Формальные языки и грамматики

Языки и цепочки символов. Способы задания языков

Цепочки символов. Операции над цепочками символов

Цепочкой символов (или строкой) называют произвольную упорядоченную конечную последовательность символов, записанных один за другим. Понятие *символа* (или буквы) является базовым в теории формальных языков и не нуждается в определении.

Далее цепочки символов будут обозначаться греческими буквами: α, β, γ.

Цепочка символов — это последовательность, в которую могут входить любые символы. Строка, которую вы сейчас читаете, является примером цепочки, символы в которой — строчные и заглавные русские буквы, знаки препинания и символ пробела. Но цепочка — это необязательно некоторая осмысленная последовательность символов. Последовательность «*аввв..аагрьь,..лл*» — тоже пример цепочки символов.

Цепочка символов — это упорядоченная последовательность символов. Это значит, что для цепочки символов имеют значение три фактора: состав входящих в цепочку символов, их количество, а также порядок символов в цепочке. Поэтому цепочки «*а*» и «*аа*», а также «*аб*» и «*ба*» — это различные цепочки символов.

Цепочки символов α и β *равны* (совпадают) $\alpha = \beta$, если они имеют один и тот же состав символов, одно и то же их количество и одинаковый порядок следования символов в цепочке.

Количество символов в цепочке называют *длиной цепочки*. Длина цепочки символов α обозначается как $|\alpha|$. Очевидно, что если $\alpha = \beta$, то и $|\alpha| = |\beta|$.

Основной операцией над цепочками символов является операция конкатенации (объединения или сложения) цепочек.

Конкатенация (сложение, объединение) двух цепочек символов — это дописывание второй цепочки в конец первой. Конкатенация цепочек α и β обозначается как $\alpha\beta$. Выполнить конкатенацию цепочек просто: например, если α = *аб*, а β = *вг*, то $\alpha\beta$ = *абвг*.

ВНИМАНИЕ

> Так как в цепочке важен порядок символов, то очевидно, что для операции конкатенации двух цепочек символов важно, в каком порядке записаны цепочки. Иными словами, конкатенация цепочек символов не обладает свойством коммутативности, то есть в общем случае $\exists\ \alpha$ и β, такие, что: $\alpha\beta \neq \beta\alpha$ (например, для α = ***аб*** и β = ***вг***: $\alpha\beta$ = ***абвг***, а $\beta\alpha$ = ***вгаб*** и $\alpha\beta \neq \beta\alpha$).

Также очевидно, что конкатенация обладает свойством ассоциативности, то есть $(\alpha\beta)\gamma = \alpha(\beta\gamma)$.

Любую цепочку символов можно представить как конкатенацию составляющих ее частей — разбить цепочку на несколько подцепочек. Такое разбиение можно выполнить несколькими способами произвольным образом. Например, цепочку γ = *абвг* можно представить в виде конкатенации цепочек α = *аб* и β = *вг* ($\gamma = \alpha\beta$), а можно — в виде конкатенации цепочек υ = *а* и ω = *бвг* ($\gamma = \upsilon\omega$). Чем длиннее исходная цепочка, тем больше вариантов разбиения ее на составляющие подцепочки.

Если некоторую цепочку символов разбить на составляющие ее подцепочки, а затем заменить одну из подцепочек на любую произвольную цепочку символов, то в результате получится новая цепочка символов. Такое действие называется *заменой* или *подстановкой* цепочки. Например, возьмем все ту же цепочку γ = *абвг*, разобьем ее на три подцепочки: α = *а,* ω = *б* и β = *вг* ($\gamma = \alpha\omega\beta$), и выполним подстановку цепочки υ = *аба* вместо подцепочки ω. Получим новую цепочку γ' = *аабавг* ($\gamma' = \alpha\upsilon\beta$). Любая подстановка выполняется с помощью разбиения исходной цепочки на подцепочки и операции конкатенации.

Можно выделить еще две операции над цепочками:

Обращение цепочки — это запись символов цепочки в обратном порядке. Обращение цепочки α обозначается как α^R. Если α = «*абвг*», то α^R = «*гвба*». Для операции обращения справедливо следующее равенство $\forall\ \alpha,\beta$: $(\alpha\beta)^R = \beta^R\alpha^R$.

Итерация (повторение) цепочки n раз, где $n \in \mathbf{N}$, $n > 0$ — это конкатенация цепочки самой с собой n раз. Итерация цепочки α n раз обозначается как α^n. Для операции повторения справедливы следующие равенства $\forall\ \alpha$: $\alpha^1 = \alpha$, $\alpha^2 = \alpha\alpha$, $\alpha^3 = \alpha\alpha\alpha$, … и т. д. Итерация цепочки символов определена и для $n = 0$ — в этом случае результатом итерации будет пустая цепочка символов.

Пустая цепочка символов — это цепочка, не содержащая ни одного символа. Пустая цепочка здесь везде будет обозначаться греческой буквой λ (в литературе ее иногда обозначают латинской буквой e или греческой ε).

Для пустой цепочки справедливы следующие равенства:

1. $|\lambda| = 0$
2. $\forall\alpha$: $\lambda\alpha = \alpha\lambda = \alpha$
3. $\lambda^R = \lambda$
4. $\forall n \geq 0$: $\lambda^n = \lambda$
5. $\forall\alpha$: $\alpha^0 = \lambda$

Понятие языка. Формальное определение языка

В общем случае язык — это заданный набор символов и правил, устанавливающих способы комбинации этих символов между собой для записи осмысленных текстов. Основой любого естественного или искусственного языка является алфавит, определяющий набор допустимых символов языка.

Алфавит — это счетное множество допустимых символов языка. Будем обозначать это множество символом **V**. Интересно, что согласно формальному определению, алфавит не обязательно должен быть конечным множеством, но реально все существующие языки строятся на основе конечных алфавитов.

Цепочка символов α является *цепочкой над алфавитом* **V**: $\alpha(\mathbf{V})$, если в нее входят только символы, принадлежащие множеству символов **V**. Для любого алфавита **V** пустая цепочка λ может как являться, так и не являться цепочкой $\lambda(\mathbf{V})$. Это условие оговаривается дополнительно.

Если **V** — некоторый алфавит, то:

$\mathbf{V}^+$ — множество всех цепочек над алфавитом **V** без λ;

$\mathbf{V}^*$ — множество всех цепочек над алфавитом **V**, включая λ.

Справедливо равенство: $\mathbf{V}^* = \mathbf{V}^+ \cup \{\lambda\}$.

Языком **L** над алфавитом **V**: L(**V**) называется некоторое счетное подмножество цепочек конечной длины из множества всех цепочек над алфавитом **V**. Из этого определения следует два вывода: во-первых, множество цепочек языка не обязано быть конечным; во-вторых, хотя каждая цепочка символов, входящая в язык, обязана иметь конечную длину, эта длина может быть сколь угодно большой и формально ничем не ограничена.

Все существующие языки подпадают под это определение. Большинство реальных естественных и искусственных языков содержат бесконечное множество цепочек. Также в большинстве языков длина цепочки ничем не ограничена (например, этот длинный текст — пример цепочки символов русского языка). Цепочку символов, принадлежащую заданному языку, часто называют *предложением* языка, а множество цепочек символов некоторого языка L(**V**) — множеством предложений этого языка.

Для любого языка L(**V**) справедливо: $L(\mathbf{V}) \subseteq \mathbf{V}^*$.

Язык L(**V**) *включает в себя* язык L'(**V**): $L'(\mathbf{V}) \subseteq L(\mathbf{V})$, если $\forall \alpha \in L(\mathbf{V})$: $\alpha \in L'(\mathbf{V})$. Множество цепочек языка L'(**V**) является подмножеством множества цепочек языка L(**V**) (или эти множества совпадают). Очевидно, что оба языка должны строится на основе одного и того же алфавита.

Два языка L(**V**) и L'(**V**) *совпадают* (эквивалентны): L'(**V**) = L(**V**), если $L'(\mathbf{V}) \subseteq L(\mathbf{V})$ и $L(\mathbf{V}) \subseteq L'(\mathbf{V})$; или, что то же самое: $\forall \alpha \in L'(\mathbf{V})$: $\alpha \in L(\mathbf{V})$ и $\forall \beta \in L(\mathbf{V})$: $\beta \in L'(\mathbf{V})$. Множества допустимых цепочек символов для эквивалентных языков равны.

Два языка L(**V**) и L'(**V**) *почти эквивалентны*: $L'(\mathbf{V}) \cong L(\mathbf{V})$, если $L'(\mathbf{V}) \cup \{\lambda\} = L(\mathbf{V}) \cup \{\lambda\}$. Множества допустимых цепочек символов почти эквивалентных языков могут различаться только на пустую цепочку символов.

Способы задания языков. Синтаксис и семантика языка

Итак, каждый язык — это множество цепочек символов над некоторым алфавитом. Но кроме алфавита язык предусматривает также правила построения допустимых цепочек, поскольку обычно далеко не все цепочки над заданным алфавитом принадлежат языку. Символы могут объединяться в слова или лексемы —

элементарные конструкции языка, на их основе строятся предложения — более сложные конструкции. И те, и другие в общем виде являются цепочками символов, но предусматривают некоторые правила построения. Таким образом, необходимо указать эти правила, или, строго говоря, задать язык.

В общем случае язык можно определить тремя способами:

1. перечислением всех допустимых цепочек языка;
2. указанием способа порождения цепочек языка (заданием грамматики языка);
3. определением метода распознавания цепочек языка.

Первый из методов является чисто формальным и на практике не применяется, так как большинство языков содержат бесконечное число допустимых цепочек и перечислить их просто невозможно. Трудно себе представить, чтобы появилась возможность перечислить, например, множество всех правильных текстов на русском языке или всех правильных программ на языке Pascal. Иногда, для чисто формальных языков, можно перечислить множество входящих в них цепочек, прибегнув к математическим определениям множеств. Однако этот подход уже стоит ближе ко второму способу.

Например, запись $L(\{0,1\}) = \{0^n1^n,\ n > 0\}$ задает язык над алфавитом $\mathbf{V} = \{0,1\}$, содержащий все последовательности с чередующимися символами 0 и 1, начинающиеся с 0 и заканчивающиеся 1. Видно, что пустая цепочка символов в этот язык не входит. Если изменить условие в этом определении с $n > 0$ на $n \geq 0$, то получим почти эквивалентный язык $L'(\{0,1\})$, содержащий пустую цепочку.

Второй способ предусматривает некоторое описание правил, с помощью которых строятся цепочки языка. Тогда любая цепочка, построенная с помощью этих правил из символов алфавита языка, будет принадлежать заданному языку. Например, с правилами построения цепочек символов русского языка вы долго и упорно знакомились в средней школе.

Третий способ предусматривает построение некоторого логического устройства (распознавателя) — автомата, который на входе получает цепочку символов, а на выходе выдает ответ: принадлежит или нет эта цепочка заданному языку. Например, читая этот текст, вы сейчас в некотором роде выступаете в роли распознавателя (надеюсь, что ответ на вопрос о принадлежности текста русскому языку будет положительным).

Говоря о любом языке, можно выделить его синтаксис и семантику. Кроме того, трансляторы имеют дело также с лексическими конструкциями (лексемами), которые задаются лексикой языка. Ниже даны определения для всех этих понятий.

Синтаксис языка — это набор правил, определяющий допустимые конструкции языка. Синтаксис определяет «форму языка» — задает набор цепочек символов, которые принадлежат языку. Чаще всего синтаксис языка можно задать в виде строгого набора правил, но полностью это утверждение справедливо только для чисто формальных языков. Даже для большинства языков программирования набор заданных синтаксических конструкций нуждается в дополнительных пояснениях, а синтаксис языков естественного общения вполне соответствует общепринятому мнению о том, что «исключения только подтверждают правило».

Например, любой окончивший среднюю школу может сказать, что строка «3 + 2» является арифметическим выражением, а «3 2 +» — не является. Правда, не каждый задумается при этом, что он оперирует синтаксисом алгебры.

Семантика языка — это раздел языка, определяющий значение предложений языка. Семантика определяет «содержание языка» — задает смысл для всех допустимых цепочек языка. Семантика для большинства языков определяется неформальными методами (отношения между знаками и тем, что они обозначают, изучаются семиотикой). Чисто формальные языки лишены какого-либо смысла. Возвращаясь к примеру, приведенному выше, и используя семантику алгебры, мы можем сказать, что строка «3 + 2» есть сумма чисел 3 и 2, а также то, что «3 + 2 = 5» — это истинное выражение. Однако изложить любому ученику синтаксис алгебры гораздо проще, чем ее семантику, хотя в случае алгебры семантику как раз можно определить формально.

Лексика — это совокупность слов (словарный запас) языка. Слово или лексическая единица (лексема) языка — это конструкция, которая состоит из элементов алфавита языка и не содержит в себе других конструкций. Иначе говоря, лексическая единица может содержать только элементарные символы и не может содержать других лексических единиц.

Лексическими единицами (лексемами) русского языка являются слова русского языка, а знаки препинания и пробелы представляют собой разделители, не образующие лексем. Лексическими единицами алгебры являются числа, знаки математических операций, обозначения функций и неизвестных величин. В языках программирования лексическими единицами являются ключевые слова, идентификаторы, константы, метки, знаки операций; в них также существуют и разделители (запятые, скобки, точки с запятой и т. д.).

