Лабораторная работа №1. Формальные языки, грамматики и их свойства

Теоретические сведения

Тексты на любом языке (естественном или формальном) представляют собой цепочки символов некоторого алфавита.

> **Цепочкой символов** называют произвольную упорядоченную конечную последовательность символов, записанных один за другим.

Понятие **символа** является базовым в теории формальных языков. Для цепочки символов имеют значение три фактора: состав входящих в цепочку символов, их количество и порядок символов в цепочке. Далее цепочки символов будем обозначать греческими буквами: α, β, γ и др.

Цепочки символов α и β **равны** (α = β), 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 определяет её **длину**. Длина цепочки α обозначается как | α |.

Можно выделить следующие операции над цепочками символов:

- ✓ *конкатенация* (объединение, сложение двух цепочек) – это дописывание второй цепочки в конец первой. Конкатенация цепочек α и β обозначается как αβ. Например: α = *аб*, β = *вг*, тогда αβ = *абвг*. При этом αβ ≠ βα, так как в цепочке важен порядок символов. Но конкатенация обладает свойством ассоциативности: (αβ)γ = α(βγ);
- ✓ *замена* (подстановка) – замена подцепочки символов на любую произвольную цепочку символов. В результате получается новая цепочка символов. Например: γ = *абвг*, разобьём эту цепочку символов на подцепочки: α = *а*, ω = *б*, β = *вг* и выполним подстановку цепочки υ = *аба* вместо подцепочки ω. Получим новую цепочку γ' = *ааабавг*. Таким образом,

подстановка выполняется путём разбиения исходной цепочки на подцепочки и конкатенации;

- ✓ *обращение* – запись символов цепочки в обратном порядке. Эта операция обозначается как α^R. Если α = *абвг*, то α^R = *гвба*. Для операции обращения справедливо следующее: $(\alpha\beta)^R = \beta^R\alpha^R$;
- ✓ *итерация* – повторение цепочки *n* раз, где *n* > 0 – это конкатенация цепочки с собой *n* раз, обозначается как α^n. Если *n* = 0, то результатом итерации будет пустая цепочка символов.

> **Пустая цепочка символов** – это цепочка, не содержащая ни одного символа. Будем обозначать такую цепочку как ε.

Для пустой цепочки справедливы следующие равенства:

$|\varepsilon| = 0$

$\varepsilon\alpha = \alpha\varepsilon = \alpha$

$\varepsilon^R = \varepsilon$

$\varepsilon^n = \varepsilon, n \geq 0$

$\alpha^0 = \varepsilon$

Основой любого языка является алфавит, определяющий набор допустимых символов языка.

> **Алфавит** – это счётное множество допустимых символов языка. Будем обозначать это множество как **V**.

Цепочка символов α является **цепочкой над алфавитом V**: α(**V**), если в неё входят только символы, принадлежащие множеству символов **V**. Для любого алфавита **V** пустая цепочка может как являться, так и не являться цепочкой над алфавитом.

Если **V** – некоторый алфавит, то:

- ✓ $\mathbf{V}^+$ – множество всех цепочек над алфавитом V без пустой цепочки;
- ✓ $\mathbf{V}^*$ – множество всех цепочек над алфавитом V, включая пустую цепочку.

> **Языком L** над алфавитом **V** (**L (V)**) называется некоторое счётное подмножество цепочек конечной длины из множества всех цепочек над алфавитом **V**.

Множество цепочек языка не обязано быть конечным; хотя каждая цепочка символов, входящая в язык, обязана иметь конечную длину, эта длина может быть сколь угодно большой и формально ничем не ограничена.

Цепочку символов, принадлежащую заданному языку, часто называют **предложением** языка, а множество цепочек символов некоторого языка **L (V)** – множеством предложений этого языка.

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:

- ✓ перечислением всех допустимых цепочек языка;

- ✓ указанием способа порождения цепочек языка (заданием грамматики языка);
- ✓ 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. 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}) = $\{0^n 1^n, n > 0\}$ задаёт язык над алфавитом **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.

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.

Третий способ предусматривает построение некоторого логического устройства (распознавателя) – автомата, который на входе получает цепочку символов, а на выходе выдаёт ответ, принадлежит или нет эта цепочка заданному языку.

> **Грамматика** – это описание способа построения предложений некоторого языка.

Она относится ко второму способу определения языков – порождению цепочек символов. Грамматику языка можно описать различными способами. Например, 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 α → β (или α ::= β). 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 даётся описание семантических правил.

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 (G)** ∪ {ε} = **L (G')** ∪ {ε}.

Формально грамматика **G** определяется как четвёрка **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 α → β, где α ∈ (**VN** ∪ **VT**)⁺, β ∈ (**VN** ∪ **VT**)*;
- ✓ *S* – целевой (начальный) символ грамматики, *S* ∈ **VN**.

Алфавиты терминальных и нетерминальных символов грамматики не пересекаются: **VN** ∩ **VT** = ∅. Это значит, что каждый символ в грамматике может быть либо терминальным,

либо нетерминальным, но не может быть терминальным и нетерминальным одновременно. Целевой символ грамматики – это всегда нетерминальный символ. Множество **V** = **VN** ∪ **VT** называют **полным алфавитом** грамматики **G**.

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.

Во множестве правил грамматики может быть несколько правил, имеющих одинаковые левые части вида: $\alpha \to \beta_1$, $\alpha \to \beta_2$, …, $\alpha \to \beta_n$. Эти правила можно объединить вместе и записать 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** [1]. Форма Бэкуса-Наура (англ. Backus-Naur Form (BNF)), как правило, предусматривает также, что нетерминальные символы берутся в угловые скобки: < >.

Пример грамматики, которая определяет язык целых десятичных чисел со знаком в форме Бэкуса-Наура:

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 различные левые части правил);
- ✓ целевым символом грамматики является символ <*число*>.

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:
$S \to T \mid +T \mid -T$
$T \to F \mid TF$
$F \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

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

Особенность рассмотренных выше формальных грамматик в том, что они позволяют определить бесконечное множество цепочек языка с помощью конечного набора правил.

---

[1] Джон Бэкус (John Backus, 1924 – 2007) – американский учёный в области информатики, был руководителем команды, разработавшей высокоуровневый язык программирования ФОРТРАН, разработчиком одной из самых универсальных нотаций, используемых для определения синтаксиса формальных языков.

