

# Нисходящие распознаватели КС-языков без возвратов

- 1 Метод рекурсивного спуска
- 2 LL(k) - грамматики

# Метод рекурсивного спуска

Для улучшения алгоритма с подбором альтернатив для нисходящего разбора в первую очередь необходимо на каждом шаге алгоритма обеспечить однозначный выбор из возможных альтернатив. В таком случае алгоритм не будет требовать возврата на предыдущие шаги и будет обладать линейными характеристиками.

Для этих целей в методе рекурсивного спуска предлагается ориентироваться на терминальный символ входной цепочки. Если текущий символ входной цепочки равен  $a$  и текущий выбор необходимо делать для нетерминального символа  $A$ , то из всех правил вида  $A \rightarrow \gamma$  ищется такое, в котором в цепочке  $\gamma$  на первом месте стоит терминальный символ  $a$ . Если такое правило найдено, то считывающая головка передвигается на следующий входной символ, а алгоритм запускается рекурсивно для каждого нетерминального символа в цепочке  $\gamma$ . Если для нетерминального символа  $A$  существует только одно правило  $A \rightarrow \gamma$ , где  $\gamma$  не начинается с терминального символа, то алгоритм аналогично запускается рекурсивно для каждого нетерминального символа в цепочке  $\gamma$ . В любом другом противном случае входная цепочка не принимается.

Начальный разбор идет с символа  $S$ .

Условия применимости алгоритма рекурсивного спуска можно получить из его описания. Ясно, что для применимости необходимо, чтобы все правила грамматики удовлетворяли двум условиям:

- 1 Если для нетерминального символа  $A$  существует правило вида  $A \rightarrow \gamma$ , где  $\gamma$  начинается с нетерминального символа и состоит как из терминальных, так и из нетерминальных символов, то это должно быть единственное правило для  $A$ .
- 2 Во всех остальных случаях правила для нетерминального символа  $A$  должны иметь вид  $A \rightarrow a\gamma$ , где  $a$  - терминальный символ,  $\gamma$  состоит как из терминальных, так и из нетерминальных символов.

Данные условия являются достаточными, но не необходимыми. Может возникнуть ситуация, когда грамматика не удовлетворяет этим требованиям, но ее можно преобразовать к такому эквивалентному виду, чтобы эти требования выполнялись. Однако, не существует алгоритма, который бы позволил преобразовать произвольную КС-грамматику к указанному выше виду, равно как не существует и алгоритма, который бы позволял проверить, возможны ли такого рода преобразования. Все это ограничивает область применения данного алгоритма.

## Пример использования метода рекурсивного спуска

Рассмотрим грамматику:

$$S \rightarrow aA|bB$$

$$A \rightarrow a|bA|cC$$

$$B \rightarrow b|aB|cC$$

$$C \rightarrow AaBb$$

Требуется проверить выводимость цепочки:

