УДК 519.682.1

# ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ И ОПТИМИЗАЦИЯ АЛГОРИТМА НИСХОДЯЩЕГО РАЗБОРА С ОГРАНИЧЕННЫМИ ВОЗВРАТАМИ ДЛЯ PEG-ГРАММАТИК

**Д. И. Соломатин**

*Воронежский государственный университет*

**Поступила в редакцию 30.07.2013 г.**

**Аннотация**. В статье рассматривается двухпроходная модификация алгоритма нисходящего разбора с ограниченными возвратами для PEG-грамматик с семантическими вставками. Получены экспериментальные данные эффективности алгоритма на примере языка Java. Предлагаются подходы сокращения потребляемой памяти, необходимой для хранения промежуточных результатов разбора.

**Ключевые слова:** нисходящий синтаксический разбор с ограниченными возвратами, PEG-грамматики, генератор синтаксических анализаторов, оптимизация.

**Annotation.** The article describes two-pass modification of top-down parsing algorithm with limited backtracking for PEG-grammars with semantics entries, experimental results of this algorithm for Java language and methods to reduce memory usage for storing intermediate parsing results.

**Keywords:** top-down syntax parsing with limited backtracking, PEG-grammars (Parsing Expression Grammars), parser generator, optimization.

## ВВЕДЕНИЕ

Отличительной чертой алгоритмов нисходящего синтаксического анализа является их целенаправленность, т.е. на каждом шаге анализа в таком алгоритме присутствует цель – распознать часть входной строки в соответствии с текущем испытываемым выражением из правой части какого-либо правила грамматики языка. Благодаря этому свойству алгоритмы нисходящего синтаксического анализа более естественны для человека, в отличие от алгоритмов восходящего или смешанного анализа.

Рассмотрим работу общего нисходящего алгоритма синтаксического разбора. Предположим, что нам необходимо распознать некоторую часть входной строки с применением правила (нетерминала) $A$, который состоит из нескольких альтернатив $a_1, a_2, ..., a_n$. В случае нисходящего разбора данные альтернативы испытываются по порядку, независимо от того, были ли успешными в данной позиции предыдущие альтернативы или нет. Такой алгоритм может затрачивать на разбор экспоненциальное время. Кроме того, данный алгоритм не гарантирует единственность разбора входной строки.

Избавиться от указанных недостатков позволяет простая модификация данного алгоритма, суть которой заключается в том, что в случае, если часть необработанной входной строки распознается альтернативой $a_i$, то остальные альтернативы $a_{i+k}$ правила A в данной позиции вообще не испытываются в независимости от того, будет ли общий разбор входной строки с выбранной альтернативой $a_i$ успешным или нет. Формализм описания грамматик, основанный на данной модификации общего алгоритма нисходящего разбора, получил названия языка нисходящего разбора с ограниченными возвратами (ЯНРОВ) [1].

Важной особенностью ЯНРОВ с точки зрения реализации синтаксического анализатора по ЯНРОВ-программе (или грамматике) является возможность представлять правила грамматики в виде функций, параметром при вызове которых является входная позиция указателя, с которой требуется распознавать часть входной строки в соответствии с данным правилом. Результатом работы любой такой функции будет ответ, распознается ли часть входной строки в данной позиции в соответствии с правилом или нет и, если распознается, то какой префикс входной строки был распознан, т.е. в какую позицию переместился указатель. При этом такие функции могут рекурсивно вызывать друг друга, но левая рекурсия, как прямая, так и косвенная, не допускается.

PEG-грамматики (Parsing Expression Grammars – грамматики, разбирающие выражения), предложенные Брайном Фордом (Bryan Ford) в 2002 г. в работе [3], базируются на концепциях ЯНРОВ, соответственно, алгоритмы, применимые для разбора по ЯНРОВ-программам и PEG-грамматикам эквивалентны.

