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

Вл. Пономарев

Конспективное изложение теории языков программирования и методов трансляции

Книга 1. Формальные языки и грамматики

Учебно-методическое пособие

Озерск, 2019

УДК 681.3.06
П56

Вл. Пономарев. Конспективное изложение теории языков программирования и методов трансляции. Учебно-методическое пособие. В 4-х книгах. Книга 1. Формальные языки и грамматики. Озерск: ОТИ НИЯУ МИФИ, 2019. — 42 с., ил.

В книге 1 кратко излагается теория формальных языков и грамматик, преобразование контекстно-свободных грамматик, нормальные их формы, общая схема трансляции.

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

Рецензенты:
1. Р. Р. Акопян, к.ф.-м.н., зав. кафедрой ПМ ОТИ НИЯУ МИФИ.
2. В. Е. Синяков, начальник УИТ ВГУП «ПО «Маяк».

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

Содержание

Введение

Теория языков программирования разрабатывалась одновременно с появлением вычислительных машин и первых языков программирования. Ей несколько десятков лет, но за это время ее базовые положения не претерпели каких-либо существенных изменений, особенно в части анализа. Фундаментом теории языков программирования являются теория формальных языков и теория автоматов. Этот формальный аппарат формирует прочную математическую основу языков программирования, и должен входить в программу обучения студентов, специализирующихся в области информационных технологий.

В этой книге пособия вводятся основные понятия, используемые в теории повсеместно: формальные языки и грамматики, классификация грамматик, вывод и деревья вывода. Здесь же рассматриваются преобразования контекстно-свободных грамматик и основные понятия трансляции языков: лексика, синтаксис и семантика языков, распознаватели языков, структура трансляции. Приводится пример трансляции простого языка, разработанный в соответствии с теорией.

В конце прилагаются вопросы для самопроверки и упражнения.

Во второй книге пособия излагается лексический анализ: лексемы, регулярные языки и теория конечных автоматов.

В третьей книге пособия рассматривается синтаксический анализ: контекстно-свободные языки и МП-автоматы, основные методы нисходящего и восходящего синтаксического разбора.

Четвертая книга посвящена принципам синтаксически управляемого перевода, в частности, атрибутного перевода.

Изложение материала имеет целью ввести студента в теорию языков программирования настолько, чтобы он понимал принципы и механизмы, лежащие в основе перевода. Эта книга не для теоретиков с одной стороны, и не для профессионалов, разрабатывающих компиляторы, с другой. Изложение в ней максимально упрощено и сжато, и преследует методические цели больше, чем теоретические или практические. Тем не менее, ее чтение требует внимания и тщательного анализа приводимых сведений.

Все приведенные в пособии алгоритмы смоделированы и проверены. Теоремы и их доказательства приводятся в отдельной книге.

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

1. Формальные языки и грамматики

1.1. Основные понятия

1.1.1. Знаки и алфавит

Язык есть множество текстов, которые, в свою очередь, состоят из отдельных знаков. Знак, называемый также буквой или символом, — минимальная единица текста, неразложимая на другие составляющие. В этом смысле между буквой и знаком нет разницы.

Между буквой или знаком и символом разница может существовать в том смысле, что некоторые символы представляются как множество глифов (отдельных изображений). Например, для обозначения операции присваивания в языке Pascal используется символ из двух знаков ":=".

Если это не вызывает неоднозначности, мы будем считать понятия знака, буквы и символа эквивалентными.

Основу любой знаковой системы составляет ее алфавит.

Алфавит — это конечное непустое множество знаков.

Пример алфавита может привести даже школьник, — это родной алфавит, состоящий из 33 букв кириллицы. Однако для записи текста на родном языке этих букв недостаточно, неудобно записывать предложения без знаков препинания и цифр, хотя в речи букв достаточно.

В теории формальных языков все строже. Если какого-то знака нет в алфавите, использовать его в записях недопустимо. При описании формальных языков алфавит задается в первую очередь. Обычно это всего лишь несколько знаков типа a и b, например: $A = \{a, b\}$. При этом знаки a и b часто не обозначают самих себя. Например, может подразумеваться, что a — это буква, а b — цифра.

1.1.2. Цепочки

Цепочка — это основное понятие в формальных языках, потому что языки состоят из цепочек, и слова, предложения и тексты в целом — это цепочки. Цепочка — это, обычно, конечная последовательность знаков некоторого алфавита.

Будем обозначать цепочки буквами греческого алфавита, например, α, β, γ, и буквами латинского алфавита из его конечной части, такими, как w, v, x, y, z. Для обозначения отдельных знаков будем использовать буквы начальной части латинского алфавита, например, a, b, c, d.

Говорят, что некоторая цепочка α является цепочкой *над алфавитом* (или *в алфавите*) Σ: $\alpha(\Sigma)$, если все знаки α принадлежат Σ.

Две цепочки, α и β, являются равными: α = β, если совпадают количество, состав и порядок их знаков. Если α = "abc", β = "rst", то α и β задают одинаковое количество знаков, но разный состав и порядок, они не равны (кавычки "" здесь и далее ограничивают длину цепочки).

Длину цепочки α обозначают |α|. Для приведенных выше цепочек α и β можно записать |α| = |β| (длина цепочки α равна длине цепочки β).

Над цепочками выполняются следующие операции: конкатенация, обращение и итерация.

Конкатенация — это присоединение одной цепочки к другой. Например, конкатенация α и β дает новую цепочку δ = α·β = "abcrst". Знак конкатенации "·" обычно опускают.

Конкатенация ассоциативна: α(βγ) = (αβ)γ, но не коммутативна: в общем случае αβ ≠ βα, хотя в частном случае возможно αβ = βα.

Обращение — это запись знаков цепочки в обратном порядке. Для цепочки α обращение записывается α^R. Если α = "кит", то α^R = "тик". Для обращения справедливо следующее равенство: $(\alpha\beta)^R = \alpha^R\beta^R$.

Итерация — это повторение цепочки произвольное количество раз.

Итерация цепочки α *n* раз, $n \geq 0$, записывается как α^n.

Если α = "ab", то: α^0 = "", α^1 = "ab", α^2 = "abab", α^3 = "ababab" и т.д.

Заметим, что α^0 порождает цепочку нулевой длины, она называется *пустой*. В теории языков пустые цепочки чрезвычайно важны. Будем обозначать пустую цепочку греческой буквой λ. Для ее обозначения используют также греческую букву ε и латинскую e.

Для пустой цепочки очевидно справедливы следующие равенства:

$|\lambda| = 0$;

$\lambda\alpha = \alpha\lambda = \alpha$;

$\lambda^R = \lambda$;

$\lambda^n = \lambda$, $n \geq 0$;

Другие операции над цепочками включают в себя выделение префикса, суффикса и подцепочки.

Цепочка β называется префиксом цепочки α, если α = βγ, и $|\beta| \leq |\alpha|$.

Цепочка β называется суффиксом цепочки α, если α = γβ, и $|\beta| \leq |\alpha|$.

Цепочка β называется подцепочкой цепочки α, если α = γβδ, $|\beta| \leq |\alpha|$.

Например, если цепочка равна "abcd", то:

- цепочки "a", "ab", "abc" и "abcd" являются префиксами;
- цепочки "d", "cd", "bcd", и "abcd" являются суффиксами;
- любая часть этой цепочки, включая саму цепочку, является подцепочкой, например, "ab", "bcd", "cd"; префиксы и суффиксы, — это тоже подцепочки. Пустая цепочка λ является одновременно префиксом, суффиксом и подцепочкой некоторой цепочки α.

1.1.3. Формальные языки

Несмотря на кажущуюся сложность понятия «язык», в теории формальных языков это всего лишь множество цепочек. Для обозначения множеств возможных цепочек над алфавитом Σ в теории языков используют следующие обозначения:

Σ* — множество всех возможных цепочек над Σ, включая пустую;

Σ^+ — множество всех возможных цепочек над Σ, исключая пустую.

Σ^n — множество всех возможных цепочек над Σ длиной n.

Формально язык определяется при помощи Σ* следующим образом.

Язык L над алфавитом (или в алфавите) Σ: $L(\Sigma)$ — это подмножество цепочек из множества Σ*: $L(\Sigma) \subseteq \Sigma^*$.

Пусть заданы два языка: $L_1(\Sigma)$ и $L_2(\Sigma)$. Тогда:

1. Язык L_1 включает в себя язык L_2: $L_2 \subseteq L_1$, если любая цепочка L_2 входит в L_1: $\alpha \in L_2$: $\alpha \in L_1$.

2. Языки L_1 и L_2 эквивалентны: $L_1 = L_2$, если одновременно $L_2 \subseteq L_1$, и $L_1 \subseteq L_2$.

3. Языки L_1 и L_2 почти эквивалентны: $L_1 \approx L_2$, если их множества цепочек различаются максимум на пустую цепочку: $L_1 \cup \{\lambda\} = L_2 \cup \{\lambda\}$.

В язык входят цепочки, удовлетворяющие некоторым правилам.

Например, в русский язык входят цепочки, удовлетворяющие правилам грамматики, синтаксиса и семантики русского языка. В язык программирования Pascal входят цепочки, соответствующие лексическим, синтаксическим и семантическим правилам этого языка.

1.1.4. Способы задания языков

Задать язык можно одним из следующих способов:

1) перечислить все цепочки языка;

2) указать процедуру порождения цепочек (например, формулу);

3) определить метод распознавания допустимых цепочек.

Перечислить все цепочки языка возможно только в отдельных случаях. Было бы невозможно выучить русский или иной язык, если бы нужно было заучивать все возможные тексты на этом языке.

С другой стороны, некая программная система может управляться небольшим числом команд, таких, как «открыть», «закрыть», «выход». Перечислить все цепочки в этом случае гораздо проще.

Для порождения цепочек многих языков известны или можно найти простые процедуры. Вместо заучивания текстов обучаемым предлагают выучить правила, по которым строятся правильные слова и предложения, из которых затем получаются правильные тексты.

В языках программирования и того проще. Например, для порождения двоичного целого числа нужно взять цифру 0 или 1, и произвольное число раз приписать к ней справа цифру 0 или 1. Аналогично задается язык, описывающий идентификаторы: возьмите букву, и припишите справа произвольное количество цифр или букв. В теории формальных языков для порождения цепочек языка используются формальные (порождающие) грамматики.

Для реализации третьего способа задания языка используется некоторое логическое устройство (распознаватель), которое, получив на входе цепочку, на выходе выдает «истина», если цепочка принадлежит языку, и «ложь» в противном случае. Трансляторы как раз используют этот способ. Каждый программист неоднократно сталкивался с ситуацией, когда транслятор обнаруживает ошибки в тексте программы, то есть отвергает программный текст, как не принадлежащий языку. Интересное следствие: каждый транслятор задает свой собственный язык.

1.2. Формальные грамматики

Формальная порождающая грамматика (*generative grammar*) — это математическая система, описывающая правила построения цепочек некоторого (формального) языка. Грамматики были введены в 50-х г.г. XX века американским лингвистом Н. Хомским (Avram Noam Chomsky) и впервые использованы при описании презентации языка Алгол-60 в так называемой форме Бэкуса-Наура (БНФ).

Формальную грамматику определяют следующие четыре элемента:

Σ — множество терминальных символов (константы);

N — множество нетерминальных символов (переменные);

P — множество продукций или правил грамматики (формулы);

$S \in N$ — целевой символ (аксиома), называемый также стартовым или начальным символом.

Грамматику записывают в виде $G(\Sigma, N, P, S)$ или $G = (\Sigma, N, P, S)$.

Заметим, что могут использоваться другие обозначения элементов, а также иной порядок элементов в записи. Например, встречается другой порядок элементов: (N, Σ, P, S), или одновременно и другой порядок, и другие обозначения: (T, N, S, R).

Рассмотрим отдельные элементы грамматики.

Терминальный символ (*терминал*, *константа*) — это знак алфавита, используемого при построении цепочек языка (иначе говоря, цепочки языка состоят только из терминальных символов). Название «терминальный» (конечный) связано с тем, что цепочка терминалов завершает процесс порождения, так как в ней нет переменных.

Нетерминальный символ (*нетерминал*, *переменная*) — это символ вспомогательного алфавита, используемого для обозначения вспомогательных (промежуточных) цепочек языка. Один из нетерминальных символов помечается как начальный: он обозначает цель языка и соответствует порождаемым в конечном итоге правильным цепочкам.

Множества терминальных и нетерминальных символов не пересекаются: $\Sigma \cap N = \varnothing$. Объединение множеств терминальных и нетерминальных символов составляет *полный алфавит грамматики* V: $V = \Sigma \cup N$.

Продукция (*rewriting rule*, *правило подстановки*, *формула*) — это правило вывода, состоящее из упорядоченной пары цепочек (α, β). Во время порождения цепочка β заменяет цепочку α. Правую часть продукции называют также телом продукции.

Пример простейшей продукции: $\alpha \rightarrow \beta$.

Эта продукция читается одним из следующих способов: «α по определению есть β», «из α следует β», «α — это β».

