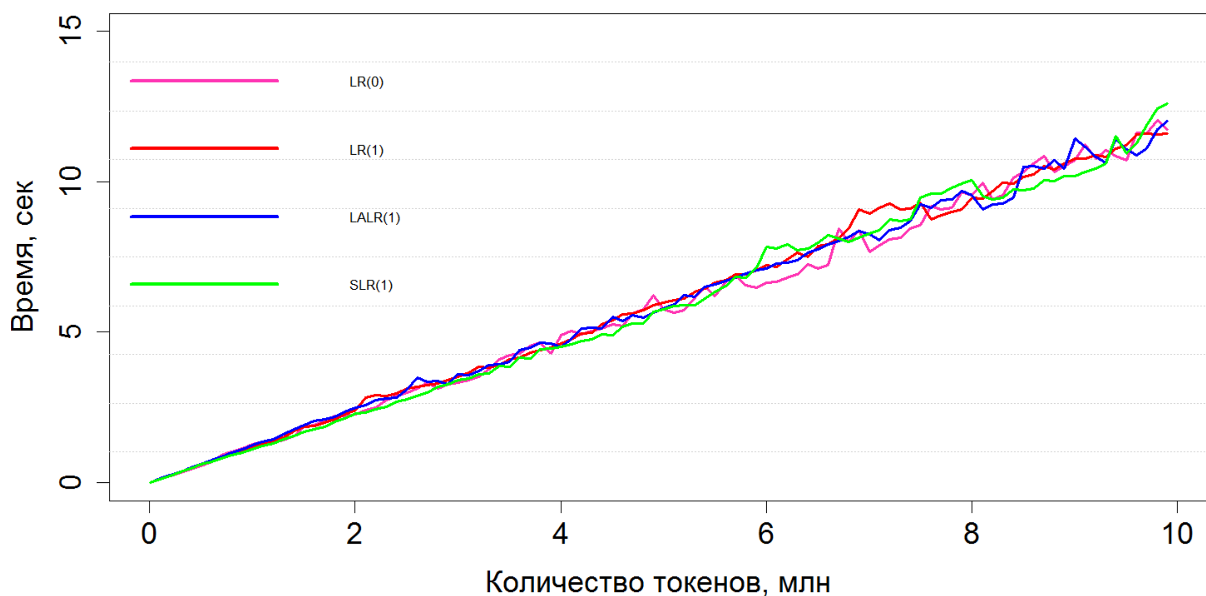

Рис. 11: Время работы анализаторов для грамматики *EasyCalc*

4.3. Эксперимент 2

Рассмотроим более сложную грамматику *Calc* калькулятора с выражениями (листинг 8).

```
[<Start>]
calc : statementList EOF { $1 }
statementList : statement SEP statementList { $1::$3 }
              | statement SEP { [$1] }
statement : expr { $1 }
          | ID EQ expr { $3 }
expr : multExpression PLUS multExpression { $1 + $3 }
     | multExpression MINUS multExpression { $1 - $3 }
     | multExpression { $1 }
atom : ID { $1 }
     | INT { $1 }
     | LBRACE expr RBRACE { $2 }
multExpression : atom MULT atom { $1 * $3 }
               | atom DIV atom { $1 / $3 }
               | atom { $1 }
```

Listing 8: Грамматика *Calc*

Данная грамматика кроме бинарных операций позволяет записать ещё и выражения. Например:

$$x=5;y=x*2;z=((17*x+36*y)+8);$$

Результаты сравнения

Поскольку данная грамматика не является LR(0), но является SLR(1)-грамматикой, проведено сравнение на алгоритмах LR(1), SLR(1) и LALR(1). На рис. 12, 13, 14 изображена зависимость среднего времени работы синтаксического анализатора по 10-ти запускам от длины входной цепочки. Длина входной цепочки изменяется от 4 000 000 до 80 000 000 токенов.