$$S \Rightarrow abcbaacsaabbb$$

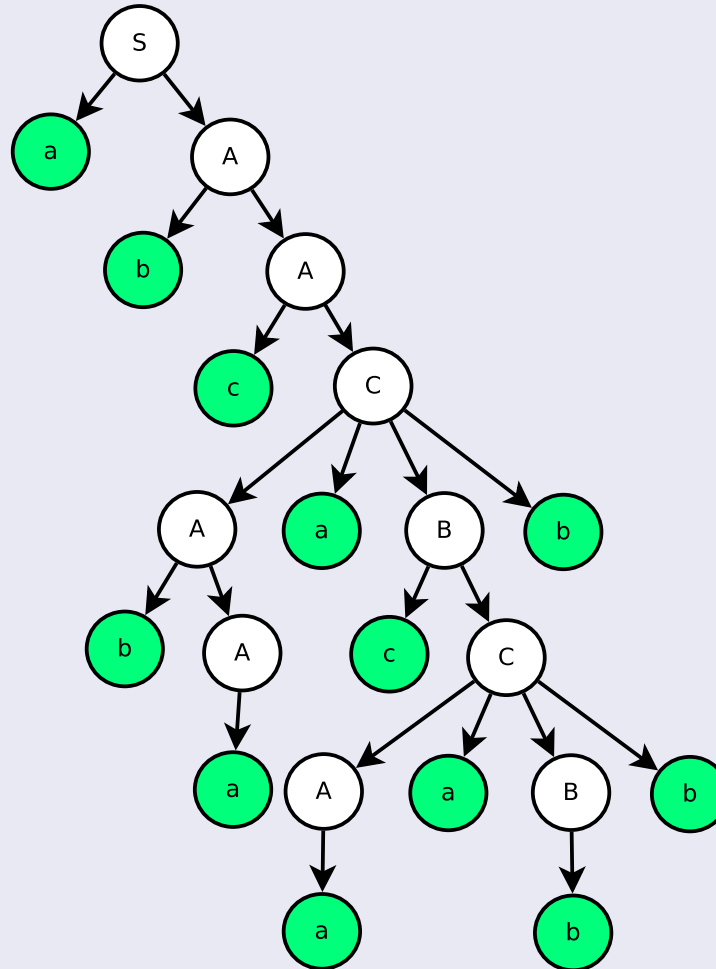
Заметим, что все правила удовлетворяют требованиям метода рекурсивного спуска: для символов  $S, A, B$  правая часть любого их правила начинается с терминального символа, у нетерминального символа  $C$  правая часть не начинается с терминального символа, но оно единственное для символа  $C$ .

## Пример использования метода рекурсивного спуска

Остаток входной цепочки	Текущая цепочка вывода	Полученная цепочка вывода
abcbaacaabbbb	S	aA
bcbaacaabbbb	A	bA
cbaacaabbbb	A	cC
baacaabbbb	C	AaBb
baacaabbbb	AaBb	bAaBb
aacaabbbb	AaBb	aaBb
caabbbb	Bb	cCb
aabbbb	Cb	AaBbb
aabbbb	AaBbb	aaBbb
bbb	Bbb	bbb
ε	ε	ε

## Пример использования метода рекурсивного спуска

На рисунке представлено полученное дерево вывода.



# Метод рекурсивного спуска

Можно рекомендовать ряд преобразований, которые способствуют приведению грамматики к виду, требуемому для применения алгоритма рекурсивного спуска. Эти преобразования заключаются в следующем:

- исключение  $\epsilon$ -правил;
- исключение левой рекурсии;
- левая факторизация, которая позволяет исключить для каждого нетерминального символа правила, начинающиеся с одних и тех же терминальных символов: если правило имеет вид  $A \rightarrow a\alpha_1 | a\alpha_2 | \dots | a\alpha_n | b_1\beta_1 | b_2\beta_2 | \dots | b_m\beta_m$  и ни одна цепочка  $\beta_i$  не начинается с символа  $a$ , то вводим новый нетерминальный символ  $A'$  и два правила:

$$A \rightarrow aA' | b_1\beta_1 | b_2\beta_2 | \dots | b_m\beta_m, A' \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$$

- замена нетерминальных символов в правилах на цепочки их выводов. Например, если имеются правила:

$$A \rightarrow B_1 | B_2 | \dots | B_n | b_1\beta_1 | b_2\beta_2 | \dots | b_m\beta_m,$$

$$B_1 \rightarrow \alpha_{11} | \alpha_{12} | \dots | \alpha_{1k}$$

...

$$B_n \rightarrow \alpha_{n1} | \alpha_{n2} | \dots | \alpha_{np}$$

то заменяем их на одно правило:

$$A \rightarrow \alpha_{11} | \alpha_{12} | \dots | \alpha_{1k} | \dots | \alpha_{n1} | \alpha_{n2} | \dots | \alpha_{np} | b_1\beta_1 | b_2\beta_2 | \dots | b_m\beta_m,$$



# Задание 1

В грамматике, заданной правилами:

$$S \rightarrow aABb|bBAa|cCc$$

$$A \rightarrow aA|bB|cC$$

$$B \rightarrow b|aAC$$

$$C \rightarrow aA|bA|cC$$

Выполните вывод цепочки  $aababbbsabbb$  методом рекурсивного спуска

## Задание 2

Реализуйте алгоритм рекурсивного спуска на языке программирования Python.

