МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

А. А. Скворцов

ВВЕДЕНИЕ В ТЕОРИЮ ГРАММАТИК

УЧЕБНОЕ ПОСОБИЕ

Печатается по решению редакционно-издательского совета
Вятского государственного университета

УДК 681.332
БКК У 291 – 212 (07)
Ю 943

Скворцов А. А. Введение в теорию грамматик: учебное пособие / А. А. Скворцов. – Киров: Изд-во ВятГУ, 2017. – 122 с.

В учебном пособии рассмотрены теоретические аспекты построения порождающих и распознающих грамматик, а также примеры применения грамматик в задании языков, в компиляторах и поисковых системах.

Учебное пособие рекомендуется для преподавателей и студентов, обучающихся по направлению «Информатика и вычислительная техника» и изучающих дисциплину «Теория автоматов». Также пособие может быть полезно для студентов и аспирантов других специальностей, интересующихся программированием, проектированием и применением компьютерной техники.

Редактор Е. Г. Козвонина

Подписано в печать		Усл. печ. л.
Бумага для офисной техники		Печать цифровая
Заказ № 5	Тираж 200	Бесплатно

Текст напечатан с оригинала-макета, представленного авторами.

610000, г. Киров, ул. Московская, 36
Оформление обложки, изготовление ПРИП ВятГУ

© А. А. Скворцов, 2017
© Вятский государственный университет, 2017

Введение

В последние годы с большой интенсивностью ведутся работы по созданию и применению различных языков программирования, описания документов, конструирования и т.п., а также сред разработки, САПР, поисковых систем, переводчиков. В основе данного программного обеспечения лежат грамматики различных видов: порождающие, распознающие и преобразующие. Кроме того, с помощью порождающих грамматик можно задать как входные данные, так и выходные, а решение задачи оформить в виде грамматики, преобразующей входные данные в выходные.

Применение грамматик не ограничивается данными областями. Построение грамматик может применяться во всех областях, связанных с разработкой новых структур данных, распознаванием структур данных и преобразованием одних структур данных в другие. Структурами данных могут быть различные языки, упорядоченные данные различных объектов и систем.

Теория грамматик так же, как и теория автоматов, рассматривает поведение некоторой системы методом «черного ящика», в котором рассматриваются законы получения выходных данных в зависимости от входных. В отличие от теории автоматов теория грамматик оперирует с множествами входных и выходных данных, которые могут быть заданы и проанализированы с помощью одного и того же набора правил.

Основные понятия и определения теории грамматик

Приведем некоторые базовые определения теории грамматик.

Алфавит V - конечное непустое множество элементов, называемых символами (буквами).

Цепочкой (или словом) ***a*** в алфавите V называется любая конечная последовательность символов этого алфавита. Например, пусть алфавит V = {a,b,c}. Тогда *baaa* является словом в алфавите V.

Цепочка, которая не содержит ни одного символа, называется *пустой* цепочкой и обозначается $\varepsilon(\lambda)$.

Длиной цепочки w называется число составляющих ее символов (обозначается $|w|$), причём каждый символ считается столько раз, сколько раз он встречается в *w*. Например, $|baaa| = 4$ и $|\varepsilon| = 0$.

Обозначим через V^* множество, содержащее все цепочки в алфавите V*,* включая пустую цепочку ε*,* а через V_+ множество, содержащее все цепочки в алфавите *V,* исключая пустую цепочку ε. Например, пусть V = {1,0}, тогда $V^*=\{\varepsilon,0,l,00,01,10,11, 000,\ldots\}$, а $V_+=\{0,1,00,01,10,11,000,...\}$.

Если *x* и *y* - слова в алфавите *V*, то слово *xy* (результат приписывания слова *y* в конец слова *x)* называется *конкатенацией (катенацией, сцеплением)* слов *x* и *y.* Иногда конкатенацию слов *x* и *y* обозначают $x \cdot y$.

Если *x* - слово и $n \in N$, то через x^n обозначается слово $\underline{x \cdot x \cdot \ldots \cdot x}$. По определению $x^0 = \varepsilon$. Например, $ba^3 = baaa$ и $(ba)^3 = bababa$.

Говорят, что слово *x* - *префикс (начало)* слова *y,* если *y = xu.*

Говорят, что слово *x* - *суффикс (конец)* слова *y,* если *y = ux.*

Говорят, что слово *x* — *подслово* слова *y,* если *y = uxv* для некоторых слов *u* и *v.*

Через $|w|_a$ обозначается количество вхождений символа *a* в слово *w.* Например, если V = *{a,b,c}*, то $|baaa|_a = 3$, $|baaa|_b = 1$ и $|baaa|_c = 0$.

Формальный язык – это множество конечных слов (строк, цепочек) над конечным алфавитом V. Например, множество *{a, abb}* является языком над алфавитом *{*a,b*}*, множество $\{a^k b a^l \mid k \le 1\}$ является языком над алфавитом *{a,b}*.

Поскольку каждый язык является множеством, можно рассматривать операции объединения, пересечения, разности и дополнения языков, заданных над одним и тем же алфавитом (обозначения $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 — L_2$).

Необходимо различать *пустой язык* L=0 и язык, содержащий только пустую цепочку: $L=\{\varepsilon\}\neq 0$.

Пусть L – язык над алфавитом V*. Тогда язык $V^* — L$ называется *дополнением* языка L относительно алфавита V. Когда из контекста ясно, о каком алфавите идёт речь, говорят просто, что язык $V^* — L$ является дополнением языка L.

Грамматика – система правил, предназначенная для задания множества цепочек и символов данного алфавита. G – грамматика; L(G) – язык этой грамматики.

Выделяют 3 группы формальных грамматик.

1. *Порождающие грамматики* позволяют строить правильную цепочку в заданном алфавите с описанием ее строения и не позволяют строить ни одной неправильной цепочки.

2. *Распознающие грамматики* позволяют определить, является ли входная цепочка правильной; в случае положительного ответа распознающая ФГ выдает строение цепочки.

3. *Преобразующие грамматики* для каждой правильно построенной цепочки способны построить ее отображение в виде другой цепочки и вывести информацию о порядке проведения изображения.

Теперь перейдем к подробному рассмотрению каждой группы грамматик.

Порождающие грамматики

Конечный язык можно описать простым перечислением его цепочек. Поскольку формальный язык может быть и бесконечным, требуются механизмы, позволяющие конечным образом представлять бесконечные языки. Одним из таких механизмов является использование порождающих грамматик, которые иногда называют грамматиками Хомского.

Порождающей формальной грамматикой называется четверка вида

$$G = (V_T, V_N, P, S),$$

где V_T - множество терминальных символов грамматики (обычно строчные латинские буквы, цифры, и т.п.)***;***

V_N - конечное множество нетерминальных символов грамматики (обычно прописные латинские буквы), $V_T \cap V_N = \boldsymbol{0}$;

P - множество правил вывода грамматики; элемент (α,β) множества P называется правилом вывода и записывается в виде $\alpha\to\beta$ (читается: «из цепочки α выводится цепочка β »)

S - начальный символ грамматики, $S \in V_N$.

Например, грамматика G_1=({0, 1}, *{A, S}*, P_1, *S),* где множество P_1 состоит из правил вида: 1) $S\to 0A1$; 2) $0A\to 00A1$; 3) $A\to\varepsilon$.

Для записи правил вывода с одинаковыми левыми частями вида $\alpha\to\beta_1$, $\alpha\to\beta_2,\ldots,\alpha\to\beta_n$ используется *сокращенная форма* записи $\alpha\to\beta_1|\beta_2|\ldots|\beta_n$.

Цепочка $\beta \in (V_T \cup V_N)^*$ *непосредственно выводима* из непустой цепочки $\alpha \in (V_T\cup V_N)^+$ в грамматике $G = (V_T,V_N,P,S)$ (обозначается: $\alpha\Rightarrow\beta$), если $\alpha=\xi_1\gamma\xi_2$ и $\beta=\xi_1\delta\xi_2$, где $\xi_1,\xi_2,\delta \in V^*$, $\gamma\in V^+$ и правило вывода $\gamma\to\delta$ содержится во множестве P.

Цепочка $\beta\in V^*$ *выводима* из непустой цепочки $\alpha\in V^+$ в грамматике $G=(V_T,V_N,P,S)$ (обозначается: $\alpha\Rightarrow^*\beta$), если существует последовательность цепочек $\gamma_0,\gamma_1,\ldots,\gamma_n$ ($n\geq 0$) такая, что $\alpha=\gamma_0\Rightarrow\gamma_1\Rightarrow\ldots\Rightarrow\gamma_n=\beta$. Например, в грамматике G_1 S=>*000111, т.к. существует вывод S => 0A1=> 00A11 => 000A111 => 000111.

Языком, порожденным грамматикой $G = (V_T,V_N,P,S)$**,** называется множество всех цепочек в алфавите V_T, которые выводимы из начального символа грамматики

S с помощью правил множества P, т.е. множество $L(G)= \{\alpha \in V^* \mid S \Rightarrow^* \alpha\}$. Например, для грамматики G_1 $L(G1)=\{0^n 1^n \mid n>0\}$.

Цепочка $\alpha \in V^*$, для которой существует вывод $S \Rightarrow^* \alpha$, называется *сентенциальной формой* в грамматике $G = (V_T, V_N, P, S)$.

Грамматики G_1 и G_2 называются *эквивалентными*, если они порождают один и тот же язык: $L(G_1) = L(G_2)$. Например, для грамматики G_1 эквивалентной будет грамматика $G_2 = (\{0, 1\}, \{S\}, P_2, S)$, где множество правил вывода P_2 содержит правила вида $S \to 0S1|01$.

Классификация порождающих грамматик по Хомскому

Ограничения на виды правил позволяют выделить классы грамматик. Рассмотрим классификацию, которую предложил Н. Хомский.

Грамматики типа 0. Грамматика $G = (V_T, V_N, P, S)$ называется грамматикой типа 0, если на ее правила вывода не накладывается никаких ограничений, кроме тех, которые указаны в определении грамматики. Любое правило $\alpha \to \beta$ может быть построено с использованием произвольных цепочек $\alpha, \beta \in (V_T \cup V_N)$. Этот тип грамматик самый общий, включающий все грамматики. Однако некоторые грамматики могут принадлежать только к этому типу. Практического применения в силу своей сложности такие грамматики не имеют. Например: P = TR→HT или abC→xDa.

