МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Н.Н. Ефанов, А. И. Отращенко

НИСХОДЯЩИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ

Допущено
Учебно-методическим объединением
высших учебных заведений Российской Федерации
по образованию в области прикладных математики и физики
в качестве учебного пособия для студентов вузов
по направлению подготовки «Прикладные математика и физика»

МОСКВА
МФТИ
2024

УДК 519.22
З56

Рецензенты:
Институт теории прогноза землетрясений и математической геофизики РАН
Доктор физико-математических наук *М. В. Родкин*
Институт прикладной математики им. М. В. Келдыша РАН
Доктор физико-математических наук *В. Т. Жуков*

Ефанов, Н. Н., Отращенко, А. И.
Нисходящий синтаксический анализ: учебно-методическое пособие – М.: МФТИ, 2024. – 94 с.

В пособии рассматриваются теоретические и алгоритмические основы, а также практические аспекты реализации безоткатных нисходящих синтаксических анализаторов строк как на основе построения управляющих таблиц, так и методом рекурсивного спуска. Предназначено для студентов старших курсов и аспирантов, специализирующихся в области компьютерных наук.

УДК 519.22

ISBN 978-5-7417-0606-0

2024

Оглавление

Глава 1. **Алгоритмы нисходящего синтаксического анализа**

1.1. Нисходящий синтаксический анализ

Нисходящий синтаксический анализ идеологически можно рассматривать как задачу поиска левого порождения входной строки, либо, что эквивалентно, как процесс построения дерева разбора добавлением узлов в прямом порядке обхода в глубину, начиная с корня.

Для контекстно-свободной (КС) грамматики $G = \langle \Sigma, N, S, P \rangle$ организация нисходящего анализа выглядит следующим образом: анализатор, разбирающий входную строку w, заканчивающуюся символом конца строки \$, в каждый момент работы содержит в своей памяти пару (α, v), где v – ещё не прочитанная часть входной строки. Алгоритм анализа пытается разобрать v как конкатенацию $\alpha = X_1 \dots X_l$,где $l \geq 0, X_1, \dots, X_l \in \Sigma \cup N$ — последовательность символов, хранящаяся на стеке так, что X_1 лежит на вершине. На каждом шаге ключевым действием является определение правила, применяемого для раскрытия соответствующего нетерминала. Когда правило выбрано, следует произвести проверку соответствия входной строки и терминальных символов правой части правила, и выполнить дальнейшие шаги для её нетерминальных символов. Если в конце разбора $v = \$$, то есть удалось дойти до конца строки, и при этом все нетерминалы удалось раскрыть, — строка успешно разобрана.

В данной главе мы рассмотрим два подхода к нисходящему анализу: LL-анализ, использующий входной буфер, стек для хранения промежуточных данных, и управляющую таблицу, хранящую правила, применяемые в ходе анализа, а также анализ методом рекурсивного спуска, использующий в качестве стека стек вызовов программных процедур, реализующих применение правил в соответствующих ситуациях.

Дадим необходимые определения.

▲ **Определение 1.** Говорят, что грамматика содержит *левую рекурсию*, если в ней существует вывод вида $A \vdash^* A\alpha$. Если при этом в грамматике содержится правило $A \vdash A\alpha$, левая рекурсия называется непосредственной, или явной. В противном случае левая рекурсия называется косвенной, или неявной.

▲ **Определение 2.** Грамматика называется *однозначной*, если у каждого слова имеется не более одного дерева разбора в этой грамматике.

▲ **Определение 3.** *Левым порождением*, или левосторонним выводом слова ω называется такой вывод ω, в котором каждая последующая строка получена из предыдущей заменой самого левого встречающегося в строке нетерминала по одному из правил. Символически, левое порождение обозначается как $\vdash^*_{lm}$, а любой его шаг — как $\vdash_{lm}$.

Лемма 1 *Пусть $G = \langle \Sigma, N, S, P \rangle$ — КС-грамматика. Предположим, что существует дерево разбора с корнем, отмеченным A, и кроной $\omega \in \Sigma^*$. Тогда в грамматике G существует левое порождение $A \vdash^*_{lm} \omega$.*

Доказательство производится индукцией по высоте дерева (рекомендуем читателям проделать его самим).

Лемма 2 *Для каждой грамматики $G = \langle \Sigma, N, S, P \rangle$ и $\omega \in \Sigma^*$, цепочка ω имеет два разных дерева разбора тогда и только тогда, когда ω имеет два разных левых порождения из P.*

Для описания построения нисходящего анализатора введем два вспомогательных множества, содержащих, соответсвенно, все возможные первые и непосредственно следующие k символов в результирующем выводе.

▲ **Определение 4.** Пусть $G = \langle N, \Sigma, P, S \rangle$ — КС-грамматика. Множество $FIRST_k$ определяется для сентенциальной формы α как:
$FIRST_k(\alpha) = \{\omega \in \Sigma^* \mid \alpha \rightarrow^* \omega$ и $|\omega| < k$ либо $\exists \beta : \alpha \rightarrow^* \omega\beta$ и $|\omega| = k\}$, где $\alpha, \beta \in (N \cup \Sigma)^*$.

