Приложения теории формальных языков и синтаксического анализа

Семён Григорьев

8 июля 2021 г.

Оглавление

Глава 1

Алгоритм на основе восходящего анализа

В данном разделе будут рассмотрены алгоритмы восходящего синтаксического анализа LR-семейсва, в том числе Generalized LR (GLR). Также будет рассмотрено обощение алгоритма GLR для решения задачи поиска путей с контекстно-свободными ограничениями в графах.

1.1 Восходящий синтаксический анализ

Существует большое семейство LR(k) алгоритмов — алгоритм восходящего синтаксического анализа. Основная идея, лежащая в основе семейства, заключается в следующем: входная последовательность символов считывается слева направо с попутным добавлением в стек и выполнением сворачивания на стеке — замены последовательности терминалов и нетерминалов, лежащих наверху стека, на нетерминал, если существует соответствующее правило в исходной грамматике.

Как и в случае с LL используется магазинный автомат, управляемый таблицами, построенными по грамматике. При этом, у LR анализатора есть два типа команд:

1. shift — прочитать следующий символ входной последовательности, положив его в стек, и перейти в следующее состояние;

2. reduce(k) — применить k-ое правило грамматики, правая часть которого уже лежит на стеке: снимаем со стека правую часть продукции и кладём левую часть.

А управляющая таблица выглядит следующим образом.

States	t_0	$\dots$	t_a	$\dots$	$\$$	N_0	$\dots$	N_b	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
10	$\dots$	$\dots$	s_i	$\dots$	r_k	$\dots$	$\dots$	j	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$	acc	$\dots$	$\dots$	$\dots$	$\dots$

Здесь

- s_i — shift: перенести соответствующи символ в стек и перейти в состояние i.
- r_k — reduce(k): в стеке накопилась правая часть продукции k, пора производить свёртку.
- j — goto: выполняется после reduce. Сама по себе команда reduce не переводит автомат в новое состояние. Команда goto переведёт автомат в состояние j.
- acc — accept: разбор завершился успешно.

Если ячейка пустая и в процессе работы мы пропали в неё — значит произошла ошибка. Для детерминированной работы анализатора требуется, чтобы в каждой ячейке было не более одной команды. Еcди это не так, то говорят о возникновении конфликтов.

- shift-reduce — ситуация, когда не понятно, читать ли следующий символ или выполнить reduce. Например, если правая часть одного из правил является префиксом правой части другого правила: $N \to w, M \to ww'$.
- reduce-reduce — ситуация, когда не понятно, к какому правилу нужно применить reduce. Например, если есть два правила с одинаковыми правыми частями: $N \to w, M \to w$.

Принцип работы LR анализаторов следующий. Пусть у нас есть входная строка, LR-автомат со стеком и управляющая таблица. В начальный момент на стеке лежит стартовое состояние LR-автомата, позиция во входной строке соответствует её началу. На каждом шаге анализируется текущий символ входа и текущее состояние, в котором находится автомат, и совершается одно из действий:

- Если в управляющей таблице нет инструкции для текущего состояния автомата и текущего символа на входе, то завершаем разбор с ошибкой.
- Иначе выполняем одну из инструкций:
  - в случае acc — успешно завершаем разбор.
  - в случае shift — кладем на стек текущий символ входа, сдвигая при этом текущую позицию, и номер нового состояния. Переходим в новое состояние.
  - в случае reduce(k) — снимаем со стека 2l элементов: l состояний и l терминалов/нетерминалов (где l — длина правой части k-ого правила), кладём на стек нетерминал левой части правила. Тперь на вершине стека у нас нетерминал N_a, а следующий элемент — состяние i. Если в ячейке (i, N_a) управляющей таблицы лежит состояние j, то кладём его на вершину стека. Иначе завершаемся с ошибкой.

Разные алгоритмы из LR-семейсва строят таблицы разными способами и, соответсвенно, могут избегать тех или иных конфликтов. Рассмотрим некорых представителей.

1.1.1 LR(0) алгоритм

Данный алгоритм самый “слабый” из семейства — разбирает наименьший класс языков. Для построения используются LR(0) пункты.

