МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Вл. Пономарев

Конспективное изложение теории языков программирования и методов трансляции

Книга 3. Синтаксический разбор

Учебно-методическое пособие

Озерск, 2019

УДК 681.3.06
П 56

Пономарев В.В. Конспективное изложение теории языков программирования и методов трансляции. Учебно-методическое пособие. В 4-х книгах. Книга 3. Синтаксический разбор. Озерск: ОТИ НИЯУ МИФИ, 2019. — 55 с., ил.

В книге 3 описываются основные методы синтаксического анализа. В этой части рассматриваются КС-языки и МП-автоматы, универсальные методы разбора цепочек КС-языков, детерминированные методы нисходящего и восходящего анализа.

В качестве вспомогательного материала пособие предназначено для студентов старших курсов, обучающихся по направлению подготовки 09.03.01 «Информационная и вычислительная техника», или по специальности 09.05.01 «Применение и эксплуатация автоматизированных систем специального назначения».

Рецензенты:
1.
2.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

Содержание

Введение

Эта предварительная версия документа находится в стадии написания.

Текст пособия постоянно совершенствуется, поэтому рекомендуется использовать его электронную версию, которая периодически обновляется на сайте http://revol.ponocom.ru.

3. Синтаксический разбор

Целью синтаксического разбора является проверка соответствия входного текста синтаксическим правилам, которые в языках программирования описываются при помощи бесконтекстных грамматик. Результатом разбора является синтез внутреннего представления текста в ходе семантического анализа дерева вывода (рисунок 3.1).

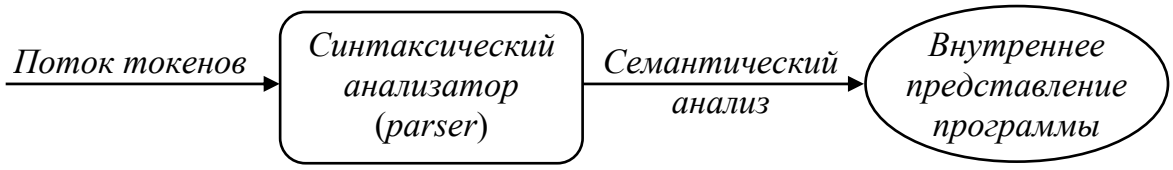

Рисунок 3.1. Синтаксический разбор

На вход синтаксического анализатора поступают полученные в результате лексического анализа токены (лексемы), которые для синтаксического анализатора являются терминальными символами.

Различают нисходящий и восходящий синтаксический разбор.

Нисходящий разбор (*top-down parsing*) начинается с целевого символа грамматики и заканчивается терминальными символами. Нисходящему разбору соответствует построение дерева вывода сверху вниз.

Разбор завершается, когда все нетерминальные символы раскрываются в соответствующие им терминальные символы.

Восходящий разбор (*bottom-up parsing*), наоборот, начинается с терминальных символов и завершается целевым символом. Восходящему разбору соответствует построение дерева вывода снизу вверх. Разбор завершается при достижении корневой вершины дерева, соответствующей целевому символу грамматики. Восходящий разбор сложнее нисходящего, но позволяет распознать большее подмножество КС-языков.

Для синтаксического анализа можно использовать несколько классов контекстно-свободных языков и грамматик. В пособии рассматриваются только наиболее характерные и важные из них.

Напомним, что для КС-грамматики $G=(\Sigma, N, P, S)$, $V=\Sigma\cup N$, правила имеют следующий вид: $A\to\beta$, $A\in N$. Если при этом $\beta\in V^*$, грамматика укорачивающая КС (УКС), если $\beta\in V^+$, грамматика неукорачивающая КС (НКС). Эти два типа грамматик определяют почти эквивалентные языки, различающиеся максимум на пустую цепочку.

В этой главе рассматривается начальная часть синтаксического анализа, называемая *разбором*, подразумевающим получение цепочки правил. Семантический анализ и построение внутреннего представления рассматривается в следующей главе.

3.1. Контекстно-свободные языки

Некоторые языки отличаются от регулярных языков тем, что цепочки языка образуют симметричную структуру. Примерами являются языки парности скобок и палиндромов. Язык парности скобок описывает цепочки, в которых правых скобок столько же, сколько и левых.

Язык парности скобок не регулярный. Попытка построить конечный автомат для этого языка окажется неудачной. Парность скобок описывает следующая бесконтекстная грамматика:

$$S \rightarrow aSb \mid \lambda \qquad (3.1)$$

Здесь скобки условно обозначены терминалами a и b. Грамматика такого вида называется само-вложенной (*self-embedding*). Нетерминал S вставляется между скобками a и b, и эта рекурсия обеспечивает парность. Терминалы a и b расположены с разных сторон нетерминала, и это не дает возможности построить конечный автомат, который может приписывать терминалы к цепочке только с одной ее стороны.

Язык парности скобок описывает также формула $a^n b^n$, $n>0$. Как было показано ранее, этот язык не регулярный по лемме о разрастании для регулярных языков. Эта лемма утверждает, что в цепочке регулярного языка есть итерация, которая может иметь любое число повторений. В цепочке бесконтекстного языка есть две итерации, которые могут иметь любое одинаковое число повторений, и это утверждает лемма о разрастании для бесконтекстных языков, которую мы рассмотрим далее.

В цепочке регулярного языка может быть две итерации, и произвольное число повторений любой одной из них не нарушает язык.

Точно так же в цепочке бесконтекстного языка может быть две пары итераций, но произвольное число повторений итераций одной пары не нарушает язык. В качестве примера множественности пар итераций рассмотрим язык, описываемый следующей грамматикой:

$$S \rightarrow SS \mid aSb \mid \lambda \qquad (3.2)$$

Этот язык также описывает парность скобок, но групп сбалансированных скобок может быть сколько угодно, и повторение скобок любой одной группы не нарушает язык.

Палиндромы четной длины, состоящие из символов a и b, описывает следующая бесконтекстная само-вставляющая грамматика:

$$S \rightarrow aSa \mid bSb \mid \lambda \qquad (3.3)$$

Примером цепочки является *abbaabba*. В ней можно найти, например, две подцепочки *bb*, которые можно сколь угодно раз повторить, и новая цепочка также будет палиндромом четной длины.

Симметрия цепочек может быть выражена отражением. Следующая грамматика описывает язык цепочек из равного числа символов *a* и *b*:

$S \rightarrow aSb \mid bSa \mid \lambda$ (3.4)

Симметрия здесь заключается в том, что каждый символ *a* начала цепочки отражается в символ *b* конца цепочки и наоборот. Повторение подцепочек с равным числом символов *a* и *b* левой и правой частей и здесь не нарушает язык.

Если внимательно посмотреть на классическую грамматику G_1 языка выражений со скобками, то в ней обнаруживается рекурсия символа *E*, к которому привязаны скобки:

$E \rightarrow E+T \mid T$

$T \rightarrow T*P \mid P$

$P \rightarrow a \mid (E)$

В этой грамматике можно построить следующий вывод:

$E \Rightarrow^* (E) \Rightarrow^* ((E)) \ldots$

Здесь также присутствует само-вложенность. Поэтому любой язык, вычисляющий выражения со скобками, будет бесконтекстным.

Другой особенностью бесконтекстных грамматик является парность нетерминалов. Следующая грамматика абстрагирует парность ветвей if и else условного оператора:

$S \rightarrow iSS \mid o$

Здесь *i* соответствует if, *o* соответствует оператору ветви if или else, а else для простоты опущено. Симметрия этого языка выражается в парности цепочек *io* и *o*. Поэтому для достаточно длинной цепочки языка повторение подцепочки *io* в начальной части и подцепочки *o* в конечной части не нарушает язык.

Парность нетерминалов не всегда порождает бесконтекстный язык. Следующая грамматика абстрагирует последовательность операторов программы:

$S \rightarrow SS \mid a \mid b \mid \lambda$

Здесь *a* и *b*, условно, — некоторые операторы. Порождаемый язык представляет собой произвольную последовательность символов *a* и *b*, которую можно получить при помощи конечного автомата, и любую одну подцепочку можно повторить без нарушения языка.

Интересен вопрос, можно ли найти в цепочке бесконтекстного языка не две, а три цепочки итераций с одинаковым числом повторений? Например, является ли бесконтекстным язык $L = \{a^n b^n c^n, n > 0\}$?

Следующая лемма отвечает на этот вопрос.

3.1.1. Лемма о разрастании для бесконтекстных языков

Неформально лемма о разрастании бесконтекстных языков звучит так. Если взять достаточно длинную цепочку символов, принадлежащую произвольному бесконтекстному языку, то в ней всегда можно выделить две подцепочки, длина которых в сумме больше нуля, таких, что, повторив их сколь угодно большое число раз, можно получить новую цепочку символов, принадлежащую данному языку.

Формально лемма записывается следующим образом.

Лемма. Если некоторый язык L является бесконтекстным языком, то существует константа $p > 0$, такая, что если $\alpha \in L$ и $|\alpha| \geq p$, то $\alpha = uvwxy$, где $vx \neq \lambda$, $|vwx| \leq p$, и тогда $\alpha' = uv^i wx^i y \in L \ \forall i \geq 0$. ●

Доказано, что если для некоторого языка не выполняется лемма о разрастании, то этот язык не является бесконтекстным.

В качестве примера докажем, что язык $L = \{a^n b^n c^n \mid n > 0\}$ не является бесконтекстным. Предположим, что L является КС-языком. Возьмем цепочку $\alpha = a^p b^p c^p$, принадлежащую языку, $|\alpha| > p$. Запишем цепочку в виде $\alpha = uvwxy$. Возьмем цепочку uv^2wx^2y. Если $v \in a^+$ или $x \in c^+$, то в цепочке α символов b меньше, чем a или c. Если $v \in a^+b^+$ или $x \in b^+c^+$, то в цепочке α символы чередуются. Следовательно, язык L не является контекстно-свободным.

Не являются бесконтекстными также языки: $L_{ww} = \{ww \mid w \in (a, b)^*\}$ и $L_{abcd} = \{a^m b^n c^m d^n \mid m \geq 1, n \geq 1\}$. Доказательство этого оставляем в качестве упражнения. Язык L_{ww} абстрагирует проблему описания идентификатора до его первого использования. Несмотря на то, что язык кажется простым, описать его при помощи бесконтекстной грамматики нельзя. Язык L_{abcd} абстрагирует проблему проверки соответствия формальных и фактических параметров в описании и вызове процедуры.

Различие между регулярными и КС-языками хорошо видно из сравнения регулярного языка a^*b^* и КС-языка $a^n b^n$. Для языка a^*b^* выполняется также лемма о разрастании для бесконтекстных языков, так как регулярные языки являются подмножеством бесконтекстных языков.

Обоснованием леммы о разрастании для бесконтекстных языков является тот факт, что бесконтекстный язык описывается грамматикой, которая содержит хотя бы один само-вложенный нетерминал.

Это следует из того, что, во-первых, бесконечный язык описывается только при помощи рекурсии какого-либо нетерминала.

Во-вторых, рекурсивное правило не может не иметь символов и слева и справа от рекурсивного символа. Если это условие не выполняется, язык является регулярным. Само-вложенность нетерминала формирует определенную симметрию цепочек и две итерации.

3.2. Автоматы с магазинной памятью

Распознавателем контекстно-свободных языков является автомат с магазинной памятью, МП-автомат. Его можно рассматривать как недетерминированный конечный автомат с λ-переходами, снабженный магазином (стеком), в который автомат может записывать любые символы грамматики и специальный магазинный символ, маркер дна.

Формально МП-автомат описывается семеркой

$P = (Q, \Sigma, Z, \delta, q_0, \perp, F)$,

где Q — конечное множество состояний, Σ — конечный входной алфавит, Z — конечный магазинный алфавит, δ — функция переходов, отображает множество $Q\times(\Sigma\cup\{\lambda\})\times Z$ на множество $Q\times Z^*$, $q_0\in Q$ — начальное состояние, ⊥ — начальный символ на вершине стека (маркер дна), $F\subseteq Q$ — множество допускающих состояний.

Схема МП-автомата показана на рисунке 3.2.

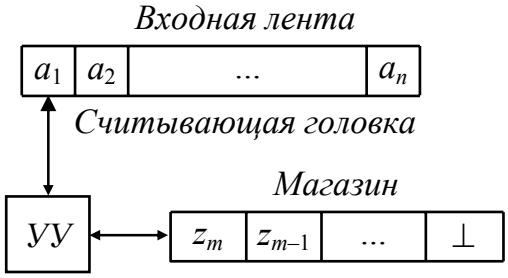

Рисунок 3.2. Схема МП-автомата

Конфигурация МП-автомата описывается тройкой (q, *w*, γ), где q — текущее состояние, *w* — непрочитанная часть входа, γ — содержимое стека. По соглашению вершина стека этого автомата находится слева.

Функция переходов описывает возможные переходы МП-автомата, учитывая не только текущее состояние и обозреваемый символ, но и содержимое стека, а именно, символ на вершине. Аргументами функции переходов δ являются текущее состояние q, символ входа *a*, и символ на вершине стека *z*. Возвращаемым значением функции δ является множество пар (r, α), где r — новое состояние автомата, α — цепочка, заменяющая собой верхний символ стека *z*.

Шаг или такт работы МП-автомата представляет собой следующее бинарное отношение ⊢ на множестве конфигураций:

если $(q_2, \beta)\in\delta(q_1, a, z)$ *то* $(q_1, aw, z\gamma)\vdash(q_2, w, \beta\gamma)$.

Оно показывает, что если существует переход $(q_2, \beta)\in\delta(q_1, a, z)$, то, находясь в состоянии q_1, обозревая на входе символ *a*, на стеке символ *z*, автомат переходит в состояние q_2, заменяя символ *z* на стеке цепочкой β, и перемещая считывающую головку к следующему символу входа.

Замыкание $\vdash^*$ отношения $\vdash$ используется для описания нуля и более переходов. Оно определяется индуктивно следующим образом.

Базис. Если I — некоторая конфигурация МП-автомата, то $I \vdash^* I$.

Индукция. Если I, J и K — конфигурации МП-автомата, то $I \vdash^* K$, если $I \vdash J$ и $J \vdash^* K$.

МП-автомат может произвольно выполнять λ-переходы, когда обозреваемый символ входа не принимается во внимание, и считывающая головка не перемещается, но при этом может изменяться состояние и (или) содержимое стека. В этом случае функция переходов записывается следующим образом: $(q_2, \beta) \in \delta(q_1, \lambda, z)$. Данный автомат не может выполнять такты, если магазин пуст, но может выполнять пустые такты по прочтении всей входной цепочки.

Начальной конфигурацией ПМ-автомата является $(q_0, w, \perp)$, — автомат находится в начальном состоянии q_0, на входе находится цепочка терминалов w, на стеке находится маркер дна $\perp$.

Заключительной конфигурацией является (q, λ, α), где $q \in F$, $\alpha \in Z^*$, то есть автомат находится в допускающем состоянии q, на входе пусто (цепочка прочитана), на стеке находится некоторая цепочка α.

Пример 3.1. Рассмотрим МП-автомат $P(\{q_0, q_1, q_2\}, \{a, b\}, \{\perp, a, b\}, \delta, q_0, \perp, q_2)$, распознающий язык палиндромов четной длины.

Зададим функцию переходов δ следующим образом:

1. $\delta(q_0, a, \perp) = \{(q_0, a\perp)\}$ — проталкивает a, когда на стеке маркер.
2. $\delta(q_0, b, \perp) = \{(q_0, b\perp)\}$ — проталкивает b, когда на стеке маркер.
3. $\delta(q_0, a, a) = \{(q_0, aa)\}$ — проталкивает a, когда на стеке a.
4. $\delta(q_0, a, b) = \{(q_0, ab)\}$ — проталкивает b, когда на стеке b.
5. $\delta(q_0, b, a) = \{(q_0, ba)\}$ — проталкивает a, когда на стеке a.
6. $\delta(q_0, b, b) = \{(q_0, bb)\}$ — проталкивает b, когда на стеке b.
7. $\delta(q_0, \lambda, \perp) = \{(q_1, \perp)\}$ — середина цепочки, когда на стеке маркер.
8. $\delta(q_0, \lambda, a) = \{(q_1, a)\}$ — середина цепочки, когда на стеке a.
9. $\delta(q_0, \lambda, b) = \{(q_1, b)\}$ — середина цепочки, когда на стеке b.
10. $\delta(q_1, a, a) = \{(q_1, \lambda)\}$ — выталкивает a, если на стеке и на входе a.
11. $\delta(q_1, b, b) = \{(q_1, \lambda)\}$ — выталкивает b, если на стеке и на входе b.
12. $\delta(q_1, \lambda, \perp) = \{(q_2, \perp)\}$ — допускающее, если на стеке маркер.

Этот МП-автомат распознает цепочки вида ww^R, то есть цепочку w и ее обращение w^R. Сначала автомат записывает на стек символы цепочки w, находясь в состоянии q_0. В какой-то момент автомат решает, что он находится в середине цепочки, и переходит в состояние q_1, в котором сравнивает символы на стеке с символами цепочки w^R, одновременно выталкивая символы из стека. При совпадении всех символов цепочек w и w^R автомат обнаруживает на вершине стека маркер дна, и переходит в допускающее состояние q_2.

