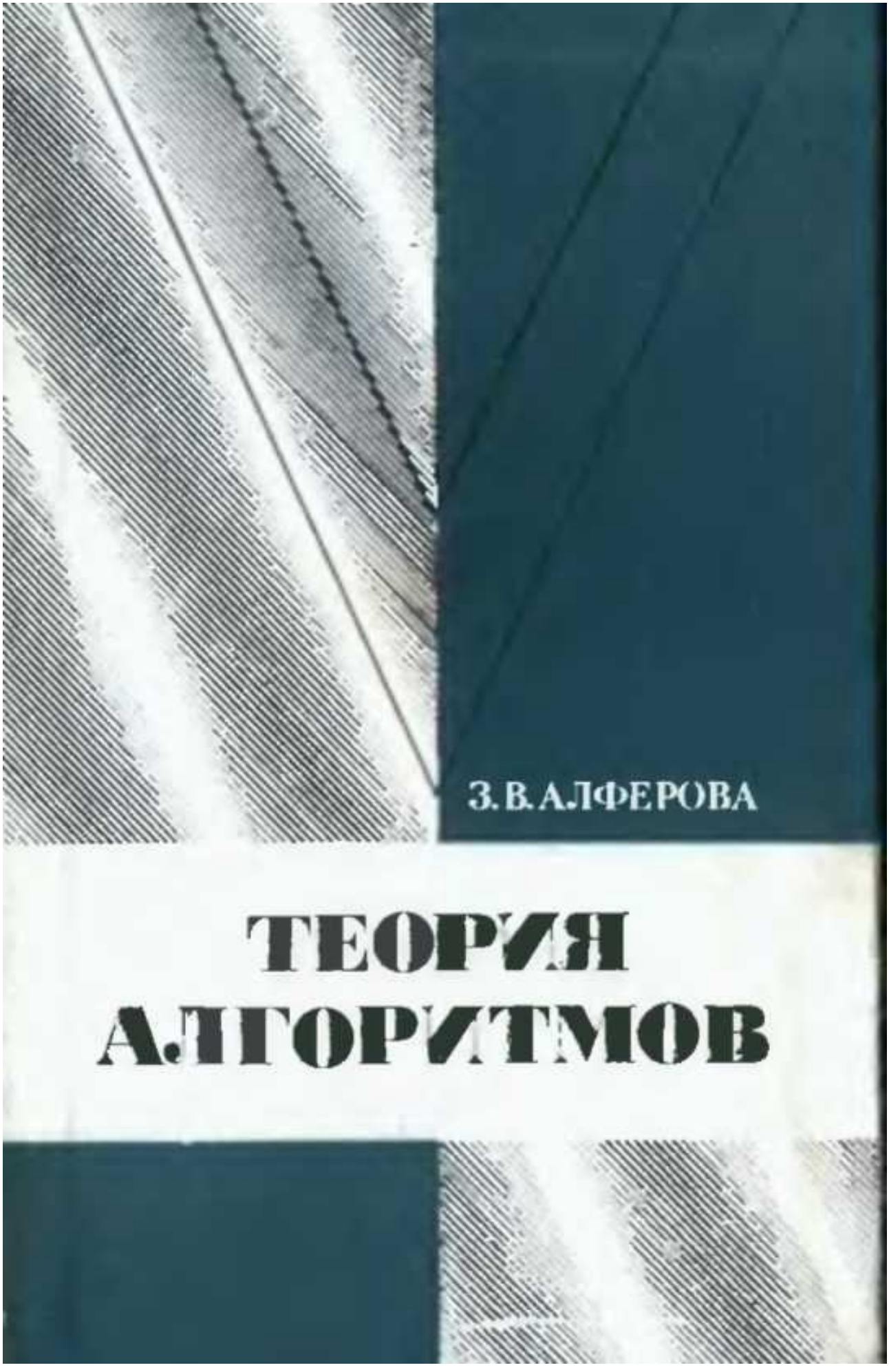

З. В. АЛФЕРОВА
ТЕОРИЯ
АЛГОРИТМОВ

ОГЛАВЛЕНИЕ

З. В. АЛФЕРОВА

ТЕОРИЯ АЛГОРИТМОВ

Допущено Министерством высшего и среднего специального образования СССР в качестве учебного пособия для студентов вузов, обучающихся по специальности "Организация механизированной обработки экономической информации"