Определение 1.1.1. LR(0) пункт (LR(0) item) — правило грамматики, в правой части которого имеется точка, отделяющая уже разобранную часть правила (слева от точки) от того, что еще предстоит распознать (справа от точки): $A \to \alpha \cdot \beta$, где $A \to \alpha\beta$ — правило грамматики. □

Состояние LR(0) автомата — множество LR(0) пунктов. Для того чтобы изх построить используется операция *closure* или *замыкание*.

Определение 1.1.2. $closure(X) = closure(X \cup \{M \to \cdot\gamma \mid N_i \to \alpha \cdot M\beta \in X\})$ □

Определение 1.1.3. Ядро — исходное множество пунктов, до применения к нему замыкания. □

Для перемещения точки в пункте используется функция *goto*.

Определение 1.1.4. $goto(X, p) = \{N_j \to \alpha p \cdot \beta \mid N_j \to \alpha \cdot p\beta \in X\}$ □

Теперь мы можем построить LR(0) автомат. Первым шагом необходимо расширить грамматику: добавить к исходной граммтике правило вида $S' \to S\$$, где S — стартовый нетерминал исходной граммтики, S' — новый стартовый нетерминал (не использовался ранее в грамматике), \$ — маркер конца строки (не входил в терминальный алфавит исходной граммтики).

Далее строим автомат по следующим принципам.

- Состояния — множества пунктов.
- Переходы между состяниями осуществляются по символам грамматики.
- Начальное состояние — $closure(\{S' \to \cdot S\$\})$.
- Следующее состояние по текущему состоянию X и смволу p вычисляются как $closure(goto(X, p))$

Управляющая таблица по автомату строится следующим образом.

- acc в ячейку, соответствующую финальному состоянию и \$
- s_i в ячейку (j, t), если в автомате есть переход из состояния j по терминалу t в состояние i
- i в ячейку (j, N), если в автомате есть переход из состояния j по нетерминалу N в состояние i
- r_k в ячейку (j, t), если в состоянии j есть пункт $A \to \alpha\cdot$, где $A \to \alpha$ — k-ое правило грамматики, t — терминал грамматики

1.1.2 SLR(1) алгоритм

SLR(1) анализатор отличается от LR(0) анализатора построением таблицы по автомату (автомат в точности как у LR(0). А именно, r_k добавляется в ячейку (j, t), если в состоянии j есть пункт $A \to \alpha\cdot$, где $A \to \alpha$ — k-ое правило грамматики, $t \in FOLLOW(A)$

1.1.3 CLR(1) алгоритм

Canonical LR(1), он же LR(1). Данный алгоритм является дальнейшим расширением SLR(1): к пунктам добавляются множества предпросмотра (lookahead).

Определение 1.1.5. Множество предпросмотра для правила P — терминалы, которые должны встретиться в выведенной строке сразу после строки, выводимой из данного правила. □

Определение 1.1.6. CLR пункт: $[A \to \alpha \cdot \beta, \{t_0, \ldots, t_n\}]$, где $t_0, \ldots, t_n$ — множество пердпросмотра для правила $A \to \alpha\beta$. □

Определение 1.1.7. Пусть дана грамматика $G = \langle \Sigma, N, R, S \rangle$.

$$
\begin{aligned}
closure(X) = closure(X \cup \{[B \to \cdot\delta, \{FIRST(\beta t_0), \ldots, FIRST(\beta t_n)\}] \\
\mid B \to \beta \in R, [A \to \alpha \cdot B\beta, \{t_0, \ldots, t_n\}] \in closure(X)\})
\end{aligned}
$$

□

Функция *goto* определяется аналогично LR(0), автомат строится по тем же принципам.

При построении управляющей таблицы усиливается правило добавлеия команды *redice*. А именно, добавляем r_k в ячейку (j, t_i), если в состоянии j есть пункт $[A \to \alpha\cdot, \{t_0, \ldots, t_n\}]$, где $A \to \alpha$ — k-ое правило грамматики.

1.1.4 Примеры

Расмотрим построение автоматов и таблиц для различных модефикаций LR алгоритма.

Возьмем следующую грамматику:

$$
\begin{aligned}
&0) S \to aSbS \\
&1) S \to \varepsilon
\end{aligned}
$$

Расширим вышеупомянутую грамматику, добавив новый стартовый нетерминал S', и далее будем работать с этой расширенной грамматикой:

$$
\begin{aligned}
&0) S \to aSbS \\
&1) S \to \varepsilon \\
&2) S' \to S\$
\end{aligned}
$$

Пример 1.1.1. Пример ядра и замыкания.

