Санкт-Петербургский государственный университет

Кафедра системного программирования

Ковалев Дмитрий Александрович

Реализация и оценка эффективности алгоритма обобщенного синтаксического анализа с уменьшенной активностью стека

Курсовая работа

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

Санкт-Петербург
2016

Оглавление

Введение 3

1. Постановка задачи 5

2. Обзор 6

2.1. Контекстно-свободные грамматики и вложенная рекурсия 6

2.2. RIGLR-алгоритм синтаксического анализа 6

2.3. Проект YaccConstructor 8

3. Реализация RIGLR-алгоритма 9

3.1. Архитектура 9

3.2. Модуль генератора 10

3.2.1. Реализация RIA и RCA 10

3.2.2. Алгоритм удаления вложенной рекурсии 10

3.2.3. Алгоритм работы генератора 11

3.3. Модуль интерпретатора 12

4. Анализ и сравнение 13

Заключение 15

Список литературы 16

Введение

Многие существующие сегодня программы во время выполнения формируют из строковых литералов выражения на некотором языке, которые могут переданы для исполнения в соответствующее окружение. Примерами этого могут служить динамическое создание SQL-запросов к базам данных и генерация HTML-страниц.

Одним из недостатков динамической генерации кода является то, что во время компиляции основной программы невозможно провести синтаксический анализ формируемых выражений. Это приводит к отсутствию информации о корректности построенного выражения до момента выполнения программы, что, в свою очередь, существенно увеличивает затраты на разработку и отладку.

Для решения данной проблемы применяют методы статического анализа множества значений динамически формируемого выражения. В рамках исследовательского проекта YaccConstructor [6] был предложен алгоритм [5] синтаксического анализа регулярной аппроксимации множества значений формируемого выражения. Данный алгоритм позволяет получить компактное представление множества деревьев разбора корректных значений выражения (при этом некорректные значения отбрасываются).

Основой для разработанного алгоритма служит RNGLR-алгоритм [4] синтаксического анализа, который позволяет работать с любыми контекстно-свободными грамматиками, в том числе неоднозначными. Для обработки конфликтов, вызванных неоднозначностью грамматики, RNGLR-алгоритм разделяет стек анализатора на две или более ветви, которые в дальнейшей работе также могут быть разделены. Для эффективного представления множества ветвей стека в RNGLR используется специальная структура, называемая структурированным в виде графа стеком (Graph Structured Stack). Несмотря на это, необходимость одновременно работать с многими состояниями стека оказывает существенное влияние на производительность алгоритма.

В статье Elizabeth Scott и Adrian Johnstone был представлен алго-

ритм [3] синтаксического анализа, позволяющий, как и RNGLR, работать с произвольными КС-грамматиками, при этом значительно сокращая использование стека. Данный алгоритм получил название *Reduction Incorporated Generalized LR (RIGLR)*. Использование RIGLR-алгоритма вместо RNGLR, предположительно, могло быть бы увеличить производительность компоненты, отвечающей за синтаксический анализ динамически формируемых выражений в проекте YaccConstructor.

1. Постановка задачи

Целью данной работы является улучшение производительности алгоритма синтаксического анализа динамически формируемого кода. Для ее достижения были поставлены следующие задачи.

- Изучить RIGLR-алгоритм синтаксического анализа.
- Реализовать генератор синтаксических анализаторов, основанный на RIGLR-алгоритме.
- Провести сравнение производительности анализаторов, созданных на основе RNGLR и RIGLR.

2. Обзор

В данном разделе приведен необходимый теоретический материал, дано краткое описание RIGLR-алгоритма, а также описан проект, в рамках которого проводилась его реализация.

2.1. Контекстно-свободные грамматики и вложенная рекурсия

Контекстно-свободная грамматика G состоит из множества нетерминалов N, множества терминалов T, такого, что $N \cap T = \emptyset$, стартового нетерминала $S \in N$ и набора правил вида $A ::= \alpha$, где $A \in N$ и α — строка из терминалов и нетерминалов (возможно пустая).