Грамматики типа 1. К этому типу относятся контекстно-зависимые (КЗ) грамматики и неукорачивающие грамматики. Грамматика $G = (V_T, V_N, P, S)$ называется *контекстно-зависимой*, если каждое правило вывода из множества P имеет вид $\alpha A \beta \to \alpha \gamma \beta$, где $\alpha, \beta \in V^*$, $\gamma \in V_+$, $A \in V_N$. Грамматика $G = (V_T, V_N, P, S)$ называется *неукорачивающей*, если каждое правило вывода из множества P имеет вид $\alpha A \beta \to \alpha \gamma \beta$, где $\alpha, \beta \in V^*$, $\gamma \in V_+$, $A \in V_N$ и $|A| <= |\gamma|$. Эти классы грамматик эквивалентны. Могут использоваться при анализе текстов на естественных языках, однако при построении компиляторов практически не используются в силу своей сложности. Для контекстно-зависимых грамматик доказано утверждение: по некоторому алгоритму за конечное число шагов можно установить, принадлежит цепочка терминальных символов данному языку или нет.

Грамматики типа 2. К этому типу относятся контекстно-свободные грамматики (КС-грамматики, бесконтекстные грамматики). Грамматика $G = (V_T, V_N, P, S)$ называется *контекстно-свободной грамматикой* (КС-грамматикой), если ее правила вывода имеют вид: $A \rightarrow \beta$, где $A \in V_N$; $\beta \in V+$ для *неукорачивающих КС-грамматик*, $\beta \in V^*$ для *укорачивающих*. То есть грамматика допускает появление в левой части правила только нетерминального символа. КС-грамматики широко применяются для описания синтаксиса компьютерных языков (программирования).

Грамматики типа 3. К третьему типу относятся *регулярные грамматики* (автоматные) — самые простые из формальных грамматик. Они являются контекстно-свободными, но с ограниченными возможностями. Все регулярные грамматики могут быть разделены на два эквивалентных класса:

- Грамматика $G = (V_T, V_N, P, S)$ называется *праволинейной*, если ее правила вывода имеют вид $A \rightarrow \gamma B$ или $A \rightarrow \gamma$, где $\gamma \in V_T^*$, $A, B \in V_N$;
- Грамматика $G = (V_T, V_N, P, S)$ называется *леволинейной*, если ее правила вывода имеют вид $A \rightarrow B\gamma$ или $A \rightarrow \gamma$, где $\gamma \in V_T^*$, $A, B \in V_N$.

Регулярные грамматики применяются для описания простейших конструкций: идентификаторов, строк, констант, а также языков ассемблера, командных процессоров и др.

При разработке программного обеспечения ЭВМ широко применяются грамматики двух последних типов. Разберем их подробнее.

Контекстно-свободные грамматики и языки

КС грамматики широко используются в практике программирования как способ задания формализованных языков. КС грамматики способны выразить большую часть синтаксиса языков программирования. Применяются также при описании языков HTML, XML, языка описания документов DTD и других.

Язык называется *контекстно-свободным*, если существует контекстно-свободная грамматика, его порождающая.

Описание языка с помощью грамматики состоит из четырех важных компонентов.

1. Конечное множество символов, из которых состоят цепочки определяемого языка. Эти символы называют *терминальными*, или *терминалами*.
2. Конечное множество переменных, называемых *нетерминалами*, или *синтаксическими категориями*. Каждая переменная представляет язык, т.е. множество цепочек.
3. Одна из переменных представляет определяемый язык, она называется *стартовым символом*. Другие переменные представляют дополнительные классы цепочек, которые помогают определить язык, заданный стартовым символом.
4. Конечное множество *правил вывода*, которые представляют рекурсивное определение языка. Каждое правило вывода состоит из следующих частей:

   а) переменная (нетерминал), определяемая правилом вывода;

   б) символ $\rightarrow$;

   в) конечная цепочка, состоящая из терминалов и переменных, возможно, пустая.

   Она представляет способ образования цепочек языка, обозначаемого переменной.

КС-грамматику обычно представляют в виде:

$$\mathbf{G = (V, T, P, S),}$$

где V – множество переменных, T – терминалов, P – правил вывода, S – стартовый символ.

Пример 1. Разработаем грамматику языка палиндромов. Это цепочки символов, читающиеся одинаково справа и слева.

otto madam madamimadam

Для упрощения рассмотрим палиндромы в алфавите {0, 1}.

00100 10101

Выразим определение языка палиндромов в виде системы правил:

1. $P \to \varepsilon$
2. $P \to 0$
3. $P \to 1$
4. $P \to 0P0$
5. $P \to 1P1$

Первые три правила говорят, что язык палиндромов включает цепочки из пустых символов, 0 и 1.

Четвертое и пятое правила – если взять произвольную цепочку *w* из языка P, то 0*w*0 и 1*w*1 также будут в языке P.

Выразим определение языка палиндромов в виде КС-грамматики:

$$G_{pal} = (\{P\}, \{0, 1\}, A, P),$$

где A – приведенные выше правила вывода.

Сокращенная запись грамматики: $G_{pal} = (P \to \varepsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1)$.

Грамматика G_{pal} порождает цепочки языка палиндромов: 10101, 01010, 0110, … Выведем первую цепочку с помощью грамматики палиндромов:

$P \to 1P1 \to 10P01 \to 10101$.

Пример 2. Допустим язык выражений типичного языка программирования состоит из идентификаторов, которые начинаются с буквы a или b, за которой следует цепочка {a,b,0,1} и арифметических операторов **+** и ***** , например,
(a+b)(a+b+a0+a1).*

Введем для составления грамматики две переменные:

E – выражения языка, стартовый символ;

I – идентификаторы языка.

Тогда правила вывода выражений языка программирования:

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$
5. $I \to a$
6. $I \to b$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

Грамматика выражений:

$$G_{выр} = (\{E, I\}, \{a, b, 0, 1, +, *\}, A, E),$$

где A = { E→I | E+E | E*E | (E), I→a | b | Ia | Ib | I0 | I1 }.

Сокращенная запись грамматики:

$$G_{выр} = (E \to I \mid E{+}E \mid E{*}E \mid (E),\ \ I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1)$$

Теперь рассмотрим, как выводится по построенной грамматике вышеуказанная цепочка:

E → E*E → (E)*(E) → (E+E)*(E+E) → (I+I)*(E+E+E+E) → (a+b)*(I+I+I+I) →

→ (a+b)*(a+b+I0+I1) → (a+b)*(a+b+a0+a1).

Вывод цепочек в КС-грамматике удобно представлять с помощью *дерева вывода*.

Контрольные вопросы и задачи

1. Как принято представлять контекстно-свободную грамматику в формальном виде?
2. Составить последовательность вывода цепочки 101101 по грамматике палиндромов

   $G_{pal} = (P \to \varepsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1)$
3. Составить последовательность вывода цепочки (a+b0)*a0+b по грамматике выражений.

   $G_{выр} = (E \to I \mid E{+}E \mid E{*}E \mid (E),\ \ I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1)$

Дерево вывода и неоднозначность грамматик

Деревом вывода цепочки $w \in$ T в грамматике G = (V, T, P, S) называется упорядоченное дерево (связный ациклический граф), узлы которого помечены символами из множеств V, T так, что *корень дерева* помечен стартовым символом, *внутренние узлы* – нетерминалами, а *листья* – терминалами.

Пример 1. Построим дерево вывода цепочки 10101 по грамматике палиндромов G_{pal} = (P $\rightarrow$ ε | 0 | 1 | 0P0 | 1P1). Корень дерева и внутренние узлы – P, листья – заданная цепочка. Вывод цепочки идет от крайних единиц к середине. Если обойти все листья дерева слева направо, то получим в точности выводимую цепочку (см. рисунок 1).

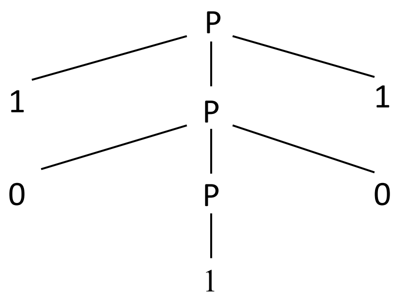


Рисунок 1 – Дерево вывода цепочки 10101 по грамматике палиндромов G_{pal} = (P $\rightarrow$ ε | 0 | 1 | 0P0 | 1P1)

Пример 2. Построим дерево вывода цепочки (a + b) * (a + b + a0 + a1) по грамматике выражений $G_{выр}$ = (E $\rightarrow$ I | E+E | E*E | (E), I $\rightarrow$ a | b | Ia | Ib | I0 | I1). Корень дерева E, внутренние узлы E и I, листья – заданная цепочка. Начнем с умножения выражений в скобках, затем перейдем к сложению внутри скобок. Если обойти все листья дерева слева направо, то получим в точности выводимую цепочку (см. рисунок 2).

Основная роль дерева вывода состоит в том, что оно связывает синтаксис и семантику (смысл) выводимой цепочки. Например, семантика естественного языка – смысл фразы, а для компьютерной программы – это алгоритм решения задачи. Чтобы сохранить семантику языковой цепочки, грамматика и дерево вывода должны быть однозначны.