▲ **Определение 5.** Пусть $G = \langle N, \Sigma, P, S \rangle$ — КС-грамматика. Множество $FOLLOW_k$ строки формы $\beta \in (\Sigma \cup \Gamma)^*$ как:
$FOLLOW_k(\beta) = \{\omega \in \Sigma^* \mid \exists \gamma, \alpha : S \vdash^* \gamma\beta\alpha$ и $\omega \in FIRST_k(\alpha)\}$

Согласно определениям, $FIRST_k(A)$ и $FOLLOW_k(A)$ содержат, соответсвенно, все возможные первые и непосредственно следующие k символов в результирующем выводе, при использовании нетерминала A

Пусть дана грамматика $\langle \Sigma, N, S, P \rangle$. Алгоритм построения $FIRST_k$ следующий:

$\forall A \in N, FIRST_k(A) \leftarrow \varnothing$
$\forall a \in \Sigma, FIRST_k(a) \leftarrow \{a\}$
пока $FIRST_k(A)|_{a \in N}$ изменяется
    for $A \vdash X_1 \dots X_l \in P$
        $FIRST_k(A) \leftarrow FIRST_k(FIRST_k(X_1) \cdot \dots \cdot FIRST_k(X_l))$
    end for
end пока

Для построения $FOLLOW_k$ нужно выполнить сдедующее:

$FOLLOW_k(S) \leftarrow \{\varepsilon\}$
$\forall A \in N \setminus \{S\}\ FOLLOW_k(A) \leftarrow \varnothing$
пока $FOLLOW_k(A)|_{A \in N}$ изменяется
    for $B \vdash \beta \in P$
        for $\beta = \mu A \nu$ разбиений, где $A \in N, \mu, \nu \in (\Sigma \cup \{\$\} \cup N)^*$
            $FOLLOW_k(A) \leftarrow FOLLOW_k(A) \cup FIRST_k(FIRST_k(\nu) \cdot$
            $\cdot FOLLOW_k(B))$
        end for
    end for
end пока

Введём понятие таблицы, управляющей разбором. ▲
Определение 6. Управляющая таблица для грамматики $G = \langle \Sigma, N, P, S \rangle$ – это частичная фунция $T_k : N \times \Sigma^{\leq k} \vdash P \cup \{-\}$, отображающая пару: нетерминал A и m терминальных символов — $t_1 \dots t_m$, где $m \leq k$ — в правило, которое нужно применять, если такое правило есть в P.

По строкам в управляющей таблице размещаются все нетерминалы грамматики, по столбцам – всевозможные последовательности терминалов, длиной не более k[1,2], а также столбец для маркера конца строки — \$. В ячейке таблицы указано правило, которое нужно применять, если рассматривается нетерминал A, а следующие m символов строки — $t_1 \dots t_m$, где $m \leq k$, либо прочерк, если такого правила нет.

	$\dots$	$t_1 \dots t_m$	$\dots$	\$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
A	$\dots$	$A \vdash \alpha$	$\dots$	$\dots$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$

Управляющая таблица строится алгоритмически на основании построения для каждого нетерминала A вспомогательных множеств $FIRST_k(A)$ и $FOLLOW_k(A)$.

Приведём алгоритм построения T_k для всех $A \in N$ и $x \in \Sigma^{\leq k} \cup \{\$\}, k > 0$ по $FIRST_k$ и $FOLLOW_k$ (в начале элементы T_k инициализированы '—').

---

for $A \vdash \alpha \in P$
    for $x \in FIRST_k(FIRST(\alpha) \cdot FOLLOW_k(A))$
        если $T_k(A, x) =$'—' **то**
            $T_k(A, x) \leftarrow (A \vdash \alpha)$
        иначе
            Произошел конфликт: нет однозначного правила для A, x
        end если
    end for
end for

Заметим, что в псевдокоде построения таблицы ветвь с сообщением о конфликте нужна для сигнализирования о неоднозначности в заполнении ячейки: не для всех КС-грамматик по множествам $FIRST_k$ и $FOLLOW_k$ возможно выбрать применяемое правило, следовательно, нельзя и построить однозначную управляющую таблицу. Класс грамматик, для которых управ-

[1] На практике таблица может получиться довольно разреженной, поэтому столбцы для последовательностей, не выводимых из нетерминалов грамматики, опускают

[2] Теоретически показательный характер роста количества столбцов от k на практике, как правило, не реализуется, так как реальные языки программирования обычно не задаются грамматиками, дающими теоретически худший случай

ляющую таблицу можно построить без конфликтов, называют классом LL(k)-грамматик.

▲ **Определение 7.** LL(k) грамматика — грамматика, для которой для некоторого $k \geq 1$ существует управляющая таблица T_k, по которой можно однозначно определить, какое правило применять.

Теорема 3. (о построении управляющей таблицы) *Для LL(k)-грамматики* $G = \langle N, \Sigma, P, S\rangle$, *для любого её правила вида* $A \vdash X_1 \dots X_l \in P$, *если* $x \in FIRST_k(FIRST_k(X_1) \cdot \ldots \cdot FIRST_k(X_l) \cdot \cdot FOLLOW_k(A))$, *то ячейка управляющей таблицы* $T_k(A, x)$ *содержит единственное правило* $A \vdash X_1 \dots X_l$..

Теорема 3 утверждает, что для LL(k)-грамматики управляющая таблица может быть построена без возникновения конфликтов. Если же её условие приводит к противоречиям, то грамматика не является LL(k).

Критерий того, что грамматика является LL(k) грамматикой, непосредственно следует из определения:

Лемма 4 (об эквивалентном определении) $G = \langle N, \Sigma, P, S\rangle$ *является* $LL(k)$ *грамматикой тогда и только тогда, когда* $(\forall A \vdash \alpha | \beta \in P) \Rightarrow (FIRST_k(\alpha\gamma) \cap FIRST_k(\beta\gamma) = = \varnothing)$ *при всех таких* $\omega A \gamma$, *что* $S \vdash_{lm}^* \omega A \gamma$.

Дальнейшие рассуждения и построения будут проводиться для $k = 1$. Важно заметить, что при больших k управляющая таблица сильно разрастается[3], поэтому на практике алгоритм применим для небольших k.