«СТАТИСТИКА»
1973 Москва

В учебном пособии излагаются основы теории алгоритмов и теории формальных грамматик, рассматриваются различные алгоритмические системы, методы оценки и преобразования алгоритмов, связь теории алгоритмов с теорией формальных грамматик, классификация грамматик, связь теории формальных грамматик с теорией автоматов.

Пособие предназначено для студентов вузов, специализирующихся по механизированной обработке экономической информации. Им могут пользоваться и специалисты, работающие в этой области.

Замечания и пожелания по учебному пособию просим посылать по адресу: Москва, Б. Савинский пер., д. 14. Московский экономико-статистический институт, кафедра математического обеспечения ЭВМ.

ВВЕДЕНИЕ

Понятие алгоритма является одним из основных понятий современной математики. Еще на самых ранних ступенях развития математики (Древний Египет, Вавилон, Греция) в ней стали возникать различные вычислительные процессы чисто механического характера. С их помощью искомые величины ряда задач вычислялись последовательно из исходных величин по определенным правилам и инструкциям. Со временем все такие процессы в математике получили название алгоритмов (алгорифмов).

Термин *алгоритм* происходит от имени средневекового узбекского математика Аль-Хорезми, который еще в IX в. (825 г.) дал правила выполнения четырех арифметических действий в десятичной системе счисления. Процесс выполнения арифметических действий был назван *алгоризмом.*

С 1747 г. вместо слова *алгоризм* стали употреблять *алгорисмус*, смысл которого состоял в комбинировании четырех операций арифметического исчисления — сложения, вычитания, умножения, деления.

К 1950 г. *алгорисмус стал алгорифмом.* Смысл алгорифма чаще всего связывался с алгорифмами Евклида — процессами нахождения наибольшего общего делителя двух многочленов, наибольшей общей меры двух отрезков и т. п.

Вплоть до 30-х годов понятие алгоритма имело скорее методологическое, чем математическое значение. Под алгоритмом понимали конечную совокупность точно сформулированных правил, которые позволяют решать те или иные классы задач. Такое определение алгоритма не является строго математическим, так как в нем не содержится точной характеристики того, что следует понимать под классом задач и под правилами их решения. В течение длительного времени математики довольствовались этим определением, поскольку общей теории алгоритмов фактически не существовало. Однако практически не было серьезных случаев, когда математики разошлись бы во мнениях относительно того, является ли алгоритмом тот или иной конкретно заданный процесс.

Положение существенно изменилось, когда на первый план выдвинулись такие алгоритмические проблемы, положительное решение которых было сомнительным. Действительно, одно дело доказать существование алгоритма, другое — доказать отсутствие алгоритма. Первое можно сделать путем фактического описания процесса, решающего задачу. В этом случае достаточно и интуитивного понятия алгоритма, чтобы удостовериться в том, что описанный процесс есть алгоритм. Доказать несуществование алгоритма таким путем невозможно. Для этого надо точно знать, что такое алгоритм.

В двадцатых годах нашего века задача такого определения понятия алгоритма стала одной из центральных математических проблем. Решение ее было получено в середине 30-х годов в работах известных математиков Гильберта, Гёделя, Черча, Клини, Поста и Тьюринга в двух формах. Первое решение было основано на понятии особого класса арифметических функций, получивших название рекурсивных функций, второе — на описании точно очерченного класса процессов. Впоследствии в работах Маркова, Калужнина появилось другое толкование теории алгоритмов, поставившее в основу определение алгоритма как особого соответствия между словами в том или ином абстрактном алфавите.

Первоначально теория алгоритмов возникла в связи с внутренними потребностями теоретической математики. Математическая логика, основания математики, алгебра, геометрия и анализ остаются и сегодня одной из основных областей приложения теории алгоритмов.