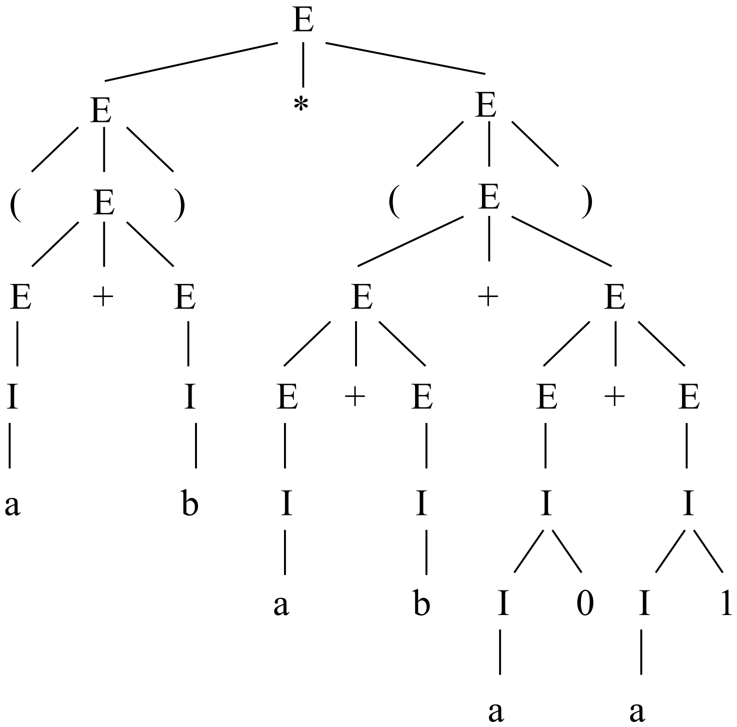


Рисунок 2 – Дерево вывода цепочки (a+b)*(a+b+a0+a1) по грамматике выражений $G_{выр} = (E \rightarrow I \mid E+E \mid E*E \mid (E),\ I \rightarrow a \mid b \mid Ia \mid Ib \mid I0 \mid I1$

Грамматика называется *однозначной*, если каждая цепочка выводимого языка представляется единственным деревом вывода, и *неоднозначной*, если найдется цепочка, представленная двумя различными деревьями вывода.

Неоднозначность – отрицательное свойство грамматики. Если программа кодирует 2 различные последовательности машинных инструкций, то одна из них наверняка реализует не тот алгоритм.

Пример 1. Грамматика G = {E→E+E | E*E | 0 | 1 | … | 9} является неоднозначной, т. к. цепочка *4+2*3* имеет 2 дерева вывода (см. рисунок 3).

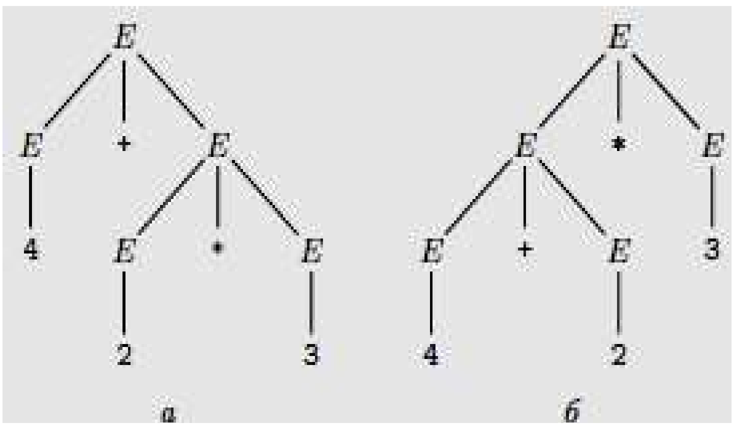


Рисунок 3 – Два дерева вывода цепочки 4+2*3

Дерево «а» показывает, что сложение должно применяться к результату умножения 4+(2*3)=10, а дерево «б» - в умножении участвует результат сложения (4+2)*3=18.

Неоднозначность данной грамматики приводит к невозможности однозначно определить значение выражения, и следовательно, к непригодности данной грамматики для порождения арифметических выражений.

Аналогичная грамматика GA_1 = {E→E+E | E*E | (E) | x} также является неоднозначной, т. к. отсутствует порядок (приоритет) выполнения операций. Эта грамматика порождает и правильную и неправильную цепочки арифметических выражений:

E → E*E → (E+E)*x → (x+x)*x

E → E+E → x+(E) → x+(E*E) → x+(x*x)

Для устранения неоднозначности уточним грамматику согласно определению арифметического выражения: *арифметическое выражение* – это сумма одного или более слагаемых, каждое из которых – произведение одного или более множителей, каждый из которых есть буква «x» или арифметическое выражение в скобках.

Введем дополнительно нетерминал «T» для слагаемых и нетерминал «F» для множителей.

GA_2 = { E→ E+T | T, T→ T*F | F, F→ (E+T) | x }

Попробуем вывести правильную и неправильную цепочки

E → T → T*F → F*x → (E)*x → (E+T)*x → (T+F)*x → (F+x)*x → (x+x)*x

E → E+T → T+T*F → F+F*x → x+x*x

Неправильную цепочку вывести не удалось, вместо нее вывелась правильная, без скобок.

Пример 2. Язык двоичных чисел порождается грамматикой

GN_1 = { S→ L | .L | L.L, L→ LB | B, B→ 0 | 1 }.

Выведем число 0110.1100_2 (6,75).

S → L.L → LB.LB → LB0.LB0 → LB10.LB00 → B110.B100 → 0110.1100

Данная грамматика порождает неправильные цепочки, начинающиеся и заканчивающиеся нулем.

Если целая часть всегда начинается единицей, а дробная единицей заканчивается, то вводим нетерминал «R», обозначающий правую часть числа и уточняем правила вывода левой и правой части:

$$GN_2 = \{ S\to L \mid .R \mid L.R,\ \ L\to L0 \mid L1 \mid 1,\ \ R\to 0R \mid 1R \mid 1 \}.$$

Теперь выводится только правильная цепочка:

$S \to L.R \to L0.1R \to L10.11 \to 110.11$

Контрольные вопросы и задачи

1. Что такое дерево вывода? Как проверить правильность составленного дерева вывода?
2. Составьте дерево вывода цепочки 110.11 по грамматике $GN_2 = \{ S \to L \mid .R \mid L.R,\ \ L\to L0 \mid L1 \mid 1,\ \ R\to 0R \mid 1R \mid 1 \}$.
3. Что такое неоднозначность грамматики и как она устраняется?
4. Составьте деревья вывода правильной *(x+x)*x+x* и неправильной *(x+x)*(x*x)* цепочек по грамматике $GA_1 = \{E\to E+E \mid E*E \mid (E) \mid x\}$.
5. Составьте деревья вывода этих же цепочек по грамматике $GA_2 = \{ E\to E+T \mid T,\ \ T\to T*F \mid F,\ \ F\to (E) \mid x \}$.

Применение КС-грамматик в языках описания документов

В описании языка гипертекстовых ссылок HTML применяется КС-грамматика:

Char → a | A | …

Text → ε | Char Text

Doc → ε | Element Doc

Element → Text |

<EM> Doc </EM> |

<P> Doc |

<OL> List </OL>

…

ListItem → <LI> Doc

List → ε | ListItem List

Text (текст) – это произвольная цепочка символов, не имеющая дескрипторов.

Char (символ) – цепочка, состоящая из одного символа, допустимого в HTML.

Doc (документ) представляет документы, которые являются последовательностями «элементов».

Element (элемент) – это цепочка типа Text, или пара соответствующих дескрипторов и документ между ними, или непарный дескриптор, за которым следует документ.

ListItem (элемент списка) есть дескриптор <LI> со следующим за ним документом, который представляет собой одиночный элемент списка.

List (список) – последовательность из элементов списка.

Рассмотрим вывод цепочки языка HTML, соответствующего тексту:

Вещи, которые я *люблю*:
1. Бананы.
2. Людей общительных, ответственных.

Element → <P>Doc → <P> Element Doc → <P> Text Element Doc →
<P>Char Text <EM>Doc</EM>Element Doc →
<P>В Char Text <EM> Element Doc </EM><OL>List</OL> Doc →
<P>Ве Char Text <EM> Text </EM> <OL>Listitem List</OL> →
<P>Вещ Char Text <EM> Char Text </EM> <OL><LI>Doc Listitem</OL> →
<P>Вещи Char Text <EM> л Char Text </EM> <OL><LI>Element Doc <LI>Doc </OL> →
<P>Вещи, Char Text <EM> лю Char Text </EM> <OL><LI>Text Doc <LI> Element Doc </OL> →
<P>Вещи, Char Text <EM> люб Char Text </EM> <OL><LI> Char Text <LI> Text </OL> →
<P>Вещи, к Char Text <EM> любл Char Text </EM> <OL><LI> Б Char Text <LI> Char Text </OL>
→
…
<P>Вещи, которые я <EM>люблю</EM>:
<OL>
<LI>Бананы.
<LI>Людей общительных, ответственных.
</OL>

КС-грамматики часто используются в описании языков программирования. В этом случае грамматики записываются в определенных формах (метаязыках). Рассмотрим наиболее применимые из них – формы Бэкуса-Наура.

Бэкуса-Наура формы (БНФ)

Метаязык, предложенный Бэкусом и Науром, впервые использовался для описания синтаксиса реального языка программирования Алгол 60. Наряду с новыми обозначениями метасимволов, в нем использовались содержательные обозначения нетерминалов. Это сделало описание языка нагляднее и позволило в дальнейшем широко использовать данную нотацию для описания реальных языков программирования. Были использованы следующие обозначения:

– символ «::=» отделяет левую часть правила от правой;

– нетерминалы обозначаются произвольной символьной строкой, заключенной в угловые скобки «<» и «>»;

– терминалы – это символы, используемые в описываемом языке;

– каждое правило определяет порождение нескольких альтернативных цепочек, отделяемых друг от друга символом вертикальной черты «|».

Пример описания идентификатора с использованием БНФ:

1. <буква> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z |a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
2. <цифра> ::= 0|1|2|3|4|5|6|7|8|9
3. <идентификатор> ::= <буква> | <идентификатор> <буква> | <идентификатор> <цифра>

Правила можно задавать и раздельно:

<идентификатор> :: = <буква>
<идентификатор> :: = <идентификатор> <буква>
<идентификатор> :: = <идентификатор> <цифра>

Расширенные Бэкуса-Наура формы (РБНФ)

Для повышения удобства и компактности описания, целесообразно вести в язык дополнительные конструкции. Существуют различные расширенные формы метаязыков, незначительно отличающиеся друг от друга. Их разнообразие зачастую объясняется желанием разработчиков языков программирования по-своему описать создаваемый язык. К примерам таких широко известных метаязыков можно отнести: метаязык PL/I, метаязык Вирта, метаязык Кернигана-Ритчи, описывающий Си. Зачастую такие языки называются **расширенными формами Бэкуса-Наура** (**РБНФ**).

