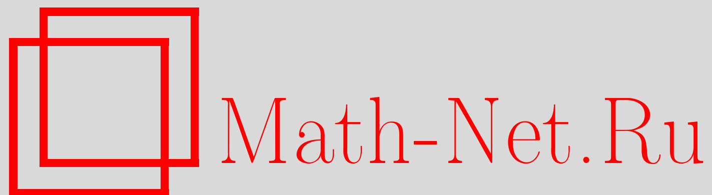

Общероссийский математический портал

С. В. Григорьев, А. К. Рагозина, Обобщенный табличный LL-анализ, *Системы и средства информ.*, 2015, том 25, выпуск 1, 89–107

DOI: 10.14357/08696527150106

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

Параметры загрузки:
IP: 18.191.212.146
16 октября 2024 г., 12:19:00

ОБОБЩЕННЫЙ ТАБЛИЧНЫЙ LL-АНАЛИЗ*

***С. В. Григорьев*[1], *А. К. Рагозина*[2]**

Аннотация: Синтаксический анализ является важным шагом анализа кода. Для работы с неоднозначными грамматиками используются обобщенные алгоритмы синтаксического анализа: Generalized LR (GLR) и Generalized LL (GLL). Для работы со встроенными языками — поддержки их в IDE (Integrated Development Environment), анализа в целях реинжиниринга или поиска уязвимостей (SQL-инъекций) — используется абстрактный синтаксический анализ, основанный на классическом табличном анализе. Ранее был предложен алгоритм обобщенного нисходяшего анализа без использования предиктивных таблиц анализа. В данной статье описан подход к созданию табличного GLL-анализатора на основе предложенного алгоритма, который в дальнейшем будет использоваться для получения абстрактного анализатора. Также в статье описан алгоритм обобщенного нисходящего анализа, модификации, которым он подвергся, и результаты сравнения с алгоритмом обобщенного восходящего анализа, который был реализован ранее.

Ключевые слова: синтаксический анализ; GLL; обобщенный анализ; RNGLR; абстрактный анализ; встроенные языки

DOI: 10.14357/08696527150106

1 Введение

Существует множество средств автоматической и автоматизированной обработки исходного кода программ. В качестве примера можно привести компиляторы, верификаторы, средства анализа исходного кода. Многие из них предполагают перевод кода из текстового представления в специальное структурное представление — абстрактное синтаксическое дерево, предназначенное для дальнейшей обработки. Для такого преобразования исходного кода в абстрактное синтаксическое дерево обычно используется синтаксический анализ, для реализации которого существует большое количество алгоритмов, различающихся классом обрабатываемых языков. Для работы с неоднозначными и недетерминированными грамматиками существуют алгоритмы обобщенного синтаксического анализа: GLR и GLL. Данные алгоритмы анализируют все возможные варианты разбора и строят все деревья. Алгоритм обобщенного нисходящего анализа (GLL) отличается своей простотой и прямолинейностью из-за тесной связи с грамма-

*Статья рекомендована к публикации в журнале Программным комитетом конференции «Tools & Methods of Program Analysis» («Инструменты и методы анализа программ», TMPA-2014), г. Кострома, 14–15 ноября 2014 г.

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

[2]Санкт-Петербургский государственный университет, ragozina.anastasiya@gmail.com

тикой, в то время как алгоритм обобщенного восходящего анализа (GLR) имеет более сложную структуру.

Многие языки программирования позволяют конструировать выражения на других языках, называемых встроенными, из строковых литералов и затем передавать их на выполнение соответствующему окружению. Чаще всего такой подход используется для генерации HTML-страниц и SQL-запросов. Трудность при использовании встроенных выражений заключается в том, что ошибки в них могут быть обнаружены только во время выполнения. Это делает систему, созданную с использованием такого подхода, ненадежной и уязвимой. Еще одна проблема, которая появляется при использовании встроенных языков — SQL-инъекции, в результате которых могут быть выполнены действия, не предусмотренные создателем скрипта. Эта ситуация возникает из-за отсутствия фильтрации внешних входных данных, поступающих на сервер. Логика запроса меняется таким образом, что злоумышленник может получить данные из таблиц или даже изменить что-то в базе данных. Для обработки встроенных языков используется статический анализ динамически формируемых выражений (абстрактный синтаксический анализ) [1].

Целью данной работы является получение табличного синтаксического анализатора на основе алгоритма GLL, который станет основой для абстрактного анализатора в дальнейшем. Существует несколько подходов к созданию абстрактных синтаксических анализаторов. Так как в абстрактном анализе процесс разбора сильно зависит от структуры входных данных, которая не является линейным входным потоком, то использование явной генерации кода анализатора затруднено. В статье [1] описан абстрактный анализ на основе классического алгоритма синтаксического анализа LALR. Ранее в рамках проекта YaccConstructor [2] был реализован алгоритм обобщенного восходящего анализа RNGLR (Right Nulled GLR) [3], а позже на его основе был создан абстрактный табличный анализатор [4]. Управляющие таблицы синтаксического анализатора при переходе к абстрактному анализу не меняются: достаточно модифицировать только интерпретатор таблиц. Таким образом, чтобы получить возможность использовать GLL-алгоритм для абстрактного анализа, необходимо изменить процесс построения анализатора по грамматике, так как в оригинальном алгоритме GLL не используются таблицы синтаксического анализа, а код анализатора генерируется явно. В данной работе описан алгоритм обобщенного нисходящего анализа и модификации, которые были внесены в него для получения табличного анализатора. Также обсуждаются особенности реализации табличного GLL-анализатора.