Питер Наур (Peter Naur; 1928) – датский учёный в области информатики, известен как один из разработчиков первого языка структурного программирования Алгол-60 и, совместно с Бэкусом, как изобретатель формы Бэкуса-Наура.

Возможность пользоваться конечным набором правил достигается в такой форме записи за счёт **рекурсивных правил**. Рекурсия в правилах грамматики выражается в том, что один из нетерминальных символов определяется сам через себя. Рекурсия может быть *непосредственной* (явной) – символ определяется сам через себя в одном правиле, либо *косвенной* (неявной) – тогда тоже 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 $T \to TF$.

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. Такими правилами являются <*чс*> → <*цифра*> – в грамматике **G** и $T \to F$ – в грамматике **G'**.

Более упрощённой по сравнению с БНФ является **расширенная форма Бэкуса-Наура** (англ. Extended Backus-Naur Form (EBNF)). Она отличается более «ёмкими» конструкциями, позволяющими при той же выразительной способности упростить и сократить описание в объёме. Набор возможных конструкций РБНФ невелик. Это конкатенация, выбор, условное вхождение и повторение.

Конкатенация не имеет специального обозначения, определяется последовательной записью символов в выражении. *Выбор* обозначается вертикальной чертой (|), как и в БНФ. Квадратные скобки ([ ]) выделяют необязательный элемент выражения, который может присутствовать, а может и отсутствовать (*условное вхождение*). Фигурные скобки ({}) обозначают конкатенацию любого числа (включая нуль) записанных в них элементов (*повторение*). Помимо основных операций в РБНФ могут использоваться обычные круглые скобки (( )). Они применяются для группировки элементов при формировании сложных выражений.

Следующая грамматика определяет запись десятичного числа общего вида (с ведущим знаком, возможной дробной частью и порядком) в расширенной форме Бэкуса-Наура:

Число → [+ | –] *НатЧисло*[. [*НатЧисло*]][(*e* | *E*)[+ | –] *НатЧисло*]
НатЧисло → *Цифра*{*Цифра*}
Цифра → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

В дальнейших примерах для описания грамматик будет использоваться обычная форма Бэкуса-Наура.

Классификация грамматик и языков

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 определённому 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*), $\mathbf{V} = \mathbf{VN} \cup \mathbf{VT}$, правила имеют вид $\alpha \to \beta$, где $\alpha \in \mathbf{V}^+$, $\beta \in \mathbf{V}^*$.

Это самый общий тип грамматик. В него попадают все без исключения формальные грамматики, но часть из них может быть также отнесена и к другим классификационным типам.

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 $\alpha_1 A \alpha_2 \rightarrow \alpha_1 \beta \alpha_2$, 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: $\alpha \rightarrow \beta$, 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 заменён на ту или иную цепочку символов в зависимости от того контекста, в котором он встречается. Именно поэтому эти грамматики называют *контекстно-зависимыми*. 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.

Тип 2: контекстно-свободные (КС) грамматики

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

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

Эти два класса составляют тип контекстно-свободных грамматик. Разница между ними заключается лишь в том, что в УКС-грамматиках в правой части правил может присутствовать пустая цепочка, а в НКС-грамматиках – нет. Отсюда ясно, что язык, заданный НКС-грамматикой, не может содержать пустой цепочки.

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: $A \rightarrow B\gamma$ или $A \rightarrow \gamma$, 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 $A, B \in$ **VN**, $\gamma \in$ **VT***.

Языки классифицируются в соответствии с типами грамматик, с помощью которых они заданы. Причём, поскольку один и тот же язык в общем случае может быть задан сколь угодно большим количеством грамматик, которые могут относиться к различным классификационным типам, 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), сам язык должен быть отнесён к типу 3 и является регулярным языком.

Например, грамматика типа 0 $\mathbf{G_1}$ ({0, 1}, {*A*, *S*}, $\mathbf{P_1}$, *S*) и КС-грамматика $\mathbf{G_2}$ ({0, 1}, {*S*}, $\mathbf{P_2}$, *S*), где

$\mathbf{P_1}$: $S \to 0A1$
$0A \to 00A1$
$A \to \varepsilon$

$\mathbf{P_2}$: $S \to 0S1 \mid 01$

описывают один и тот же язык $\mathbf{L} = \mathbf{L}(\mathbf{G_1}) = \mathbf{L}(\mathbf{G_2}) = \{0^n1^n \mid n > 0\}$. Язык **L** называют КС-языком, т.к. существует КС-грамматика, его описывающая. Но он не является регулярным языком, т.к. не существует регулярной грамматики, описывающей этот язык.

Цепочки вывода

> **Выводом** называется процесс порождения предложения языка на основе правил определяющей язык грамматики.

Цепочка $\beta = \delta_1\gamma\delta_2$ называется **непосредственно выводимой** из цепочки $\alpha = \delta_1\omega\delta_2$ в грамматике **G** (**VT**, **VN**, **P**, *S*), **V** = **VN** ∪ **VT**, $\delta_1, \gamma, \delta_2 \in$ **V***, $\omega \in$ **V**$^+$, если в грамматике существует правило $\omega \to \gamma$. 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. Непосредственная выводимость цепочки β из цепочки α обозначается как: $\alpha \Rightarrow \beta$.

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

- ✓ β непосредственно выводима из α ($\alpha \Rightarrow \beta$);
- ✓ существует γ такая, что γ выводима из α, и β непосредственно выводима из γ ($\alpha \Rightarrow \gamma, \gamma \Rightarrow \beta$).

Пример 1. Грамматика для языка целых десятичных чисел со знаком:

G ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, +}, {*S*, *T*, *F*}, **P**, *S*)
P:
$S \to T \mid +T \mid -T$
$T \to F \mid TF$
$F \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Построим несколько цепочек вывода в этой грамматике (для понимания каждого шага вывода подцепочка, для которой выполняется подстановка, выделена жирным шрифтом и цветом):

$$\boldsymbol{S} \Rightarrow -\boldsymbol{T} \Rightarrow -\boldsymbol{T}F \Rightarrow -\boldsymbol{T}FF \Rightarrow -\boldsymbol{F}FF \Rightarrow -4\boldsymbol{F}F \Rightarrow -47\boldsymbol{F} \Rightarrow -479$$

S ⇒ ***T*** ⇒ *T****F*** ⇒ ***T***8 ⇒ ***F***8 ⇒ 18
T ⇒ *T****F*** ⇒ ***T***0 ⇒ *T****F***0 ⇒ ***T***50 ⇒ ***F***50 ⇒ 350
F ⇒ 5

Получаем, что: $S \Rightarrow^* -479$, $S \Rightarrow^* 18$, $T \Rightarrow^* 350$, $F \Rightarrow^* 5$.