Данный МП-автомат является недетерминированным, так как функция его переходов содержит пустые переходы. Предсказать момент середины цепочки этот автомат не может, поэтому он порождает копии, пытающиеся распознать возможные ветви, точно так же, как это делает недетерминированный конечный автомат. При входной цепочке "*abba*", например, автомат выполняет следующие шаги, ведущие к допуску:

$(q_0, abba, \perp) \vdash (q_0, bba, a\perp)$
$\vdash (q_0, ba, ba\perp)$
$\vdash (q_1, ba, ba\perp)$
$\vdash (q_1, a, a\perp)$
$\vdash (q_1, \lambda, \perp)$
$\vdash (q_2, \lambda, \perp).$

Одновременно этот автомат выполняет и другие шаги. Следующая последовательность шагов переводит автомат в допускающее состояние при непрочитанном входе:

$(q_0, abba, \perp) \vdash (q_1, abba, \perp)$
$\vdash (q_2, abba, \perp)$

Выполнение спонтанного пустого перехода в состояние q_1 не в середине цепочки приводит к «умиранию» исследуемой ветви, например:

$(q_0, abba, \perp) \vdash (q_0, bba, a\perp)$
$\vdash (q_1, bba, a\perp)$

Автомат может также перенести на стек всю цепочку и «умереть»:

$(q_0, abba, \perp) \vdash (q_0, bba, a\perp)$
$\vdash (q_0, ba, ba\perp)$
$\vdash (q_0, a, bba\perp)$
$\vdash (q_0, \lambda, abba\perp)$
$\vdash (q_1, \lambda, abba\perp)$ ●

3.2.1. Расширенный МП-автомат

Существует вариант МП-автомата, способный заменять на стеке не верхний символ, а цепочку нескольких самых верхних символов другой цепочкой. Формально расширенный МП-автомат, — это семерка

$P = (Q, \Sigma, Z, \delta, q_0, \perp, F)$,

где δ отображает множество $Q\times(\Sigma\cup\{\lambda\})\times Z^*$ на множество $Q\times Z^*$. Это отличие от обычного МП-автомата позволяет расширенному автомату выполнять такты при пустом магазине. Шаг этого автомата выполняется следующим образом:

если $(q_2, \beta)\in\delta(q_1, a, \alpha)$ *то* $(q_1, aw, \alpha) \vdash (q_2, w, \beta)$,

при этом автомат перемещает считывающую головку, удаляя символ *a* со входа, и заменяет на стеке цепочку α цепочкой β.

Пример 3.2. Рассмотрим расширенный МП-автомат

$P = (\{q, p\}, \{a, b\}, \{\perp, a, b, S\}, \delta, q, \perp, \{p\})$,

распознающий язык палиндромов четной длины, функция переходов δ которого задана следующим образом:

1. $\delta(q, a, \lambda) = \{(q, a)\}$
2. $\delta(q, b, \lambda) = \{(q, b)\}$
3. $\delta(q, \lambda, \lambda) = \{(q, S)\}$
4. $\delta(q, \lambda, aSa) = \{(q, S)\}$
5. $\delta(q, \lambda, bSb) = \{(q, S)\}$
6. $\delta(q, \lambda, S\perp) = \{(p, \lambda)\}$

Для цепочки "*abba*" автомат P может выполнить следующие шаги:

$$
\begin{aligned}
(q, abba, \perp) &\vdash (q, bba, a\perp) \\
&\vdash (q, ba, ba\perp) \\
&\vdash (q, ba, ba\perp) \\
&\vdash (q, ba, Sba\perp) \\
&\vdash (q, a, bSba\perp) \\
&\vdash (q, a, Sa\perp) \\
&\vdash (q, \lambda, S\perp) \\
&\vdash (p, \lambda, \lambda)
\end{aligned}
$$

Сначала автомат переносит символы первой половины цепочки на стек, после чего помещает на стек маркер середины цепочки S. Символы второй половины поочередно цепочки помещаются в стек, и при совпадении символов слева и справа от маркера S из стека удаляется цепочка aSa или bSb. Когда на стеке остаются только маркеры S и $\perp$, и автомат удаляет их и переходит в допускающее состояние. ●

3.2.2. Допускание МП-автоматами

Сначала рассмотрим важный аспект поведения МП-автомата, интуитивно понятный. Он заключается в следующих трех принципах:

1) если некоторая последовательность шагов допустима, то она допустима также, если в каждой конфигурации приписать к дну магазина одну и ту же цепочку γ:

если $(q, x, \alpha) \vdash^* (p, y, \beta)$ то $(q, x, \alpha\gamma) \vdash^* (p, y, \beta\gamma)$

2) если некоторая последовательность шагов допустима, то она допустима также, если в каждой конфигурации приписать к входным цепочкам одну и ту же цепочку w:

если $(q, x, \alpha) \vdash^* (p, y, \beta)$ то $(q, xw, \alpha) \vdash^* (p, yw, \beta)$

3) если некоторая последовательность шагов допустима, и некоторый суффикс входной цепочки остался непрочитанным, то этот суффикс можно удалить из каждой конфигурации:

если $(q, xw, \alpha) \vdash^* (p, yw, \beta)$ то $(q, x, \alpha) \vdash^* (p, y, \beta)$.

Символы, которые МП-автомат не читает, не влияют на его функционирование, хотя, возможно, они понадобятся впоследствии.

Конечные автоматы допускают цепочки, переходя в допускающее, заключительное состояние. МП-автоматы могут допускать цепочки как по заключительному состоянию, так и по опустошению магазина. Если автомат допускает цепочку по заключительному состоянию, содержимое магазина в этот момент не имеет значения. Если автомат допускает цепочку по опустошению магазина, маркер дна не требуется для работы, так как он выполняет роль цепочки γ из описанного выше принципа 1.

МП-автоматы из приведенных выше примеров допускают цепочку, переходя в допускающее состояние. Распознаваемый при этом язык:

$L(P) = \{w \mid (q_0, w, \perp) \vdash^* (q, \lambda, \alpha), q \in F\}$.

Некоторые КС-языки таковы, что по окончании распознавания их цепочек магазин опустошается, что позволяет МП-автоматам допускать цепочки по пустому магазину. В этом случае говорят, что МП-автомат определяет язык $L_\lambda(P)$, допускаемый по опустошению магазина:

$L_\lambda(P) = \{w \mid (q_0, w, \alpha) \vdash^* (q, \lambda, \lambda)$, q — какое-то состояние$\}$.

Пример 3.3. МП-автомат $P = (\{q, r\}, \{a\}, \{\perp\}, \delta, q, \perp, \{r\})$ распознает регулярный язык aa^*. Функция δ содержит переходы

$\delta(q, a, \perp) = \{(r, \perp)\}$ и $\delta(r, a, \perp) = \{(r, \perp)\}$.

Этот автомат не может допускать цепочки языка по пустому магазину, в магазин ничего не записывается. Однако небольшая модификация позволит это сделать. Запишем в конец цепочки маркер ⊥, называемый в этом случае *концевым маркером*. Если δ содержит переходы

$\delta(q, a, \perp) = \{(q, \perp)\}$ и $\delta(q, \perp, \perp) = \{(q, \lambda)\}$,

автомат будет допускать цепочки вида $aa^*\perp$ по пустому магазину. Здесь используется описанный выше принцип 2. ●

Доказано, что недетерминированный МП-автомат допускает цепочку по пустому магазину тогда и только тогда, когда существует МП-автомат, допускающий эту цепочку по заключительному состоянию, и наоборот, МП-автомат допускает цепочку по заключительному состоянию тогда и только тогда, когда существует МП-автомат, допускающий эту цепочку по опустошению магазина.

Недетерминированный МП-автомат, допускающий по заключительному состоянию, можно преобразовать в МП-автомат, допускающий по опустошению магазина, и наоборот, МП-автомат, допускающий по пустому магазину, можно преобразовать в МП-автомат, допускающий по заключительному состоянию. У нас нет необходимости делать эти преобразования, но их описание можно найти в [0] и [0].

Далее мы рассмотрим, каким образом связаны МП-автоматы и КС-грамматики.

3.2.3. Недетерминированный нисходящий распознаватель

Пусть задана бесконтекстная грамматика $G = (\Sigma, N, P, S)$, $V = \Sigma \cup N$.

Теорема 3.1. Можно построить такой МП-автомат

$R = (\{q\}, \Sigma, V, \delta, q, S, \varnothing)$, для которого $L_\lambda(R) = L(G)$.

Будем моделировать левосторонние выводы в G, для чего определим функцию δ на основании правил G следующим образом.

1) $(q, \beta) \in \delta(q, \lambda, A)$ для каждого правила $A \to \beta$.

Этот шаг автомата R называется «подбор альтернативы». Если на вершине стека находится нетерминал A, он заменяется цепочкой β тела любого из правил (альтернатив) нетерминала A. Обозреваемый символ на входе игнорируется и считывающая готовка не перемещается.

2) $(q, \lambda) \in \delta(q, a, a)$ для всех $a \in \Sigma$.

Этот шаг называется «выброс» (*pop*). Если терминал на вершине стека совпадает с символом входа, он выталкивается, считывающая головка перемещается к следующему символу. Алгоритм, моделирующий подобный МП-автомат, называют алгоритмом *с подбором альтернатив*.

Докажем, что $A \Rightarrow^m w$ тогда и только тогда, когда $(q, w, A) \vdash^n (q, \lambda, \lambda)$ для некоторых $m \geq 1$ и $n \geq 1$.

Необходимость. Покажем, что если существует вывод $A \Rightarrow^m w$, то автомат R выполнит последовательность шагов $(q, w, A) \vdash^+ (q, \lambda, \lambda)$.

Используем индукцию по m. Если $m = 1$, то вывод из A содержит только терминалы: $A \Rightarrow a_1 \ldots a_k$, $k \geq 0$. Тогда автомат R может выполнить один шаг «подбор альтернативы» и k шагов «выброс»:

$$\begin{aligned}(q, a_1 \ldots a_k, A) &\vdash (q, a_1 \ldots a_k, a_1 \ldots a_k)\\ &\vdash (q, a_2 \ldots a_k, a_2 \ldots a_k)\\ &\vdots\\ &\vdash (q, \lambda, \lambda)\end{aligned}$$

Пусть $m > 1$. Первый шаг вывода должен иметь вид $A \Rightarrow X_1 X_2 \ldots X_k$, где каждый из символов X_i порождает цепочку терминалов x_i за число шагов $m_i < m$. Первый шаг автомата R — выбор альтернативы:

$(q, w, A) \vdash (q, w, X_1 X_2 \ldots X_k)$.

Если $X_i \in N$, то по предположению индукции $X_i \Rightarrow^{m_i} x_i$:

$(q, x_i, X_i) \vdash^* (q, \lambda, \lambda)$.

Если $X_i \in \Sigma$, то выполняется шаг «выброс»:

$(q, x_i, X_i) \vdash (q, \lambda, \lambda)$.

Объединяя эти последовательности шагов работы автомата R, получим $(q, w, \lambda) \vdash^+ (q, \lambda, \lambda)$.

Достаточность. Покажем, что если автомат R выполнит последовательность шагов $(q, w, A) \vdash^n (q, \lambda, \lambda)$, то существует вывод $A \Rightarrow^+ w$.

Используем индукцию по n. Если $n = 1$, то $w = \lambda$, и $(A \to \lambda) \in P$.

Предположим, что утверждение верно для всех $n' < n$. Первый шаг автомата R должен быть «подбор альтернативы» для A:

$(q, w, A) \vdash (q, w, X_1X_2\ldots X_k)$,

где $w = x_1x_2\ldots x_k$, и $(A \to X_1X_2\ldots X_k) \in P$.

Если $X_i \in \Sigma$, то автомат R выполняет шаг «выброс»:

$(q, x_i\ldots x_k, X_i\ldots X_k) \vdash (q, x_{i+1}\ldots x_k, X_{i+1}\ldots X_k)$,

что соответствует порождению $X_i \Rightarrow^0 x_i$.

Если $X_i \in N$, то по предположению индукции

$(q, x_i\ldots x_k, X_i\ldots X_k) \vdash (q, x_{i+1}\ldots x_k, X_{i+1}\ldots X_k)$.

что соответствует порождению $X_i \Rightarrow^+ x_i$.

При этом автомат R выполняет последовательность шагов

$$\begin{aligned}(q, w, A) &\vdash (q, w, X_1X_2\ldots X_k)\\ &\vdash^* (q, w_1, X_2\ldots X_k)\\ &\vdash^* (q, w_2, X_3\ldots X_k)\\ &\vdots\\ &\vdash^* (q, \lambda, \lambda),\end{aligned}$$

а в грамматике существует вывод

$$\begin{aligned}A &\Rightarrow X_1X_2\ldots X_k\\ &\Rightarrow^* x_1X_2\ldots X_k\\ &\vdots\\ &\Rightarrow^* x_1x_2\ldots x_{k-1}X_k\\ &\Rightarrow^* x_1x_2\ldots x_{k-1}x_k = w.\end{aligned}$$

Полагая $A = S$, можно заключить, что $S \Rightarrow^+ w$ тогда и только тогда, когда $(q, w, S) \vdash^+ (q, \lambda, \lambda)$, и $L_\lambda(R) = L(G)$. ●

Пример 3.3. Построим автомат

$R = (\{q\}, \{a, b\}, \{S, a, b\}, \delta, q, S, \varnothing)$

для грамматики

$S \to aSb \mid bSa \mid ab \mid ba$

Определим функцию переходов следующим образом:

$\delta(q, \lambda, S) = \{(q, aSb), (q, bSa), (q, ab), (q, ba)\}$

$\delta(q, a, a) = \{(q, \lambda)\}$

$\delta(q, b, b) = \{(q, \lambda)\}$

Тогда для входной цепочки "*abab*" автомат может выполнить шаги:

$$\begin{aligned}(q, abab, S) &\vdash (q, abab, aSb))\\ &\vdash (q, bab, Sb)\\ &\vdash (q, bab, bab)\\ &\vdash (q, ab, ab)\\ &\vdash (q, b, b)\\ &\vdash (q, \lambda, \lambda)\end{aligned}$$

Цепочка вывода $S \Rightarrow aSb \Rightarrow abab$. ●

3.2.4. Восходящее распознавание

При восходящем распознавании используется обращенное правое порождение, в котором продукции применяются в обратном порядке.

Пусть грамматика имеет следующие правила:

$S \rightarrow aSA \mid b$

$A \rightarrow aS \mid ab$

Построим правосторонний вывод цепочки "*abab*":

$S \Rightarrow_{rm} aSA \Rightarrow_{rm} aSaS \Rightarrow_{rm} aSab \Rightarrow_{rm} abab$

Запишем обращенное правое порождение:

$abab \Rightarrow_{rm} aSab \Rightarrow_{rm} aSaS \Rightarrow_{rm} aSA \Rightarrow_{rm} S$

Посмотрим, как моделируется обращенное правое порождение. Сначала во входной цепочке нужно найти ту ее часть, которая была получена в последнем шаге вывода. Для этого будем переносить символы из цепочки в стек до тех пор, пока вся цепочка последнего примененного правила не окажется в стеке. Последним было применено правило $S \rightarrow b$, при этом был порожден первый слева терминал *b*. Поэтому переносим в стек символы *ab*, при этом *b* находится на вершине.

Теперь заменим терминал *b* нетерминалом *S*, из которого *b* был порожден. Этот процесс называется *свёрткой*, — цепочка правила сворачивается в порождающий ее нетерминал. В результате в стеке окажется цепочка *aS*. Следующим должно быть применено тоже правило $S \rightarrow b$, поэтому переносим в стек еще два символа, и получим цепочку *aSab*. Сворачивая *b* в нетерминал *S*, получим в стеке цепочку *aSaS*. Следующим было применено правило $A \rightarrow aS$, поэтому сворачиваем цепочку *aS* в *A* и получим в стеке *aSA*. Заметим, что сворачивается та цепочка *aS*, которая находится на вершине, а не внутри стека. Полученная цепочка сворачивается в *S*, и процесс завершается, так как на входе больше нет символов и в стеке получен целевой символ грамматики.

Следующая таблица наглядно показывает процесс распознавания.

#	Стек	Вход	Действие
1	λ	*abab*	*перенос*
2	*a*	*bab*	*перенос*
3	*ab*	*ab*	*свертка*
4	*aS*	*ab*	*перенос*
5	*aSa*	*b*	*перенос*
6	*aSab*	λ	*свертка*
7	*aSaS*	λ	*свертка*
8	*aSA*	λ	*свертка*
9	*S*	λ	*допуск*

3.3.5. Основа, активный префикс и обрезка основ

Задача восходящего распознавания заключается в определении той цепочки на вершине стека, которая должна быть свернута. Эту цепочку называют *основой* (в оригинале *handle*, дескриптор). Основа в точности равна телу одного из правил грамматики. Определение основы осложняется тем, что в грамматике может быть несколько правил с равными цепочками тел, а на вершине стека может быть несколько таких цепочек.

Заметим, что вершину стека удобнее располагать справа, а не слева, как в нисходящем распознавании. Суммарная цепочка, находящаяся на стеке и на входе, в любой момент распознавания представляет собой *шаг обращенного правого порождения*. Например, в строке 3 на стеке находится цепочка *ab*, которую можно свернуть в нетерминал *A*, однако при этом получится суммарная цепочка *Aab*, которая не является шагом обращенного правого порождения, и ее дальнейшая свертка не ведет к получению целевого символа. Поэтому *b* является основой, а *ab* — нет.

Основа всегда находится на вершине, а не внутри стека. Например, в строке 7 на стеке находится цепочка *aSaS*, и сворачивается цепочка *aS*, которая находится справа (на вершине).

Цепочка справа от основы содержит нераспознанные терминалы, а цепочка слева — только активные префиксы, образовавшиеся на стеке в результате переносов и сверток. *Активный префикс* (*viable prefix*) — это префикс правосентенциальной формы, правая часть которого всегда находится в пределах правого края крайней справа основы. Перенесенная в стек часть входа является допустимой только тогда, когда она может быть свернута в активный префикс.

Процесс свертки называют *обрезкой основы*. При этом ветки дерева вывода постепенно отсекаются до получения корневого узла, в котором находится целевой символ. Обрезанные основы исчезают, — они становятся другими активными префиксами (рисунок 3.3).