Шагом вывода в грамматике называют выражение вида $\gamma \Rightarrow \delta$, где γ, δ — строки из терминалов и нетерминалов, и δ получается из γ путем замены нетерминала A на строку α, где $A ::= \alpha$ — правило грамматики. *Вывод* δ из γ — это последовательность шагов вывода $\gamma \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow ... \Rightarrow \beta_{n-1} \Rightarrow \delta$.

Говорят, что в грамматике G присутствует *вложенная рекурсия*, если для некоторого нетерминала A и строк $\alpha, \beta \neq \epsilon$, где ϵ — обозначение для пустой строки, в этой грамматике существует вывод $A \Rightarrow^* \alpha A \beta$.

2.2. RIGLR-алгоритм синтаксического анализа

RIGLR-алгоритм позволяет свести обращения к стеку в процессе анализа к обработке правил, отражающих вложенную рекурсию. Для достижения данного результата алгоритм использует управляющие таблицы, отличные от стандартных LR-таблиц.

Процесс их построения начинается с удаления из исходной грамматики вложенной рекурсии — если для некоторого нетерминала A существует вывод $A \Rightarrow^* \alpha A \beta$, $\alpha, \beta \neq \epsilon$, в правой части последнего правила в выводе вхождение A заменяется на особый терминал $A^{\perp}$. Для примера рассмотрим грамматику правильных скобочных последовательностей (1), которая после преобразования принимает вид (2).

$$S \to (\ S\ S\ )\ |\ a\ |\ \epsilon$$

Листинг 1: Грамматика G_0

$$S \to (\ S^{\perp}\ S^{\perp}\ )\ |\ a\ |\ \epsilon$$

Листинг 2: Грамматика G_1

Далее, для каждого нетерминала A, за исключением стартового, рассматривается грамматика G_A с тем же набором правил, что и в исходной, но со стартовым правилом вида $S_A ::= A$. Для каждой такой грамматики строится *Reduction Incorporated Automaton (RIA)* — автомат, схожий с LR(0)-автоматом, но включающий в себя особые ребра, соответствующие действиям свертки. Наконец, все RIA(G_A) объединяются в финальный автомат, получивший название *Recursion Call Automaton (RCA)*. Объединение происходит следующим образом: все ребра (s, t) с меткой $A^{\perp}$ в каком-либо из автоматов удаляются, а вместо них добавляются ребра $(s, startState(RIA(G_A)))$ с меткой $push(t)$. Табличное представление RCA и называют управляющей таблицей RIGLR.

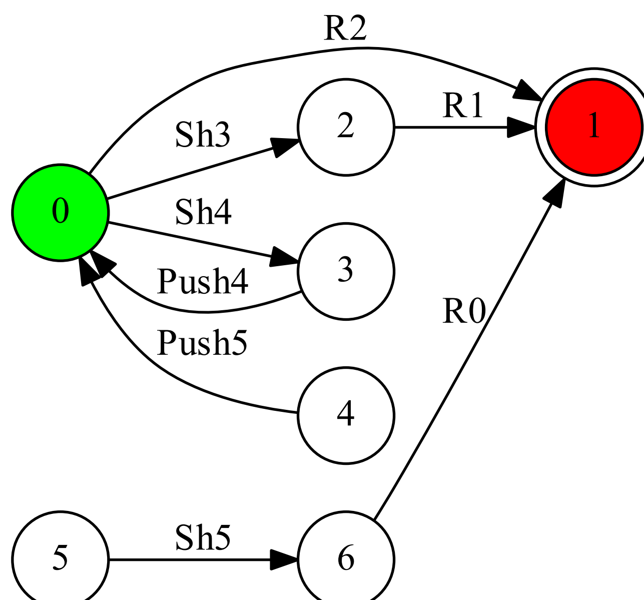

Рис. 1: RCA для грамматики G_1

Стек RIGLR-алгоритма представляется в виде *Recursion Call Graph (RCG)* — ориентированного графа, вершина которого хранит номер состояния, в которое необходимо совершить возврат после разбора рекурсивного нетерминала. В процессе работы RCG изменяется лишь в том случае, если запись в управляющей таблице для текущих состояния и токена содержит действие $push(l)$ (т.е. в RCA из текущего состояния выходит ребро с меткой $push$), добавляющее в RCG вершину с меткой l и ребро из новой вершины в текущую. Действия переноса и свертки создают узлы в дереве разбора, не модифицируя стек.