Пример 2. Задана грамматика **G** ({*a, b, c*}, {*B, C, D, S*}, **P**, *S*) с правилами:
P:
$S \to BD$
$B \to aBbC \mid ab$
$Cb \to bC$
$CD \to Dc$
$bDc \to bcc$
$abD \to abc$

Эта грамматика задаёт язык **L** (**G**) = $\{a^n b^n c^n \mid n > 0\}$. Пример вывода предложения *aaaabbbbcccc* для этого языка на основе грамматики **G** выглядит следующим образом (для понимания каждого шага вывода подцепочка, для которой выполняется подстановка, выделена жирным шрифтом и цветом):

S ⇒ ***B***D ⇒ a***B***bCD ⇒ aa***B***bCbCD ⇒ aaa***B***bCbCbCD ⇒ aaaabb***Cb***CbCD ⇒ aaaabbbC***Cb***CD ⇒ aaaabbb***Cb***CCD ⇒ aaaabbbbCC***CD*** ⇒ aaaabbbbC***CD***c ⇒ aaaabbbb***CD***cc ⇒ aaaabbb***bD***ccc ⇒ aaaabbbbcccc

Таким образом, цепочка *aaaabbbbcccc* выводима из *S*: $S \Rightarrow^*$ *aaaabbbbcccc*.

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. Цепочка β, полученная в результате законченного вывода, называется **конечной** цепочкой вывода.

В рассмотренном выше примере 1 все построенные выводы являются законченными, вывод $S \Rightarrow^* -4FF$ (из первой цепочки в примере) будет незаконченным.

Сентенциальная форма грамматики

Цепочка символов $\alpha \in \mathbf{V}^*$ называется **сентенциальной формой** грамматики **G** (**VT**, **VN**, **P**, *S*), $\mathbf{V} = \mathbf{VT} \cup \mathbf{VN}$, если она выводима из целевого символа грамматики *S*: $S \Rightarrow^* \alpha$. 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 *F*8 из вывода 2, например, тоже является сентенциальной формой, поскольку справедливо $S \Rightarrow^* F8$, но она не является конечной сентенциальной формой вывода. В то же время в выводах 3 и 4 примера 1 явно не присутствуют сентенциальные формы.

Левосторонний и правосторонний выводы

Вывод называется **левосторонним**, если в нём на каждом шаге вывода правило грамматики применяется всегда к крайнему левому нетерминальному символу в цепочке. Другими словами, 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 нём на каждом шаге вывода правило грамматики применяется всегда к крайнему правому нетерминальному символу в цепочке.

Если рассмотреть цепочки вывода из примера 1, то в нём выводы 1 и 4 являются левосторонними, выводы 2, 3 и 4 – правосторонними (вывод 4 является одновременно и правосторонним, и левосторонним).

Для грамматик типов 2 и 3 (КС-грамматик и регулярных грамматик) для любой сентенциальной формы всегда можно построить левосторонний или правосторонний вывод. 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.

Рассмотренный в примере 2 вывод $S \Rightarrow^* aaaabbbbcccc$ для грамматики **G**, задающей язык $\mathbf{L(G)} = \{a^n b^n c^n \mid n > 0\}$, не является ни левосторонним, ни правосторонним. Грамматика относится к типу 1, и в данном случае для неё нельзя построить такой вывод, на каждом шаге которого только один нетерминальный символ заменялся бы на цепочку символов.

В грамматике для одной и той же цепочки может быть несколько выводов, эквивалентных в том смысле, что в них в одних и тех же местах применяются одни и те же правила вывода, но в различном порядке.

Например, для цепочки $a+b+a$ в грамматике

G ({a, b, +}, {S, T}, **P**, S)
P:
$S \to T \mid T+S$
$T \to a \mid b$

можно построить выводы:

$$\mathbf{S} \Rightarrow T+\mathbf{S} \Rightarrow T+T+\mathbf{S} \Rightarrow \mathbf{T}+T+T \Rightarrow a+\mathbf{T}+T \Rightarrow a+b+\mathbf{T} \Rightarrow a+b+a$$
$$\mathbf{S} \Rightarrow \mathbf{T}+S \Rightarrow a+\mathbf{S} \Rightarrow a+\mathbf{T}+S \Rightarrow a+b+\mathbf{S} \Rightarrow a+b+\mathbf{T} \Rightarrow a+b+a$$
$$\mathbf{S} \Rightarrow T+\mathbf{S} \Rightarrow T+T+\mathbf{S} \Rightarrow T+T+\mathbf{T} \Rightarrow T+\mathbf{T}+a \Rightarrow \mathbf{T}+b+a \Rightarrow a+b+a$$

Во втором случае применяется левосторонний вывод, в третьем – правосторонний, а первый вывод не является ни левосторонним, ни правосторонним, но все эти выводы являются эквивалентными в указанном выше смысле.

Дерево вывода

Можно ввести удобное графическое представление вывода, называемое деревом вывода, причём для всех эквивалентных выводов деревья вывода совпадают.