Алгоритмы разбора для PEG-грамматик

Языки, описанные с помощью PEG-грамматик, могут быть распознаны за линейное время на автомате с произвольным доступом к памяти. Такой результат достигается за счет хранения всех промежуточных результатов разбора, при этом объем требуемой памяти в общем случае составляет m Ч (n + 1), где m – количество правил (нетерминалов) в грамматике, а n – длина входной строки. Рассмотрим суть данного алгоритма на примере PEG-грамматики для разбора обычных арифметических выражений:

```
D → '0'..'9'
P → D / "(" A ")"
M → P "*" M / P "/" M / P
A → M "+" A / M "-" A / M
S → A
```

Стоит заметить, что в приведенной грамматике нарушена ассоциативность операторов: операторы описаны как правоассоциативные, хотя в действительности они левоассоциативные. Такое допущение может приводить к построению некорректного (с точки зрения последовательности вычисления выражения) дерева разбора, но на возможность распознавания соответствия входной строки описанному языку никак не влияет. В корректной в этом смысле грамматике следовало бы воспользоваться операциями повторения шаблона, однако для иллюстрации рассматриваемого алгоритма разбора лишнее усложнение грамматики нежелательно.

Пусть на вход данной программе подается входная строка “5 + (2 - 1)*3”. При этом таблица для хранения промежуточных результатов разбора будет иметь следующий вид:

В момент начала разбора таблица промежуточных результатов пустая. В ходе разбора таблица постепенно заполняется результатами применения правил в соответствующих позициях входной строки. При этом алгоритм применения правила R схематично можно описать следующим образом:

```
boolean function ruleApply(R) {
if (parseResults[R, pos] != null)
if (parseResults[R, pos] == false)
return false;
else {
pos = parseResults[R, pos];
return true;
}
else {
int startPos = pos;
boolean result = R();
if (result)
parseResults[R, startPos] = pos;
else {
parseResults[R, startPos] = false;
pos = startPos;
}
return result;
}
}
```

Здесь parseResults – таблица для хранения промежуточных результатов разбора, pos – позиция указателя. При первом обращении к правилу в конкретной позиции в таблицу parseResults записывается новая позиция указателя в случае успешного вызова правила (R()) либо false в противном случае (в таблице обозначается F). Функция ruleApply гарантирует, что в каждой позиции любое правило будет вызвано не более одного раза, в случае повторного применения данного правила в данной позиции будет использован сохраненный промежуточный результат разбора. Это обеспечивает линейное относительно длины входной строки время работы данного алгоритма.

Таблица 1

*Таблица промежуточных результатов разбора (начало разбора)*

| позиция / правило | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 5 | + | ( | 2 | - | 1 | ) | * | 3 | $ |
| D | | | | | | | | | | |
| P | | | | | | | | | | |
| M | | | | | | | | | | |
| A | | | | | | | | | | |
| S | | | | | | | | | | |

Для рассматриваемого примера разбора математических выражений в конце разбора таблица промежуточных результатов будет заполнена следующим образом:

Таблица 2

*Таблица промежуточных результатов разбора (окончание разбора)*

| позиция / правило | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 5 | + | ( | 2 | - | 1 | ) | * | 3 | $ |
| D | 2 | | F | 5 | | 7 | | | 10 | |
| P | 2 | | 8 | 5 | | 7 | | | 10 | |
| M | 2 | | 10 | 5 | | 7 | | | 10 | |
| A | 10 | | 10 | 7 | | 7 | | | | |
| S | 10 | | | | | | | | | |

Предположим, что кроме собственно разбора нам необходимо выполнять по ходу разбора некоторые семантические действия, т.е. действия, которые вызываются в случае успешного применения правила или его части в случае, если общий вывод с применением данного правила был успешным. Учитывая, что даже в случае успеха текущего правила в вызываемой позиции общий вывод может оказаться неудачным (при этом удачным может оказаться вывод с применением другой альтернативы в каком-либо правиле), выполнять семантические действия непосредственно по ходу разбора в общем случае нельзя. Таким образом, семантические действия могут быть выполнены только при успешном разборе всей входной строки целиком. При этом должны быть выполнены только те семантические действия, которые соответствуют успешному дереву разбора входной строки.