Результатом работы RIGLR-анализатора является сжатое представление леса разбора — *Shared Packed Parse Forest (SPPF)* [2], компактно представляющее все возможные варианты разбора входной цепочки.

2.3. Проект YaccConstructor

YaccConstructor (далее YC) [6] — исследовательский проект лаборатории языковых инструментов JetBrains на математико-механическом факультете СПбГУ, направленный на исследования в области лексического и синтаксического анализа, а также статического анализа встроенных языков. Проект включает в себя одноименную модульную платформу для разработки лексических и синтаксических анализаторов, содержащую большое количество компонент: язык описания грамматик YARD, преобразования над грамматиками и др. Основным языком разработки является F#.

3. Реализация RIGLR-алгоритма

3.1. Архитектура

В рамках данной работы был реализован новый модуль платформы YC, который, в терминах проекта, представляет из себя генератор. На рисунке 2 изображена архитектура платформы, цветом выделен реализованный модуль.

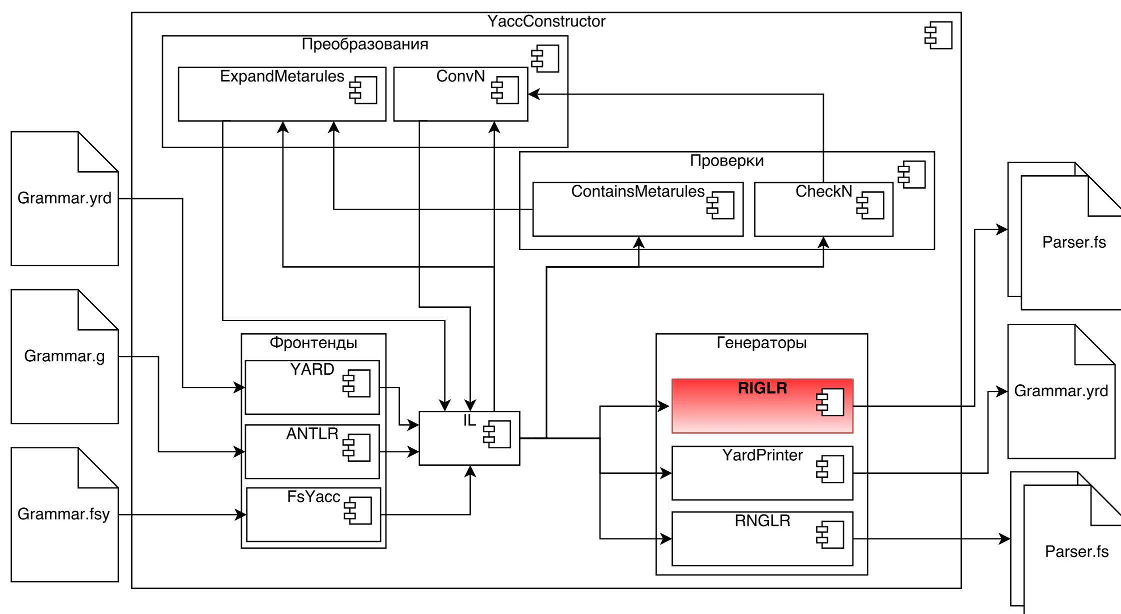

Рис. 2: Архитектура YC

Более подробную структуру модуля можно увидеть на рисунке 3. Основными компонентами являются генератор, который по входной грамматике создает файл с управляющими таблицами и дополнительной информацией для интерпретатора, и интерпретатор, содержащий основную логику алгоритма и структуры данных, необходимые для его работы.

Пользователь при создании приложения, использующего модуль, добавляет в свой проект сгенерированный файл, ссылку на интерпретатор и файл, содержащий лексический анализатор (полученный с помощью другого модуля YC, который не описывается в данной работе), и вызывает соответствующую функцию для синтаксического анализа.