Особенности языков программирования

Языки программирования занимают некоторое промежуточное положение между формальными и естественными языками. С формальными языками их объединяют строгие синтаксические правила, на основе которых строятся предложения языка. От языков естественного общения в языки программирования перешли лексические единицы, представляющие основные ключевые слова (чаще всего это слова английского языка, но существуют языки программирования, чьи ключевые слова заимствованы из русского и других языков). Кроме того, из алгебры языки программирования переняли основные обозначения математических операций, что также делает их более понятными человеку.

Для задания языка программирования необходимо решить три вопроса:

- определить множество допустимых символов языка;
- определить множество правильных программ языка;
- задать смысл для каждой правильной программы.

Только первые два вопроса полностью или частично удается решить с помощью теории формальных языков. Для решения остальных вопросов приходится прибегать к другим, неформальным методам.

Первый вопрос решается легко. Определяя алфавит языка, мы автоматически определяем множество допустимых символов. Для языков программирования алфавит — это чаще всего тот набор символов, которые можно ввести с клавиатуры. Основу его составляет младшая половина таблицы международной кодировки символов (таблицы ASCII), к которой добавляются символы национальных алфавитов.

Второй вопрос решается в теории формальных языков только частично. Для всех языков программирования существуют правила, определяющие синтаксис языка, но как уже было сказано, их недостаточно для того, чтобы строго определить все допустимые предложения языков программирования. Дополнительные ограничения накладываются семантикой языка. Эти ограничения оговариваются в неформальном виде для каждого отдельного языка программирования. К таким ограничениям можно отнести необходимость предварительного описания переменных и функций, необходимость соответствия типов переменных и констант в выражениях, формальных и фактических параметров в вызовах функций и другие.

Отсюда следует, что практически все языки программирования, строго говоря, не являются формальными языками. И именно поэтому во всех трансляторах кроме синтаксического разбора и анализа предложений языка дополнительно предусмотрен семантический анализ.

Третий вопрос в принципе не относится к теории формальных языков, поскольку, как уже было сказано, такие языки лишены какого-либо смысла. Для ответа на него нужно использовать другие подходы.

В следующей главе, посвященной основным принципам построения трансляторов и компиляторов, будут более подробно указаны отличия языков программирования от формальных языков, которые нужно принимать во внимание при создании трансляторов и компиляторов для языков программирования.

Грамматики и распознаватели

Формальное определение грамматики. Форма Бэкуса-Наура

Грамматика — это описание способа построений предложений некоторого языка. Иными словами, грамматика — это математическая система, определяющая язык.

Фактически, определив грамматику языка, мы указываем правила порождения цепочек символов, принадлежащих этому языку. Таким образом, грамматика — это генератор цепочек языка. Она относится ко второму способу определения языков — порождению цепочек символов.

Грамматику языка можно описать различными способами. Например, грамматика русского языка описывается довольно сложным набором правил, которые изучают в начальной школе. Для некоторых языков (в том числе для синтаксических конструкций языков программирования) можно использовать формальное описание грамматики, построенное на основе системы правил (или продукций).

Правило (или продукция) — это упорядоченная пара цепочек символов (α,β). В правилах важен порядок цепочек, поэтому их чаще записывают в виде $\alpha \rightarrow \beta$ (или $\alpha ::= \beta$). Такая запись читается как «α порождает β» или «α по определению есть β».

Грамматика языка программирования содержит правила двух типов: первые (определяющие синтаксические конструкции языка) довольно легко поддаются формальному описанию; вторые (определяющие семантические ограничения языка) обычно излагаются в неформальной форме. Поэтому любое описание (или стандарт) языка программирования обычно состоит из двух частей: вначале формально излагаются правила построения синтаксических конструкций, а потом на естественном языке дается описание семантических правил.

ВНИМАНИЕ

> Далее, говоря о грамматиках языков программирования, будем иметь в виду только правила построения синтаксических конструкций языка. Однако следует помнить, что грамматика любого языка программирования в общем случае не ограничивается только этими правилами.

Язык, заданный грамматикой **G**, обозначается как L(**G**).

Две грамматики **G** и **G'** называются эквивалентными, если они определяют один и тот же язык: L(**G**) = L(**G'**). Две грамматики **G** и **G'** называются почти эквивалентными, если заданные ими языки различаются не более чем на пустую цепочку символов: $L(\mathbf{G})\cup\{\lambda\} = L(\mathbf{G'})\cup\{\lambda\}$.

Формально грамматика **G** определяется как четверка G(**VT**,**VN**,**P**,S), где:

VT — множество терминальных символов или алфавит терминальных символов;

VN — множество нетерминальных символов или алфавит нетерминальных символов;

P — множество правил (продукций) грамматики, вида $\alpha\rightarrow\beta$, где $\alpha\in(\mathbf{VN}\cup\mathbf{VT})^{+}$, $\beta\in(\mathbf{VN}\cup\mathbf{VT})^{*}$;

S — целевой (начальный) символ грамматики $S\in\mathbf{VN}$.

Алфавиты терминальных и нетерминальных символов грамматики не пересекаются: $\mathbf{VN}\cap\mathbf{VT} = \varnothing$. Это значит, что каждый символ в грамматики может быть либо терминальным, либо нетерминальным, но не может быть терминальным и нетерминальным одновременно. Целевой символ грамматики — это всегда нетерминальный символ. Множество $\mathbf{V} = \mathbf{VN}\cup\mathbf{VT}$ называют полным алфавитом грамматики **G**.

Далее будут даны строгие формальные описания того, как связаны различные элементы грамматики и порождаемый ею язык. А пока предварительно опишем смысл множеств **VN** и **VT**. Множество терминальных символов **VT** содержит символы, которые входят в алфавит языка, порождаемого грамматикой. Как правило, символы из множества **VT** встречаются только в цепочках правых частей правил. Множество нетерминальных символов **VN** содержит символы, которые определяют слова, понятия, конструкции языка. Каждый символ этого множества может встречаться в цепочках как левой, так и правой частей правил грамматики, но он обязан хотя бы один раз быть в левой части хотя бы одного правила. Правила

грамматики строятся так, чтобы в левой части каждого правила был хотя бы один нетерминальный символ.

Во множестве правил грамматики может быть несколько правил, имеющих одинаковые левые части, вида: $\alpha \to \beta_1$, $\alpha \to \beta_2$, … $\alpha \to \beta_n$. Тогда эти правила объединяют вместе и записывают в виде: $\alpha \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$. Одной строке в такой записи соответствует сразу n правил.

Такую форму записи правил грамматики называют формой Бэкуса-Наура. Форма Бэкуса-Наура предусматривает, как правило, также, что нетерминальные символы берутся в угловые скобки: < >. Иногда знак → в правилах грамматики заменяют на знак **::=** (что характерно для старых монографий), но это всего лишь незначительные модификации формы записи, не влияющие на ее суть.

Ниже приведен пример грамматики, которая определяет язык целых десятичных чисел со знаком:

G({0,1,2,3,4,5,6,7,8,9,-
,+},{<число>,<чс>,<цифра>},**P**,<число>)

P:

<число> → <чс> | +<чс> | -<чс>

<чс> → <цифра> | <чс><цифра>

<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Рассмотрим составляющие элементы грамматики **G**:

- множество терминальных символов **VT** содержит двенадцать элементов: десять десятичных цифр и два знака;
- множество нетерминальных символов **VN** содержит три элемента: символы <число>, <чс> и <цифра>;
- множество правил содержит 15 правил, которые записаны в три строки (то есть имеется только три различных правых части правил);
- целевым символом грамматики является символ <число>.

Следует отметить, что символ <чс> — это бессмысленное сочетание букв русского языка, но это обычный нетерминальный символ грамматики, такой же, как и два других. Названия нетерминальных символов не обязаны быть осмысленными, это сделано просто для удобства понимания правил грамматики человеком. В принципе, в любой грамматике можно полностью изменить имена всех нетерминальных символов, не меняя при этом языка, заданного грамматикой — точно также, например, в программе на языке Pascal можно изменить имена идентификаторов, и при этом не изменится смысл программы.

Для терминальных символов это неверно. Набор терминальных символов всегда строго соответствует алфавиту языка, определяемого грамматикой.

Вот, например, та же самая грамматика для языка целых десятичных чисел со знаком, в которой нетерминальные символы обозначены большими латинскими буквами (далее это будет часто применяться в примерах):

G'({0,1,2,3,4,5,6,7,8,9,−,+},{S,T,F},**P**,S)

P:

S → T | +T | −T

T → F | TF

F → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Здесь изменилось только множество нетерминальных символов. Теперь **VN** = {S,T,F}. Язык, заданный грамматикой, не изменился — можно сказать, что грамматики **G** и **G'** эквивалентны.

Принцип рекурсии в правилах грамматики

Особенность рассмотренных выше формальных грамматик в том, что они позволяют определить бесконечное множество цепочек языка с помощью конечного набора правил (конечно, множество цепочек языка тоже может быть конечным, но даже для простых реальных языков это условие обычно не выполняется). Приведенная выше в примере грамматика для целых десятичных чисел со знаком определяет бесконечное множество целых чисел с помощью 15 правил.

В такой форме записи грамматики возможность пользоваться конечным набором правил достигается за счет рекурсивных правил. Рекурсия в правилах грамматики выражается в том, что один из нетерминальных символов определяется сам через себя. Рекурсия может быть непосредственной (явной) — тогда символ определяется сам через себя в одном правиле, либо косвенной (неявной) — тогда то же самое происходит через цепочку правил.

В рассмотренной выше грамматике **G** непосредственная рекурсия присутствует в правиле: <чс> → <чс><цифра>, а в эквивалентной ей грамматике **G'** — в правиле: T → TF.

Чтобы рекурсия не была бесконечной, для участвующего в ней нетерминального символа грамматики должны существовать также и другие правила, которые определяют его, минуя самого себя, и позволяют избежать бесконечного рекурсивного определения (в противном случае этот символ в грамматике был бы просто не нужен). Такими правилами являются <чс> → <цифра> — в грамматике **G** и T → F — в грамматике **G'**.

В теории формальных языков более ничего сказать о рекурсии нельзя. Но, чтобы полнее понять смысл рекурсии, можно прибегнуть к семантике языка — в рассмотренном выше примере это язык целых десятичных чисел со знаком. Рассмотрим его смысл.

Если попытаться дать определение тому, что же является числом, то начать можно с того, что любая цифра сама по себе есть число. Далее можно заметить, что любые две цифры — это тоже число, затем — три цифры и так далее. Если строить определение числа таким методом, то оно никогда не будет закончено (в математике разрядность числа ничем не ограничена). Однако можно заметить, что каждый раз, порождая новое число, мы просто дописываем цифру справа (поскольку привыкли писать слева направо) к уже написанному ряду цифр. А этот ряд цифр, начиная от одной цифры,

тоже в свою очередь является числом. Тогда определение для понятия «число» можно построить таким образом: «число — это любая цифра, либо другое число, к которому справа дописана любая цифра». Именно это и составляет основу правил грамматик **G** и **G'** и отражено в правилах `<чс>` → `<цифра>` | `<чс><цифра>` и `T` → `F` | `TF` (вторая строка правил). Другие правила в этих грамматиках позволяют добавить к числу знак (первая строка правил) и дают определение понятию «цифра» (третья строка правил). Они элементарны и не требуют пояснений.

Принцип рекурсии (иногда его называют «принцип итерации», что не меняет сути) — важное понятие в представлении о формальных грамматиках. Так или иначе, явно или неявно рекурсия всегда присутствует в грамматиках любых реальных языков программирования. Именно она позволяет строить бесконечное множество цепочек языка, и говорить об их порождении невозможно без понимания принципа рекурсии. Как правило, в грамматике реального языка программирования содержится не одно, а целое множество правил, построенных с помощью рекурсии.

Другие способы задания грамматик

Форма Бэкуса-Наура — удобный с формальной точки зрения, но не всегда доступный для понимания способ записи формальных грамматик. Рекурсивные определения хороши для формального анализа цепочек языка, но не удобны с точки зрения человека. Например, то, что правила `<чс>` → `<цифра>` | `<чс><цифра>` отражают возможность для построения числа дописывать справа любое число цифр, начиная от одной, неочевидно и требует дополнительного пояснения.

Но при создании языка программирования важно, чтобы его грамматику понимали не только те, кому предстоит создавать компиляторы для этого языка, но и пользователи языка — будущие разработчики программ. Поэтому существуют другие способы описания правил формальных грамматик, которые ориентированы на большую понятность для человека.

Далее рассмотрим два наиболее распространенных из этих способов: запись правил грамматик с использованием метасимволов и запись правил грамматик в графическом виде.

Запись правил грамматик с использованием метасимволов

Запись правил грамматик с использованием метасимволов предполагает, что в строке правила грамматики могут встречаться специальные символы — метасимволы, — которые имеют особый смысл и трактуются специальным образом. В качестве таких метасимволов чаще всего используются следующие символы: `( )` (круглые скобки), `[ ]` (квадратные скобки), `{ }` (фигурные скобки), **`" "`** (кавычки) и **`,`** (запятая).