2 Обобщенный синтаксический анализ

Синтаксические анализаторы можно разделить на два класса — нисходящие и восходящие, каждый из которых имеет как преимущества, так и недостатки. Структура нисходящих парсеров полностью соответствует структуре грамматики,

что упрощает процесс их написания и отладки. Существенным недостатком таких синтаксических анализаторов является то, что класс языков, обрабатываемых ими, весьма ограничен. Причина этих ограничений заключается в том, что любая LL(k)-грамматика должна быть однозначной, но далеко не все языки программирования однозначны. Например, леворекурсивные грамматики не принадлежат классу LL(k) ни для какого k. Иногда удается преобразовать неLL-грамматику в эквивалентную ей LL-грамматику с помощью факторизации и устранения левой рекурсии, но проблема существования эквивалентной LL(k)-грамматики для произвольной неLL(k)-грамматики неразрешима [5]. Расширить класс обрабатываемых языков позволяет использование восходящих LR-анализаторов [6]. Такие анализаторы позволяют обрабатывать более широкий класс грамматик, но имеют более сложную структуру. Восходящие анализаторы позволяют работать с леворекурсивными грамматиками, но даже они не могут бороться со скрытой левой рекурсией [7]. К сожалению, усложнение структуры парсера часто приводит к снижению скорости его работы и усложняет процесс диагностики ошибок.

Существует еще один класс синтаксических анализаторов — обобщенные анализаторы, которые используются для работы с неоднозначными грамматиками. К этому классу относятся алгоритмы обобщенного восходящего (GLR) и нисходящего (GLL) анализа [8]. Алгоритм обобщенного восходящего анализа широко известен и впервые был предложен Томитой [9]. Классический вариант алгоритма не способен обрабатывать любые контекстно-свободные грамматики. В дальнейшем было описано множество модификаций данного алгоритма, однако принцип работы всегда оставался схожим: для конкретной грамматики GLR-алгоритм обрабатывает все возможные варианты разбора входной последовательности, используя обход графа в ширину. GLR-анализаторы используют таблицы синтаксического анализа подобно LR-анализаторам. Отличие таких таблиц от классических состоит в том, что допускается несколько переходов в различные состояния, которые определяются по исходному состоянию парсера и входному терминальному символу. Это обусловлено тем, что в грамматике могут присутствовать неоднозначности и входная последовательность может иметь несколько вариантов разбора. В результате в процессе работы могут возникать конфликты вида сдвиг/свертка (shift/reduce) и свертка/свертка (reduce/reduce). В таких конфликтных ситуациях следует либо выбрать один вариант разбора, что может привести к получению некорректного результата, либо рассмотреть все варианты разбора, что и происходит в процессе работы алгоритма. При обработке каждой конфликтной ситуации необходимо создавать новый стек, верхнее состояние которого соответствует возможному переходу, однако хранение большого числа стеков требует слишком большого объема памяти. Для экономии памяти общие части стеков переиспользуются, а в местах возникновения конфликтов происходит разветвление стека. В случае когда на вершинах разных ветвей находятся одинаковые состояния, ветви могут быть объединены. Таким образом, стек организуется в виде графа, и такая структура данных называется Graph Structured Stack (GSS) [9]. Если для какой-либо вершины стека и входного сим-

вола в таблице парсера не существует ни одного перехода, то соответствующая ветка разбора считается ошибочной и отбрасывается.

Тот же принцип работы лежит и в основе алгоритма обобщенного нисходящего анализа. В процессе работы парсера рассматриваются все возможные варианты, что осуществляется за счет использования дескрипторов. Дескриптор — четверка, описывающая текущее состояние анализатора: индекс во входном потоке, метка функции разбора, текущая вершина стека и фрагмент дерева вывода, которое было построено на момент создания дескриптора. Такой алгоритм в худшем случае работает за кубическое время от длины входной цепочки, а для LL-грамматик [8] — за линейное. Синтаксические анализаторы, построенные с помощью такого алгоритма, позволяют обрабатывать грамматики как со скрытой, так и с обычной левой рекурсией, значительно расширяя класс обрабатываемых нисходящими синтаксическими анализаторами языков.

3 Подходы к генерации синтаксических анализаторов

Для автоматического создания синтаксических анализаторов существует несколько подходов. В рамках первого подхода весь код парсера генерируется по грамматике. Чаще всего такой подход используется при генерации анализаторов, построенных методом рекурсивного спуска. При генерации нисходящих анализаторов для каждого нетерминала генерируются функции, которые последовательно вызываются в процессе разбора. Несмотря на то что нисходящие анализаторы просты для написания и отладки и поэтому чаще всего создаются вручную, существуют инструменты для автоматической генерации таких анализаторов. Например, инструмент ANTLR (Another Tool for Language Recognition) [10] —

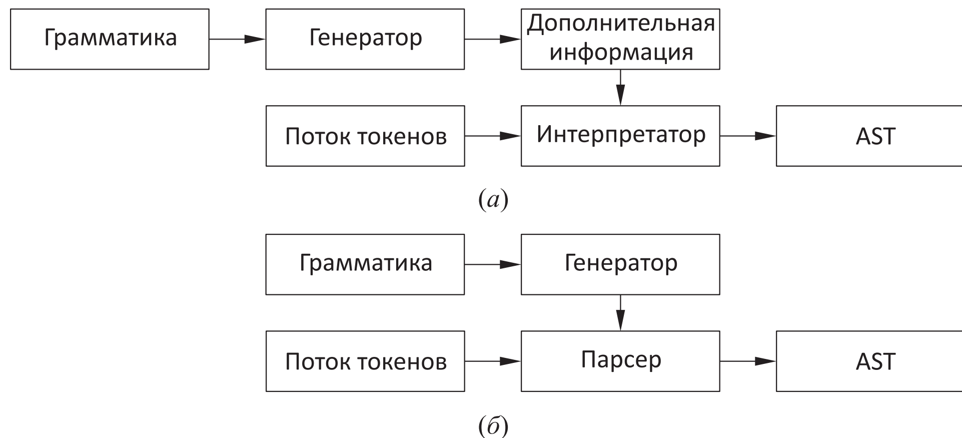


Рис. 1 Структура генератора синтаксических анализаторов: генерирующего весь анализатор (*а*) и только дополнительную информацию (*б*)

генератор парсеров, позволяющий автоматически создавать анализаторы на одном из целевых языков программирования по описанию LL(*)-грамматики на языке, близком к EBNF (Extended Backus–Naur Form). Структура генераторов такого типа изображена на рис. 1, *а*.