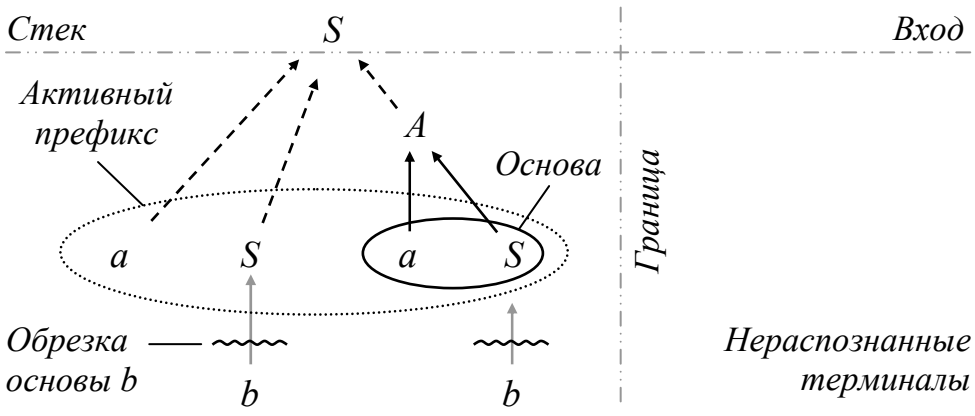

Рисунок 3.1. Основа, активный префикс, обрезка основ

3.3.6. Недетерминированный восходящий распознаватель

Пусть задана бесконтекстная грамматика $G = (\Sigma, N, P, S)$, $V = \Sigma \cup N$.

Теорема 3.2. Можно построить такой расширенный МП-автомат $R = (\{q, r\}, \Sigma, V \cup \{\perp\}, \delta, q, \perp, \{r\})$, для которого $L(R) = L(G)$.

Будем моделировать обращенное правое порождение в G, для чего определим функцию δ на основании правил G следующим образом.

1) $(q, a) \in \delta(q, a, \lambda)\ \forall a \in \Sigma$.

Этот шаг автомата R называется «перенос» или «сдвиг» (*shift*). Текущий терминал на входе переносится на вершину стека, считывающая головка перемещается к следующему символу.

2) $(q, A) \in \delta(q, \lambda, \beta)$, если $(A \to \beta) \in P$.

Этот шаг называется «свёртка» (*reduce*). Цепочка β на вершине стека заменяется нетерминалом A, если в грамматике есть правило $A \to \beta$.

3) $(r, \lambda) \in \delta(q, \lambda, \perp S)$.

Этот шаг переводит автомат R в допускающее состояние, если на стеке находится только целевой символ грамматики и маркер дна.

Алгоритм, моделирующий подобный автомат, называют алгоритмом «перенос-свертка» или «сдвиг-свертка» (*shift-reduce*).

Можно показать, что в процессе выполнения шагов автомат R строит правовыводимые цепочки грамматики G, начиная с цепочки терминалов на входе R, и заканчивая цепочкой S.

Индукцией по n докажем, что

$S \Rightarrow_r^* \alpha A y \Rightarrow_r^n xy$ влечет $(q, xy, \perp) \vdash^* (q, y, \perp\alpha A)$.

Базис. При $n = 0$ автомат не делает одного шага.

Индукция. Предположим, что утверждение верно для $n' < n$. Так как из A выводится цепочка β, можно записать $\alpha A y \Rightarrow_r \alpha\beta y \Rightarrow_r^{n-1} xy$.

Если $\alpha\beta \in \Sigma^*$, то $\alpha\beta = x$ и $(q, xy, \perp) \vdash^* (q, y, \perp\alpha\beta) \vdash (q, y, \perp\alpha A)$.

Если $\alpha\beta \in V^*$, то $\beta = \gamma B z$, где B — самый правый нетерминал. В этом случае из $S \Rightarrow_r^* \alpha\gamma B z y \Rightarrow_r^{n-1} xy$ следует $(q, xy, \perp) \vdash^* (q, zy, \perp\alpha\gamma B)$, и, следовательно, возможно также $(q, zy, \perp\alpha\gamma B) \vdash^* (q, y, \perp\alpha\gamma B z) \vdash (q, y, \perp\alpha A)$.

Запись в виде таблицы более наглядно поясняет сказанное:

#	Стек	Вход	Действие
	$\perp$	xy	*перенос*
	$\perp\alpha\gamma v$	zy	*свертка* $v \to B$
	$\perp\alpha\gamma B$	zy	*перенос*
	$\perp\alpha\gamma B z$	y	*свертка* $\gamma B z \to A$
	$\perp\alpha A$	y	

Полагаем, что утверждение верно для всех n.

Так как $(q, \lambda, \perp S) \vdash (r, \lambda, \lambda)$, заключаем, что $L(R) \subseteq L(G)$.

Теперь докажем, что $(q, xy, \perp) \vdash^n (q, y, \perp\alpha A)$ влечет $\alpha Ay \Rightarrow_r^* xy$.

Базис. При $n = 0$ утверждение выполняется автоматически.

Индукция. Предположим, что утверждение верно для всех $n < m$.

Так как на вершине стека нетерминал, последним шагом автомата R была свертка и можно записать $(q, xy, \perp) \vdash^{m-1} (q, y, \perp\alpha\beta) \vdash (q, y, \perp\alpha A)$, где $(A \to \beta) \in P$. Если цепочка $\alpha\beta$ содержит нетерминал, то по предположению индукции $\alpha\beta y \Rightarrow_r^* xy$, и тогда $\alpha Ay \Rightarrow_r \alpha\beta y \Rightarrow_r^* xy$.

Полагая $A = S$, $(q, w, \perp) \vdash^* (q, \lambda, \perp S)$ влечет $S \Rightarrow_r^* w$. Так как R допускает w тогда и только тогда, когда $(q, w, \perp) \vdash^* (q, \lambda, \perp S) \vdash (r, \lambda, \lambda)$, то, следовательно, $L(G) \subseteq L(R)$, и $L(R) = L(G)$. ●

Пример 3.4. Построим расширенный автомат

$R = (\{q, r\}, \{a, b\}, \{\perp, S, a, b\}, \delta, q, \perp, \{r\})$

для грамматики

$S \to aSb \mid bSa \mid ab \mid ba$

Определим функцию переходов следующим образом:

1) $\delta(q, a, \lambda) = \{(q, a)\}$
2) $\delta(q, b, \lambda) = \{(q, b)\}$
3) $\delta(q, \lambda, aSb) = \{(q, S)\}$
4) $\delta(q, \lambda, bSa) = \{(q, S)\}$
5) $\delta(q, \lambda, ab) = \{(q, S)\}$
6) $\delta(q, \lambda, ba) = \{(q, S)\}$
7) $\delta(q, \lambda, \perp S) = \{(r, \lambda)\}$

Тогда для входной цепочки "*abab*" автомат может выполнить шаги:

$(q, abab, \perp) \vdash (q, bab, \perp a)$
$\vdash (q, ab, \perp ab)$
$\vdash (q, b, \perp aba)$
$\vdash (q, b, \perp aS)$
$\vdash (q, \lambda, \perp aSb)$
$\vdash (q, \lambda, \perp S)$
$\vdash (r, \lambda, \lambda)$

Запись вывода в виде таблицы:

#	Стек	Вход	δ	Действие
1	$\perp$	*abab*	1	*перенос*
2	$\perp a$	*bab*	2	*перенос*
3	$\perp ab$	*ab*	1	*перенос*
4	$\perp aba$	*b*	6	*свертка*
5	$\perp aS$	*b*	2	*перенос*
6	$\perp aSb$	λ	3	*свертка*
9	$\perp S$	λ	7	*к допускающему состоянию*

Цепочка вывода $S \Rightarrow_r aSb \Rightarrow_r abab$. ●

3.3. Детерминированные МП-автоматы

В практическом плане нас интересуют детерминированные автоматы с магазинной памятью, ДМП-автоматы, так как их легче моделировать. Эти автоматы обладают меньшей мощностью, чем недетерминированные МП-автоматы, они распознают только некоторое подмножество бесконтекстных языков, однако класс этих языков чрезвычайно важен для практических приложений. Языки, распознаваемые ДМП-автоматами, будем называть детерминированными КС-языками.

МП-автомат $P = (Q, \Sigma, Z, \delta, q_0, \perp, F)$ является детерминированным, если для каждых $q \in Q$ и $z \in Z$ выполняется одно из двух:

1) $|\delta(q, a, z)| \le 1$ для всех $a \in \Sigma$ и $\delta(q, \lambda, z) = \varnothing$;

2) $\delta(q, a, z) = \varnothing$ для всех $a \in \Sigma$ и $|\delta(q, \lambda, z)| \le 1$.

Таким образом, в любой конфигурации ДМП-автомат может выполнить только один какой-то шаг — либо чтение входа, либо λ-переход.

Пример 3.5. Ранее мы показали, что язык палиндромов четной длины $L_{wwr} = \{ww^R \mid w \in (a + b)^*\}$ распознается недетерминированным МП-автоматом. Оказывается, для этого языка не существует никакого ДМП-автомата. Однако если в середину цепочки ww^R вставить какой-нибудь символ, например, c, тогда для нового языка $L_{wcwr} = \{wcw^R \mid w \in (a + b)^*\}$ ДМП-автомат можно построить.

Определим автомат $P = (\{q_0, q_1, q_2\}, \{a, b, c\}, \{\perp, a, b\}, \delta, q_0, \perp, \{q_2\})$, в котором функция δ определяется следующим образом:

$\delta(q_0, a, \perp) = (q_0, a\perp)$
$\delta(q_0, b, \perp) = (q_0, b\perp)$
$\delta(q_0, a, a) = (q_0, aa)$
$\delta(q_0, a, b) = (q_0, ab)$
$\delta(q_0, b, a) = (q_0, ba)$
$\delta(q_0, b, b) = (q_0, bb)$
$\delta(q_0, c, \perp) = (q_1, \perp)$
$\delta(q_0, c, a) = (q_1, a)$
$\delta(q_0, c, b) = (q_1, b)$
$\delta(q_1, a, a) = (q_2, \lambda)$
$\delta(q_1, b, b) = (q_2, \lambda)$
$\delta(q_2, \lambda, \perp) = (q_0, \perp)$

Этот автомат также сначала помещает в стек символы цепочки w, находясь в состоянии q_0. Обнаружив символ c, автомат переходит в состояние q_1, в котором символы входа сравниваются с символами стека, и если символы совпадают, из стека выталкивается один символ. Так продолжается до тех пор, пока на стеке не окажется маркер дна. Если при этом на входе пусто, автомат переходит в допускающее состояние. ●

3.3.1. Детерминированный нисходящий распознаватель

Можно также построить ДМП-автомат R с опустошением магазина. Класс языков, которые допускаются при этом, ограничен, так как язык должен обладать *префиксным свойством*. Некоторый язык L обладает этим свойством, если ни одна цепочка языка не является префиксом никакой другой цепочки языка. Язык L_{wwr} не обладает префиксным свойством, так как в нем найдется много цепочек с одинаковым префиксом, например, *abba*, *abaaba*, *abbbba*. Язык L_{wcwr}, наоборот, этим свойством обладает. Это легко доказать. Предположим, что wcw^R и xcx^R — две различные цепочки, и wcw^R является префиксом xcx^R. Тогда цепочка wcw^R должна быть короче цепочки xcx^R. В этом случае позиция символа c в цепочке wcw^R совпадает с позицией символа a или символа b в цепочке xcx^R, но тогда wcw^R не является префиксом xcx^R.

Пример 3.6. Построим ДМП-автомат $R = (\{q\}, \{a, b, c\}, \{S, a, b, c\}, \delta, q, S, \varnothing)$ на основе грамматики, описывающей цепочки wcw^R:

$S \rightarrow aSa \mid bSb \mid c$

Функция переходов δ имеет следующий вид:

$\delta(q, a, S) = (q, aSa)$
$\delta(q, b, S) = (q, bSb)$
$\delta(q, c, S) = (q, c)$
$\delta(q, a, a) = (q, \lambda)$
$\delta(q, b, b) = (q, \lambda)$
$\delta(q, c, c) = (q, \lambda)$

В соответствии с этой функцией, правило для S выбирается в зависимости от текущего символа на входе. Если на входе a, выбирается правило $S \rightarrow aSa$, если на входе b, выбирается правило $S \rightarrow bSb$, если на входе c, выбирается правило $S \rightarrow c$.

При распознавании цепочки "*abcba*" автомат R выполнит следующую единственно возможную последовательность шагов:

$(q, abcba, S) \vdash (q, abcba, aSa)$
$\vdash (q, bcba, Sa)$
$\vdash (q, bcba, bSba)$
$\vdash (q, cba, Sba)$
$\vdash (q, cba, cba)$
$\vdash (q, ba, ba)$
$\vdash (q, a, a)$
$\vdash (q, \lambda, \lambda)$

Видно, как автомат сначала переносит цепочку wc на стек, после чего цепочка на стеке совпадает с цепочкой на входе, и символы один за другим удаляются со стека и входа шагами «выброс». ●

3.3.2. Детерминированный восходящий распознаватель

Расширенный МП-автомат $P(Q, \Sigma, Z, \delta, q_0, \perp, F)$ является детерминированным, если выполняются следующие условия:

1) $|\delta(q, a, \gamma)| \le 1$ для всех $q \in Q$, $a \in \Sigma \cup \{\lambda\}$ и $\gamma \in Z$;

2) если $\delta(q, a, \alpha) \ne \varnothing$, $\delta(q, a, \beta) \ne \varnothing$ и $\alpha \ne \beta$, то ни одна из цепочек α и β не является суффиксом другой (при том, что вершина стека справа);

3) если $\delta(q, a, \alpha) \ne \varnothing$ и $\delta(q, \lambda, \beta) \ne \varnothing$, то ни одна из цепочек α и β не является суффиксом другой (при том, что вершина стека справа).

Необходимость первого условия очевидна, — при одних и тех же символе на входе и цепочке на стеке автомат в любом своем состоянии должен выполнять один и тот же шаг. Второе и третье условия гарантируют однозначность цепочки на стеке при чтении входа, или при выборе между чтением входа и λ-переходом.

Пример 3.7. Построим расширенный ДМП-автомат

$R = (\{q, r\}, \{a, b, c\}, \{\perp, S, a, b, c\}, \delta, q, \perp, \{r\})$

на основе грамматики, описывающей цепочки wcw^R:

$S \to aSa \mid bSb \mid c$

Функция переходов δ имеет следующий вид:

$\delta(q, a, \lambda) = (q, a)$
$\delta(q, b, \lambda) = (q, b)$
$\delta(q, c, \lambda) = (q, c)$
$\delta(q, \lambda, aSa) = (q, S)$
$\delta(q, \lambda, bSb) = (q, S)$
$\delta(q, \lambda, c) = (q, S)$
$\delta(q, \lambda, \perp S) = (r, \lambda)$

Для входной цепочки "*abcba*" автомат R выполнит следующую единственно возможную последовательность шагов:

$(q, abcba, \perp) \vdash (q, bcba, \perp a)$
$\vdash (q, cba, \perp ab)$
$\vdash (q, ba, \perp abc)$
$\vdash (q, ba, \perp abS)$
$\vdash (q, a, \perp abSb)$
$\vdash (q, a, \perp aS)$
$\vdash (q, \lambda, \perp aSa)$
$\vdash (q, \lambda, \perp S)$
$\vdash (r, \lambda, \lambda)$

Цепочка вывода:

$S \Rightarrow_r aSa \Rightarrow_r abSba \Rightarrow_r abcba.$

Обращенное правое порождение:

$abcba \Rightarrow_r abSba \Rightarrow_r aSa \Rightarrow_r S.$ ●

3.3.3. Языки ДМП-автоматов

Можно показать, что если L — регулярный язык, то для него существует ДМП-автомат. Заметим, что МП-автомат является обобщением конечного автомата. Поскольку для любого регулярного языка можно построить ДКА, его можно преобразовать в ДМП-автомат трансформацией функции переходов.

Пусть язык L задан конечным автоматом $A = (Q, \Sigma, \delta, q_0, F)$. Тогда ему соответствует ДМП-автомат $R = (Q, \Sigma, \{\perp\}, \delta', q_0, \perp, F)$, где δ' определяется на основе δ следующим образом: переходу ДКА $\delta(q, a) = p$ соответствует переход $\delta'(q, a, \perp) = (p, \perp)$ ДМП-автомата.

Автомат R допускает язык L по заключительному состоянию, так как легко можно показать, что $(q, w, \perp) \vdash^* (f, \lambda, \perp)$, где $f \in F$. Из этого можно заключить, что регулярные языки составляют собственное подмножество языков, допускаемых детерминированными МП-автоматами по заключительному состоянию.

ДМП-автоматы могут допускать языки вроде L_{wcwr}, которые не являются регулярными. С другой стороны, существуют КС-языки вроде L_{wwr}, которые не могут допускаться по заключительному состоянию никаким ДМП-автоматом. Из этого можно заключить, что языки, допускаемые ДМП-автоматами по заключительному состоянию, составляют собственное подмножество КС-языков.

Заметим, что допускание по заключительному состоянию и по пустому магазину для детерминированных МП-автоматов не эквивалентны, в отличие от допускания недетерминированных МП-автоматов. Детерминированные КС-языки, допускаемые по пустому магазину — это те, и только те языки, которые допускаются ДМП-автоматом по заключительному состоянию и обладают префиксным свойством. Например, простой регулярный язык aa^* не допускается никаким ДМП-автоматом с опустошением магазина, — каждая цепочка языка является префиксом любой другой цепочки другой длины (см. также пример 3.3).