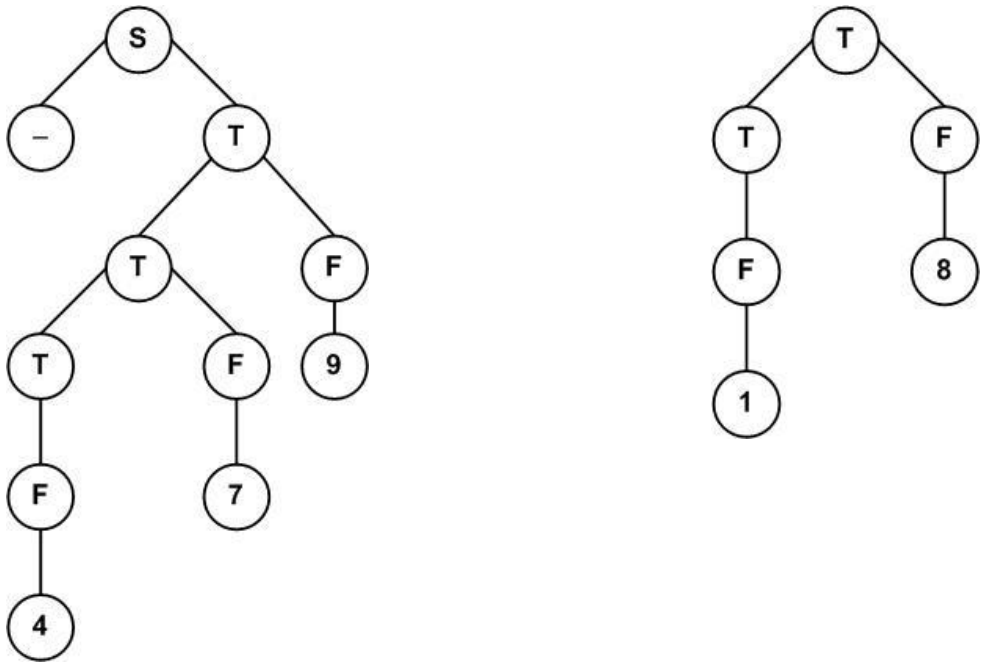
Деревом вывода грамматики **G** (**VT**, **VN**, **P**, S) называется дерево, которое соответствует некоторой цепочке вывода и удовлетворяет следующим условиям:

- ✓ каждая вершина дерева обозначается символом грамматики $A \in (\mathbf{VT} \cup \mathbf{VN} \cup \{\varepsilon\})$;
- ✓ корнем дерева является вершина, обозначенная целевым символом грамматики – S;

- ✓ листьями дерева (концевыми вершинами) являются вершины, обозначенные терминальными символами грамматики или символом пустой цепочки ε;
- ✓ если некоторый узел дерева обозначен нетерминальным символом $A \in$ **VN**, а связанные с ним узлы – символами $b_1, b_2, \ldots, b_n$; $n > 0$, $\forall i$, $0 \le i \le n$: $b_i \in$ (**VT** $\cup$ **VN** $\cup$ {ε}), то в грамматике **G** (**VT**, **VN**, **P**, S) существует правило $A \to b_1 \mid b_2 \mid \ldots \mid b_n \in$ **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.

На основе примера 1 из раздела «Цепочки вывода»Цепочки вывода построим деревья вывода для цепочек выводов 1 и 2. Эти деревья приведены на рисунке 1 (*а* – для вывода 1, *б* – для вывода 2).



а) *б*)

Рисунок 1. 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 определённым 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 идёт 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.

Если для каждой цепочки символов языка, заданного грамматикой, можно построить единственный левосторонний (и единственный правосторонний) вывод или, что то же самое, построить единственное дерево вывода, то такая грамматика называется **однозначной**. Иначе грамматика называется **неоднозначной.**

Пример 3. Условный оператор, включённый во многие языки программирования, описывается с помощью грамматики с правилами:

$S \to$ ***if*** b ***then*** S ***else*** S | ***if*** b ***then*** S | a

где b – логическое выражение, а a – безусловный оператор. Эта грамматика неоднозначна, т.к. в ней возможны два левых вывода цепочки ***if*** b ***then if*** b ***then*** a ***else*** a:

1) $\boldsymbol{S} \Rightarrow$ ***if*** b ***then S else*** $S \Rightarrow$ ***if*** b ***then if*** b ***then S else*** $S \Rightarrow$ ***if*** b ***then if*** b ***then*** a ***else S*** $\Rightarrow$ ***if*** b ***then if*** b ***then*** a ***else*** a

2) $\boldsymbol{S} \Rightarrow$ ***if*** b ***then S*** $\Rightarrow$ ***if*** b ***then if*** b ***then S else*** $S \Rightarrow$ ***if*** b ***then if*** b ***then*** a ***else S*** $\Rightarrow$ ***if*** b ***then if*** b ***then*** a ***else*** a

Наличие двух различных выводов предполагает две различные интерпретации цепочки ***if*** b ***then if*** b ***then*** a ***else*** a: первая из них – ***if*** b ***then*** *(****if*** b ***then*** a*)* ***else*** a, а вторая – ***if*** b ***then*** *(****if*** b ***then*** a ***else*** a*)*. При описании языка программирования эту неоднозначность преодолевают, добавляя к семантическим правилам правило вида: «ключевое слово ***else*** ассоциируется с ближайшим слева ключевым словом ***if***».

Для КС-грамматик существуют определённого вида правила, по наличию которых во всём множестве правил грамматики можно утверждать, что она является неоднозначной:

$A \to AA \mid \alpha$
$A \to A\alpha A \mid \beta$
$A \to \alpha A \mid A\beta \mid \gamma$
$A \to \alpha A \mid \alpha A \beta A \mid \gamma$

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, то такая грамматика точно будет неоднозначной. Однако отсутствие подобных правил во всём множестве правил грамматики не означает, что данная грамматика является однозначной.

Для грамматики из примера 3 можно показать, что она является неоднозначной, поскольку она содержит правила четвёртого вида.

Регулярные грамматики и конечные автоматы

Практически во всех трансляторах (и в компиляторах, и в интерпретаторах) в том или ином виде присутствует фаза лексического анализа. В основе лексических анализаторов лежат

регулярные грамматики, поэтому рассмотрим грамматики этого класса более подробно. В дальнейшем под регулярной грамматикой будем понимать леволинейную грамматику.

Для грамматик этого типа существует алгоритм определения того, принадлежит ли анализируемая цепочка языку, порождаемому этой грамматикой (*алгоритм разбора*):

- ✓ первый символ исходной цепочки $a_1a_2...a_n\perp$ заменяем нетерминалом A, для которого в грамматике есть правило вывода $A \to a_1$ (другими словами, производим «свёртку» терминала a_1 к нетерминалу A);
- ✓ затем многократно (до тех пор, пока не считаем признак конца цепочки) выполняем следующие шаги: полученный на предыдущем шаге нетерминал A и расположенный непосредственно справа от него очередной терминал a_i исходной цепочки заменяем нетерминалом B, для которого в грамматике есть правило вывода $B \to Aa_i$, $i = 2, 3, ..., n$.

Это эквивалентно построению дерева разбора восходящим методом: на каждом шаге алгоритма строим один из уровней в дереве разбора, поднимаясь от листьев к корню.