Для практического использования продукции нумеруют. Множество всех продукций грамматики называют также «схемой подстановки».

Язык, порождаемый грамматикой G, обозначают $L(G)$ или L_G.

Грамматики не являются алгоритмами, так как они не задают порядок применения правил. Их следует рассматривать как устройство (в оригинале device [1]) для генерирования цепочек некоторого языка.

В форме Бэкуса-Наура:

- между цепочками α и β записывается символ вывода "::=";
- нетерминальные символы заключаются в треугольные скобки, что дает возможность придавать им осмысленные названия;
- все цепочки β, соответствующие одной цепочке α, записываются в форме одного правила с помощью вертикальной черты, имеющей смысл «или», например:

$\alpha \rightarrow \beta \mid \gamma \mid \delta$ (альфа это бета, или гамма, или дельта).

Запись только правил грамматики одновременно определяет множества терминалов и нетерминалов, при этом целевой символ определяется по символу левой части первого правила или по соглашению.

Грамматики описывают правила порождения текстов и используются для проверки принадлежности некоторой цепочки некоторому языку.

Практическое применение той или иной грамматики определяется возможностью решения *проблемы распознавания*, — существует или нет алгоритм, который за конечное число шагов определяет принадлежность цепочки заданному языку. Если такой алгоритм существует, язык называется распознаваемым. Если к тому же время распознавания зависит от длины цепочки и его можно вычислить до начала распознавания, то язык называется *легко распознаваемым*.

1.2.1. Примеры грамматик

1) Следующая простейшая грамматика описывает язык чисел из одной-единственной двоичной цифры:

$$G = (\{0, 1\}, \{S\}, \mathrm{P}, S), \mathrm{P} = \{$$
$$S \to {}^{(1)} 0 \mid {}^{(2)} 1$$
$$\}. \qquad (1.1)$$

Множество терминалов состоит из цифр 0 и 1. Множество нетерминалов состоит из символа *S*, который является целевым, и обозначает запись числа из одной цифры. Первое правило имеет смысл: «число из одной двоичной цифры — это 0», второе — «число из одной двоичной цифры — это 1». Грамматика порождает две цепочки: "0" и "1".

Порождение начинается с целевого символа грамматики:

S

Далее ищем правила, в левой части которых находится нетерминал *S*. Есть два таких правила. Применяя правило 1, мы заменяем цепочку *S* телом правила, и получаем цепочку "0":

$S \Rightarrow^{(1)} 0$

Полученная новая цепочка состоит из терминала, следовательно, порождение цепочки завершено, в цепочке нет переменных. Аналогичным образом, применяя к цепочке *S* правило 2, получаем цепочку "1".

2) Чтобы описать язык целых двоичных чисел, в грамматику (1.1) нужно ввести рекурсивное правило. Задаваясь вопросом, как породить целое двоичное число, можно предложить следующую процедуру из двух шагов:

- целое двоичное число из одной цифры — это 0 или 1.
- целое двоичное число из нескольких цифр — это имеющееся целое двоичное число, к которому приписана цифра 0 или 1.

Записывая эти рассуждения формально, получим грамматику:

$$G = (\{d\}, \{S\}, \mathrm{P}, S), \mathrm{P} = \{$$
$$S \to {}^{(1)} 0 \mid {}^{(2)} 1 \mid {}^{(3)} 0S \mid {}^{(4)} 1S$$
$$\}. \qquad (1.2)$$

При помощи правил 1 и 2 можно получить число из одной цифры. Чтобы получить число из двух цифр, сначала применим рекурсивное правило 3 или 4, которое позволит в полученной цепочке "0*S*" или "1*S*" заменить *S* на 0 или 1 (то есть приписать справа к 0 или 1 еще 0 или 1).

Грамматика порождает цепочки "0", "1", "00", "01", "10", "11", "000" и т.д. Рассмотрим, как грамматика порождает цепочку "01".

Применяя к целевому символу *S* правило 3, получаем:

$S \Rightarrow^{(3)} 0S$

Новая цепочка содержит нетерминал *S*. Применяя к нему правило 2, получаем цепочку терминалов:

$S \Rightarrow^{(3)} 0S \Rightarrow^{(2)} 01$

Таким образом, процесс порождения допустимых цепочек языка заключается в поочередной замене всех нетерминальных символов так, чтобы получить цепочку, состоящую только из терминалов.

3) Следующая грамматика описывает тот же самый язык целых двоичных чисел, что и грамматика (1.2):

$$
\begin{aligned}
&G = (\{0, 1\}, \{S, A\}, \mathrm{P}, S), \mathrm{P} = \{\\
&\quad S \rightarrow {}^{(1)} A \mid {}^{(2)} SA,\\
&\quad A \rightarrow {}^{(3)} 0 \mid {}^{(4)} 1\\
&\}.
\end{aligned}
\qquad (1.3)
$$

Здесь вместо терминалов 0 и 1 в правилах для *S* используется нетерминал *A*, который может быть заменен непосредственно цифрой.

Грамматика порождает запись "010" следующим образом:

$S \Rightarrow^{(2)} SA \Rightarrow^{(2)} SAA \Rightarrow^{(1)} AAA \Rightarrow^{(3)} 0AA \Rightarrow^{(4)} 01A \Rightarrow^{(3)} 010$

4) Рассмотрим грамматику, описывающую синтаксическую конструкцию «объявление переменной» в стиле языка Си:

$$
\begin{aligned}
&G = (\{t, ";", i, ","\}, \{D, L\}, \mathrm{P}, D), \mathrm{P} = \{\\
&\quad D \rightarrow {}^{(1)} t\ L\ ;\\
&\quad L \rightarrow {}^{(2)} i \mid {}^{(3)} L\ ,\ i\\
&\}.
\end{aligned}
\qquad (1.4)
$$

В этой грамматике:

- символ *t* обозначает тип объявляемых переменных, такой, как int;
- символ *i* обозначает произвольный идентификатор;
- символы ";" и "," обозначают сами себя.
- символ *D* является целевым и обозначает оператор объявления;
- символ *L* обозначает список объявляемых переменных.

Символы ";" и "," заключены в кавычки, чтобы отличить их от соответствующих знаков пунктуации.

Правило $D \rightarrow t\ L$ ";" имеет смысл: строка объявления — это тип, список переменных, точка с запятой.

Правило $L \rightarrow i$ имеет смысл: список — это идентификатор.

Правило $L \rightarrow L\ ,\ i$ имеет смысл: список — это (имеющийся) список переменных, запятая, идентификатор.

Грамматика порождает цепочку "*t i* ;" следующим образом:

$D \Rightarrow^{(1)} t\ L\ ; \Rightarrow^{(2)} t\ i\ ;$

Цепочка вида "*t i* , *i* ;" порождается так:

$D \Rightarrow^{(1)} t\ L\ ; \Rightarrow^{(3)} t\ L\ ,\ i\ ; \Rightarrow^{(2)} t\ i\ ,\ i\ ;$

5) Следующая важная грамматика описывает язык арифметических выражений со скобками, мы дадим ей специальное обозначение G_1:

$$
\begin{aligned}
&G = (\{+, *, (,)\}, \{E, T, P\}, \mathrm{P}, E), \mathrm{P} = \{\\
&\quad E \to {}^{(1)}\ E+T \mid {}^{(2)}\ T\\
&\quad T \to {}^{(3)}\ T*P \mid {}^{(4)}\ P\\
&\quad P \to {}^{(5)}\ a \mid {}^{(6)}\ (E)\\
&\}. \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad G_1
\end{aligned}
$$

Грамматика G_1 построена по иерархическому принципу, где каждый уровень иерархии определяет приоритет операций. Самый высокий приоритет находится на уровне нетерминала P (*primary*), описывающего элемент данных a (число) и выражение в скобках.

Следующий уровень описывает последовательность умножений, а также делений, если добавить соответствующее правило. Эта последовательность называется термом и описывается нетерминалом T (*term*).

Следующий уровень описывает выражение в целом и соответствует нетерминалу E (*expression*). Выражение — это последовательность сумм термов (а также разностей, если добавить соответствующее правило).

Эту грамматику можно расширить, добавив в нее дополнительные уровни для операций отношений, логических и других, поэтому она в той или иной форме входит в каждый язык программирования, в котором есть вычисление выражений.

Цепочку "$a+a*a$" можно вывести в этой грамматике за восемь шагов следующим образом:

$$
\begin{aligned}
&E \Rightarrow^{(1)} E+T \Rightarrow^{(2)} T+T \Rightarrow^{(4)} P+T \Rightarrow^{(5)} a+T \Rightarrow^{(3)} a+T*P \Rightarrow^{(4)} a+P*P \Rightarrow\\
&\Rightarrow^{(5)} a+a*P \Rightarrow^{(5)} a+a*a
\end{aligned}
$$

Эту грамматику нужно знать наизусть.

1.2.2. Соглашение о записи грамматик

В настоящем пособии нетерминалы грамматик всегда записываются одиночными прописными буквами (возможно с индексами), терминалы — одиночными строчными буквами (возможно с индексами), или непосредственно знаками. Символ грамматики, который может быть терминальным или нетерминальным, обычно обозначается нами как X. Если возможно неоднозначное или неверное толкование какого-либо символа, его тип дополнительно уточняется.

Кроме того, без необходимости грамматика записывается в форме, содержащей только правила, при этом целевой символ либо содержится в левой части первого правила, либо указывается явно, либо целевым символом считается S (при этом S определен в грамматике).

1.2.3. Запись грамматик с помощью метасимволов

Наличие в БНФ рекурсивных правил усложняет грамматику и затрудняет ее прочтение. В расширенной форме Бэкуса-Наура (РБНФ), правило записывается в форме

идентификатор = выражение **.**

Здесь "идентификатор" обозначает нетерминал, "=" заменяет "::=", "выражение" — это цепочка нетерминалов, терминалов и метасимволов, точка "**.**" обозначает конец правила. Метасимволы [] и {} позволяют простым образом записывать повторения:

- [выражение] — выражение в скобках включается в цепочку либо один раз, либо ни разу;
- {выражение} — выражение в скобках повторяется в цепочке любое число раз, в том числе ни разу.

Круглые скобки группируют выражения обычным образом. В РБНФ грамматика целого двоичного числа записывается следующим образом:

целое-двоичное-число = цифра { цифра } .

цифра = "0" | "1" **.**

Заметим, что цепочки терминалов в РБНФ заключены в двойные кавычки. Пример использования метасимволов []:

целое-двоичное-число-со-знаком = ["+" | "–"] цифра { цифра } .

цифра = "0" | "1" **.**

1.2.4. Синтаксические диаграммы

Синтаксические диаграммы — способ графического представления грамматик. Терминал изображается в виде круглого узла, нетерминал — в виде прямоугольного узла. Правило изображается в виде направленного графа, прохождение которого из начальной точки по доступным направлениям включает в вывод значения всех терминалов.

Если во время обхода встречается узел нетерминала, нужно перейти на диаграмму его правила, обойти ее, вернуться в исходную диаграмму. В качестве примера на рисунке 1.1 изображены синтаксические диаграммы грамматики, описывающей двоичные целые числа.

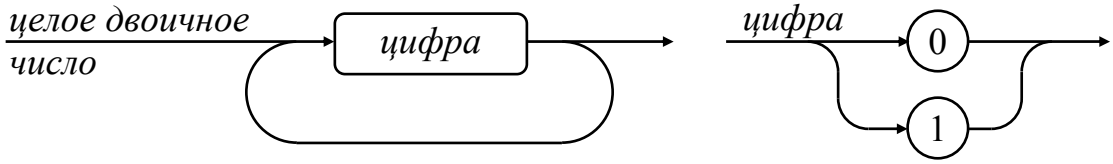

Рисунок 1.1. Запись грамматики в форме синтаксических диаграмм

1.3. Классификация грамматик и языков

Классификация грамматик и языков по Н. Хомскому (1956-1959) состоит из 4-х типов или классов. Критерием, относящим грамматику к тому или иному классу, является структура правил грамматики. Далее подразумеваются обозначения $G = (\Sigma, N, P, S)$, $V = \Sigma \cup N$.

Тип 0. Грамматики общего вида (*unrestricted grammar*).

На структуру правил не накладывается никаких ограничений:

$\varphi \to \psi, \quad \varphi \in V^+, \quad \psi \in V^*$

то есть из любой непустой цепочки φ можно вывести произвольную цепочку ψ. Языки общего вида являются рекурсивно перечислимыми множествами, эквивалентными по мощности общей машине Тьюринга. Эти языки и грамматики нам не интересны вследствие их сложности и общности.

Тип 1. Контекстно-зависимые (КЗ, *context-sensitive grammar*), а также неукорачивающие грамматики.