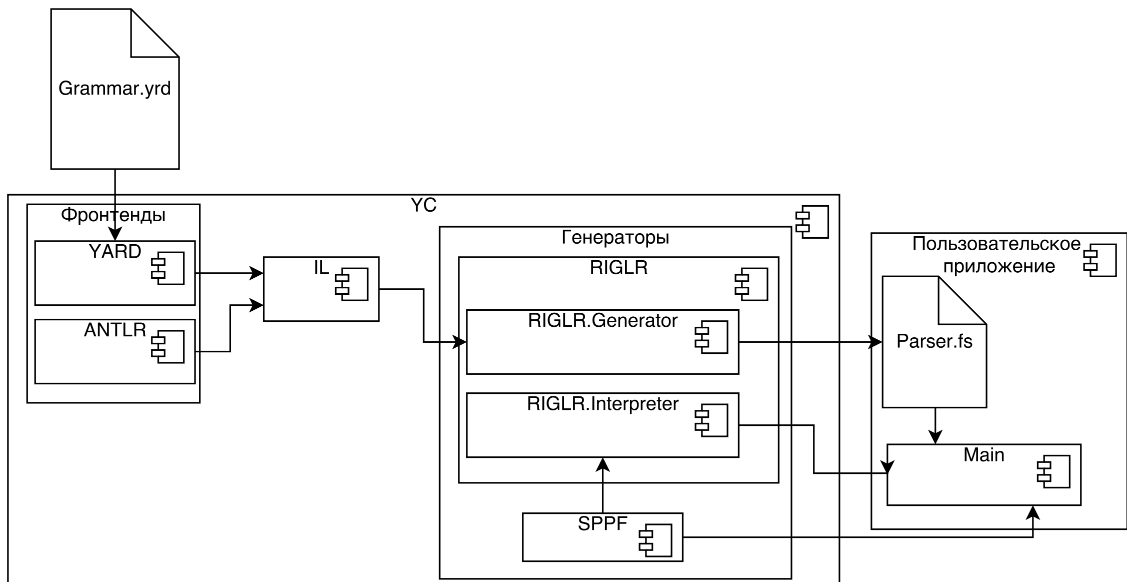

Рис. 3: Архитектура реализованного модуля

3.2. Модуль генератора

Модуль генератора содержит в себе реализацию классов автоматов (RIA и RCA), необходимых для построения управляющих таблиц, а также реализацию алгоритма удаления вложенной рекурсии, который основан на алгоритме поиска данного вида рекурсии, описанном в статье [1].

3.2.1. Реализация RIA и RCA

Классы, описывающие RIA и RCA, унаследованы от класса FSA<'T> библиотеки QuickGraph [1]. Данный класс представляет реализацию конечного автомата с базовым набором функций (например, ϵ-замыканием). RIA и RCA добавляют новые конструкторы, позволяющие инициализировать соответствующий автомат по грамматике, которая была передана как параметр, используя алгоритмы из статьи [3].

3.2.2. Алгоритм удаления вложенной рекурсии

Для удаления вложенной рекурсии исходная грамматика представляется в виде *Labelled Production Graph (LPG)* — помеченного ориенти-

[1]Проект с открытым исходным кодом. Ссылка — https://github.com/YaccConstructor/QuickGraph

рованного графа, вершинами которого являются нетерминалы грамматики, а ребра строятся следующим образом: ребро $A \to B$ добавляется в граф, если в грамматике существует правило $A \to \alpha B \beta$. Метка данного ребра определяется так:

$$label(A \to B) = \begin{cases} L, & \text{если для всех } A \to \alpha B \beta \text{ выполнено } \alpha \neq \epsilon, \beta = \epsilon \\ R, & \text{если для всех } A \to \alpha B \beta \text{ выполнено } \alpha = \epsilon, \beta \neq \epsilon \\ B & \text{в ином случае.} \end{cases}$$

Путь в графе носит тип $L(R)$, если метки всех ребер в этом пути равны $L(R)$; в ином случае тип пути — B. Нетерминал является самовставленным, если для его вершины существует цикл типа B, либо существуют два цикла — типа L и R. Для исключения вложенной рекурсии из грамматики необходимо разомкнуть подобные циклы для всех нетерминалов.

Реализованный алгоритм разделяет LPG на компоненты сильной связности, затем в каждой компоненте для каждой вершины находит указанные типы циклов при помощи обхода в глубину и размыкает их — пусть найден цикл для нетерминала B с последним ребром вида $A \to B$, тогда для всех правил в грамматике вида $A \to \alpha B \beta$ вхождения B в правой части заменяются на специальный терминал $B^{\perp}$; ребро удалется из графа.