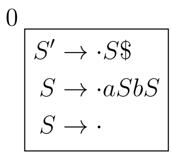
Возьмем правило 2 нашей грамматики, предположим, что мы только начинаем разбирать данное правило.

Ядром в таком случае является item исходного правила: $S' \to .S\$$

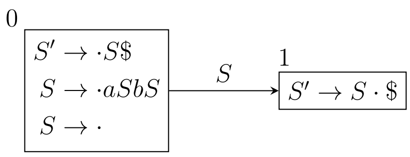
При замыкании добавятся ещё два item'а с правилами по выводу нетерминала 'S', поэтому получаем три item'а: $S' \to .S\$$, $S \to .aSbS$ и $S \to .\varepsilon$

Пример 1.1.2. Пример построения LR(0)-автомата для нашей грамматики с применением замыкания.

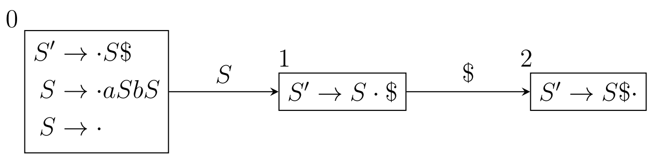
1. Добавляем стартовое состояние: item правила 0 и его замыкание (вместо item'а $S \to .\varepsilon$ будем писать $S \to .$).



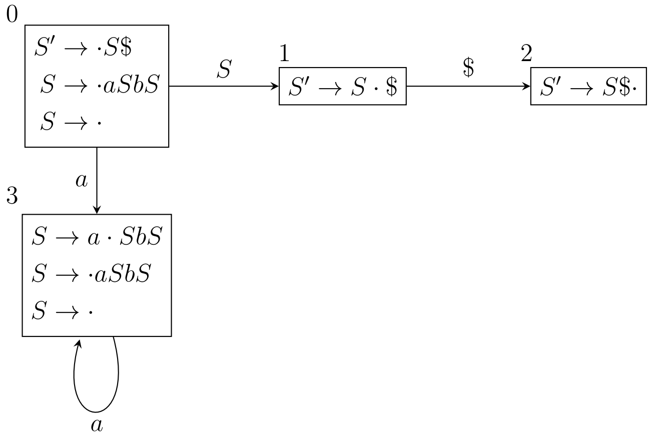
2. По 'S' добавляем переход из стартового состояния в новое состояние 1.



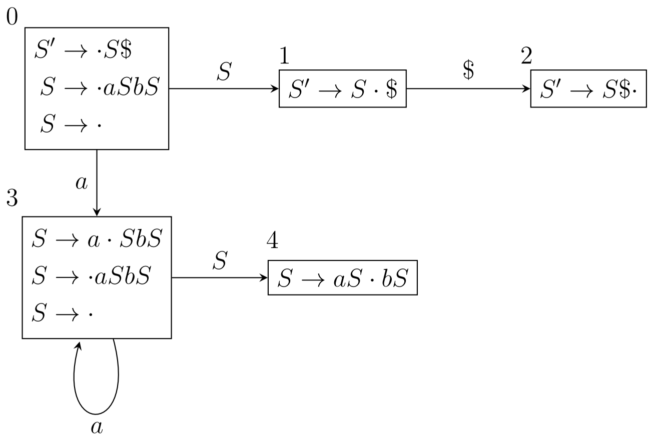
3. По '\$' добавляем переход из состояния 1 в новое состояние 2.