Существует еще один подход к генерации синтаксических анализаторов, который используется для получения табличных анализаторов. Отдельно создается интерпретатор, который содержит в себе основную логику алгоритма. Интерпретатор пишется вручную и постоянно переиспользуется. По грамматике каждый раз генерируется дополнительная информация, которая необходима интерпретатору в процессе работы. Структура такого генератора представлена на рис. 1, *б*. Чаще всего в качестве дополнительной информации генерируются таблицы синтаксического анализа, управляющие процессом разбора.

4 Основные принципы обобщенного LL-анализа

Группа ученых во главе с Elizabeth Scott и Adrian Johnstone занимается разработкой алгоритмов синтаксического анализа. Ими было предложено несколько алгоритмов на основе обобщенного восходящего анализа [3, 11, 12] и алгоритм обобщенного нисходящего анализа GLL, о котором и пойдет речь в данной статье.

Синтаксические анализаторы, реализованные с помощью метода рекурсивного спуска, представляют собой набор функций, каждая из которых содержит в себе код для обработки той или иной альтернативы. В алгоритме обобщенного нисходящего анализа для каждого нетерминала генерируется функция, содержащая в себе код для его разбора. Если какой-то нетерминал может быть разобран несколькими способами, то для него генерируется несколько функций, помеченных разными метками. Это метки первого типа, которые используются в алгоритме. Кроме того, бывают метки второго типа — возвратные. Этими метками помечаются фрагменты кода, которые должны быть вызваны после того, как нетерминал был обработан. Для того чтобы запоминать все возможные варианты разбора, используются дескрипторы, содержащие метку функции, которая должна быть вызвана, позицию во входном потоке, стек и дерево разбора. Дескрипторы хранятся в очереди, и выполнение каждый раз возобновляется с точки, которая описана в текущем дескрипторе. Новые дескрипторы добавляются в очередь в процессе работы синтаксического анализатора перед обработкой нетерминалов или после того, как правило было разобрано до конца. Если в процессе выполнения один из вариантов разбора не может быть завершен, то вместо завершения процесса работы парсера с ошибкой из очереди извлекается следующий дескриптор, и процесс продолжается. Разбор завершается, как только очередь становится пустой. Работа парсера контролируется с помощью управляющей функции, которая выполняет следующие действия:

– проверяет, что очередь дескрипторов не пуста;
– извлекает дескриптор;

– присваивает значение глобальным переменным, которые используются в процессе разбора: позиция во входном потоке, стек и дерево;

– вызывает функцию с меткой, изъятой из дескриптора.

На каждом шаге работы алгоритма хранится два указателя: один — в грамматике (слот), а второй — во входном потоке. Слоты имеют следующий вид: $X{::=}x_1 \cdots x_i \bullet x_{i+1} \cdots x_q$, где $\bullet$ указывает позицию перед символом $xi+1$. На каждом этапе разбора имеются слот вида $X{::=}a \bullet Xb$ или $x{::=}a\bullet$ и позиция во входном потоке i. В процессе разбора рассматриваются следующие ситуации.

1. Если текущий символ в грамматике является терминалом x и совпадает с текущим символом во входном потоке, то указатель в грамматике нужно сдвинуть на одну позицию вправо, $X{::=}ax \bullet b$, и увеличить указатель во входном потоке на единицу. Никаких дополнительных действий со стеком при этом не производится. Иначе, если терминал X не совпадает с текущим символом во входном потоке, текущая ветка разбора считается ошибочной, отбрасывается и разбор продолжается с использованием следующего дескриптора.
2. Если текущий символ в грамматике является нетерминалом A, то необходимо в стек записать слот, по которому продолжить разбор после того, как правило для a будет разобрано. Указатель в грамматике перемещается на $A{::=} \bullet b$, а указатель во входном потоке остается без изменений.
3. Если указатель в грамматике имеет вид $X{::=}a\bullet$ и стек не пуст, то слот $Y{::=}bX \bullet c$, который хранится на вершине стека, извлекается и становится текущим.
4. Если текущий слот имеет вид $S{::=}c\bullet$ и весь входной поток рассмотрен, то возвращается дерево, построенное в процессе разбора, иначе разбор заканчивается ошибкой.

Процесс работы синтаксического анализатора не детерминирован, и может возникнуть ситуация, когда один и тот же дескриптор будет создаваться снова и снова. Повторное создание дескриптора приведет к тому, что процесс зациклится и никогда не завершится. Для того чтобы избежать создания одинаковых дескрипторов, отдельно поддерживается множество U, содержащее ранее созданные дескрипторы. Каждый раз, прежде чем добавить новый дескриптор в очередь дескрипторов, выполняется проверка того, не был ли он уже добавлен в множество U.

4.1 Организация стека

Как упоминалось ранее, в таблицах синтаксического анализа для алгоритма обобщенного анализа в каждой ячейке может храниться несколько альтернатив для разбора нетерминала. В таких ситуациях каждый раз создаются новые стеки, верхнее состояние которых соответствует каждому возможному переходу.