Для преодоления данного ограничения Брайном Фордом (Bryan Ford) в работе [3] была предложена «ленивая» модификация данного алгоритма, так называемый алгоритм «старьевщика» (packrat parsing algorithm). Суть алгоритма «старьевщика» заключалась в сохранении в качестве промежуточных результатов разбора не только новой позиции указателя после применения правила, но и «ленивого» значения, при вычислении которого должны быть выполнены все семантические действия, связанные с конкретным применением данного правила. Под «ленивым» значением в рамках концепции «ленивых» или отложенных вычислений (lazy evaluation) понимается значение, которое еще не вычислено, но может быть вычислено в любой момент, когда понадобится его результат. Таким образом, в результате успешного разбора входной строки алгоритм «старьевщика» получает результирующее «ленивое» значение, в результате вычисления которого будут в нужном порядке выполнены все необходимые семантические действия, которые встречались по ходу успешного разбора.

Предположим, что в рассматриваемом примере разбора математических выражений семантические действия по ходу разбора должны вычислять значение выражения, записанного во входной строке. В этом случае таблица хранения промежуточных «ленивых» значений после успешного разбора входной строки будет содержать следующие значения (через {action} обозначены ленивые значения, результат которых должен быть получен в результате выполнения action):

Таблица 3

*Таблица промежуточных «ленивых» значений в алгоритме «старьевщика» (окончание разбора)*

| позиция / правило | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 5 | + | ( | 2 | - | 1 | ) | * | 3 | $ |
| D | {2} | | F | {2} | | {1} | | | {3} | |
| P | {D[1]} | | {A[4]} | {D[4]} | | {D[6]} | | | {D[9]} | |
| M | {P[1]} | | {P[3]*M[9]} | {P[4]} | | {P[6]} | | | {P[9]} | |
| A | {M[1]+A[3]} | | {M[3]} | {M[4]+A[6]} | | {M[6]} | | | | |
| S | {A[1]} | | | | | | | | | |

Заметим, что в таблице промежуточных результатов в качестве «ленивых» значений могут быть сохранены любые вычисления, встречающиеся по ходу разбора, а не только те, которые возвращают какой-либо результат.

## ДВУХПРОХОДНЫЙ АЛГОРИТМ РАЗБОРА ДЛЯ PEG-ГРАММАТИК С СЕМАНТИЧЕСКИМИ ВСТАВКАМИ

Для языка, поддерживающего ленивые вычисления (например, Haskell), PEG-грамматика с семантическими действиями (которые задаются в виде инструкций целевого языка) легко может быть оттранслирована в программный код, реализующий описанный алгоритм «старьевщика».

Рассмотрим, как подобный алгоритм может быть реализован для императивного языка на примере языка Java. Для классических объектно-ориентированных языков ленивые вычисления можно эмулировать, однако учитывая, что в коде семантических действий могут использоваться лексические переменные, объявленные ранее, возникает необходимость в поддержке в целевом языке механизма замыканий. Механизм замыканий (англ. closure) в программировании позволяет функциям, ссылаясь на свободные переменные в своём лексическом контексте (т.е. месте, где функция объявлена), тем самым «захватывать» их. Таким образом, свободные переменные связываются с экземпляром функции (делегатом в C#, анонимным классом в Java) и доступны внутри функции при последующем вызове. Однако в языке Java могут быть связаны только переменные, объявленные с модификатором `final`, т.е. неизменяемые. Это ограничение делает невозможным реализацию «ленивого» алгоритма «старьевщика» для языка Java в условиях первоначальной постановки задачи (лексические переменные объявляются в коде любого семантического действия и доступны, в том числе для изменения, в коде семантических действий, описанных ниже с учетом порядка вложенности).