При работе этого алгоритма возможны следующие ситуации:

- ✓ прочитана вся цепочка; на каждом шаге находилась единственная нужная «свёртка»; на последнем шаге «свёртка» произошла к символу S. Это означает, что исходная цепочка $a_1a_2...a_n\perp \in$ **L (G)**;
- ✓ прочитана вся цепочка; на каждом шаге находилась единственная нужная «свёртка»; на последнем шаге «свёртка» произошла к символу, отличному от S. Это означает, что исходная цепочка $a_1a_2...a_n\perp \notin$ **L (G)**;
- ✓ на некотором шаге не нашлось нужной «свёртки», т.е. для полученного на предыдущем шаге нетерминала A и расположенного непосредственно справа от него очередного терминала a_i исходной цепочки не нашлось нетерминала B, для которого в грамматике было бы правило вывода $B \to Aa_i$. Это означает, что исходная цепочка $a_1a_2...a_n\perp \notin$ **L (G)**;
- ✓ на некотором шаге работы алгоритма оказалось, что есть более одной подходящей «свёртки», т.е. в грамматике разные нетерминалы имеют правила вывода с одинаковыми правыми частями, и поэтому непонятно, к какому из них производить «свёртку». Это говорит о *недетерминированности разбора.* Анализ этой ситуации будет дан ниже.

Допустим, что разбор на каждом шаге детерминированный. Для того чтобы быстрее находить правило с подходящей правой частью, зафиксируем все возможные «свёртки». Это можно сделать в виде таблицы, строки которой помечены нетерминальными символами грамматики, столбцы – терминальными. Значение каждого элемента таблицы – это нетерминальный символ, к которому можно свернуть пару «нетерминал-терминал», которыми помечены соответствующие строка и столбец.

Пример 4. Для грамматики **G** ({a, b, $\perp$}, {S, A, B, C}, **P**, S) такая таблица будет выглядеть следующим образом:

P: $S \to C\perp$
$C \to Ab \mid Ba$
$A \to a \mid Ca$
$B \to b \mid Cb$

	a	***b***	$\perp$
C	A	B	S
A	–	C	–
B	C	–	–
S	–	–	–

Знак «–» ставится в том случае, если для пары «терминал-нетерминал» «свёртки» нет.

Но чаще информацию о возможных «свёртках» представляют в виде диаграммы состояний (ДС) – неупорядоченного ориентированного помеченного графа, который строится следующим образом:

1) строятся вершины графа, помеченные нетерминалами грамматики (для каждого нетерминала – одну вершину), и ещё одну вершину, помеченную символом, отличным от

нетерминальных (например, H). Эти вершины называют *состояниями*. H – *начальное состояние*;

2) соединяем эти состояния дугами по следующим правилам:
   а) для каждого правила грамматики вида $W \to t$ ($W \in \mathbf{VN}^+$, $t \in \mathbf{VT}^+$) соединяем дугой состояния H и W (от H к W) и помечаем дугу символом t;
   б) для каждого правила $W \to Vt$ ($W, V \in \mathbf{VN}^+$, $t \in \mathbf{VT}^+$) соединяем дугой состояния V и W (от V к W) и помечаем дугу символом t.

Диаграмма состояний для грамматики **G** из примера 4 изображена на рисунке 2.

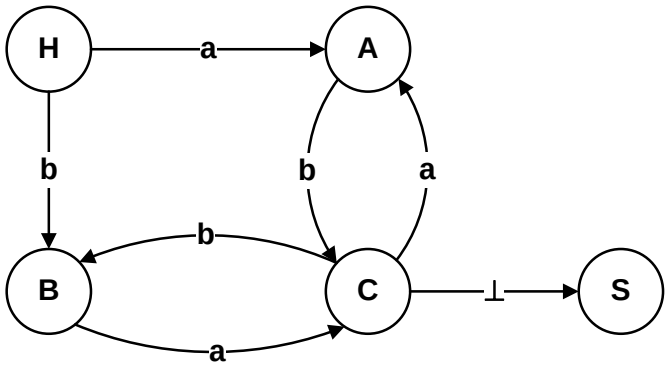


Рисунок 2. Диаграмма состояний для грамматики из примера 4

Алгоритм разбора по диаграмме состояний:

- ✓ объявляем текущим состояние H;
- ✓ затем многократно (до тех пор, пока не считаем признак конца цепочки) выполняем следующие шаги: считываем очередной символ исходной цепочки и переходим из текущего состояния в другое состояние по дуге, помеченной этим символом. Состояние, в которое мы при этом попадаем, становится текущим.

При работе этого алгоритма возможны ситуации, аналогичные тем, которые возникают при разборе непосредственно по регулярной грамматике.

Диаграмма состояний определяет конечный автомат, построенный по регулярной грамматике, который допускает множество цепочек, составляющих язык, определяемый этой грамматикой. Состояния и дуги ДС – это графическое изображение функции переходов конечного автомата из состояния в состояние при условии, что очередной анализируемый символ совпадает с символом-меткой дуги.

Среди всех состояний выделяется *начальное* (считается, что в начальный момент своей работы автомат находится в этом состоянии) и *конечное* (если автомат завершает работу переходом в это состояние, то анализируемая цепочка им допускается).

Таким образом, **конечный автомат** (КА) – это пятёрка (**K**, **VT**, **F**, H, S), где **K** – конечное множество состояний; **VT** – конечное множество допустимых входных символов; **F** – отображение множества $\mathbf{K} \times \mathbf{VT} \to \mathbf{K}$, определяющее поведение автомата; отображение **F** часто называют *функцией переходов*; $H \in \mathbf{K}$ – начальное состояние; $S \in \mathbf{K}$ – заключительное состояние (либо конечное множество заключительных состояний).

$F(A, t) = B$ означает, что из состояния A по входному символу t происходит переход в состояние B.

Конечный автомат *допускает цепочку* $a_1a_2...a_n$, если $F(H, a_1) = A_1$; $F(A_1, a_2) = A_2$; ...; $F(A_{n-2}, a_{n-1}) = A_{n-1}$; $F(A_{n-1}, a_n) = S$, где $a_i \in \mathbf{VT}$, $A_j \in \mathbf{K}$, $j = 1, 2, ..., n-1$; $i = 1, 2, ..., n$; H – начальное состояние, S – одно из заключительных состояний.