На структуру правил накладываем ограничение 1: если в грамматике существует правило $\varphi \to \psi$, то тогда $\varphi = \alpha_1 A \alpha_2$, $\psi = \alpha_1 \beta \alpha_2$. Иначе говоря, из нетерминала A выводится непустая цепочка β, только если и A и β находятся в одинаковом окружении α_1 и α_2, называемом контекстом:

$\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2, \quad \alpha_1, \alpha_2 \in V^*, \quad A \in N, \quad \beta \in V^+$

Контекстно-зависимые языки распознаваемые: принадлежность цепочки языку можно установить за ограниченное число шагов. Языки типа 1 порождают также неукорачивающие грамматики. Правила этих грамматик имеют общий вид, но порождаемые цепочки только растут:

$\varphi \to \psi, \quad \varphi, \psi \in V^+, \quad |\varphi| \le |\psi|$

Языки типа 1 по мощности эквивалентны линейно-ограниченным автоматам. Они лишь частично описывают естественные языки, и могут применяться для описания языков программирования, но распознаватели этих языков слишком сложны и недостаточно производительны.

Тип 2. Контекстно-свободные, или бесконтекстные грамматики (КС, *context-free grammar*, CFG).

На структуру правил накладываем ограничение 2: если в грамматике существует правило $\varphi \to \psi$, то тогда $\varphi = \alpha_1 A \alpha_2$, $\psi = \alpha_1 \beta \alpha_2$, $\alpha_1 = \alpha_2 = \lambda$. Иначе говоря, из нетерминала A выводится произвольная цепочка β:

$A \to \beta, \quad A \in N, \quad \beta \in V^*$

Контекстно-свободные языки легко распознаваемые, а их мощность эквивалентна автоматам с магазинной памятью (МП-автоматам). Бесконтекстные грамматики чрезвычайно важны, обычно именно ими описывают языки программирования. Для распознавания бесконтекстных языков существуют очень эффективные методы.

Различают укорачивающие и неукорачивающие КС-грамматики, сокращенно УКС и НКС. Приведенные выше правила относятся к УКС грамматикам. Для НКС грамматик правила отличаются тем, что цепочка β не может быть пустой:

$A \to \beta, \quad A \in \mathrm{N}, \quad \beta \in V^+$

Языки УКС и НКС грамматик почти эквивалентны.

Тип 3. Регулярные, или автоматные грамматики.

На структуру правил накладываем ограничение 3: если в грамматике существует правило $\varphi \to \psi$, то тогда $\varphi = A$, $\psi = a \mid aB$, a — терминал:

$A \to a \mid aB, \quad A, B \in \mathrm{N}, \quad a \in \Sigma$

Регулярные языки обладают наименьшей выразительной мощностью и эквивалентны конечным автоматам, их называют также автоматными языками. Они описывают простейшие языковые конструкции — лексемы (слова). Существуют эквивалентные грамматики, в них нетерминал B находится с другой стороны терминала a:

$A \to Ba \mid a, \quad A, B \in \mathrm{N}, \quad a \in \Sigma$

Первые грамматики праволинейные (*right-linear*), вторые — леволинейные (*left-linear*). Различаются они только порядком вывода цепочек.

Если L_i — язык типа i, то справедливо $L_0 \subseteq L_1 \subseteq L_2 \subseteq L_3$. Соотношение грамматик такое же: грамматика типа 3 является грамматикой типов 2, 1 и 0, грамматика типа 2 является грамматикой типов 1 и 0. Грамматика типа 1 является грамматикой типа 0 (рисунок 1.2).

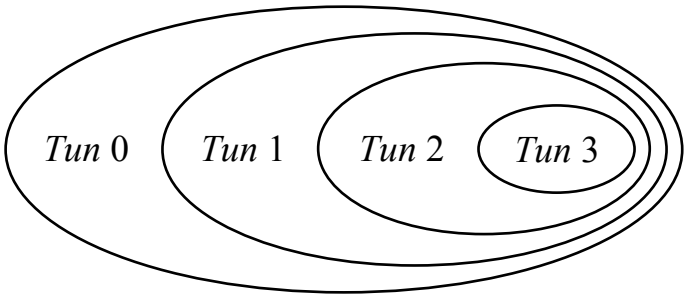

Рисунок 1.2. Соотношение типов грамматик и языков

Можно выделить еще подкласс линейных (*linear*) грамматик:

$A \to \alpha B \beta \mid \gamma, \quad A, B \in \mathrm{N}, \quad \alpha, \beta, \gamma \in \Sigma^*$

В иерархии они занимают промежуточное положение между бесконтекстными и регулярными грамматиками, и могут порождать цепочки как бесконтекстных, так и регулярных языков.

Класс грамматики определяется по правилу, которое относит ее к наиболее сложному типу. Так, если все правила грамматики удовлетворяют ограничению типа 2, а одно правило удовлетворяет ограничению типа 1, то вся грамматика является грамматикой типа 1.

Грамматики определяют языки, поэтому тип грамматики косвенно указывает на тип языка. Так как один и тот же язык можно задать с помощью разных по типу грамматик, тип языка определяется наибольшим по номеру типом грамматики, его определяющей.

Заметим, что практически ценные грамматики типов 3 и 2 имеют вид правил, в левой части которых находится один-единственный нетерминальный символ. Выводы в этих грамматиках могут быть представлены деревьями, что упрощает построение распознавателя языка.

1.3.1. Примеры классификации грамматик и языков

Классифицируем каждое правило. Если слева единственный нетерминал, тип правила 2/3, иначе 0/1. Если тип правила 2/3, и в теле не более одного нетерминала, тип правила 3. Если тип правила 0/1 и правило не укорачивающее, тип правила 1. Грамматика типа 3, если все правила типа 3, и терминалы всегда слева или справа от нетерминалов. Грамматика типа 2, если минимальный тип правил 2, иначе типа 1 или 0.

Рассмотрим грамматику целого двоичного числа (1.3):

$S \to A \mid SA$
$A \to 0 \mid 1$

В левых частях один нетерминал. В правых частях есть более одного нетерминала. Класс грамматики 2 (контекстно-свободная).

Другая грамматика, порождающая тот же язык, имеет вид (1.2):

$S \to 0 \mid 1 \mid 0S \mid 1S$

В левых частях один нетерминал. В правых частях не более одного нетерминала и терминалы всегда слева. Класс грамматики 3 (автоматная, причем праволинейная). Следовательно, язык целых двоичных чисел является автоматным.

Следующая грамматика описывает язык цепочек вида $a^n b^n a^n$, $n > 0$:

$S \to ABA$
$A \to$ a
$B \to ABCA \mid b$
$bC \to bb$
$AC \to DC$
$DC \to DA$
$DA \to CA$

Грамматика относится к классу 0 или 1, — в левых частях правил встречаются пары символов. Грамматика является неукорачивающей, то есть имеет класс 1. Порождаемый язык является контекстно-зависимым.

1.4. Выводы в грамматиках

Пусть есть грамматика $G = (\Sigma, N, P, S)$, $V = \Sigma \cup N$. Если в G есть правило $w \to x$, то оно задает отношение между цепочками $\gamma_1 w \gamma_2$ и $\gamma_1 x \gamma_2$, называемое *выводимостью*, и обозначаемое стрелкой $\Rightarrow_G$. Если понятно, о какой грамматике идет речь, знак G часто опускают и пишут $\Rightarrow$.

Введем несколько определений.

Будем называть шагом вывода однократное применение какой-либо продукции к цепочке. Последовательность цепочек, полученных в шагах вывода — это цепочка вывода или вывод (*derivation*).

1. Непосредственная выводимость: $\alpha \Rightarrow \beta$.

Цепочка β называется непосредственно выводимой из цепочки α, если β можно представить как $\gamma_1 x \gamma_2$, α можно представить как $\gamma_1 w \gamma_2$, и в грамматике существует правило $w \to x$, $\gamma_1, \gamma_2, x \in V^*$, $w \in V^+$. Если одновременно $\gamma_1 = \lambda$ и $\gamma_2 = \lambda$, то в грамматике должно существовать правило $\alpha \to \beta$.

Непосредственная выводимость соответствует ровно одному шагу.

2. Нетривиальная выводимость: $\alpha \Rightarrow^+ \beta$.

Транзитивное замыкание $\Rightarrow^+$ отношения $\Rightarrow$ называется нетривиальной выводимостью. Цепочка β нетривиально выводима из цепочки α, если вывод содержит один или более шагов. Если число шагов известно и равно k, вместо знака "+" может быть указано число k: $\alpha \Rightarrow^k \beta$. В этом случае говорят об отношении k-той степени.

3. Выводимость: $\alpha \Rightarrow^* \beta$.

Рефлексивное замыкание $\Rightarrow^*$ отношения $\Rightarrow^+$ называется выводимостью. Цепочка β называется выводимой из цепочки α тогда и только тогда, когда либо $\alpha = \beta$, либо $\alpha \Rightarrow^+ \beta$. Это отношение соответствует нулю и более шагов вывода.

В качестве примера будем рассматривать выводы в грамматике (1.3) целых двоичных чисел, правила которой имеют вид:

$S \to {}^{(1)} A \mid {}^{(2)} SA$
$A \to {}^{(3)} 0 \mid {}^{(4)} 1$

Пусть есть вывод в грамматике (1.3):

$S \Rightarrow SA \Rightarrow SAA \Rightarrow AAA \Rightarrow 0AA \Rightarrow 01A \Rightarrow 010$

Тогда следующие записи показывают тот же самый вывод:

$S \Rightarrow SA \Rightarrow^* 01A \Rightarrow 010$
$S \Rightarrow^* 0AA \Rightarrow 01A \Rightarrow 010$
$S \Rightarrow SA \Rightarrow SAA \Rightarrow AAA \Rightarrow^* 010$
$S \Rightarrow^* 010$
$S \Rightarrow^+ 010$
$S \Rightarrow^6 010$

4. Вывод называется законченным, если из цепочки, полученной в результате вывода, нельзя сделать ни одного шага, а цепочка содержит только терминальные символы.

Следующие выводы являются законченными:

$S \Rightarrow SA \Rightarrow SAA \Rightarrow AAA \Rightarrow 0AA \Rightarrow 01A \Rightarrow 010$

$AAA \Rightarrow 0AA \Rightarrow 01A \Rightarrow 010$

$01A \Rightarrow 010$

5. Цепочка α называется сентенциальной формой или сентенцией грамматики, если она выводится из целевого символа.

Следующие результирующие цепочки — сентенции:

$S \Rightarrow SA \Rightarrow SAA \Rightarrow AAA \Rightarrow 0AA \Rightarrow 01A \Rightarrow 010$

$S \Rightarrow SA \Rightarrow SAA \Rightarrow AAA$

$S \Rightarrow SA$

6. Цепочка α называется конечной сентенциальной формой, если она выводится из целевого символа, и вывод закончен.

Следующие результирующие цепочки — конечные сентенции:

$S \Rightarrow SA \Rightarrow SAA \Rightarrow AAA \Rightarrow 1AA \Rightarrow 01A \Rightarrow 010$

$S \Rightarrow SA \Rightarrow S1 \Rightarrow A1 \Rightarrow 01$

$S \Rightarrow A \Rightarrow 0$

Тогда определение формального языка может быть следующим.

7. Язык — это множество всех конечных сентенциальных форм:

$L(G) = \{ \alpha \in \Sigma^* \mid S \Rightarrow^* \alpha \}$

8. Вывод называется левосторонним (обозначается *lm*, *leftmost derivation*), если на каждом шаге вывода продукция применяется к крайнему левому нетерминалу цепочки.

Пример левостороннего вывода:

$S \Rightarrow_{lm} SA \Rightarrow_{lm} SAA \Rightarrow_{lm} AAA \Rightarrow_{lm} 0AA \Rightarrow_{lm} 01A \Rightarrow_{lm} 010$

9. Вывод называется правосторонним (обозначается *rm*, *rightmost derivation*), если на каждом шаге вывода продукция применяется к крайнему справа нетерминалу.

Пример правостороннего вывода:

$S \Rightarrow_{rm} SA \Rightarrow_{rm} S0 \Rightarrow_{rm} SA0 \Rightarrow_{rm} S10 \Rightarrow_{rm} A10 \Rightarrow_{rm} 010$

10. Левовыводимая цепочка — это сентенциальная форма левостороннего вывода. Все цепочки пункта 8 являются левовыводимыми.

11. Правовыводимая цепочка — это сентенциальная форма правостороннего вывода. Все цепочки пункта 9 являются правовыводимыми.

12. Обращенное порождение — обратный вывод цепочки от цепочки терминалов к целевому символу.

Пример обращенного вывода:

$010 \Rightarrow^{(3)} 01A \Rightarrow^{(4)} 0AA \Rightarrow^{(3)} AAA \Rightarrow^{(1)} SAA \Rightarrow^{(2)} SA \Rightarrow^{(2)} S$

1.5. Деревья вывода

Пусть задана грамматика $G = (\Sigma, N, P, S)$, $V = \Sigma \cup N$, и вывод $S \Rightarrow^* \alpha$.

Дерево вывода (дерево разбора) — это граф, в котором:

- корневой узел обозначен целевым символом S,
- другие узлы обозначены нетерминалами,
- листья обозначены терминалами или нетерминалами,
- если $A \to X_1 X_2 \ldots X_n \in P$, $X_i \in V$, узел A имеет n потомков $X_1, X_2, \ldots, X_n$.