Эквивалентное «ленивому» алгоритму «старьевщика» поведение синтаксического анализатора может быть получено в алгоритме двухпроходного синтаксического анализа. На первом шаге анализа (первый проход) проводится только разбор входной строки с помощью рассмотренного алгоритма разбора для PEG-грамматик, в ходе которого в таблицу промежуточных результатов для каждой обработанной альтернативы в правилах грамматики дополнительно заносится номер выбранной альтернативы (а для операций повторения шаблона – число повторений). Обладая данной информацией, на втором шаге анализа производится повторный однозначный (без подбора альтернатив и числа повторений для соответствующих узлов правил PEG-грамматики) проход по дереву примененных правил с вызовом сопутствующих семантических действий. Ключевым преимуществом данного алгоритма для императивных языков программирования является возможность трансляции исходных PEG-грамматик в код синтаксических анализаторов без обозначенных выше «лексических» сложностей для «ленивого» алгоритма «старьевщика».

Для пояснения сути алгоритма ниже приводится содержимое таблицы промежуточных результатов после первого прохода двухпроходного алгоритма для рассматриваемого выше примера арифметических выражений:

Таблица 4

*Таблица промежуточных результатов в двухпроходном алгоритме разбора (окончание разбора)*

| позиция / правило | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| | 5 | + | ( | 2 | - | 1 | ) | * | 3 | $ |
| D | 2 | | F | 5 | | 7 | | | 10 | |
| № альт. / в P | 1 | | 2 | 1 | | 1 | | | 1 | |
| P | 2 | | 8 | 5 | | 7 | | | 10 | |
| № альт. / в M | 3 | | 1 | 3 | | 3 | | | 3 | |
| M | 2 | | 10 | 5 | | 7 | | | 10 | |
| № альт. / в A | 1 | | 3 | 2 | | 3 | | | | |
| A | 10 | | 10 | 7 | | 7 | | | | |
| S | 10 | | | | | | | | | |

Здесь, например, числа 3 и 2 в строке «№ альт. / в A» в столбце 3 и 4 означают, что при выборе альтернативы правила A в позиции '(' была выбрана альтернатива M, а в позиции '2' альтернатива M "-" A.

Предложенный двухпроходный алгоритм реализован в системе построения синтаксических анализаторов PEG-PG (PEG Parser Generator) для языка Java [2].

## ЭКСПЕРИМЕНТАЛЬНОЕ ИССЛЕДОВАНИЕ АЛГОРИТМА

Для изучения поведения алгоритма нисходящего разбора с ограниченными возвратами был выполнен ряд экспериментов. Очевидно, что любые полученные таким образом результаты будут зависеть от исходного языка, для которого проводились эксперименты, от конкретного описания данного языка в виде PEG-грамматики и от набора входных строк. Поэтому для исследования целесообразно провести такой подбор исходных параметров, чтобы результаты исследования можно было объективно сравнивать с другими алгоритмами разбора. В данном случае входным языком для тестирования был выбран язык Java, PEG-грамматика данного языка была разработана на основе грамматики, приводимой в спецификации языка Java [4].

Полученная PEG-грамматика состоит из 119-ти основных правил (нетерминалов), а также 2-х вспомогательных правил – start и skip. (Очевидно, что полностью грамматика в данной статье приведена быть не может.) Правило start – начальное правило, с вызова этого правила начинается разбор входной строки (содержимого *.java-файла). Про правило skip следует рассказать более подробно.

Несмотря на то, что при задании языка с помощью PEG-грамматик фазы лексического и синтаксического анализа явно не выделены, при разборе важно отличать правила грамматики, распознающие лексемы (термы) или их части. Будем такие правила условно называть *лексическими*, а остальные – *синтаксическими*. Лексемы (ключевые слова языка, идентификаторы, пунктуационные знаки и т.д.) являются неразрывными цепочками символов. Лексемы (единицы языка с точки зрения классического синтаксического анализа), наоборот, друг от друга, как правило, могут отделяться произвольным количеством пробельных символов. Правило skip отвечает за распознавание и пропуск пробельных элементов между лексемами и автоматически вызывается при применении синтаксических правил после распознавания любого терминального символа или строки, а также вызова лексического правила. Внутри лексических правил skip никогда не вызывается. В рассматриваемой PEG-грамматике для языка Java правило skip описано следующим образом:

```
~skip: ( WhiteSpace+ / Comment )* ;
```