В частном случае для $k = 1$:

▲ **Определение 8.** $FIRST(\alpha) = \{a \in \Sigma \mid \exists \gamma \in (N \cup \Sigma)^* : \alpha \vdash^* a\gamma\}$, где $\alpha \in (N \cup \Sigma)^*$

▲ **Определение 9.** $FOLLOW(\beta) = \{a \in \Sigma \mid \exists \gamma, \alpha \in (N \cup \Sigma)^* : S \vdash^* \gamma\beta a\alpha\}$, где $\beta \in (N \cup \Sigma)^*$ Множество $FIRST$ можно вычислить, пользуясь следующими соотношениями:

[3]Хоть и не показательно, как в теоретически худшем случае

- $FIRST(a\alpha) = \{a\}, a \in \Sigma, \alpha \in (N \cup \Sigma)^*$
- $FIRST(\varepsilon) = \{\varepsilon\}$
- $FIRST(\alpha\beta) = FIRST(\alpha) \cup (FIRST(\beta)$, если $\varepsilon \in FIRST(\alpha))$
- $FIRST(A) = FIRST(\alpha) \cup FIRST(\beta)$, если в грамматике есть правило $A \vdash \alpha \mid \beta$

Algorithm 1 Алгоритм для вычисления множества *FOLLOW*

Вход: Грамматика $G = \langle \Sigma, N, S, P \rangle$
Выход: $FOLLOW(A)$ для всех $A \in N$
  Положим $FOLLOW(X) \leftarrow \varnothing, \forall X \in N$
  $FOLLOW(S) \leftarrow FOLLOW(S) \cup \{\$\}$, где S — стартовый нетерминал
  пока множества *FOLLOW* меняются
    Для всех правил вида $A \vdash \alpha X \beta$:
    $FOLLOW(X) \leftarrow FOLLOW(X) \cup (FIRST(\beta) \setminus \{\varepsilon\})$.
    Для всех правил вида $A \vdash \alpha X$ и $A \vdash \alpha X \beta$, где $\varepsilon \in FIRST(\beta)$:
    $FOLLOW(X) \leftarrow FOLLOW(X) \cup FOLLOW(A)$
  end пока

Алгоритм для вычисления множества *FOLLOW* представлен в 1.

▲ **Задача 1** Рассмотрим грамматику G со следующими правилами:

- $S \vdash aS'$
- $A' \vdash b \mid a$
- $S' \vdash AbBS' \mid \varepsilon$
- $B \vdash c \mid \varepsilon$
- $A \vdash aA' \mid \varepsilon$

Посчитать множества FIRST и FOLLOW.

Решение

$FIRST$ для нетерминалов грамматики G:

$$FIRST(S) = \{a\} \qquad FIRST(B) = \{c, \varepsilon\}$$
$$FIRST(A) = \{a, \varepsilon\} \qquad FIRST(S') = \{a, b, \varepsilon\}$$
$$FIRST(A') = \{a, b\}$$

$FOLLOW$ для нетерминалов грамматики G:

$$\begin{aligned} FOLLOW(S) &= \{\$\} & \\ FOLLOW(S') &= \{\$\} & (S \vdash aS') \\ FOLLOW(A) &= \{b\} & (S' \vdash AbBS') \\ FOLLOW(A') &= \{b\} & (A \vdash aA') \\ FOLLOW(B) &= \{a, b, \$\} & (S' \vdash AbBS', \varepsilon \in FIRST(S')) \end{aligned}$$

Задача решена.

Теперь рассмотрим пример грамматики, не являющейся LL(1).

▲ **Задача 2** Построить не-$LL(1)$ грамматику.

Решение

Грамматика $S \vdash aS|a$, согласно теореме 4, не является LL(1)-грамматикой, так как $FIRST(aS) = FIRST(a) = \{a\}$ и $FIRST(aS) \cap FIRST(a) = \{a\}$, но LL(2)-грамматикой, так как $FIRST_2(aS) = \{aa\}$, $FIRST_2(a) = \{a\}$, и $FIRST_2(aS) \cap FIRST_2(a) = \varnothing$ — является.

Задача решена.

Очевидно, что в случае LL(1)-грамматики управляющая таблица заполняется следующим образом: правила $A \vdash \alpha, \alpha \neq \varepsilon$ помещаются в ячейки с индексами (A, a), где $a \in FIRST(\alpha)$, правила $A \vdash \alpha$ — в ячейки (A, a), где $a \in FOLLOW(A)$, если $\varepsilon \in FIRST(\alpha)$, а если при этом и $\$ \in FOLLOW(A)$, то и в ячейку $(A, \$)$. Иногда, для небольших грамматик, в целях наглядности в таблицу добавляют 2 столбца с $FIRST, FOLLOW$ множествами для нетерминалов.

▲ **Задача 3** Построить таблицу для грамматики $S \vdash aSbS \mid \varepsilon$

Решение

N	$FIRST$	$FOLLOW$	a	b	\$
S	$\{a, \varepsilon\}$	$\{b, \$\}$	$S \vdash aSbS$	$S \vdash \varepsilon$	$S \vdash \varepsilon$

Задача решена.

Теорема о связи LL(1)-грамматики с видом множеств $FIRST$ и $FOLLOW$ приведена ниже:

Лемма 5 (об эквивалентном определении) *Грамматика* $G = \langle \Sigma, N, S, P \rangle$ *и* $\omega \in \Sigma^*$ *является LL(1) тогда и только тогда, когда выполнены 2 условия:*

1) $(\forall A \vdash \alpha|\beta \in P) \Rightarrow$ $\Rightarrow (FIRST(\alpha) \cap FIRST(\beta) = \varnothing)$

2) $(\forall A \vdash \alpha|\beta \in P : \varepsilon \in FIRST(\alpha)) \Rightarrow$ $\Rightarrow (FOLLOW(A) \cap FIRST(\beta) = \varnothing)$

Здесь $\alpha, \beta \in (N \cup \Sigma)^*$ *— две сентенциальные формы* G*.*

Вернёмся к решению задачи 2 в свете леммы 5.