Логическим продолжением идеи, положенной в основу метода рекурсивного спуска, является предложение использовать для выбора одной из множества альтернатив не один, а несколько символов входной цепочки. Однако напрямую переложить алгоритм выбора альтернативы, как для одного символа, не удастся, поскольку два соседних символа в цепочке на самом деле могут быть выведены с использованием различных правил грамматики, поэтому неверным будет напрямую искать их в одном правиле.

Тем не менее существует класс грамматик, в котором осуществляется выбор одной альтернативы из множества возможных именно на основе нескольких начальных символов из входной цепочки. Этот класс грамматик имеет название  $LL(k)$ -грамматики. Первая буква L обозначает, что входная цепочка читается слева направо, вторая буква L обозначает обстоятельство, что осуществляется левосторонний вывод,  $k$  обозначает количество символов, просматриваемых во входной строке. Обычно ведут речь о грамматиках с конкретным значением  $k$ , так существуют  $LL(1)$  - грамматики,  $LL(2)$  - грамматики и т.д.

# LL( $k$ ) - грамматики

Грамматика обладает свойством  $LL(k)$ ,  $k > 0$ , если на каждом шаге вывода для однозначного выбора очередной альтернативы достаточно знать символ на верхушке стека и рассмотреть первые  $k$  символов от текущего положения считывающей головки во входной строке.

Для  $LL(k)$  грамматик существует алгоритм, позволяющий проверить, является ли заданная грамматика  $LL(k)$ -грамматикой для строго определенного числа  $k$ .

Кроме того, известно, что все грамматики, допускающие разбор по методу рекурсивного спуска, являются подклассом  $LL(1)$ -грамматик.

Есть, однако, неразрешимые проблемы для  $LL(k)$ -грамматик:

- не существует алгоритма, который бы мог проверить, является ли заданная КС-грамматика  $LL(k)$ -грамматикой для некоторого произвольного  $k$ ;
- не существует алгоритма, который бы мог преобразовать произвольную КС-грамматику к виду  $LL(k)$ -грамматики для некоторого  $k$  (или доказать, что преобразование невозможно).

Для  $LL(k)$ -грамматики при  $k > 1$  совсем не обязательно, чтобы все правые части правил грамматики для каждого нетерминального символа начинались с  $k$  различных терминальных символов. Принципы распознавания предложений входного языка такой грамматики накладывают менее жесткие ограничения на правила грамматики, поскольку  $k$  соседних символов, по которым однозначно выбирается очередная альтернатива, могут встречаться в нескольких правилах грамматики. Поскольку все  $LL(k)$ -грамматики используют левосторонний нисходящий распознаватель, основанный на алгоритме с подбором альтернатив, очевидно, что они не могут допускать левую рекурсию.

Класс  $LL(k)$ -грамматик широк, но все же он недостаточен для того, чтобы покрыть все возможные синтаксические конструкции в языках программирования. Известно, что существуют детерминированные КС-языки, которые не могут быть заданы  $LL(k)$ -грамматикой ни для каких  $k$ . Однако  $LL(k)$ -грамматики удобны для использования, поскольку позволяют построить линейные распознаватели. Для построения распознавателей языков, заданных  $LL(k)$ -грамматиками, используются два множества, определяемые следующим образом:

- $FIRST(k, \alpha)$  - множество терминальных цепочек, укороченных до  $k$  символов выводимых из  $\alpha$  - строки из терминальных и нетерминальных символов;
- $FOLLOW(k, A)$  - множество укороченных до  $k$  символов терминальных цепочек, которые могут следовать непосредственно за нетерминальным символом  $A$  в цепочках вывода.