Весьма важным свойством детерминированных МП-автоматов является то, что допускаемые ими языки имеют однозначные грамматики. Заметим, что класс детерминированных языков не совпадает с подмножеством КС-языков, не являющихся существенно неоднозначными. Например, язык L_{wwr} имеет однозначную грамматику, но не допускается никаким ДМП-автоматом. Можно показать, что если язык L допускается ДМП-автоматом с опустошением магазина, то L имеет однозначную грамматику. Это следует из того, что ДМП-автомат с опустошением магазина строит единственный левосторонний вывод, а единственность вывода (либо лево, либо правостороннего) является необходимым и достаточным условием однозначности грамматики.

3.4. Свойства КС-языков

Распознаваемые недетерминированным МП-автоматом бесконтекстные языки обладают свойством замкнутости относительно операции подстановки, в которой каждый терминал a_i языка L заменяется цепочками языка L_{a_i}.

Например, если подстановка выполняется в алфавите $\Sigma = \{0, 1\}$, и подстановки определяются как языки $s(0) = \{a^n b^n \mid n \geq 1\}$ и $s(1) = \{aa, bb\}$, то для цепочки $w =$ "01" $s(w) = s(0)s(1) = \{a^n b^n aa, a^n b^n bb\}$.

Бесконтекстные языки замкнуты относительно объединения, конкатенации, итерации, позитивной итерации и гомоморфизма. Доказательство заключается в нахождении подходящей подстановки.

Пусть L_1 и L_2 — бесконтекстные языки в алфавите Σ. Тогда:

1. $L_1 \cup L_2 = s(L)$, где $L = \{0, 1\}$, $s(0) = L_1$, $s(1) = L_2$.
2. $L_1 L_2 = s(L)$, где $L = \{01\}$, $s(0) = L_1$, $s(1) = L_2$.
3. $L_1^* = s(L)$, где $L^* = \{0\}^*$, $s(0) = L_1$.
4. $L_1^+ = s(L)$, где $L^+ = \{0\}^+$, $s(0) = L_1$.
5. $h(L_1) = s(L_1)$, где $s(a) = \{h(a)\}$ $\forall a \in \Sigma$, h — гомоморфизм над Σ.

КС-языки замкнуты также относительно операций:

- обратного гомоморфизма;
- обращения;
- пересечения с регулярным множеством.

КС-языки не замкнуты относительно операций:

- пересечения: язык $L_1 \cap L_2$ не является контекстно-свободным.

Доказательство: если

$L_1 = \{a^n b^n c^m \mid n \geq 1, m \geq 1\}$ — КС-язык,

$L_2 = \{a^m b^n c^n \mid n \geq 1, m \geq 1\}$ — КС-язык,

$L = L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 1\}$, — не КС-язык.

- дополнения: язык $\Sigma^* — L$ не является контекстно-свободным.

Доказательство: в силу закона де Моргана класс языков, замкнутый относительно объединения и дополнения, должен быть замкнутым относительно пересечения.

Детерминированные КС-языки обладают несколько иными свойствами. Эти языки не замкнуты относительно пересечения и объединения, но замкнуты относительно дополнения.

Для КС-языков разрешимы проблема пустоты языка и принадлежности некоторой цепочки КС-языку. Принадлежность цепочки языку можно установить при помощи алгоритма Кока-Янгера-Касами.

Неразрешимы проблемы проверки эквивалентности двух КС-языков, проблема однозначности грамматики, проблема пустоты пересечения двух КС-языков.

3.5. Универсальные анализаторы КС-языков

Далее мы рассмотрим алгоритмы разбора цепочек КС-языков при помощи МП-анализаторов. Если цель распознавателя — установить принадлежность цепочки языку, цель анализатора — получить *разбор*, то есть цепочку π правил грамматики, ведущих к выводу. Поэтому анализаторы строятся исключительно на основе правил грамматик. В этом разделе рассматриваются универсальные анализаторы, которые могут быть применены к достаточно широким классам КС-грамматик.

Разбор с возвратами моделирует левосторонний или правосторонний вывод при помощи недетерминированного МП-автомата. Анализатор последовательно применяет правила грамматики к цепочке, пытаясь вывести ее. Если анализатор заходит в тупик, он откатывается к некоторой предыдущей конфигурации, выполняя возврат. Если удается построить вывод, анализатор выводит цепочку правил. Если полный перебор всех возможных последовательностей конфигураций не позволяет вывести входную цепочку, анализатор сигнализирует об ошибке.

Затем мы рассмотрим два универсальных табличных анализатора, алгоритм Кока-Янгера-Касами и алгоритм Эрли. Эти анализаторы находят разбор, не выполняя возвратов. Первый из алгоритмов используется для проверки принадлежности цепочки КС-языку. Второй алгоритм интересен тем, что находит все возможные разборы.

3.5.1. Нисходящий разбор с возвратами

Нисходящий анализатор с возвратами моделирует левосторонний вывод, используя алгоритм с подбором альтернатив (раздел 3.2.3).

На входе алгоритма КС-грамматика $G = (\Sigma, N, P, S)$, $V = \Sigma \cup N$, без левой рекурсии, и цепочка терминалов $w = a_1 a_2 \ldots a_n$, $n \geq 0$. Правила грамматики занумерованы числами от 1 до некоторого значения.

На выходе алгоритма левый разбор входной цепочки, если таковой существует, в противном случае алгоритм сигнализирует ошибку.

Предварительно альтернативы каждого нетерминала упорядочим и пронумеруем: правила $A \to \gamma_1 \mid \ldots \mid \gamma_k$ запишем в виде $A_1 \to \gamma_1, \ldots, A_k \to \gamma_k$.

Алгоритм использует два стека.

Стек L1 содержит историю выбранных альтернатив и разобранные терминалы, вершина стека справа. Стек L2 содержит текущую левовыводимую цепочку, вершина стека слева.

Конфигурацию алгоритма обозначим четверкой (s, i, α, β), где s — состояние алгоритма (q — нормальное, b — возврат, t — допускающее), i — позиция в цепочке w, α — содержимое стека L1, β — содержимое стека L2. Начальная конфигурация $(q, 1, \lambda, S\perp)$.

Алгоритм 3.1. Нисходящий разбор с возвратами.

Пусть $n = |w|$, функция $top(\mathrm{L})$ показывает символ на стеке L.

Шаг 1. Разрастание.

$(q, i, \alpha, A\beta) \vdash (q, i, \alpha A_1, \gamma_1\beta)$

где γ_1 — первая из альтернатив нетерминала A.

Условия: $s = q$, $top(\mathrm{L2}) \in \mathrm{N}$.

Шаг 2. Успешное сравнение.

$(q, i, \alpha, a\beta) \vdash (q, i+1, \alpha a, \beta)$

Если символ a_i на входе совпадает с символом a на стеке L2, символ a переносим на стек L1, позицию i смещаем вправо.

Условия: $s = q$, $i \le n$, $top(\mathrm{L2}) = a_i$.

Шаг 3. Успешное завершение.

$(q, n+1, \alpha, \perp) \vdash (t, n+1, \alpha, \lambda)$

Разбор завершен, цепочка w выведена, вычисляем цепочку правил π при помощи гомоморфизма h: $h(a) = \lambda\ \forall a \in \Sigma$, $h(A_i)$ = номер правила A_i.

Условия: $s = q$, $i = n+1$, $top(\mathrm{L2}) = \perp$.

Шаг 4. Неуспешное сравнение.

$(q, i, \alpha, a\beta) \vdash (b, i, \alpha, a\beta)$

Символ a_i на входе не совпадает с символом a на стеке L2, переходим к состоянию возврата b.

Условия: $s = q$, $i \le n$, $top(\mathrm{L2}) \ne a_i$.

Шаг 5. Возврат по входу.

$(b, i, \alpha a, \beta) \vdash (b, i-1, \alpha, a\beta)$

для всех $\alpha \in \Sigma$. Входные символы переносим обратно из L1 в L2, позицию i смещаем влево.

Условия: $s = b$, $i \le n$, $top(\mathrm{L1}) \in \Sigma$.

Шаг 6. Другая альтернатива.

$(b, i, \alpha A_j, \gamma_j\beta) \vdash$

а) $(q, i, \alpha A_{j+1}, \gamma_{j+1}\beta)$

если существует альтернатива $j+1$ для A. Выходим из возврата.

б) если $i = 1$, $A = S$, и j — последняя альтернатива S, следующая конфигурация невозможна, сигнализируем ошибку.

в) $(b, i, \alpha, A\beta)$

Альтернативы для A исчерпаны, A_j удаляем со стека L1, на стеке L2 цепочку γ_j заменяем обратно нетерминалом A, продолжаем возврат. ●

Заметим, что порядок нумерации альтернатив имеет значение.

Пример 3.8. Пусть для КС-грамматики правила, порядок правил и альтернативы нетерминалов заданы следующим образом:

(1) $S_1 \to aA$

(2) $A_1 \to Sb$

(3) $A_2 \to b$

Для входной цепочки "*aabb*" алгоритм выполнит следующие шаги:

$(q, 1, \lambda, S\perp)$	
$\vdash (q, 1, S_1, aA\perp)$	1
$\vdash (q, 2, S_1a, A\perp)$	2
$\vdash (q, 2, S_1aA_1, Sb\perp)$	1
$\vdash (q, 2, S_1aA_1S_1, aAb\perp)$	1
$\vdash (q, 3, S_1aA_1S_1a, Ab\perp)$	2
$\vdash (q, 3, S_1aA_1S_1aA_1, Sbb\perp)$	1
$\vdash (q, 3, S_1aA_1S_1aA_1S_1, aAbb\perp)$	1
$\vdash (b, 3, S_1aA_1S_1aA_1S_1, aAbb\perp)$	4
$\vdash (b, 3, S_1aA_1S_1aA_1, Sbb\perp)$	6в
$\vdash (q, 3, S_1aA_1S_1aA_2, bb\perp)$	6а
$\vdash (q, 4, S_1aA_1S_1aA_2b, b\perp)$	2
$\vdash (q, 5, S_1aA_1S_1aA_2bb, \perp)$	2
$\vdash (t, 5, S_1aA_1S_1aA_2bb, \perp)$	3

Цепочка правил $\pi = h(S_1aA_1S_1aA_2bb) = 1213$.

Цепочка левостороннего вывода:

$S \Rightarrow aA \Rightarrow aSb \Rightarrow aaAb \Rightarrow aabb$. ●

3.5.2. Восходящий разбор с возвратами

Восходящий анализатор с возвратами моделирует правосторонний вывод, используя алгоритм «перенос-свертка» (раздел 3.2.6).

На входе алгоритма КС-грамматика $G = (\Sigma, N, P, S)$, $V = \Sigma \cup N$, без пустых и цепных правил, и цепочка терминалов $w = a_1a_2\ldots a_n$, $n \geq 1$. Правила грамматики занумерованы числами от 1 до некоторого значения и произвольным образом упорядочены.

На выходе алгоритма правый разбор входной цепочки, если таковой существует, в противном случае алгоритм сигнализирует ошибку.

Алгоритм также использует два стека. Стек L1 содержит цепочку терминалов и нетерминалов, из которой выводится разобранная часть входной цепочки, вершина стека справа. Стек L2 содержит историю переносов и сверток, вершина стека слева.

Конфигурацию алгоритма обозначим той же четверкой (s, i, α, β), которая используется в алгоритме нисходящего разбора с возвратами, но начальная конфигурация $(q, 1, \perp, \lambda)$.

На каждом шаге алгоритм принимает решение, что выполнить — сдвиг или свертку, а если свертку, то какую цепочку использовать и к какому нетерминалу ее сворачивать (если в грамматике есть несколько одинаковых правых частей правил разных нетерминалов). Заметим, что свертка выполняется при каждой возможности сделать это, однако сначала нужно перенести на стек как минимум один символ.

Алгоритм 3.2. Восходящий разбор с возвратами.

Пусть $n = |w|$.

Шаг 1. Попытка свертки.

$(q, i, \alpha\beta, \gamma) \vdash (q, i, \alpha A, j\gamma)$

где $A \to \beta$ — правило с номером j, и β — первая правая часть правила в принятом порядке правил. Если возможно, повторяем шаг 1.

Шаг 2. Перенос.

$(q, i, \alpha, \gamma) \vdash (q, i+1, \alpha a_i, s\gamma)$

Если $i \neq n+1$, символ с входа переносим в стек L1, в стек L2 запишем символ s (*shift*), обозначающий перенос. Переходим к шагу 1.

Если $i = n+1$, переходим к шагу 3.

Шаг 3. Успешное завершение.

$(q, n+1, \perp S, \gamma) \vdash (t, n+1, \perp S, \gamma)$

Разбор завершен, цепочка w выведена, вычисляем обращенную цепочку правил π при помощи гомоморфизма h: $h(s) = \lambda$, $h(j) = j$.

Шаг 4. Переход к возврату.

$(q, n+1, \alpha, \gamma) \vdash (b, n+1, \alpha, \gamma)$

при условии, что $\alpha \neq \perp S$.

Шаг 5. Возврат.

а) $(b, i, \alpha A, j\gamma) \vdash (q, i, \alpha' B, k\gamma)$

если $A \to \beta$ — правило с номером j, а следующим правилом в принятом порядке правил, правая часть которого является суффиксом цепочки $\alpha\beta$, является правило $B \to \beta'$ с номером k, учитывая, что $\alpha\beta = \alpha'\beta'$ (пытаемся заменить свертку $\beta \to A$ другой сверткой и выходим из возврата).

б) $(b, n+1, \alpha A, j\gamma) \vdash (b, n+1, \alpha\beta, \gamma)$

если $A \to \beta$ — правило с номером j, и для цепочки $\alpha\beta$ нет никакой другой свертки (отменяем свертку $\beta \to A$ и продолжаем возврат).

в) $(b, i, \alpha A, j\gamma) \vdash (q, i+1, \alpha\beta a, s\gamma)$

если $i \neq n+1$, $A \to \beta$ — правило j, и для $\alpha\beta$ нет никакой другой свертки (отменяем свертку $\beta \to A$ и выполняем перенос и выходим из возврата).

г) $(b, i, \alpha a, s\gamma) \vdash (b, i-1, \alpha, \gamma)$

если на вершине стека L2 находится символ s (отменяем перенос и продолжаем возврат). Если шаг невозможен, сигнализируем ошибку. ●

Пример 3.9. Пусть для КС-грамматики правила и порядок правил заданы следующим образом:

(1) $S \to AB$

(2) $A \to b$

(3) $A \to ab$

(4) $B \to c$

Для входной цепочки "*abc*" алгоритм выполнит следующую последовательность шагов:

$(q, 1, \perp, \lambda) \vdash (q, 2, \perp a, s)$ 2
$\vdash (q, 3, \perp ab, ss)$ 2
$\vdash (q, 3, \perp aA, 2ss)$ 1
$\vdash (q, 4, \perp aAc, s2ss)$ 2
$\vdash (q, 4, \perp aAB, 4s2ss)$ 1
$\vdash (q, 4, \perp aS, 14s2ss)$ 1
$\vdash (b, 4, \perp aS, 14s2ss)$ 4
$\vdash (b, 4, \perp aAB, 4s2ss)$ 5б
$\vdash (b, 4, \perp aAc, s2ss)$ 5б
$\vdash (b, 3, \perp aA, 2ss)$ 5г
$\vdash (q, 3, \perp A, 3ss)$ 5а
$\vdash (q, 4, \perp Ac, s3ss)$ 2
$\vdash (q, 4, \perp AB, 4s3ss)$ 1
$\vdash (q, 4, \perp S, 14s3ss)$ 1
$\vdash (t, 4, \perp S, 14s3ss)$ 3

Обращенная цепочка правил $\pi = h(14s3ss) = 143$.

Алгоритм строит обращенное правое порождение, цепочка вывода: $S \Rightarrow_r AB \Rightarrow_r Ac \Rightarrow_r abc$. ●

Рассмотренные алгоритмы с возвратами имеют экспоненциальную зависимость времени разбора от длины входной цепочки в худшем случае. В лучшем случае эта зависимость может быть линейной, но только на определенных цепочках и при определенном порядке правил. Их единственное достоинство заключается в применимости к большому классу КС-грамматик. Для создания компиляторов они мало пригодны, хотя некоторые ранние компиляторы использовали их.

3.5.3. Алгоритм Кока-Янгера-Касами

Этот алгоритм разбора независимо разработали Дж. Кок, Д. Янгер и Т. Касами в 1965-67 годах. Его иногда называют алгоритмом *CYK* по первым буквам фамилий Cocke, Younger, Kasami.

Алгоритм позволяет проверить принадлежность цепочки КС-языку за время $O(n^3)$ при требуемой памяти $O(n^2)$, n — длина цепочки. Для практических целей он малопригоден, как и алгоритмы с возвратами.

Пусть есть КС-грамматика в нормальной форме Хомского без пустых правил $G = (\Sigma, N, P, S)$, $V = \Sigma \cup N$, и исследуемая цепочка терминалов $w = a_1 a_2 \ldots a_n$, $n \geq 1$, $n = |w|$. Для данной грамматики G и данной цепочки w алгоритм строит треугольную таблицу разбора $T[n \times n]$ в которой нетерминал A принадлежит $T[i, j]$ тогда и только тогда, когда существует вывод $A \Rightarrow^+ x_i x_{i+1} \ldots x_{i+j-1}$ (из A выводятся j входных символов, начиная с позиции i). Тогда вывод $S \Rightarrow^* w$ существует, если $S \in T[1, n]$.

Алгоритм 3.3. Алгоритм Кока-Янгера-Касами.