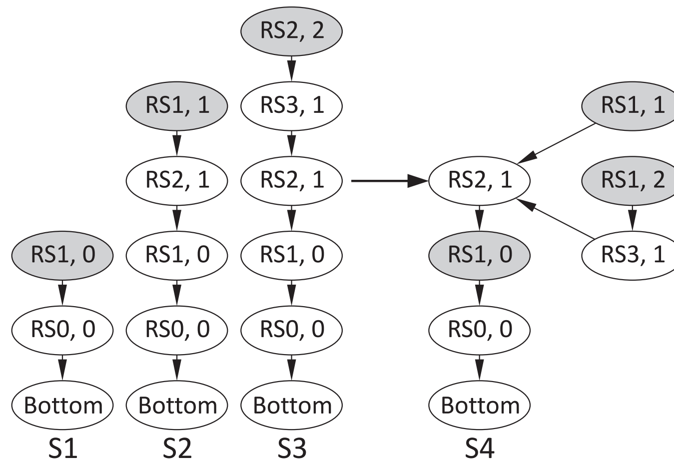


Рис. 2 Переиспользование общих элементов стеков

Проблема такого подхода состоит в том, что для леворекурсивных грамматик число стеков может экспоненциально зависеть от входного потока. Каждый раз при создании нового дескриптора будет создаваться новый стек. Необходимости полностью копировать и каждый раз хранить отдельно каждый стек нет, так как у них есть общие части. Решением проблемы является комбинирование стеков в один с использованием структуры данных GSS (Generic Security Services). С помощью этой структуры можно представить стек в виде графа. Это позволяет вместо хранения всего стека целиком хранить только отдельную вершину графа. Для работы со стеком используется две функции: create( ) для создания новых вершин и pop( ) для изъятия вершин со стека. На вершинах стека хранятся возвратные метки, по которым нужно продолжить разбор после того, как нетерминал будет разобран. На ребрах хранятся фрагменты дерева, которое было построено на момент создания новой вершины стека. Каждая вершина создается лишь однажды и никогда не удаляется в процессе работы алгоритма. Пример переиспользования общих частей стеков представлен на рис. 2: в результате объединения стеков S1, S2 и S3 будет получен стек S4.

4.2 Компактное представление леса разбора

Результатом работы классического синтаксического анализатора для однозначных грамматик является абстрактное синтаксическое дерево, корень которого помечен стартовым нетерминалом, а листья — терминалами. Это дерево используется в дальнейшем для анализа или трансляции. Как упоминалось ранее, для неоднозначных грамматик для одной и той же входной цепочки может существовать несколько деревьев разбора. Для некоторых грамматик число деревьев

может экспоненциально зависеть от размера входа. Для того чтобы уменьшить объем требуемой для хранения деревьев памяти, используется структура Shared Packed Parse Forests (SPPF) [13], которая позволяет хранить деревья более компактно. В SPPF узлы, которые имеют одинаковые поддеревья под ними, переиспользуются, а узлы, которые соответствуют разному выводу одной и той же цепочки из одного и того же нетерминала, комбинируются в упакованный узел.

В алгоритме GLL используется бинаризированная версия SPPF [14], в которой есть промежуточные узлы. Такие промежуточные узлы помечены слотами, т. е. правой частью правила грамматики с указанием позиции в нем. Таким образом, в бинаризированной версии SPPF существуют узлы трех видов: символьные узлы для терминалов или нетерминалов, промежуточные узлы, помеченные слотами, и упакованные узлы, которые позволяют представлять несколько деревьев для одной и той же цепочки. У терминальных узлов нет потомков, а потомками нетерминальных узлов являются упакованные узлы.

4.3 Пример работы алгоритма

Чтобы проиллюстрировать работу оригинального алгоритма и показать, как строятся стек и лес разбора в процессе его работы, обратимся к простому примеру. Рассмотрим неоднозначную грамматику G0 вида

```
0 : S ::= S S
1 : S ::= b
2 : START ::= S
```

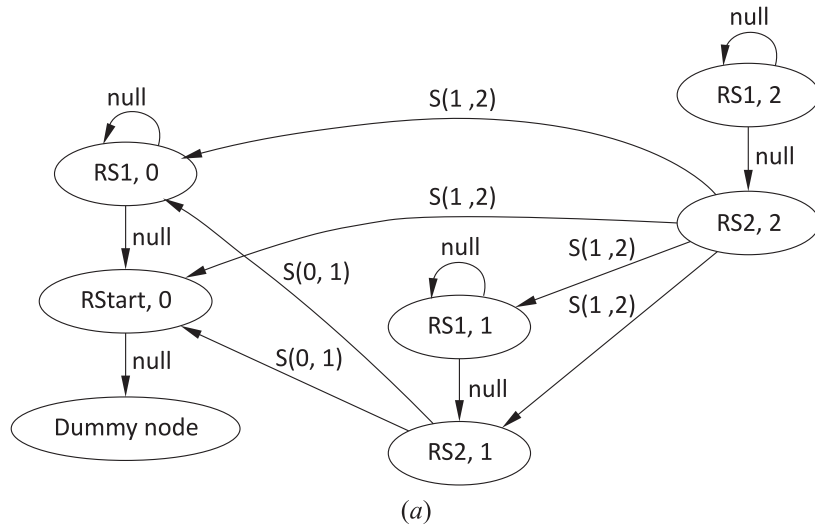
В результате работы генератора для такой грамматики будет получен следующий код (более подробно о генерации кода изложено в статье [15]):

```
read the input into I and set I[m] = $
create GSS node u0 = (L0, 0)
index = 0
GSSNode = u0
cN = emptyAST
cR = emptyAST
setR = empty
setP = empty
goto L_START
L0:
 if (IsNotEmpty(setR)) {
 context = setR.Dequeue()
 index = context.Index
 GSSNode = context.Vertex
label = context.Label
 cN = context.AST
```