В частности, РБНФ, используемые Виртом, имеют следующие особенности:

– квадратные скобки «[« и «]» означают, что заключенная в них синтаксическая конструкция может отсутствовать;

– фигурные скобки «{» и «}» означают ее повторение (возможно, 0 раз);

– круглые скобки «(» и «)» используются для ограничения альтернативных конструкций;

– сочетание фигурных скобок и косой черты «{/» и «/}» используется для обозначения повторения один и более раз. Нетерминальные символы изображаются словами, выражающими их интуитивный смысл и написанными на русском языке.

Если нетерминал состоит из нескольких смысловых слов, то они должны быть написаны слитно. В этом случае для повышения удобства в восприятии фразы целесообразно каждое ее слово начинать с заглавной буквы или разделять слова во фразах символом подчеркивания. Терминальные символы изображаются словами, написанными буквами латинского алфавита (зарезервированные слова) или цепочками знаков, заключенными в кавычки. Синтаксическим правилам предшествует знак «$» в начале строки. Каждое правило оканчивается знаком «.» (точка). Левая часть правила отделяется от правой знаком «=» (равно), а альтернативы - вертикальной чертой «|». В соответствии с данными правилами синтаксис идентификатора будет выглядеть следующим образом:

$ буква = ""A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"|
"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"|"a"|"b"|"c"|"d"|"e"|"f"|"g"|"h|
"i"|"j"|"k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z".

$ цифра = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".

$ идентификатор = буква {буква | цифра}.

В качестве примера опишем синтаксис демонстрационного языка программирования с помощью расширенных форм Бэкуса-Наура.

Демонстрационный язык программирования DPL

Можно упростить любой из существующих языков программирования до уровня, удобного для изучения основных методов разработки компиляторов. Такой подход используется достаточно часто. Он удобен и позволяет легко разобрать различные методы построения компиляторов. Назовем разрабатываемый язык DPL (Demonstration Programming Language).

Синтаксис и семантика DPL. DPL содержит основные операторы обработки данных и управления, которые позволяют строить простые программы. Вместе с тем, в нем отсутствуют конструкции, широко применяемые в развитых языках. В частности, нет процедур и функций, блочной структуры, пользовательских типов данных, классов. Описание языка строится по традиционному принципу. В начале будут рассмотрены элементарные конструкции, а затем структура программы.

Элементарные конструкции. К элементарным конструкциям языка обычно относятся его понятия, состоящие из терминальных символов, принадлежащих алфавиту языка.

К элементарным относятся такие понятия, как **идентификатор**, **числа** (целые, действительные, двоичные, десятичные), **комментарии**, **метки**, **знаки операций**, **разделители**, **строки символов**. Список можно продолжить и дальше. Эти понятия уточняются и конкретизируются при описании семантики языка. Например, идентификатор может служить в качестве имени переменной, процедуры, функции или типа.

Ниже приводится синтаксис элементарных конструкций DPL. Их описанию предшествует определение групп основных символов в качестве отдельных понятий, разделяющих эти символы на отдельные категории (классы).

$ **буква** = "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"| "O"|"P"|"Q"| "R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"|"a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z"

$ **цифра** = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

$ **идентификатор** = (буква | "_") { буква | цифра | "_" }

$ **число** = целое | действительное

$ **целое** = двоичное | восьмиричное | десятичное | шестнадцатиричное

$ **двоичное** = "{2}" {/ "0" | "1" /}

$ **восьмиричное** = "{8}"{ "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7" }

$ **десятичное** = ["{10}"] {/ цифра /}

$ **шестнадцатиричное** = "{16}" {/ цифра |"A"|"B"|"C"|"D"|"E"|"F"|
"a"|"b"|"c"|"d"|"e"|"f" /}

$ **действительное** = числовая_строка порядок | числовая_строка
"." [числовая_строка] [порядок] | "." числовая_строка [порядок]

$ **числовая_строка** = {/ цифра /}

$ **порядок** = ("E"|"e")["+"|"-"] числовая_строка

$ **пробельный_символ** = {/ пробел | табуляция | перевод_строки |
комментарий /}

$ **комментарий** = "/*" { символ } "*/"

$ **строка** = """ { символ | """"} """

Дополнительным описанием следует также снабдить понятия **комментария** и **строки**. Комментарий может задаваться в любом месте программы, где можно поставить пробел или разделитель. Он начинается парой символов "/*" и заканчивается другой парой "*/". Между ними могут быть любые печатаемые символы, включая отдельные группы звездочек и наклонных линий, не образующих завершающую комбинацию, а также символы табуляции, перевода строки. Строка может содержать только видимые символы, заключенные между двумя кавычками. Если в строке необходимо поставить кавычку, то она дублируется при написании:

"Строка, в которой следующее ""слово"" взято в кавычки".

Ключевые слова и разделители используются для формирования выражений, описаний и операторов. В DPL они определяются следующим образом:

$ **ключевое_слово** = abort | begin | case | end | float | goto | int | loop |
or | read | skip | space | tab | var | write |

$ **разделитель** = "(" | ")" | "[" | "]" | "," | ";" | ":" | ":=" | "*" | "/" | "%" |
"+" | "-" | "->" |"<" | "=" | ">" | "<=" | ">=" | "!="

Составные конструкции. Организация программы. К составным конструкциям относятся понятия, определяющие структуру программы, ее операторов, описаний и выражений:

$ **программа** = begin (описание | оператор)
{ ";" (описание | оператор) } end

$ **описание** = var идентификатор [размер]
{ "," идентификатор [размер] } ":" тип

$ **тип** = int | float

$ **размер** = целое

$ **оператор** = метка непомеченный

$ **метка** = идентификатор ":"

$ **непомеченный** = присваивания | условный | цикла | пустой |
ошибки | ввода | вывода | перехода

$ **присваивания** = переменная ":=" выражение |
переменная "," присваивания "," выражение

$ **переменная** = идентификатор ["[" выражение "]"]

$ **выражение** = ["-"] операнд { операция ["-"] операнд }

$ **операнд** = "(" выражение ")" | число | переменная

$ **операция** = "*" | "/" | "%" | "+" | "-" | "<" | "=" | ">" | ">=" |
"<=" | "!="

$ **условный** = case [набор_ резервируемых] end

$ **цикла** = loop [набор_ резервируемых] end

$ **набор_резервируемых** = резервируемые [or резервируемые]

$ **резервируемые** = выражение "->" оператор { ";" оператор }

$ **пустой** = skip |

$ **прерывания** = abort [строка]

$ **ввода** = read переменная { "," переменная }

$ **вывода** = write (выражение | спецификатор | строка)
{ "," (выражение | спецификатор | строка) }

$ **перехода** = goto идентификатор

$ **спецификатор** = (space | tab | skip) [выражение]

Приведенные правила требуют некоторых дополнительных пояснений, раскрывающих особенности семантики языка.

Краткое описание семантики языка. Оператор присваивания имеет нетрадиционную форму и, в общем случае, обеспечивает одновременное присваивание нескольким переменным, расположенным слева от знака «:=» значений предварительно вычисленных выражений, расположенных в правой части. Каждой переменной соответствует свое выражение. Присваивания начинаются только после вычисления всех выражений, результаты которых временно сохраняются. Это позволяет произвести обмен значений переменных с использованием только одного оператора присваивания:

x, **y** := **y**, **x**

В обычных языках программирования необходимо написать три отдельных оператора:

t := **x**; **x** := **y**; **y** := **t**

Выражения задаются в традиционной инфиксной форме. Порядок выполнения операций определяется их приоритетом и скобками. В начале выполняются выражения в скобках. Наивысший приоритет имеет унарный минус «-», далее следуют мультипликативные операции «*», «/», «%» (вычисление остатка), затем аддитивные «+», «-» и, наконец операции отношения «<», «=», «>», «<=», «>=», «!=».

Для экстренного выхода из любой точки программы в языке используется оператор прерывания **abort**. Необязательная строка символов предназначена для пояснения причины выхода из программы.

В соответствии с концепциями безошибочного программирования, разработанными Дейкстрой, определены **условный оператор** и **оператор цикла**. Их тела содержат наборы операторов, выполнение которых возможно только при истинности условий, задаваемых предваряющими их охраняющими выражениями. Выражения отделяются от зарезервированных ими операторов стрелками «->» и, начиная с первого, последовательно анализируются до тех пор, пока не встретится «истинное». Истинным считается ненулевое значение выражения. Предполагается, что в рассматриваемой версии языка операции отношения возвращают в качестве результата целое число, равное 1, при выполнении условия и, равное 0, если условие не выполняется. Если в условном операторе все охраняющие выражения дают ложь, то он выполняется как оператор ошибки (abort). Оператор цикла в данной ситуации эквивалентен пустому оператору (skip). Возникновение такой ситуации обеспечивает выход из цикла. При наличии истинного охраняющего выражения происходит выполнение охраняемых операторов и повторное выполнение оператора цикла. Оператор abort также эквивалентен конструкции case end (пустое тело в условном операторе), а оператор skip - оператору loop end.

Спецификаторы **space**, **tab** и **skip** используются в операторе вывода для форматирования выходного потока данных и означают пробел, табуляцию и перевод строки. Выражение, следующее за спецификатором, определяет количество его повторений. Строка символов используется для вывода пояснительного текста.

Примеры программ на DPL

Алгоритм Евклида (нахождение наибольшего общего делителя)

```
begin
    var x, y: int; /* описание переменных */
    read x, y; /* ввод операндов */
    /* выполнять до равенства аргументов */
    loop x != y ->
        case
             x > y -> x := x - y
        or
             y > x -> y := y - x
        end
    end;
    write x /* полученный НОД */
end
```

Пример 1:

Найти НОД для 30 и 21.
x = 30 – 21 = 9
y = 21 – 9 = 12
y = 12 – 9 = 3
x = 9 – 3 = 6
x = 6 – 3 = 3
НОД = x = 3

Пример 2:

Найти НОД для 42 и 18.
x = 42 – 18 = 24
x = 24 – 18 = 6
y = 18 – 6 = 12
y = 12 – 6 = 6
НОД = x = 6

Суммирование n элементов из входного потока

```
begin
    var a, i, s, n: int;
    i,s := 0,0;
    read n;
    loop
        i < n -> read a; s,i := s+a,i+1
    end;
    write s
end
```

Применение КС-грамматик в синтаксических анализаторах

В ОС UNIX с помощью Генератора синтаксических анализаторов YACC по грамматике можно построить *синтаксический анализатор* – функция, создающая по исходным программам деревья разбора или фрагменты машинного кода. Эта функция проверяет корректность исходной программы, входит в состав любого компилятора.