Для более удобной работы с диаграммами состояний введём несколько соглашений:

а) если из одного состояния в другое выходит несколько дуг, помеченных разными символами, то будем изображать одну дугу, помеченную всеми этими символами;

б) непомеченная дуга будет соответствовать переходу при любом символе, кроме тех, которыми помечены другие дуги, выходящие из этого состояния;

в) введём *состояние ошибки* (ER); переход в это состояние будет означать, что исходная цепочка языку не принадлежит.

По диаграмме состояний можно написать анализатор для регулярной грамматики.

Для грамматики из примера 4 анализатор будет таким:

Листинг 1. Анализатор для грамматики из примера 4.

```
#include <stdio.h>
int scan_G()
{
  /* множество состояний */
  enum state {H, A, B, C, S, ER};
  enum state CS; /* CS - текущее состояние */
  FILE *fp;
  int c;
  /* текущее состояние = начальное состояние */
  CS = H;
  /* открываем файл и считываем первый символ */
  fp = fopen("data","r");
  c = fgetc(fp);
  do
  {
    switch(CS)
    {
      /* начальное состояние */
      case H:
         if(c == 'a')
         {
           c = fgetc(fp);
           CS = A;
         }
         else if(c == 'b')
         {
           c = fgetc(fp);
           CS = B;
         }
         else CS = ER;
         break;
      /* состояние A */
      case A:
         if(c == 'b')
         {
           c = fgetc(fp);
           CS = C;
```

Листинг 1. Анализатор для грамматики из примера 4.

```
            }
            else CS = ER;
            break;
        /* состояние B */
        case B:
            if(c == 'a')
            {
              c = fgetc(fp);
              CS = C;
            }
            else CS = ER;
            break;
        /* состояние C */
        case C:
            if(c =='a')
            {
              c = fgetc(fp);
              CS = A;
            }
            else if(c == 'b')
            {
              c = fgetc(fp);
              CS = B;
            }
            else if(c == '⊥') CS = S;
            else CS = ER;
            break;
      }
   }while(CS != S && CS != ER);
   if(CS == ER) return -1;
   else return 0;
}
```

При анализе по регулярной грамматике может оказаться, что несколько нетерминалов имеют одинаковые правые части, и поэтому неясно, к какому из них делать «свёртку» (см. описание алгоритма). В терминах диаграммы состояний это означает, что из одного состояния выходит несколько дуг, ведущих в разные состояния, но помеченных одним и тем же символом.

Пример 5. Для грамматики **G** ({*a, b*, ⊥}, {*S, A ,B*}, **P**, *S*), где

P:
$S \to A\perp$
$A \to a \mid Bb$
$B \to b \mid Bb$

разбор будет недетерминированным (т.к. у нетерминалов *A* и *B* есть одинаковые правые части – *Bb*). Такой грамматике будет соответствовать недетерминированный конечный автомат.

Недетерминированный конечный автомат (НКА) – это пятёрка (**K**, **VT**, **F**, **H**, **S**), где **K** – конечное множество состояний; **VT** – конечное множество допустимых входных символов; **F** – отображение множества **K** × **VT** в множество подмножеств **K**; **H** ⊂ **K** – конечное множество начальных состояний; **S** ⊂ **K** – конечное множество заключительных состояний. $F(A, t) = \{B_1, B_2, ..., B_n\}$ означает, что из состояния A по входному символу t можно осуществить переход в любое из состояний B_i, $i = 1, 2, ..., n$.

В этом случае можно предложить алгоритм, который будет перебирать все возможные варианты «свёрток» (переходов) один за другим; если цепочка принадлежит языку, то будет найден путь, ведущий к успеху; если будут просмотрены все варианты, и каждый из них будет завершаться неудачей, то цепочка языку не принадлежит. Однако такой алгоритм практически неприемлем, поскольку при переборе вариантов мы, скорее всего, снова окажемся перед проблемой выбора и, следовательно, будем иметь «дерево отложенных вариантов».

Один из наиболее важных результатов теории конечных автоматов состоит в том, что класс языков, определяемых недетерминированными конечными автоматами, совпадает с классом языков, определяемых детерминированными конечными автоматами.

Это означает, что для любого НКА всегда можно построить детерминированный КА, определяющий тот же язык.

Алгоритм построения детерминированного КА по НКА

Вход: **M** (**K**, **VT**, **F**, **H, S**) – недетерминированный конечный автомат.

Выход: **M′** (**K′**, **VT**, **F′**, **H′, S′**) – детерминированный конечный автомат, допускающий тот же язык, что и автомат **M**.

Метод:

1. Множество состояний **K′** состоит из всех подмножеств множества **K**. Каждое состояние из **K′** будем обозначать $[A_1A_2 ... A_n]$, где $A_i \in$ **K**.

2. Отображение **F′** определим как **F′** $([A_1A_2 ... A_n], t) = [B_1B_2 ... B_m]$, где для каждого $1 \le j \le m$, $F(A_i, t) = B_j$ для каких-либо $1 \le i \le n$.

3. Пусть **H** $= \{H_1, H_2, ..., H_k\}$, тогда **H′** $= \{H_1, H_2, ..., H_k\}$.

4. Пусть **S** $= \{S_1, S_2, ..., S_p\}$, тогда **S′** – все состояния из **K′**, имеющие вид $[... S_i ...]$, $S_i \in$ **S** для какого-либо $1 \le i \le p$.

Во множестве **K′** могут оказаться состояния, которые недостижимы из начального состояния, их можно исключить.

Пример 6.

Пусть задан НКА **M** $= (\{H, A, B, S\}, \{0, 1\}, \mathbf{F}, \{H\}, \{S\})$, где

$F(H, 1) = B$, $F(B, 0) = A$, $F(A, 1) = B$, $F(A, 1) = S$,

тогда соответствующий детерминированный конечный автомат будет таким:

K′ $= \{[H], [A], [B], [S], [HA], [HB], [HS], [AB], [AS], [BS], [HAB], [HAS], [ABS], [HBS], [HABS]\}$

$F'([A], 1) = [BS]$
$F'([H], 1) = [B]$
$F'([B], 0) = [A]$
$F'([HA], 1) = [BS]$
$F'([HB], 1) = [B]$
$F'([HB], 0) = [A]$
$F'([HS], 1) = [B]$
$F'([AB], 1) = [BS]$
$F'([AB], 0) = [A]$
$F'([AS], 1) = [BS]$

$F'([BS], 0) = [A]$
$F'([HAB], 0) = [A]$
$F'([HAB], 1) = [BS]$
$F'([HAS], 1) = [BS]$
$F'([ABS], 1) = [BS]$
$F'([ABS], 0) = [A]$
$F'([HBS], 1) = [B]$
$F'([HBS], 0) = [A]$
$F'([HABS], 1) = [BS]$
$F'([HABS], 0) = [A]$

$\mathbf{S}' = \{[S], [HS], [AS], [BS], [HAS], [ABS], [HBS], [HABS]\}$

Достижимыми состояниями в получившемся КА являются $[H]$, $[B]$, $[A]$ и $[BS]$, поэтому остальные состояния удаляются.