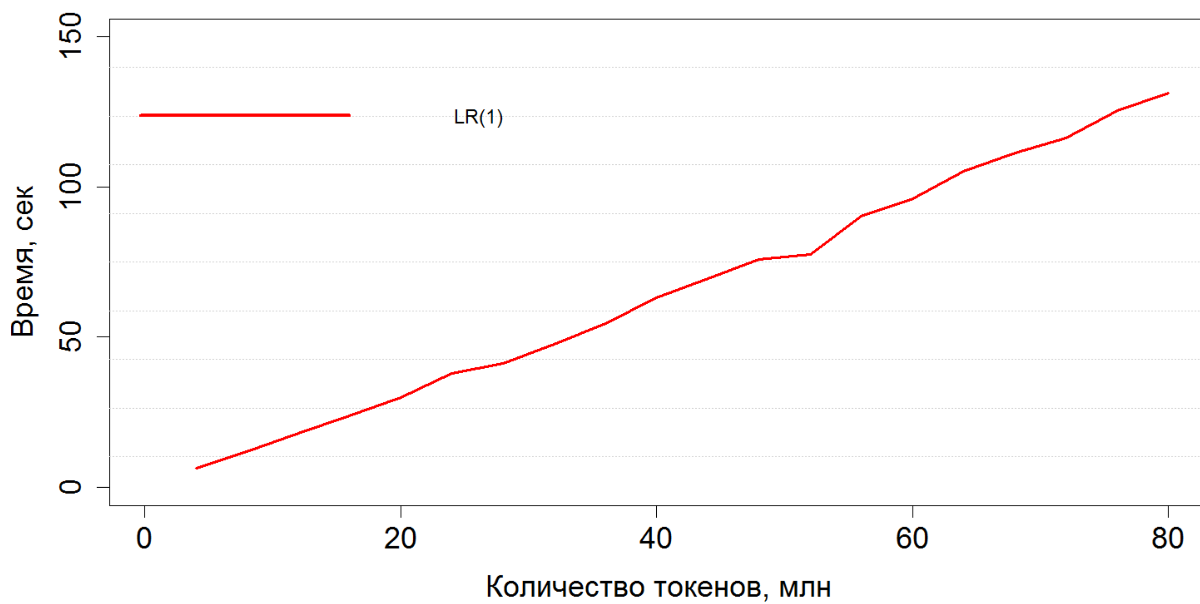

Рис. 12: Время работы LR(1)-анализатора для грамматики *Calc*

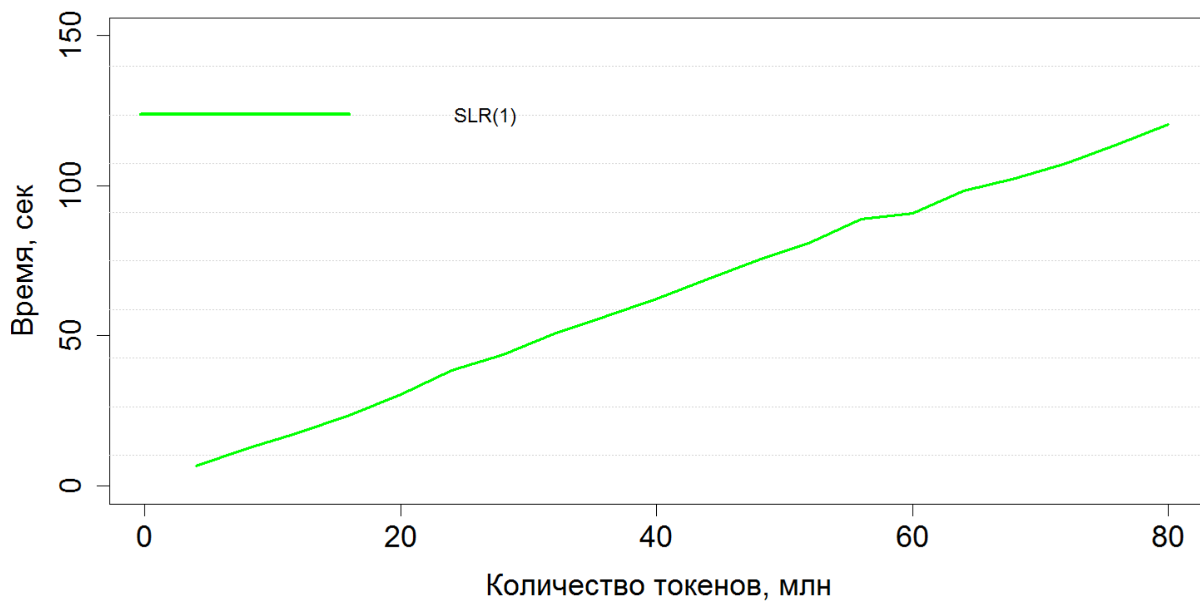

Рис. 13: Время работы SLR(1)-анализатора для грамматики *Calc*

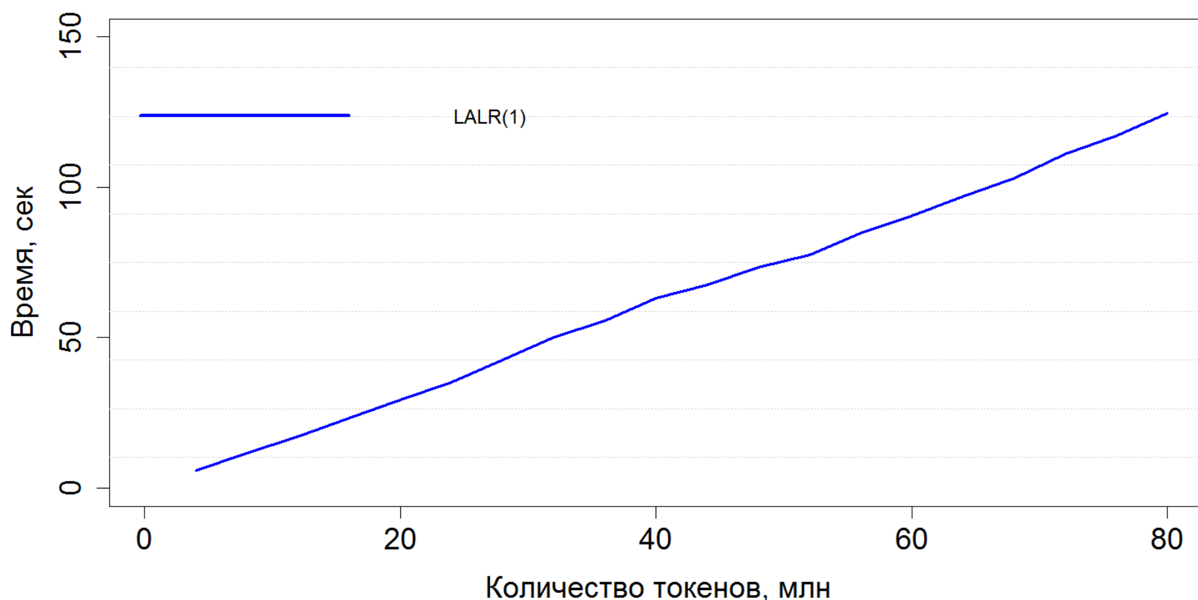

Рис. 14: Время работы LALR(1)-анализатора для грамматики *Calc*

В табл. 2 представлены некоторые результаты измерения среднего времени работы анализатора и среднеквадратического отклонения для грамматики *Calc*.

	Кол-во токенов	LR(1)	SLR(1)	LALR(1)
Среднее время, сек	4 000 000	6.2	6.1	5.8
	15 000 000	24.1	23.5	22.3
	30 000 000	49.9	43.8	42.5