Правила вывода записываются в РБНФ. С любым правилом можно связать действие - набор операторов языка Си, которые будут выполняться при каждом распознавании конструкции во входном тексте.

Действие заключается в фигурные скобки и помещается вслед за правой частью правила, т.е. правило с действием имеет вид:

```
<имя_нетерминального_символа>: определение {действие};
```

При использовании сокращенной записи правил с общей левой частью следует иметь в виду, что действие может относиться только к отдельному правилу, а не к их совокупности.

```
statement: assign_stat
|
if_then_stat {printf("if_оператор\n");}
|
goto_stat {kgoto++; printf("goto_оператор\n");}
```

Контрольные вопросы и задачи

1. Составить последовательность вывода цепочки языка HTML, соответствующего тексту:

   Мне нравится весна:
   1) травой зеленой;
   2) **теплым** солнцем.

2. Составить последовательность вывода цепочки языка DPL, соответствующего программе суммирования n элементов из входного потока

```
begin
    var a, i, s, n: int;
    i,s := 0,0;
    read n;
    loop
        i < n -> read a; s,i := s+a,i+1
    end;
    write s
end
```

Регулярные грамматики и выражения

Регулярная грамматика – формальная грамматика типа 3 по иерархии Хомского. Регулярные грамматики являются подмножеством КС-грамматик.

Регулярные грамматики определяют все регулярные языки и эквивалентны регулярным выражениям. Например, правая регулярная грамматика $G = (V_T, V_N, P, S)$, заданная $V_N = \{S, A\}$, $V_T = \{a, b, c\}$, состоит из следующих правил P:

S → aS

S → bA

A → ε

A → cA

и S является начальным символом. Эта грамматика описывает тот же язык, что и регулярное выражение **a*bc***.

Регулярные выражения (РВ) используются в качестве входного языка во многих системах, обрабатывающих цепочки. Например, команды поиска в браузерах или системах форматирования текста, формальные описания лексем в генераторах лексических анализаторов.

Прежде чем дать определение РВ, рассмотрим 3 операции над языками, соответствующие операторам РВ.

1. *Объединение* двух языков L и M, обозначаемое $L \cup M$, — это множество цепочек, которые содержатся либо в L, либо в M, либо в обоих языках. Например, если $L = \{001, 10, 111\}$ и $M = \{\varepsilon, 001\}$, то $L \cup M = \{\varepsilon, 10, 001, 111\}$.
2. *Конкатенация* языков L и M — это множество цепочек, которые можно образовать путем дописывания к любой цепочке из L любой цепочки из M. Выше в разделе 1.5.2 было дано определение конкатенации двух цепочек: результатом ее является запись одной цепочки вслед за другой. Конкатенация языков обозначается либо точкой, либо вообще никак не обозначается, хотя оператор конкатенации часто называют "точкой". Например, если $L = \{001, 10, 111\}$ и $M = \{\varepsilon, 001\}$, то $L.M$, или просто LM, — это $\{001, 10, 111, 001001, 10001, 111001\}$. Первые три цепочки в LM — это цепочки из L, соединенные с ε. Поскольку ε является единицей (нейтральным элементом) для операции конкатенации, результирующие цепочки будут такими же, как и цепочки из L. Последние же три цепочки в LM образованы путем соединения каждой цепочки из L со второй цепочкой из M, т.е. с 001. Например, 10 из L, соединенная с 001 из M, дает 10001 для LM.
3. *Итерация* ("*звездочка*", или *замыкание Клини*[2]) языка L обозначается L^* и представляет собой множество всех тех цепочек, которые можно образовать путем конкатенации любого количества цепочек из L. При этом допускаются повторения, т.е. одна и та же цепочка из L может быть выбрана для конкатенации более одного раза. Например, если $L = \{0, 1\}$, то L^* — это все цепочки, состоящие из нулей и единиц. Если $L = \{0, 11\}$, то в L^* входят цепочки из нулей и единиц, содержащие четное количество единиц, например, цепочки 011, 11110 или ε, и не входят цепочки 01011 или 101. Более формально язык L^* можно представить как бесконечное объединение $\bigcup_{i\geq 0} L^i$, где $L^0 = \{\varepsilon\}$, $L^1 = L$ и L^i для $i > 1$ равен $LL \ldots L$ (конкатенация i копий L).

Пример

Дан язык $L=\{0, 11\}$. Определить язык L^*.

$L^0=\{\varepsilon\}$
$L^1=\{0, 11\}$
$L^2=\{00, 011, 110, 1111\}$
$L^3=\{000, 0011, 0110, 01111, 1100, 11011, 11110, 111111\}$
…

Для вычисления L^* необходимо вычислить L^i для каждого i и объединить эти языки.

Алгебраические законы регулярных выражений

Определение. Два РВ являются эквивалентными, если при подстановке любых языков вместо переменных оба выражения представляют один и тот же язык.

- $L + M = M + L$, т.е. *коммутативный закон для объединения* утверждает, что два языка можно объединять в любом порядке.
- $(L + M) + N = L + (M + N)$. Этот закон — *ассоциативный закон объединения* — говорит, что для объединения трех языков можно сначала объединить как два первых, так и два последних из них. Обратите внимание на то, что вместе с коммутативным законом объединения этот закон позволяет объединять любое количество языков в произвольном порядке, разбивая их на любые группы, и результат будет одним и тем же. Очевидно, что некоторая цепочка принадлежит объединению $L_1 \cup L_2 \cup \ldots \cup L_k$ тогда и только тогда, когда она принадлежит одному или нескольким языкам L_i.
- $(LM)N = L(MN)$. Этот *ассоциативный закон конкатенации* гласит, что для конкатенации трех языков можно сначала соединить как два первых, так и два последних из них.
- $L(M + N) = LM + LN$. Этот закон называется *левосторонним дистрибутивным законом конкатенации относительно объединения*.
- $(M + N)L = ML + NL$. Этот закон называется *правосторонним дистрибутивным законом конкатенации относительно объединения*.
- $(L^*)^* = L^*$. Этот закон утверждает, что при повторной итерации язык уже итерированного выражения не меняется. Язык выражения $(L^*)^*$ содержит все цепочки, образованные конкатенацией цепочек языка L^*. Последние же цепочки построены из цепочек языка L. Таким образом, цепочки языка $(L^*)^*$ также являются конкатенациями цепочек из L и, следовательно, принадлежат языку L^*.
- $L^+ = LL^* = L^*L$. Напомним, что L^+ по определению равно $L + LL + LLL + \ldots$. Поскольку $L^* = \varepsilon + L + LL + LLL + \ldots$, то

  $$LL^* = L\varepsilon + LL + LLL + LLLL + \ldots$$

Построение РВ

Алгебра РВ строится по той же схеме, что обычная арифметическая алгебра: используются константы и переменные для обозначения языков и операторы для обозначения 3 операций: объединение, конкатенация и итерация.

Базис. Состоит из 3 правил:

1. Константы ϵ и $\varnothing$ являются регулярными выражениями, определяющими языки $\{\epsilon\}$ и $\varnothing$, соответственно, т.е. $L(\epsilon) = \{\epsilon\}$ и $L(\varnothing) = \varnothing$.
2. Если a — произвольный символ, то $\mathbf{a}$ — регулярное выражение, определяющее язык $\{a\}$, т.е. $L(\mathbf{a}) = \{a\}$. Заметим, что для записи выражения, соответствующего символу, используется жирный шрифт. Это соответствие, т.е. что $\mathbf{a}$ относится к a, должно быть очевидным.
3. Переменная, обозначенная прописной курсивной буквой, например, L, представляет произвольный язык.

Индукция. Состоит из 4 правил вывода:

1. Если E и F — регулярные выражения, то $E + F$ — регулярное выражение, определяющее объединение языков $L(E)$ и $L(F)$, т.е. $L(E + F) = L(E) \cup L(F)$.
2. Если E и F — регулярные выражения, то EF — регулярное выражение, определяющее конкатенацию языков $L(E)$ и $L(F)$. Таким образом, $L(EF) = L(E)L(F)$. Заметим, что для обозначения оператора конкатенации — как операции над языками, так и оператора в регулярном выражении — можно использовать точку. Например, регулярное выражение **0.1** означает то же, что и **01**, и представляет язык $\{01\}$. Однако мы избегаем использовать точку в качестве оператора конкатенации в регулярных выражениях[3].
3. Если E — регулярное выражение, то E^* — регулярное выражение, определяющее итерацию языка $L(E)$. Таким образом, $L(E^*) = (L(E))^*$.
4. Если E — регулярное выражение, то (E) — регулярное выражение, определяющее тот же язык $L(E)$, что и выражение E. Формально, $L((E)) = L(E)$.

Приоритеты операторов РВ

1. Итерация « * »
2. Конкатенация « . »
3. Объединение « + ».

Например, выражение 01^*+1 группируется как $(0(1^*))+1$.

Пример.

Напишем РВ для множества цепочек из чередующихся нулей и единиц.

1. Построим РВ для языка, состоящего из одной цепочки «01».

2. Построим выражение для всех цепочек вида 0101…01.

1. **0** и **1** – выражения для языков {0} и {1}, **01** – для языка {01}.

2. **(01)*** - для всех вхождений «01».

Это еще не все, есть другие варианты правильных цепочек.

(10)* - для всех вхождений «10».

0(10)* - для цепочек, которые начинаются и заканчиваются нулем.

(10)***1** - для цепочек, которые начинаются и заканчиваются единицей.

Объединяя эти цепочки получим итоговое РВ:

$\mathbf{(01)^* + (10)^* + 0(10)^* + 1(01)^*}$

Применение регулярных выражений

Регулярные выражения широко применяются в ОС UNIX. Рассмотрим запись выражений в этой системе и затем – два важных класса приложений, основанных на регулярных выражениях: лексические анализаторы и поиск в тексте.