В дальнейшем мы будем рассматривать только  $LL(1)$ -грамматики. Очевидно, для каждого нетерминального символа  $LL(1)$ -грамматики не может быть двух правил, начинающихся с одного и того же терминального символа. Однако это менее жесткое условие, чем то, которое накладывает распознаватель по методу рекурсивного спуска, поскольку, в принципе,  $LL(1)$ -грамматика допускает в правой части правил цепочки, начинающиеся с нетерминальных символов, а также  $\epsilon$ -правила.  $LL(1)$ -грамматики позволяют построить достаточно простой и эффективный распознаватель.

Для  $LL(1)$ -грамматик алгоритм работы распознавателя на шаге выбора альтернативы заключается в проверке двух условий (предполагается, что  $a$  - очередной символ, обозреваемый считывающей головкой,  $A$  - нетерминал на верхушке стека:

- 1 необходимо выбрать в качестве альтернативы правило  $A \rightarrow \alpha$ , если  $a \in FIRST(1, \alpha)$
- 2 необходимо выбрать в качестве альтернативы правило  $A \rightarrow \epsilon$ , если  $a \in FOLLOW(1, A)$

Если ни одно из этих условий не выполняется, то цепочка не принадлежит заданному языку, и алгоритм должен сигнализировать об ошибке.

Чтобы проверить, что заданная грамматика является  $LL(1)$ -грамматикой, необходимо и достаточно проверить для каждого нетерминального символа  $A$ , для которого существует более одного правила вида  $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$ , выполнение требования:

$$FIRST(1, \alpha_i FOLLOW(1, A)) \cap FIRST(1, \alpha_j FOLLOW(1, A)) = \emptyset, \forall i \neq j$$

Иными словами, если для нетерминального символа  $A$  отсутствует правило  $A \rightarrow \epsilon$ , то все множества  $FIRST(1, \alpha_1), FIRST(1, \alpha_2), \dots, FIRST(1, \alpha_n)$  должны попарно не пересекаться, если же присутствует правило  $A \rightarrow \epsilon$ , то они не должны также пересекаться с множеством  $FOLLOW(1, A)$ .

# Алгоритм построения $FIRST(1, A)$

Для  $FIRST(1, \alpha)$ , если цепочка  $\alpha$  начинается с терминального символа  $b$ , то  $FIRST(1, \alpha) = \{b\}$ , если же она начинается с нетерминального символа  $B$ , то  $FIRST(1, \alpha) = FIRST(1, B)$ . Рассмотрим более детальный алгоритм, который строит множества  $FIRST(1, \alpha)$  сразу для всех нетерминальных символов грамматики. Алгоритм работает для грамматик, не содержащих  $\epsilon$ -правил (в противном случае ее нужно преобразовать к такому виду).

Алгоритм состоит из следующих шагов:

- 1 Для всех нетерминальных символов  $A$  формируем  $FIRST_0(1, A) = \{X | A \rightarrow X\alpha\}$  - включает все символы  $X$ , стоящие в начале правых частей правил для символа  $A$ .
- 2 Для всех нетерминальных символов  $A$  формируем  $FIRST_{i+1}(1, A) = FIRST_i(1, A) \cup FIRST_i(1, B)$ , где  $B$  - все нетерминальные символы множества  $FIRST_i(1, A)$
- 3 Если существует нетерминальный символ  $A$ , для которого  $FIRST_{i+1}(1, A) \neq FIRST_i(1, A)$  возвратиться к шагу 2, иначе перейти на шаг 4
- 4 Исключаем из полученных множеств  $FIRST_i(1, A)$  все нетерминальные символы.