Т.е. в качестве пробельных элементов считаются подряд идущие пробельные символы (пробелы, табуляции, переносы строк) и комментарии (+ означает один или больше повторений, * – ноль или больше повторений; ~ указывает на то, что skip является лексическим правилом). WhiteSpace и Comment – лексические правила, результаты применения которых, кроме того, не сохраняются в таблице промежуточных результатов разбора (для этого данные правила особым образом помечены). Такие правила будем называть *несохраняемыми*.

Очевидно, что в большинстве случаев относительно других правил skip при разборе будет применяться наибольшее количество раз.

В качестве выборки для проведения эксперимента использовался набор исходных текстов ряда свободно распространяемых Java-проектов в количестве 4595 файлов (в предположении, что большое количество различных файлов приведет к получению объективных (среднестатистических) результатов для языка Java).

В ходе эксперимента ставилась задача, прежде всего, получить зависимость количества применений правил грамматики от длины входной строки, а также исследовать возможности сокращения необходимой памяти для хранения промежуточных результатов разбора. Скорость разбора интересовала в меньшей степени, так как сильно зависит от конкретных деталей реализации алгоритма разбора, средств реализации (в данном случае также использовался язык программирования Java) и т.д.

На рис. 1 приводится диаграмма, на которой точками отмечены результаты эксперимента для всех файлов выборки. Под «значимым» размером понимается размер исходного текста java-модуля, из которого исключены все комментарии и пробельные символы (за исключением пробелов в строках-литералах). Под «значимыми» правилами понимаются все правила, кото-

рые не участвуют в разборе пробельных элементов (skip, WhiteSpace, Comment и некоторые другие). Использование «значимых» значений объясняется тем, что в *.java-файлах размер пробелов и особенно комментариев может очень сильно отличаться от файла к файлу, что ощутимо влияет на результат (отношение одной величины к другой), хотя с точки зрения структурной сложности файлы при этом могут быть одинаковыми.

Как видно из результатов эксперимента, теоретические данные о линейной зависимости времени разбора (если принять время вызова правил константным) от длины входной строки подтверждаются.

Здесь дополнительно стоит уточнить, что в тестируемой PEG-грамматике для языка Java присутствуют лексические правила, которые распознают только один входной символ (JavaLetter, Digit, JavaLetterOrDigit), т.е. на распознавание одного значимого терминального символа во входной строке чаще всего приходится один или больше вызовов правил разбора. Например, правило Identifier описано следующим образом:

```
~Identifier:  !Keyword
!BooleanLiteral !NullLiteral
   JavaLetter JavaLetterOrDigit* ;
```

Т.е. распознавание одного идентификатора повлечет за собой вызов более десятка правил (Keyword, BooleanLiteral, NullLiteral, а также правила, вызываемые из Keyword) только для того, чтобы отличить возможный идентификатор от зарезервированных слов языка Java (здесь символ ! означает отрицание). Далее на каждый символ идентификатора требуется вызвать правило JavaLetter или JavaLetterOrDigit, которое в свою очередь также описано через другие лексические правила. Если вместо выражения JavaLetter JavaLetterOrDigit* выполнить inline-подстановку описаний данных правил ([a-z] / [A-Z] / [_,$]) ([a-z] / [A-Z] / [_,$] / [0-9])* , то количество обращений к правилам разбора на один символ заметно сократится. Однако, как было сказано выше, тестируемая PEG-грамматика должна быть максимально похожа на грамматику, приведенную в спецификации языка Java, поэтому такая замена не была сделана.