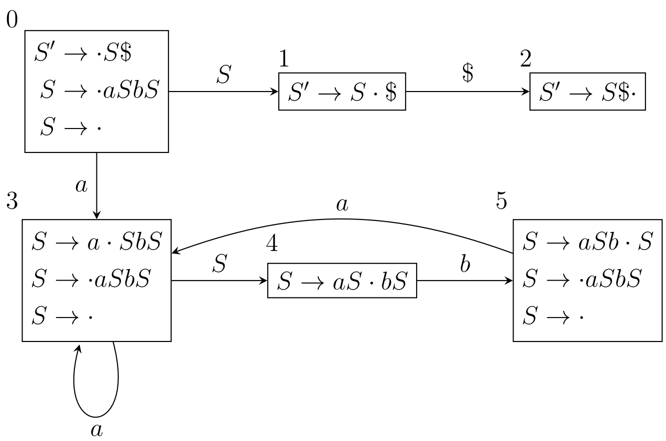
4. По 'a' добавляем переход из стартового состояния в новое состояние 3 и делаем его замыкание. Также добавляем переход по 'a' из этого состояния в себя же.



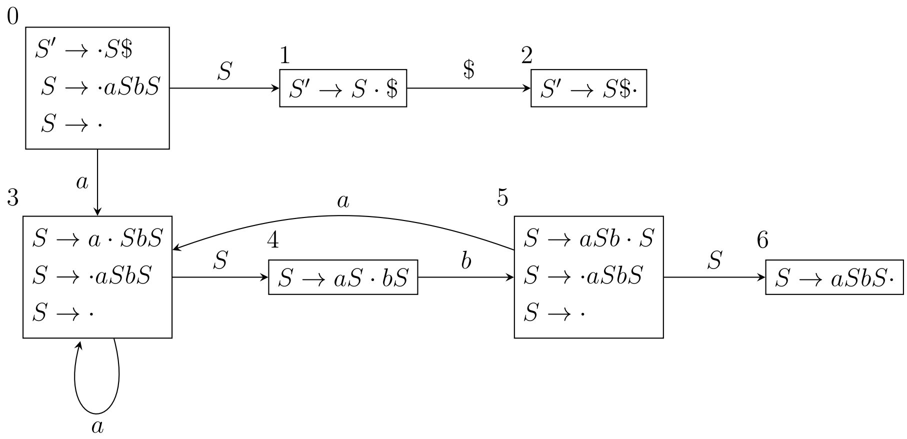
5. По 'S' добавляем переход из состояния 3 в новое состояние 4.



6. По 'b' добавляем переход из состояния 4 в новое состояние 5 и делаем его замыкание. Также добавляем переход по 'a' из этого состояния в состояние 3.



7. По 'S' добавляем переход из состояния 5 в новое состояние 6. Завершаем построение LR-автомата.



Далее будем использовать этот автомат для построения управляющей таблицы.

Пример 1.1.3. Пример управляющей LR(0) таблицы.

	a	b	$	S
0	s_3, r_1	r_1	r_1	1
1			acc	
2	r_2	r_2	r_2	
3	s_3, r_1	r_1	r_1	4
4		s_5		
5	s_3, r_1	r_1	r_1	6
6	r_0	r_0	r_0	

Как видим, в данном случае в таблице присутствуют shift-reduce конфликты. В случае, когда не удаётся построить таблицу без конфликтов, говорят, что грамматика не LR(0).

□

Пример 1.1.4. Пример управляющей LR(1) таблицы. Автомат тот же, однако команды *reduce* расставляются с использованием FOLLOW.

$$FOLLOW_1(S) = \{b, \$\}$$

	a	b	$	S
0	s_3	r_1	r_1	1
1			acc	
2				
3	s_3	r_1	r_1	4
4		s_5		
5	s_3	r_1	r_1	6
6		r_0	r_0	

В данном случае в таблице отсутствуют shift-reduce конфликты. То есть наша грамматика SLR(1), но не LR(0).

□

Пример 1.1.5. Пример LR-разбора входного слова abab$ из языка нашей грамматики с использованием построенных ранее LR-автомата и управляющей таблицы.

1. Начало разбора. На стеке — стартовое состояние 0.

Вход:	a	b	a	b	$
Стек:	0				

2. Выполняем shift 3: сдвигаем указатель на входе, кладем на стек ’a’, новое состояние 3 и переходим в него.

Вход:	a	b	a	b	$
Стек:	0	a	3		

3. Выполняем reduce 1 (кладем на стек ’S’), кладем новое состояние 4 и переходим в него.

Вход:	a	b	a	b	$	
Стек:	0	a	3	S	4	

4. Выполняем shift 5: сдвигаем указатель на входе, кладем на стек ’b’, новое состояние 5 и переходим в него.