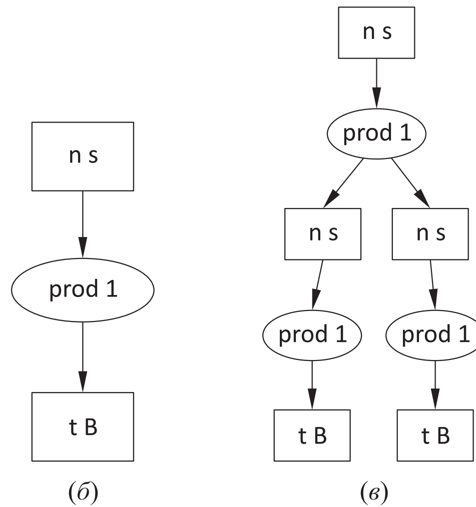
```
 cR = dummyAST
 } else {
 if (there is an SPPF node (START, 0, m)){
 report success
 } else {report failure}
}

L_START:
 if (test(I[index], S, b)) {
 add(LS, GSSNode, index, emptyAST)}
L_S:
 if (test(I[index], S, b)) {
 add(LS1, GSSNode, index, emptyAST)}
 if (test(I[index], S, SS)) {
 add(LS2, GSSNode, index, emptyAST)}
 goto L0
LS1 :
 cN = getNodeT(b, index)
 index = index + 1
 pop(GSSNode, index, cN)
 goto L0
LS2 :
 GSSNode = create(RS1, GSSNode, index, cN)
 goto LS
RS1 :
 if (test(I[currentIndex], S, S)) {
 GSSNode = create(RS2, GSSNode, index, cN)
 goto LS
 } else { goto L0 }
RS2 :
 pop(GSSNode, index, cN)
 goto L0
```

На вход такому синтаксическому анализатору подается цепочка bbb. Построенный в процессе работы стек изображен на рис. 3, *а*. На вершинах стека хранятся возвратные метки и индекс во входном потоке на момент создания вершины. Эти два параметра позволяют уникально определить вершину. На ребрах стека лежат части дерева, которые были получены до того, как начался разбор данного нетерминала. В данном примере на ребрах лежат деревья только такого вида, как показано на рис. 3, *б*. Деревья, получаемые в процессе выполнения функции pop( ) или create( ), подробное описание которых можно посмотреть в оригинальной статье [14], представлены на рис. 3, *в*. (На рис. 3 на ребрах лежит null для пустых деревьев или нетерминал с указанием соответствующего интервала во входном потоке.)



(*а*)



(*б*) (*в*)

Рис. 3 Процесс вывода цепочки bbb и грамматики G0: (*а*) GSS; (*б*) деревья, хранящиеся на стеке; (*в*) деревья, получаемые в результате операций pop( ) или create( )

Результатом работы анализатора является лес разбора. В данном случае лес разбора состоит из двух деревьев, что проиллюстрировано на рис. 4. На рисунке места неоднозначностей отмечены серым цветом. Нетерминальные ячейки помечены буквой n и именем нетерминала, а под ними находятся узлы с номерами продукций, по которым цепочка была выведена. Терминальные ячейки помечены буквой t и терминалом и не имеют потомков.

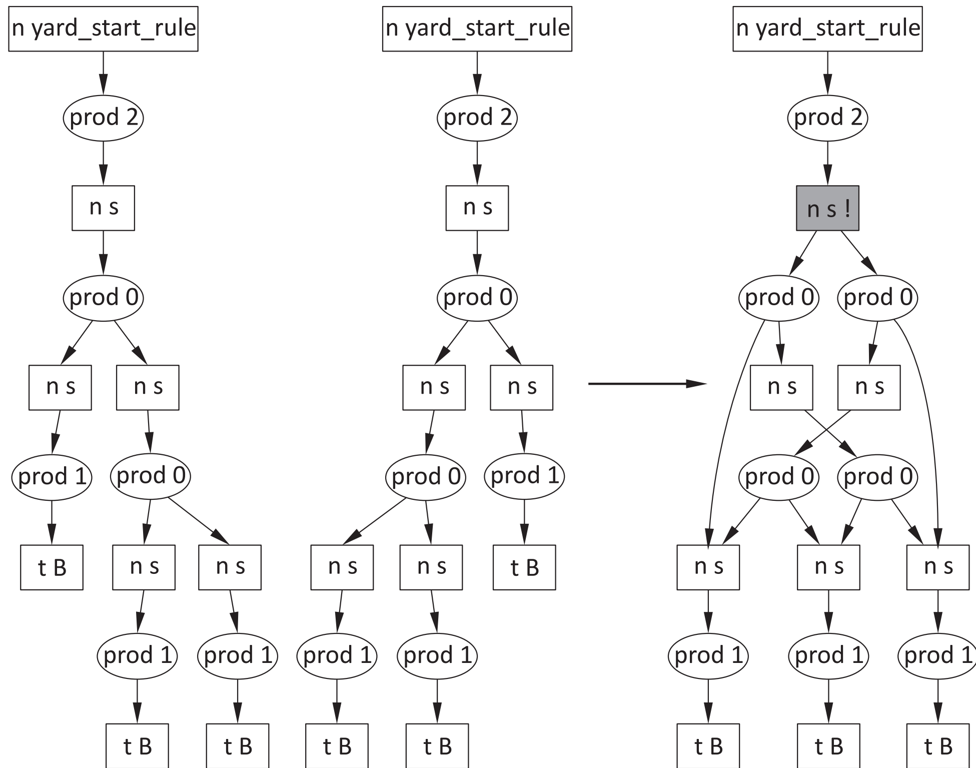


Рис. 4 Сжатие леса разбора в SPPF

5 Абстрактный синтаксический анализ

Абстрактный синтаксический анализ используется для работы со встроенными языками — такими языками, выражения на которых собираются из строк во время исполнения программы и затем передаются соответствующему окружению для выполнения. Выражения на таких языках называются динамически формируемыми выражениями, так как действительное значение, передаваемое на выполнение, будет получено только во время выполнения основной программы. Наиболее распространенные примеры встроенных языков — генерация HTML-страниц, встроенный (в C#, Java, PHP) SQL, динамический SQL.

Так как динамически формируемые выражения — тоже код на некотором языке программирования, то для обеспечения его надежности (отсутствия ошибок, отсутствия мест потенциальных инъекций), так же как и для обычного кода, необходимо выполнение различных анализов, первым шагом многих из которых является синтаксический анализ. Задача осложняется тем, что анализировать каждое возможное значение динамически формируемого выражения нецелесо-

образно и часто невозможно, так как количество таких значений может быть бесконечным. Поэтому предлагаются подходы, которые позволяют проводить анализ некоторого компактного представления множества значений выражения: dataflow уравнения [1], регулярного выражения [1].