На рис. 2 приводится гистограмма отношения кол-ва применений «значимых» правил к «значимому» размеру входного файла с шагом 0,5.

Неожиданный всплеск в гистограмме значений от 2,5 до 3 объясняется наличием в исходных кодах Java-программ большого кол-ва (порядка 20 % от всех файлов) описаний типов без реализации методов (прежде всего интерфейсы, а также абстрактные классы и классы с native-методами). Очевидно, что структура таких файлов более простая, чем файлов, содержащих не только описание, но и реализацию (непосредственно исполняемый код). Соответственно, для таких файлов в большинстве случаев искомое отношение будет меньше среднестатистического.

Если указанные файлы исключить из выборки, то рассматриваемая гистограмма примет вид, показанный на рис. 3.

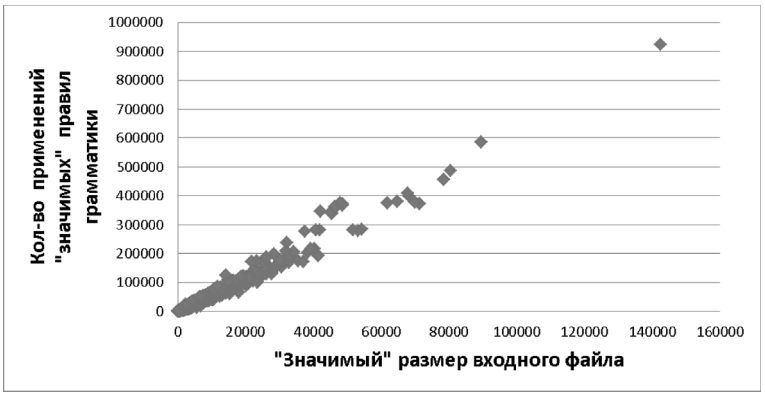


*Рис. 1.* Зависимость количества применений при разборе правил грамматики от размера входной строки

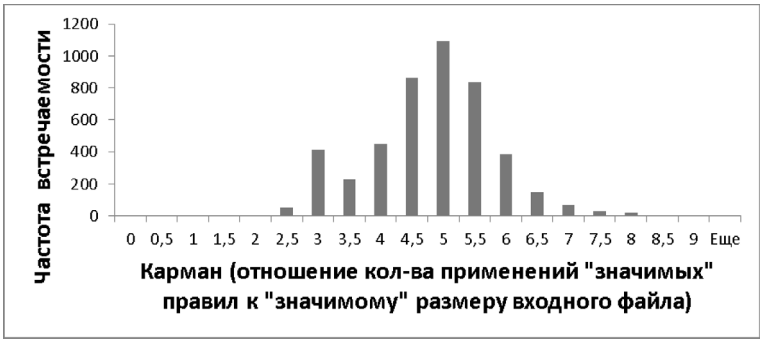


*Рис. 2.* Распределение отношения количества применений правил грамматики к размеру входной строки

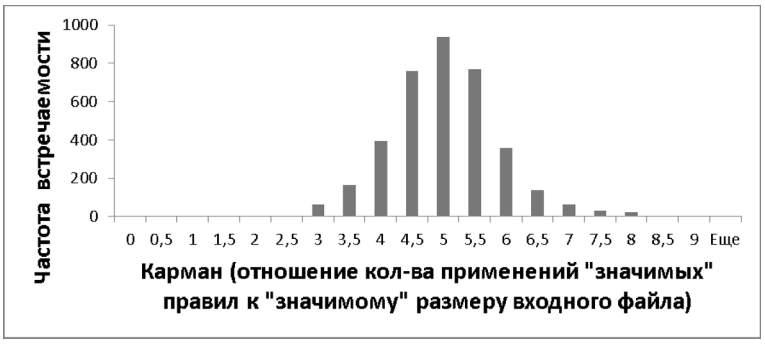


*Рис. 3.* Распределение отношения количества применений правил грамматики к размеру входной строки (отброшены описания интерфейсов и абстрактных классов)

Как видно из рис. 3 при отбрасывании из выборки описаний интерфейсов и т.п. распределение изучаемого отношения очень похоже на нормальное распределение (однако по критерию Хи-квадрат распределение нормальным не является). Среднее значение отношения – 4,77 (т.е. на один «значимый» символ входного файла приходится почти 5 обращений к «значимым» правилам грамматики языка Java). Стандартное отклонение – 0,86. Наименьшее значение – 2,43, наибольшее значение – 9,12.