Таким образом, $\mathbf{M}'$ ($\{[H], [B], [A], [BS]\}$, $\{0, 1\}$, $\mathbf{F}'$, H, $\{[BS]\}$), где $F'([A], 1) = [BS]$, $F'([H], 1) = [B]$, $F'([B], 0) = [A]$, $F'([BS], 0) = [A]$.

Регулярные выражения

Регулярные выражения представляют собой еще один способ определения регулярных языков. Регулярные выражения определяют точно те же языки, что и конечные автоматы. Но в отличие от автоматов, они позволяют определять допустимые цепочки декларативным способом. Поэтому регулярные выражения используются в качестве входного языка во многих системах, обрабатывающих цепочки символов, например:

- ✓ в утилите `grep`, использующейся в операционной системе GNU/Linux для поиска строк по заданному регулярному выражению;
- ✓ в генераторах лексических анализаторов Lex и Flex как формальное описание лексем (ключевых слов, идентификаторов, операторов, числовых констант) языка.

Алгебра регулярных выражений состоит из констант (ε, ∅) и переменных для обозначения языков и операторов для задания трех операций: объединение (+ или |), конкатенация (.) и итерация (*).

Константы ε и ∅ являются регулярными выражениями, определяющими языки {ε} и ∅, соответственно, т.е. $\mathbf{L}(\varepsilon) = \{\varepsilon\}$ и $\mathbf{L}(\varnothing) = \varnothing$.

Если a – произвольный символ, то регулярное выражение, определяющее язык $\{a\}$, состоит просто из символа a.

Для группировки констант и переменных в регулярных выражениях используются круглые скобки ( ).

Если E и F – регулярные выражения, то E | F (или E + F) – регулярное выражение, определяющее **объединение** языков $\mathbf{L}(E)$ и $\mathbf{L}(F)$.

Регулярное выражение, определяющее **конкатенацию** языков $\mathbf{L}(E)$ и $\mathbf{L}(F)$, записывается в виде E.F или просто EF.

Итерация E* – это регулярное выражение, определяющее объединение конкатенаций E с самим собой 0 или более раз.

Пример 7. Напишем регулярное выражение для множества цепочек из чередующихся нулей и единиц.

Сначала построим регулярное выражение для языка, состоящего из одной-единственной цепочки 01. Затем используем оператор итерации (*) для того, чтобы построить выражение для всех цепочек вида 0101...01.

0 и 1 — это выражения, обозначающие языки {0} и {1}, соответственно. Если соединить эти два выражения, то получится регулярное выражение 01 для языка {01}. Как правило, если

мы хотим написать выражение для языка, состоящего из одной цепочки, то используем саму эту цепочку как регулярное выражение.

Далее, для получения всех цепочек, состоящих из нуля или нескольких вхождений 01, используем регулярное выражение (01)*. Заметим, что выражение 01 заключается в скобки, чтобы не путать его с выражением 01*. Причина такой интерпретации состоит в том, что операция * имеет высший приоритет по сравнению с операцией конкатенации, и поэтому аргумент оператора итерации выбирается до выполнения любых конкатенаций.

Однако **L**((01)*) – не совсем тот язык, который нам нужен. Он включает только те цепочки из чередующихся нулей и единиц, которые начинаются с 0 и заканчиваются 1. Мы должны также учесть возможность того, что вначале стоит 1 и/или в конце 0. Одним из решений является построение еще трех регулярных выражений, описывающих три другие возможности. Итак, (10)* представляет те чередующиеся цепочки, которые начинаются символом 1 и заканчиваются символом 0, 0(10)* можно использовать для цепочек, которые начинаются и заканчиваются символом 0, а 1(01)* – для цепочек, которые и начинаются, и заканчиваются символом 1. Полностью это регулярное выражение имеет следующий вид:

(01)* | (10)* | 0(10)* | 1(01)*

Заметим, что оператор | используется для объединения тех четырех языков, которые вместе дают все цепочки, состоящие из чередующихся символов 0 и 1.

Однако существует еще одно решение, приводящее к регулярному выражению, которое имеет значительно отличающийся и к тому же более краткий вид. Снова начнем с выражения (01)*. Можем добавить необязательную единицу в начале, если слева к этому выражению допишем выражение ε | 1. Аналогично, добавим необязательный 0 в конце с помощью конкатенации с выражением ε | 0. Таким образом, совокупность цепочек из чередующихся нулей и единиц может быть представлена следующим выражением:

(ε | 1)(01)* (ε | 0)

Как и в других алгебрах, операторы регулярных выражений имеют определенные приоритеты.

Самый высокий приоритет имеет оператор итерации (*), а это значит, что он применяется только к наименьшей последовательности символов, находящейся слева от него и являющейся правильно построенным регулярным выражением.

Далее по порядку приоритетности следует оператор конкатенации. В заключение, со своими операндами связываются операторы объединения (|). Конечно, иногда необходимо, чтобы группировка в регулярном выражении определялась не только приоритетом операторов. В таких случаях можно расставить скобки, сгруппировав операнды по своему усмотрению.

Например, выражение 01* | 1 с учетом приоритетности группируется как (0(1*)) | 1. Сначала выполняется оператор *. Поскольку символ 1, находящийся непосредственно слева от оператора, является допустимым регулярным выражением, то он один будет операндом для итерации. Далее группируем конкатенацию 0 и (1*) и получаем выражение (0(1*)). Наконец, оператор объединения связывает последнее выражение с выражением, которое находится справа, т.е. с 1.

Рассмотрим более сложные примеры регулярных выражений, описывающих конструкции современных языков программирования.

Пример 8. Целочисленная константа без знака представляет собой 0 или цифру, отличную от нуля, за которой может следовать последовательность других цифр. Регулярное выражение будет иметь такой вид: 0 | (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)*. Для удобства можно ввести сокращенное обозначение диапазона значений, и тогда вместо перечисления всех цифр от 0 до 9 можно просто записать как [0…9], а все регулярное выражение – 0 | [1…9][0…9]*.