Эти метасимволы имеют следующий смысл:

- круглые скобки означают, что из всех перечисленных внутри них цепочек символов в данном месте правила грамматики может стоять только одна цепочка;
- квадратные скобки означают, что указанная в них цепочка может встречаться, а может и не встречаться в данном месте правила грамматики (то есть может быть в нем один раз или ни одного раза);

- фигурные скобки означают, что указанная внутри них цепочка может не встречаться в данном месте правила грамматики ни одного раза, встречаться один раз или сколь угодно много раз;
- запятая служит для того, чтобы разделять цепочки символов внутри круглых скобок;
- кавычки используются в тех случаях, когда один из метасимволов нужно включить в цепочку обычным образом — то есть когда одна из скобок или запятая должны присутствовать в цепочке символов языка (если саму кавычку нужно включить в цепочку символов, то ее надо повторить дважды — этот принцип знаком разработчикам программ).

Вот как должны выглядеть правила рассмотренной выше грамматики **G**, если их записать с использованием метасимволов:

<число> → [(+,-)]<цифра>{<цифра>}

<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Вторая строка правил не нуждается в комментариях, а первое правило читается так: «число есть цепочка символов, которая может начинаться с символов + или −, должна содержать дальше одну цифру, за которой может следовать последовательность из любого количества цифр». В отличие от формы Бэкуса-Наура, в форме записи с помощью метасимволов, как видно, во-первых, убран из грамматики малопонятный нетерминальный символ <чс>, а во-вторых — удалось полностью исключить рекурсию. Грамматика в итоге стала более понятной.

Форма записи правил с использованием метасимволов — это удобный и понятный способ представления правил грамматик. Она во многих случаях позволяет полностью избавиться от рекурсии, заменив ее символом итерации { } (фигурные скобки). Как будет понятно из дальнейшего материала, эта форма наиболее употребительна для одного из типов грамматик — регулярных грамматик.

Кроме указанных выше метасимволов в целях удобства записи в описаниях грамматик иногда используют и другие метасимволы, при этом предварительно дается разъяснение их смысла. Принцип записи от этого не меняется. Также иногда дополняют смысл уже существующих метасимволов. Например, для метасимвола { } (фигурные скобки) существует удобная форма записи, позволяющая ограничить число повторений цепочки символов, заключенной внутри них: $\{\ \}^n$, где $n \in \mathbf{N}$ и $n > 0$. Такая запись означает, что цепочка символов, стоящая в фигурных скобках, может быть повторена от 0 до n раз (не более n раз). Это очень удобный метод наложения ограничений на длину цепочки.

Для рассмотренной выше грамматики **G**, таким способом можно, например, записать правила, если предположить, что она должна порождать целые десятичные числа, содержащие не более 15 цифр:

<число> → [(+,-)]<цифра>{<цифра>}^{14}

<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Для записи того же самого ограничения в форме Бэкуса-Наура или в форме с метасимволами потребовалось бы пятнадцать правил.

Запись правил грамматик в графическом виде

При записи правил в графическом виде вся грамматика представляется в форме набора специальным образом построенных диаграмм. Эта форма была предложена при описании грамматики языка Pascal, а затем она получила широкое распространение в литературе. Она доступна не для всех типов грамматик, а только для тех типов, где в левой части правил присутствует не более одного символа, но этого достаточно, чтобы ее можно было использовать для описания грамматик известных языков программирования.

В такой форме записи каждому нетерминальному символу грамматики соответствует диаграмма, построенная в виде направленного графа. Граф имеет следующие типы вершин:

- точка входа (на диаграмме никак не обозначена, из нее просто начинается входная дуга графа);
- нетерминальный символ (на диаграмме обозначается прямоугольником, в который вписано обозначение символа);
- цепочка терминальных символов (на диаграмме обозначается овалом, кругом или прямоугольником с закругленными краями, внутрь которого вписана цепочка);
- узловая точка (на диаграмме обозначается жирной точкой или закрашенным кружком);
- точка выхода (никак не обозначена, в нее просто входит выходная дуга графа).

Каждая диаграмма имеет только одну точку входа и одну точку выхода, но сколько угодно вершин других трех типов. Вершины соединяются между собой направленными дугами графа (линиями со стрелками). Из входной точки дуги могут только выходить, а во входную точку — только входить. В остальные вершины дуги могут как входить, так и выходить (в правильно построенной грамматике каждая вершина должна иметь как минимум один вход и как минимум один выход).

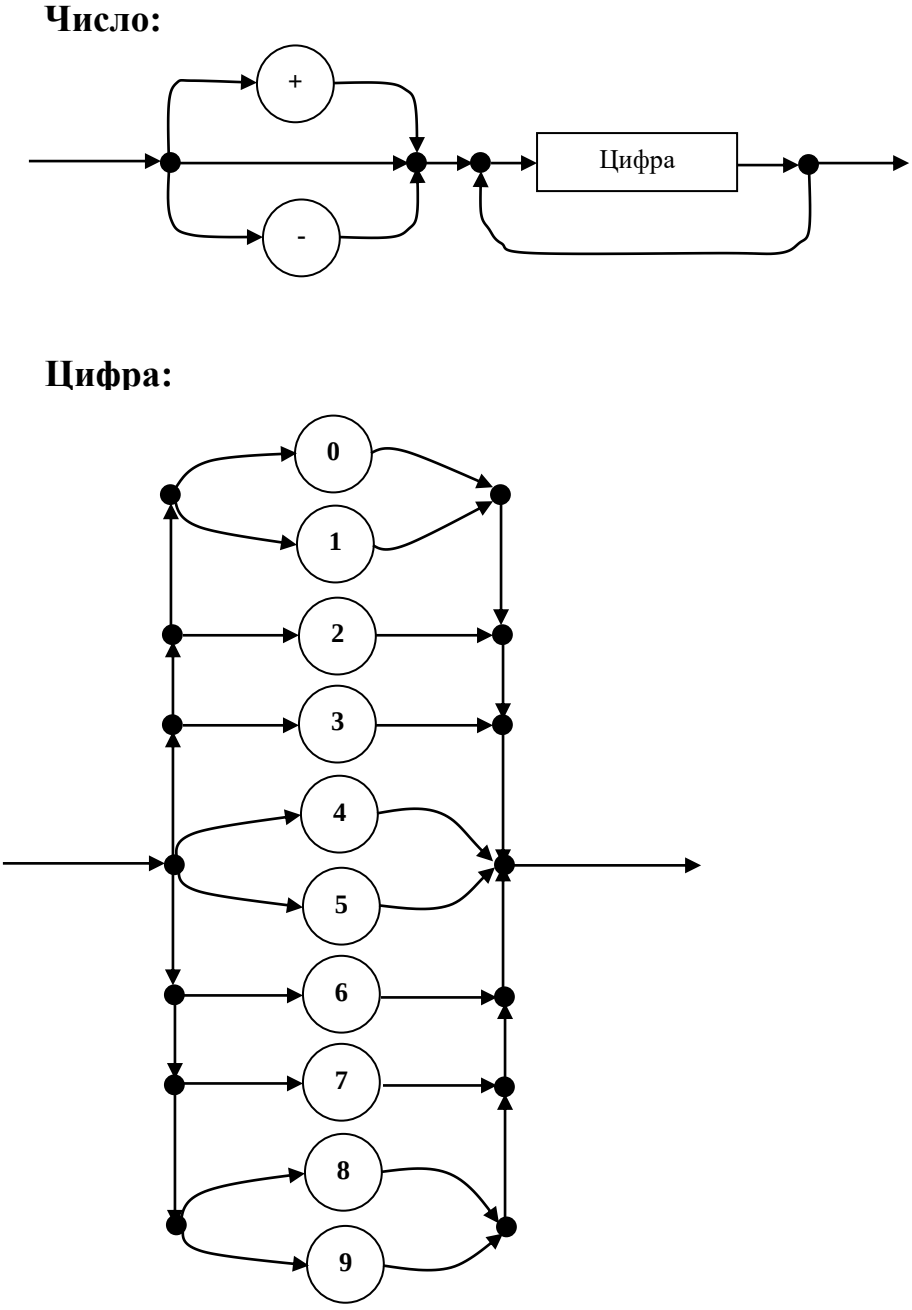
Чтобы построить цепочку символов, соответствующую какому-либо нетерминальному символу грамматики, надо рассмотреть диаграмму для этого символа. Тогда, начав движение от точки входа, надо двигаться по дугам графа диаграммы через любые вершины вплоть до точки выхода. При этом, проходя через вершину, обозначенную нетерминальным символом, этот символ следует поместить в результирующую цепочку. При прохождении через вершину, обозначенную цепочкой терминальных символов, эти символы также следует поместить в результирующую цепочку. При прохождении через узловые точки диаграммы над результирующей цепочкой никаких действий выполнять не надо. Через любую вершину графа диаграммы, в зависимости от возможного пути движения, можно пройти один раз, ни разу или сколь угодно много раз. Как только мы попадем в точку выхода диаграммы, построение результирующей цепочки закончено.

Результирующая цепочка, в свою очередь, может содержать нетерминальные символы. Чтобы заменить их на цепочки терминальных символов, нужно, опять же,

рассматривать соответствующие им диаграммы. И так до тех пор, пока в цепочке не останутся только терминальные символы. Очевидно, что для того, чтобы построить цепочку символов заданного языка, надо начать рассмотрение с диаграммы целевого символа грамматики.

Это удобный способ описания правил грамматик, оперирующий образами, а потому ориентированный исключительно на людей. Даже простое изложение его основных принципов здесь оказалось довольно громоздким, в то время как суть способа довольно проста. Это можно легко заметить, если посмотреть на описание понятия «число» из грамматики **G** с помощью диаграмм на рис. 1.1.

Рис. 1.1. Графическое представление грамматики целых десятичных чисел со знаком



Как уже было сказано выше, данный способ в основном применяется в литературе при изложении грамматик языков программирования. Для пользователей — разработчиков программ — он удобен, но практического применения в компиляторах пока не имеет.

Существуют и другие способы описания грамматик, но поскольку они не так часто встречаются в литературе, как два описанных выше способа, в данном учебном пособии они не рассматриваются.

Распознаватели. Общая схема распознавателя

Распознаватель (или разборщик) — это специальный автомат, который позволяет определить принадлежность цепочки символов некоторому языку. Задача распознавателя заключается в том, чтобы на основании исходной цепочки дать ответ на вопрос, принадлежит ли она заданному языку или нет. Распознаватели, как было сказано выше, представляют собой один из способов определения языка.

В общем виде распознаватель можно отобразить в виде условной схемы, представленной на рис. 1.2.

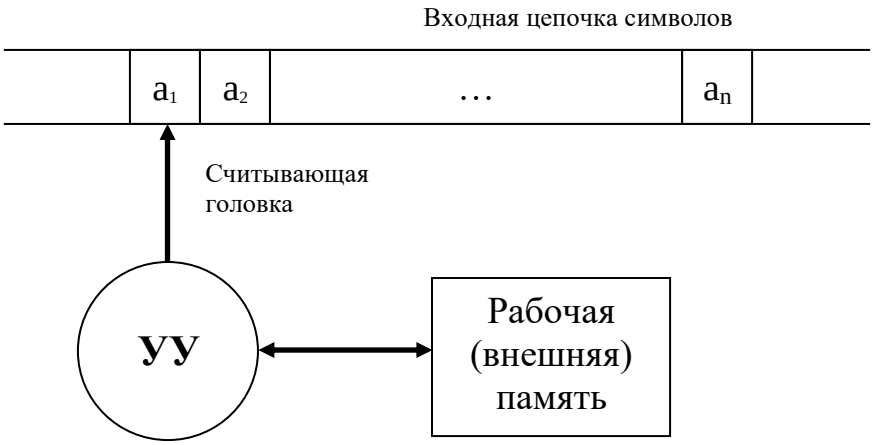


Рис. 1.2. Условная схема распознавателя

Следует подчеркнуть, что представленный рисунок — всего лишь условная схема, отображающая работу алгоритма распознавателя. Ни в коем случае не стоит искать подобного устройства в составе компьютера. Распознаватель, являющийся частью компилятора, представляет собой часть программного обеспечения компьютера.

Как видно из рисунка, распознаватель состоит из следующих основных компонентов:

- ленты, содержащей входную цепочку символов, и считывающей головки, обозревающей очередной символ в этой цепочке;
- устройства управления (УУ), которое координирует работу распознавателя, имеет некоторый набор состояний и конечную память (для хранения своего состояния и некоторой промежуточной информации);

- внешней (рабочей) памяти, которая может хранить некоторую информацию в процессе работы распознавателя и, в отличие от памяти УУ, имеет неограниченный объем.

Распознаватель работает с символами своего алфавита — алфавита распознавателя. Алфавит распознавателя конечен. Он включает в себя все допустимые символы входных цепочек, а также некоторый дополнительный алфавит символов, которые могут обрабатываться УУ и храниться в рабочей памяти распознавателя.

В процессе своей работы распознаватель может выполнять некоторые элементарные операции:

- чтение очередного символа из входной цепочки;
- сдвиг входной цепочки на заданное количество символов (вправо или влево);
- доступ к рабочей памяти для чтения или записи информации;
- преобразование информации в памяти УУ, изменение состояния УУ.

То, какие конкретно операции должны выполняться в процессе работы распознавателя, определяется в УУ.

Распознаватель работает по шагам или тактам. В начале такта, как правило, считывается очередной символ из входной цепочки, и в зависимости от этого символа УУ определяет, какие действия необходимо выполнить. Вся работа распознавателя состоит из последовательности тактов. В начале каждого такта состояние распознавателя определяется его конфигурацией. В процессе работы конфигурация распознавателя меняется.