Как видно из гистограммы, существуют Java-классы, для исходного кода которых значение рассматриваемого отношения заметно больше среднего значения (правый «хвост» гистограммы убывает более плавно). Детальное изучение таких классов показало, что большую часть из них составляют автоматически сгенерированные классы (в тестовой выборке использовался код ANTLR), тестовые классы для проверки разбора синтаксиса языка Java, а также классы, в которых инициализируются константами большие массивы данных. Про все эти классы можно сказать, что их структура заметно отличается от типичных классов, описанных на языке Java.

Также было экспериментально получено отношение количества повторных применений правил в одной и той же позиции к общему количеству применений правил. Оказалось, что при хранении промежуточных результатов разбора правила применяются повторно примерно в 4,5 % случаев (для только синтаксических правил этот процент еще ниже – примерно 1,5 %). Применение правил заканчивается успехом примерно в 35,5 % случаев, неудачей – в 64,5 % (для только синтаксических правил эти цифры составляют 26 % и 74 % соответственно). В процессе разбора синтаксические правила вызываются в 34 % случаев.

Такой низкий процент повторного использования промежуточных результатов разбора на первый взгляд показался довольно странным, поэтому точно такой же эксперимент был проведен для реализации алгоритма разбора с ограниченными возвратами без запоминания

промежуточных результатов. Характер полученных данных в ходе этого эксперимента не изменился, т.е. также наблюдается линейная зависимость количества вызовов правил от длины входной строки, однако среднее кол-во вызовов «значимых» правил к кол-ву «значимых» символов входной строки изменилось и стало равно 5,81. Это соотношение также дает распределение, близкое к нормальному, со стандартным отклонением в 1,07.

Полученные в ходе второго эксперимента данные говорят о том, что на практике при отказе от запоминания промежуточных результатов разбора кол-во выполняемых операций, следовательно, и время разбора экспоненциально не растут (во всяком случае, для языка Java), а наблюдается лишь незначительное увеличение, при этом линейный характер зависимости от размера входной строки сохраняется. Это косвенно говорит о том, что в программах, написанных человеком, большая вложенность синтаксических конструкций практически не встречается.

## ОПТИМИЗАЦИЯ АЛГОРИТМА

Несмотря на то, что на практике выигрыш от запоминания промежуточных результатов разбора крайне незначительный, для реализации 2-х проходного алгоритма разбора по грамматике с семантическими вставками хранение промежуточных результатов разбора все же необходимо. При этом в самом общем случае требуется памяти $(m + m') \times (n + 1)$, где $n$ – длина входной строки, $m$ – кол-во правил в грамматике, а $m'$ – кол-во в правилах грамматики выражений выбора альтернативы или повторений шаблона. Для языка Java (121 правило в грамматике и около 200 выражений выбора альтернативы или повторения шаблона) таблица промежуточных результатов потребует значительного объема оперативной памяти. Ниже рассматриваются приемы, которые могут этот объем памяти уменьшить.

**Прием 1.** Как нетрудно заметить в таблицах 2-4 присутствуют пустые столбцы, в ячейках которых не хранится никаких данных. Поэтому вместо первоначального выделения целиком всей необходимой памяти под таблицу разбора, память можно выделять отдельно по столбцам по мере необходимости во время разбора.

Более того, память под столбец может выделяться опять же не целиком, а по частям по $k$ ячеек (очевидно, что оптимальное значение $k$ зависит от разбираемого языка и его описания в виде грамматики).