	55 000 000	87.7	81.6	84.6
	80 000 000	131.1	120.3	124.4
Средне-квадратич. откл., сек	4 000 000	0.197	0.108	0.037
	15 000 000	0.201	0.13	0.076
	30 000 000	0.328	0.462	0.421
	55 000 000	0.463	0.901	0.363
	80 000 000	1.987	0.862	1.201

Таблица 2: Результаты замеров для грамматики *Calc*

На рис. 15 представлены результаты времени работы для выбранных алгоритмов.

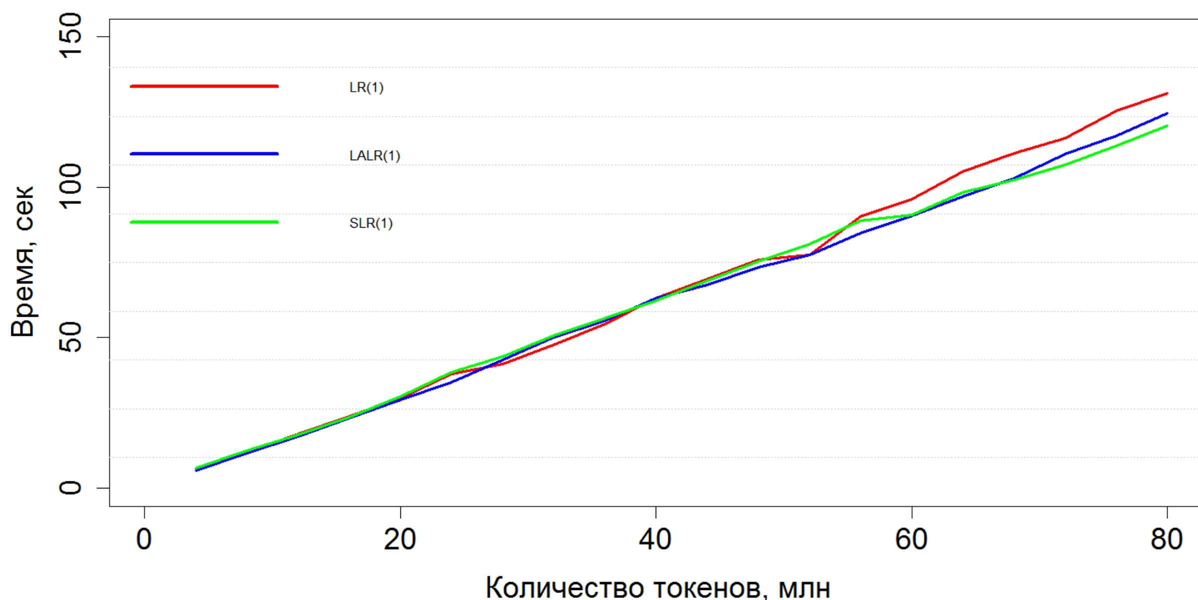

Рис. 15: Время работы анализаторов для грамматики *Calc*

4.4. Вывод

Полученная в результате интеграции платформа была протестирована на различных грамматиках и входах.