Конфигурация распознавателя определяется следующими параметрами:

- содержимое входной цепочки символов и положение считывающей головки в ней;
- состояние УУ;
- содержимое внешней памяти.

Для распознавателя всегда задается определенная конфигурация, которая считается начальной конфигурацией. В начальной конфигурации считывающая головка обозревает первый символ входной цепочки, УУ находится в заданном начальном состоянии, а внешняя память либо пуста, либо содержит строго определенную информацию.

Кроме начального состояния для распознавателя задается одна или несколько конечных конфигураций. В конечной конфигурации считывающая головка, как правило, находится за концом исходной цепочки (часто для распознавателей вводят специальный символ, обозначающий конец входной цепочки).

Распознаватель *допускает входную цепочку символов* α, если, находясь в начальной конфигурации и получив на вход эту цепочку, он может проделать последовательность шагов, заканчивающуюся одной из его конечных конфигураций.

Формулировка «может проделать последовательность шагов» более точна, чем прямое указание «проделает последовательность шагов», так как для многих

распознавателей при одной и той же входной цепочке символов из начальной конфигурации могут быть допустимы различные последовательности шагов, не все из которых ведут к конечной конфигурации.

Язык, определяемый распознавателем, — это множество всех цепочек, которые допускает распознаватель.

Далее в этой книге рассмотрены конкретные типы распознавателей для различных языков. Но все, что было сказано здесь, относится ко всем без исключения типам распознавателей для всех типов языков.

Классификация распознавателей по структуре

Распознаватели можно классифицировать в зависимости от вида составляющих их компонентов: считывающего устройства, устройства управления (УУ) и внешней памяти.

По видам считывающего устройства распознаватели могут быть двусторонние и односторонние.

Односторонние распознаватели допускают перемещение считывающей головки по ленте входных символов только в одном направлении. Это значит, что на каждом шаге работы распознавателя считывающая головка может либо переместиться по ленте символов на некоторое число позиций в заданном направлении, либо остаться на месте. Поскольку все языки программирования подразумевают нотацию чтения исходной программы «слева направо», то так же работают и все распознаватели. Поэтому, когда говорят об односторонних распознавателях, то прежде всего имеют в виду левосторонние, которые читают входную цепочку слева направо и не возвращаются назад к уже прочитанной части цепочки.

Двусторонние распознаватели допускают, что считывающая головка может перемещаться относительно ленты входных символов в обоих направлениях: как вперед, от начала ленты к концу, так и назад, возвращаясь к уже прочитанным символам.

По видам устройства управления распознаватели бывают детерминированные и недетерминированные.

Распознаватель называется *детерминированным* в том случае, если для каждой допустимой конфигурации распознавателя, которая возникла на некотором шаге его работы, существует единственно возможная конфигурация, в которую распознаватель перейдет на следующем шаге работы.

В противном случае распознаватель называется *недетерминированным*. Недетерминированный распознаватель может иметь такую допустимую конфигурацию, для которой существует некоторое конечное множество конфигураций, возможных на следующем шаге работы. Достаточно иметь хотя бы одну такую конфигурацию, чтобы распознаватель был недетерминированным.

По видам внешней памяти распознаватели бывают следующих типов:

- распознаватели без внешней памяти;
- распознаватели с ограниченной внешней памятью;

- распознаватели с неограниченной внешней памятью.

У *распознавателей без внешней памяти* внешняя память полностью отсутствует. В процессе их работы используется только конечная память УУ.

Для *распознавателей с ограниченной внешней памятью* размер внешней памяти ограничен в зависимости от длины входной цепочки символов. Эти ограничения могут налагаться некоторой зависимостью объема памяти от длины цепочки — линейной, полиномиальной, экспоненциальной и т. д. Кроме того, для таких распознавателей может быть указан способ организации внешней памяти — стек, очередь, список и т п.

Распознаватели с неограниченной внешней памятью предполагают, что для их работы может потребоваться внешняя память неограниченного объема (вне зависимости от длины входной цепочки). У таких распознавателей предполагается память с произвольным методом доступа.

Вместе эти три составляющих позволяют организовать общую классификацию распознавателей. Например, в этой классификации возможен такой тип: «двусторонний недетерминированный распознаватель с линейно ограниченной стековой памятью».

Тип распознавателя в классификации определяет сложность создания такого распознавателя, а, следовательно, сложность разработки соответствующего программного обеспечения для компилятора. Чем выше в классификации стоит распознаватель, тем сложнее создавать алгоритм, обеспечивающий его работу. Разрабатывать двусторонние распознаватели сложнее, чем односторонние. Можно заметить, что недетерминированные распознаватели по сложности выше детерминированных. Зависимость затрат на создание алгоритма от типа внешней памяти также очевидна.

Классификация языков и грамматик

Выше уже упоминались различные типы грамматик, но не было указано, как и по какому принципу они подразделяются на типы. Для человека языки бывают простые и сложные, но это сугубо субъективное мнение, которое в первую очередь зависит от того, какой язык является для человека родным, а также от личности человека.

Для компиляторов языки также можно разделить на простые и сложные, но в данном случае существуют жесткие критерии для такого подразделения. Как будет показано далее, от того, к какому типу относится тот или иной язык программирования, зависит сложность компилятора для этого языка. Чем сложнее язык, тем выше вычислительные затраты компилятора на анализ цепочек исходной программы, написанной на этом языке, а, следовательно, сложнее сам компилятор и его структура. Для некоторых типов языков в принципе невозможно построить компилятор, который бы анализировал исходные тексты на этих языках за приемлемое время на основе ограниченных вычислительных ресурсов (именно поэтому до сих пор невозможно создавать программы на естественных языках, например, на русском или английском).

Классификация грамматик. Четыре типа грамматик по Хомскому

Согласно классификации, предложенной американским лингвистом Ноамом Хомским, профессором Массачусетского Технологического института, формальные грамматики классифицируются по структуре их правил. Если все без исключения правила грамматики удовлетворяют некоторой заданной структуре, то такую грамматику относят к определенному типу. Достаточно иметь в грамматике одно правило, не удовлетворяющее требованиям структуры правил, и она уже не попадает в заданный тип.

По классификации Хомского выделяют четыре типа грамматик.

Тип 0: грамматики с фразовой структурой

На структуру их правил не накладывается никаких ограничений: для грамматики вида G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT** правила имеют вид: $\alpha \to \beta$, где $\alpha \in V^+$, $\beta \in V^*$.

Это самый общий тип грамматик. В него подпадают все без исключения формальные грамматики, но часть из них, к общей радости, может быть также отнесена и к другим классификационным типам. Дело в том, что грамматики, которые относятся только к типу 0 и не могут быть отнесены к другим типам, являются самыми сложными по структуре.

Практического применения грамматики, относящиеся только к типу 0, не имеют.

Тип 1: контекстно-зависимые (КЗ) и неукорачивающие грамматики

В этот тип входят два основных класса грамматик:

Контекстно-зависимые грамматики G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT** имеют правила вида: $\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$, где $\alpha_1, \alpha_2 \in \mathbf{V}^*$, $A \in \mathbf{VN}$, $\beta \in \mathbf{V}^+$.

Неукорачивающие грамматики G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT** имеют правила вида: $\alpha \to \beta$, где $\alpha, \beta \in \mathbf{V}^+$, $|\beta| \geq |\alpha|$.

Структура правил КЗ-грамматик такова, что при построении предложений заданного ими языка один и тот же нетерминальный символ может быть заменен на ту или иную цепочку символов в зависимости от того контекста, в котором он встречается. Именно поэтому эти грамматики называют «контекстно-зависимыми» (КЗ). Цепочки α_1 и α_2 в правилах грамматики обозначают контекст (α_1 — левый контекст, а α_2 — правый контекст), в общем случае любая из них (или даже обе) может быть пустой. Говоря иными словами, значение одного и того же символа может быть различным в зависимости от того, в каком контексте он встречается.

Неукорачивающие грамматики имеют такую структуру правил, что при построении предложений языка, заданного грамматикой, любая цепочка символов может быть заменена на цепочку символов не меньшей длины. Отсюда и название «неукорачивающие».

Доказано, что эти два класса грамматик эквивалентны. Это значит, что для любого языка, заданного КЗ-грамматикой, можно построить неукорачивающую грамматику, которая будет задавать эквивалентный язык, и наоборот: для любого языка,

заданного неукорачивающей грамматикой, можно построить КЗ-грамматику, которая будет задавать эквивалентный язык.

При построении компиляторов такие грамматики не применяются, поскольку синтаксические конструкции языков программирования, рассматриваемые компиляторами, имеют более простую структуру и могут быть построены с помощью грамматик других типов. Что касается семантических ограничений языков программирования, то с точки зрения затрат вычислительных ресурсов их выгоднее проверять другими методами, а не с помощью КЗ-грамматик.

Тип 2: контекстно-свободные (КС) грамматики

Контекстно-свободные (КС) грамматики G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT** имеют правила вида: $A \to \beta$, где $A \in \mathbf{VN}$, $\beta \in \mathbf{V}^+$. Такие грамматики также иногда называют неукорачивающими контекстно-свободными (НКС) грамматиками (видно, что в правой части правила у них должен всегда стоять как минимум один символ).

Существует также почти эквивалентный им класс грамматик — укорачивающие контекстно-свободные (УКС) грамматики G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT**, правила которых могут иметь вид: $A \to \beta$, где $A \in \mathbf{VN}$, $\beta \in \mathbf{V}^*$.

Разница между этими двумя классами грамматик заключается лишь в том, что в УКС-грамматиках в правой части правил может присутствовать пустая цепочка (λ), а в НКС-грамматиках — нет. Отсюда ясно, что язык, заданный НКС-грамматикой, не может содержать пустой цепочки. Доказано, что эти два класса грамматик почти эквивалентны. В дальнейшем, когда речь будет идти о КС-грамматиках, уже не будет уточняться, какой класс грамматики (УКС или НКС) имеется в виду, если возможность наличия в языке пустой цепочки не имеет принципиального значения.

КС-грамматики широко используются при описании синтаксических конструкций языков программирования. Синтаксис большинства известных языков программирования основан именно на КС-грамматиках, поэтому в данном учебнике им уделяется большое внимание.

Внутри типа КС-грамматик кроме классов НКС и УКС выделяют еще целое множество различных классов грамматик, и все они относятся к типу 2. Далее, когда КС-грамматики будут рассматриваться более подробно, на некоторые из этих классов грамматик и их характерные особенности будет обращено особое внимание.

Тип 3: регулярные грамматики

К типу регулярных относятся два эквивалентных класса грамматик: леволинейные и праволинейные.

Леволинейные грамматики G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT** могут иметь правила двух видов: $A \to B\gamma$ или $A \to \gamma$, где $A,B \in \mathbf{VN}$, $\gamma \in \mathbf{VT}^*$.

В свою очередь, праволинейные грамматики G(**VT**,**VN**,**P**,S), **V** = **VN**∪**VT** могут иметь правила тоже двух видов: $A \to \gamma B$ или $A \to \gamma$, где $A,B \in \mathbf{VN}$, $\gamma \in \mathbf{VT}^*$.

Эти два класса грамматик эквивалентны и относятся к типу регулярных грамматик.

Регулярные грамматики используются при описании простейших конструкций языков программирования: идентификаторов, констант, строк, комментариев и т. д. Эти грамматики исключительно просты и удобны в использовании, поэтому в компиляторах на их основе строятся функции лексического анализа входного языка (принципы их построения будут рассмотрены далее).

Классификация языков

Языки классифицируются в соответствии с типами грамматик, с помощью которых они заданы. Причем, поскольку один и тот же язык в общем случае может быть задан сколь угодно большим количеством грамматик, которые могут относиться к различным классификационным типам, то для классификации самого языка среди всех его грамматик выбирается грамматика с максимально возможным классификационным типом. Например, если язык **L** может быть задан грамматиками $\mathbf{G}_1$ и $\mathbf{G}_2$, относящимися к типу 1 (КЗ), грамматикой $\mathbf{G}_3$, относящейся к типу 2 (КС) и грамматикой $\mathbf{G}_4$, относящейся к типу 3 (регулярные), то сам язык должен быть отнесен к типу 3 и является регулярным языком.

От классификационного типа языка зависит не только то, с помощью какой грамматики можно построить предложения этого языка, но также и то, насколько сложно распознать эти предложения. Распознать предложения — значит построить распознаватель для языка (третий способ задания языка). Классификация распознавателей рассмотрена далее, здесь же можно указать, что сложность распознавателя языка напрямую зависит от классификационного типа, к которому относится язык.

Сложность языка убывает с возрастанием номера классификационного типа языка. Самыми сложными являются языки типа 0, самыми простыми — языки типа 3.

Согласно классификации грамматик, также существует четыре типа языков.

Тип 0: языки с фразовой структурой

Это самые сложные языки, которые могут быть заданы только грамматикой, относящейся к типу 0. Для распознавания цепочек таких языков требуются вычислители, равномощные машине Тьюринга. Поэтому можно сказать, что если язык относится к типу 0, то для него невозможно построить компилятор, который гарантированно выполнял бы разбор предложений языка за ограниченное время на основе ограниченных вычислительных ресурсов.