3.2.3. Алгоритм работы генератора

Общий ход работы генератора выглядит следующим образом:

- генератор принимает на вход промежуточное представление грамматики, IL, полученное при помощи одного из фронтендов YC из текстового файла соответствующего формата;
- внутри генератора IL преобразуется в другое представление, называемое FinalGrammar, которое упрощает дальнейшую работу с грамматикой;

- после получения необходимого представления грамматики, генератор модифицирует ее, удаляя вложенную рекурсию;
- по грамматике, не содержащей вложенной рекурсии, строится RCA;
- RCA преобразуется в табличный вид, таблица и дополнительная информация записываются в сгенерированный файл *.fs.

3.3. Модуль интерпретатора

В модуле интерпретатора находится описание функции buildAst<'T>, реализующей основную логику RIGLR-анализатора. Функция принимает на вход управляющие таблицы и поток токенов и возвращает лес разбора (SPPF) для данной строки, либо сообщение об ошибке. Модуль переиспользует компоненту с описанием SPPF, реализованную ранее для RNGLR-алгоритма.

4. Анализ и сравнение

Для проведения сравнительного анализа использовалась сильно неоднозначная грамматика, приведенная на листинге 3. Данная грамматика инициирует худший случай работы для GLR-алгоритмов.

$$S \rightarrow S\ S\ S \mid S\ S \mid a$$

Листинг 3: Грамматика G_2

Алгоритмы сравнивались по следующим показателям: размер стека (GSS для RNGLR и RCG для RIGLR; таблица 1), количество просмотров ребер стека в процессе работы (рис. 4), время работы (рис. 5).

Токены	RNGLR	RIGLR
10	30	20
20	60	40
30	90	60
40	120	80
50	150	100

Таблица 1: Кол-во вершин в стеке

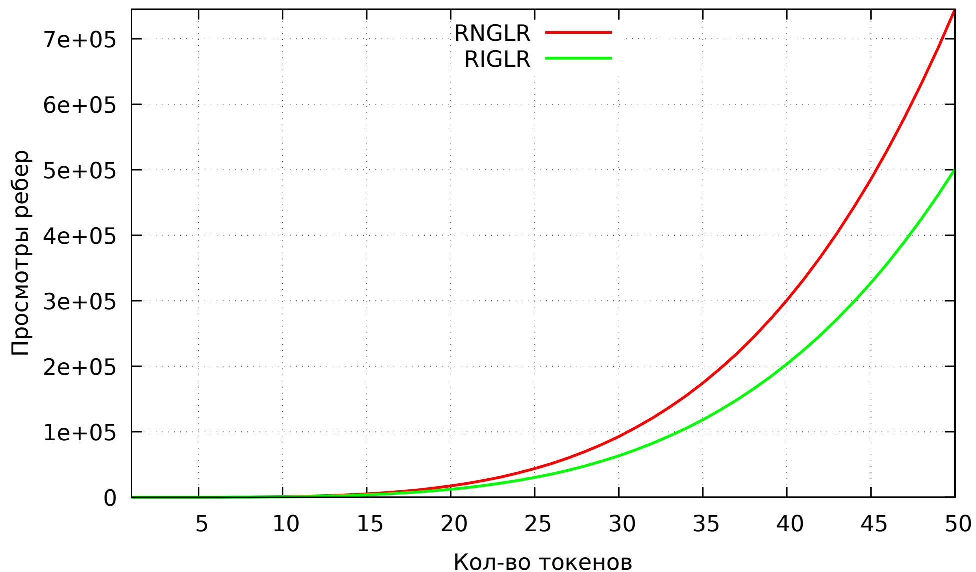

Рис. 4: Просмотры ребер стека

В таблице 1 можно увидеть, что количество вершин в стеке RIGLR-алгоритма меньше примерно в 1.5 раза, что соответствует теоретиче-

ской оценке из оригинальной статьи. Асимптотическое поведение, выражаемое в количестве просмотров ребер стека, также оказалось лучше для RIGLR, что можно увидеть на рисунке 4.

По времени работы RIGLR уступает текущей реализации RNGLR-алгоритма (рисунок 5). Для увеличения производительности необходима оптимизация структур данных, используемых во время выполнения, и модификация самого алгоритма.