▲ **Задача 3** Проверить, что грамматика, задающая язык строк с равным количеством символов a и b: $S \vdash aSbS|bSaS|\varepsilon$, не является LL(1).

Решение

Но грамматика содержит правило $S \vdash \varepsilon$, и $\varepsilon \in FIRST(\varepsilon)$, следовательно, нужно проверять (2). $FOLLOW(S) = \{a, b, \$\}$ имеет непустое пересечение как с $FIRST(aSbS)$, так и с $FIRST(bSaS)$, поэтому (2) не выполняется, и грамматика не является LL(1).

Но грамматика содержит правило $S \vdash \varepsilon$, и $\varepsilon \in FIRST(\varepsilon)$, следовательно, нужно проверять (2). $FOLLOW(S) = \{a, b, \$\}$ имеет непустое пересечение как с $FIRST(aSbS)$, так и с $FIRST(bSaS)$, поэтому (2) не выполняется, и грамматика не является LL(1).

Задача решена.

Условия критерия накладывают довольно серьёзные ограничения на вид грамматики. В особенности:

1) Грамматика должна быть однозначной:

Пример 1.

$$\begin{aligned} &G: \\ S &\vdash aA|B|c \\ A &\vdash b|aA \\ B &\vdash aA|a\varepsilon \end{aligned}$$

Если анализируемая строка начинается с a, невозможно сделать однозначный выбор между $S \vdash aA$ и $S \vdash B$.

2) Даже вывод ε из двух правил альтернативы невозможен:

Пример 2.

$$\begin{aligned} &G: \\ S &\vdash aA \\ A &\vdash BC|B \\ C &\vdash b|\varepsilon \\ B &\vdash \varepsilon \end{aligned}$$

Рассмотрим два разных левых порождения a в G:

- $S \vdash_{lm} \underline{aA} \vdash_{lm} \underline{aB} \vdash_{lm} a$
- $S \vdash_{lm} \underline{aA} \vdash_{lm} \underline{aBC} \vdash_{lm} a$

В виду того, что из $B \vdash_{lm}^{*} \varepsilon$ и $BC \vdash_{lm}^{*} \varepsilon$, нельзя однозначно произвести подчёркнутый шаг левого порождения, а в G имеет два различных дерева вывода, и грамматика не является LL(1).

1.1.1. LL-алгоритм синтаксического анализа

LL(k) — синтаксический анализ — семейство алгоритмов нисходящего анализа без отката, с предпросмотром. Решение о том, какое правило применять, принимается по управляющей таблице T_k на основании просмотра k символов, непосредственно следующих за текущим во входной строке. Временная сложность алгоритма $O(n)$, где n — длина входной строки.

Для КС грамматики $\langle \Sigma, N, P, T \rangle$ алгоритм использует:

- входной буфер с указателем на позицию текущего символа
- стек с алфавитом $\Gamma = N \cup \Sigma \cup \{\$\}$ для хранения промежуточных данных
- таблицу T_k, управляющую разбором.

При чтении анализируемой строки во входе, алгоритм может заглядывать вперёд на k символов.

Конфигурацией алгоритма назовём пару $\langle x, X\alpha \rangle$ из множества таких пар Q, где x — неразобранная часть входной строки, $X\alpha \in \Gamma^*$ — содержимое стека, $X \in \Gamma$ — символ на вершине. При анализе строки w будем называть конфигурацию $\langle w, S\$ \rangle$ — стартовой, конфигурацию $\langle \$, \$ \rangle$ — конечной. Алгоритм, начиная со стартовой, на каждом шаге анализирует текущую конфигурацию, и выполняет действия, с учётом прочитанной части анализируемой строки: определяется цепочка исследуемых входных символов $u, |u| \leq k$ и символ на вершине стека X, затем, если $X \in N$, рассматривается элемент управляющей таблицы $T_k(X, u)$, и замена содержимого вершины стека правой частью правила из этого элемента; если X – терминальный символ, происходит сравнение с первым символом u, и в случае совпадения – извлечение X и сканирование очередного символа из ввода.

Опишем действия над конфигурациями, $\{f : Q \to Q, f \in \{match, lookup, success, error\}\}$, выполняемые в ходе работы алгоритма:

- **match** — в случае, когда на вершине стека — терминал, и символ на текущей позиции равен этому терминалу, то снять элемент со стека, сдвинуть указатель на 1 позицию вправо. $\langle x, X\alpha\rangle$ переводится в $\langle x', \alpha\rangle$, если $x = ax'$ и $X = a$

- **lookup** — в случае, когда текущая врешина стека — нетерминал X_i, и предпросмотрена подстрока u, найти в управляющей таблице T ячейку с координатами (X_i, u), положить на стек содержимое правой части этой ячейки так, чтобы самый левый символ оказался на вершине. $\langle x, X\alpha\rangle$ переводится в $\langle x, \beta\alpha\rangle$, если $T_k(X, u) = X \vdash \beta$ и $X = a$

- **success** — равершить работу при достижении конфигурации $\langle \$, \$\rangle$

- **error** — сигнализировать об ошибке и завершить работу.

Если алгоритм оказался в конечной конфигурации — разбор успешно завершён.

Алгоритм LL(1)-анализа

Опишем работу алгоритма LL(1)-анализа, как частного случая LL-анализа с предпросмотром на $k = 1$ символ. Алгоритм по-прежнему использует входную строку, управляющую таблицу, стек, и работает следующим образом:

- На каждом шаге алгоритма его конфигурация — это позиция во входной строке, начиная с которой расположена неразобранная её часть, и стек.

- В начале работы стек пуст, а позиция во входной строке соответствует её началу. На певом шаге в стек добавляются последовательно \$ и стартовый нетерминал S.