К сожалению, практически все естественные языки общения между людьми, строго говоря, относятся именно к этому типу языков. Дело в том, что структура и значение фразы естественного языка может зависеть не только от контекста данной фразы, но и от содержания того текста, где эта фраза встречается. Одно и то же слово в естественном языке может не только иметь разный смысл в зависимости от контекста, но и играть различную роль в предложении. Именно поэтому столь велики сложности в автоматизации перевода текстов, написанных на естественных языках, а также отсутствуют (и видимо, никогда не появятся) компиляторы, которые бы воспринимали программы на основе таких языков.

Далее языки с фразовой структурой рассматриваться не будут.

Тип 1: контекстно-зависимые (КЗ) языки

Тип 1 — второй по сложности тип языков. В общем случае время на распознавание предложений языка, относящегося к типу 1, экспоненциально зависит от длины исходной цепочки символов.

Языки и грамматики, относящиеся к типу 1, применяются в анализе и переводе текстов на естественных языках. Распознаватели, построенные на их основе, позволяют анализировать тексты с учетом контекстной зависимости в предложениях входного языка (но они не учитывают содержание текста, поэтому в общем случае для точного перевода с естественного языка требуется вмешательство человека). На основе таких грамматик может выполняться автоматизированный перевод с одного естественного языка на другой, ими могут пользоваться сервисные функции проверки орфографии и правописания в языковых процессорах.

В компиляторах КЗ-языки не используются, поскольку языки программирования имеют более простую структуру, поэтому здесь они подробно не рассматриваются.

Тип 2: контекстно-свободные (КС) языки

КС-языки лежат в основе синтаксических конструкций большинства современных языков программирования, на их основе функционируют некоторые довольно сложные командные процессоры.

В общем случае время на распознавание предложений языка, относящегося к типу 2, полиномиально зависит от длины входной цепочки символов (в зависимости от класса языка это либо кубическая, либо квадратичная зависимость). Однако среди КС-языков существует много классов языков, для которых эта зависимость линейна. Практически все языки программирования можно отнести к одному из таких классов.

КС-языки подробно рассматриваются в главе «Синтаксические анализаторы» данного учебника.

Тип 3: регулярные языки

Регулярные языки — самый простой тип языков. Поэтому они являются самым широко используемым типом языков в области вычислительных систем. Время на распознавание предложений регулярного языка линейно зависит от длины входной цепочки символов.

Как уже было сказано выше, регулярные языки лежат в основе простейших конструкций языков программирования (идентификаторов, констант и т. п.), кроме того, на их основе строятся многие мнемокоды машинных команд (языки ассемблеров), а также простейшие командные процессоры, символьные управляющие команды и другие подобные структуры.

Регулярные языки — очень удобное средство. Для работы с ними можно использовать регулярные множества и выражения, конечные автоматы. Регулярные языки подробно рассматриваются в главе «Лексические анализаторы».

Задача разбора. Классификация распознавателей по типам

Для каждого языка программирования важно не только уметь построить текст программы на этом языке, но и определить принадлежность имеющегося текста к данному языку. Именно эту задачу решают компиляторы в числе прочих задач (компилятор должен не только распознать исходную программу, но и построить эквивалентную ей результирующую программу). В отношении исходной программы компилятор выступает как распознаватель, а человек, создавший программу на некотором языке программирования, выступает в роли генератора цепочек этого языка.

Грамматики и распознаватели — два независимых метода, которые реально могут быть использованы для определения какого-либо языка. Однако при создании компилятора для некоторого языка программирования возникает задача, которая требует связать между собой эти методы задания языков. Разработчики компилятора всегда имеют дело с уже определенным языком программирования. Грамматика для синтаксических конструкций этого языка известна. Задача разработчиков заключается в том, чтобы построить распознаватель для заданного языка, который затем будет основой синтаксического анализатора в компиляторе.

Таким образом, *задача разбора* в общем виде заключается в следующем: на основе имеющейся грамматики некоторого языка построить распознаватель для этого языка. Заданная грамматика и распознаватель должны быть эквивалентны, то есть определять один и тот же язык (часто допускается, чтобы они были почти эквивалентны, поскольку пустая цепочка во внимание обычно не принимается).

Задача разбора в общем виде может быть решена не для всех языков. Разработчиков компиляторов интересуют прежде всего синтаксические конструкции языков программирования. Для этих конструкций доказано, что задача разбора для них разрешима. Более того, для них найдены формальные методы ее решения. Описанию и обоснованию именно методов решения задачи разбора будет посвящена большая часть материала последующих глав данной книги.

Поскольку языки программирования не являются чисто формальными языками и несут в себе некоторый смысл (семантику), то задача разбора для создания реальных компиляторов понимается несколько шире, чем она формулируется для чисто формальных языков. Компилятор должен не просто установить принадлежность входной цепочки символов заданному языку, но и определить ее смысловую нагрузку. Для этого необходимо выявить те правила грамматики, на основании которых цепочка была построена.

Если же входная цепочка символов не принадлежит заданному языку — исходная программа содержит ошибку — разработчику программы не интересно просто узнать сам факт наличия ошибки. В данном случае задача разбора также расширяется: распознаватель в составе компилятора должен не только установить факт присутствия ошибки во входной программе, но и по возможности определить тип ошибки и то место во входной цепочке символов, где она встречается.

Как было показано ранее, классификация распознавателей определяет сложность алгоритма работы распознавателя. Но сложность распознавателя также напрямую связана с типом языка, входные цепочки которого может принимать (допускать)

распознаватель. Выше было определено четыре основных типа языков. Доказано, что для каждого из этих типов языков существует свой тип распознавателя.

Для *языков с фразовой структурой* (тип 0) необходим распознаватель, равномощный машине Тьюринга — недетерминированный двусторонний автомат, имеющий неограниченную внешнюю память. Поэтому для языков данного типа нельзя гарантировать, что за ограниченное время на ограниченных вычислительных ресурсах распознаватель завершит работу. Отсюда можно заключить, что практического применения языки с фразовой структурой не имеют.

Для *контекстно-зависимых языков* (тип 1) распознавателями являются двусторонние недетерминированные автоматы с линейно ограниченной внешней памятью. Алгоритм работы такого автомата в общем случае имеет экспоненциальную сложность — количество шагов (тактов), необходимых автомату для распознавания входной цепочки, экспоненциально зависит от длины этой цепочки.

Такой алгоритм распознавателя уже может быть реализован в программном обеспечении компьютера — зная длину входной цепочки, всегда можно сказать, за какое максимально возможное время будет принято решение о принадлежности цепочки данному языку, и какие вычислительные ресурсы для этого потребуются. Однако экспоненциальная зависимость времени разбора от длины цепочки существенно ограничивает применение распознавателей для КЗ-языков. Как правило, такие распознаватели применяются для автоматизированного перевода и анализа текстов на естественных языках (следует также напомнить, что, поскольку естественные языки более сложны, чем КЗ-языки, после такой обработки часто требуется вмешательство человека).

В рамках этого учебника КЗ-языки не рассматриваются.

Для *контекстно-свободных языков* (тип 2) распознавателями являются односторонние недетерминированные автоматы с магазинной (стековой) внешней памятью — МП-автоматы. При простейшей реализации алгоритма работы такого автомата он имеет экспоненциальную сложность, однако путем некоторых усовершенствований алгоритма можно добиться полиномиальной (кубической) зависимости времени, необходимого на разбор входной цепочки, от длины этой цепочки. Следовательно, можно говорить о полиномиальной сложности распознавателя для КС-языков.

Среди всех КС-языков можно выделить класс детерминированных КС-языков (ДКС), распознавателями для которых являются детерминированные МП-автоматы — ДМП-автоматы. Для таких ДКС-языков существует алгоритм работы распознавателя с квадратичной сложностью.

Среди всех ДКС-языков существуют такие классы языков, для которых возможно построить линейный распознаватель — распознаватель, у которого время разбора цепочки имеет линейную зависимость от длины цепочки. Именно эти языки представляют интерес при построении компиляторов. Синтаксические конструкции всех существующих языков программирования могут быть отнесены к одному из таких классов языков. Поэтому в главе, посвященной синтаксическим анализаторам, в первую очередь будет уделено внимание именно этим классам языков.

Тем не менее, следует помнить, что только синтаксические конструкции языков программирования допускают разбор с помощью распознавателей КС-языков. Сами

языки программирования, как уже было сказано, не могут быть полностью отнесены к типу КС-языков, поскольку предполагают контекстную зависимость в тексте исходной программы (например, такую, как необходимость предварительного описания переменных). Поэтому кроме синтаксического разбора все компиляторы предполагают дополнительный семантический анализ текста исходной программы. Этого можно было бы избежать, если построить компилятор на основе КЗ-распознавателя, но скорость работы такого компилятора была бы недопустима низка, поскольку время разбора в таком варианте будет экспоненциально зависеть от длины исходной программы. Комбинация из распознавателя КС-языка и дополнительного семантического анализатора является более эффективной с точки зрения скорости разбора исходной программы.

Для *регулярных языков* (тип 3) распознавателями являются односторонние недетерминированные автоматы без внешней памяти — конечные автоматы (КА). Это очень простой тип распознавателя, который предполагает линейную зависимость времени разбора входной цепочки от ее длины. Кроме того, КА имеют важную особенность: любой недетерминированный КА всегда может быть преобразован в детерминированный. Это обстоятельство существенно упрощает разработку программного обеспечения, обеспечивающего функционирование распознавателя.

Простота и высокая скорость работы распознавателей определяют широкую область применения регулярных языков.

В компиляторах распознаватели на основе регулярных языков используются для лексического анализа текста исходной программы — выделения в нем простейших конструкций языка (лексем), таких как идентификаторы, строки, константы и т. п. Это позволяет существенно сократить объем исходной информации и упрощает синтаксический разбор программы. Более подробно взаимодействие лексического и синтаксического анализаторов текста программы рассмотрено дальше, в главе, посвященной лексическим анализаторам.

Кроме компиляторов регулярные языки находят применение еще во многих областях, связанных с разработкой программного обеспечения. На их основе функционируют многие командные процессоры как в системном, так и в прикладном программном обеспечении. Для регулярных языков существуют развитые математически обоснованные методы, которые позволяют облегчить создание распознавателей. Они положены в основу существующих программных средств, которые позволяют автоматизировать этот процесс.

Регулярные языки и связанные с ними математические методы рассматриваются в главе «Лексические анализаторы» данного учебника.

Примеры классификации языков и грамматик

Классификация языков идет от простого к сложному. Если мы имеем дело с регулярным языком, то можно утверждать, что он также является и КС, и КЗ и даже языком с фразовой структурой. В то же время, известно, что существуют КС-языки, которые не являются регулярными, и существуют КЗ-языки, которые не являются ни регулярными, ни контекстно-свободными.

Далее приводятся примеры некоторых языков указанных типов.

Рассмотрим в качестве первого примера ту же грамматику для целых десятичных чисел со знаком $G_1\{0,1,2,3,4,5,6,7,8,9,-,+\},\{S,T,F\},\mathbf{P}_1,S)$:

$\mathbf{P}_1$:

$S \rightarrow T \mid +T \mid -T$

$T \rightarrow F \mid TF$

$F \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

По структуре своих правил данная грамматика $\mathbf{G}_1$ относится к КС-грамматикам (тип 2). Конечно, ее можно отнести к типу 0 и к типу 1, но максимально возможным является тип 2, поскольку к типу 3 эту грамматику отнести нельзя: строка $T \rightarrow F \mid TF$ содержит правило $T \rightarrow TF$, которое недопустимо для типа 3. И хотя все остальные правила типу 3 соответствует, одного несоответствия достаточно.

Для того же самого языка (целых десятичных чисел со знаком) можно построить и другую грамматику $\mathbf{G}_1'(\{0,1,2,3,4,5,6,7,8,9,-,+\},\{S,T\},\mathbf{P}_1',S)$:

$\mathbf{P}_1'$:

$S \rightarrow T \mid +T \mid -T$

$T \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \mid 0T \mid 1T \mid 2T \mid 3T \mid 4T \mid 5T \mid 6T \mid 7T \mid 8T \mid 9T$

По структуре своих правил эта грамматика $\mathbf{G}_1'$ является праволинейной и может быть отнесена к регулярным грамматикам (тип 3).

Для этого же языка можно построить эквивалентную леволинейную грамматику (тип 3) $\mathbf{G}_1''(\{0,1,2,3,4,5,6,7,8,9,-,+\},\{S,T\},\mathbf{P}_1'',S)$:

$\mathbf{P}_1''$:

$T \rightarrow + \mid - \mid \lambda$

$S \rightarrow T0 \mid T1 \mid T2 \mid T3 \mid T4 \mid T5 \mid T6 \mid T7 \mid T8 \mid T9 \mid S0 \mid S1 \mid S2 \mid S3 \mid S4 \mid S5 \mid S6 \mid S7 \mid S8 \mid S9$

Следовательно, язык $\mathbf{L}_1$ целых десятичных чисел со знаком, заданный грамматиками $\mathbf{G}_1$, $\mathbf{G}_1'$ и $\mathbf{G}_1''$, относится к регулярным языкам (тип 3).

В качестве второго примера возьмем грамматику $G_2(\{0,1\},\{A,S\},\mathbf{P}_2,S)$ с правилами $\mathbf{P}_2$:

$S \rightarrow 0A1$

$0A \rightarrow 00A1$

$A \rightarrow \lambda$

Эта грамматика относится к типу 0. Она определяет язык, множество предложений которого можно было бы записать так: $L(G_2) = \{0^n1^n \mid n>0\}$.