Дерево соответствует некоторому выводу. Перечень листьев дерева слева направо называется *кроной*. Если вывод не законченный, листьями окажутся некоторые нетерминалы. Если вывод законченный, крона состоит только из терминалов.

1.5.1. Построение дерева сверху вниз

Для построения дерева нужна цепочка левостороннего вывода.

Сначала в корневой узел помещается целевой символ, который «раскрывается» в узлы и листья нижележащего уровня в соответствии с первым правилом в цепочке. Далее последовательно раскрываются узлы нетерминальных символов в соответствии с последующими правилами.

Рассмотрим построение дерева для вывода в грамматике (1.3):

$S \Rightarrow^{(2)} S_1A_1 \Rightarrow^{(2)} S_2A_2A_1 \Rightarrow^{(1)} A_3A_2A_1 \Rightarrow^{(3)} 0A_2A_1 \Rightarrow^{(4)} 01A_1 \Rightarrow^{(3)} 010$

Сначала добавляем узел с целевым символом S (рисунок 1.3).

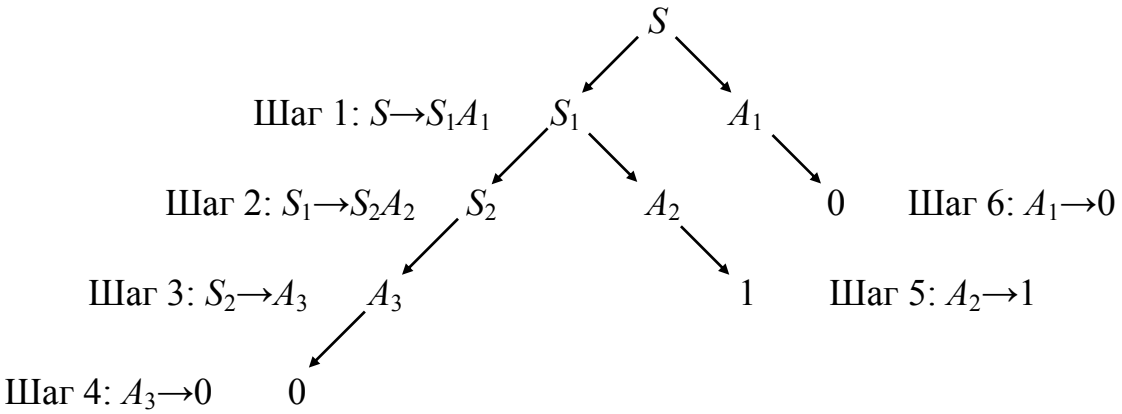

Рисунок 1.3. Построение дерева вывода сверху вниз

Первый шаг вывода заменяет S на два символа S_1 и A_1. На дереве из узла S при этом появляются два ребра, ведущие к узлам S_1 и A_1. Второй шаг вывода заменяет символ S_1 символами S_2 и A_2, поэтому из узла S_1 появляются два ребра к узлам S_2 и A_2. Далее узел S_2 раскрывается в узел A_3, который раскрывается в лист "0". Аналогичным образом строятся другие ветки дерева (шаги 5 и 6).

1.5.2. Построение дерева снизу вверх

Для построения дерева нужна цепочка правостороннего вывода. Для ручного построения лучше иметь обращенный вывод. Правосторонний вывод в грамматике (1.3) той же цепочки 010:

$S \Rightarrow_{rm}^{(2)} S_1A_1 \Rightarrow_{rm}^{(3)} S10 \Rightarrow_{rm}^{(2)} S_2A_20 \Rightarrow_{rm}^{(4)} S_210 \Rightarrow_{rm}^{(1)} A_310 \Rightarrow_{rm}^{(3)} 010$

Это обращенное правое порождение:

$010 \Rightarrow_{rm}^{(3)} A_310 \Rightarrow_{rm}^{(1)} S_210 \Rightarrow_{rm}^{(4)} S_2A_20 \Rightarrow_{rm}^{(2)} S10 \Rightarrow_{rm}^{(3)} S_1A_1 \Rightarrow_{rm}^{(2)} S$

Построим дерева вывода снизу вверх шаг за шагом, используя обращенное правое порождение и обращение правил (рисунок 1.4).

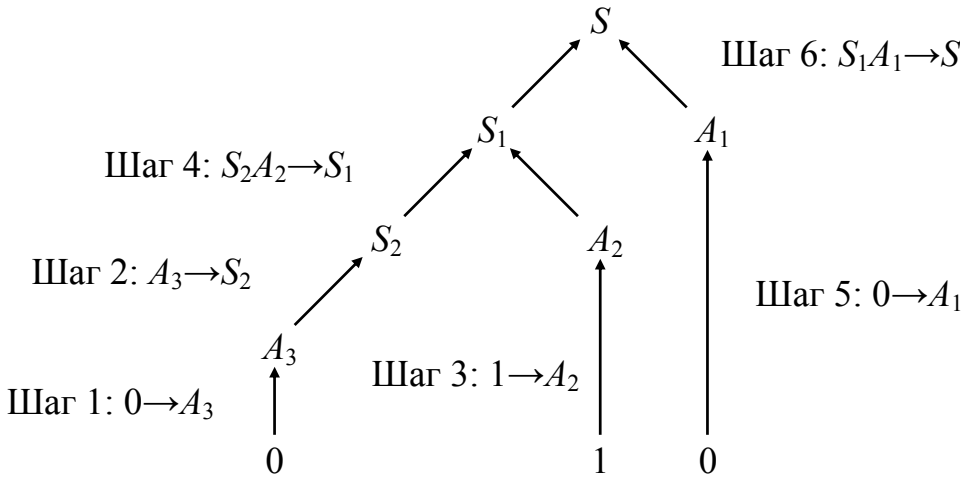

Рисунок 1.4. Построение дерева вывода снизу вверх

Шаг 1. Обращенное правило 3, $0 \rightarrow A$, раскрывает нетерминальный символ A3 в первый терминальный символ цепочки "0", поэтому сначала добавляются лист "0" и узел A_3, которые соединяются ребром.

Шаг 2. Следующим применяется обращенное правило 1, $A \rightarrow S$, в дерево добавляется узел S_2, который соединяется ребром с узлом A_3.

Шаг 3. Следующее обращенное правило 4, $1 \rightarrow A$, добавляет в дерево лист "1" и узел A_2, они также соединяются ребром.

Шаг 4. Следующее обращенное правило 2, $SA \rightarrow S$, добавляет в дерево узел S_1, который соединяется ребрами с узлами S_2 и A_2.

Шаг 5. Следующее обращенное правило 3, $0 \rightarrow A$, добавляет в дерево лист "0" и узел A_1, которые соединяются ребром.

Шаг 6. Наконец, рассматриваем обращенное правило 2, $SA \rightarrow S$, при этом в дерево добавляется узел с целевым символом S, который соединяется ребрами с узлами S_1 и A_1.

Результирующее дерево для данной цепочки должно полностью совпадать с деревом, построенным методом сверху вниз, и иметь одинаковые кроны.

1.5.3. Линейная запись дерева

Линейная (скобочная) запись дерева представляет собой последовательную запись узлов и листьев дерева при его обходе сверху вниз и слева направо, что соответствует левостороннему выводу. Правила построения линейной записи дерева предельно просты: узел записывается своим обозначением, после которого в скобках перечисляются потомки через запятую или пробел. Для дерева на рисунке 1.3:

- применяя правило к нетерминалу S, получим: S (S_1 A_1);
- применяя правило к нетерминалу S_1, получим: S (S_1 (S_2 A_2) A_1);
- применяя правило к нетерминалу S_2, получим:
S (S_1 (S_2 (S_3 A_3) A_2) A_1);
- применяя правило к нетерминалу A_3, получим:
S (S_1 (S_2 (S_3 A_3 ("0")) A_2) A_1);
- применяя правило к нетерминалу A_2, получим:
S (S_1 (S_2 (S_3 A_3 ("0")) A_2 ("1")) A_1);
- применяя правило к нетерминалу A_1, окончательно получим:
S (S_1 (S_2 (S_3 A_3 ("0")) A_2 ("1")) A_1 ("0")).

1.6. Неоднозначность грамматик

Рассмотрим грамматику, описывающую язык выражений со скобками, для которой мы используем специальное обозначение G_2:

$S \rightarrow {}^{(1)} S+S \mid {}^{(2)} S*S \mid {}^{(3)} a \mid {}^{(4)} (S)$ $\qquad G_2$

Для цепочки "$a+a*a$" в этой грамматике можно построить два дерева (рисунок 1.5), соответствующих двум правосторонним выводам:

1) $S \Rightarrow_{rm}{}^{(1)} S+S \Rightarrow_{rm}{}^{(2)} S+S*S \Rightarrow_{rm}{}^{(3)} S+S*a \Rightarrow_{rm}{}^{(3)} S+a*a \Rightarrow_{rm}{}^{(3)} a+a*a$

2) $S \Rightarrow_{rm}{}^{(2)} S*S \Rightarrow_{rm}{}^{(3)} S*a \Rightarrow_{rm}{}^{(1)} S+S*a \Rightarrow_{rm}{}^{(3)} S+a*a \Rightarrow_{rm}{}^{(3)} a+a*a$

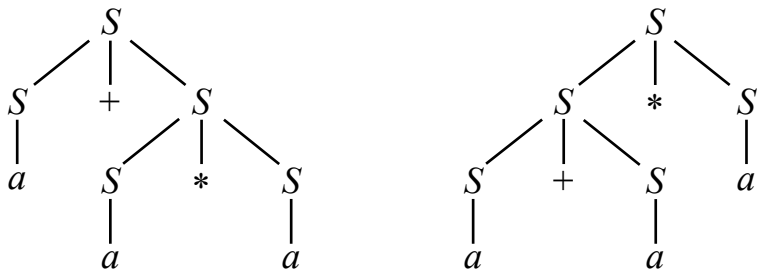

Рисунок 1.5. Неоднозначные выводы в грамматике G_2

С точки зрения формальных языков порядок вывода не имеет значения, так как формальные языки не несут никакой смысловой нагрузки. Однако с точки зрения языков программирования это не так.

Арифметические операции имеют разный приоритет, и порядок выполнения операций имеет значение. В первом выводе сначала выполняется операция умножения, а затем операция сложения, что является правильным порядком выполнения.

Ситуация, когда в грамматике можно построить более одного дерева вывода для одной и той же цепочки, называется неоднозначностью, а грамматика называется неоднозначной (*ambiguous*). И наоборот, если в грамматике для одной и той же цепочки можно построить только одно дерево вывода, то грамматика называется однозначной (*unambiguous*).

Неоднозначность — обычно свойство грамматики, а не языка. Для языка, заданного неоднозначной грамматикой, иногда можно найти эквивалентную однозначную грамматику. Например, тот же язык выражений со скобками описывается грамматикой G_1:

$$E \to {}^{(1)}\ E+T \mid {}^{(2)}\ T$$
$$T \to {}^{(3)}\ T*P \mid {}^{(4)}\ P$$
$$P \to {}^{(5)}\ a \mid {}^{(6)}\ (E) \qquad G_1$$

Цепочку "*a*+*a***a*" в этой грамматике можно вывести двумя способами, но однозначность определяется как единственность левостороннего или правостороннего вывода. Единственный левосторонний вывод был приведен ранее, единственный правосторонний вывод таков:

$$E \Rightarrow_{rm}^{(1)} E+T \Rightarrow_{rm}^{(3)} E+T*P \Rightarrow_{rm}^{(5)} E+T*a \Rightarrow_{rm}^{(4)} E+P*a \Rightarrow_{rm}^{(5)} E+a*a \Rightarrow$$
$$\Rightarrow_{rm}^{(2)} T+a*a \Rightarrow_{rm}^{(4)} P+a*a \Rightarrow_{rm}^{(5)} a+a*a$$

Единственное дерево вывода, соответствующее и левостороннему, и правостороннему выводу, приведено на рисунке 1.6.

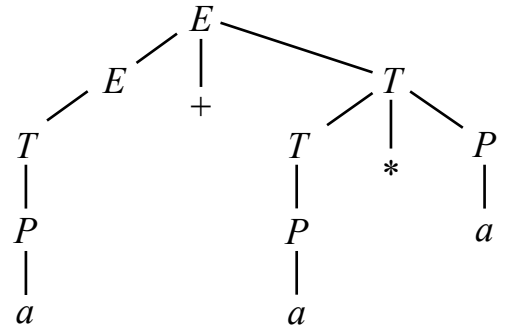

Рисунок 1.6. Однозначный вывод в грамматике G_1

Важной неоднозначной грамматикой является также грамматика, описывающая условный оператор в упрощенном виде:

$$S \to {}^{(1)}\ o \mid {}^{(2)}\ iS \mid {}^{(3)}\ iSeS \qquad (1.5)$$

Здесь "*o*" — это некоторый оператор, "*i*" — обобщение конструкции условного оператора «*if* выражение *then*», "*e*" — ключевое слово *else*.

На рисунке 1.7 приведены два дерева вывода цепочки "*iioeo*".