Существует несколько подходов к созданию абстрактных синтаксических анализаторов, но чаще всего используется табличный подход. В статье [1] описан пример создания табличного абстрактного анализатора на основе LR-таблиц. В статье [4] описано решение, позволяющее создать абстрактный анализатор на основе обобщенного алгоритма RNGLR. Показано, что использование обобщенного анализа позволяет улучшить абстрактный анализ. Это достигается во многом благодаря использованию структур данных и способов управления ими, свойственных обобщенному анализу. GSS позволяет переиспользовать стеки для общих частей динамически сформированного выражения, а с помощью SPPF можно компактно хранить лес разбора. Это позволяет существенно снизить расход памяти, так как в общем случае при анализе динамически формируемого выражения должно быть получено дерево разбора для каждого возможного значения выражения. Однако на практике часто оказывается, что деревья имеют много общих частей, которые можно переиспользовать.

На основе этого же принципа можно построить абстрактный анализатор с использованием LL-таблиц и алгоритма GLL. Его применение дает следующие преимущества:

(1) высокую скорость работы для леворекурсивных грамматик;

(2) возможность более качественной и простой диагностики и восстановления после ошибок.

Второй пункт является особенно важным, так как качественная и понятная для пользователя диагностика ошибок в абстрактном анализе существенно затруднена сложностью структуры входных данных. По этой причине упрощение механизма диагностики ошибок при сохранении или повышении качества — важная задача.

Для получения перечисленных преимуществ была поставлена задача создания табличного анализатора с использованием алгоритма обобщенного нисходящего синтаксического анализа.

6 Описание модификаций

Поскольку в рамках работы необходимо было реализовать табличный анализатор, процесс работы которого отличается от описанного в статьях [8, 15], в алгоритм были внесены некоторые изменения.

Грамматика перед началом работы синтаксического анализатора представляется в удобной для работы с ней форме. Каждому символу грамматики сопоставляется число; правила хранятся как пара массивов (левые и правые части). В качестве дополнительной информации, которая необходима интерпретатору в процессе работы, генерируются функции для работы с грамматикой,

например, сопоставляющие числам их строковое представление, и таблицы синтаксического анализа. Таблицы, как и в классическом LL-анализе, используются для определения альтернативы, по которой будет осуществляться разбор. В зависимости от символа во входном потоке и текущего нетерминала выбирается альтернатива и в очередь добавляется новый дескриптор. В местах конфликтов создается и записывается дескриптор для каждой из альтернатив. В ячейках таблицы хранятся номера продукций, по которым осуществляется разбор. В дескрипторах вместо метки функции, которую необходимо вызвать, хранится пара чисел: номер правила и позиция в нем. Таким образом, последовательный вызов функций заменяется на обход грамматики с использованием входной цепочки. Вместо нескольких функций, соответствующих нетерминалам, используется пара взаимно рекурсивных функций: управляющая dispatcher( ) и обрабатывающая processing( ). Функция dispatcher( ) выполняет ту же роль, что и раньше: извлекает дескрипторы из очереди, устанавливает значение локальных переменных (позиция в грамматике, позиция во входном потоке, стек и дерево) и вызывает обрабатывающую функцию. Обрабатывающая же функция состоит из последовательности инструкций if-then-else, в телах которых содержится код для обработки всех возможных ситуаций.

1. Если текущий рассматриваемый символ в грамматике $x = A$, где A — терминал, то необходимо перейти к рассмотрению следующего символа в правиле, а указатель во входном потоке увеличить на единицу.
2. Если x — это нетерминальный символ, то в стек записывается текущее правило и запоминается позиция в нем. С использованием этой информации будет осуществляться обработка после того, как нетерминал x будет обработан до конца. А рассматриваемым становится правило, по которому раскрывается x в зависимости от текущего символа во входном потоке. Указатель во входном потоке остается без изменений.
3. Если какое-то правило рассмотрено до конца и текущий стек не пуст, то извлекается дескриптор с вершины стека и продолжается работа с этими данными.

Таким образом, обрабатывающая функция просто выполняет разные действия в зависимости от ситуации. В дескрипторах вместо хранения метки функции, по которой будет осуществляться разбор, теперь хранится пара чисел: номер правила и позиция в нем. В процессе разбора указатель в грамматике сдвигается.

В листинге, приведенном ниже, представлен код получившегося интерпретатора. Вспомогательные функции, такие как pop( ), getNodeP( ), getNodeT, add( ) и create( ), описаны в статье [15].

Инициализация глобальных переменных:

```
read the input into I and set I[m] := $
label = (startRule, 0)
GSSNode = create dummy GSS
```

```
cN = dummyAST
cR = dummyAST
setR = empty
setP = empty
condition = false
stop = false
index = 0
```

Управляющая функция, которая при генерации кода носила название L_Dispatch, выполняет все те же действия, что и ранее:

```
let rec dispatcher () =
  if setR.Count <> 0
  then
    currentContext := setR.Dequeue()
    currentIndex := currentContext.Value.Index
    currentGSSNode := currentContext.Value.Vertex
    currentLabel := currentContext.Value.Label
    currentN := currentContext.Value.Ast
    currentR := dummyAST
    condition := false
  else
    stop := true
```

Обрабатывающая функция, выполняющая различные действия в зависимости от ситуации:

```
and processing () =
 condition := true
 if length rule = 0
 then pop !currentGSSNode !currentIndex !currentN
 elif length rule <> position
 then
  if !currentIndex < inputLength
  then
   if isTerminal curSymbol
   then
    if curSymbol = curToken
    then
     if !currentN = dummyAST
     then currentN := getNodeT !currentIndex
     else currentR := getNodeT !currentIndex
     currentIndex := !currentIndex + 1
     currentLabel := createNewLabel rule (position + 1)
     if !currentR <> dummyAST
     then currentN := getNodeP !currentLabel !currentN !currentR
```

```
     condition := false
   else
    let rules = table[curSymbol][curToken]
    currentGSSNode :=
     create (createNewLabel rule (position + 1))
           !currentGSSNode !currentIndex !currentN
    if Array.length rules <> 0
    then
     for rule in rules do
      let newLabel = createNewLabel rule 0
      addContext newLabel !currentIndex !currentGSSNode dummyAST
  else condition := true
 else
  let curRight = !currentN
  let extension = curRight.extension
  let resTree = findTree key fam
  if key = finalExtension
  then resultAST := Some resTree
  pop !currentGSSNode !currentIndex resTree
       currentN.Value.extension
let control () =
  while not !stop do
     if !condition then dispatcher() else processing()
control()
```

Таким образом, в алгоритме рассмотрены все возможные ситуации, которые могут возникнуть во время разбора. Вместо частного поведения функций, построенных по грамматике, были выделены общие ситуации.