Для того же самого языка можно построить также КЗ-грамматику $G_2'(\{0,1\},\{A,S\},\mathbf{P}_2',S)$ с правилами $\mathbf{P}_2'$:

$S \rightarrow 0A1 \mid 01$

$0A \rightarrow 00A1 \mid 001$

Однако для того же самого языка можно использовать и КС-грамматику $G_2''(\{0,1\},\{S\},\mathbf{P}_2'',S)$ с правилами $\mathbf{P}_2''$:

$S \rightarrow 0S1 \mid 01$

Следовательно, язык $\mathbf{L}_2 = \{0^n1^n \mid n>0\}$ является КС-языком (тип 2).

В третьем примере рассмотрим грамматику $G_3(\{a,b,c\},\{B,C,D,S\},\mathbf{P}_3,S)$ с правилами $\mathbf{P}_3$:

$S \rightarrow BD$

$B \rightarrow aBbC \mid ab$

$Cb \rightarrow bC$

$CD \rightarrow Dc$

$bDc \rightarrow bcc$

$abD \rightarrow abc$

Эта грамматика относится к типу 1. Очевидно, что она является неукорачивающей. Она определяет язык, множество предложений которого можно было бы записать так: $L(\mathbf{G}_3) = \{a^nb^nc^n \mid n>0\}$. Известно, что этот язык не является КС-языком, поэтому для него нельзя построить грамматики типов 2 или 3.

Но для того же самого языка можно построить КЗ-грамматику $G_3'(\{a,b,c\},\{B,C,D,E,F,S\},\mathbf{P}_3', S)$ с правилами $\mathbf{P}_3'$:

$S \rightarrow abc \mid AE$

$A \rightarrow aABC \mid aBC$

$CBC \rightarrow CDC$

$CDC \rightarrow BDC$

$BDC \rightarrow BCC$

$CCE \rightarrow CFE$

$CFE \rightarrow CFc$

$CFc \rightarrow CEc$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bCE \rightarrow bCc$

$bCc \rightarrow bc$

Язык $\mathbf{L}_3 = \{a^n b^n c^n \mid n>0\}$ является контекстно-зависимым (тип 1).

Конечно, для произвольного языка, заданного некоторой грамматикой, в общем случае довольно сложно определить его тип. Не всегда можно так просто построить грамматику максимально возможного типа для произвольного языка. К тому же требуется еще доказать, что две грамматики (первоначально имеющаяся и вновь построенная) эквивалентны, а также то, что для того же языка не существует грамматики с большим по номеру типом.

Для многих языков, и в частности, для КС-языков и регулярных языков, существуют специальным образом сформулированные утверждения, которые позволяют проверить принадлежность языка к указанному типу. Такие утверждения (леммы) можно найти в [4 т.1, 15, 29]. Тогда для произвольного языка достаточно лишь доказать нужную лемму, и после этого можно утверждать, что данный язык относится к тому или иному типу. Преобразование грамматик в этом случае не требуется.

Тем не менее иногда возникает задача построения для имеющегося языка грамматики более простого типа, чем данная. И даже в том случае, когда тип языка уже известен, эта задача в общем случае не имеет формального решения (проблема преобразования грамматик рассматривается далее).

Цепочки вывода. Сентенциальная форма

Вывод. Цепочки вывода

Выводом называется процесс порождения предложения языка на основе правил определяющей язык грамматики. Чтобы дать формальное определение процессу вывода, необходимо ввести еще несколько дополнительных понятий.

Цепочка $\beta = \delta_1\gamma\delta_2$ называется *непосредственно выводимой* из цепочки $\alpha = \delta_1\omega\delta_2$ в грамматике $G(\mathbf{VT},\mathbf{VN},\mathbf{P},S)$, $\mathbf{V} = \mathbf{VT}\cup\mathbf{VN}$, $\delta_1, \gamma, \delta_2 \in \mathbf{V}^*$, $\omega \in \mathbf{V}^+$, если в грамматике **G** существует правило: $\omega \to \gamma \in \mathbf{P}$. Непосредственная выводимость цепочки β из цепочки α обозначается так: $\alpha \Rightarrow \beta$. Согласно определению при выводе $\alpha \Rightarrow \beta$ выполняется подстановка подцепочки γ вместо подцепочки ω. Иными словами, цепочка β выводима из цепочки α в том случае, если можно взять несколько символов в цепочке α, поменять их на другие символы согласно некоторому правилу грамматики и получить цепочку β.

В формальном определении непосредственной выводимости любая из цепочек δ_1 или δ_2 (а равно и обе эти цепочки) может быть пустой. В предельном случае вся цепочка α может быть заменена цепочкой β, тогда в грамматике **G** должно существовать правило: $\alpha \to \beta \in \mathbf{P}$.

Цепочка β называется *выводимой* из цепочки α (обозначается $\alpha \Rightarrow^* \beta$) в том случае, если выполняется одно из двух условий:

- β непосредственно выводима из α ($\alpha \Rightarrow \beta$);
- $\exists\, \gamma$, такая, что: γ выводима из α и β непосредственно выводима из γ ($\alpha \Rightarrow^* \gamma$ и $\gamma \Rightarrow \beta$).

Это рекурсивное определение выводимости цепочки. Суть его заключается в том, что цепочка β выводима из цепочки α, если $\alpha \Rightarrow \beta$ или же если можно построить последовательность непосредственно выводимых цепочек от α к β следующего вида: $\alpha \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_i \Rightarrow \ldots \Rightarrow \gamma_n \Rightarrow \beta$, $n \geq 1$. В этой последовательности каждая последующая цепочка γ_i непосредственно выводима из предыдущей цепочки γ_{i-1}.

Такая последовательность непосредственно выводимых цепочек называется выводом или *цепочкой вывода*. Каждый переход от одной непосредственно выводимой цепочки к следующей в цепочке вывода называется *шагом вывода*. Очевидно, что шагов вывода в цепочке вывода всегда на один больше, чем промежуточных цепочек. Если цепочка β непосредственно выводима из цепочки α: $\alpha \Rightarrow \beta$, то имеется всего один шаг вывода.

Если цепочка вывода из α к β содержит одну или более промежуточных цепочек (два или более шагов вывода), то она имеет специальное обозначение $\alpha \Rightarrow^+ \beta$ (говорят, что цепочка β *нетривиально выводима* из цепочки α). Если количество шагов вывода известно, то его можно указать непосредственно у знака выводимости цепочек. Например, запись $\alpha \Rightarrow^4 \beta$ означает, что цепочка β выводится из цепочки α за 4 шага вывода.[1]

Возьмем в качестве примера ту же грамматику для целых десятичных чисел со знаком `G({0,1,2,3,4,5,6,7,8,9,-,+},{S,T,F},`**`P`**`,S)`:

P:

S → T | +T | −T

T → F | TF

F → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Построим в ней несколько произвольных цепочек вывода (для понимания каждого шага вывода подцепочка, для которой выполняется подстановка, выделена жирным шрифтом):

1. **S** ⇒ −**T** ⇒ −**T**F ⇒ −**T**FF ⇒ −**F**FF ⇒ −4**F**F ⇒ −47**F** ⇒ −479
2. **S** ⇒ **T** ⇒ T**F** ⇒ **T**8 ⇒ **F**8 ⇒ 18
3. **T** ⇒ T**F** ⇒ **T**0 ⇒ T**F**0 ⇒ **T**50 ⇒ **F**50 ⇒ 350
4. **T**FT ⇒ TFF**T** ⇒ **T**FFF ⇒ **F**FFF ⇒ 1FF**F** ⇒ 1**F**F4 ⇒ 10**F**4 ⇒ 1004
5. **F** ⇒ 5

Получили следующие выводы:

1. S ⇒ * −479 или S ⇒ + −479 или S ⇒ ^{7} −479

---

[1] В литературе встречается также обозначение: $\alpha \Rightarrow^0 \beta$, которое означает, что цепочка α выводима из цепочки β за 0 шагов — иными словами, в таком случае эти цепочки равны: α = β. Подразумевается, что обозначение вывода $\alpha \Rightarrow^* \beta$ допускает и такое толкование — включает в себя вариант $\alpha \Rightarrow^0 \beta$.

2. S $\Rightarrow$ * 18 или S $\Rightarrow$ + 18 или S $\Rightarrow^{5}$ 18

3. T $\Rightarrow$ * 350 или T $\Rightarrow$ + 350 или T $\Rightarrow^{6}$ 350

4. TFT $\Rightarrow$ * 1004 или TFT $\Rightarrow$ + 1004 или TFT $\Rightarrow^{7}$ 1004

5. F $\Rightarrow$ * 5 или F $\Rightarrow^{1}$ 5 (утверждение F $\Rightarrow$ + 5 неверно!)

Все эти выводы построены на основе грамматики **G**. В принципе, в этой грамматике (как, практически, и в любой другой грамматике реального языка) можно построить сколь угодно много цепочек вывода.

Возьмем в качестве второго примера грамматику $G_3(\{a,b,c\},\{B,C,D,S\},\mathbf{P}_3,S)$ с правилами $\mathbf{P}_3$, которая уже рассматривалась выше:

S $\rightarrow$ BD

B $\rightarrow$ aBbC | ab

Cb $\rightarrow$ bC

CD $\rightarrow$ Dc

bDc $\rightarrow$ bcc

abD $\rightarrow$ abc

Как было сказано ранее, она задает язык $L(\mathbf{G}_3) = \{a^n b^n c^n \mid n>0\}$. Рассмотрим пример вывода предложения aaaabbbbcccc языка $L(\mathbf{G}_3)$ на основе грамматики $\mathbf{G}_3$:

S $\Rightarrow$ **B**D $\Rightarrow$ a**B**bCD $\Rightarrow$ aa**B**bCbCD $\Rightarrow$ aaa**B**bCbCbCD $\Rightarrow$ aaaabb**Cb**CbCD $\Rightarrow$ aaaabbbC**Cb**CD $\Rightarrow$ aaaabbb**Cb**CCD $\Rightarrow$ aaaabbbbCC**CD** $\Rightarrow$ aaaabbbbC**CD**c $\Rightarrow$ aaaabbbb**CD**cc $\Rightarrow$ aaaabbb**bDc**cc $\Rightarrow$ aaaabbbbcccc.

Тогда для грамматики $\mathbf{G}_3$ получаем вывод: S $\Rightarrow^*$ aaaabbbbcccc.

Сентенциальная форма грамматики. Язык, заданный грамматикой

Вывод называется *законченным* (или *конечным*), если на основе цепочки β, полученной в результате этого вывода, нельзя больше сделать ни одного шага вывода. Иначе говоря, вывод называется законченным, если цепочка β, полученная в результате этого вывода, пустая или содержит только терминальные символы грамматики G(**VT**,**VN**,**P**,S): $\beta \in \mathbf{VT}^*$. Цепочка β, полученная в результате законченного вывода, называется *конечной* цепочкой вывода.

В рассмотренном выше примере все построенные выводы являются законченными, а например, вывод S $\Rightarrow^*$ −4FF (из первой цепочки в примере) будет незаконченным.

Цепочка символов $\alpha \in \mathbf{V}^*$ называется *сентенциальной формой* грамматики G(**VT**,**VN**,**P**,S), **V** = **VT**$\cup$**VN**, если она выводима из целевого символа грамматики S: S $\Rightarrow^*$ α. Если цепочка $\alpha \in \mathbf{VT}^*$ получена в результате законченного вывода, то она называется *конечной сентенциальной формой*.

Из рассмотренного выше примера можно заключить, что цепочки символов `−479` и `18` являются конечными сентенциальными формами грамматики целых десятичных чисел со знаком, так как существуют выводы `S` $\Rightarrow^*$ `−479` и `S` $\Rightarrow^*$ `18` (выводы 1 и 2). Цепочка `F8` из вывода 2, например, тоже является сентенциальной формой, поскольку справедливо `S` $\Rightarrow^*$ `F8`, но она не является конечной цепочкой вывода. В то же время, в выводах 3, 4 и 5 примера явно не присутствуют сентенциальные формы. На самом деле, цепочки `350`, `1004` и `5` тоже являются конечными сентенциальными формами. Чтобы доказать это, необходимо просто построить другие цепочки вывода (например, для цепочки `5` строим: `S` $\Rightarrow$ `T` $\Rightarrow$ `F` $\Rightarrow$ `5` и получаем `S` $\Rightarrow^*$ `5`). А вот цепочка `TFT` (вывод 4) не выводима из целевого символа грамматики S, а потому сентенциальной формой не является.

Язык **L**, заданный грамматикой G(**VT**,**VN**,**P**,S), — это множество всех конечных сентенциальных форм грамматики **G**. Язык **L**, заданный грамматикой **G**, обозначается как L(**G**). Очевидно, что алфавитом такого языка L(**G**) будет множество терминальных символов грамматики **VT**, поскольку все конечные сентенциальные формы грамматики — это цепочки над алфавитом **VT**.

Следует помнить, что две грамматики G(**VT**,**VN**,**P**,S) и G'(**VT**',**VN**',**P**',S') называются эквивалентными, если эквивалентны заданные ими языки: L(**G**) = L(**G**'). Очевидно, что эквивалентные грамматики должны иметь, по крайней мере, пересекающиеся множества терминальных символов **VT**$\cap$**VT**' $\neq \varnothing$ (как правило, эти множества даже совпадают **VT**=**VT**'), а вот множества нетерминальных символов, правила грамматики и целевой символ у них могут кардинально отличаться.