Регулярные выражения в UNIX

Позволяют создавать *классы символов* для представления множеств символов в наиболее кратком виде. Существуют следующие правила для классов символов.

- Символ . (точка) обозначает "любой символ".
- Последовательность $[a_1a_2\ldots a_k]$ обозначает регулярное выражение $a_1 + a_2 + \ldots + a_k$

Такое обозначение позволяет записывать примерно вдвое меньше символов, поскольку нет необходимости писать знак "+". Например, четыре символа, используемые в операторах сравнения языка *C*, можно выразить в виде `[<>=!]`.

- В квадратных скобках записывается диапазон вида *x–y* для обозначения всех символов от *x* до *y* из последовательности символов в коде ASCII. Поскольку коды цифр, а также символов верхнего и нижнего регистров упорядочены, то многие важные классы символов можно записывать с помощью нескольких ключевых цепочек. Например, все цифры могут быть представлены в виде `[0-9]`, символы верхнего регистра могут быть выражены как `[A-Z]`, а множество всех букв и цифр можно записать как `[A-Za-z0-9]`. Если необходимо включить в такой список символов знак минуса, то его помещают в самом начале или в самом конце списка, чтобы не было путаницы с использованием его для обозначения диапазона символов. Например, набор цифр вместе с точкой и знаками плюс и минус, используемый для образования десятичных чисел со знаком, можно записать в виде выражения `[-+.0-9]`. Квадратные скобки и другие символы, имеющие специальные значения в регулярных выражениях UNIX, задаются в качестве обычных символов с помощью обратной косой черты (\)перед ними.
- Для некоторых наиболее часто используемых классов символов введены специальные обозначения. Рассмотрим несколько примеров:
  а) `[:digit:]` обозначает множество из десяти цифр, как и `[0-9]`[4];
  б) `[:alpha:]` обозначает любой символ (латинского) алфавита, как и `[A-Za-z]`;
  в) `[:alnum:]` обозначает все цифры и буквы (буквенные и цифровые символы), как и `[A-Za-z0-9]`.

Дополнительные операторы, облегчающие построение и читаемость выражений:

1. Оператор | используется вместо + для обозначения объединения.
2. Оператор ? значит "ни одного или один из". Таким образом, *R*? в UNIX означает то же, что и $\varepsilon + R$ в системе записи регулярных выражений, принятой в этой книге.
3. Оператор + значит "один или несколько из". Следовательно, *R*+ в UNIX является сокращением для RR^* в наших обозначениях.
4. Оператор {*n*} обозначает "*n* копий". Таким образом, *R*{5} в UNIX является сокращенной записью для *RRRRR* в наших обозначениях.

Лексический анализ

Процесс компиляции состоит из следующих этапов:

1. Лексический анализ. На этом этапе последовательность символов исходного файла преобразуется в последовательность лексем.
2. Синтаксический (грамматический) анализ. Последовательность лексем преобразуется в дерево разбора.
3. Семантический анализ. Дерево разбора обрабатывается с целью установления его семантики (смысла) — например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т. д. Результат обычно называется «промежуточным представлением/кодом», и может быть дополненным деревом разбора, новым деревом, абстрактным набором команд или чем-то ещё, удобным для дальнейшей обработки.
4. Оптимизация. Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла. Оптимизация может быть на разных уровнях и этапах — например, над промежуточным кодом или над конечным машинным кодом.
5. Генерация кода. Из промежуточного представления порождается код на целевом языке.

Лексический анализатор – компонент компилятора, который сканирует исходную программу и распознает все *лексемы,* т. е. подцепочки последовательных символов, логически составляющие единое целое.

UNIX-команда `lex` и ее GNU-версия `flex` получают на вход список регулярных выражений в стиле UNIX, за каждым из которых в фигурных скобках следует код, указывающий, что должен делать лексический анализатор, если найдет экземпляр этой лексемы. Такая система называется *генератором лексического анализатора*, поскольку на ее вход поступает высокоуровневое описание лексического анализатора и по этому описанию она создает функцию, которая представляет собой работающий лексический анализатор.

Ниже приведен пример фрагмента кода лексического анализатора. Левая часть кода представляет собой регулярные выражения для искомых лексем (if, переменная, число, унарная или бинарная операция), а правая - выполняемый код анализатора в случае их нахождения.

```
%%
if          printf ("IF statement\n");
[a-z]+      printf ("tag, value %s\n", yytext);
{D}+        printf ("decimal number %s\n", yytext);
"++"        printf ("unary op\n");
"+"         printf ("binary op\n");
```

Поиск образцов в тексте

РВ являются основным средством приложений для поиска *образцов*, *шаблонов* в тексте. Они задают «схему» образца поиска. С помощью РВ можно не прилагая больших усилий, описывать такие образцы и быстро менять такие описания, если результат не устраивает.

РВ компилируются в ДКА или НКА, которые затем моделируются для получения *программы распознавания образов* в тексте.

Пример 1.

Зададим шаблон (образец) для распознавания названий улиц поисковой системой при просмотре сайтов.

Название улицы может начинаться с «улица», «ул.», «проспект», «пр.», «переулок», «пер.».

улица | ул\. | проспект | пр\. | переулок | пер\.

Затем идет название улицы. Оно начинается с прописной буквы и затем идет несколько строчных.

[А-Я][а-я]*

Название улицы может состоять из нескольких слов с заглавной буквы (например, ул. Карла Маркса)

[А-Я][а-я]*([А-Я][а-я]*)*

Итоговое выражение

(улица| ул\.|проспект|пр\.|переулок|пер\.) [А-Я][а-я]*([А-Я][а-я]*)*

Таким образом, распознавание адресов на web-страницах с помощью компилятора РВ намного проще по сравнению с программой на традиционном языке программирования.

Контрольные вопросы и задачи

1. Построить РВ для множества цепочек из 0 и 1, в которых каждая пара смежных нулей находится перед парой смежных единиц.

2. Написать РВ для поиска адресов, состоящих из названия улицы, номера дома и номера квартиры, записанных через запятую.

Распознаватели

Ранее мы рассмотрели задание языков через механизм порождения, с помощью грамматики. Теперь рассмотрим второй основной подход задания языков – через механизм распознавания.

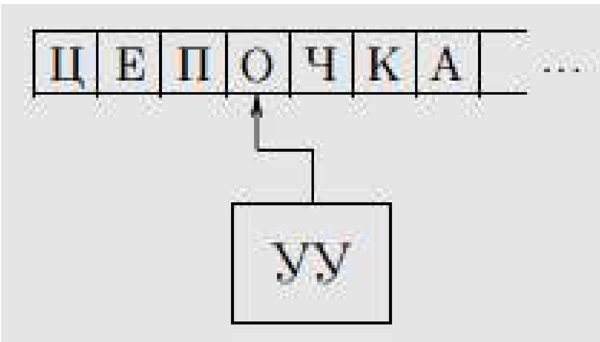
Распознаватель, по сути, является процедурой специального вида, которая по заданной цепочке определяет, принадлежит ли она языку. Если принадлежит, то процедура останавливается с ответом «да», т. е. допускает цепочку; иначе – останавливается с ответом «нет» или зацикливается. Язык, определяемый распознавателем – это множество всех цепочек, которые он допускает.

Определение. *Распознавателем языка* называется алгоритм или физическое устройство, которое по произвольной цепочке определяет, принадлежит ли она данному языку или нет.

Начнем с распознавателя регулярных языков – конечного автомата.

Конечные автоматы – распознаватели

Определение. Детерминированным конечным автоматом – распознавателем называется устройство или алгоритм $A = (Q, S, \delta, q_0, F)$, где Q – непустое конечное множество состояний, S – конечный входной алфавит, $q_0 \in Q$ – начальное состояние, δ - функция переходов, $F \in Q$ – множество заключительных состояний.



На автомат можно смотреть как на физическое устройство, состоящее из устройства управления и входной ленты, которая разделена на ячейки и в них записаны символы. В каждый момент времени УУ находится в некотором состоянии и видит одну ячейку ленты. УУ считывает символ, определяет новое состояние, переходит в него и сдвигается вправо. Такт работы состоит в переходе из текущего состояния при текущем входном символе в следующее состояние.

Определение. Автомат *распознает* (допускает) цепочку, если по окончании цепочки перейдет в одно из заключительных состояний. *Распознаваемым* языком называется множество всех цепочек, распознаваемых этим автоматом.

Задание распознающего КА

1) расширенная таблица переходов
2) диаграмма переходов

Расширенная таблица переходов – таблица значений функции переходов δ, первая строка которой соответствует начальному состоянию, а заключительные состояния помечены единицей в дополнительном столбце.

Диаграммой переходов называется ориентированный граф, вершины которого – состояния автомата, а дуги помечены элементами алфавита.

Начальное состояние помечено входящей дугой, а конечные состояния помечены двойными кружками (см. примеры).

	a	b	*F*
q_0	q_0	q_1	0
q_1	q_2	q_3	0
q_2	q_2	q_3	1
q_3	q_3	q_3	0

Пример 1.

Дан входной алфавит Q={0,1}, цепочка P из Q.
Построить автомат, допускающий все цепочки, содержащие подцепочку «01»

	0	1	F
q_0	q_2	q_0	0
q_1	q_1	q_1	1
q_2	q_2	q_1	0

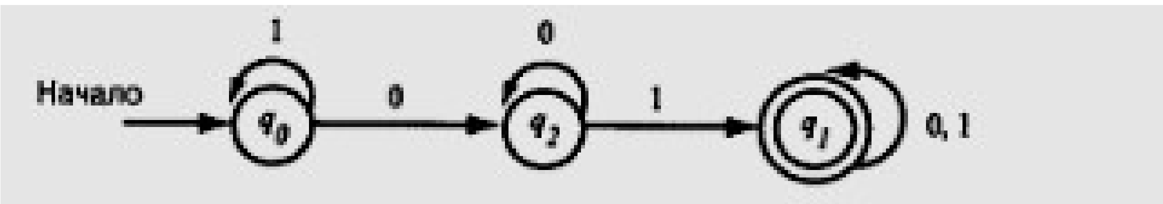


Диаграмма переходов для ДКА, допускающего все цепочки, которые содержат подцепочку 01

Пример 2.