7 Обсуждение

В данной работе описан подход к созданию табличного анализатора на основе алгоритма GLL. В рамках проекта YaccConstructor [2] на языке программирования F# [16] создана рабочая версия анализатора, которая способна работать с левой, правой и скрытыми рекурсиями. К сожалению, его производительность на данном этапе ниже, чем у имеющейся реализации алгоритма RNGLR. В [17] приведено сравнение анализаторов, созданных с использованием разных алгоритмов обобщенного анализа, демонстрирующее, что производительность GLL-анализатора значительно выше, чем производительность RNGLR-анализатора. В дальнейшем планируется оптимизация структур данных для улучшения производительности анализатора. В статье, описывающей технические детали реализации алгоритма GLL, представлены достаточно сложные структуры данных для хранения деревьев и стека, использование которых и планируется в дальнейшем.

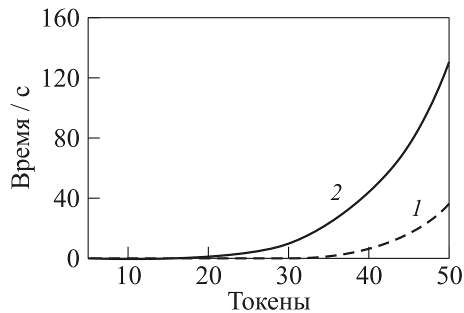


Рис. 5 Сравнение GLL-анализатора (*1*) и RNGLR-анализатора (*2*)

На данном этапе уже были произведены некоторые оптимизации, что позволило значительно улучшить скорость работы самой первой версии анализатора. Для хранения меток, например, используются сжатые числа, что позволяет быстро сравнивать их. Для работы с вершинами стека также используются два сжатых числа — координаты в таблице. Это позволяет быстро осуществлять поиск и обращение к нужной вершине, а сравнение вершин превращается в обычное сравнение чисел. На рис. 5 отображены результаты сравнения производительности RNGLR-анализатора и GLL-анализатора на неоднозначной леворекурсивной грамматике G0, которая ранее рассматривалась в качестве примера.

После этого на базе классического анализатора необходимо получить абстрактный анализатор для работы со встроенными языками. Алгоритм RNGLR имеет сложности с диагностикой ошибок, а в абстрактном анализе процесс работы и структура анализатора усложняются, что делает диагностику ошибок неточной [18]. Диагностика ошибок в классическом LL-анализе гораздо более качественная и простая, чем в LR-анализе. Анализатор на основе GLL наследует преимущества LL-анализаторов, и предполагается, что таких проблем не возникнет в абстрактном GLL-анализаторе. Однако данный вопрос требует отдельных исследований.

Литература

1. *Kyung-Goo D., Hyunha K., Schmidt D. A.* Abstract parsing: Static analysis of dynamically generated string output using LR parsing technology // 16th Symposium (International) on Static Analysis Proceedings. — Berlin: Springer-Verlag, 2009. P. 256–272.
2. YaccConstructor. https://github.com/YaccConstructor/YaccConstructor.
3. *Scott E., Johnstone A.* Right nulled GLR parsers // ACM Trans. Program. Lang. Syst., 2006. Vol. 28. No. 4. P. 577– 618. doi: 10.1145/1146809.1146810.
4. *Grigoriev S. V., Kirilenko I. A.* GLR-based abstract parsing // 9th Central & Eastern European Software Engineering Conference in Russia Proceedings. — New York, NY, USA: ACM, 2013. Article No. 5.
5. *Rosenkrantz D. J., Stearns R. E.* Properties of deterministic top down grammars // 1st Annual ACM Symposium on Theory of Computing Proceedings. — New York, NY, USA: ACM, 1969. P. 165–180.

6. *Aho A. V., Lam M. S., Sethi R., Ullman J. D.* Compilers: Principles, techniques, and tools. — 2nd ed. — Boston: Addison-Wesley Longman Publs. Co., Inc., 2006. 1184 p.
7. *Nederhof M.-J., Sarbo J. J.* Increasing the applicability of LR parsing // 3rd Workshop (International) on Parsing Technologies. — Tilburg, 1993. P. 187–201.
8. *Scott E., Johnstone A.* GLL parsing // Electronic Notes Theoretical Computer Sci., 2010. Vol. 253. Iss. 7. P. 177–189. doi: 10.1016/j.entcs.2010.08.041.
9. *Tomita M.* Efficient parsing for natural language: A fast algorithm for practical systems. — Norwell: Kluwer Academic Publs., 1985. 201 p.
10. ANTLR. http://www.antlr.org.
11. *Scott E., Johnstone A.* Generalized bottom up parsers with reduced stack activity // Computer J., 2005. Vol. 48. No. 5. P. 565–587. doi: 10.1093/comjnl/bxh102.
12. *Scott E., Johnstone A., Economopoulos R.* BRNGLR: A cubic Tomita-style GLR parsing algorithm // Acta Inform., 2007. Vol. 44. No. 6. P. 427– 461. doi: 10.1007/s00236-007-0054-z.
13. *Rekers J. G.* Parser generation for interactive environments. — Amsterdam: University of Amsterdam, 1992. PhD Thesis. 174 p.
14. *Scott E., Johnstone A.* GLL parse-tree generation // Sci. Comput. Program., 2013. Vol. 78. No. 10. P. 1828–1844. doi: 10.1016/j.scico.2012.03.005.
15. *Johnstone A., Scott E.* Modelling GLL parser implementations // 3rd Conference (International) on Software Language Engineering. — Berlin: Springer-Verlag, 2010. P. 42–61.
16. *Syme D., Granicz A., Cisternino A.* Expert F#. — New York, NY, USA: Apress, 2007. 609 p.
17. *Economopoulos R. G.* Generalised LR parsing algorithms. — London, U.K.: Department of Computer Science, Royal Holloway, University of London, 2006. PhD Thesis. 232 p.
18. *Вербицкая Е. А.* Диагностика ошибок при анализе встроенных языков. — СПб.: СПбГУ, 2014. Курсовая работа. 18 с.

