Обобщённый Табличный LL-анализ

Рагозина Анастасия Константиновна
Санкт-Петербургский Государственный Университет
Санкт-Петербург, Россия
ragozina.anastasiya@gmail.com

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

***Аннотация* — Синтаксический анализ является важным шагом анализа любой системы, поскольку позволяет создать её структурное представление, которое используется в дальнейшем. Для работы с неоднозначными грамматиками используются обобщенные алгоритмы синтаксического анализа (GLR, GLL). Для работы со встроенными языками используется абстрактный синтаксический анализ, основанный на классическом табличном синтаксическом анализе. В данной статье описан подход к созданию табличного GLL-анализатора, который в дальнейшем будет использоваться для получения абстрактного анализатора.**

Ключевые слова — синтаксический анализ; GLL; обобщённый анализ; RNGLR; абстрактный анализ

I. 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 абстактное синтаксическое дерево обычно используется синтаксический анализ, для реализации которого существует большое количество алгоритмов, различающихся классом обрабатываемых языков. Для работы с неоднозначными и недетерменированными грамматиками существуют алгоритмы обобщённого синтаксического анализа: Generalized LR и Generalized LL. Данные алгоритмы анализируют все возможные варианты разбора и строят все деревья. Алгоритм обобщённого нисходящего анализа (GLL) отличается своей простотой и прямолинейностью из-за тесной связи с грамматикой, в то время как алгоритм обобщённого 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, ненадёжной и уязвимой. Ещё 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 для создания 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 обобщённого восходящего анализа RNGLR [3], а после 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 обобщённого нисходящего анализа и модификации, которые были внесены в него для получения табличного анализатора. Также обсуждаются особенности реализации табличного GLL анализатора.

II. Обобщённый 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 не LL-грамматику в эквивалентную ей LL-грамматику с помощью факторизации и устранения левой рекурсии, но проблема существования эквивалентной LL(k)-грамматики для произвольной не LL(k)-грамматики неразрешима [5]. Расширить класс обрабатываемых языков позволяет использование восходящих LR-анализаторов [4]. 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 [6]. К сожалению, усложнение структуры парсера часто приводит к снижению скорости его работы и усложняет процесс диагностики ошибок.

Существует ещё один класс синтаксических анализаторов — обобщенные анализаторы, которые используются для работы с неоднозначными грамматиками. К этому классу относятся алгоритмы обобщённого восходящего анализа Generalised LR (GLR) и алгоритм обобщённого нисходящего анализа Generalised LL (GLL) [7]. Алгоритм обобщённого восходящего анализа широко известен и впервые был предложен Томитой [8]. 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 объёма 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) [7]. Если для какой-либо вершины стека и входного символа 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 обобщённого нисходящего анализа. В процессе работы парсера рассматриваются все возможные варианты, что осуществляется за счёт использования дескрипторов. Дескриптор — четвёрка, 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 [7] — за линейное. Синтаксические анализаторы, построенные с помощью такого алгоритма, позволяют обрабатывать грамматики как со скрытой, так и обычной левой рекурсией, значительно расширяя класс обрабатываемых нисходящими синтаксическими анализаторами языков.

III. 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 [9] — генератор парсеров, позволяющий автоматически создавать анализаторы на одном из целевых языков программирования по описанию LL(*)-грамматики на языке, близком к EBNF. Структура генераторов такого типа изображена на Рисунке 1.

Существует ещё один подход для генерации синтаксических анализаторов, который используется для получения табличных анализаторов. Отдельно создаётся 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 Рисунке 2. Чаще всего в качестве дополнительной информации генерируются таблицы синтаксического анализа, управляющие процессом разбора.