- На каждом шаге анализируется текущая конфигурация и совершается одно из действий:

```
stack.push($, S)
c ← input.scan()
пока stack.top() ≠ $
    X ← stack.top()
    если X = c то // match:
        stack.pop()
        c ← input.scan()
    иначе если X ∈ N то // lookup(X,c):
        если T[X, c] = X ⊢ X_1 ... X_m то
            stack.pop()
            stack.push(X_m, ..., X_1)
        иначе
            ошибка: пустая ячейка таблицы! //error
        end если
    иначе
        ошибка! //error
    end если
end пока
если c ≠ $ то
    ошибка: не вся строка разобрана! //error
end если//success
```

– Действие **success**. Если текущая позиция — конец строки, и вершина стека — символ конца строки, то разбор успешно завершен. Иначе, если стоим на конце строки — **error**.

– Действие **match**. Если текушая вершина стека — терминал, то анализатор проверяет, что позиция в строке соответствует этому терминалу. Если да, то снимает элемент со стека, сдвигает указатель на 1 позицию вправо, и продолжает разбор. Иначе — завершает разбор с ошибкой — **error**.

– Действие **lookup**. Если текущая врешина стека — нетерминал X_i, и текущий входной символ c, то ищет в управляющей таблице T ячейку с координатами (X_i, c) и кладёт на стек содержимое правой части этой ячейки так, чтобы самый левый символ оказался на вершине (операция $stack.push$ применена к символам правой части справа налево), иначе сигнализирует об ошибке — **error**.

Пример 3. *Пример работы LL(1) анализатора. Рассмотрим грамматику* $S \vdash aSbS \mid \varepsilon$ *и выводимое слово* $\omega = abab$.

Расмотрим пошагово работу LL(1)-алгоритма. Используем управляющую таблицу, построенную в предыдущем примере. Символ строки, доступный по указателю позиции в строке, выделен жирным шрифтом.

1) *Начало работы.*

   Стек:

   Входное слово:

$\boldsymbol{a}$	b	a	b	\$

   Стек пуст, по указателю доступен первый символ слова.

2) *Кладём* \$ *и стартовый символ* S *на стек*

   Стек:

S	\$

   Входное слово:

$\boldsymbol{a}$	b	a	b	\$

3) *Ищем ячейку с координатами (S, a), применяем правило из ячейки.*

   Стек:

a	S	b	S	\$

   Входное слово:

$\boldsymbol{a}$	b	a	b	\$

4) Снимаем терминал a со стека и сдвигаем указатель.

Стек:

S	b	S	$

Входное слово:

a	**b**	a	b	$

5) Ищем ячейку с координатами (S, b), применяем правило из ячейки.

Стек:

b	S	$

Входное слово:

a	**b**	a	b	$

6) Снимаем терминал b со стека и сдвигаем указатель.

Стек:

S	$

Входное слово:

a	b	**a**	b	$

7) Ищем ячейку с координатами (S, a), применяем правило из ячейки.

Стек:

a	S	b	S	$

Входное слово:

a	b	**a**	b	$

8) Снимаем терминал a со стека и сдвигаем указатель.

Стек:

S	b	S	$

Входное слово:

a	b	a	**b**	$

9) Ищем ячейку с координатами (S, b), применяем правило из ячейки.

Стек:

b	S	$

Входное слово:

a	b	a	**b**	$

10) Снимаем терминал b со стека и сдвигаем указатель.

Стек:

S	$

Входное слово:

a	b	a	b	**$**

11) Ищем ячейку с координатами (S, $), применяем правило из ячейки.

Стек:

$

Входное слово:

a	b	a	b	**$**

12) Оказались в конце строки и на вершине стека символ конца — завершаем разбор.

Шаг	**Стек**	**Остаток строки**	**Текущее действие**
0	|	**abab$**	$stack.push(\$, S)$
1	|$|S	**abab$**	$lookup(S, a)$
2	|$|S|b|S|a	**abab$**	$match$
3	|$|S|b|S	**bab$**	$lookup(S, b)$
4	|$|S|b	**bab$**	$match$
5	|$|S	**ab$**	$lookup(S, a)$
6	|$|S|b|S|a	**ab$**	$match$
7	|$|S|b|S	**b$**	$lookup(S, b)$
8	|$|S|b	**b$**	$match$
9	|$|S	**$**	$lookup(S, \$)$
10	|$	**$**	$match$

Таблица 1.1. Разбор слова $abab$ в грамматике $S \vdash aSbS \mid \varepsilon$ по LL(1)-алгоритму

Можно расширить данный алгоритм так, чтобы он строил дерево вывода. Дерево будет строиться сверху вниз, от корня к листьям. Для этого необходимо расширить действия:

- В ситуации, когда выполняется **match** (на вершине стека и во входе — одинаковые терминалы), нужно создать листовую вершину с соответствующим терминалом.

- В ситуации, когда нетерминал в стеке заменяется на правую часть правила в ходе выполнения **lookup**, нужно создать нелистовую вершину, соответствующую нетерминалу в левой части применяемого правила.

Дерево вывода для LL(1), как и в целом для LL(k), будет строиться однозначно, что следует из однозначности грамматик.

Также отметим, что LL-анализ, как и безоткатный рекурсивный спуск, не работает с леворекурсивными грамматиками: алгоритм может зациклиться. Таким образом, по некоторым грамматикам можно построить LL(k)-анализатор (для LL(k) грамматик), но не по всем. Методы борьбы с левой рекурсией даны в следующих разделах, а вот с неоднозначностями ничего не поделаешь.

1.1.2. Рекурсивный спуск

Идея рекурсивного спуска основана на использовании стека вызовов программы в качестве стека анализатора следующим образом:

- Для каждого нетерминала программируется функция, принимающая необработанный остаток строки s и возвращающая пару: результат вывода префикса данной строки из соответствующего нетерминала (выводится/не выводится) и новый необработанный остаток строки.
- Каждая функция реализует обработку цепочки согласно правым частям правил для соответствующих нетерминалов: считывание символа ввода при обработке терминального символа, вызов соответствующей функции при обработке нетерминального.