Следующие два цикла строят таблицу T. В первом цикле находятся нетерминалы, из которых выводятся терминалы цепочки w. Во втором цикле ищутся нетерминалы, из которых выводится пара нетерминалов, уже записанных в таблицу:

for $i := 1$ ***to*** n ***do*** (* *первый цикл* *)
    $A \in \mathrm{T}[i, 1]$ *если* $A \to a_i \in \mathrm{P}$
for $j := 2$ ***to*** n ***do*** (* *второй цикл* *)
    for $i := 1$ ***to*** $n - j + 1$ ***do***
        for $k := 1$ ***to*** $j - 1$ ***do***
            $A \in \mathrm{T}[i, j]$ *если* $B \in \mathrm{T}[i, k]$, $C \in \mathrm{T}[i + k, j - k]$ *и* $A \to BC \in \mathrm{P}$

Если после построения таблицы $S \in \mathrm{T}[1, n]$, разбор можно получить при помощи вызова $R(1, n, S)$ следующей рекурсивной процедуры.

Процедура $R(i, j, A)$:

- если $j = 1$ и $(A \to a_i) \in \mathrm{P}$, то выдать номер правила $A \to a_i$.
- если $j > 1$, то взять k, меньшее из диапазона $1 \ldots j - 1$, для которого $(A \to BC) \in \mathrm{P}$, где $B \in \mathrm{T}[i, k]$ и $C \in \mathrm{T}[i + k, j - k]$, выдать номер правила, затем рекурсивно вызвать $R(i, k, B)$ и $R(i + k, j - k, C)$. ●

Пример 3.10. Пусть КС-грамматика в нормальной форме Хомского имеет следующие правила:

$S \to^{(1)} AS \mid {}^{(2)} a$
$A \to^{(3)} BC \mid {}^{(4)} b$
$B \to^{(5)} b$
$C \to^{(6)} SA$

Для цепочки $w = baba$ алгоритм во втором цикле найдет правила:
для $i = 1, j = 2, k = 1$: $S \to AS$, где $S \in \mathrm{T}[1, 2]$, $A \in \mathrm{T}[1, 1]$, $S \in \mathrm{T}[2, 1]$
для $i = 2, j = 2, k = 1$: $C \to SA$, где $C \in \mathrm{T}[2, 2]$, $S \in \mathrm{T}[2, 1]$, $A \in \mathrm{T}[3, 1]$
для $i = 3, j = 2, k = 1$: $S \to AS$, где $S \in \mathrm{T}[3, 2]$, $A \in \mathrm{T}[3, 1]$, $S \in \mathrm{T}[4, 1]$
для $i = 1, j = 3, k = 1$: $A \to BC$, где $A \in \mathrm{T}[1, 3]$, $B \in \mathrm{T}[1, 1]$, $C \in \mathrm{T}[2, 2]$
для $i = 1, j = 3, k = 2$: $C \to SA$, где $C \in \mathrm{T}[1, 3]$, $S \in \mathrm{T}[1, 2]$, $A \in \mathrm{T}[3, 1]$
для $i = 1, j = 4, k = 3$: $S \to AS$, где $S \in \mathrm{T}[1, 4]$, $A \in \mathrm{T}[1, 3]$, $S \in \mathrm{T}[1, 1]$
и построит следующую таблицу:

$j = 4$	S			
$j = 3$	A, C			
$j = 2$	S	C	S	
$j = 1$	A, B	S	A, B	S
цепочка	b	a	b	a
	$i = 1$	$i = 2$	$i = 3$	$i = 4$

В таблице существует дерево вывода, так как $S \in \mathrm{T}[1, 4]$.

Разбор 1356242 выдает последовательность вызовов процедуры R:
$R(1, 4, S)$ выдает 1, вызывает $R(1, 3, A)$, $R(4, 1, S)$.
$R(1, 3, A)$ выдает 3, вызывает $R(1, 1, B)$, $R(2, 2, C)$.
$R(1, 1, B)$ выдает 5.
$R(2, 2, C)$ выдает 6, вызывает $R(2, 1, S)$, $R(3, 1, A)$.
$R(2, 1, S)$ выдает 2.
$R(3, 1, A)$ выдает 4.
$R(4, 1, S)$ выдает 2.
Цепочка левостороннего вывода:
$S \Rightarrow_1 AS \Rightarrow_3 BCS \Rightarrow_5 bCS \Rightarrow_6 bSAS \Rightarrow_2 baAS \Rightarrow_4 babS \Rightarrow_2 baba$. ●

3.5.4. Алгоритм Эрли

Алгоритм Эрли (*Jay Earley*, 1968) для КС-грамматики $G = (\Sigma, N, P, S)$, $V = \Sigma \cup N$, и входной цепочки терминалов $w = a_1 a_2 \ldots a_n$, $n = |w|$, за время $O(n^3)$ и используя память объемом $O(n^2)$, строит последовательность списков ситуаций $I_0, \ldots, I_j, \ldots, I_n$, $0 \le j \le n$. Ситуация представляет собой запись вида $[A \to X_1 \ldots X_k \bullet X_{k+1} \ldots X_m, i]$, при этом в грамматике есть правило $A \to X_1 \ldots X_m$, $X_k \in V$, $0 \le k \le m$, применимое к цепочке w. Ситуация — это позиция в правиле, в которой часть правила до точки • выводит часть w до позиции j. Номер списка j — это позиция в цепочке w, а i — позиция, в которой ситуация применима. Позицию обозначает метасимвол $\bullet \notin V$, он может находиться в начале, в середине или в конце цепочки w. Для правила вида $A \to \lambda$ ситуация имеет вид $[A \to \bullet, i]$.

Для каждой позиции j от 0 до n ситуация $[A \to \alpha \bullet \beta, i]$, $0 \le i \le j$, входит в I_j тогда и только тогда, когда $S \Rightarrow^* \gamma A \delta$, $\gamma \Rightarrow^* a_1 \ldots a_i$, и $\alpha \Rightarrow^* a_{i+1} \ldots a_j$.

Вывод $S \Rightarrow^* w$ существует тогда и только тогда, когда в список I_n входит ситуация вида $[S \to \alpha \bullet, 0]$.

Алгоритм 3.4. Алгоритм Эрли.

Шаг 1. Построение списка I_0.

а) Включим в I_0 ситуацию $[S \to \bullet \alpha, 0]$ $\forall (S \to \alpha) \in P$.

Следующие правила повторяем, пока возможно:

б) Если $[A \to \alpha \bullet B \beta, 0] \in I_0$, включим в I_0 $[B \to \bullet \gamma, 0]$ $\forall (B \to \gamma) \in P$.

в) Если $[B \to \gamma \bullet, 0] \in I_0$, включим в I_0 $[A \to \alpha B \bullet \beta, 0]$ $\forall [A \to \alpha \bullet B \beta, 0] \in I_0$.

Шаг 2. Построение списков I_j, $j > 0$.

Пусть построены списки $I_0, \ldots, I_{j-1}$.

г) Включим в I_j ситуацию $[B \to \alpha a \bullet \beta, i]$ $\forall [B \to \alpha \bullet a \beta, i] \in I_{j-1}$, если $a = a_j$.

Следующие правила повторяем, пока возможно:

д) Если $[A \to \alpha \bullet, i] \in I_j$, включим в I_j $[B \to \alpha A \bullet \beta, k]$ $\forall [B \to \alpha \bullet A \beta, k] \in I_i$.

е) Если $[A \to \alpha \bullet B \beta, i] \in I_j$, включим в I_j $[B \to \bullet \gamma, j]$ $\forall (B \to \gamma) \in P$.

Шаг 3. Если $[S \to \alpha \bullet, 0] \in I_n$, то можно получить разборы.

Разборы вычисляет рекурсивная процедура R, которая применяется ко всем ситуациям вида $[S\to\alpha\bullet, 0]$ списка I_n, при $j = n$.

Процедура $R([A\to\alpha\bullet, i], j)$ выполняется следующим образом:

1) Если h — номер правила $A\to\alpha$, $\pi = \pi h$.

2) Пусть $\alpha = X_1\ldots X_m$. Положить $k = m$, $l = j$.

3. Пока $k \neq 0$, повторять шаги:

а) если $X_k\in\Sigma$, принять $k = k - 1$, $l = l - 1$;

б) если $X_k\in N$, найти в списке I_l ситуацию $[X_k\to\gamma\bullet, r]$, для которой в списке I_r есть ситуация $[A\to X_1\ldots X_{k-1}\bullet X_k\ldots X_m, i]$, вызвать $R([X_k\to\gamma\bullet, r], l)$, после чего принять $k = k - 1$, $l = r$. ●

Пример 3.11. Пусть КС-грамматика имеет следующие правила:

$S\to^{(1)} aA \mid {}^{(2)}\lambda$

$A\to^{(3)} Sb \mid {}^{(4)} b$

Для входной цепочки $w = aabb$ алгоритм построит списки ситуаций, приведенные в следующей таблице:

I_0, $\bullet aabb$	I_1, $a\bullet abb$	I_2, $aa\bullet bb$	I_3, $aab\bullet b$	I_4, $aabb\bullet$
1$[S\to\bullet aA, 0]$ 2$[S\to\bullet, 0]$	3$[S\to a\bullet A, 0]$ 4$[A\to\bullet Sb, 1]$ 5$[A\to\bullet b, 1]$ 6$[S\to\bullet aA, 1]$ 7$[S\to\bullet, 1]$ 8$[A\to S\bullet b, 1]$	9$[S\to a\bullet A, 1]$ 10$[A\to\bullet Sb, 2]$ 11$[A\to\bullet b, 2]$ 12$[S\to\bullet aA, 2]$ 13$[S\to\bullet, 2]$ 14$[A\to S\bullet b, 2]$	15$[A\to b\bullet, 2]$ 16$[A\to Sb\bullet, 2]$ 17$[S\to aA\bullet, 1]$ 18$[A\to S\bullet b, 1]$	19$[A\to Sb\bullet, 1]$ 20$[S\to aA\bullet, 0]$

В список I_0 попадают две ситуации в соответствии с правилом а).

По правилу г) в список I_1 переходит ситуация 1, так как символ 1 цепочки w совпадает с первым символом правила ситуации, и становится ситуацией 3 после перехода точки через a. В ситуации 3 точка перед нетерминалом A, поэтому по правилу е) в I_1 входят также ситуации 4 и 5. В ситуации 4 точка перед нетерминалом S, поэтому по правилу е) в I_1 входят также ситуации 6 и 7. По правилу д) в I_1 входит ситуация 8.

По правилу г) в список I_2 переходит только ситуация 6 и становится ситуацией 9. Как в предыдущем случае, по правилу е) в I_2 добавляются сначала ситуации 10 и 11, а затем ситуации 12 и 13. По правилу д) добавляется ситуация 14.

По правилу г) в список I_3 переходят две ситуации, номера 11 и 14, так как первые символы правил этих ситуаций совпадают с символом 3 цепочки w. Они становятся ситуациями 15 и 16. В этих ситуациях точка находится в конце правил, и через нетерминалы этих правил можно перейти в тех ситуациях, где точка находится перед этими нетерминалами, по правилу д). Так в списке I_3 появляется ситуация 17, перешедшая из ситуации 9 списка I_2. В ситуации 17 точка в конце правила, поэтому

опять по правилу д) в I_3 переходит ситуация 4 списка I_1. Она становится ситуацией 18.

Ситуация 18 по правилу г) переходит в список I_4 и становится ситуацией 19. В этой ситуации точка в конце, поэтому по правилу д) в I_4 переходит ситуация 3 списка I_1 и становится искомой ситуацией 20.

Так как $[S \to aA\bullet, 0] \in I_4$, вывод цепочки существует.

Вычислим разборы π, вызывая $R([S \to aA\bullet, 0], 4)$.

1) $\pi = 1$.

2) $k = 2$, $l = 4$.

3.б) $X_2 = A$. Вызываем $R([A \to Sb\bullet, 1], 4)$.

1) $\pi = 13$.

2) $k = 2$, $l = 4$.

3.а) $X_2 = b$. $k = 1$, $l = 3$.

3.б) $X_1 = S$. Вызываем $R([S \to aA\bullet, 1], 3)$.

1) $\pi = 131$.

2) $k = 2$, $l = 3$.

3.б) $X_2 = A$. Вызываем $R([A \to b\bullet, 2], 3)$.

1) $\pi = 1314$.

2) $k = 1$, $l = 3$.

3.а) $X_1 = b$. $k = 0$, $l = 2$.

$k = 1$, $l = 2$.

3.а) $X_1 = a$. $k = 0$, $l = 1$.

$k = 1$, $l = 1$.

3.а) $X_1 = a$. $k = 0$, $l = 0$.

В списке I_3 есть еще одна подходящая ситуация, поэтому:

3.б) Вызываем $R([A \to Sb\bullet, 2], 3)$.

1) $\pi = 1313$.

2) $k = 2$, $l = 3$.

3.а) $X_2 = b$. $k = 1$, $l = 2$.

3.б) $X_1 = S$. Вызываем $R([S \to \bullet, 2], 2)$.

1) $\pi = 13132$.

2) $k = 0$, $l = 2$.

$k = 1$, $l = 2$.

$k = 1$, $l = 1$.

3.а) $X_1 = a$. $k = 0$, $l = 0$.

Таким образом, мы получили два разбора, $\pi = 1314$ и $\pi = 13132$.

Цепочка вывода для первого разбора:

$S \Rightarrow aA \Rightarrow aSb \Rightarrow aaAb \Rightarrow aabb$

Цепочка вывода для второго разбора:

$S \Rightarrow aA \Rightarrow aSb \Rightarrow aaAb \Rightarrow aaSbb \Rightarrow aabb$ ●

3.6. Метод рекурсивного спуска

Далее мы рассмотрим основные методы нисходящего и восходящего анализа, которые могут быть применены для построения трансляторов. Эти методы распознают детерминированные КС-языки определенного класса, и каждый из них накладывает свои ограничения на грамматику.

Метод рекурсивного спуска (*recursive descent method*) является простым и эффективным методом нисходящего распознавания цепочек КС-языков, исторически одним из первых. Он используется в основном при ручном программировании трансляторов простых языков.

В рекурсивном спуске каждый нетерминальный символ описывается функцией, которая разбирает все правила нетерминала. Поскольку правила могут содержать нетерминалы, в процессе разбора правил одного нетерминала вызываются функции разбора других нетерминалов, что приводит к возникновению рекурсии. Это обстоятельство дало повод для названия метода. Заметим, что роль магазина МП-автомата в этом методе выполняет стек потока, реализующего функции.

Не каждая грамматика может быть разобрана методом рекурсивного спуска. Прежде всего, в грамматике не может быть левой рекурсии. Это просто объясняется. Предположим, грамматика имеет правило $A \rightarrow A\alpha$. В этом случае должна быть написана примерно следующая функция:

```
void A() {
    A();
    // разбор цепочки α
}
```

Очевидно, что вызов этой функции приведет к немедленному зацикливанию, переполнению стека потока и выбросу исключения. Заметим, что рекурсия может быть неявной и возникать через несколько функций.

Другие ограничения на грамматику зависят от того, какие действия должен выполнять транслятор. Если эти действия привязаны к правилам грамматики, то разбор правил некоторого нетерминала A должен определять правило по его первым символам. Например, в грамматике есть правила $A \rightarrow aBb$ и $A \rightarrow aBc$. Мы не можем определить правило для A до тех пор, пока не будет распознан последний символ правила. Поэтому невозможно установить правильную последовательность применения правил для нетерминалов A и B, потому что сначала мы применяем правило для B, а затем узнаем, какое правило было применено для A. Если же дальнейшие действия транслятора не имеют привязки к применяемому правилу, и зависят только от последовательности символов цепочки, то распознать эту последовательность не составляет труда. Таким образом, если нужно установить последовательность применения правил, на грамматику накладывается следующее ограничение.

Пусть есть КС-грамматика $G = (\Sigma, N, P, S)$, $V = \Sigma \cup N$. Если в G есть несколько правил для некоторого нетерминала A:

$A \to \alpha_1\beta\gamma_1 \mid \alpha_2\beta\gamma_2 \mid \ldots \mid \alpha_k\beta\gamma_k,\ k > 1, \alpha_i \in \Sigma^+, \beta \in N^+, \gamma_i \in V^*,$

то должно выполняться $\alpha_i \neq \alpha_j$ для всех $i \neq j$. В этом случае правило для A можно определить до первого нетерминала цепочки β.

На практике на грамматику накладывают еще более жесткое ограничение, а именно, — все правила для A должны начинаться с разных терминалов. Тогда если для нетерминала A есть несколько правил:

$A \to a_1\beta_1 \mid a_2\beta_2 \mid \ldots \mid a_k\beta_k,\ k > 1, a_i \in \Sigma, \beta_i \in V^*,$

то должно выполняться $a_i \neq a_j$ для всех $i \neq j$.

Это ограничение сужает класс грамматик и языков, разбираемых рекурсивным спуском, но гарантирует детерминизм разбора. Отметим также, что в грамматике не должно быть пустых правил, а для выполнения жесткого ограничения иногда нужно применить левую факторизацию.

Пример 3.12. Пусть КС-грамматика имеет следующие правила:

$$S \to^{(1)} viR$$
$$R \to^{(2)} ,iR \mid^{(3)} :t; \qquad (3.5)$$

Грамматика описывает объявление переменных в стиле языка Pascal:

```
var a, b, c : integer;
```

Для этой грамматики выполняется жесткое ограничение, и ее можно использовать для рекурсивного спуска. Тогда должны быть определены две функции, поскольку в грамматике два нетерминала.