Вход:	a	b	a	b	$

Стек:	0	a	3	S	4	b	5	

5. Выполняем shift 3.

Вход:	a	b	a	b	$					
Стек:	0	a	3	S	4	b	5	a	3	

6. Выполняем reduce 1, кладем новое состояние 4 и переходим в него.

Вход:	a	b	a	b	$							
Стек:	0	a	3	S	4	b	5	a	3	S	4	

7. Выполняем shift 5.

Вход:	a	b	a	b	$									
Стек:	0	a	3	S	4	b	5	a	3	S	4	b	5	

8. Выполняем reduce 1, кладем новое состояние 6 и переходим в него.

Вход:	a	b	a	b	$											
Стек:	0	a	3	S	4	b	5	a	3	S	4	b	5	S	6	

9. Выполняем reduce 0 (снимаем со стека 8 элементов и кладем ’S’), оказываемся в состоянии 5 и делаем переход в новое состояние 6 с добавлением его на стек.

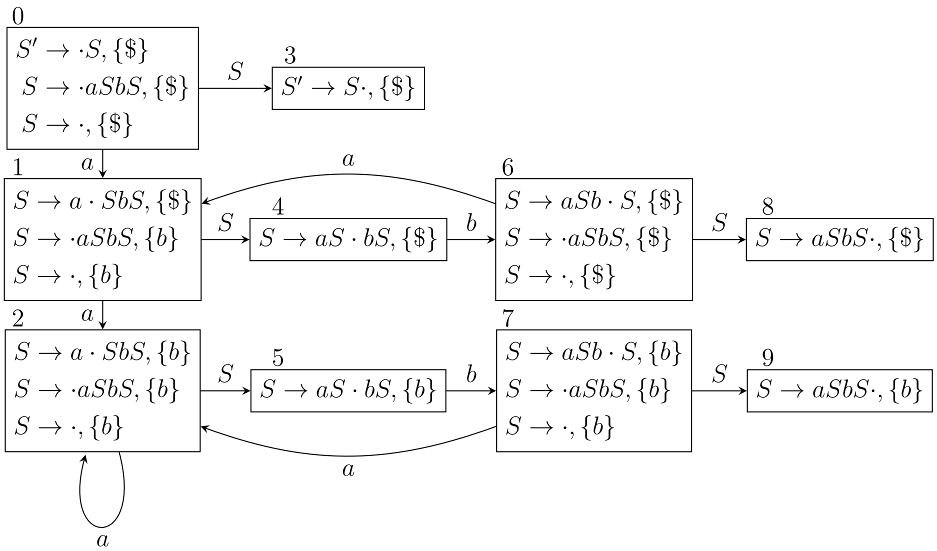
Вход:	a	b	a	b	$					
Стек:	0	a	3	S	4	b	5	S	6	

10. Снова выполняем reduce 0, оказываемся в состоянии 0 и делаем переход в новое состояние 1 с добавлением его на стек. Заканчиваем разбор.

Вход:	a	b	a	b	$
Стек:	0	S	1		

□

Пример 1.1.6. Пример CLR автомата.



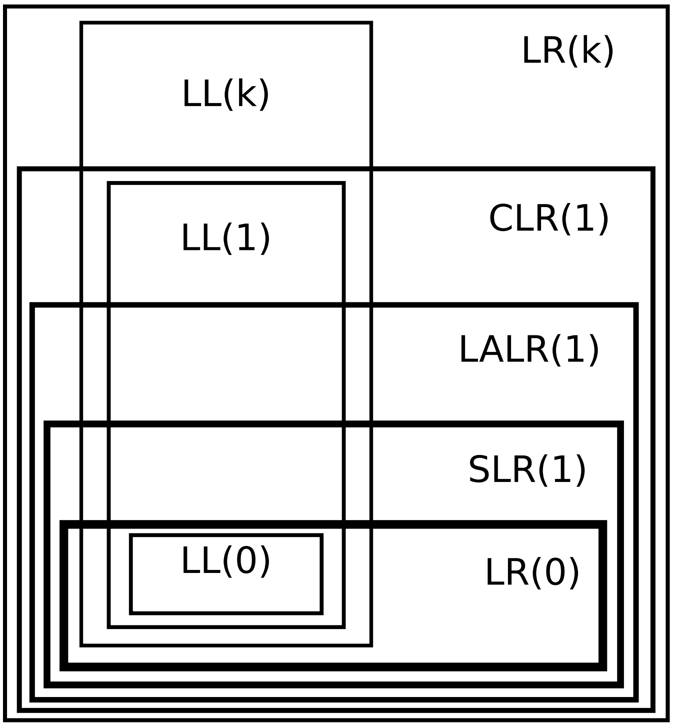
□

Существуют и другие модификации, например LALR(1)