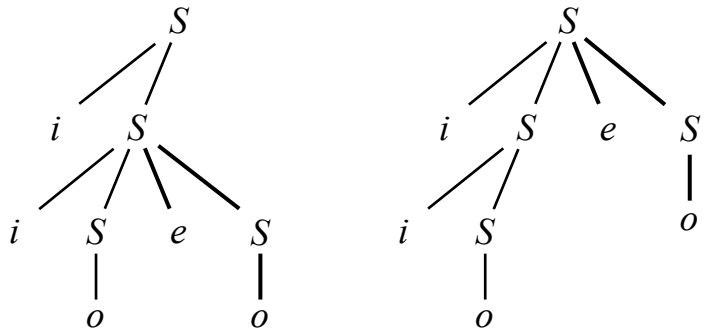

Рисунок 1.7. Неоднозначные выводы в грамматике (1.5)

Неоднозначность грамматики вида (1.5) носит название «*кочующий else*» (*dangling else*). Конструкция "*eS*" «как бы» перемещается от одного символа *S* к другому. По общепринятому соглашению, неоднозначность данной грамматики разрешается в пользу ближайшего в потоке слева символа, обозначающего *if*. Вывод, приведенный на рисунке 1.7 слева, является правильным.

Для грамматики (1.5) можно предложить следующую однозначную грамматику (1.6) [2]

$$S \to {}^{(1)}\,\mathrm{o} \mid {}^{(2)}\,iS \mid {}^{(3)}\,iS'eS$$
$$S' \to {}^{(4)}\,\mathrm{o} \mid {}^{(5)}\,iS'eS' \qquad (1.6)$$

Здесь конструкции *eS* и *eS′* привязываются к дополнительному символу *S′*. Это гарантирует, что конструкция *else* всегда будет относиться к ближайшему слева *if*.

Проблема однозначности грамматик типов 0, 1 и 2 является неразрешимой, — не существует метода или алгоритма, который бы позволил убедиться в том, что некоторая грамматика является однозначной.

Если для некоторого языка, заданного неоднозначной грамматикой, можно построить однозначную грамматику, возникает вопрос о доказательстве эквивалентности этих грамматик.

Проблема эквивалентности грамматик типов 0, 1 и 2 также неразрешима, — не существует метода или алгоритма, доказывающего, что две грамматики являются эквивалентными.

Для грамматик класса 2 существует ряд правил, наличие которых в грамматике ведет к ее неоднозначности:

1. $A \to AA \mid \alpha$
2. $A \to A\alpha A \mid \beta$
3. $A \to \alpha A \mid A\beta \mid \gamma$
4. $A \to \alpha A \mid \alpha A\beta A \mid \gamma$

Здесь $A \in \mathrm{N}$, $\alpha, \beta, \gamma \in V^*$.

Наличие в грамматике хотя бы одного из правил подобного вида ведет к ее неоднозначности. Так, в грамматике G_2 присутствуют правила вида 2, а в грамматике (1.5) — правила вида 4.

Заметим, что отсутствие в грамматике правил указанного вида не ведет автоматически к однозначности грамматики, — она может оказаться как однозначной, так и неоднозначной.

Существуют также языки, для которых нельзя найти однозначную грамматику. Эти языки называют существенно неоднозначными. В [6] приводится пример языка, формула которого достаточно сложна:

$L_{abcd} = \{a^n b^n c^m d^m \mid n \geq 1, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n \geq 1, m \geq 1\}$.

В цепочках этого языка поровну знаков *a* и *b*, и *c* и *d*, либо поровну знаков *a* и *d*, и *b* и *c*. Следующая грамматика описывает язык L_{abcd}:

$S \rightarrow AB \mid C$
$A \rightarrow aAb \mid ab$
$B \rightarrow cBd \mid cd$
$C \rightarrow aCd \mid aDd$
$D \rightarrow bDc \mid bc$

На рисунке 1.8 приведены два дерева вывода цепочки "*aabbccdd*".

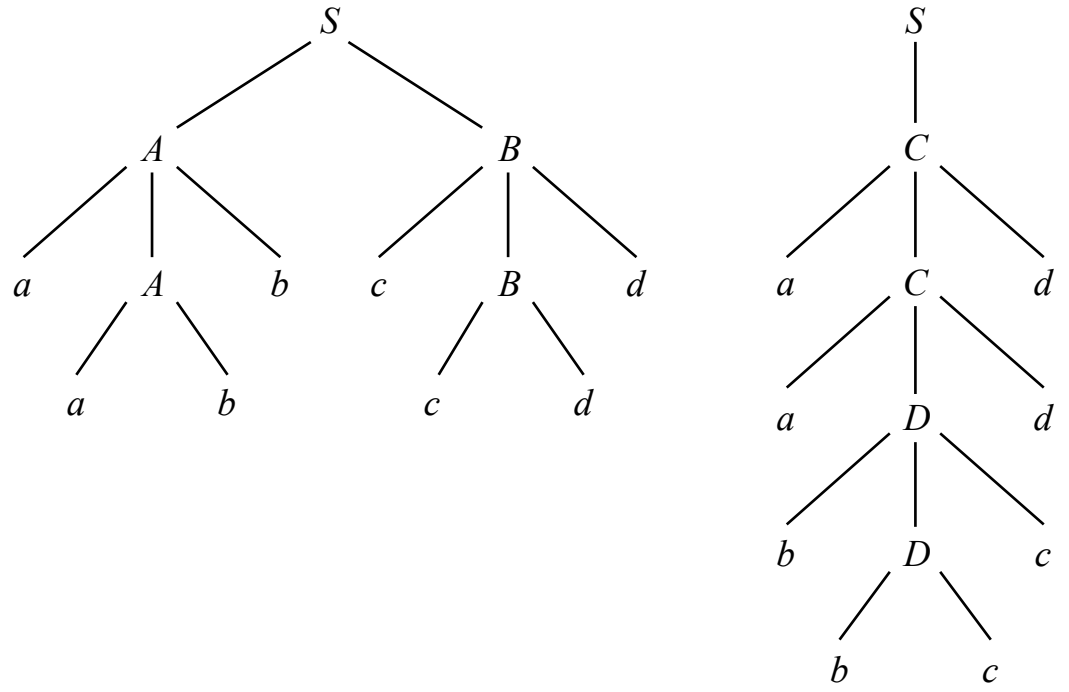

Рисунок 1.8. Два дерева вывода цепочки *aabbccdd*

Такие языки представляют скорее научный интерес, а не практический. В проектировании трансляторов неоднозначные грамматики применяются крайне редко, при этом однозначность выводов обеспечивается за счет других механизмов.

1.7. Преобразование грамматик

Преобразование грамматик к некоторому эквивалентному виду имеет целью получить определенную структуру правил. Это позволяет преобразованную грамматику в определенных алгоритмах распознавания.

При преобразованиях часто грамматика разрастается, в ней увеличивается количество нетерминальных символов и правил.

1.7.1. Приведенные грамматики

Эквивалентные преобразования применяются обычно к контекстно-свободным (КС) грамматикам. Поскольку любая регулярная грамматика по определению является контекстно-свободной, описываемые преобразования в полной мере применимы к регулярным грамматикам.

Для определенности далее будем говорить о преобразованиях именно КС-грамматик, как более общих.

Приведенной называют КС-грамматику, в которой нет бесплодных и недостижимых символов, пустых и цепных правил. Приведенную грамматику называют также грамматикой в каноническом виде.

Для преобразования произвольной КС-грамматики к каноническому виду необходимо:

1) сначала удалить пустые правила,
2) затем удалить цепные правила,
3) затем удалить бесплодные символы,
4) наконец, удалить недостижимые символы.

Замечания к преобразованиям

Далее в пособии используются следующие соглашения, сокращающие многократные повторения обозначений грамматик. Везде, где это не ведет к неоднозначности или неясности, буквой G будем обозначать запись $G=(\Sigma, N, P, S)$ и символы записи Σ, N, P, S, $V = N \cup \Sigma$. Буквой G со штрихом (или иным знаком) будем обозначать запись $G'=(\Sigma', N', P', S')$ и символы записи Σ', N', P', S', $V' = N' \cup \Sigma'$.

Большинство алгоритмов строят вспомогательные множества, которые затем используются для построения правил результирующей грамматики. Эти множества формируются в итерациях бесконечного цикла. Цикл завершается, когда в двух последовательных итерациях множество не изменяется.

По алгоритмам преобразований часто в грамматику должны добавляться одни и те же символы и правила, но практически дважды в одну грамматику ничего добавлять не нужно, алгоритмы подразумевают это, но неявным образом.

1.7.2. Устранение бесплодных символов

Будем называть символ X грамматики G порождающим, если он порождает цепочку терминальных символов, возможно, пустую. Заметим, что терминальный символ порождает сам себя за 0 шагов.

Будем называть символ X грамматики G бесплодным, если он не порождает никакой цепочки терминальных символов, даже пустой.

Алгоритм 1.1. Устранение бесплодных символов.

Шаг 1. Построение множества порождающих символов R.

Базис. Если $(A\to\alpha)\in P$ и $\alpha\in\Sigma^*$, то $A\in R$, $X\in R\ \forall X\in\alpha$.

Индукция. Если $(A\to\alpha)\in P$ и $\alpha\in(\Sigma\cup R)^*$, то $A\in R$, $X\in R\ \forall X\in\alpha\cap\Sigma$.

По определению базиса, сначала включаем в R все символы правила, тело которого содержит только терминалы, или пусто. По определению индукции, включаем в R все символы правила, тело которого содержит терминалы и символы из R. Соединяя определения, включаем в R все символы правила, тело которых содержит нетерминалы только из R.

Шаг 2. Удаление бесплодных символов и правил.

Если $(A\to\alpha)\in P$, и $A\in N$, и $\alpha\in R^*$, то $(A\to\alpha)\in P'$ — если все символы правила есть в R, то правило принадлежит целевой грамматике G'.

Шаг 3. Элементы грамматики G'. $\Sigma' = \Sigma\cap R$, $N' = N\cap R$, $S' = S$. ●

Пример 1.1. Устранение бесплодных символов.

Пусть грамматика G имеет следующие правила:

$$\begin{aligned} &S\to a\\ &S\to A\\ &A\to AB\\ &B\to C\\ &C\to b \end{aligned} \qquad (1.7)$$

В этой грамматике символ A является бесплодным, так как выводы из A всегда начинаются с A, и не порождаются терминальные цепочки.

На шаге 1 алгоритм выполняет итерации, просматривая все правила грамматики до тех пор, пока в R добавляются новые символы. Для G в первой итерации в R включаем символы S, a, C и b. Во второй итерации в R включаем символ B. В третьей итерации в R нечего добавить, итерации завершаются, $R = \{S, a, C, b, B\}$.

Выполняя шаг 2, получим правила грамматики G':

$$\begin{aligned} &S\to a\\ &B\to C\\ &C\to b \end{aligned} \qquad (1.8)$$

Мы удалили правила $S\to A$ и $A\to AB$, так как $A\notin R$.

Элементы грамматики: $\Sigma' = \{S, C, B\}$, $N' = \{a, b\}$, $S' = S$. ●

1.7.3. Устранение недостижимых символов

Будем называть символ X грамматики G недостижимым, если он не встречается ни в одной сентенциальной форме грамматики.

Алгоритм 1.2. Устранение недостижимых символов.

Шаг 1. Построение множества достижимых символов R.

Базис. $S \in R$. Индукция. Если $(A \to \alpha X \beta) \in \mathrm{P}$, и $A \in R$, то $X \in R$.

По определению базиса, сначала включаем в R целевой символ. По определению индукции, включаем в R символы, которые выводятся из символов, которые есть в R. Соединяя определения, включаем в R символы, которые выводятся из символов, которые есть в R.

Шаг 2. Удаление недостижимых символов.

Если $(A \to \alpha) \in \mathrm{P}$, и $A \in \mathrm{R}$, и $\alpha \in R^*$, тогда $(A \to \alpha) \in \mathrm{P}'$ — если левый символ правила есть в R, и в теле правила нет символов, которых нет в R, то это правило принадлежит целевой грамматике G'.

Шаг 3. Элементы грамматики G'. $\Sigma' = \Sigma \cap R$, $\mathrm{N}' = \mathrm{N} \cap R$, $S' = S$. ●

Пример 1.2. Устранение недостижимых символов.

Пусть грамматика G состоит из правил:

$S \to a$
$B \to C$
$C \to b$

В этой грамматике символы B, C и b являются недостижимыми.

На шаге 1 алгоритм выполняет итерации, просматривая все правила грамматики до тех пор, пока в R добавляются новые символы. Для G в первой итерации в R добавляется символ a. Во второй итерации в R нечего добавить, так как $B \notin R$, и $C \notin R$, итерации завершаются, $R = \{S, a\}$.

Выполняя шаг 2, получим единственное правило грамматики G':

$S \to a$

Мы удалили правила $B \to C$ и $C \to b$, так как $B \notin R$, и $C \notin R$.

Элементы грамматики: $\Sigma' = \{S\}$, $\mathrm{N}' = \{a\}$, $S' = S$. ●

Заметим, что если из грамматики (1.7) сначала удалить недостижимые символы, грамматика не изменится, — все символы в ней являются достижимыми. Удаление далее бесплодных символов, как было показано, приводит к грамматике (1.8).