Другая ее область возникла в 40-х годах в связи с созданием быстродействующих электронных вычислительных и управляющих машин. Появление ЭВМ способствовало развитию теории алгоритмов, вызвало к жизни разделы этой теории, имеющие ярко выраженную прикладную направленность. Это прежде всего алгоритмические системы и

алгоритмические языки, являющиеся основой современной теории программирования для универсальных ЭВМ, и способы точного описания отображений, реализуемых цифровыми автоматами.

Наконец, теория алгоритмов оказалась тесно связанной и с рядом областей лингвистики, экономики, физиологии мозга и психологии, философии, естествознания. Примером одной из задач этой области может служить точное описание алгоритмов, реализуемых человеком в процессе умственной деятельности (в любой из областей его деятельности).

Чтобы иметь возможность более уверенно решать алгоритмические задачи, возникающие в различных отделах теоретической и прикладной математики, необходимо иметь достаточно развитую теорию алгоритмов. В настоящее время такая теория алгоритмов уже создана. Изучение этой теории и является целью данного курса.

В процессе изучения курса «Теория алгоритмов» целесообразно определить понятие алгоритма, рассмотреть такие алгоритмические системы, как рекурсивные функции, машины Тьюринга, нормальные алгоритмы Маркова и др.; исследовать связь теории алгоритмов с теорией автоматов и с универсальными электронными вычислительными машинами; изучить теоретические основы построения и анализа алгоритмических языков, формальных преобразований и оценки алгоритмов.

Теоретические положения курса «Теория алгоритмов» являются основой для успешного освоения серии специальных дисциплин, которые будут изучаться на последующих курсах.

Так, при составлении алгоритма конкретной задачи актуальное значение имеет такое представление алгоритма, которое позволяет наиболее быстро реализовать его механизированным путем, и в частности с помощью ЭВМ. Известно, что для постановки задачи на ЭВМ ее необходимо запрограммировать, т.е. представить алгоритм решения задачи в виде последовательности команд, которые может выполнять машина.

Однако процесс программирования очень длительный, трудоемкий и его также можно автоматизировать, если использовать для записи алгоритмов алгоритмические языки, представляющие собой набор символов и терминов, связанных синтаксической структурой. С помощью алгоритмических языков можно по определенным правилам описывать алгоритмы решения задач. Алгоритмические языки и правила их использования излагаются в курсе «Алгоритмические языки». Алгоритмы, записанные в алгоритмическом языке, автоматически самой ЭВМ с помощью специальной программы «Транслятор» могут быть переведены в машинные программы для конкретной ЭВМ. С программами-трансляторами, их строением и использованием знакомит курс «Математическое обеспечение ЭВМ».

Во всех этих курсах используются теоретические основы теории алгоритмов. Отсюда вытекает то значение курса, которое он занимает как теоретическая основа современной теории программирования, построения алгоритмических языков и ЭВМ, анализа алгоритмов с целью выбора наиболее рационального для решения на ЭВМ и, наконец, анализа алгоритмических языков и их синтаксического контроля при разработке трансляторов.

Глава 1.АЛГОРИТМИЧЕСКИЕ СИСТЕМЫ

1.1. Основные понятия теории алгоритмов

Под алгоритмом с давних времен понимается конечная совокупность точно сформулированных правил решения некоторого класса задач.

Первоначально для записи алгоритмов пользовались средствами обычного языка. И поэтому изучение теории алгоритмов целесообразно начать со словесного представления алгоритмов.

Уточним понятие словесного алгоритма на примере умножения я чисел a_1, a_2, ... ,a_n, т.е. вычисления по формуле

$$c = \prod_{i=1}^{n} a_i.$$

Этот процесс может быть записан в виде следующей системы последовательных указаний:

1. Полагаем c равным единице и переходим к следующему указанию.
2. Полагаем i равным единице и переходим к следующему указанию.
3. Полагаем c равным c-a_i и переходим к следующему указанию.
4. Проверяем, равно ли i числу n. Если $i = n$, то вычисления прекращаем. Если $i <$ n, то увеличиваем i на единицу и переходим к 3 указанию.