На практике конфликты стараются решать ещё и на этапе генерации. Прикладные инструменты могут сгенерировать парсер по неоднозначной грамматике: из переноса или свёртки выбирать перенос, из нескольких свёрток — первую в каком-то порядке (обычно в порядке появления соответствующих продукций в грамматике).

1.1.5 Сравнение классов LL и LR

Иерархию языков, распознаваемых различными классами алгоритмов, можно представить следующим образом.



Из диаграммы видно, что класс языков, распознаваемых LL(k) алгоритмом уже, чем класс языков, распознаваемый LR(k) алгоритмом, при любом конечном k. Приведём несколько примеров.

1. $L = \{a^m b^n c \mid m \geq n \geq 0\}$ является LR(0), но для него не существует LL(1) граммтики.

2. $L = \{a^n b^n + a^n c^n \mid n > 0\}$ является LR, но не LL.

3. Больше примеров можно найти в работе Джона Битти [1].

1.2 GLR и его применение для КС запросов

Алгоритм LR довольно эффективен, однако позволяет работать не со всеми КС-грамматиками, а только с их подмножеством LR(k). Если грамматика находится за рамками допускаемого класса, некоторые ячейки управляющей таблицы могут содержать несколько значений. В этом случае грамматика отвергалась анализатором.

Чтобы допустить множественные значения в ячейках управляющей таблицы, потребуется некоторый вид недетерминизма, который даст возможность анализатору обрабатывать несколько возможных вариантов синтаксического разбора параллельно. Именно это и предлагает анализатор Generalized LR (GLR) [4]. Далее мы рассмотрим общий принцип работы, проиллюстрируем его с помощью примера, а также рассмотрим модификации GLR.

1.2.1 Классический GLR алгоритм

Впервые GLR парсер был представлен Масару Томитой в 1987 [4]. В целом, алгоритм работы идентичен LR той разницей, что управляющая таблица модифицирована таким образом, чтобы допускать множественные значения в ячейках. Интерпретатор автомата изменён соответствующим образом.

Для того, чтобы избежать дублирования информации при обработке неоднозначностей, стоит использовать более сложную структуру стека: *граф-структурированный стек* или (*GSS*, Graph Structured Stack). Это направленный граф, в котором вершины соответствуют элементам стека, а ребра их соединяют по правилам управляющей таблицы. У каждой вершины может быть несколько входящих и исходящих дуг: таким образом реализуется то объединение одинаковых состяний и ветвление в случае неоднозначности.

Пример 1.2.1. Рассмотрим пример GLR разбора с использованием GSS.

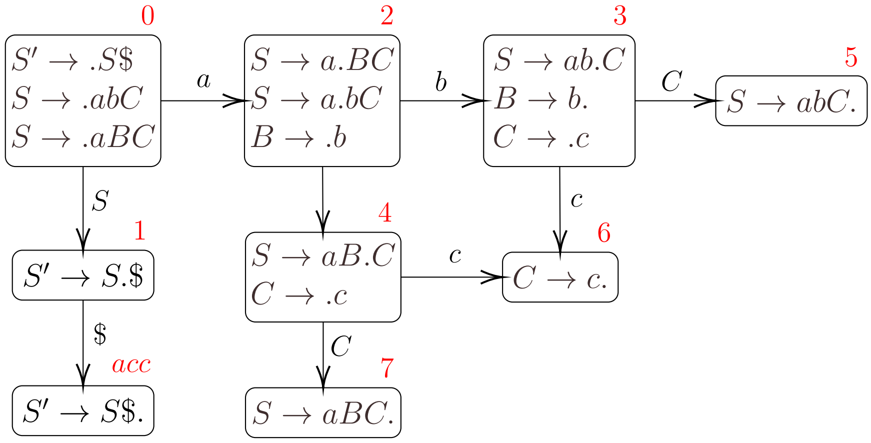
Возьмем грамматику G следующего вида:

$$\begin{array}{ll} 0. & S' \to S\$ \\ 1. & S \to abC \\ 2. & S \to aBC \\ 3. & B \to b \\ 4. & C \to c \end{array}$$