Пример 9. Вещественное число с фиксированной точкой без знака можно записать так:

(0 | [1…9] [0…9]*) (ε | . [0…9]*)

В данном случае регулярное выражение состоит из двух частей: регулярного выражения для целого без знака, за которым может следовать точка и дробная часть или ничего (ε).

Пример 10. Для представления строковой константы в двойных кавычках введем дополнительный символ – ^, обозначающий **исключение** некоторого символа, следующего за этим оператором. Оператор исключения имеет наивысший приоритет.

Используя этот оператор можно следующим образом записать регулярное выражение для строковой константы: “(^”)*”. Здесь регулярное выражение в скобках означает «любой символ, кроме “».

Другие интересные примеры записи регулярных выражений для различных лексем языков программирования приведены в [3].

Задание на лабораторную работу

1. Дана грамматика. Постройте вывод заданной цепочки:

a) $S \to T \mid T{+}S \mid T{-}S$
$T \to F \mid F{*}T$
$F \to a \mid b$
Цепочка $a{-}b{*}a{+}b$

b) $S \to aSBC \mid abC$
$CB \to BC$
$bB \to bb$
$bC \to bc$
$cC \to cc$
Цепочка $aaabbbccc$

2. Какой язык порождается грамматикой с правилами:

a) $S \to aaCFD$
$AD \to D$
$F \to AFB \mid AB$
$Cb \to bC$
$AB \to bBA$
$CB \to C$
$Ab \to bA$
$bCD \to \varepsilon$

б) $S \to A\perp \mid B\perp$
$A \to a \mid Ba$
$B \to b \mid Bb \mid Ab$

3. Построить грамматику, порождающую язык:

a) $\mathbf{L} = \{a^n b^m c^k \mid n, m, k > 0\}$
б) $\mathbf{L} = \{0^n (10)^m \mid n, m \geq 0\}$
в) $\mathbf{L} = \{a_1 a_2 \ldots a_n a_n \ldots a_2 a_1 \mid a_i \in \{0, 1\}\}$

4. К какому типу по Хомскому относится грамматика с правилами:

a) $S \to 0A1 \mid 01$
$0A \to 00A1$
$A \to 01$

б) $S \to Ab$
$A \to Aa \mid ba$

5. Эквивалентны ли грамматики с правилами:

$S \to aSL \mid aL$
$L \to Kc$
$cK \to Kc$
$K \to b$

и

$S \to aSBc \mid abc$
$cB \to Bc$
$bB \to bb$

6. Построить КС-грамматику, эквивалентную грамматике с правилами:

$S \to AB \mid ABS$
$AB \to BA$
$BA \to AB$
$A \to a$
$B \to b$

7. Построить регулярную грамматику, эквивалентную грамматике с правилами:

$S \to A \,.\, A$
$A \to B \mid BA$
$B \to 0 \mid 1$

8. Постройте контекстно-свободную грамматику для:

а) выражения *while* в языке Си;
б) выражения *for* в языке Си;
в) выражения *do-while* в языке Си.

9. Дана грамматика **G**:

$$S \to aSbS \mid bSaS \mid \varepsilon$$

а) Постройте все возможные деревья вывода для цепочки *abab*.
б) Является ли эта грамматика неоднозначной?

10. Напишите регулярное выражение для:

а) множества идентификаторов, где идентификатор – это последовательность букв или цифр, начинающаяся с буквы или '_';
б) множества вещественных констант с плавающей точкой, состоящих из целой части, десятичной точки, дробной части, символа e или E, целого показателя степени с необязательным знаком и необязательного суффикса типа – одной из букв f, F, l или L. Целая и дробная части состоят из последовательностей цифр. Может отсутствовать либо целая, либо дробная часть (но не обе сразу).

11. Написать леволинейную регулярную грамматику, эквивалентную данной праволинейной, допускающую детерминированный разбор:

а)
$$S \to 0S \mid 0B$$
$$B \to 1B \mid 1C$$
$$C \to 1C \mid \perp$$

б)
$$S \to aA \mid aB \mid bA$$
$$A \to bS$$
$$B \to aS \mid bB \mid \perp$$

12. Даны две грамматики **G1** и **G2**, порождающие языки **L1** и **L2**. Построить регулярную грамматику для $\mathbf{L1} \cap \mathbf{L2}$. Для полученной грамматики построить детерминированный конечный автомат.

G1:
$$S \to S1 \mid A0$$
$$A \to A1 \mid 0$$

G2:
$$S \to A1 \mid B0 \mid E1$$
$$A \to S1$$
$$B \to C1 \mid D1$$
$$C \to 0$$
$$D \to B1$$
$$E \to E0 \mid 1$$

Контрольные вопросы

1. Дайте определение цепочки, языка. Что такое синтаксис и семантика языка?
2. Какие существуют методы задания языков? Какие дополнительные вопросы необходимо решить при задании языка программирования?
3. Что такое грамматика? Дайте определения грамматики.
4. Как выглядит описание грамматики в форме Бэкуса-Наура?
5. Какие типы грамматик существуют?
6. Какие грамматики относятся к регулярным грамматикам?
7. Можно ли для языка, заданного леволинейной грамматикой, построить праволинейную грамматику, задающую эквивалентный язык?
8. Всякая ли регулярная грамматика является однозначной?

9. В чём заключается отличие автоматных грамматик от других регулярных грамматик? Всякая ли регулярная грамматика является автоматной?
10. Что такое конечный автомат (КА)? Дайте определение детерминированного и недетерминированного конечных автоматов.

Список литературы

1. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Т. 1: Синтаксический анализ: Пер. с англ. – М. : Мир, 1978. – 613 с.
2. Хопкрофт Дж. Э., Мотвани Р., Ульман Дж. Д. Введение в теорию автоматов, языков и вычислений, 2-е изд. : Пер. с англ. – М. : Издательский дом «Вильямс», 2002. – 528 с.
3. Молчанов А.Ю. Системное программное обеспечение: Учебник для вузов. 3-е изд. – СПб.: Питер, 2010. – 400 с.
4. Cooper K.D., Torczon L. Engineering a Compiler, 2nd ed. – Elsevier, Inc., 2012. – 825 p.