Левосторонний и правосторонний выводы

Вывод называется *левосторонним*, если в нем на каждом шаге вывода правило грамматики применяется всегда к крайнему левому нетерминальному символу в цепочке. Другими словами, вывод называется левосторонним, если на каждом шаге вывода происходит подстановка цепочки символов на основании правила грамматики вместо крайнего левого нетерминального символа в исходной цепочке.

Аналогично, вывод называется *правосторонним*, если в нем на каждом шаге вывода правило грамматики применяется всегда к крайнему правому нетерминальному символу в цепочке.

Если рассмотреть цепочки вывода из того же примера, то в нем выводы 1 и 5 являются левосторонними, выводы 2, 3 и 5 — правосторонними (вывод 5 одновременно является и лево- и правосторонним), а вот вывод 4 не является ни левосторонним, ни правосторонним.

Для грамматик типов 2 и 3 (КС-грамматик и регулярных грамматик) для любой сентенциальной формы всегда можно построить левосторонний или правосторонний выводы. Для грамматик других типов это не всегда возможно, так как по структуре их правил не всегда можно выполнить замену крайнего левого или крайнего правого нетерминального символа в цепочке.

А вот рассмотренный выше вывод `S` $\Rightarrow^*$ `aaaabbbbcccc` для грамматики $\mathbf{G}_3$, задающей язык $L(\mathbf{G}_3) = \{0^n1^n \mid n>0\}$ не является ни левосторонним, ни правосторонним. Грамматика относится к типу 1, и в данном случае для нее нельзя

построить такой вывод, на каждом шаге которого только один нетерминальный символ заменялся бы на цепочку символов.

Дерево вывода. Методы построения дерева вывода

Деревом вывода грамматики G(**VT**,**VN**,**P**,S) называется дерево (граф), которое соответствует некоторой цепочке вывода и удовлетворяет следующим условиям:

- каждая вершина дерева обозначается символом грамматики $A \in (\mathbf{VT} \cup \mathbf{VN} \cup \{\lambda\})$;
- корнем дерева является вершина, обозначенная целевым символом грамматики — S;
- листьями дерева (концевыми вершинами) являются вершины, обозначенные терминальными символами грамматики или символом пустой цепочки λ;
- если некоторый узел дерева обозначен нетерминальным символом $A \in \mathbf{VN}$, а связанные с ним узлы — символами $b_1,b_2,\ldots,b_n$; $n > 0$, $\forall n \geq i > 0$: $b_i \in (\mathbf{VT} \cup \mathbf{VN} \cup \{\lambda\})$, то в грамматике G(**VT**,**VN**,**P**,S) существует правило $A \rightarrow b_1,b_2,\ldots,b_n \in \mathbf{P}$.

Из определения видно, что по структуре правил дерево вывода в указанном виде всегда можно построить только для грамматик типов 2 и 3 (контекстно-свободных и регулярных). Для грамматик других типов дерево вывода в таком виде можно построить не всегда (либо же оно будет иметь несколько иной вид).

На основе рассмотренного выше примера построим деревья вывода для цепочек вывода 1 и 2. Эти деревья приведены на рис. 1.3.

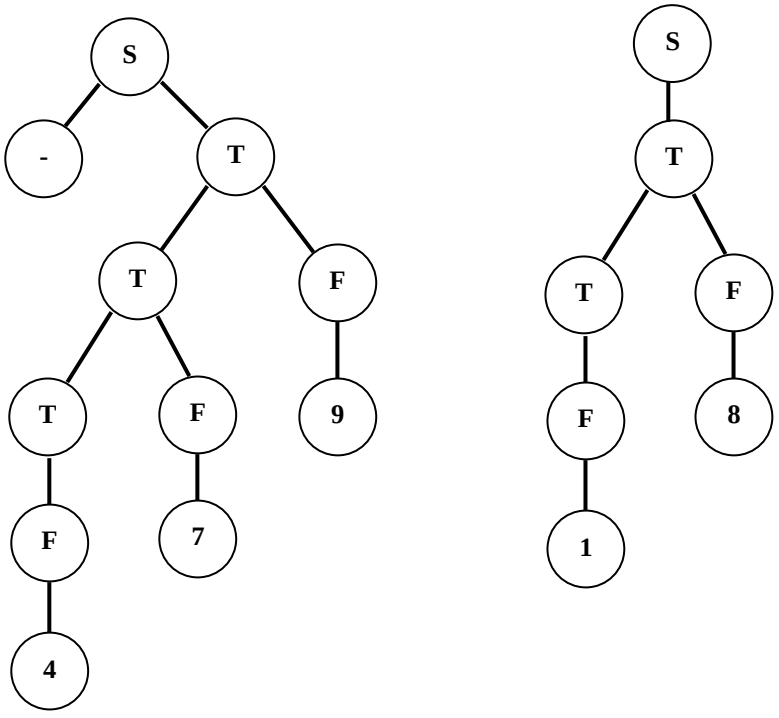


Рис. 1.3. Примеры деревьев вывода для грамматики целых десятичных чисел со знаком

Для того чтобы построить дерево вывода, достаточно иметь только цепочку вывода. Дерево вывода можно построить двумя способами: сверху вниз и снизу вверх. Для строго формализованного построения дерева вывода всегда удобнее пользоваться строго определенным выводом: либо левосторонним, либо правосторонним.

При построении дерева вывода сверху вниз построение начинается с целевого символа грамматики, который помещается в корень дерева. Затем в грамматике выбирается необходимое правило, и на первом шаге вывода корневой символ раскрывается на несколько символов первого уровня. На втором шаге среди всех концевых вершин дерева выбирается крайняя (крайняя левая — для левостороннего вывода, крайняя правая — для правостороннего) вершина, обозначенная нетерминальным символом, для этой вершины выбирается нужное правило грамматики, и она раскрывается на несколько вершин следующего уровня. Построение дерева заканчивается, когда все концевые вершины обозначены терминальными символами, в противном случае надо вернуться ко второму шагу и продолжить построение.

Построение дерева вывода снизу вверх начинается с листьев дерева. В качестве листьев выбираются терминальные символы конечной цепочки вывода, которые на первом шаге построения образуют последний уровень (слой) дерева. Построение дерева идет по слоям. На втором шаге построения в грамматике выбирается правило, правая часть которого соответствует крайним символам в слое дерева (крайним правым символам при правостороннем выводе и крайним левым — при левостороннем). Выбранные вершины слоя соединяются с новой вершиной, которая выбирается из левой части правила. Новая вершина попадает в слой дерева вместо выбранных вершин. Построение дерева закончено, если достигнута корневая вершина (обозначенная целевым символом), а иначе надо вернуться ко второму шагу и повторить его над полученным слоем дерева.

Поскольку все известные языки программирования имеют нотацию записи «слева — направо», компилятор также всегда читает входную программу слева на право (и сверху вниз, если программа разбита на несколько строк). Поэтому для построения дерева вывода методом «сверху вниз», как правило, используется левосторонний вывод, а для построения «снизу вверх» — правосторонний вывод. На эту особенность компиляторов стоит обратить внимание. Нотация чтения программ «слева направо» влияет не только на порядок разбора программы компилятором (для пользователя это, как правило, не имеет значения), но и на порядок выполнения операций — при отсутствии скобок большинство равноправных операций выполняются в порядке слева направо, а это уже имеет существенное значение.

Проблемы однозначности и эквивалентности грамматик

Однозначные и неоднозначные грамматики

Рассмотрим некоторую грамматику `G({+,-,*,/,(,),a,b},{S},P,S)`:

P:

$S \rightarrow S+S \mid S-S \mid S*S \mid S/S \mid (S) \mid a \mid b$

Видно, что представленная грамматика определяет язык арифметических выражений с четырьмя основными операциями (сложение, вычитание, умножение и деление) и скобками над операндами a и b. Примерами предложений этого языка могут служить: `a*b+a`, `a*(a+b)`, `a*b+a*a` и т. д.

Возьмем цепочку `a*b+a` и построим для нее левосторонний вывод. Получится два варианта:

$$S \Rightarrow S+S \Rightarrow S*S+S \Rightarrow a*S+S \Rightarrow a*b+S \Rightarrow a*b+a$$

$$S \Rightarrow S*S \Rightarrow a*S \Rightarrow a*S+S \Rightarrow a*b+S \Rightarrow a*b+a$$

Каждому из этих вариантов будет соответствовать свое дерево вывода. Два варианта дерева вывода для цепочки `a*b+a` приведены на рис. 1.4.

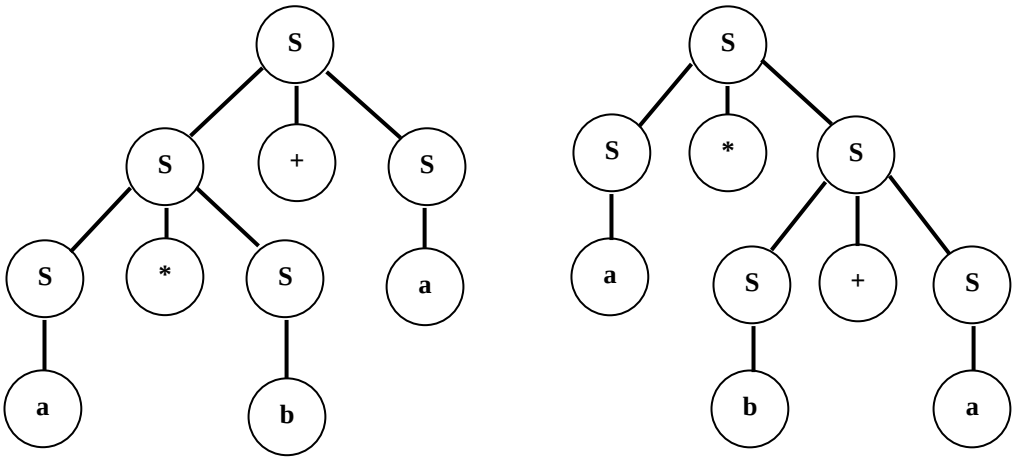


Рис. 1.4. Два варианта дерева цепочки a*b+a вывода для неоднозначной грамматики арифметических выражений

С точки зрения формального языка, заданного грамматикой, не имеет значения, какая цепочка вывода и какое дерево вывода из возможных вариантов будут построены. Однако в реальных языках структура предложения и его значение (смысл) взаимосвязаны. Это справедливо как для естественных языков, так и для языков программирования. Дерево вывода (или цепочка вывода) является формой представления структуры предложения языка. Поэтому для языков программирования, которые несут смысловую нагрузку, имеет принципиальное значение то, какая цепочка вывода будет построена для предложения языка.

Например, если принять во внимание, что рассмотренная здесь грамматика определяет язык арифметических выражений, то с точки зрения семантики арифметических выражений порядок построения дерева вывода соответствует порядку выполнения арифметических действий. В арифметике, как известно, при отсутствии скобок умножение всегда выполняется раньше сложения (умножение имеет более высокий приоритет), но в рассмотренной выше грамматике это ниоткуда не следует — в ней все операции равноправны. Поэтому с точки зрения арифметических операций приведенная грамматика имеет неверную семантику — в ней нет приоритета операций, а кроме того, для равноправных операций не

определен порядок выполнения (в арифметике принят порядок выполнения действий «слева направо»), хотя синтаксическая структура построенных с ее помощью выражений будет правильной.

Такая ситуация называется неоднозначностью в грамматике. Естественно, для построения компиляторов и языков программирования нельзя использовать грамматики, допускающие неоднозначности. Дадим более точное определение неоднозначной грамматики.

Грамматика называется *однозначной*, если для каждой цепочки символов языка, заданного этой грамматикой, можно построить единственный левосторонний (и единственный правосторонний) вывод. Или, что то же самое: грамматика называется однозначной, если для каждой цепочки символов языка, заданного этой грамматикой, существует единственное дерево вывода. В противном случае грамматика называется *неоднозначной*.

Рассмотренная в примере грамматика арифметических выражений, очевидно, является неоднозначной.

Проверка однозначности и эквивалентности грамматик

Поскольку грамматика языка программирования, по сути, всегда должна быть однозначной, то возникают два вопроса, которые необходимо в любом случае решить:

- как проверить, является ли данная грамматика однозначной?
- если заданная грамматика является неоднозначной, то как преобразовать ее к однозначному виду?

Однозначность — это свойство грамматики, а не языка. Для некоторых языков, заданных неоднозначными грамматиками, иногда удается построить эквивалентную однозначную грамматику (однозначную грамматику, задающую тот же язык).

Чтобы убедиться в том, что некоторая грамматика не является однозначной (является неоднозначной), согласно определению, достаточно найти в заданном ею языке хотя бы одну цепочку, которая бы допускала более чем один левосторонний или правосторонний вывод (как это было в рассмотренном примере). Однако не всегда удается легко обнаружить такую цепочку символов. Кроме того, если такая цепочка не найдена, мы не можем утверждать, что данная грамматика является однозначной, поскольку перебрать все цепочки языка невозможно — как правило, их бесконечное количество. Следовательно, нужны другие способы проверки однозначности грамматики.