У данного подхода есть два ограничения:

1) Неприменим для грамматик, содержащих левую рекурсию. Иначе анализатор может зациклиться.
2) Шаги должны быть однозначными. Иначе нет возможности детерминированно выбрать конкретную функцию для вызова в некоторых ситуациях.

Если в грамматике, для которой разрабатывается рекурсивный спуск, есть альтернатива $A \vdash u_1|\ldots|u_z$, то однозначный выбор применяемой функции обработки нетерминала A (либо применяемого правила в вызываемой функции, если для каждого правила в альтернативе не реализована отдельная функция) может быть автоматизированно осуществлён по проверке условия наличия префикса ещё не обработанной части строки s длины не более k в $FIRST_k(u_j), j \in [1, z]$, причём условие должно выполняться не более чем для одного j, иначе грамматика неоднозначна. Если такой j не найден, но существует $\hat{j} \in [1, z]$, такой, что, $u_{\hat{j}} \vdash^* \varepsilon$, и данный префикс принадлежит $FOLLOW_k(A)$, то можно положить $j = \hat{j}$. В данных и только в данных случаях правило $A \vdash u_j$ может быть выбрано для применения. Следовательно, для однозначного выбора правила требуется проанализировать $FIRST_k(A)$ и $FOLLOW_k(A)$, и, классически, рекурсивному спуску подлежат языки, задаваемые классом LL(k) грамматик[4].

Приведём алгоритм выбора правила из альтернативы для $k = 1$.

Рассматривается альтернатива: $A \vdash u_1 | \dots | u_z$
$inSym$ – первый символ необработанной части строки
если $(\exists j \in [1, z]) : inSym \in FIRST(u_j))$ **то**
    Выбрать правило $A \vdash u_j$
иначе если $(\exists \hat{j} \in [1, z]) : u_{\hat{j}} \vdash^* \varepsilon \ \& \ inSym \in FOLLOW(A))$ **то**
    Выбрать правило $A \vdash u_{\hat{j}}$
иначе
    Ошибка!
end если

Приведём общий вид функции обработки $funcA$ нетерминала A, символически обозначая считывание символа из входного потока s, моделируемого объектом класса строки, реализующего методы $s.current$, возвращающий символ в текущей позиции, и $s.scan$, который возвращает терминальный символ и модифицирует s так, что в нём после вызова $c = s.scan()$ остаток строки, расположенный за c. Если возвращаемое значение самой первой в стеке вызовов функции — пара вида *(True, [])*, то разбор завершился успехом. Временная сложность алгоритма от длины строки n — $O(n)$, так как строка сканируется только один раз.

[4] на практике это ограничение может быть ослаблено различными ухищрениями, вроде откатов и пр.

```
если len(s) = 0 то
    return(True, w)
end если
Текущее правило: A ⊢ X_1 X_2 ... X_k
for i ∈ [1, k]
    если X_i ∈ N то
        res, s ← funcX_i(s) // call, C()
        если res = False то
            return (False, s) // return, R()
        end если
    иначе если (X_i ∈ Σ ∪ {ε}) & ((X_i = ε)||X_i = s.current()) то
        if X_i ≠ ε: s.scan() // match_terminal, M_Σ()
        если i = k то
            return (True, s) // return, R()
        end если
    иначе
        return (False, s) // return, R()
    end если
end for
```

Заметим, что алгоритм совершает 3 типа действий[5]:

1) **call**, $C()$: если символ на текущей позиции рассматриваемого правила — нетерминал, совершить вызов функции его обработки.

2) **return**, $R()$: возврат из вызова. Производится при попытке сдвига с крайней правой позиции в рассматриваемом правиле, либо в случае пустого слова во вводе, либо в случае ошибки.

3) **match terminal**, $M_\Sigma()$: если символ на текущей позиции в правиле — ε, просто сдвинуть позицию на 1. Если терминал — проверить соответствие его текущему входному символу, и, если они равны, то сдвинуть позицию в правиле на 1 и считать следующий символ.

Заметим, что действия **call** и **return** реализуют логику **lookup** из алгоритма анализа по таблице, **match terminal** — логику **match**.

[5]Как правило, на практике эти действия не формализуют

Рассмотрим работу рекурсивного спуска реализующего разбор слова $aabb$ по грамматике $S \vdash aSbS \mid \varepsilon$

Шаг	Стек вызовов	Остаток строки	Текущее действие
0	|main|**S(aabb$)**	**aabb$**	$M_\Sigma(S \vdash \mathbf{a}SbS, aabb\$)$
1	|main|**S(aabb$)**	**abb$**	$C(S \vdash a\mathbf{S}bS, S)$
2	|main|S|**S(abb$)**	**abb$**	$M_\Sigma(S \vdash \mathbf{a}SbS, abb\$)$
3	|main|S|**S(abb$)**	**bb$**	$C(S \vdash a\mathbf{S}bS, S)$
4	|main|S|S|**S(bb$)**	**bb$**	$M_\Sigma(S \vdash \varepsilon, bb\$), R()$
5	|main|S|**S(abb$)**	**bb$**	$M_\Sigma(S \vdash aS\mathbf{b}S, bb\$)$
6	|main|S|**S(abb$)**	**b$**	$C(S \vdash aSb\mathbf{S}, S)$
7	|main|S|S|**S(b$)**	**b$**	$M_\Sigma(S \vdash \varepsilon, b\$), R()$
8	|main|S|**S(abb$)**	**b$**	$R()$
9	|main|**S(aabb$)**	**b$**	$M_\Sigma(S \vdash aS\mathbf{b}S, b\$)$
10	|main|**S(aabb$)**	**$**	$C(S \vdash aSb\mathbf{S}, S)$
11	|main|S|**S($)**	**$**	$M_\Sigma(S \vdash \varepsilon, \$), R()$
12	|main|**S(aabb$)**	**$**	$R()$

Таблица 1.2. Разбор слова $aabb$ в грамматике $S \vdash aSbS \mid \varepsilon$ рекурсивным спуском.