1.7.4. Устранение пустых правил

Пустым, или λ-правилом, называется правило вида $A \to \lambda$, $A \in \mathrm{N}$.

Грамматика называется грамматикой без λ-правил, если в ней существует только одно пустое правило $S \to \lambda$, если $\lambda \in L(G)$, и при этом S не является правой частью ни одного правила грамматики.

Будем называть нетерминал A вырождающимся, если он порождает пустую цепочку: $A \Rightarrow^* \lambda$.

Алгоритм 1.3. Устранение пустых правил.

Шаг 1. Построение множества вырождающихся нетерминалов R.

Базис. Если $(A \to \lambda) \in P$, то $A \in R$.

Индукция. Если $(A \to \alpha) \in P$ и $\alpha \in R^+$, то $A \in R$.

По определению базиса, включаем в R символ A, если из него выводится пустая цепочка. По определению индукции, включаем в R символ A, если тело правила для A не содержит символов, которых нет в R. Соединяя определения, включаем в R символ A, если тело правила для A не содержит символов, которых нет в R.

Шаг 2. Удаление пустых правил.

Если $(A \to \alpha) \in P$, и $\alpha \neq \lambda$, то $(A \to \alpha) \in P'$ — если в грамматике G есть непустое правило, то это правило принадлежит грамматике G'.

Шаг 3. Компенсация удаленных пустых правил.

Если $(A \to \alpha X \beta) \in P'$, и $X \in R$, и $\alpha\beta \neq \lambda$, то $(A \to \alpha\beta) \in P'$ — если символ A порождает цепочку $\alpha X \beta$, и $X \in R$, то A должен порождать цепочку $\alpha\beta$, если она не пуста, и тогда правило для $A \to \alpha\beta$ также принадлежит G'.

Если алгоритм 1.3 корректен, то он доказывает, что для любой грамматики G найдется такая грамматика G' без пустых правил, для которой $L(G') = L(G) - \{\lambda\}$.

Чтобы обеспечить эквивалентность языков, порождаемых грамматиками G и G', нужно выполнить следующий дополнительный шаг.

Шаг 4. Целевой символ результирующей грамматики.

Если целевой символ S грамматики G не является вырождающимся, то он становится целевым символом грамматики G'. В противном случае добавим в грамматику целевой символ S' и правила $S' \to S \mid \lambda$. ●

Пример 1.3. Устранение пустых правил.

Пусть грамматика G состоит из правил:

$S \to AB$
$A \to a \mid BB$
$B \to b \mid \lambda$

В этой грамматике все нетерминалы вырождающиеся.

На шаге 1 алгоритм выполняет итерации, просматривая все правила грамматики до тех пор, пока в R добавляются новые символы. Для G в первой итерации в R включаем символ B, так как из него выводится пустая цепочка. Во второй итерации в R включаем символ A, так как из A выводится BB, и $B \in R$. В третьей итерации в R включаем символ S, так как из S выводится ABC, и $A \in R$, $B \in R$, $C \in R$. В четвертой итерации в R добавить нечего, итерации завершаются, $R = \{B, A, S\}$.

Выполняя шаг 2, получим непустые правила грамматики G':

$S \to AB$
$A \to a \mid BB$
$B \to b$

Мы удалили единственное пустое правило $B \to \lambda$.

Выполняя шаг 3, исключаем из правил вырождающиеся нетерминалы. Рассмотрим правило $S \to AB$. Символы A и B входят в R. Если бы символ A порождал пустую цепочку, то S порождал бы B. Точно так же, если бы символ B порождал пустую цепочку, то S порождал бы A. Если бы и A и B порождали пустую цепочку, то S также порождал бы пустую цепочку. Так как S — целевой символ G, $\lambda \in L(G)$.

Рассмотрим правило $A \to BB$. Символ B входит в R. Он может порождать пустую и непустую цепочку, поэтому символ A может порождать цепочки BB и B. Если бы B порождал пустую цепочку в обоих случаях, символ A порождал бы пустую цепочку, но эта цепочка уже учтена в правиле для символа S. Поэтому при исключении комбинаций вырождающихся символов из первоначальных цепочек получающиеся пустые цепочки не учитываются, и пустых правил не образуют.

Выполнив шаг 3, получим грамматику G' без пустых правил:

$S \to AB \mid B \mid A$
$A \to a \mid BB \mid B$
$B \to b$

Эта грамматика порождает те же непустые цепочки терминалов, что и исходная грамматика. Выполняя шаг 4, окончательно получим:

$S' \to S \mid \lambda$
$S \to AB \mid B \mid A$
$A \to a \mid BB \mid B$
$B \to b$

Теперь грамматика G' также порождает пустую цепочку. ●

Заметим, что если в грамматике есть правила вида $A \to \alpha B \beta$, когда одновременно $\alpha \Rightarrow^* \lambda$ и $\beta \Rightarrow^* \lambda$, это преобразование ведет к появлению правил вида $A \to B$ (цепных), поэтому устранение пустых правил предшествует устранению цепных правил.

1.7.5. Устранение цепных правил

Пусть есть грамматика G. *Циклом* или *циклическим выводом* называется вывод $A \Rightarrow^* A$, $A \in \mathrm{N}$. Очевидно, что такой вывод бесполезен. Циклы возможны в случае, если в грамматике присутствуют *цепные правила* (*unit rules*) вида $A \to B$, $A, B \in \mathrm{N}$.

Алгоритм устранения цепных правил для каждого нетерминала A строит множество R цепных нетерминалов. Тогда, если $B \in R$, $B \neq A$, и есть не цепные правила $B \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$, правило $A \to B$ заменяется правилами $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$.

Алгоритм 1.4. Устранение цепных правил.

Шаг 1. Удаление цепных правил.

Записываем в грамматику G' все не цепные правила грамматики G.

Шаг 2. Обозначение нетерминалов.

Пусть n равно количеству нетерминалов грамматики G. Обозначим нетерминалы символами $A_1, A_2, \ldots, A_n$, $i = 1$.

Шаг 3. Построение множества цепных нетерминалов R для A_i.

Базис. $R = \{A_i\}$.

Индукция. Если $(B \to C) \in \mathrm{P}$, $B, C \in \mathrm{N}$, и $B \in R$, то $C \in R$.

По определению базиса, включаем в R нетерминал A_i. По определению индукции, если есть правило $B \to C$, и $B \in R$, то включаем C в R.

Шаг 4. Замена цепных правил.

Просматриваем все правила грамматики G. Если есть не цепное правило $B \to \alpha$, и $B \in R$, и $B \neq A_i$, то запишем в грамматику G' правило $A_i \to \alpha$.

Если $i = n$, то перейти к шагу 5, иначе $i = i+1$, перейти к шагу 3.

Шаг 5. Элементы грамматики G'.

$\Sigma' = \Sigma$, $\mathrm{N}' = \mathrm{N}$, $\mathrm{S}' = \mathrm{S}$. ●

Пример 1.4.1. Устранение цепных правил.

Пусть грамматика G состоит из правил:

$S \to AB \mid B \mid A$

$A \to a \mid BB \mid B$

$B \to b$

На шаге 3 алгоритм сначала включает в R нетерминал A_i, а затем выполняет итерации, просматривая все правила грамматики до тех пор, пока в R добавляются новые символы.

Для нетерминала S в первой итерации включаем в R символы B и A. Во второй итерации в R включить больше нечего, $R = \{S, B, A\}$.

Для нетерминала A в первой итерации включаем в R символ B. Во второй итерации в R включить больше нечего, $R = \{A, B\}$.

Для нетерминала B в первой итерации в R включить нечего, $R = \{B\}$.

На шаге 4 добавим в грамматику G' правила:

- для нетерминала S: $S \to b$, $S \to a$, $S \to BB$;
- для нетерминала A: $A \to BB$, $A \to b$.

Получим эквивалентную грамматику без цепных правил:

$S \to AB \mid b \mid a \mid BB$

$A \to a \mid BB \mid b$

$B \to b$ ●

1.7.6. Устранение левой рекурсии

Нетерминальный символ A называется *рекурсивным*, если возможен вывод $A \Rightarrow^* \alpha A \beta$. Если $\alpha = \lambda$, возникает *левая рекурсия*, если $\beta = \lambda$, возникает *правая рекурсия*; здесь $A \in N$, $\alpha, \beta \in (\Sigma \cup N)^*$.

Грамматика называется *леворекурсивной*, если в ней возможен вывод с левой рекурсией, и *праворекурсивной*, если в ней возможен вывод с правой рекурсией. Грамматика может быть леворекурсивной, праворекурсивной, или одновременно и той и другой.

Некоторые алгоритмы левостороннего разбора языков не допускают применения леворекурсивных грамматик, поэтому возникает необходимость исключения левой рекурсии из выводов грамматики. Полностью исключить рекурсию из выводов нельзя — можно только избавиться от левой или правой рекурсии. Доказано, что любую грамматику можно преобразовать к нелеворекурсивному или неправорекурсивному виду.

Рекурсия может быть явной, например, $A \to A\alpha$, или неявной, когда она проявляется через несколько правил, например, $A \to B\beta$, $B \to A\alpha$.

Алгоритм устранения левой рекурсии преобразует правила исходной леворекурсивной грамматики так, чтобы в правиле вида $A \to \alpha$ цепочка α начиналась либо с терминала, либо с нетерминала B такого, который определяется в грамматике после нетерминала A.

Для этой цели все нетерминалы грамматики обозначаются символами A_i, $1 \le i \le n$, n — количество нетерминалов. Тогда цепочка α должна начинаться либо с терминала, либо с нетерминала A_j такого, что $j > i$.

Алгоритм 1.5. Устранение левой рекурсии.

Шаг 1. Обозначение нетерминалов $A_1, A_2, \ldots, A_n$, $A_1 \equiv S$; $i = 1$.

Шаг 2. Устранение явной рекурсии.

Если правила для нетерминала A_i не содержат левой рекурсии, пропускаем этот шаг. Иначе записываем правила для A_i в виде

$A_i \to A_i\alpha_1 \mid A_i\alpha_2 \mid \ldots \mid A_i\alpha_p \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_q$,

Здесь ни одна из цепочек βj не начинается с A_k такого, что $k \le i$.

Вместо этих правил в записываем две группы правил вида:

$A_i \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_q \mid \beta_1 A_i' \mid \beta_2 A_i' \mid \ldots \mid \beta_q A_i'$

$A_i' \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_p \mid \alpha_1 A_i' \mid \alpha_2 A_i' \mid \ldots \mid \alpha_p A_i'$

Нетерминал A_i' включаем в N.

Шаг 3. Завершение. Если $i = n$, перейти к шагу 6, иначе $i = i+1$, $j = 1$.

Шаг 4. Преобразование неявной рекурсии в явную.

Для символа A_j заменяем правила вида $A_i \to A_j\alpha$ на правила вида

$A_i \to \beta_1\alpha \mid \beta_2\alpha \mid \ldots \mid \beta_k\alpha$,

где $A_j \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_k$ — все правила для A_j.

Шаг 5. Если $j = i-1$, перейти к шагу 2, иначе $j = j+1$, перейти к шагу 4.

Шаг 6. Обратная замена обозначений нетерминалов. ●

Пример 1.5.1. Устранение левой рекурсии.

Пусть грамматика G состоит из правил:

$A \rightarrow Bb \mid a$
$B \rightarrow Aa \mid Bb \mid b$

Целевой символ A получает индекс 1, символ B — индекс 2.

Выполняем следующую последовательность шагов алгоритма.

Шаг 1. $i = 1$.

Шаг 2. Правила для A не содержат явной рекурсии.

Шаг 3. $i = 2, j = 1$.

Шаг 4. Правило $B \rightarrow Aa$ заменяем правилами $B \rightarrow Bba$ и $B \rightarrow aa$.

Получим грамматику:

$A \rightarrow Bb \mid a$
$B \rightarrow Bba \mid aa \mid Bb \mid b$

Шаг 2. Устраняем явную рекурсию нетерминала B.

Получим грамматику:

$A \rightarrow Bb \mid a$
$B \rightarrow aa \mid b \mid aaB' \mid bB'$
$B' \rightarrow ba \mid b \mid baB' \mid bB'$

●

Пример 1.5.2. Устранение левой рекурсии.

Нам интересно также устранить левую рекурсию в грамматике G_1, описывающей язык выражений со скобками:

$E \rightarrow E+T \mid T$
$T \rightarrow T*P \mid P$
$P \rightarrow a \mid (E)$

Неявной рекурсии здесь нет, поэтому устраняем явную рекурсию в соответствии с шагом 2 алгоритма.

Для группы правил $E \rightarrow E+T \mid T$ получим $E \rightarrow T \mid TE'$, $E' \rightarrow +T \mid +TE'$.

Для группы правил $T \rightarrow T*P \mid P$ получим $T \rightarrow P \mid PT'$, $T' \rightarrow *P \mid *PT'$.