Первая функция разбирает два правила для R, она рекурсивная:

```
int R() {
    char c = next_token();
    if (c == ',') {
        printf("2");
        if (next_token() != 'i') return 0;
        return R();
    } else if (c == ':') {
        printf("3");
        if (next_token() != 't') return 0;
        if (next_token() != ';') return 0;
        return 1;
    } else return 0;
}
```

Вторая функция разбирает правило для целевого символа:

```
int S() {
    printf("1");
    if (next_token() != 'v') return 0;
    if (next_token() != 'i') return 0;
    return R();
}
```

Для полноты программы нужна также основная функция:

```
void main() {
    if (S()) printf("\nSuccess\n"); else printf("\nFailure\n");
}
```

В программе должна быть определена также входная цепочка, позиция в ней и функция next_token, возвращающая очередной символ. ●

3.6.1. Расширенное применение рекурсивного спуска

Рекурсивный спуск можно применять всегда, когда удается выполнить детерминированный разбор и действия, связанные с правилами грамматики. Рассмотрим классический пример применения рекурсивного спуска для разбора выражений по грамматике G_0:

$E \rightarrow E+T \mid T$

$T \rightarrow T*P \mid P$

$P \rightarrow a \mid (E)$

Поскольку эта грамматика имеет левую рекурсию, формально она не подходит для рекурсивного спуска. Однако если рассмотреть выводы, например, из нетерминала T:

$T \Rightarrow T*P \Rightarrow T*P*P \Rightarrow P*P*P \ldots$

то можно заметить, что фактически нетерминал T есть последовательность нетерминалов P, соединенных знаками операции умножения.

Тогда разобрать вход можно, если вместо правил $T \rightarrow T*P$ и $T \rightarrow P$ использовать правило $T \rightarrow P*P$, и, анализируя не первый, а второй символ правила, решать, какое правило применить. Если второй символ является знаком операции, применяется правило $T \rightarrow T*P$, а если нет, то $T \rightarrow P$.

Правила для E имеют ту же форму, что и правила для T, поэтому все приведенные выше суждения применимы к ним. Описываемый метод вычисления выражений широко известен и описан во множестве источников, мы лишь приведем пример практической реализации.

Начальная часть программы описывает входную цепочку, позицию в ней и объявляет функцию выражения E:

```
#include <stdio.h>
char w[] = "(2+3)*4";
int pos = 0;
int E(int & value);
```

Далее следует функция, возвращающая очередной символ входа, а также две вспомогательные функции, удостоверяющие, что на входе находится знак определенной операции:

```
char next_token() { return w[pos++]; }
char is_plus() { if (w[pos] == '+') return 1; return 0; }
char is_mult() { if (w[pos] == '*') return 1; return 0; }
```

Функция нетерминала *P* возвращает число через параметр value:

```
int P(int & value) {
    char c = next_token();
    if (c == '+' || c == '*') return 0;
    if (c == '(') {
        if (!E(value)) return 0;
        if (next_token() != ')') return 0;
    } else value = c - '0'; // число - только одна цифра
    return 1;
}
```

Функции нетерминалов *T* и *E* проверяют второй символ правила при помощи вспомогательных функций is_mult и is_plus:

```
int T(int & value) {
    int mul = 0;
    if (!P(value)) return 0;
    while (is_mult()) {
        next_token();
        if (!P(mul)) return 0;
        value *= mul;
    }
    return 1;
}
int E(int & value) {
    int add = 0;
    if (!T(value)) return 0;
    while (is_plus()) {
        next_token();
        if (!T(add)) return 0;
        value += add;
    }
    return 1;
}
```

Основная функция в конце программы объявляет переменную для результата, и вызывает функцию для выражения:

```
void main() {
    int result = 0;
    if (E(result)) printf("%d\n", result);
    else printf("Failure\n");
}
```

Если функция для выражения завершается успешно, программа выводит результат вычисления. Описанное применение метода рекурсивного спуска иногда называют его расширением. Отметим, что получить разбор в виде цепочки примененных правил в этом случае сложно в силу описанных ранее причин. Тем не менее, сделать это можно, если записывать номера правил в цепочку с разных ее концов, а также накапливая цепочки правил во вспомогательных цепочках. Примеры этого можно найти в различных источниках.

3.7. Предиктивный анализ

В этом разделе мы рассмотрим один из самых красивых методов детерминированного разбора, называемый предиктивным анализом. Этот метод интуитивно понятен и естественен, а в отличие от метода рекурсивного спуска, допускает большее подмножество КС-языков.

В основе метода лежит алгоритм с подбором альтернатив (см. 3.2.3), выполняющий нисходящий разбор на основе правил грамматики, выбирая подходящее правило для нетерминала A на стеке МП-автомата. Для детерминированного разбора этот выбор должен быть однозначным, а информация, которая доступна в момент выбора — это распознанная часть входа, k первых символов нераспознанной части входа, и правила грамматики. Если этой информации достаточно для выбора одного из правил для A, то будем говорить, что грамматика обладает свойством предсказуемости (предиктивности).

3.7.1. Свойство предсказуемости

Пример 3.13. Рассмотрим левосторонний вывод цепочки baa в грамматике, порождающей регулярный язык $(ba)^*(a + bb)$:

$S \to {}^{(1)} bA \mid {}^{(2)} a$
$A \to {}^{(3)} aS \mid {}^{(4)} b$

Шаги разбора представим в виде таблицы, в которой столбец «Вход» показывает нераспознанную часть входа, столбец «Стек» показывает содержимое стека, столбец «Шаг» содержит номер выбранного правила для нетерминала на стеке, или слова «выброс» и «допуск»:

#	Вход	Стек	Шаг
1	baa	S	1
2	baa	bA	выброс
3	aa	A	3
4	a	aS	выброс
5	a	S	2
6	a	a	выброс
7	λ	λ	допуск

Каждый раз, когда на стеке нетерминал, правило выбирается однозначно потому, что первый символ нераспознанной части входа совпадает с первым символом правила. Будем говорить в этом случае, что грамматика является 1-предиктивной (1-предсказуемой). Заметим, что эта грамматика удовлетворяет жестким требованиям, предъявляемым к грамматикам для метода рекурсивного спуска. ●

Пример 3.14. Есть похожая грамматика, которая предиктивной не является, она порождает регулярный язык $b + (ba)^*(1 + b)$:

$S \rightarrow bA \mid b$
$A \rightarrow aS \mid a$

Однозначный выбор правила для нетерминалов этой грамматики невозможен, потому что все правила этих нетерминалов начинаются с одинаковых символов, грамматика не является предиктивной.

Стоит также рассмотреть языки, порождаемые этими грамматиками. Как известно, левосторонний детерминированный разбор с опустошением магазина возможен только для языков, обладающих префиксным свойством. Вспомним, что язык имеет префиксное свойство, если никакая цепочка языка не является префиксом никакой другой цепочки. Язык первой грамматики этим свойством обладает, а язык второй грамматики — нет. Легко видеть, что вторая грамматика порождает цепочки *ba* и *bab*, и первая цепочка является префиксом второй.

Иначе говоря, для языка, порождаемого второй грамматикой, не существует никакой грамматики, обладающей свойством предиктивности. Из этого можно сделать вывод, что класс языков, распознаваемых предиктивным анализом, включает в себя языки, обладающие префиксным свойством, для которых найдена грамматика, обладающая свойством предиктивности. Однако язык, не обладающий префиксным свойством, можно попробовать преобразовать в язык, который этим свойством обладает (пример 3.3). Для этого в конец каждой цепочки входа припишем концевой маркер $\perp$.

Тогда для рассматриваемого языка $b + (ba)^*(1 + b)$ можно найти следующую 1-предиктивную грамматику:

$S \rightarrow {}^{(1)} bA$
$A \rightarrow {}^{(2)} aB \mid {}^{(3)} \perp$
$B \rightarrow {}^{(4)} bA \mid {}^{(5)} \perp$

В следующей таблице показано, как разбирается цепочка $bab\perp$.

#	Вход	Стек	Шаг
1	$bab\perp$	S	1
2	$bab\perp$	bA	выброс
3	$ab\perp$	A	2
4	$a\perp$	aB	выброс
5	$\perp$	B	2
6	$\perp$	$\perp$	выброс
7	λ	λ	допуск

Разбор в этом случае ведет к опустошению стека. ●

Пример 3.15. Покажем теперь, что класс предиктивных грамматик шире класса грамматик, разбираемых рекурсивным спуском. Рассмотрим грамматику, порождающую язык $(a + 1)(ba)^*d + (b + 1)(ab)^*c$:

$S \rightarrow A \mid B$
$A \rightarrow aB \mid c$
$B \rightarrow bA \mid d$

Эта грамматика не подходит для рекурсивного спуска, потому что невозможно выбрать правило для S. Она не позволяет также выбрать правило, рассматривая только символы нераспознанной части входа.

Однако рассмотрим выводы из A и из B. Цепочки, выводимые из A, всегда начинаются символом a или c. Цепочки, выводимые из B, всегда начинаются символом b или d. Если на этапе анализа грамматики сформировать множества первых терминалов, которые выводятся из A и B, то на этапе разбора можно сделать выбор, сравнивая терминалы входа с терминалами этих множеств. ●

Пример 3.16. Рассмотрим грамматику, порождающую язык aa^*b:

$S \rightarrow aS \mid ab$

Грамматика не является 1-предиктивной, она 2-предиктивная. В ней выводы из S начинаются с aa или с ab, и на каждом шаге вывода можно выбрать единственное правило, рассматривая два символа входа.

Следующая 1-предиктивная грамматика тоже порождает язык aa^*b:

$S \rightarrow^{(1)} aAb$
$A \rightarrow^{(2)} aA \mid^{(3)} \lambda$

Рассмотрим, как выводится в этой грамматике цепочка ab:

#	Вход	Стек	Шаг
1	ab	S	1
2	ab	aAb	выброс
3	b	Ab	3
4	b	b	выброс
5	λ	λ	допуск

В строке 3 применяется пустое правило 3 потому, что в выводах за A может следовать только терминал b. Заметим, что множество терминалов, следующих в выводе непосредственно за A, здесь не пересекается с множеством терминалов, которыми начинаются выводы из A. ●

Пример 3.17. Следующая грамматика не является предиктивной:

$S \rightarrow^{(1)} aSa \mid^{(2)} \lambda$

Здесь за S может следовать только a, и однозначно выбрать правило для S нельзя — множество первых терминалов и множество терминалов, которые могут следовать за нетерминалом S, пересекаются. ●

3.7.2. Множества FIRST и FOLLOW

Есть два важных множества, используемых для анализа грамматик синтаксического разбора. Как было показано в предыдущем разделе, это множество первых терминалов, которые выводятся из некоторых цепочек, и множество терминалов, которые в выводах могут следовать за некоторым нетерминалом. Для вычисления этих множеств используются функции FIRST и FOLLOW соответственно. Часто говорят, что FIRST и FOLLOW — это не функции, а возвращаемые ими множества.

Пусть есть грамматика $G = (\Sigma, N, P, S)$.

Тогда $FIRST_k(\alpha)$ возвращает множество цепочек терминалов длиной не более k, которые выводятся из цепочки $\alpha \in (\Sigma \cup N)^*$:

$FIRST_k(\alpha) = \{ x \mid \alpha \Rightarrow^* x\beta$ и $|x| = k$ или $\alpha \Rightarrow^* x$ и $|x| < k \}$.

Если цепочка α вырождается, то $\lambda \in FIRST_k(\alpha)$.

$FOLLOW_k(A)$ возвращает множество цепочек терминалов длиной не более k, которые в выводе могут следовать непосредственно за A:

$FOLLOW_k(A) = \{ w \mid S \Rightarrow^* \alpha A y$ и $w \in FIRST_k(y) \}$

Заметим, что $\lambda \in FOLLOW_k(A)$, если $S \Rightarrow^* \alpha A$.

Множества FIRST и FOLLOW нужно научиться вычислять. Нам эти множества не потребуются для $k > 1$, поэтому алгоритмы их вычисления приведем для $k = 1$, и далее будем подразумевать $k = 1$, если k опущено.

Алгоритм 3.5. Функция FIRST.

Пусть $A_i \in N$, $A_i \rightarrow \beta_1, \beta_2, \ldots, \beta_n$, $n \geq 1$, и $\beta_j = X_1 X_2 \ldots X_m$, $m \geq 0$, $X_k \in \Sigma \cup N$.

Вычисляем $FIRST(\beta_j)$ и $FIRST(A_i)$ одновременно для всех β_j и A_i, повторяя шаги алгоритма, пока в какое-нибудь множество можно добавить какой-нибудь символ или λ.

Для каждого i от 1 до $p = |P|$, для каждого j от 1 до $n = |A_i|$ выполнять:

1. Если $\beta_j = \lambda$, то $FIRST(\beta_j) = \{\lambda\}$, $\lambda \in FIRST(A_i)$, следующее j (или i).

2. $k = 1$.

3. Если $X_k \in \Sigma$, то $X_k \in FIRST(\beta_j)$, $X_k \in FIRST(A_i)$, следующее j (или i).

4. $L = FIRST(X_k) — \{\lambda\}$, $L \in FIRST(\beta_j)$, $L \in FIRST(A_i)$.

5. Если $\lambda \notin FIRST(X_k)$, то следующее j (или i).

6. Если $k = |\beta_j|$, то $\lambda \in FIRST(\beta_j)$, $\lambda \in FIRST(A_i)$, следующее j (или i).

7. $k = k+1$, перейти к 3. ●

Алгоритм 3.6. Функция FOLLOW.

Вычисляем $FOLLOW(A)$ одновременно для всех $A \in N$, повторяя шаги алгоритма, пока в какое-нибудь множество можно добавить символ.

1. $FOLLOW(S) = \{\perp\}$, $\perp$ — концевой маркер.

2. Если $(A \rightarrow \alpha B \beta) \in P$, $\alpha \in (\Sigma \cup N)^*$, то $(FIRST(\beta) — \{\lambda\}) \in FOLLOW(B)$.

3. Если $[(A \rightarrow \alpha B) \in P]$ или $[(A \rightarrow \alpha B \beta) \in P$ и $\lambda \in FIRST(\beta)$ (когда $\beta \Rightarrow^* \lambda)]$, $\alpha \in (\Sigma \cup N)^*$, то $FOLLOW(A) \in FOLLOW(B)$. ●

3.7.3. LL-грамматики и языки

Грамматики, обладающие свойством k-предиктивности, называются грамматиками класса LL(k). Языки, порождаемые этими грамматиками, называют LL-языками, алгоритм разбора цепочек LL(k)-языка называют k-предсказывающим алгоритмом, а разбор — предиктивным анализом.

Название LL происходит от двух слов *left*. Первое слово обозначает чтение входной цепочки слева направо, второе — левосторонний вывод. LL(k)-грамматики разрабатывали П. Льюис и Р. Стирнз (1968), а также Д. Розенкранц, построившие компиляторы для языков Алгол и Фортран на основе LL-анализатора. Первоначально LL-грамматики назывались TD-грамматиками, название LL предложил Д. Кнут.

Грамматика $G = (\Sigma, N, P, S)$ называется LL(k)-грамматикой для некоторого фиксированного k, если из существования двух левых выводов

$S \Rightarrow_{lm}^* wA\alpha \Rightarrow_{lm} w\beta\alpha \Rightarrow_{lm}^* wx$

$S \Rightarrow_{lm}^* wA\alpha \Rightarrow_{lm} w\gamma\alpha \Rightarrow_{lm}^* wy$

для которых $\mathrm{FIRST}_k(x) = \mathrm{FIRST}_k(y)$, следует, что $\beta = \gamma$.

Это означает, что для цепочки $wA\alpha$ и первых k символов, выводимых из $A\alpha$, существует ровно одно правило для A, которое выводит цепочку терминалов w, за которой следуют эти k символов. Предсказывающий алгоритм разбора моделирует нисходящий распознаватель с подбором альтернатив, который выбирает правило, рассматривая не более k первых символов нераспознанной части входа, называемых *аванцепочкой*, при этом уже распознанная часть входа w не имеет значения.

Не каждая грамматика порождает LL-язык. Следующая теорема показывает, какими свойствами должна обладать LL(k)-грамматика.

Теорема 3.3. КС-грамматика $G = (\Sigma, N, P, S)$ является LL(k)-грамматикой тогда и только тогда, когда для двух правил $A \to \beta$ и $A \to \gamma$, $\beta \neq \gamma$, и цепочки $wA\alpha$ такой, что $S \Rightarrow_{lm}^* wA\alpha$, $\mathrm{FIRST}_k(\beta\alpha) \cap \mathrm{FIRST}_k(\gamma\alpha) = \varnothing$. ●

Пример 3.18. Рассмотрим грамматику из двух правил $S \to aS \mid a$.

Пусть 1) $wA\alpha = aS$, где $\alpha = \lambda$, и 2) $wA\alpha = a$, где $w = \alpha = \lambda$. Тогда:

$\mathrm{FIRST}_1(aS) = \mathrm{FIRST}_1(a) = a$, поэтому эта грамматика не LL(1).

$\mathrm{FIRST}_2(aS) = aa$, $\mathrm{FIRST}_2(a) = a$, поэтому это LL(2)-грамматика.

После левой факторизации получим эквивалентную грамматику с правилами $S \to aA$, $A \to S \mid \lambda$, которая является LL(1). ●

Пример 3.19. Рассмотрим грамматику из двух правил $S \to Sa \mid b$.

Рассмотрим вывод $S \Rightarrow_{lm}^{i} Sa^i$, где $i \geq 0$, $A = S$, $\alpha = \lambda$, $\beta = Sa$, $\gamma = b$.

Тогда для $i \geq k$, $\mathrm{FIRST}_k(Saa^i) \cap \mathrm{FIRST}_k(ba^i) = ba^{k-1}$, следовательно, эта грамматика не является LL(k) ни для какого k.

Однако после устранения левой рекурсии получим эквивалентную грамматику с правилами $S \to bA$, $A \to aA \mid \lambda$, которая является LL(1). ●

Для LL(k)-грамматик справедливы следующие утверждения.

1. Каждая LL(k)-грамматика однозначна и нелеворекурсивна.

2. Можно удостовериться в том, что произвольная грамматика является LL(k)-грамматикой для данного k.