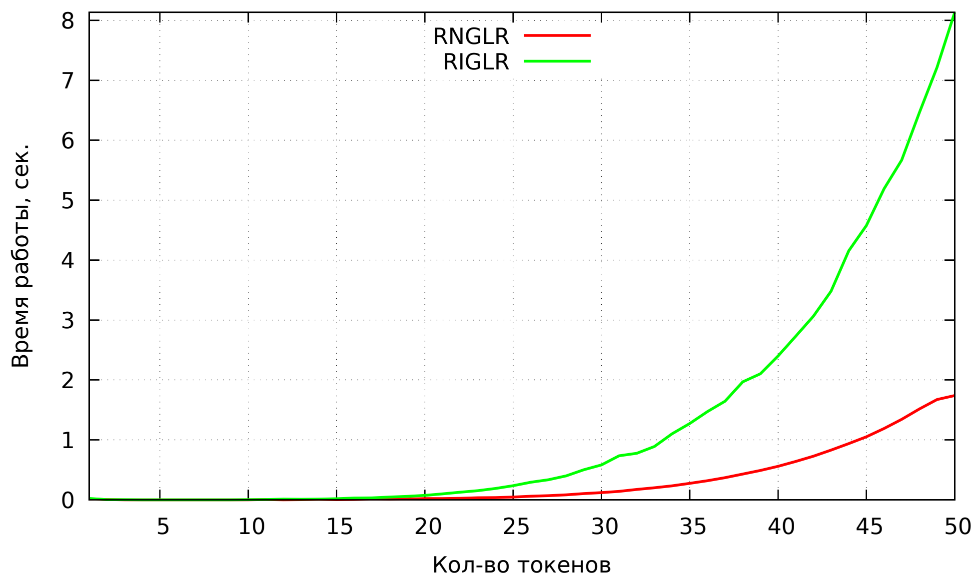

Рис. 5: Время работы алгоритмов

Заключение

В ходе данной работы получены следующие результаты.

- Изучен RIGLR-алгоритм синтаксического анализа.
- Реализован генератор синтаксических анализаторов, основанный на RIGLR-алгоритме.
- Проведено сравнение производительности анализаторов, созданных на основе RNGLR и RIGLR.

В дальнейшем планируется:

— увеличить производительность алгоритма, оптимизировав используемые структуры данных;

— реализовать версию RIGLR-алгоритма, способную производить разбор нелинейного входа, для использования в компоненте анализа динамически формируемого кода.

Исходный код данной работы можно найти в репозитории проекта YaccConstructor (https://github.com/YaccConstructor/YaccConstructor). Автор работал под учетной записью $Lares77$.

Список литературы

[1] Anselmo Marcella, Giammarresi Dora, Varricchio Stefano. Finite Automata and Non-self-embedding Grammars // Proceedings of the 7th International Conference on Implementation and Application of Automata. — CIAA'02. — Berlin, Heidelberg : Springer-Verlag, 2003. — P. 47–56. — URL: http://dl.acm.org/citation.cfm?id=1756384.1756390.

[2] Rekers Jan. Parser Generation for Interactive Environments. — 1992.

[3] Scott Elizabeth, Johnstone Adrian. Generalized Bottom Up Parsers With Reduced Stack Activity // Comput. J. — 2005. — sep. — Vol. 48, no. 5. — P. 565–587. — URL: http://dx.doi.org/10.1093/comjnl/bxh102.

[4] Scott Elizabeth, Johnstone Adrian. Right Nulled GLR Parsers // ACM Trans. Program. Lang. Syst. — 2006. — jul. — Vol. 28, no. 4. — P. 577–618. — URL: http://doi.acm.org/10.1145/1146809.1146810.

[5] Verbitskaia E., Grigorev S., Avdyukhin D. Relaxed Parsing of Regular Approximations of String-Embedded Languages // Preliminary Proceedings of the PSI 2015: 10th International Andrei Ershov Memorial Conference. — PSI'15. — 2015. — P. 1–12.

[6] YaccConstructor. — URL: http://yaccconstructor.github.io/YaccConstructor/ (online; accessed: 04.09.2016).