Поступила в редакцию 20.01.15

GENERALIZED TABLE-BASED LL-PARSING

S. V. Grigorev and A. K. Ragozina

Saint-Petersburg State University, 7-9 Universitetskaya Nab., St. Petersburg199034, Russian Federation

Abstract: Syntax analysis is an important step of code analysis. The problem is that the grammars have to be in a form which is deterministic, or at least near-deterministic for the chosen parsing technique. Generalized parsing algorithms — Generalized LR and Generalized LL (GLL) — make it possible to remove these restrictions. Abstract analysis makes it possible to parse embedded languages for supporting them in IDE, reengineering tasks, or finding vulnerabilities (SQL-injection). Abstract syntax analysis is based on the classic table-based analysis.

The generalized algorithm of top-down parsing without the use of predictive tables was described earlier in order to extend the class of languages processed by descent analyzers. This paper describes an approach to creation of a table-based GLL-analyzer based on the proposed algorithm, which will be used later for an abstract analyzer. This article describes the algorithm of generalized top-down analysis, its modifications, and the results of comparison with the generalized bottom-up parsing algorithm, which was implemented earlier.

Keywords: generalized parsing; GLL; RNGLR; abstract parsing; string-embedded languages

DOI: 10.14357/08696527150106

References

1. Kyung-Goo, D., K. Hyunha, and D. A. Schmidt. 2009. Abstract parsing: Static analysis of dynamically generated string output using LR-Parsing technology. *16th Symposium (International) on Static Analysis Proceedings*. Berlin: Springer-Verlag. 256–272.
2. YaccConstructor. Available at: http://yaccconstructor.github.io/YaccConstructor/ (accessed February 03, 2015).
3. Scott, E., and A. Johnstone. 2006. Right nulled GLR parsers. *ACM Trans. Program. Lang. Syst.* 28(4):577–618. doi: 10.1145/1146809.1146810.
4. Grigoriev, S. V., and I. A. Kirilenko. 2013. GLR-based abstract parsing. *9th Central & Eastern European Software Engineering Conference in Russia Proceedings*. New York, NY, USA. Article No. 5. 1–9.
5. Rosenkrantz, D. J., and R. E. Stearns. 1969. Properties of deterministic top down grammars. *1st Annual Symposium on Theory of Computing Proceedings*. New York, NY, USA: ACM. 165–180.
6. Aho, A. V., M. S. Lam, R. Sethi, and J. D. Ullman. 2006. *Compilers: Principles, techniques, and tools*. 2nd ed. Boston: Addison-Wesley Longman Publs. Co., Inc. 1184 p.
7. Nederhof, M.-J., and J. J. Sarbo. 1993. Increasing the applicability of LR parsing. *3rd Workshop (International) on Parsing Technologies*. Tilburg. 187–201.
8. Scott, E., and A. Johnstone. 2010. GLL parsing. *Electronic Notes Theoretical Computer Sci.* 253(7):177–189. doi: 10.1016/j.entcs.2010.08.041.
9. Tomita, M. 1985. *Efficient parsing for natural language: A fast algorithm for practical systems*. Norwell: Kluwer Academic Publs. 201 p.
10. ANTLR. Available at: http://www.antlr.org/ (accessed February 03, 2015).
11. Scott, E., and A. Johnstone. 2005. Generalized bottom up parsers with reduced stack activity. *Comput. J.* 48(5):565–587. doi: 10.1093/comjnl/bxh102.
12. Scott, E., A. Johnstone., and R. Economopoulos. 2007. BRNGLR: A cubic Tomita-style GLR parsing algorithm. *Acta Inform.* 44(6):427–461. doi: 10.1007/s00236-007-0054-z.
13. Rekers, J. G. 1992. Parser generation for interactive environments. Amsterdam: Universty of Amsterdam. PhD Thesis. 174 p.

14. Scott, E., and A. Johnstone. 2013. GLL parse-tree generation. *Sci. Comput. Program.* 78(10):1828–1844. doi: 10.1016/j.scico.2012.03.005.
15. Johnstone, A., and E. Scott. 2010. Modelling GLL parser implementations. *3rd Conference (International) on Software Language Engineering*. Berlin: Springer-Verlag. 42–61.
16. Syme, D., A. Granicz, and A. Cisternino. 2007. *Expert F#*. New York, NY, USA: Apress. 609 p.
17. Economopoulos, R. G. 2006. Generalised LR parsing algorithms. London: Department of Computer Science, Royal Holloway, University of London. PhD Thesis. 232 p.
18. Verbitskaya, E. A. 2014. Diagnostika oshibok pri analize vstroennykh yazykov [Error detection in string-embedded languages]. St. Petersburg: Saint-Petersburg State University. Course work. 18 p.

Received January 20, 2015

Contributors

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

Ragozina Anastasiya K. (b. 1992) — student, Saint-Petersburg State University, 7-9 Universitetskaya Nab., St. Petersburg 199034, Russian Federation; ragozina.anastasiya@gmail.com