# Алгоритм построения $FIRST(1, A)$

## Пример

Применим алгоритм построения  $FIRST(1, A)$  для грамматики:

$$S \rightarrow T|TR$$

$$R \rightarrow +T| - T| + TR| - TR$$

$$T \rightarrow E|EF$$

$$F \rightarrow *E|/E|*EF|/EF$$

$$E \rightarrow (S)|a|b$$

Шаг 1:

$$FIRST_0(1, S) = \{T\}, FIRST_0(1, R) = \{+, -\}, FIRST_0(1, T) = \{E\}$$

$$FIRST_0(1, F) = \{*, /\}, FIRST_0(1, E) = \{(\, a, b\}$$

Шаг 2:

$$FIRST_1(1, S) = \{T, E\}, FIRST_1(1, R) = \{+, -\}, FIRST_1(1, T) = \{E, (\, a, b\}$$

$$FIRST_1(1, F) = \{*, /\}, FIRST_1(1, E) = \{(\, a, b\}$$

Шаг 3: возвращаемся к шагу 2.



# Алгоритм построения $FIRST(1, A)$

## Пример

Шаг 2:

$$FIRST_2(1, S) = \{T, E, (, a, b\}, FIRST_2(1, R) = \{+, -\}, FIRST_2(1, T) = \{E, (, a, b\}$$
$$FIRST(1, F) = \{*, /\}, FIRST_2(1, E) = \{(, a, b\}$$

Шаг 3: возвращаемся к шагу 2.

Шаг 2:

$$FIRST_2(1, S) = \{T, E, (, a, b\}, FIRST_2(1, R) = \{+, -\}, FIRST_2(1, T) = \{E, (, a, b\}$$
$$FIRST_2(1, F) = \{*, /\}, FIRST_2(1, E) = \{(, a, b\}$$

Шаг 3: переходим к шагу 4.

Шаг 4:

$$FIRST(1, S) = \{(, a, b\}, FIRST(1, R) = \{+, -\}, FIRST(1, T) = \{(, a, b\}$$
$$FIRST(1, F) = \{*, /\}, FIRST(1, E) = \{(, a, b\}$$

Построение закончено.

# Алгоритм построения $FOLLOW(1, A)$

## Пример

Алгоритм состоит из следующих шагов:

- 1 Для всех нетерминальных символов  $A$  формируем  $FOLLOW_0(1, A) = \{X | \exists B \rightarrow \alpha AX\beta\}$  - вносим все символы, которые в правой части всех правил непосредственно следуют за  $A$ .
- 2  $FOLLOW_0(1, S) = FOLLOW_0(1, S) \cup \{\epsilon\}$  - вносим пустую цепочку в множество последующих символов для целевого символа  $S$  - это означает, что в конце разбора за целевым символом цепочка кончается.
- 3 Для всех нетерминальных символов  $A$  определяем  $FOLLOW'_i(1, A) = FOLLOW_i(1, A) \cup FIRST(1, B)$  для всех нетерминальных символов  $B$ , входящих в  $FOLLOW_i(1, A)$ .
- 4 Для всех нетерминальных символов  $A$  определяем  $FOLLOW''_i(1, A) = FOLLOW'_i(1, A) \cup FOLLOW'_i(1, B)$  для всех нетерминальных символов  $B$ , входящих в  $FOLLOW'_i(1, A)$ , если существует правило  $B \rightarrow \epsilon$ .
- 5 Для всех нетерминальных символов  $A$ :  $FOLLOW_{i+1}(1, A) = FOLLOW''_i(1, A) \cup FOLLOW''_i(1, B)$  для всех нетерминальных символов  $B$ , если существует правило  $B \rightarrow \alpha A$ .
- 6 Если  $FOLLOW_{i+1}(1, A) = FOLLOW_i(1, A)$ , то перейти к шагу 7, иначе к шагу 3.
- 7 Из всех построенных  $FOLLOW_i(1, A)$  исключаем все нетерминальные символы.