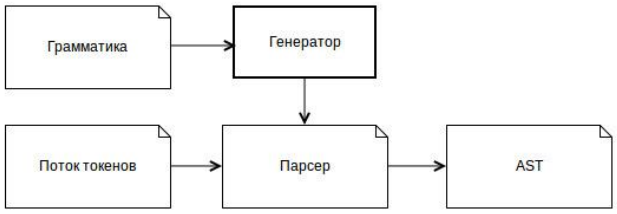

Рисунок 1. Структура генератора синтаксических анализаторов, генерирующего код всего анализатора

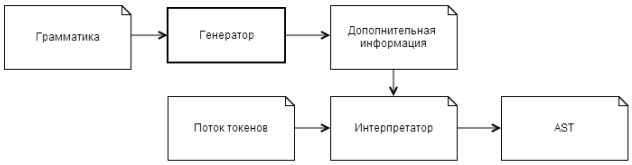

Рисунок 2. Структура генератора синтаксических анализаторов, генерирующего только дополнительную информацию

IV. Основные принципы обобщённого LL-анализа

Группа учёных во главе с Elizabeth Scott и Adrian Johnstone занимается разработкой алгоритмов синтаксического анализа. Ими было предложено несколько алгоритмов на основе обобщённого восходящего анализа [10, 11, 12] и алгоритм обобщённого нисходящего анализа GLL [7], о котором и пойдёт речь в данной статье.

Синтаксические анализаторы, реализованные с помощью метода рекурсивного спуска, представляют из себя набор функций, каждая из которых содержит в себе код для обработки той или иной альтернативы. В алгоритме обобщённого 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, 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 завершён, 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 ::= x1 . . . xi • xi+1 . . . xq, где • указывает позицию перед символом xi+1. На каждом этапе разбора имеется слот вида X ::= α • Xβ или x ::= α• и позиция во входном потоке i. В процессе разбора рассматриваются следующие ситуации.

- 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 ::= αx • β, 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.

- 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 ::= •γ, а указатель во входном потоке остаётся без изменений.

- Если указатель в грамматике имеет следующий вид X ::= α• и стек не пуст, то слот Y ::= δX • μ, который хранится на вершине стека, извлекается и становится текущим.

- Если текущий слот имеет вид S ::= τ• и весь входной поток рассмотрен, то возвращается дерево, построенное в процессе разбора, иначе разбор заканчивается ошибкой.

Процесс работы синтаксического анализатора недетерминирован и может возникнуть ситуация, когда один и тот же дескриптор будет создаваться снова и снова. Повторное создание дескриптора приводит 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.

A. *Организация стека*

Как упоминалось ранее, в таблицах синтаксического анализа для алгоритма обобщённого 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. Проблема такого подхода состоит в том, что для леворекурсивных грамматик количество 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 Graph Structured Stack (GSS) [13]. 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 которой нужно продолжить разбор после того, как нетерминал будет разобран. На рёбрах хранятся фрагменты дерева, которое было построено на момент создания новой вершины стека. Каждая вершина создаётся лишь однажды и никогда не удаляется в процессе работы алгоритма.

Пример переспользования общих частей стеков представлен на Рисунке 3: в результате объединения стеков S1, S2 и S3 будет получен стек S4.

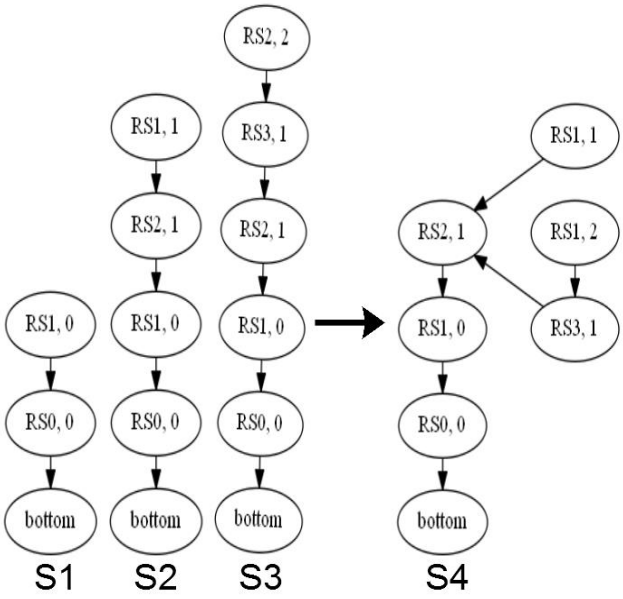

Рисунок 3. Переиспользование общих элементов стеков

B. *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 количество деревьев может экспоненциально зависеть от размера входа. Для того чтобы уменьшить количество 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 нём. Таким образом, в бинаризированной версии SPPF существуют узлы трёх 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.