Данный подход применяется как для ручного написания синтаксических анализаторов, так и при генерации анализаторов по грамматике, например средствами ANTLR.

1.1.3. Преобразования грамматики к LL(1)

Иногда грамматику $G = \langle \Sigma, N, S, P \rangle$, не являющуюся LL(1), можно привести к LL(1) грамматике. В первую очередь, можно применить методы устранения левой рекурсии и левую факторизацию. Следует отметить, что в ходе преобразований не всякая грамматика становится LL(1), а также то, что грамматика может стать менее понятной. Также доказано, что существование LL грамматики, эквивалентной G, является алгоритмически неразрешимой задачей.

Устранение левой рекурсии

Непосредственная левая рекурсия, то есть правила вида $A \vdash A\alpha$, можно устранить следующим образом.

1) Группируем правила с A в левой части: $A \vdash A\alpha_1|\ldots|A\alpha_m|\beta_1|\ldots|\beta_n$, где никакая из сентенциальных форм β_i не начинается с A.

2) Добавляем новый нетерминал A'

3) Заменяем этот набор правил на

$$A \vdash \beta_1 A'|\ldots|\beta_n A'$$
$$A' \vdash \alpha_1 A'|\ldots|\alpha_m A'|\varepsilon$$

Теперь из A можно вывести те же строчки, что и раньше, но без левой рекурсии. Заметим, что в ходе данного преобразования появляются новые ε-правила, по одному на каждый добавленный нетерминал. Метод выше устраняет только непосредственную левую рекурсию.

Пусть дана грамматика $G = \langle \Sigma, N, S, P \rangle$, не содержащая ε-правил. Для удаления из G скрытой левой рекурсии, включающей два и более шага, применяется следующий алгоритм:

```
Нетерминалы пронумерованы в произвольном порядке, n ← |N|
for i ∈ [1, n]
    for j ∈ [1, i − 1]
        A_j ⊢ β_1|...|β_k — все текущие правила для A_j
        Заменить все A_i ⊢ A_j α на A_i ⊢ β_1 α|...|β_k α
    end for
    удалить правила A_i ⊢ A_i
    устранить непосредственную левую рекурсию для A_i.
end for
```

Полученная грамматика не содержит левой рекурсии. В ходе преобразования могут появиться ε-правила.

Левая факторизация

Идея левой факторизации лежит в том, чтобы в случае, когда неясно, какую из альтернатив применять для раскрытия нетерминала A, изменить правила для A так, чтобы отложить решение до тех пор, пока не будет достаточно информации для принятия однозначного решения.

Преобразование: для правил $A \vdash \alpha\beta_1|\alpha\beta_2$ грамматики $G = \langle \Sigma, N, S, P \rangle$ и непустой строчки с префиксом, выводимым из α, когда неизвестно, какое правило применять, можно добавить новое правило $A \vdash \alpha A'$, и после анализа того, что выводимо из α, попробовать применить новое правило $A' \vdash \beta_1$ либо $A' \vdash \beta_2$.

пока В грамматике есть альтернативы с общим префиксом

  Для каждого $A \in N$ найти самый длинный префикс α для альтернатив в P с A в левой части.

  если $\alpha \neq \varepsilon$ **то**

    Заменить все $A \vdash \alpha\beta_1|\dots|\alpha\beta_m|\gamma$ на:

    $A \vdash \alpha A'|\gamma$

    $A' \vdash \beta_1|\dots|\beta_m$

  end если

end пока

После преобразования грамматика может стать не LL(1) (см. задачу 3).

Глава 2. **Практические задания**

2.1. Задания по теоретическому блоку

▲ Для самостоятельного решения

1) **a)** Приведите пример LL(0) грамматики
   b) Приведите пример LL(2), но не LL(1) грамматики.
   c*) Приведите пример не LL(2) грамматики.

2) Для грамматик из задачи 1 постройте LL-анализаторы и разберите слово длины хотя бы 5 символов.

3) Является ли грамматика G LL(1) грамматикой:

$$\begin{aligned} G&: \\ S &\vdash A|B \\ A &\vdash aA|c \\ B &\vdash aB|b \end{aligned}$$

4) В литературе по рекурсивному спуску иногда приводят так называемый канонический вид грамматики для рекурсивного спуска:

   ▲ **Определение.** Грамматика $G = \langle \Sigma, N, S, P \rangle$ и $\omega \in \Sigma^*$ называется грамматикой в *каноническом виде для рекурсивного спуска*, если её правила удовлетворяют одному из следующих видов:

   - $A \vdash \alpha, \alpha \in (N \cup \Sigma)^*$ — единственное правило вывода для $A \in P$
   - $A \vdash a_1\alpha_1|\dots|a_m\alpha_m : \alpha_i \in (N \cup \Sigma)^*, a_i \in \Sigma, \forall i = 1 \dots m$, и $\alpha_i \neq \alpha_j$, если $i \neq j$
   - $A \vdash a_1\alpha_1|\dots|a_m\alpha_m|\varepsilon : \alpha_i \in (N \cup \Sigma)^*, a_i \in \Sigma, \forall i = 1 \dots m$, и $\alpha_i \neq \alpha_j$, если $i \neq j$, и $FIRST(A) \cap FOLLOW(A) = \varnothing$.

   Докажите, что грамматики, заданные определением, являются LL(1).

   Следует отметить, что в реализациях безоткатного рекурсивного спуска[1] для реальных языков программирования условия из задачи 4 могут нарушаться.