Заменяя E' символом R, T' символом F, получим следующую грамматику для выражений со скобками без левой рекурсии:

$E \rightarrow T \mid TR$
$R \rightarrow +T \mid +TR$
$T \rightarrow P \mid PF$
$F \rightarrow *P \mid *PF$
$P \rightarrow a \mid (E)$ (1.9)

●

1.7.7. Левая факторизация

Это преобразование устраняет одинаковые префиксы в правых частях правил для некоторого нетерминального символа. Для этой цели множитель α (*factor*) выносится за скобки: $\alpha\beta_1 + \alpha\beta_2 = \alpha(\beta_1 + \beta_2)$.

Алгоритм 1.6. Левая факторизация

Шаг 1. Поиск и устранение префикса правил нетерминала A.

Пусть правила для нетерминала $A \in N$ имеют вид:

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_m \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$

где ни одна из цепочек γ_j не начинается с цепочки α.

Заменим правила для A на следующие две группы правил:

$A \to \alpha A' \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$

$A' \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$

Шаг 2. Поиск новых префиксов.

Повторим шаг 1, если в грамматике есть одинаковые префиксы. ●

Пример 1.6. Левая факторизация.

Нам интересна левая факторизация грамматики (1.9):

$E \to T \mid TR$

$R \to +T \mid +TR$

$T \to P \mid PF$

$F \to *P \mid *PF$

$P \to a \mid (E)$

Следуя алгоритму, из правил $E \to T \mid TR$ мы должны получить:

$E \to TA$,

$A \to R \mid \lambda$

Удаляя бесполезный цепной символ A, окончательно получим:

$E \to TR$

$R \to \lambda$

Все другие правила имеют такую же структуру, и их факторизация приводит к аналогичному результату, поэтому опустим подробности.

Результирующая грамматика для выражений со скобками, без левой рекурсии и одинаковых префиксов, обозначается нами как G_3:

$E \to TR$

$R \to +TR$

$R \to \lambda$

$T \to PF$

$F \to *PF$

$F \to \lambda$

$P \to a \mid (E)$ G_3

●

1.7.8. Грамматики в нормальной форме Хомского

КС-грамматика $G(\Sigma, N, P, S)$ называется *грамматикой в нормальной форме Хомского* (CNF), если правила грамматики имеют вид:

1) $A \to BC$, $A, B, C \in N$;

2) $A \to a$, $A \in N$, $a \in \Sigma$;

3) $S \to \lambda$, если $\lambda \in L(G)$ и S не встречается в правых частях правил.

Такую грамматику называют также бинарной нормальной формой (БНФ, дерево вывода в этой грамматике является бинарным). В БНФ можно преобразовать любую приведенную КС-грамматику.

Приведенная грамматика содержит правила только двух видов: $A \to a$ или $A \to \gamma$, $\gamma \in (\Sigma \cup N)^+$, $|\gamma| \geq 2$. Если заменить каждое вхождение терминала a в цепочку γ новым нетерминалом B, правила вида $A \to \gamma$ превратятся в правила вида $A \to B_1 B_2 \ldots B_k$, $k \geq 2$. При $k = 2$ правило становится допустимым, а при $k \geq 2$ заменим цепочки $B_1 B_2 \ldots B_k$ парами нетерминалов при помощи рекурсивной процедуры:

Базис. Если $k = 2$, правило $A \to B_1 B_2$ допустимо.

Индукция. Если $k \geq 2$, заменим $A \to B_1 B_2 \ldots B_k$ на $A \to B_1 C_1$ и $C_1 \to B_2 \ldots B_k$, и если $|B_2 \ldots B_k| \geq 2$, применим индукцию к правилу $C_1 \to B_2 \ldots B_k$.

Алгоритм 1.7. Приведение к нормальной форме Хомского.

Шаг 1. $N' = N$, $\Sigma' = \Sigma$, $S' = S$.

Шаг 2. В правилах длиной более одного символа каждый терминал a заменим новым нетерминалом B, $N' = N' \cup \{B\}$, $P' = P' \cup \{B \to a\}$.

Шаг 3. Правила вида $A \to a$, $A \to BC$, или $S \to \lambda$ запишем в P' как есть.

Шаг 4. Вместо правила вида $A \to B_1 B_2 \ldots B_k$, $k \geq 3$, в P' добавим правила

$A \to B_1 C_1$, $C_1 \to B_2 C_2$, …, $C_{k-3} \to B_{k-2} C_{k-2}$, $C_{k-2} \to B_{k-1} B_k$,

в N' добавим новые нетерминалы $C_1, C_2 \ldots C_{k-2}$. ●

Пример 1.7. Приведение к нормальной форме Хомского

Приведем к нормальной форме Хомского грамматику G_2:

$S \to S{+}S \mid S{*}S \mid a \mid (S)$

Выполняя шаг 2, запишем в P' правило $S \to a$.

Выполняя шаг 3, вместо правила $S \to S{+}S$ запишем правило $S \to SAS$, вместо правила $S \to S{*}S$ запишем $S \to SBS$, вместо правила $S \to (S)$ запишем $S \to CSD$, в P' запишем правила $A \to$ "+", $B \to$ "*", $C \to$ "(", $D \to$ ")".

Выполняя шаг 4, в P' запишем правила $S \to SE$, $E \to AS$, $S \to SF$, $F \to BS$, $S \to CG$, $G \to SD$.

Получим грамматику в нормальной форме Хомского:

$S \to SE \mid SF \mid CG \mid a$

$E \to AS$, $F \to BS$, $G \to SD$

$A \to$ "+", $B \to$ "*", $C \to$ "(", $D \to$ ")" ●

1.7.9. Грамматики в нормальной форме Грейбах

КС-грамматика $G(\Sigma, N, P, S)$ называется грамматикой в нормальной форме Грейбах (*Greibach*), если она содержит только правила вида:

1) $A \to a\gamma$, где $a \in \Sigma$ и $\gamma \in N^*$, (иногда при этом $|\gamma| \le 2$);

2) $S \to \lambda$, если $\lambda \in L(G)$ и S не встречается в правых частях правил.

Доказано, что любая КС-грамматика может быть приведена к нормальной форме Грейбах. Грамматика в нормальной форме Грейбах не имеет левой рекурсии, поэтому ее можно использовать при левостороннем выводе. Кроме того, вывод цепочки длиной n в этой грамматике имеет ровно n шагов.

Пример приведения грамматики к нормальной форме Грейбах

Формальный алгоритм приведения произвольной КС-грамматики к нормальной форме Грейбах достаточно сложный и здесь не приводится (см., например, [2][3]). Однако в качестве примера рассмотрим неформальное приведение к нормальной форме Грейбах грамматики (1.9):

$E \to {}^{(1)} T \mid {}^{(2)} TR$
$R \to (3) +T \mid {}^{(4)} +TR$
$T \to {}^{(5)} P \mid {}^{(6)} PF$
$F \to {}^{(7)} *P \mid {}^{(8)} *PF$
$P \to {}^{(9)} a \mid {}^{(10)} (E)$

Правила 3, 4, 7, 8 и 9 уже находятся в необходимой форме.

Рассмотрим правило 10. В нем последний терминал должен быть заменен нетерминалом, поэтому для нетерминала P запишем правила:

$P \to a \mid (EA$
$A \to)$

Теперь рассмотрим правила 5 и 6. Они начинаются с нетерминала P, для которого определены правила $P \to a \mid (EA$. Подставляя правые части этих правил вместо P, получим новые правила для нетерминала T:

$T \to a \mid (EA \mid aF \mid (EAF$

Теперь правые части этих правил подставим в правила 1 и 2:

$E \to a \mid (EA \mid aF \mid (EAF \mid aR \mid (EAR \mid aFR \mid (EAFR$

Результирующая грамматика в нормальной форме Грейбах:

$E \to a \mid aF \mid aR \mid aFR \mid (EA \mid (EAF \mid (EAR \mid (EAFR$
$R \to +T \mid +TR$
$T \to a \mid aF \mid (EA \mid (EAF$
$F \to *P \mid *PF$
$P \to a \mid (EA$
$A \to)$

●

1.8. Трансляция языков

1.8.1. Лексика, синтаксис и семантика языков

Формальные языки никакой смысловой нагрузки не несут, они суть математические объекты для исследования свойств языков. Приложение теории формальных языков к практике выявляет, что интересующие нас языки могут быть иерархически структурированы по свойствам их языковых конструкций. Очевидно, что структура слова проще структуры предложения, а структура предложения проще структуры абзаца текста. Попробуйте написать слово, потом пару предложений, а затем сформировать абзац из предложений так, чтобы ваша мысль была понятна.

Различие свойств языковых конструкций приводит к выделению в языках уровней лексики, синтаксиса и семантики (рисунок 1.9).

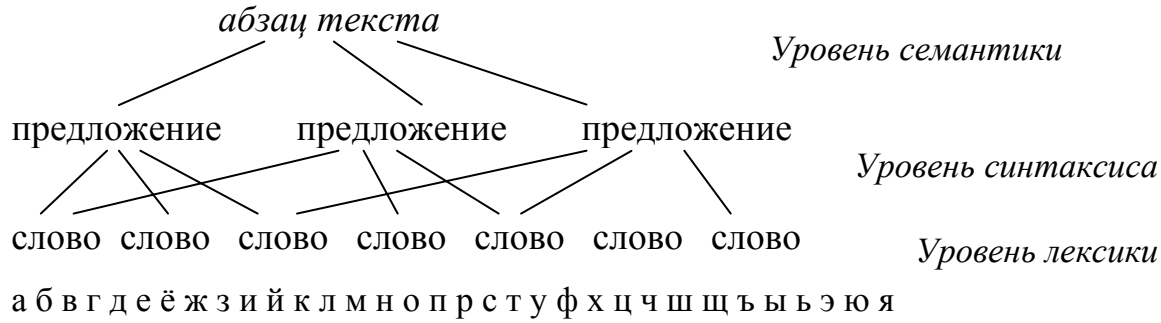

Рисунок 1.9. Иерархическая структура текста

Лексика — это словарный состав языка. Он формируется из алфавита соединением букв в определенной последовательности. Вследствие несложной структуры, слова описываются языками самого простого типа, а именно, регулярными языками. Принято называть отдельное слово лексической единицей, или лексемой. Поиск и выделение слов в тексте в теории языков программирования называют *лексическим анализом*.

В языках программирования лексемами являются идентификаторы, ключевые слова, литералы, знаки и символы операций и пунктуаторов, а также комментарии. Для описания этих простых структур используются *регулярные выражения* и, реже, регулярные грамматики.

Синтаксис описывает структуру предложений естественных языков или операторов языков программирования. Порядок слов предложения или оператора обычно таков, что мощности регулярных языков недостаточно для его описания, но линейные и бесконтекстные языки хорошо с этим справляются. Поэтому синтаксис языков программирования описывается при помощи контекстно-свободных грамматик, которые в этом случае называют *синтаксическими грамматиками*.

Проверка соответствия порядка следования лексем синтаксическим грамматикам называется *синтаксическим анализом*.

Структура бесконтекстных грамматик не задает никакого порядка предложений или операторов, она регламентирует лишь порядок слов в предложении. Иногда это приводит к бессмысленности текста в целом.

Например, следующие предложения, правильные сами по себе, составляют осмысленный текст, только если порядок предложений 1-2-3.

1) Возьмите яйцо. 2) Разбейте его и вылейте на сковородку. 3) Жарьте, пока не появится румяная корочка.

Эти предложения приобретают правильный смысл, если известен или понятен предыдущий контекст. Поэтому осмысленные тексты не могут быть описаны контекстно-свободными языками, как минимум, требуется контекстно-зависимый язык. В программировании ситуация с контекстом не лучше. Например, использованию переменной должно предшествовать ее объявление.

Семантика — это смысл правильно составленных предложений или операторов, с одной стороны, и смысл текста в целом, с другой стороны.

Семантика предложения (семантика синтаксиса) выделяет объекты и действия в предложении и устанавливает их правильное сочетание. Так, в предложении 1 объектом является яйцо, а действием взять, и по смыслу яйцо действительно можно взять.

Семантика текста устанавливает связь между предложениями через их объекты. Так, предложения 1 и 2 связаны объектом яйцо, и семантика действий с яйцом «взять-разбить-вылить» является допустимой.

В естественных языках семантика никак не задается, а определяется жизненным опытом человека. В языках программирования семантика описывается отдельно от синтаксиса и лексики в виде семантических соглашений, норм и правил, которые должны быть соблюдены, чтобы программа приобрела допустимый (не обязательно правильный) смысл.

Рассмотрим простой пример текста на языке С.

```
1   int i, j;
2   i = 2.5 / 1.1;
3   j = i + 1;
```

На уровне оператора проверяется соответствие типов. В строке 2 переменной типа int может быть присвоено только целое значение, поэтому в программу в этом месте неявно добавляется операция приведения вещественного результата к целочисленному значению. В строчках 2 и 3 проверяется совместимость типов, участвующих в операциях.

На уровне текста в целом строка 1 должна предшествовать строчкам 2 и 3, а строка 2 должна предшествовать строке 3.

Эти правила проверяет или выполняет *семантический анализатор*.