C. *Пример работы алгоритма*

Для того, чтобы проиллюстрировать, как работает оригинальный алгоритм и как строятся стек и лес разбора в процессе его работы, рассмотрим простой пример. 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 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 Рисунке 4.

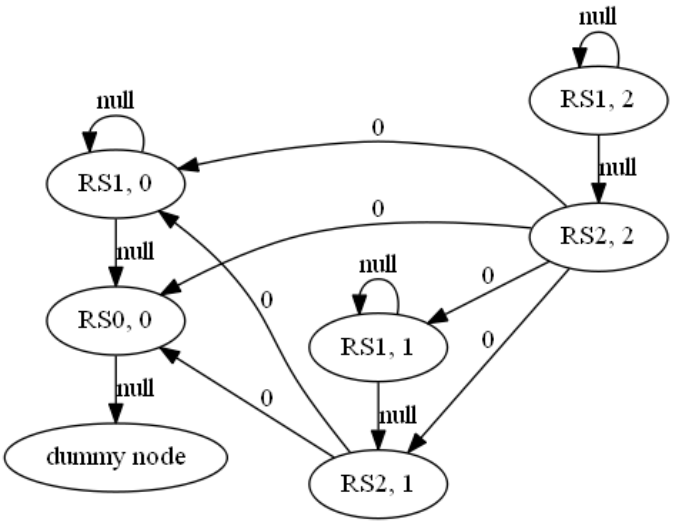

Рисунок 4. GSS для цепочки bbb и грамматики G0

На вершинах стека хранятся возвратные метки и индекс во входном потоке на момент создания вершины. Эти два параметра позволяют уникально определить вершину. На рёбрах стека лежат либо деревья, которые были получены в процессе разбора, либо пустые деревья. (На Рисунке 4 на рёбрах лежат номера продукций, по которым происходит разбор, или null для пустых деревьев.)

Результатом работы анализатора является лес разбора. В данном случае лес разбора состоит из двух деревьев, что проиллюстрировано на Рисунке 5.

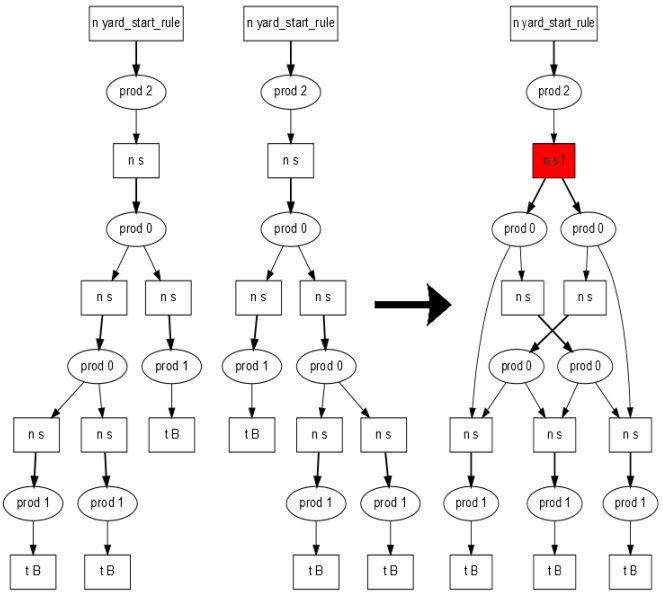

Рисунок 5. Сжатие леса разбора в SPPF

На рисунке места неоднозначностей отмечены красным цветом. Нетерминальные ячйки 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.