Проведено сравнение времени работы алгоритмов табличного восходящего синтаксического анализа на грамматиках *EasyCalc* и *Calc*. На больших входных данных, когда длина цепочки достигает порядка десятков миллионов токенов, начинает сказываться разница в размерах таблиц. Как известно, LR(1)-таблицы имеют больший размер.

В книге [10] говорится о том, что LALR-таблица компактнее, то есть она занимает меньше памяти. Но стремление уменьшить количество потребляемой памяти было актуально раньше, когда у компьютеров было мало памяти. Таким образом, современным компьютерам не важен размер таблиц.

Заключение

В ходе данной работы получены следующие результаты:

- реализована интеграция YaccConstructor с инструментом Facio (F#);
- проведено сравнение времени работы анализаторов, сгенерированных с использованием таких алгоритмов табличного восходящего синтаксического анализа, как LR(0), LR(1), LALR(1), SLR(1).

Исходный код можно посмотреть в репозитории проекта: https://github.com/YaccConstructor/YaccConstructor, автор — *emavchun*.

Дальнейшим развитием данной работы может быть разработка отдельного приложения, которое на вход получает лексическую и синтаксическую спецификации языка, желаемый алгоритм восходящего анализа для генерации синтаксического анализатора и входную цепочку. Результатом работы данной программы являются графики зависимости времени, затраченного на синтаксический анализ, от длины входной цепочки. Также должна быть возможность отображения графиков для всех алгоритмов в одном окне для наглядного сравнения. Для построения графиков можно использовать библиотеку F Charting [5].

Список литературы

[1] ANTLR parser generator. — URL: http://www.antlr.org/ (дата обращения: 15.03.2015).

[2] Aho Alfred V., Ullman. The Theory of Parsing, Translation and Compiling, Parsing of Series in Automatic Computation. — Prentice-Hall, 1972.

[3] Amin Milani Fard Arash Deldari Hossein Deldari. Quick Grammar Type Recognition: Concepts and Techniques. Department of Computer Engineering // International conf. on Compilers, Related Technologies and Applications (CoRTA). — 2007.

[4] Bison parser generator. — URL: http://www.gnu.org/software/bison/manual/ (дата обращения: 15.03.2015).

[5] F# Charting — библиотека для визуализации данных. — URL: http://fslab.org/FSharp.Charting/ (дата обращения: 3.05.2015).

[6] F# Language Specification. — URL: http://fsharp.org/specs/language-spec/ (дата обращения: 10.02.2015).

[7] Grune Dick, Jacobs J. H. Ceriel. Parsing Techniques: A Practical Guide. — Springer, 2008.

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

[9] .NET Framework. — URL: http://www.microsoft.com/net (дата обращения: 20.04.2015).

[10] The Theory of Parsing, Translation and Compiling, Parsing of Series in Automatic Computation / Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. — Pearson Education, 2006.

[11] Victor R. Basili Gianluigi Caldiera H. Dieter Rombach. The Goal Question Metric Approach // Chapter in Encyclopedia of Software Engineering, Wiley. — 1994.

[12] Yacc parser generator. — URL: `http://epaperpress.com/lexandyacc/` (дата обращения: 15.03.2015).

[13] А.Н. Терехов Н.Н. Вояковская Д.Ю. Булычев А.Е. Москаль. Разработка компиляторов на платформе .NET. — СПб.: Кафедра системного программирования СПбГУ, 2001.

[14] Домашняя страница Facio. — URL: `https://github.com/jack-pappas/facio` (дата обращения: 20.02.2015).

[15] Домашняя страница YaccConstructor. — URL: `https://code.google.com/p/recursive-ascent/wiki/YaccConstructor` (дата обращения: 15.03.2015).

[16] Кириленко Я.А. Григорьев С. В. Авдюхин Д. А. Разработка синтаксических анализаторов в проектах по автоматизированному реинжинирингу информационных систем // Научно-технические ведомости Санкт-Петербургского государственного политехнического университета информатика, телекоммуникации, управление. Номер: 174. — 2013.