[1] не говоря уже о рекурсивном спуске с откатом

5) Рассмотрим классический пример `'if, if-else'` в реализации некоторых языков программирования:

$$S \vdash if\ E:\ S|if\ E:\ S\ else:\ S|a$$
$$E \vdash b$$

После левой факторизации грамматика имеет вид:

$$S \vdash if\ E:\ SS'|a$$
$$S' \vdash else:\ S|\varepsilon$$
$$E \vdash b$$

Самостоятельно проведите левую факторизацию. Является ли полученная грамматика LL(1) грамматикой?

2.2. Разработка LL(1)-анализатора на языке Python

▲ Задача для программирование

1) Реализуйте LL(1) анализатор на языке Python.

В силу изложенного во Главе 1, анализатор будет состоять из 4 модулей:

1) Реализация алгоритма разбора по управляющей таблице.

2) Реализация алгоритма построения FIRST.

3) Реализация алгоритма построения FOLLOW.

4) Реализация алгоритма построения управляющей таблицы.

Приведем возможный вариант реализации модулей 1,4. Модули 2,3 выделим читателям на самостоятельную работу.

2.2.1. Реализация алгоритма разбора по управляющей таблице

Листинг 2.1. Реализация LL(1)-разбора

```
def ll1_algorithm(
    input_string: str,
    parsing_table: dict[tuple[str, str], dict[str, str]],
    starting_symbol: str,
    end_symbol: str,
    epsilon_symbol: str,
):
    taken_len = 0
    stack = []
    stack.append(end_symbol)
    stack.append(starting_symbol)

    non_term_term_pairs = parsing_table.keys()
    non_terminals = {pair[0] for pair in non_term_term_pairs}
    terminals = {pair[1] for pair in non_term_term_pairs}

    current = input_string[taken_len]
    taken_len += 1

    while stack[-1] != end_symbol:
        stack_top = stack[-1]
        if stack_top == epsilon_symbol:
            stack.pop()
            continue
        if stack_top == current:
            stack.pop()
            current = input_string[taken_len]
            taken_len += 1
        elif stack_top in non_terminals:
            corresponding_rule = parsing_table[stack_top, current]
            if corresponding_rule is None:
                raise LookupError(
                    f"parsing␣table␣empty␣with␣non_term\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣{stack_top}␣and␣term␣{current},␣which␣is\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣{taken_len}␣symbol␣from␣{input_string}"
                )
            stack.pop()
            stack.extend(corresponding_rule[stack_top][::-1])
        else:
            raise LookupError(
                f"terminal␣{current}␣#{taken_len}␣from\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣input␣{input_string}␣is␣not␣equal␣to\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣terminal{stack_top}␣from␣stack␣while␣parsing"
            )

    if current != end_symbol:
        raise LookupError(
            f"parsing␣of␣input␣{input_string}␣with\
␣␣␣␣␣␣␣␣␣␣␣␣len{len(input_string)}
␣␣␣␣␣␣␣␣␣␣␣␣ended␣on␣{taken_len}␣symbol"
```

```
        )
```

2.2.2. Реализация алгоритма построения управляющей таблицы

Листинг 2.2. Построение управляющей таблицы

```
def make_parsing_table(
    terminals: set[str],
    non_terminals: set[str],
    rule_set: dict[str, set[str]],
    first: dict[str, set[str]],
    follow: dict[str, set[str]],
    epsilon_symbol: str,
) -> dict[tuple[str, str], dict[str, str]]:
    parsing_table = {}
    for non_term in non_terminals:
        for term in terminals:
            parsing_table[(non_term, term)] = None

    for non_term in non_terminals:
        for expand_str in rule_set[non_term]:
            rule_to_add = {non_term: expand_str}
            term_pool = first[expand_str[0]]

            if epsilon_symbol in term_pool:
                term_pool.remove(epsilon_symbol)
                term_pool = term_pool | follow[non_term]

            for term in term_pool:
                if parsing_table[(non_term, term)] is None:
                    parsing_table[(non_term, term)] = rule_to_add
                else:
                    raise ValueError(
                        f"Conflicting rules for {non_term} and \
                        {term}:{parsing_table[(term, non_term)]} \
                        and {rule_to_add}"
                    )

    return parsing_table
```

2.3. Задания для самостоятельного программирования

1) Реализуйте функцию построения FIRST для использования в реализации LL(1)-анализатора выше.

2) Реализуйте функцию построения FIRST для использования в реализации LL(1)-анализатора выше.

3) Реализуйте программу, производящую по введенной любым удобным способом LL(1)-грамматике соответственно построение FIRST, FOLLOW, таблицы разбора, и LL(1)-анализатора, использующего данную таблицу, для разбора тестовых слов, вводимых любым удобным способом. Протестируйте на примерах из Главы 1 и отладьте полученную программу.

В заданиях предполагается, что функция, отображающая нетерминал в соответствующее множество, задается как словарь, ключами которого являются нетерминалы, заданные строками, а значениями – множества строк.

Учебное издание

Ефанов Николай Николаевич
Отращенко Алексей Иванович

НИСХОДЯЩИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ

Редактор *И. А. Волкова*. Корректор *Н. Е. Кобзева*
Компьютерная верстка *Н. Е. Кобзева*

Подписано в печать 05.08.2024. Формат 60×84 $^1/_{16}$.
Усл. печ. л. 5,9. Уч.-изд. л. 5,3. Тираж 150 экз. Заказ № 000.

Федеральное государственное автономное образовательное учреждение высшего образования
«Московский физико-технический институт
(национальный исследовательский университет)»
141700, Московская обл., г. Долгопрудный, Институтский пер., 9
Тел. (495) 408-58-22, e-mail: rio@mail.mipt.ru

Отдел оперативной полиграфии «Физтех-полиграф»
141700, Московская обл., г. Долгопрудный, Институтский пер., 9
Тел. (495) 408-84-30, e-mail: polygraph@mail.mipt.ru