V. 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 распространённые 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 надёжности (отсутствие ошибок, отсутствие 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 не целесообразно 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 [16], регулярного выражения [16].

Существует несколько подходов для создания абстрактных синтаксических анализаторов, но чаще всего используется табличный подход. В статье [1] описан пример для создания табличного абстрактного анализатора на основе LR-таблиц. В статье [4] описано решение, позволяющее создать абстрактный анализатор на основе обобщённого алгоритма RNGLR. Показано, что использование обобщённого анализа позволяет улучшить абстрактный анализ. Это достигается во многом благодаря использованию структур данных и способов управления ими, свойственных обобщённому 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.

Используя этот же принцип, возможно построить абстрактный анализатор с использованием LL-таблиц и алгоритма GLL. Его использование обладает следующими преимуществами:

1. высокая скорость работы для леворекурсивных грамматик;
2. возможность более качественной и простой диагностики и восстановления после ошибок.

Второй пункт является 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 обобщённого нисходящего синтаксического анализа.

VI. Описание модификаций

Поскольку в рамках работы необходимо было реализовать табличный анализатор, процесс работы которого отличается от описанного в статьях [7, 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 нём. Таким образом, последовательный вызов функций заменяется на обход грамматики с использованием входной цепочки. Вместо нескольких функций, соответствующих нетерминалам, используется пара взаимнорекурсивных 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 нём. 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 извлекаем дескриптор с вершины стека и продолжаем работу 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 нём. В процессе разбора указатель в грамматике сдвигается.

В листинге, представленном 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 всё те же действия, что и ранее:

```
dispatcher () {
 if (IsNotEmpty(setR)) {
   context = setR.Dequeue()
   index   = context.Value.Index
   GSSNode = context.Value.Vertex
   label   = context.Value.Label
   cN      = context.Value.Ast
   cR      = dummyAST
   condition      = false
 } else {
   if (there is an SPPF node (START; 0; m)) {
     report success
   } else {
     report failure }
   stop = true }
}
```

Обрабатывающая функция, выполняющая различные действия в зависимости от ситуации:

```
processing () {
  condition = true
/*Ситуация, когда правило выводит )*/
  if (length(rule) == 0) {
    cR = getNodeT(index)
    cN = getNodeP(label, cN, cR)
    pop(GSSNode, index, cN)
  } else {
/*Если правило ещё не закончилось*/
    if (length(rule) != position) {
/*Если входной поток ещё не закончился*/
      if (index < inputLength) {
        if (IsTerminal(symbolInGrammar)
          && symbolInGrammar == token)){

          if(cN == dummyAST) {
            cN = getNodeT(index)
          } else {
            cR = getNodeT(index)}
            index = index + 1
            IncrementPosition(label)
            if (cR != dummyAST) {
              cN = getNodeP(label, cN, cR)
              condition = false
        /*Текущий символ в грамматике нетерминал*/
        } else {
          index = getIndex(symbolInGrammar, token)
          newLabel = IncrementPosition(label)
          GSSNode = create(newLabel, GSSNode, index, cN)
        /*Массив с номерами правил, по которым может быть
        разобран текущий нетерминал*/
          rules = table[symbolInGrammar][token]
          if (length(rules) != 0) {
            foreach (rule in rules) {
              newLabel = packLabel(rule, 0)
                add(newLabel, index, GSSNode, dummyAST)
          } else {condition = true}
/*Правило для нетерминала закончилось*/
    } else {
      pop(GSSNode, index, cN)}

while not !stop do
        if (!condition){dispatcher()} else {processing()}
```

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.

VII. Обсуждение

В данной работе описан подход создания табличного анализатора на основе алгоритма GLL. В рамках проекта YaccConstructor [2] на языке программирования F# [17] 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 статье [13] 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, описаны достаточно сложные структуры данных для хранения деревьев и стека, использование которых и планируется в дальнейшем.