Рассмотрим еще пример алгоритма нахождения минимального числа x в массиве из n чисел a_1, a_2, ... ,a_n. Прежде чем записать словесный алгоритм данного примера, детально рассмотрим сам процесс поиска минимального числа. Будем считать, что процесс поиска осуществляется следующим образом. Первоначально в качестве числа x принимается a_1, т.е. полагаем $x = a_1$, после чего x сравниваем с последующими числами массива, начиная с a_2. Если $x < a_2$, то x сравнивается с a_3, если $x < a_3$, то x сравнивается с a_4, и так до тех пор, пока найдется число $a_i <$ x. Тогда полагаем $x = a_i$ и продолжаем сравнение с x последующих чисел из массива, начиная с a_{i+1}, и так до тех пор, пока не будут просмотрены n чисел. В результате просмотра всех n чисел x будет иметь значение, равное наименьшему числу из массива (i - текущий номер числа из массива). Этот процесс может быть записан в виде следующей системы последовательных указаний:

1. Полагаем $i = 1$ и переходим к следующему указанию.
2. Полагаем $x = a_i$ и переходим к следующему указанию.
3. Сравниваем i с n; если $i < n$, переходим к 4 указанию, если $i = n$, процесс поиска останавливаем.
4. Увеличиваем i на единицу и переходим к следующему указанию.
5. Сравниваем a_i с x. Если $a_i \geq x$, то переходим к 3 указанию, если $a_i < x$ (иначе), переходим ко 2 указанию.

В первом алгоритме в качестве элементарных операций используются простейшие арифметические операции умножения, которые могли бы быть разложены на еще более элементарные операции. Мы такого разбиения не делаем в силу простоты и привычности арифметических правил.

Алгоритмы, в соответствии с которыми решение поставленных задач сводится к арифметическим действиям, называются численными алгоритмами.

Алгоритмы, в соответствии с которыми решение поставленных задач сводится к логическим действиям, называются логическими алгоритмами. Примерами логических алгоритмов могут служить алгоритмы поиска минимального числа, поиска пути на графе, поиска пути в лабиринте и др.

Рассмотрим более сложный пример логического алгоритма - процесс поиска пути из вершины x_0 в вершину x_n для графа типа дерева.

В общем случае процесс поиска пути на графе можно представить следующим образом. Выйдя из вершины x_0, идем по какой-либо ветви до тех пор, насколько это возможно. Затем, проверив, что конечная вершина этой ветви не является вершиной x_n, возвращаемся в ближайший узел и отправляемся по новому, еще неизведанному направлению. Такой поиск продолжается до тех пор, пока не натолкнемся на вершину x_n. Искомый путь из x_0 в x_n будет состоять из всех тех ребер, которые в процессе поиска были пройдены ровно по одному разу.

Прежде чем записать алгоритм в виде указаний, введем ряд обозначений. Для каждой вершины x_i введем нумерацию смежных ей ребер. Нумерация производится в направлении по часовой стрелке, начиная с ребра, по которому пришли в эту вершину x_i (или некоторое произвольное ребро для начальной вершины x_0): $u_1, u_2, \ldots, u_j, \ldots, u_k$, где $k = P(x_i)$, u_1 - ребро, по которому пришли в x_i:

$$U_{xi} = \{u_1, u_2, \ldots, u_k\}.$$

Для каждого ребра u_j введем некоторую характеристику $H = \{0, 1, 2\}$, такую, что

$H(u_j) = 0$, если ребро u_j не проходилось ни разу;

$H(u_j) = 1$, если u_j проходилось один раз в любом направлении;

$H(u_j) = 2$, если u_j проходилось дважды: один раз в одном направлении и второй раз - в обратном.

При $H(u_j) = 2$ ребро считается «закрытым».