1.8.2. Распознаватели языков

Распознаватель (*recognizer*) — это алгоритм, проверяющий принадлежность некоторой цепочки языку (и тем самым задающий этот язык).

Условно распознаватель состоит из считывающей головки, устройства управления УУ, рабочей памяти и входной ленты, на которой находится цепочка символов $a_1a_2\ldots a_n$ входного языка (рисунок 1.10).

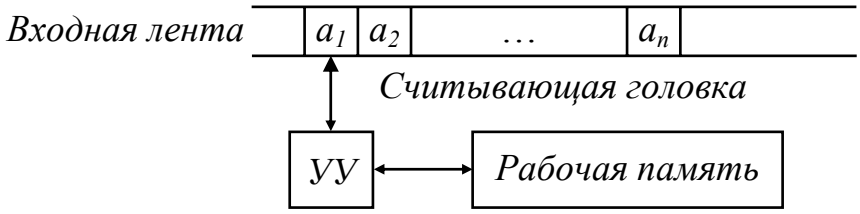

Рисунок 1.10. Схема распознавателя

Работа распознавателя заключается в выполнении шагов или тактов, которые меняют текущую *конфигурацию* распознавателя. Конфигурация задается как n-ка (p_1, p_2, …, p_n), где p_i обозначает некоторый параметр, такой как состояние устройства управления, положение считывающей головки, содержимое рабочей памяти (если такая память есть).

Переход из одной конфигурации в другую выполняется на основе правил, записываемых как функция $\delta(p_1, p_2, \ldots, p_n)$, которая определяет новые значения параметров.

Одно из состояний устройства управления является начальным, одно или несколько состояний являются финальными или допускающими. Так же называется конфигурация, если начальное или допускающее состояние входит в нее.

Распознаватель начинает работу в начальной конфигурации. Выполняя такты, он переходит от одной конфигурации к другой, пока либо не перейдет в одну из допускающих конфигураций, либо не зайдет в тупик. В зависимости от причины остановки распознаватель либо допускает цепочку, как принадлежащую языку, либо нет.

Если из одной конфигурации распознаватель может перейти только в одну другую конфигурацию, он называется детерминированным, в противном случае распознаватель недетерминированный.

По направлению движения по ленте различают распознаватели:

- односторонние (лента перемещается в одну сторону);
- двусторонние (лента может перемещаться в обе стороны).

По наличию рабочей памяти различают распознаватели:

- без рабочей памяти;
- с рабочей памятью ограниченного объема;
- имеющие неограниченную рабочую память.

Регулярные языки (типа 3) распознаются простым односторонним распознавателем без рабочей памяти, называемым конечным автоматом.

Для распознавания контекстно-свободных языков (типа 2) требуется односторонний распознаватель с небольшим объемом рабочей памяти, организованной как стек. Такой распознаватель называется автоматом с магазинной памятью (МП-автоматом).

Для распознавания языков типа 0 требуется двусторонний распознаватель с неограниченной памятью, машина Тьюринга общего вида. Для распознавания языков типа 1 требуется линейно-ограниченный автомат. Доказано, что те и другие распознаются МП-автоматом с двумя магазинами. Эти распознаватели не используются в языках программирования, потому что их реализация затруднена, а время распознавания велико.

1.8.3. Схема трансляции

Трансляция — это процесс перевода текста на одном языке в текст на другом языке. Язык исходного текста называют *входным*, а язык результирующего текста называют *выходным* или *целевым*.

Транслятор — это программа, выполняющая трансляцию текста, и написанная обычно в соответствии с теорией языков программирования. Входным языком транслятора может быть, например, некоторый язык программирования, а выходным — машинный язык.

Трансляция состоит из двух важных этапов — *анализ* и *синтез*.

Во время анализа входной текст проверяется на соответствие лексическим, синтаксическим и семантическим правилам входного языка. Если такое соответствие установлено, выполняется этап синтеза, который генерирует текст на целевом языке, семантически эквивалентный тексту на входном языке. Этот процесс называется также *переводом*.

Для анализа в составе транслятора есть соответственно лексический, синтаксический и семантический анализаторы (рисунок 1.11).

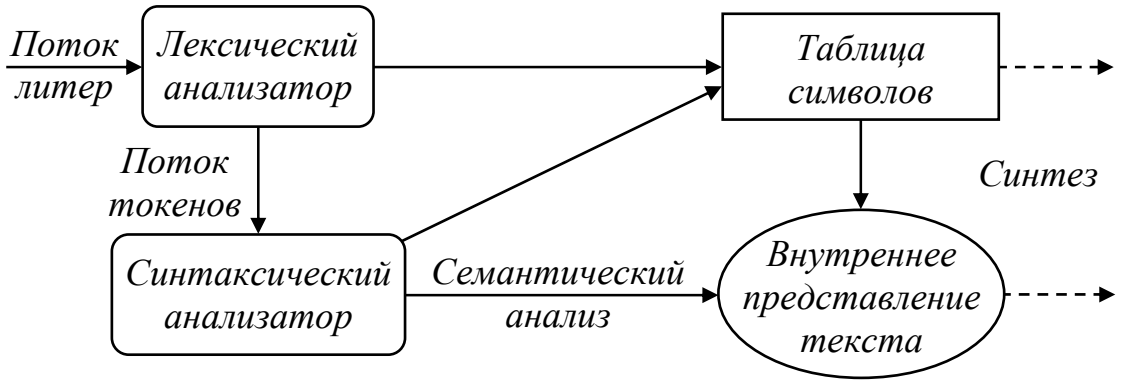

Рисунок 1.11. Анализ входного текста

Лексический анализ разделяет входной текст на его части (лексемы), с тем, чтобы далее оперировать не литерами, а словами. В трансляторе лексемы преобразуются в структуры данных — токены. Обнаружение и выделение лексем выполняют конечные автоматы.

Синтаксический анализ удостоверяет, что слова организованы в правильные предложения (операторы языка программирования). Это более сложная задача, и для ее решения требуется автомат помощнее. Таким автоматом является МП-автомат (автомат с магазинной памятью).

Семантический анализ проверяет, выполняются или нет контекстные условия, в которых предложения приобретают нужный смысл, так как синтаксис описывается бесконтекстными языками. Для семантического анализа никакого специального автомата нет, поэтому он не выделяется как отдельный анализатор. Вместо этого к элементам синтаксической грамматики приписываются семантические правила и действия, которые выполняют *синтаксически управляемый перевод*.

Есть несколько схем такого перевода, главенствующим из которых является *атрибутный перевод*, который к элементам синтаксической структуры приписывает *атрибуты* и функции их вычисления.

В результате анализа формируются две структуры данных: таблица символов и внутреннее представление текста. Таблица символов — это база данных, собирающая всю возможную информацию об объектах входного текста (например, переменных), необходимую для проверки семантики и генерации выходного текста. Внутреннее представление — это семантически эквивалентная промежуточная форма входного текста, имеющая более удобную структуру для генерации кода на выходе.

Таблица символов и промежуточная форма являются начальной стадией синтеза, называемой *генерацией промежуточного кода*. Построение этой формы преследует две цели: сформировать текст, не зависящий ни от входного, ни от выходного языка, и по возможности выполнить машинно-независимую оптимизацию.

В заключительной стадии синтеза *генератор целевого кода*, который в большинстве случаев является машинно-зависимым, формирует слова и предложения целевого языка (рисунок 1.12).

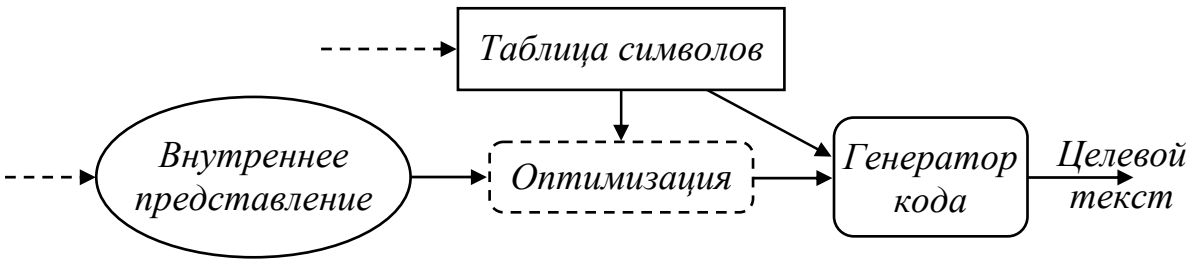

Рисунок 1.12. Синтез выходного текста

1.9. Вопросы и упражнения

Вопросы для самопроверки

1. Дайте определение символу, алфавиту, цепочке, языку.
2. Опишите операции с цепочками, приведите примеры.
3. Опишите способы задания языков.
4. Дайте определение формальной грамматике и ее элементам.
5. Поясните использование метасимволов в РБНФ.
6. Как классифицируются формальные грамматики и языки?
7. Что называется выводом, как он выполняется?
8. Какой вывод называется непосредственным, нетривиальным?
9. Какой вывод называется сентенциальным?
10. В чем различие между лево- и правосторонним выводами?
11. Дайте определение дереву вывода.
12. Как строится дерево вывода сверху вниз, снизу вверх?
13. Какая грамматика называется неоднозначной?
14. Приведите примеры неоднозначных грамматик.
15. Что называется приведенной грамматикой?
16. Какой символ называется бесплодным, недостижимым?
17. Что называется цепным правилом, пустым правилом?
18. Какой символ называется (лево-, право-) рекурсивным?
19. Что называется грамматикой в нормальной форме Хомского?
20. Что называется грамматикой в нормальной форме Грейбах?
21. В чем заключается левая факторизация?
22. Опишите распознаватели языков.
23. Дайте определение лексике, синтаксису и семантике языка.
24. Опишите процесс трансляции и назовите его составные части.

Упражнения

1. Определите класс грамматики и определяемый ею язык:
а) $S \to 0 \mid 1 \mid S0 \mid S1$
б) $S \to 0 \mid S0S1$
в) $S \to 0 \mid 01S$
г) $S \to 0 \mid 1 \mid 00S \mid 1SS$
2. Постройте левосторонний вывод и дерево вывода для цепочки:
а) "1100" в грамматике п.1а;
б) "0001000011" в грамматике п.1б;
в) "010101010" в грамматике п.1в;
г) "0000111" в грамматике п.1г;
3. Постройте вывод и дерево вывода в грамматике
$S \to AB \mid DC,\ A \to aA \mid \lambda,\ B \to bBc \mid \lambda,\ C \to cC \mid \lambda,\ D \to aDb \mid \lambda$
Цепочка "*aabbc*". Попробуйте построить два разных дерева.

4. Разработайте грамматику для:
а) записи отрицательных целых чисел;
б) записи десятичных чисел со знаком в форматах "0.", ".0", "0.0";
в) записи строк вида 01{01};
г) языка $L = \{ ab^n c \mid n > 0 \}$.
5. Устраните цепные правила и λ-правила из грамматики
$S \rightarrow A \mid B$, $A \rightarrow a$, $B \rightarrow a{=}AR$, $R \rightarrow +a \mid \lambda$
Приведите полученную грамматику к нормальной форме Хомского.
6. Устраните левую рекурсию в грамматике
$S \rightarrow AS \mid a$, $A \rightarrow SA \mid a$
7. Устраните одинаковые префиксы в грамматике
$S \rightarrow viA \mid viRA$, $A \rightarrow kts$, $R \rightarrow ci \mid ciR$

1.10. Литература

Трудно описать введение в теорию формальных языков иначе, чем это уже сделано в каждой книге по формальным языкам и трансляции, в той или иной мере здесь обязательно будет присутствовать плагиат. По той же причине затруднительно указать лучший источник информации, почти любая книга из перечня подойдет. Тем не менее, рекомендую читать Ахо [2] и Хопкрофт [6].

Текст главы сформирован в основном по источникам [2], [5] и [6], хотя были изучены все доступные источники.

1. N. Chomsky. On certain formal properties of grammars. Information and Control 2:2 (1959), pp. 137-167. (Хомский Н. О некоторых формальных свойствах грамматик. — Кибернетический сборник, вып. 5. — М.: ИЛ, 1962. — с. 279-311.).

2. А. Ахо, Дж. Ульман. Теория синтаксического анализа, перевода и компиляции. Том 1. Синтаксический анализ. Пер. с англ. М.: Мир, 1978.

3. Опалева Э.А., Самойленко В.П. Языки программирования и методы трансляции. — СПб.: БХВ-Петербург, 2005. — 480 с.: ил.

4. Пентус А.Е., Пентус М.Р. Теория формальных языков: Учебное пособие. — М.: Изд-во ЦПИ при механико-математическом ф-те МГУ, 2004. — 80 с.

5. Рейуорд-Смит В. Дж. Теория формальных языков. Вводный курс: Пер. с англ. — М.: Радио и связь, 1988. — 128 с.: ил.

6. Хопкрофт, Джон, Э., Мотвани, Раджив, Ульман, Джеффри, Д. Введение в теорию автоматов, языков и вычислений, 2-е изд. : Пер. с англ. — М.: Издательский дом «Вильямс», 2002. — 528 с.: ил.