**Прием 2.** Для сокращения потребляемой памяти следует стремиться к тому, чтобы как можно больше столбцов в таблице промежуточных результатов оставались пустыми. Как выше было отмечено, при разборе языка Java с помощью конкретной грамматики практически на каждый символ входной строки в позиции символа вызывается одно или несколько правил разбора, при этом полностью пустых столбцов, очевидно, не будет. В качестве борьбы с данной ситуацией можно предложить для некоторых правил отказаться от сохранения промежуточных результатов разбора. Самым простым решением в данном случае является возможность помечать в грамматике, какие это правила (что и было сделано). Обычно таким атрибутом следует помечать простые (в плане вычислений) лексические правила, а также правила, на которые есть единичные ссылки из других правил. Косвенным результатом описанного подхода также является сокращение числа $m$.

Также может быть предложен автоматический алгоритм определения несохраняемых правил согласно какому-либо обоснованному критерию.

**Прием 3.** В двухпроходном алгоритме разбора для грамматик с семантическими вставками для уменьшения потребляемой памяти можно попытаться сократить число $m'$. Сохранять номер правильной альтернативы или нужное количество повторений для соответствующих узлов правил грамматики необходимо только лишь для тех узлов, в ходе применения которых к входной строке могут вызываться семантические действия. Для каждого узла в правилах грамматики можно на этапе анализа грамматики по графу, построенному по описанию всех правил, найти те узлы, применение которых к входной строке не может приводить к выполнению семантических действий. Как правило, семантические действия не встречаются в лексических правилах, поэтому хранить результаты применения соответствующих узлов в таких правилах не требуется. Кроме сокращения числа $m'$ такой подход способствует увеличению числа пустых столбцов в таблице промежуточных результатов разбора.

В описанных выше экспериментах PEG-грамматика для языка Java вообще не содержа-

ла семантических действий, поэтому описанный подход позволил сократить потребляемую при разборе память в несколько раз. (Для осмысленного практического применения в грамматике естественно должны быть добавлены семантические действия, отвечающие за построение абстрактного синтаксического дерева разбираемой программы.)

С использование описанных выше приемов для синтаксического анализатора языка Java, полученного из рассматриваемой выше PEG-грамматики, на компьютере с процессором Intel Core i5-3570K 3,4 ГГц получены следующие показатели производительности и расхода памяти. Скорость разбора составила более 200 Кб исходного Java-кода в секунду, расход памяти – около 150 байт на один «значимый» символ входной строки. Естественно, результаты в дальнейшем могут быть улучшены за счет более детальной оптимизации алгоритма и его реализации.

## ЗАКЛЮЧЕНИЕ

Описанный двухпроходный алгоритм разбора для PEG-грамматик с семантическими вставками позволяет достаточно легко транслировать такие грамматики в код императивных языков программирования. Полученные экспериментальные результаты производительности и расхода памяти подтверждают возможность практического использования PEG-грамматик.

СПИСОК ЛИТЕРАТУРЫ

1. *Ахо А., Ульман Дж.* Теория синтаксического анализа, перевода и компиляции. – М.:Мир, 1978, т. 1.

2. *Львович Я.Е.* Методы и алгоритмы при построении генератора лексико-синтаксических анализаторов для PEG-грамматик / Я.Е. Львович, Д.И. Соломатин // Вестник Воронежского государственного технического университета. – Воронеж, 2008. – № 4 – С. 13–17.

3. *B. Ford.* Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking – Master's Thesis Massachusetts Institute of Technology. http://pdos.csail.mit.edu/~baford/packrat/thesis/thesis.pdf

4. The Java Language Specification, Third Edition. http://docs.oracle.com/javase/specs/

**Соломатин Дмитрий Иванович** – старший преподаватель кафедры программирования и информационных технологий, факультет компьютерных наук, Воронежский государственный университет

**Solomatin Dmitry Ivanovich** – senior lecturer of Department of Programming and Information Technologies, Faculty of Computer Sciences, Voronezh State University