Если грамматика все же является неоднозначной, то необходимо попытаться преобразовать ее в однозначный вид. Например, для рассмотренной выше неоднозначной грамматики арифметических выражений над операндами a и b существует эквивалентная ей однозначная грамматика следующего вида G'({+,-,*,/,(,),a,b},{S,T,E},**P'**,S):

P':

S → S+T | S−T | T

T → T*E | T/E | E

E → (S) | a | b

В этой грамматике для рассмотренной выше цепочки символов языка a*b+a возможен только один левосторонний вывод:

S ⇒ S+T ⇒ T+T ⇒ T*E+T ⇒ E*E+T ⇒ a*E+T ⇒ a*b+T ⇒ a*b+E ⇒ a*b+a

Этому выводу соответствует единственно возможное дерево вывода. Оно приведено на рис. 1.5. Видно, что, хотя цепочка вывода несколько удлинилась, но приоритет операций в данном случае единственно возможный и соответствует их порядку в арифметике.

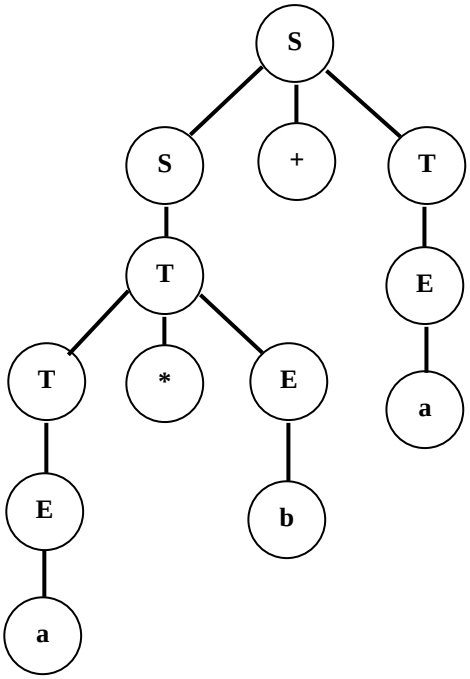


Рис. 1.5. Дерево вывода для однозначной грамматики арифметических выражений

В таком случае необходимо решить две проблемы: во-первых, доказать что две имеющиеся грамматики эквивалентны (задают один и тот же язык); во-вторых, иметь возможность проверить, что вновь построенная грамматика является однозначной.

Проблема эквивалентности грамматик в общем виде формулируется следующим образом: имеются две грамматики **G** и **G**', необходимо построить алгоритм, который бы позволял проверить, являются ли эти две грамматики эквивалентными. То есть, надо проверить утверждение L(**G**) = L(**G**').

К сожалению, доказано, что проблема эквивалентности грамматик в общем случае алгоритмически неразрешима. Это значит, что не только до сих пор не существует алгоритма, который бы позволял проверить, являются ли две заданные грамматики

эквивалентными, но и доказано, что такой алгоритм в принципе не существует, а значит, он никогда не будет создан.

Точно так же неразрешима в общем виде и проблема однозначности грамматик. Это значит, что не существует (и никогда не будет существовать) алгоритм, который бы позволял для произвольной заданной грамматики **G** проверить, является ли она однозначной или нет. Аналогично, не существует алгоритма, который бы позволял преобразовать заведомо неоднозначную грамматику **G** в эквивалентную ей однозначную грамматику **G'**.

В общем случае вопрос об алгоритмической неразрешимости проблем однозначности и эквивалентности грамматик сводится к вопросу об алгоритмической неразрешимости проблемы, известной как «проблема соответствий Поста» [4 т.1, 5].

Неразрешимость проблем эквивалентности и однозначности грамматик в общем случае совсем не означает, что они не разрешимы вообще. Для многих частных случаев — например, для определенных типов и классов грамматик (в частности, для регулярных грамматик) — эти проблемы решены. Например, приведенная выше грамматика **G'** для арифметических выражений над операндами a и b относится к классу грамматик операторного предшествования из типа КС-грамматик, который будет рассмотрен далее. На основе этой грамматики возможно построить распознаватель в виде детерминированного расширенного МП-автомата, а потому можно утверждать, что она является однозначной (см. раздел «Восходящие распознаватели КС-языков без возвратов»).

Правила, задающие неоднозначность в грамматиках

В общем виде невозможно проверить, является ли заданная грамматика однозначной или нет. Однако для КС-грамматик существуют определенного вида правила, по наличию которых во всем множестве правил грамматики G(**VT**,**VN**,**P**,S) можно утверждать, что она является неоднозначной. Эти правила имеют следующий вид:

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 $A \in \mathbf{VN}$; $\alpha,\beta,\gamma \in (\mathbf{VN} \cup \mathbf{VT})^*$.

Если в заданной грамматике встречается хотя бы одно правило подобного вида (любого из приведенных вариантов), то доказано, что такая грамматика точно будет неоднозначной. Однако если подобных правил во всем множестве правил грамматики нет, это совсем не означает, что грамматика является однозначной. Такая грамматика может быть однозначной, а может и не быть. То есть отсутствие правил указанного вида (всех вариантов) — это необходимое, но не достаточное условие однозначности грамматики.

С другой стороны, установлены условия, при удовлетворении которым грамматика заведомо является однозначной. Они справедливы для всех регулярных и многих

классов КС-грамматик. Однако известно, что эти условия, напротив, являются достаточными, но не необходимыми для однозначности грамматик.

Например, в рассмотренном выше примере грамматики арифметических выражений с операндами a и b — `G({+,-,*,/,(,),a,b},{S},P,S)` — во множестве правил **P:** `S → S+S | S-S | S*S | S/S | (S) | a | b` встречаются правила 2 типа (например, два правила `S → S+S` и `S → a`). Поэтому данная грамматика является неоднозначной, что и было показано выше.

Контрольные вопросы и задачи

Вопросы

1. Какие операции можно выполнять над цепочками символов?
2. Какие из перечисленных ниже тождеств являются истинными для двух произвольных цепочек символов α и β, а какие нет:

   $|\alpha\beta| = |\alpha| + |\beta| = |\beta\alpha|$

   $\alpha\beta = \beta\alpha$

   $|\alpha^R| = |\alpha|$

   $(\alpha^2\beta^2)^R = (\beta^R\alpha^R)^2$

   $(\alpha^2\beta^2)^R = (\beta^R)^2(\alpha^R)^2$
3. Какие существуют методы задания языков? Почему метод перечисления всех допустимых цепочек языка не находит практического применения?
4. Какие дополнительные вопросы необходимо решить при задании языка программирования? Какие из них могут быть решены в рамках теории формальных языков?
5. Кто (или что) для любого языка программирования выступает в роли генератора цепочек языка? Кто (или что) выступает в роли распознавателя цепочек?
6. Как формулируется задача разбора? Всегда ли она разрешима?
7. Что такое грамматика языка? Дайте определения грамматики.
8. Как выглядит описание грамматики в форме Бэкуса-Наура? Какие еще формы описания грамматик существуют?
9. Почему в форме Бэкуса-Наура практически невозможно построить грамматику для реального языка так, чтобы она не содержала рекурсивных правил?
10. Что такое распознаватель?
11. Как классифицируются распознаватели? Как их классификация соотносится с классификацией языков и грамматик?
12. На основе какого принципа классифицируются грамматики в классификации Хомского?

13. Какие типы грамматик выделяют по классификации Хомского? Как они между собой соотносятся?
14. Какие типы языков выделяют по классификации Хомского? Как классификация языков соотносится с классификацией грамматик?
15. Что такое сентенциальная форма грамматики?
16. Что такое левосторонний и правосторонний выводы? Можно ли построить еще какие-нибудь варианты цепочек вывода?

Задачи

1. Ниже даны различные варианты грамматик, определяющие язык десятичных чисел с фиксированной точкой. Укажите, к какому типу относится каждая из этих грамматик. К какому типу относится сам язык десятичных чисел с фиксированной точкой?

G_1({".",+,-
,0,1,2,3,4,5,6,7,8,9},{<число>,<цел>,<дроб>,<цифра>,<осн>,
<знак>},$\mathbf{P}_1$,<число>)

$\mathbf{P}_1$: <число> → <знак><осн>

<знак> → λ | + | -

<осн> → <цел>.<дроб> | <цел>

<цел> → <цифра> | <цифра><цифра>

<дроб> → λ | <цел>

<цифра><цифра> → <цифра><цифра><цифра>

<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

G_2({".",+,-
,0,1,2,3,4,5,6,7,8,9},{<число>,<часть>,<цифра>,<осн>},$\mathbf{P}_2$,<
число>)

$\mathbf{P}_2$: <число> → +<осн> | -<осн> | <осн>

<осн> → <часть>.<часть> | <часть>. | <часть>

<часть> → <цифра> | <цифра><цифра>

<цифра><цифра> → <цифра><цифра><цифра>

<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

G_3({".",+,-
,0,1,2,3,4,5,6,7,8,9},{<число>,<часть>,<осн>},$\mathbf{P}_3$,<число>)

$\mathbf{P}_3$: <число> → +<осн> | -<осн> | <осн>

<осн> → <часть>. | <часть> | <осн>0 | <осн>1 | <осн>2 | <осн>3 | <осн>4 | <осн>5 | <осн>6 | <осн>7 | <осн>8 | <осн>9

<часть> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | <часть>0 | <часть>1 | <часть>2 | <часть>3 | <часть>4 | <часть>5 | <часть>6 | <часть>7 | <часть>8 | <часть>9

G_4({".",+,-,0,1,2,3,4,5,6,7,8,9},{<знак>,<число>,<часть>,<осн>},$\mathbf{P}_4$,<число>)

$\mathbf{P}_4$: <число> → <часть> | <часть>. | <число>0 | <число>1 | <число>2 | <число>3 | <число>4 | <число>5 | <число>6 | <число>7 | <число>8 | <число>9

<часть> → <знак>0 | <знак>1 | <знак>2 | <знак>3 | <знак>4 | <знак>5 | <знак>6 | <знак>7 | <знак>8 | <знак>9 | <часть>0 | <часть>1 | <часть>2 | <часть>3 | <часть>4 | <часть>5 | <часть>6 | <часть>7 | <часть>8 | <часть>9

<знак> → λ | + | -

G_5({".",+,-,0,1,2,3,4,5,6,7,8,9},{<знак>,<число>,<часть>,<осн>},$\mathbf{P}_5$,<число>)

$\mathbf{P}_5$: <число> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | <часть>. | <осн>0 | <осн>1 | <осн>2 | <осн>3 | <осн>4 | <осн>5 | <осн>6 | <осн>7 | <осн>8 | <осн>9 | <часть>0 | <часть>1 | <часть>2 | <часть>3 | <часть>4 | <часть>5 | <часть>6 | <часть>7 | <часть>8 | <часть>9

<осн> → <часть>. | <осн>0 | <осн>1 | <осн>2 | <осн>3 | <осн>4 | <осн>5 | <осн>6 | <осн>7 | <осн>8 | <осн>9

<часть> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | <знак>0 | <знак>1 | <знак>2 | <знак>3 | <знак>4 | <знак>5 | <знак>6 | <знак>7 | <знак>8 | <знак>9 | <часть>0 | <часть>1 | <часть>2 | <часть>3 | <часть>4 | <часть>5 | <часть>6 | <часть>7 | <часть>8 | <часть>9

<знак> → + | -

2. Язык десятичных чисел с фиксированной точкой, рассмотренный в задаче №1, содержит цепочки `-83`, `239`, `10.4`. Постройте цепочки вывода для каждой из этих трех цепочек символов на основе предложенных в задаче №1 грамматик.

3. Язык десятичных чисел с фиксированной точкой, рассмотренный выше в задаче №1, не содержит цепочек вида `.29` или `.104`. Перестройте любую из предложенных в задаче №1 пяти грамматик так, чтобы заданный ею новый язык допускал такого рода цепочки. Постройте цепочки вывода для этих цепочек.

4. Что можно сказать о типе языка, если:

- язык задан регулярной грамматикой и КС-грамматикой;
- язык задан КС-грамматикой и КЗ-грамматикой;
- язык задан КЗ-грамматикой и регулярной грамматикой.

Является ли в каждом случае ответ окончательным?

5. Поездом называется произвольная последовательность локомотивов и вагонов, начинающаяся с локомотива. Постройте грамматику для понятия <поезд> в форме Бэкуса-Наура, считая что понятия <локомотив> и <вагон> являются терминальными символами. Модернизируйте грамматику для любого из следующих условий:
   - все локомотивы должны быть сосредоточены в начале поезда;
   - поезд начинается с локомотива и заканчивается локомотивом (попробуйте построить регулярную грамматику);
   - поезд не должен содержать два локомотива, либо два вагона подряд.
6. Насколько сложно построить в форме Бэкуса-Наура определение поезда (согласно задаче номер №5), содержащего не более 60 вагонов или локомотивов? Постройте такое определение в форме грамматики с метасимволами. Распространите его на одно из предложенных в задаче №5 дополнительных условий.
7. Постройте вывод для цепочки aaaabbbbcccc в грамматике G({a,b,c},{B,C,D,S},**P**,S) с правилами **P**:

   S → abc | AE

   A → aABC | aBC

   CBC → CDC

   CDC → BDC

   BDC → BCC

   CCE → CFE

   CFE → CFc

   CFc → CEc

   aB → ab

   bB → bb

   bCE → bCc

   bCc → bc

8. Дана грамматика условных выражений G({if,then,else,b,a},{E},**P**,E) с правилами: **P**: E → if b then E else E | if b then E | a

не строя цепочек вывода, покажите, что эта грамматика является неоднозначной. Проверьте это, построив некоторую цепочку вывода. Попробуйте построить эквивалентную ей однозначную грамматику.