3. Можно удостовериться в том, что две произвольные LL-грамматики эквивалентны.

4. Невозможно удостовериться в том, что произвольная грамматика является LL-грамматикой.

5. Невозможно удостовериться в том, что существует LL(1)-грамматика, эквивалентная грамматике, не являющейся LL(1)-грамматикой.

6. Для $k \geq 0$ множество LL(k)-языков является собственным подмножеством LL(k + 1)-языков.

3.7.4. LL(1)-грамматики

Нас в большей степени интересуют LL(1)-грамматики, так как для них гораздо легче смоделировать алгоритм разбора. Для этих грамматик справедливо следующее утверждение.

Теорема 3.4. КС-грамматика $G = (\Sigma, N, P, S)$ является LL(1)-грамматикой тогда и только тогда, когда для двух правил $A \to \beta$ и $A \to \gamma$, $\beta \neq \gamma$, $\text{FIRST}(\beta\text{FOLLOW}(A)) \cap \text{FIRST}(\gamma\text{FOLLOW}(A)) = \varnothing$. ●

Из теоремы 3.4 следует, что G является LL(1)-грамматикой тогда и только тогда, когда для каждого множества правил $A \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$:

1) $\text{FIRST}(\beta_i) \cap \text{FIRST}(\beta_j) = \varnothing$ для всех $i \neq j$.

2) если $\beta_i \Rightarrow^* \lambda$, то $\text{FIRST}(\beta_j) \cap \text{FOLLOW}(A) = \varnothing$, для $1 \leq j \leq n$, $i \neq j$.

Теорему 3.4 нельзя обобщить на LL(k)-грамматики для $k > 1$. Чтобы показать это, дадим определение еще одного свойства LL(k)-грамматик.

КС-грамматика, в которой для двух разных правил $A \to \beta$ и $A \to \gamma$ выполняется $\text{FIRST}_k(\beta\text{FOLLOW}_k(A)) \cap \text{FIRST}_k(\gamma\text{FOLLOW}_k(A)) = \varnothing$, называется *сильно* LL(k)-грамматикой (*strong* LL(k)-*grammar*).

Каждая LL(1)-грамматика является сильно LL(1)-грамматикой. Однако существуют LL(k)-грамматики, которые не являются сильно LL(k)-грамматиками, когда $k > 1$. Следующий пример доказывает это.

Пример 3.20. Рассмотрим грамматику, правила которой

$S \to aAaa \mid bAba$

$A \to b \mid \lambda$

Эта LL(2)-грамматика не является сильно LL(2)-грамматикой.

Пусть $S \Rightarrow aAaa$. Тогда $\text{FIRST}_2(baa) \cap \text{FIRST}_2(aa) = \varnothing$.

Пусть $S \Rightarrow bAba$. Тогда $\text{FIRST}_2(bba) \cap \text{FIRST}_2(ba) = \varnothing$.

Однако $\text{FOLLOW}_2(A) = \{aa, ba\}$, и, рассматривая правила для A, получим: $\text{FIRST}(b\text{FOLLOW}(A)) \cap \text{FIRST}(\text{FOLLOW}(A)) = \{ba\}$. ●

Пример 3.21. Грамматика палиндромов четной длины с правилами $S \to aSa \mid bSb \mid \lambda$ не является LL(1)-грамматикой. Для нее очевидно:

FIRST(aSa) = {a}, FIRST(bSb) = {b}, FOLLOW(S) = {$\perp$, a, b}. ●

Пример 3.22. Грамматика парности скобок с правилами $S \to aSb \mid \lambda$ является LL(1)-грамматикой. Для нее:

FIRST(aSb) = {a}, FOLLOW(S) = {$\perp$, b}. ●

Если в грамматике нет пустых правил, множества FOLLOW для нее не имеют значения, их можно не вычислять.

Если в грамматике нет пустых правил, и все альтернативы каждого нетерминала начинаются разными терминалами, грамматика называется *простой*, *разделенной* или *s*-грамматикой. Для такой грамматики можно не вычислять множества FIRST и FOLLOW.

3.7.5. Разбор LL(1)-языков

Рассмотрим, как осуществляется разбор цепочек языков, порождаемых LL(1)-грамматиками. Можно показать, что 1-предсказывающий алгоритм ***A***, который мы опишем далее, выполняет разбор LL(1)-языков.

Пусть $G = (\Sigma, N, P, S)$ — LL(1)-грамматика.

Алгоритм строится на основе детерминированного МП-автомата с опустошением магазина $P = (\{q\}, \Sigma, \Sigma \cup N \cup \{\perp\}, \Delta, \delta, q, S\perp, \varnothing)$, в который добавлено множество выходных символов Δ и выходная лента. Пусть правила G занумерованы числами от 1 до p, тогда $\Delta = \{1, 2, \ldots, p\}$.

Работой алгоритма руководит управляющая таблица M, отображающая множество $\{\perp\} \cup (\Sigma \cup N) \times \{\perp\} \cup \Sigma$ на множество, состоящее из:

1) пар (β, i), где β — тело правила $A \to \beta$ с номером i,

2) признаков *выброс* (*pop*),

3) признака *допуск* (*accept*),

4) признаков *ошибка* (*error*).

Конфигурация алгоритма — это тройка (x, α, π), x — нераспознанная часть цепочки w на входе, α — цепочка в стеке, π — цепочка на выходе. Начальная конфигурация $(w, S\perp, \lambda)$. Функция δ формируется так же, как описано в 3.3.1, но с учетом записи выходной цепочки. Тогда алгоритм ***A*** выполняет следующие шаги, просматривая один символ входа:

1) $(ax, A\gamma, \pi) \vdash (ax, \beta\gamma, \pi i)$, если $M(A, a) = (\beta, i)$.

2) $(ax, a\gamma, \pi) \vdash (x, \gamma, \pi)$, если $M(a, a) =$ *выброс*.

3) в конфигурации $(\perp, \perp, \pi)$ работа завершается, и π содержит левый разбор. Эта допускающая конфигурация соответствует пустому стеку.

4) если $M(X, a) =$ *ошибка* в конфигурации $(ax, X\gamma, \pi)$, работа завершается, алгоритм сигнализирует ошибку. Это ошибочная конфигурация.

Будем называть алгоритм ***A*** *левым анализатором*, а допускаемые им языки — *лево-анализируемыми*.

Таблицу M можно построить при помощи следующего алгоритма.

Алгоритм 3.7. Построение управляющей таблицы LL(1)-анализа.

Вход. LL(1)-грамматика G.

Выход. Управляющая таблица для грамматики G.

Таблица M определяется следующим образом:

1) $(\beta, i) \in M(A, a)$ $\forall a \in \text{FIRST}(\beta) - \{\lambda\}$, если $A \to \beta$ — правило i.

2) $(\beta, i) \in M(A, b)$ $\forall b \in \text{FOLLOW}(A)$, если $A \to \beta$ — правило i, и $\beta \Rightarrow^* \lambda$.

3) $M(a, a)$ = признак *выброс* $\forall a \in \Sigma$.

4) $M(\perp, \perp)$ = признак *допуск*.

5) $M(X, a)$ = *ошибка* во всех остальных случаях; $X \in (\Sigma \cup N)$, $a \in \Sigma$.

Если пункты 1) и 2) генерируют множественные записи пар (β, i) в ячейках таблицы, грамматика G не является LL(1)-грамматикой. ●

Пример 3.23. Построим управляющую таблицу для грамматики G_3, описывающую язык выражений со скобками:

$E \to^{(1)} TR$

$T \to^{(2)} PF$

$P \to^{(3)} a \mid^{(4)} (E)$

$R \to^{(5)} +TR \mid^{(6)} \lambda$

$F \to^{(7)} *PF \mid^{(8)} \lambda$

Множества FIRST и FOLLOW приведены в следующей таблице:

A	FIRST(A)	FOLLOW(A)
E	a, (	$\perp$,)
T	a, (	$\perp$,), +
P	a, (	$\perp$,), *, +
R	+, λ	$\perp$,)
F	*, λ	$\perp$,), +

В соответствии с алгоритмом 3.7 определим пары (β, i).

$M(E, a) = M(E, () = (TR, 1)$, т.к. $a \in \text{FIRST}(TR)$ и $(\in \text{FIRST}(TR)$,

$M(T, a) = M(T, () = (PF, 2)$, т.к. $a \in \text{FIRST}(PF)$ и $(\in \text{FIRST}(PF)$,

$M(P, a) = (a, 3)$, т.к. $a \in \text{FIRST}(a)$,

$M(P, () = ((E), 4)$, т.к. $(\in \text{FIRST}((E))$,

$M(R, +) = (+TR, 5)$, т.к. $+ \in \text{FIRST}(+TR)$,

$M(R, \perp) = M(R,)) = (\lambda, 6)$, т.к. $\perp \in \text{FOLLOW}(R)$ и $) \in \text{FOLLOW}(R)$

$M(F, *) = (*PF, 7)$, т.к. $* \in \text{FIRST}(*PF)$

$M(F, \perp) = M(F,)) = M(F, +) = (\lambda, 8)$, т.к. $\perp \in \text{FOLLOW}(F)$,

$) \in \text{FOLLOW}(F)$ и $+ \in \text{FOLLOW}(F)$.

Запишем *выброс* в ячейки $M(a, a)$, $M((, (), M(),))$, $M(+, +)$ и $M(*, *)$.

Запишем *допуск* в ячейку в $M(\perp, \perp)$.

Пустые ячейки пусть обозначают признак *ошибка*.

Получим следующую управляющую таблицу M для G_3:

M	$\perp$	a	(	)	+	*
$\perp$	*допуск*					
a		*выброс*				
(			*выброс*			
)				*выброс*		
+					*выброс*	
*						*выброс*
E		TR, 1	TR, 1			
T		PF, 2	PF, 2			
P		a, 3	(E), 4			
R	λ, 6			λ, 6	$+TR$, 5	
F	λ, 8			λ, 8	λ, 8	$*PF$, 7

Для определения шага алгоритм выбирает ячейку в строке, помеченной символом на стеке, и в столбце, помеченном символом на входе. Например, для входа $a+a*a$ алгоритм выполнит последовательность шагов:

$(a+a*a\perp, S\perp, \lambda) \vdash (a+a*a\perp, TR\perp, 1)$
$\vdash (a+a*a\perp, PFR\perp, 12)$
$\vdash (a+a*a\perp, aFR\perp, 123)$
$\vdash (+a*a\perp, FR\perp, 123)$
$\vdash (+a*a\perp, R\perp, 1238)$
$\vdash (+a*a\perp, +TR\perp, 12385)$
$\vdash (a*a\perp, TR\perp, 12385)$
$\vdash (a*a\perp, PFR\perp, 123852)$
$\vdash (a*a\perp, aFR\perp, 1238523)$
$\vdash (*a\perp, FR\perp, 1238523)$
$\vdash (*a\perp, *PFR\perp, 12385237)$
$\vdash (a\perp, PFR\perp, 12385237)$
$\vdash (a\perp, aFR\perp, 123852373)$
$\vdash (\perp, FR\perp, 123852373)$
$\vdash (\perp, R\perp, 1238523738)$
$\vdash (\perp, \perp, 12385237386)$

Полученная последовательность правил 12385237386 соответствует следующему левостороннему выводу:

$S \Rightarrow_1 TR \Rightarrow_2 PFR \Rightarrow_3 aFR \Rightarrow_8 aR \Rightarrow_5 a+TR \Rightarrow_2 a+PFR \Rightarrow_3 a+aFR \Rightarrow$
$\Rightarrow_7 a+a*PFR \Rightarrow_3 a+a*aFR \Rightarrow_8 a+a*aR \Rightarrow_6 a+a*a$ ●

Утверждение. Число шагов, выполняемых 1-предсказывающим алгоритмом с управляющей таблицей, построенной по LL(1)-грамматике алгоритмом 3.7, линейно зависит от длины входной цепочки. ●

Вследствие этого метод разбора LL(1)-грамматик находит практическое применение при построении трансляторов.

3.8. Восходящий анализ

Наиболее широкий класс детерминированных КС-языков разбирается при помощи восходящего анализа. Основным методом восходящего разбора является LR-анализ, который представлен LR(k)-грамматиками и языками. Подмножеством LR(k)-языков и грамматик являются языки и грамматики предшествования и некоторые другие. Напомним, что в восходящем анализе алгоритм типа «перенос-свертка» строит обращенное правое порождение. При этом на стеке МП-автомата ищется основа, свертка которой ведет к предыдущему шагу этого порождения.

Неформально, восходящий анализ разбирает более широкий класс языков потому, что в нем существует больше информации для принятия решения. Пусть входная цепочка wxy, и цепочка x может быть выведена из нетерминала A (рисунок 3.3).

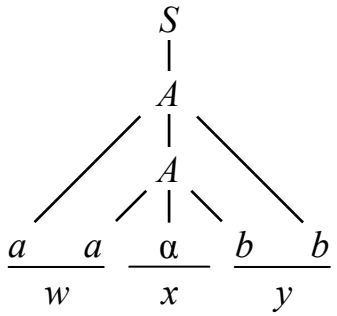

Рисунок 3.3 — Примерное дерево вывода

При нисходящем анализе правило для развертывания нетерминала A выбирается на основании k первых символов xy, цепочка же w исчезает после ее распознавания. При восходящем анализе решение о свертке α принимается на основании распознанной цепочки wx и k первых символов y, при этом история переносов и сверток wx хранится в стеке.

Пример 3.24. Рассмотрим язык $L_{abac} = \{a^n b^n \mid n \geq 1\} \cup \{a^n c^n \mid n \geq 1\}$, являющийся детерминированным, но не LL-языком.

Следующая грамматика порождает этот язык:

$S \rightarrow A \mid B$

$A \rightarrow aAb \mid \lambda$

$B \rightarrow aBc \mid \lambda$

Дерево разбора цепочки $aabb$ показано на рисунке 3.3, $\alpha = \lambda$.

Чтобы понять, какое правило следует применить первым, нужно разобрать цепочку до ее середины. Нисходящий анализатор не может этого сделать, потому что к середине цепочки ее начало должно быть разобрано. Восходящий же разбор переносит начало цепочки в стек, после чего «видит», какое правило должно быть применено. ●

Восходящий анализатор на каждом шаге принимает три решения:

1) какой шаг выполнить — перенос или свертку;

2) где находится левый конец основы, если шаг — свертка;

3) к какому символу свернуть основу, если вариантов несколько.

Есть несколько классов грамматик, для которых эти решения принимаются однозначно. Мы рассмотрим только класс LR(k)-грамматик, и подробно класс LR(1)-грамматик, так как анализаторы LR(1)-языков являются самыми эффективными среди всех анализаторов, для построения трансляторов они применяются чаще всего.

3.8.1. LR-грамматики и языки

В грамматиках класса LR(k) описанные выше решения принимаются на основании первых k символов нераспознанной части входа и текущего состояния на стеке. Название LR происходит от слов *left* и *right*, означающих чтение цепочки слева направо и построение правостороннего вывода. LR-грамматики впервые предложил Д. Кнут (1965) [1].

Для анализа методом LR исходная грамматика заменяется эквивалентной грамматикой, которую называют *пополненной* или *расширенной* (*augmented*). Пусть есть КС-грамматика $G = (\Sigma, N, P, S)$. Так как целевой символ грамматики может встречаться в правых частях правил, свертка к нему не может служить признаком допуска. Добавляя в грамматику правило $S' \to S$ с номером 0, мы гарантируем, что свертка к целевому символу S' сигнализирует успешное завершение разбора. Тогда можно дать следующее определение LR(k)-грамматики.

Грамматика G является LR(k)-грамматикой, если из существования двух выводов в расширенной грамматике

$S' \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha\beta w$

$S' \Rightarrow_{rm}^* \gamma B x \Rightarrow_{rm} \alpha\beta y$

для которых $\mathrm{FIRST}_k(w) = \mathrm{FIRST}_k(y)$, следует, что $\alpha A y = \gamma B x$.

Это означает, что если $A \to \beta$ — последнее правило, использованное в правостороннем выводе цепочки $\alpha A w$, то $\alpha\beta y$ может быть свернуто только в $\alpha A y$, то есть правило $A \to \beta$ должно быть использовано и при свертке $\alpha\beta y$ в $\alpha A y$, при этом $\alpha = \gamma$, $A = B$ и $x = y$. Так как β выводится из A независимо от w, то из определения LR(k)-грамматики следует, что множество $\mathrm{FIRST}_k(w)$ содержит информацию, достаточную для определения того, что $\alpha\beta$ за один шаг выводится из αA.

Будем называть язык, порождаемый LR-грамматикой, LR-языком.

Доказано, что класс LR(k)-грамматик, и даже класс LR(1)-грамматик порождают языки, в точности совпадающие с детерминированными КС-языками, а для каждого LR-языка существует LR(1)-грамматика.