Для каждой вершины введем полустепень характеристики $H = P^H(xi)$, такую, что

$$P(x_i) = P^0(\gamma_i) + P^1(x_i) + P^2(x_i),$$

где $P^0(x_i)$ — число ребер с характеристикой $H = 0$;

$P^1(x_i)$ — число ребер с характеристикой $H = 1$;

$P^2(x_i)$ — число ребер с характеристикой $H = 2$.

В исходном состоянии каждое ребро u_j любой вершины x_i имеет характеристику $H=0$, степень каждой вершины x_i

$$P(x_i) = P^0(x_i),\ P^1(x_i) = P^2(x_i) = 0.$$

Поиск пути начинается с вершины x_0 , т.е. $i = 0$. С учетом принятых обозначений укрупненный алгоритм поиска пути на графе можно записать следующим образом:

1. Выбрать вершину x_i со всеми ее характеристиками. Перейти к следующему указанию.

2. Сравнить x_i с x_n Если $x_i = x_n$, поиск закончить. Путь найден и проходит по ребрам с H = 1. Если $x_i = x_n$, перейти к следующему указанию.

3. Сравнить $P(x_i)$ с $P(x_i)$. Если $P(x_i)=P^2(x_i)$, поиск закончить. Это означает, что все ребра имеют характеристику $H = 2$ и пути из x_0 в x_n не существует. Если $P(x_i) <> P^2(x_i)$, перейти к следующему указанию.

4. Сравнить $P(x_i)$ с $P^1(x_i)$. Если $P(x_i)=P^1(x_i)$, возвратиться по дуге u_1 заменив при этом $H(u_1)= 1$ на $H(u_1) = 2$ (u_1— ребро, по которому пришли в x_i). Граничную для x_i вершину по ребру принять за текущую x_i. Перейти к 1 указанию. Если $P(x_i)<>P^1(x_i)$, перейти к следующему указанию.

5. Увеличить j на единицу и перейти к следующему указанию.

6. Сравнить $H(u_j)$ с нулем. Если $H(u_j) = 0$, то пойти по ребру u_j, заменив $H(u_j) = 0$ на $H(u_j)=1$. Граничную для x_i вершину по ребру u_j принять за текущую x_i. Перейти к первому указанию. Если $H(u_j)<>0$, перейти к указанию 3.

Можно составить более детальный алгоритм с моментами сложения и изменения P^0, P^1, P^2 и j. Вариант такого детального алгоритма предлагается составить самостоятельно.

Алгоритмами в современной математике принято называть конструктивно задаваемые соответствия между словами в абстрактных алфавитах.

Чтобы перейти к определению алгоритма, прежде всего следует определить понятия абстрактного алфавита и слов в этих алфавитах.

Абстрактным алфавитом называется любая конечная совокупность объектов, называемых буквами или символами данного алфавита. При этом природа этих объектов нас совершенно не интересует. Символом абстрактных алфавитов можно считать, например, буквы алфавита какого-либо языка, цифры, любые значки, рисунки и даже слова некоторого конкретного языка. Важно лишь, чтобы рассматриваемый алфавит был конечным.

Можно сказать, что алфавит — конечное множество различимых символов. Слово «абстрактный» для краткости будем опускать. Алфавит, как любое множество, задается перечислением его элементов, т.е. символов.

Примеры алфавитов: $A = \{a, \beta, \gamma, \backsim\}$, B= $\{x, y\}$.

Под словом или строкой алфавита будем понимать любую конечную упорядоченную последовательность символов.

Так, например, в алфавите A словами следует считать любые последовательности a, $a\gamma$, $\gamma\beta$, $\backsim\backsim\beta$, $\beta\beta$ и т. п., в алфавите B— x, y, xy, yx, xx, yy и т. п.

Число символов в слове называется длиной этого слова. Так, слова из алфавита A, приведенные в примере, имеют длину соответственно 1, 2, 2, 3, 2, ...

Наряду со словами положительной длины, состоящими не менее чем из одного символа, в ряде случаев целесообразно рассматривать также пустые слова, не содержащие, ни одного символа. Обычно для обозначения пустого слова употребляется малая латинская буква e. Слово p называется полсловом слова q, если слово q можно представить в виде $q = pr$, где r — любое слово, в том числе и пустое.