Входное слово w:

$$w = abc\$$$

Построим для данной грамматики LR автомат:

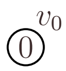


И управляющую таблицу:

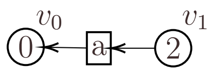
	a	b	c	$	B	C	S
0	s_2					1	
1				acc			
2		s_3			4		
3			s_6, r_3			5	
4			s_6			7	
5				r_1			
6				r_4			
7				r_2			

Разберем слово w с помощью алгоритма GLR. Использована следующая аннотация: вершины-состояния обозначены кругами, вершины-символы — прямоугольниками.

1. Инициализируем GSS стартовым состоянием v_0:

   Вход: | a | b | c | $ | GSS: 

2. Видим входной символ 'a', ищем соответствующую ему операцию в управляющей таблице — $shift$ 2, строим новый узел v_1:

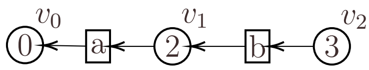
   Вход: | a | b | c | $ | GSS: 

3. Повторяем для символа 'b', операции $shift$ 3 и узла v_2:

Вход:

a	b	c	$

GSS:

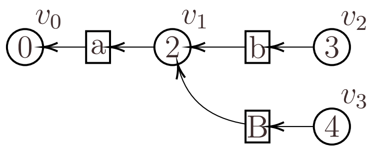


4. При обработке узла v_3 у нас возникает конфликт shift-reduce: $s_6,\ r_3$. Мы смотрим на вершины, смежные v_2, на управляющую таблицу и на правило вывода под номером 3 для поиска альтернативного построения стека. Находим *goto* 4 и строим вершину v_3 с соответствующим переходом по нетерминалу B из v_1 (т.к. количество символов в правой части правила вывода 3 равняется 1, значит мы в дереве опустимся на глубину 1 по вершинам-состояниям):

Вход:

a	b	c	$

GSS:

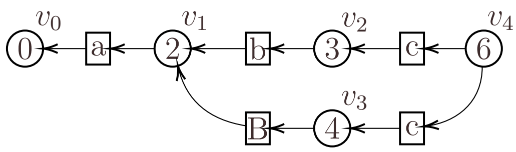


5. Читаем символ 'c' и ищем в управляющей таблице переходы из состояний 3 и 4 (так как узлы v_2 и v_3 находятся на одном уровне, то есть были построены после чтения одного символа из входного слова). Таким переходом оказывается s_6 в обоих случаях, поэтому соединяем узел v_4 с обоими рассмотренными узлами:

Вход:

a	b	c	$

GSS:

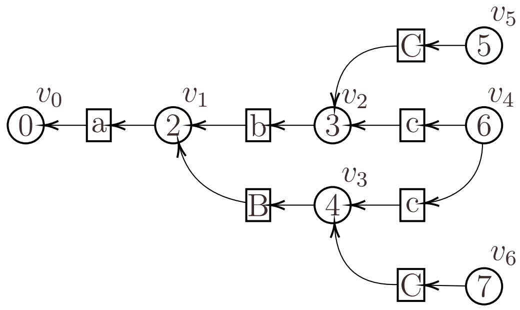


6. При обработке узла v_4 находим соответствующею 6-ому состоянию редукцию по правилу 4. Его правая часть содержит один символ 'c', 2 вершины-символа с которым достижимы из v_4. Находим вершины-состояния, которые смежны с этими вершинами-символами и обрабатываем переходы по левой части правила 4. Такими переходами по нетерминалу C оказываются 5 и 7. Строим соответствующие им вершины v_5 и v_6:

Вход:

a	b	c	$

GSS:

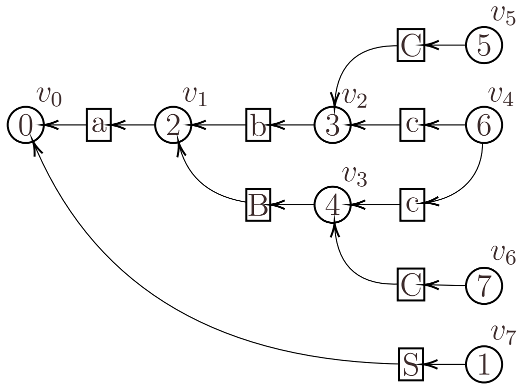


7. При обработке узлов v_5 и v_6 находим редукции с символом 'S' в левой части и тремя символами в правой. Возвращаемся на 3 вершины-состояния назад и строим вершину v_7 с переходом по S:

Вход:

a	b	c	$

GSS:

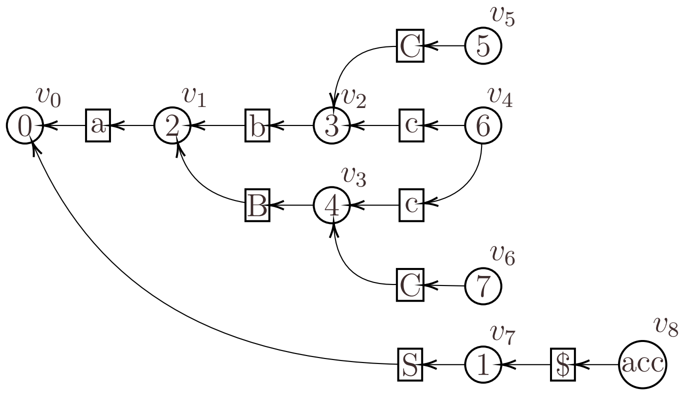


8. Наконец, обрабатывая вершину v_7, читаем символ '$' и строим узел v_8, который соответствует допускающим состоянием:

Вход:

a	b	c	$

GSS:



1.2.2 Модификации GLR

Алгоритм, представленный Томитой имел большой недостаток: он корректно работал не со всеми КС грамматиками, хоть и расширял класс допустимых LR анализаторами. Объем потребляемой памяти классическим GLR можно оценить как $O(n^3)$ с учетом оптимизаций, о которых говорилось ранее.

Спустя некоторое время после публикации Томита-парсера, Элизабет Скотт и Эндриан Джонстоун представили *RNGLR* (Right Nulled GLR) [2] — модифицированная версия GLR, которая решала проблему скрытых рекурсий. Это позволило расширить класс допускаемых грамматик до КС. Однако объем потребляемой памяти можно оценить сверху уже полиномом $O(n^{k+1})$, где k — длина самого длинного правила грамматики, что несколько ухудшило оценку классического GLR.

С этой проблемой справился BRNGLR (Binary RNGLR) [3]. За счет бинаризации удалось получить кубическую оценку сложности и при этом также, как и RNGLR, допускать все КС грамматики.

Кроме того, GLR довольно естесственно обобщается до решения задачи поиска путей с КС ограничениями. Это происходит следующим образом: элементами во входной структуре теперь будем считать не позиции символа в слове, а вершины графа (то есть “позици” и множество смежных вершин). Это приводит к тому, что при применении операции shift, следующих символов может быть несколько и каждый из них должен быть рассмотрен отдельно, сдвигаясь по соответствующему ребру и проходя входной граф в ширину. Подробное описание алгоритма и псевдокод представлены в работе [5].

Литература

[1] J. C. Beatty. Two iteration theorems for the ll(k) languages. *Theoretical Computer Science*, 12(2):193 – 228, 1980.

[2] E. Scott and A. Johnstone. Right nulled glr parsers. *ACM Trans. Program. Lang. Syst.*, 28(4):577–618, July 2006.

[3] E. Scott, A. Johnstone, and R. Economopoulos. Brnglr: A cubic tomita-style glr parsing algorithm. *Acta Inf.*, 44(6):427–461, Sept. 2007.

[4] M. Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13:31–46, 1987.

[5] E. Verbitskaia, S. Grigorev, and D. Avdyukhin. Relaxed parsing of regular approximations of string-embedded languages. In M. Mazzara and A. Voronkov, editors, *Perspectives of System Informatics*, pages 291–302, Cham, 2016. Springer International Publishing.