Построить автомат, который допускает язык L = {w|w} , содержащий четное число 0 и 1.

Решение.

Выделим 4 состояния, запоминающих четное и нечетное число 0 и 1:

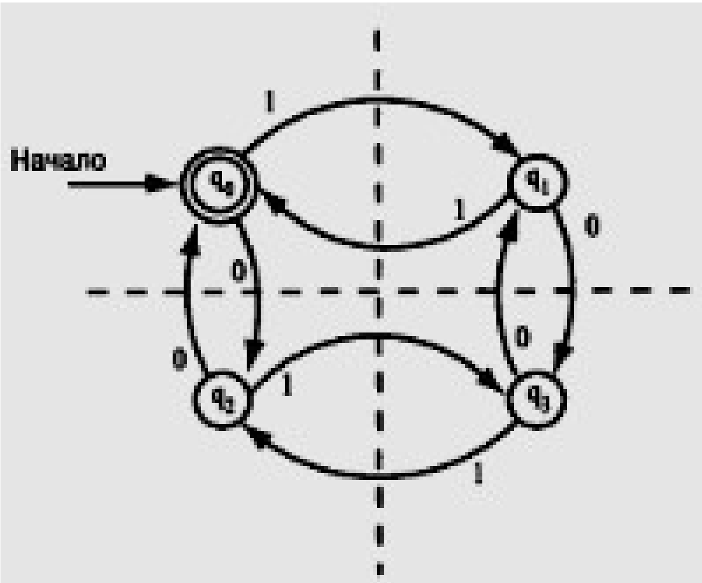
q_0 : Прочитано четное число 0 и четное число 1.

q_1 : Прочитано четное число 0 и нечетное число 1.

q_2 : Прочитано четное число 1 и нечетное число 0.

q_3 : Прочитано нечетное число 0 и нечетное число 1.

Состояние q_0 одновременно допускающее и начальное, т. к. 0 – четное число.



Определение. *Регулярный язык* для некоторого автомата – это множество цепочек, приводящих автомат из начального состояния в одно из допускающих.

Например, для первого автомата регулярный язык - множество цепочек из 0 и 1, содержащих подцепочку 01, для второго автомата язык - множество цепочек из 0 и 1, содержащих четное число 0 и четное число 1.

Таким образом, язык: множество цепочек из 0 и 1, содержащих подцепочку «01» - можно задать распознающим автоматом $A_{01} = (Q,S,\delta,q_0,F)$, где $Q=\{q_0,q_1,q_2\}$; $S=\{0,1\}$; δ - таблица или диаграмма переходов (см. выше); q_0 – начальное состояние; $F=\{q_1\}$ – множество заключительных состояний.

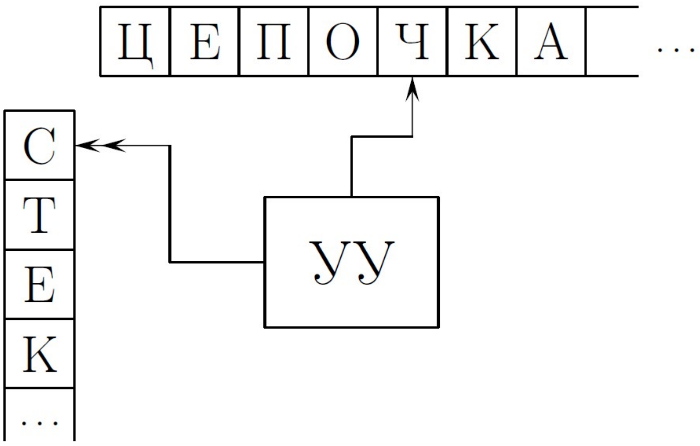
Язык L = {w|w}, содержащий четное число 0 и 1 можно задать распознающим автоматом $A_L = (Q,S,\delta,q_0,F)$, где $Q=\{q_0,q_1,q_2,q_3\}$; $S=\{0,1\}$; δ - диаграмма переходов (см. выше); q_0 – начальное состояние; $F=\{q_0\}$ – множество заключительных состояний.

Контрольные вопросы и задачи

1. Чем отличается расширенная таблица переходов автомата-распознавателя от таблицы переходов конечного автомата Мили?
2. Чем отличается диаграмма переходов автомата-распознавателя от графа конечного автомата Мили?
3. Что такое регулярный язык автомата?
4. Задать язык, содержащий подцепочку «001» распознающим автоматом.
5. Задать язык, содержащий подцепочку «011» распознающим автоматом.

Автомат с магазинной памятью (АМП)

Как известно, распознавателем КС-языков является автомат с магазинной памятью.



Операции автомата:

1. «**Вытолкнуть**» – выталкивает из стека верхний символ (↑).
2. «**Втолкнуть A**» - вталкивает в стек магазинный символ A (↓**A**).
3. «**Заменить XYZ**». Эквивалентна: ↑↓**X**↓**Y**↓**Z** (↕**XYZ**).
4. «**Состояние t**» - переход АМП в другое состояние (**[t]**).
5. «**Сдвиг**» («→») - сдвиг головки на один символ вправо относит. входной ленты.

Переход или шаг автомата – это выполнение операций над стеком и входной головкой, а также изменение состояния.

Автомат определяется:

1. Конечным **множеством входных символов**, включающим концевой маркер (┤).
2. Конечным **множеством магазинных символов**, включающим маркер дна (∇).

3. Конечным **множеством состояний**, включающим начальное состояние.

4. **Программой устройства управления** (УУ), которая каждой комбинации входного символа, магазинного символа и состояния ставит в соответствие выход или переход.

5. **Начальным содержимым магазина**, содержащим маркер дна и, возможно непустую, цепочку магазинных символов.

АМП, как и любой автомат-распознаватель имеет 2 выходных сигнала: «**Допустить**» и «**Отвергнуть**».

При помощи МП-автоматов можно **распознать** бо**льшую часть **конструкций языков программирования**. Рассмотрим некоторые из них.

Задача 1. Распознаватель скобочных выражений

Рассмотрим задачу проверки корректности вложенности круглых скобок.

Определим данный АМП:

1) Множество входных символов: { **(**, **)**, ┤}.

2) Множество магазинных символов: { **A**, ∇}

3) Множество состояний: **t**, является также и начальным состоянием автомата.

4) Алгоритм работы автомата.

- Если входная головка читает «**(**», то в магазин заталкивается символ **A**.
- Если входная головка читает «**)**», то из магазина выталкивается содержащийся там символ.
- Цепочка **принимается**, если при ее окончании всем левым скобкам нашлись правые, то есть при достижении символа ┤ магазин пуст ∇.
- Цепочка **отвергается**, если:

1. Количество правых скобок превысило количество левых, т.е. на входе остаются правые скобки «**)**», а магазин пуст ∇.
2. Входная цепочка прочитана до конца, а левым скобкам не нашлось пары, т.е. при достижении символа ┤ в магазине остаются символы **A**.

Магазинные символы	**Входные символы**		
	(	**)**	**┤**
A	↓**A**, →	↑, →	**Отвергнуть**
∇	↓**A**, →	**Отвергнуть**	**Допустить**

5) В начальном состоянии магазин содержит только маркер дна (∇).

Пример 1: (( ) ( ))

Номер шага	Содержимое стека	Остаток вх. цепочки
1	∇	(( ) ( )) ┤
2	∇**A**	( ) ( )) ┤
3	∇**AA**	) ( )) ┤
4	∇**A**	( )) ┤
5	∇**AA**	)) ┤
6	∇**A**	) ┤
7	∇	┤

Пример 2: ( )))

Номер шага	Содержимое стека	Остаток вх. цепочки
1	∇	( ))) ┤
2	∇**A**	))) ┤
3	∇	)) ┤

Задача 2. Распознаватель арифметических выражений

Определим данный АМП:

1) Множество входных символов: {**x**, +, *, **(**, **)**, ┤}.
2) Множество магазинных символов: { **A, B,** ∇}
3) Множество состояний: **t**, является также и начальным состоянием автомата.
4) Алгоритм работы автомата.

- Цепочка **принимается**, если при ее окончании всем левым скобкам нашлись правые, для всех арифметическим знаков нашлись соответствующие «**x**».
- Цепочка **отвергается**, если:

1. Количество правых скобок превысило количество левых.
2. Количество «**x**» превысило количество арифметических знаков более чем на единицу.
3. Входная цепочка прочитана до конца, а левым скобкам не нашлось пары.
4. Входная цепочка прочитана до конца, а некоторым арифметическим знакам не нашлось соответствующих «**x**».

Алгоритм работы:

- Если входная головка читает «**(**», то в магазин заталкивается **A**.
- Если входная головка читает «**)**», то из магазина выталкивается **A**.
- Если входная головка читает «**+**» или «*****», то в магазин заталкивается **B**.
- Если входная головка читает «**x**», то из магазина выталкивается **B**.
- Цепочка **принимается**, если при достижении символа ┤ магазин пуст ∇.
- Цепочка **отвергается**, если:

1. На входе остаются правые скобки «**)**» или «**x**», а магазин пуст ∇.

2. При достижении символа ⊣ в магазине остаются символы **A** или **B**.

Магазинные символы	**Входные символы**				
	+, *	**x**	**(**	**)**	**⊣**
A	**↓B, →**	**Отвергнуть**	**↓A, →**	**↑, →**	**Отвергнуть**
B	**↓B, →**	**↑, →**	**↑↓A↓B, →**	**Отвергнуть**	**Отвергнуть**
∇	**↓B, →**	**Отвергнуть**	**↓A, →**	**Отвергнуть**	**Допустить**

5) В начальном состоянии магазин содержит (**B**∇).