Очевидно, что такое понятие слова будет отличаться от понятия слова, принятого в обычном языке. При нашем определении словами следует считать любые сочетания и последовательности символов, в том числе и бессмысленные.

При расширении алфавита, т.е. при включении в его состав новых символов, понятие слова может претерпеть существенное изменение. Так, например, в алфавите A = {0, 1, 2, 3, 4, 5, 6, 7,

8, 9} выражение «69 + 73» представляет собой два слова, соединенные знаком суммы, а в алфавите $A' = \{+, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ это будет одно слово.

Кроме знаков препинания и знаков пробела в качестве символов могут использоваться отдельные слова, фразы, абзацы и даже целые книги.

Алфавитным оператором или алфавитным отображением называется всякое соответствие, сопоставляющее словам некоторого алфавита слова в том же самом или в каком-то другом фиксированном алфавите. При этом первый алфавит называется входным, второй — выходным алфавитом данного оператора. В случае совпадения входного и выходного алфавитов говорят, что алфавитный оператор задан в соответствующем алфавите.

Иначе говоря, алфавитный оператор — функция, задающая соответствие между словами входного алфавита и словами этого же или другого выходного алфавита.

Пусть нам заданы слова в алфавитах A и B и заданы соответствия между этими словами (рис. 1).

Если α — слово в алфавите A, а β — слово в алфавите B, то алфавитный оператор $\Gamma\alpha = \beta$ «перерабатывает» входное слово α в выходное слово β.

Буква Г в алфавитном операторе означает отображение.

Различают *однозначные и многозначные алфавитные операторы.*

Под однозначным алфавитным оператором понимается такой алфавитный оператор, который каждому входному слову ставит в соответствие не более одного выходного слова (рис. *2)*.

Алфавитный оператор, не сопоставляющий данному входному слову a_i никакого выходного слова b_j (в том числе и пустого), не определен на этом слове.

Совокупность всех слов, на которых алфавитный оператор определен, называется его областью определения.

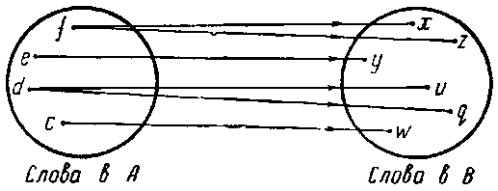


Рис. 1

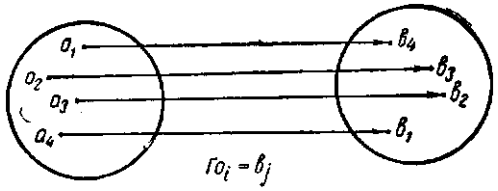


Рис. 2

С каждым алфавитным оператором связывается интуитивное представление о его сложности. Наиболее простыми являются алфавитные операторы, осуществляющие посимвольные отображения. Посимвольное отображение состоит в том, что каждый символ s входного слова a заменяется некоторым символом выходного алфавита B. Большое значение имеют так называемые *кодирующие отображения.* Под кодирующим отображением

понимается соответствие, сопоставляющее каждому символу входного алфавита некоторую конечную последовательность символов в выходном алфавите, называемую *кодом.*

Рассмотрим пример кодирующего отображения. Заданы алфавиты:

$A = \{p, r, s, t\}$ — входной;

$B = (a, b, c, d, f, g, h, m, n, q)$ — выходной; отображения символов A символами B (рис. 3).

Рис. 3

Для построения искомого кодирующего отображения достаточно заменить все символы любого слова a_i в алфавите A соответствующими кодами алфавита B.

Пусть задано слово $a = sstr$, тогда $\Gamma a = fghfghabcdmn$. Полученное таким образом слово в алфавите B называется кодом исходного слова a.

Процесс, обратный кодированию, т.е. замена в слове b_j кодов алфавита B символами из алфавита A, называется декодированием и обозначается $\Gamma^{-1} b_j = a_i$.