Пример 3.25. Рассмотрим грамматику

$S' \to S$

$S \to Ab \mid Bc$

$A \to Aa \mid a$

$B \to Ba \mid a$

Рассмотрим два вывода:

$S' \Rightarrow_{rm} S \Rightarrow_{rm}^* Aa^k b \Rightarrow_{rm} a^k ab$

$S' \Rightarrow_{rm} S \Rightarrow_{rm}^* Ba^k c \Rightarrow_{rm} a^k ac$

Принимая $\alpha = \lambda$, $\beta = \lambda$, $w = a^k ab$, $y = a^k ac$, $\mathrm{FIRST}_k(w) = \mathrm{FIRST}_k(y)$, и эта грамматика не является LR(k) ни для какого k. ●

Пример 3.26. Рассмотрим грамматику

$S' \to S$

$S \to Ab \mid Bc$

$A \to aA \mid a$

$B \to aB \mid a$

Рассмотрим два вывода:

$S' \Rightarrow_{rm} S \Rightarrow_{rm}^* a^k Ab \Rightarrow_{rm} a^k ab$

$S' \Rightarrow_{rm} S \Rightarrow_{rm}^* a^k Bc \Rightarrow_{rm} a^k ac$

Для нее $\alpha = a^k$, $\beta = \lambda$, $w = b$, $y = c$, это LR(1)-грамматика. ●

Пример 3.27. Рассмотрим грамматику

$S' \to S$

$S \to AB \mid aC$

$A \to ab$

$B \to bb$

$C \to bba$

Рассмотрим два вывода:

$S' \Rightarrow_{rm} S \Rightarrow_{rm} AB \Rightarrow_{rm} Abb \Rightarrow_{rm} \underline{ab}bb$

$S' \Rightarrow_{rm} S \Rightarrow_{rm} aC \Rightarrow_{rm} a\underline{bba}$

После считывания в стек ab невозможно определить правый конец основы, рассматривая один символ цепочки w, равной в первом выводе bb, а во втором выводе ba. Эта грамматика не LR(1), но LR(2). ●

Для LR(k)-грамматик справедливы следующие утверждения.

1. Каждая LR(k)-грамматика однозначна.

2. Можно удостовериться в том, что произвольная грамматика является LR(k)-грамматикой для данного k.

3. Невозможно удостовериться в том, что произвольная грамматика является LR-грамматикой.

4. Класс LL грамматик является собственным подмножеством класса LR грамматики, а классы LL языков являются собственными подмножествами класса LR языков.

3.8.2. Анализатор LR-грамматик

Анализатор LR-грамматик строится на основе детерминированного МП-автомата с алгоритмом «перенос-свертка» (раздел 3.3.2), в который добавлено множество выходных символов и выходная лента. Активные префиксы на стеке МП-автомата распознает детерминированный конечный автомат, который переносит на стек символы входа до тех пор, пока на стеке не обнаружится основа.

Рассмотрим вывод $S' \Rightarrow_{rm} S \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha\beta w$.

Если на стек перенесен активный префикс αβ, тогда $\alpha\beta\mathrm{FIRST}_k(w)$ содержит префикс предыдущего шага обращенного правого порождения, основа обнаружена, и МП-автомат выполняет свертку $\beta \to A$. При этом на стеке меняется состояние ДКА. Выбор действия МП-автомата осуществляется однозначно на основании текущего состояния ДКА и первых k символов нераспознанной части входа.

Далее мы рассматриваем разбор в LR(1)-грамматиках.

Для работы анализатора требуется управляющая таблица, представляющая собой таблицу переходов ДКА, в которую внесена информация о свертках. Логически эта таблица делится на две части — «действие» (*action*) и «переход» (*goto*). В столбцах управляющей таблицы в части «действие» распределены концевой маркер и терминалы. В части «переход» в столбцах распределены нетерминалы, которые появляются на стеке в процессе разбора.

Ячейки управляющей таблицы в части «действие» содержат данные, предписывающие выполняемые МП-автоматом действия:

- «сдвиг, новое состояние ДКА» — если выполняется сдвиг;
- «свертка, номер правила» — если выполняется свертка;
- «допуск» — если возможна свертка к целевому символу S';
- «ошибка» во всех других случаях.

Ячейки управляющей таблицы в части «переход» содержат состояния, в которые ДКА переходит после свертки к некоторому нетерминалу A. Пустые ячейки здесь ничего не обозначают, так как они никогда не используются.

Состояния ДКА обозначены числами $0 \ldots n-1$ (n — число состояний ДКА) и используются для нумерации строк управляющей таблицы.

На стек помещаются пары вида (x, s), где x — символ входа, а s — номер строки управляющей таблицы (состояние ДКА). Первоначально на стек помещается пара $(\perp, 0)$, где $s = 0$ соответствует первой строке управляющей таблицы и начальному состоянию ДКА. Символы x не требуются для функционирования анализатора, и помещаются на стек исключительно для наглядности.

Алгоритм анализатора грамматики LR(1) следующий.

Алгоритм 3.8. Разбор в грамматике LR(1).

x — текущий символ входной цепочки, s — состояние ДКА, первоначально $s = 0$ записано в стек, M — управляющая таблица LR анализа.

Шаг 1. Если $M(s, x) = $ (сдвиг, s'), то:

- протолкнуть в стек x, сдвинуть считывающую головку вправо на 1,
- протолкнуть в стек s',
- положить $s = s'$,
- перейти к шагу 1.

Шаг 2. Если $M(s, x) = $ (свертка, i), где $A \to \beta$ — правило номер i, то:

- вытолкнуть из стека $2|\beta|$ символов,
- запомнить состояние s' на вершине стека,
- протолкнуть в стек A,
- протолкнуть в стек $s'' = M(s', A)$,
- положить $s = s''$,
- записать i в цепочку правил π,
- перейти к шагу 1.

Шаг 3. Если $M(s, x) = $ «допуск», то фиксировать «допуск», конец.

Шаг 4. Если $M(s, x) = 0$, то фиксировать «ошибка», конец. ●

Для работы анализатора требуется управляющая таблица. Есть три метода построения управляющих таблиц, отличающиеся мощностью LR языков, которые могут быть распознаны при помощи алгоритма 3.8. Построение таблиц рассматривается далее.

Пример 3.28. Рассмотрим расширенную грамматику:

$$S' \to^{(0)} S$$
$$S \to^{(1)} aA$$
$$A \to^{(2)} b \mid^{(3)} Sb \qquad (3.6)$$

Грамматика порождает язык $a^n b^n$, $n > 0$. Управляющая таблица этой грамматики имеет следующий вид:

Таблица 1 — Управляющая таблица для грамматики (3.6)

	$\perp$	a	b	S	A
0		s, 2		1	
1	acc				
2		s, 2	s, 4	5	3
3	r, 1		r, 1		
4	r, 2		r, 2		
5			s, 6		
6	r, 3		r, 3		

Символами s обозначены сдвиги, символами r — свертки, acc обозначает допуск. В части «действие» 3 столбца, в части «переход» — 2.

На рисунке приведен граф переходов ДКА для грамматики (3.6).

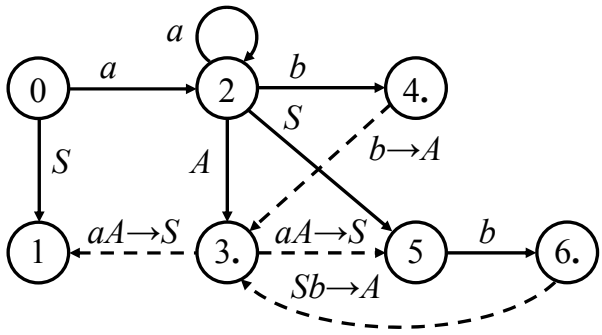

Рисунок 4 — ДКА для грамматики (3.6)

Точками обозначены состояния, в которых выполняются свертки, а пунктирными стрелками — переходы после сверток.

Рассмотрим вывод цепочки "*aabb*" в следующей таблице.

#	Стек	Вход	Действие	Правила
1	⊥ 0	*aabb*⊥	*s*, 2	
2	⊥ 0 *a* 2	*abb*⊥	*s*, 2	
3	⊥ 0 *a* 2 *a* 2	*bb*⊥	*s*, 4	
4	⊥ 0 *a* 2 *a* 2 *b* 4	*b*⊥	*r*, 2	2
5	⊥ 0 *a* 2 *a* 2 *A* 3	*b*⊥	*r*, 1	21
6	⊥ 0 *a* 2 *S* 5	*b*⊥	*s*, 6	21
7	⊥ 0 *a* 2 *S* 5 *b* 6	⊥	*r*, 3	213
8	⊥ 0 *a* 2 *A* 3	⊥	*r*, 1	2131
9	⊥ 0 *S* 1	⊥	*acc*	2131

Разбор происходит очень просто. Состояние на стеке выбирает строку управляющей таблицы, а символ на входе — столбец. Считываем значение в ячейке и выполняем шаг алгоритма.

Сначала на стеке 0, на входе *a*. В таблице видим (*s*, 2), то есть перенос и переход в состояние 2. Переносим на стек символ *a* и записываем новое состояние 2 (строка 2 таблицы разбора). Далее аналогичным образом выполняем еще два переноса и два перехода, сначала в состояние 2, затем в состояние 4.

В состоянии 4 всегда выполняется свертка *b*→*A* по правилу 2. Копируем цепочку на стеке из строки 4 в строку 5 (в таблице разбора). Умножая длину тела правила на два, получаем два. Извлекаем из стека два символа, при этом в строке 5 на стеке останется цепочка "⊥ 0 *a* 2 *a* 2". Приписываем символ свертки *A*, читаем на вершине стека "2 *A*" (подчеркнуто), находим в ячейке M(2, *A*) новое состояние 3, которое записываем на стек, получая "⊥ 0 *a* 2 *a* 2 *A* 3". ●

3.8.3. SLR-анализ

Существует три метода построения управляющих таблиц LR:
- SLR (simple LR),
- канонический LR-анализ,
- LALR (look ahead LR, или LR-анализ с предпросмотром).

Эти методы различаются своей мощностью и сложностью анализа. Самым слабым и простым является SLR-анализ, самым мощным (охватывающим большее множество КС-языков) — канонический LR-анализ. LALR-анализ самый сложный и промежуточный по мощности.

В основе SLR-анализа лежит каноническая система LR(0)-пунктов.

LR(0)-пункт (или просто пункт, ситуация, item) грамматики G' — это продукция грамматики со специальным символом • (точка) в некоторой позиции ее правой части. Например, для продукции $A \to xyz$ возможны следующие пункты: $[A \to \bullet xyz]$, $[A \to x \bullet yz]$, $[A \to xy \bullet z]$, $[A \to xyz \bullet]$.

Пункт — это состояние ДКА в некоторый момент распознавания активного префикса. Он показывает, какая часть префикса уже распознана (до точки), а какая еще нет (ожидается после точки). Например, пункт $[A \to x \bullet yz]$ показывает, что ДКА распознал символ x и ожидает получить символ y, а пункт $[A \to xyz \bullet]$ показывает, что основа распознана.

Каноническая система LR(0)-пунктов — это множество всех возможных ситуаций, которые могут встретиться во время разбора. Пункты группируются в множества I_j, которые определяют состояния ДКА j. В качестве примера построим множества пунктов для грамматики (3.6):

$S' \to {}^{(0)} S$
$S \to {}^{(1)} aA$
$A \to {}^{(2)} b \mid {}^{(3)} Sb$

Для построения канонической системы LR(0)-пунктов используются операции *closure* (замыкание) и *goto* (переход). Операция *closure* строит множество пунктов из базовых пунктов, составляющих основу множества I_j. Пусть есть грамматика $G = (\Sigma, N, P, S)$, $V = \Sigma \cup N$.

Следующий алгоритм вычисляет замыкание.

Алгоритм 3.9. Операция *closure*(J)

Исходные данные: грамматика G и множество базовых пунктов J.

Шаг 1. Положить $I_0 = J$, $i = 1$.

Шаг 2. Если $[A \to \alpha \bullet B\beta] \in (I_i \cup I_{i-1})$, то $[B \to \bullet\gamma] \in I_i \ \forall (B \to \gamma) \in P$, $\alpha, \beta \in V^*$.

Шаг 3. Если $I_j = I_{j-1}$, то 4, иначе $i = i + 1$, перейти к 2.

Шаг 4. I_j — искомое множество. ●

Исходным (начальным) множеством пунктов является множество из пункта, получаемого из правила 0: $J = \{[S' \to \bullet S]\}$, а начальное состояние ДКА определяется множеством $I_0 = closure(J)$.

Построим множество I_0. Так как точка находится в начальном пункте перед нетерминалом S, и в грамматике есть правило $S \to aA$, включаем во множество I_0 новый пункт: $[S \to \bullet aA]$. Получим:

$I_0 = \{[S' \to \bullet S], [S \to \bullet aA]\}$.

Множество I_0 содержит все возможные активные префиксы, которые могут следовать из S'. Если существует вывод

$S' \Rightarrow_{rm} \gamma = X\beta,\ X \in V, \beta \in V^*$,

то множество I_0 содержит все возможные варианты X, которые ведут к переходам в другие состояния ДКА. Анализ префиксов $X\beta$ множества I_0 показывает, что есть два разных символа X: S и a (рисунок 3.5).

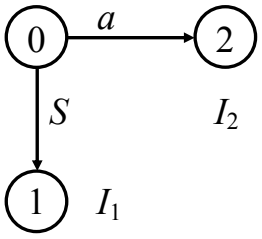

Рисунок 5 — Переходы из начального состояния ДКА

Переходы ДКА в другие состояния соответствуют формированию множеств пунктов I_1 и I_2, соответствующих состояниям 1 и 2 ДКА. Для построения этих множеств используется операция *goto*(J, X).

Алгоритм 3.10. Операция *goto*(J, X)

Исходные данные: множество пунктов J, символ перехода X.

Шаг 1. Положить $I = \varnothing$. Повторять шаг 2, пока возможно.

Шаг 2. Если $[A \to \alpha \bullet X\beta] \in J$, то $[A \to \alpha X \bullet \beta] \in I$, $\alpha, \beta \in V^*$.

Шаг 3. *closure*(I) — искомое множество. ●

Для рассматриваемого примера функция *goto*(I_0, S) на шаге 2 вычисляет множество базовых пунктов $\{[S' \to S\bullet]\}$, замыкание которого ничего не добавляет. Функция *goto*(I_0, a) на шаге 2 вычисляет множество базовых пунктов $\{[S \to a\bullet A]\}$, замыкание которого добавляет пункты $[A \to \bullet b]$ и $[A \to \bullet Sb]$ и $[S \to \bullet aA]$.

Далее процесс повторяется до тех пор, пока нельзя будет сформировать ни одного нового множества базовых пунктов. За исключением множества I_0, множество базовых пунктов всегда содержит только такие пункты, точка в которых находится не в начале тела правила.

Заметим также, что функция *goto* не всегда ведет к формированию нового множества I_j. Если эта функция возвращает множество, совпадающее с уже существующим множеством I_k, то ДКА переходит в уже существующее состояние k.

В множестве I_1 единственный пункт $[S' \to \bullet S]$ содержит точку в конце, что соответствует свертке, поэтому из этого множества нет переходов.

Рассмотрим множество $I_2 = \{[S \to a \bullet A], [A \to \bullet b], [A \to \bullet Sb], [S \to \bullet aA]\}$.

Функция $goto(I_2, A)$ формирует множество $I_3 = \{[S \to aA \bullet]\}$, а функция $goto(I_2, b)$ множество $I_4 = \{[S \to b \bullet]\}$, в которых возможна только свертка.

Функция $goto(I_2, S)$ формирует множество $I_5 = \{[A \to S \bullet b]\}$, а функция $goto(I_2, a)$ формирует множество, совпадающее с множеством I_2, поэтому на графе переходов появляется петля в состоянии 2. Наконец, функция $goto(I_5, b)$ формирует множество $I_6 = \{[A \to Sb \bullet]\}$.

Таким образом, сформирована каноническая система LR(0)-пунктов:

$I_0 = \{[S' \to \bullet S], [S \to \bullet aA]\}$,

$I_1 = \{[S' \to S \bullet]\}$,

$I_2 = \{[S \to a \bullet A], [A \to \bullet b], [A \to \bullet Sb], [S \to \bullet aA]\}$,

$I_3 = \{[S \to aA \bullet]\}$,

$I_4 = \{[S \to b \bullet]\}$,

$I_5 = \{[A \to S \bullet b]\}$,

$I_6 = \{[A \to Sb \bullet]\}$.

Формально построение канонической системы LR(0)-пунктов описывает следующий алгоритм.

Алгоритм 3.11. Построение канонической системы LR(0)-пунктов

Шаг 1. $C_0 = I_0 = closure(\{[S' \to \bullet S]\})$, $i = 1$, $k = 0$, $m = 0$.

Шаг 2. $C_i = C_i \cup goto(I_j, X)$, $\forall j = k \ldots m$, $\forall X$: $[A \to \alpha \bullet X\beta] \in I_j$.

Шаг 3. Если $C_i = C_{i-1}$, то перейти к шагу 4, иначе положить $k = m + 1$, $m = |C_i|$, $i = i + 1$, перейти к шагу 2.

Шаг 4. C_i — искомое множество. ●

Непосредственно из канонической системы может быть построена таблица переходов ДКА, так как если $I_k = goto(I_j, X)$, то $k \in (j, X)$. Однако МП-автомат должен также выполнять свертки в состояниях, содержащих единственный пункт с точкой в конце. Рассмотрим пункт $[S \to aA \bullet]$ множества I_3. Свертка по правилу 1 выполняется в случае, если текущий символ входа принадлежит FOLLOW(S). Для рассматриваемой грамматики FOLLOW(A) = FOLLOW(S) = $\{\perp, b\}$, поэтому свертки записываются в строке 3 таблицы переходов в столбцах $\perp$ и b. Следующий алгоритм показывает, как строится управляющая таблица.

Алгоритм 3.12. Построение управляющей таблицы SLR(1)-анализа

1. (*сдвиг*, j)$\in$M(i, x), если $I_j = goto(I_i, x)$, $x \in \Sigma$.

2. $j \in$M(i, A), если $I_j = goto(I_i, A)$, $A \in$N.

3. (*свертка*, i)$\in$M(j, x), если $[A \to \beta \bullet] \in I_j$, $x \in$FOLLOW(A), $A \neq S'$, и i — номер правила $A \to \beta$.

4. *допуск*$\in$M(j, $\perp$), если $[S' \to S \bullet] \in I_j$. ●

Пустые ячейки в части «действие» являются признаком ошибки.

Грамматика не является SLR(1), если в какой-либо ячейке таблицы в части «действие» будет записано более одной инструкции.