Пример 1: x+x*(x+x)

Номер шага	**Содержимое стека**	**Остаток вх. цепочки**
1	∇**B**	**x+x*(x+x) ⊣**
2	∇	**+x*(x+x) ⊣**
3	∇**B**	**x*(x+x) ⊣**
4	∇	***(x+x) ⊣**
5	∇**B**	**(x+x) ⊣**
6	∇**AB**	**x+x) ⊣**
7	∇**A**	**+x) ⊣**
8	∇**AB**	**x) ⊣**
9	∇**A**	**) ⊣**
10	∇	**⊣**

Пример 2: x+x*(+x)

Номер шага	**Содержимое стека**	**Остаток вх. цепочки**
1	∇**B**	**x+x*(+x) ⊣**
2	∇	**+x*(+x) ⊣**
3	∇**B**	**x*(+x) ⊣**
4	∇	***(+x) ⊣**
5	∇**B**	**(+x) ⊣**
6	∇**AB**	**+x) ⊣**
7	∇**ABB**	**x) ⊣**
8	∇**AB**	**) ⊣**

Теорема: класс языков, допускаемых МП-автоматами как по заключительному состоянию так и по пустому магазину совпадает с классом контекстно-свободных (КС) языков.

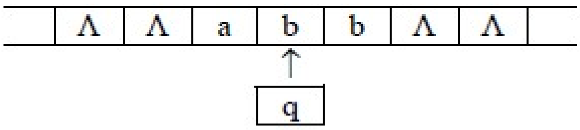
Машина Тьюринга как универсальный распознаватель

Как известно, с помощью МТ можно задать очень широкий класс языков, называемый рекурсивно перечислимым.

Определение. Рекурсивно перечислимый язык — это формальный язык, для которого существует машина Тьюринга (или другая вычислимая функция), которая остановится и примет любую входную строку из этого языка.

Все регулярные, контекстно-свободные, контекстно-зависимые и рекурсивные языки являются рекурсивно перечислимыми.

Если к модели КА добавить неограниченную внешнюю память, то получим автомат, реализующий любой алгоритм. Такой автомат называется *Машиной Тьюринга* (МТ).



МТ состоит из 2 частей: ленты и конечного автомата. Будем считать, что в МТ-распознавателе лента неподвижна, а головка движется относительно ленты под управлением автомата (влево, вправо, стоит на месте).

МТ-распознаватель может выполнять следующие команды:
1) записывать в ячейку ленты новый символ;
2) сдвигаться на одну ячейку влево или вправо;
3) переходить в новое внутреннее состояние.
Больше МТ-распознаватель ничего не может.

МТ-распознаватель задается: $МТ = (Q, S, Г, \delta, q_0, F)$,

1) набором внутренних состояний $Q=\{q_1 \ldots q_m\}$;
2) входным алфавитом $S=\{S_1 \ldots S_n)$;
3) алфавитом допустимых символов на ленте Г;
4) начальным состоянием q_0;
5) множеством заключительных состояний F;
6) программой управления δ, которую можно задавать как в текстовой, так и в табличной форме:

$\delta(q_i, Г_i) = (q', Г', \{L,R,N\})$

или

	S_1	S_2	…	S_i	…	S_n	Λ
q_1							
…							
q_i				S', [L, R, N], q'			
…							
q_m							

где {L,R,N} – команды движения головки.

Будем предполагать, что МТ, распознающая язык L, останавливается, т.е. не имеет никакого следующего движения когда входная цепочка принимается.

Пример. Задать язык $L = \{0^n1^n \mid n \geq 1\}$.

Положим МТ = (Q, S, Γ, δ, q_0, F),

где Q = {q0, q1, q2, q3, q4, q5};
S = {0,1};
Γ = {0, 1, B, X, Y};
q_0 = q0;
F = {q5}.

Головка находится на первом символе цепочки. При работе МТ-распознавателя все нули будут заменены на X, а единицы – на Y. Если МТ встретит пробел B при одинаковом количестве X и Y, то цепочка языка будет принята.

Программу МТ δ определим следующим образом:

1. δ(q0, 0) = (q1, X, R).

В состоянии q0 символ 0 заменяется на X и головка сдвигается вправо в состояние q1 в поисках 1.

2. а) δ(q1, 0) = (q1, 0, R);
б) δ(q1, Y) = (q1, Y, R);
в) δ(q1, 1) = (q2, Y, L).

Оставаясь в состоянии q1, машина продвигается вправо сквозь все нули (п. 2а) и блок Y (п. 2б). Наткнувшись на 1, заменяет ее на Y и переходит в состояние q2, начав движение влево (п. 2в).

3. а) δ(q2, Y) = (q2, Y, L);
б) δ(q2, X) = (q3, X, R);

в) δ(q2, 0) = (q4, 0, L).

Оставаясь в состоянии q2, машина продвигается влево сквозь блок Y (п. 3а). Если машина встречает X, все еще оставаясь в состоянии q2, то больше нет нулей, которые следовало бы заменять на X, и машина переходит в состояние q3, начиная движение вправо, чтобы убедиться, что не осталось единиц (п. 3б). Если же 0 встретился, машина переходит в состояние q4, чтобы продолжить движение в поисках крайнего левого 0 (п. 3в).

4. а) δ(q4, 0) = (q4, 0, L)
   б) δ(q4, X) = (q0, X, R).

Машина движется сквозь нули (п. 4а). Если встретился X, то машина прошла самый левый нуль. Она должна, сдвинувшись вправо, превратить этот 0 в X (п. 4б). Происходит переход в состояние q0, и процесс, только что описанный в п. 1–4, продолжается.

5. а) δ(q3, Y) = (q3, Y, R)
   б) δ(q3, B) = (q5, Y, R).

Машина входит в состояние q3, когда ни одного 0 не остается (см. п. 3б). Машина должна продвигаться вправо (п. 5а) сквозь блок Y. Если встречается пробел B раньше, чем 1, то ни одной 1 не осталось (п. 5б). В этой ситуации машина переходит в конечное состояние q5 и останавливается, сигнализируя тем самым прием входной цепочки.

6. Во всех случаях, кроме 1–5, функция δ не определена. МТ остановится и отвергнет входную цепочку.

Табличная запись функции δ.

	0	1	X	Y	B
q0	q1, X, R	отв.	отв.	отв.	отв.
q1	q1, 0, R	q2, Y, L	отв.	q1, Y, R	отв.
q2	q4, 0, L	отв.	q3, X, R	q2, Y, L	отв.
q3	отв.	отв.	отв.	q3, Y, R	**q5**, Y, R
q4	q4, 0, L	отв.	q0, X, R	отв.	отв.

Рассмотрим действия машины Тьюринга на входной цепочке 000111.

Шаг	Конфигурация	Шаг	Конфигурация	Шаг	Конфигурация
1	0 0 0 1 1 1 q_0	10	X X 0 Y 1 1 q_1	19	X X X Y Y Y q_2
2	X 0 0 1 1 1 q_1	11	X X 0 Y 1 1 q_1	20	X X X Y Y Y q_2
3	X 0 0 1 1 1 q_1	12	X X 0 Y Y 1 q_2	21	X X X Y Y Y q_2
4	X 0 0 1 1 1 q_1	13	X X 0 Y Y 1 q_2	22	X X X Y Y Y q_3
5	X 0 0 Y 1 1 q_2	14	X X 0 Y Y 1 q_4	23	X X X Y Y Y q_3
6	X 0 0 Y 1 1 q_4	15	X X 0 Y Y 1 q_0	24	X X X Y Y Y q_3
7	X 0 0 Y 1 1 q_4	16	X X X Y Y 1 q_1	25	X X X Y Y Y q_3
8	X 0 0 Y 1 1 q_0	17	X X X Y Y 1 q_1	26	X X X Y Y Y Y q_5
9	X X 0 Y 1 1 q_1	18	X X X Y Y 1 q_1		

В таблице приведены конфигурации в виде цепочек символов ленты с маркером состояния под сканируемым символом (в конфигурациях 25 и 26 маркер состояния находится под символом пробела).

В некоторых задачах МТ может использоваться в роли **запоминающего устройства**, для запоминания конечного количества информации. Именно: состояние записывается как пара элементов, причем один осуществляет управление, а другой запоминает символ.

Пример. Задать язык, состоящий из цепочек, в которых первый символ повторно не встречается в той же самой цепочке.

Пусть МТ = (Q, {0, 1},{0, 1, B}, δ, q_0, F),

где Q = {[q0, 0],[q0, 1], [q0, B], [q1, 0], [q1, 1], [q1, B]}, т.е. {q0, q1} × {0, 1, B},
q_0 = [q0, B] ,
F = {[q1, B]}.

Построим функцию δ (программу МТ) следующим образом:

1. а) δ([q0, B], 0) = ([q1, 0], 0, R);
б) δ([q0, B], 1) = ([q1, 1], 1, R).

Машина запоминает сканируемый символ во второй компоненте обозначения состояния и сдвигается вправо. Первой компонентой становится q1.

2. а) $\delta([q1, 0], 1) = ([q1, 0], 1, R)$;
   б) $\delta([q1, 1], 0) = ([q1, 1], 0, R)$.

Если машина помнит 0 и видит 1 или, наоборот, помнит 1 и видит 0, то она продолжает движение вправо.

3. а) $\delta([q1, 0], B) = ([q1, B], 0, L)$;
   б) $\delta([q1, 1], B) = ([q1, B], 0, L)$.

Машина входит в конечное состояние [q1, B], если она встречает символ пробела раньше, чем достигает второй копии самого левого символа. Если же машина достигает пробела в состоянии [q1, 0] или [q1, 1], то она принимает входную цепочку. Для состояния [q1, 0] и символа 0 или для состояния [q1, 1] и символа 1 функция δ не определена, так что, если машина когда-нибудь видит запомненный символ, она останавливается, не принимая.

Табличная запись функции δ.

	0	1	B
[q0,B]	[q1, 0], 0, R	[q1, 1], 1, R	[q0,B], B, R
[q1,0]	отв.	[q1, 0], 1, R	**допущена**
[q1,1]	[q1, 1], 0, R	отв.	**допущена**

В общем случае можно допустить произвольное фиксированное число компонент [q1,k], которые предназначены для запоминания информации.

Контрольные вопросы и задачи

1. Как задать язык распознающим автоматом с магазинной памятью?
2. Как задать язык распознающей машиной Тьюринга?
3. Задать язык списков типа a;[a;a;a;] распознающим АМП. Показать работу АМП на правильной и неправильной цепочках.
4. Задать язык палиндромов в алфавите {0,1} распознающей машиной Тьюринга. Программу МТ записать в таблице. Показать работу МТ на цепочке 010010.