Например, для слова $b = fghmnqdbfgh$ в алфавите B декодирование $\Gamma^{-1} b$ дает слово $a = srps$.

Если при кодировании слова a_i получаем некоторое слово b_j, а декодирование дает исходное слово $a_i (\Gamma a_i = b_j, \Gamma^{-1} b_j = a_i)$, то имеет место обратимость кодирования. Условие обратимости кодирования есть не что иное, как условие взаимной однозначности соответствующего кодирующего отображения.

В приведенном выше примере обратимость имеет место. Рассмотрим еще такой пример: даны $A = \{a, b, c\}$, $B = \{\alpha, \beta\}$, $\Gamma a = \alpha$, $\Gamma b = \beta$, $\Gamma c = \alpha\beta$ и слово *aabca* в алфавите А.

$\Gamma aabca$ — $\alpha\alpha\beta\alpha\beta\alpha$..

$\Gamma^{-1} \alpha\alpha\beta\alpha\beta\alpha = aababa$ (либо одно из слов *acaba*, *aabca*, *acca)*, т.е. обратимость отсутствует.

Для обратимости кодирования должны выполняться два следующих условия:

1. Коды разных символов исходного алфавита A должны быть различны.

2. Код любого символа алфавита A не может совпадать ни с каким из начальных подслов кодов других символов этого алфавита.

В самом деле, предположим, что оба эти условия выполнены и пусть слово $q = b_1, b_2, ..., b_n$ является кодом слова $p = a_1, a_2, ..., a_m$ в алфавите A. Покажем, что по коду q можно однозначно восстановить слово p. В силу условия 2 только одно начальное подслово слова q будет совпадать с каким-либо символом алфавита A. Ясно, что таким подсловом является символ a_1. Применяя аналогичные рассуждения к оставшемуся отрезку, однозначно восстановим все символы один за другим. Следовательно, любому данному коду может соответствовать только одно слово в A, чем и доказана обратимость кодирующего отображения. Следует отметить, что условие 2 выполняется в том случае, если коды всех символов исходного алфавита A имеет одинаковую длину. Кодирование, при котором все коды имеют одинаковую длину, называется *нормальным.*

Кодирование позволяет сводить изучение произвольных алфавитных отображений к алфавитным отображениям в некотором стандартном алфавите. Наиболее часто в качестве такого стандартного алфавита выбирается так называемый двоичный алфавит, состоящий из двух символов, которые обычно отождествляют с цифрами 0 и 1 — $C = \{0, 1\}$.

Пусть A — произвольный, а C — стандартный (двоичный) алфавиты, состоящие более чем из одного символа.

Если n — число символов в алфавите А, а m — число символов в алфавите С, то всегда можно выбрать длину слова l так, чтобы удовлетворялось условие $m^l \geq n$.

Поскольку число различных слов длины l в m-символьном алфавите равно m^l, то все символы в алфавите А можно закодировать словами длины l в алфавите С так, чтобы коды различных букв были разными. Любое такое кодирование будет нормальным и порождает обратимое кодирующее отображение слов в алфавите А в слова в алфавите С.

Обозначим это отображение через $\Gamma a = c$, а обратное ему отображение через $\Gamma^{-1} c = a$, где a — слово *в* А, а с — слово в алфавите С. Пусть φa — произвольный оператор в алфавите А такой, что $\varphi a = a'$, а φc — произвольный алфавитный оператор в алфавите С такой, что $\varphi c = c'$. Тогда отображение

$$\varphi c = \Gamma^{-1} c, \varphi a, \Gamma a', \tag{1}$$

получаемое в результате последовательного выполнения отображений $\Gamma^{-1} c$, φa, $\Gamma a'$, будет представлять собой некоторый оператор в стандартном алфавите С. Назовем этот оператор φc алфавитным оператором в алфавите С сопряженным (при помощи кодирования $\Gamma a'$ и декодирования $\Gamma^{-1} c$) с алфавитным оператором φa.