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 Рисунке 6 отображены результаты сравнения производительности RNGLR-анализатора и GLL-анализатора на неоднозначной леворекурсивной грамматике G0, которая ранее рассматривалась в качестве примера.

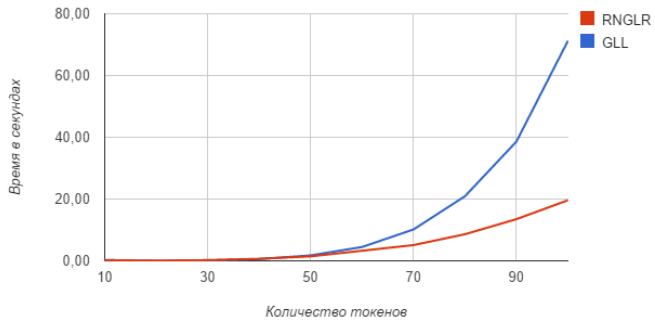

Рисунок 6. Сравнение GLL-анализатора и RNGLR-анализатора

В дальнейшем предполагает на базе классического анализатора 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, но данный вопрос требует отдельных исследований.

Список литературы

[1] Kyung-Goo Doh, Hyunha Kim, and David A. Schmidt: 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. P. 256–272.

[2] Официальный сайт проекта с открытым исходным кодом YaccConstructor recursive-ascent.googlecode.com.

[3] Grigoriev Semyon, Kirilenko Iakov. GLR-based abstract parsing // CEE-SECR'13 Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia.

[4] Alfred V. Aho, Monica S. Lam,Ravi Sethi, and Jeffrey D. Ullman,Compilers: Principles, Techniques, and Tools. Pearson Education, Inc, 2006.

[5] D. J. Rosenkrantz and R. E. Stearns. 1969. Properties of deterministic top down grammars. In Proceedings of the first annual ACM symposium on Theory of computing (STOC '69). ACM, New York, NY, USA, 165-180.

[6] Mark-Jan Nederhof, Janos J. Sarbo. Increasing the Applicability of LR Parsing.University of Nijmegen, Department of Computer Science.

[7] Elizabeth Scott and Adrian Johnstone. GLL Parsing. Electronic Notes in Theoretical Computer Science 253 (2010) pages 177–189.

[8] Masaru Tomita. Eficient parsing for natural language: a fast algorithm for practical systems. Kluwer Academic Publishers, Boston, 1986.

[9] Официальный сайт ANTLR http://www.antlr.org/.

[10] Elizabeth Scott and Adrian Johnstone Right Nulled GLR Parsers.

[11] Elizabeth Scott and Adrian Johnstone and Rob Economopoulos, BRNGLR: a cubic Tomita-style GLR parsing algorithm. "Acta Informatica", 2007, 10.1007/s00236-007-0054-z, volume 44, pages 427--461, issn 0001-5903.

[12] Elizabeth Scott and Adrian Johnstone. Generalized bottom up parsers with reduced stack activity, 2005, In : The Computer Journal. 48, 5, p. 565-587.

[13] Giorgios Robert Economopoulos. Generalised LR parsing algorithms. Department of Computer Science, Royal Holloway, University of London, August, 2006, p 232.

[14] Elizabeth Scott and Adrian Johnstone. GLL parse-tree generation, 2013 In : Science of Computer Programming. 78, 10, p. 1828–1844, article.

[15] Elizabeth Scott and Adrian Johnstone. Modeling GLL parser implementation, 2011 Software Language Engineering: Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers. Malloy, B., Staab, S. & van den Brand, M. (eds.). Springer, p. 42-61 (Lecture Notes in Computer Science; vol. 6563), conference contribution.

[16] REKERS, J. G. 1992. Parser generation for interactive environments. Ph.D. thesis, Universty of Amsterdam.

[17] Syme D., Granicz A., and Cisternino A.: Expert F#, Apress (2007).

[18] Вербицкая Екатерина. Диагностика ошибок при анализе встроенных языков. Курсовая работа, Санкт-Петербургский государственный университет, 2014.