Оператор φa однозначно восстанавливается по сопряженному оператору φc и соответствующим кодирующему Γa и декодирующему $\Gamma^{-1} c'$ отображениям.

$$\varphi a = \Gamma a, \varphi c, \Gamma^{-1} c'. \tag{2}$$

Рассмотрим взаимосвязь сопряженных операторов на графе (стр. 13).

Применение формул (1) и (2) позволяет сводить произвольные алфавитные операторы к алфавитным операторам в стандартном алфавите.

Эти соотношения справедливы и для случая, когда входной алфавит А, выходной алфавит В и стандартный алфавит С различны, т.е. для случая алфавитных операторов, у которых входной и выходной алфавиты различны.

$$\varphi c = \Gamma^{-1} c, \varphi a, \Gamma b;$$

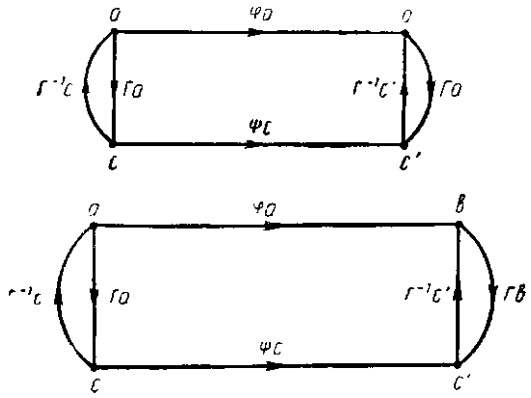
$$\varphi a = \Gamma a, \varphi c, \Gamma^{-1} c'.$$

Понятие алфавитного оператора является чрезвычайно общим. К нему фактически сводятся или могут быть сведены любые процессы преобразования информации.

Под информацией будем понимать всякие сведения о процессах и состояниях любой природы, которые могут восприниматься органами чувств человека или приборами.

К изучению алфавитных операторов фактически сводится теория любых преобразователей информации. А с преобразователями информации человек встречается в своей практике буквально на каждом шагу (приборы, средства связи, сам человек).

Для некоторых специальных видов информации, например информации лексической или числовой, применяется алфавитный способ задания. Преобразования этих видов информации сводятся к алфавитным операторам самым непосредственным образом: как входная, так и выходная информация в любом преобразователе информации в этом случае может быть представлена в виде слов, а преобразование информации сводится к установлению некоторого соответствия между словами.



Одной из характерных задач преобразования лексической информации является перевод текстов с одного языка на другой.

Если считать словами целые книги или хотя бы отдельные абзацы книги, то задача перевода полностью сводится к задаче установления соответствия между такими обобщенными словами. Таким образом, процесс перевода с одного языка на другой может трактоваться как процесс реализации некоторого алфавитного оператора.

Следует заметить, впрочем, что вполне качественный и грамотный перевод допускает, как известно, возможность известных модификаций переводного текста. Поэтому процесс перевода описывается не обычным однозначным алфавитным оператором, а многозначным.

Кроме алфавитных операторов, для перевода с одних языков на другие, можно построить алфавитные операторы, решающие и другие задачи преобразования лексической информации, например задачу редактирования текстов на том или ином языке, задачу составления рефератов статей и т. п.

Аналогичным образом нетрудно представить в виде процессов реализации алфавитных операторов и многие другие процессы преобразования информации, например оркестровку мелодии, решение математических задач, задачи планирования производства и т. п.

Может показаться сначала, что для характеристики преобразований непрерывной информации (например, зрительных или произвольных слуховых ощущений) понятие алфавитного оператора недостаточно. Однако это не совсем так.

Восприятие и преобразование непрерывной информации всегда производятся при помощи приборов, не реагирующих на слишком малые изменения характеристик преобразуемой информации. В реальных приборах, воспринимающих и преобразующих непрерывную информацию, всегда существует ряд ограничений, которые позволяют рассматривать эту информацию как алфавитную.

Первое — это ограничение разрешающей способности прибора, воспринимающего информацию. Это приводит к тому, что достаточно близкие между собой точки участка пространства, на котором распределена рассматриваемая информация, воспринимаются прибором как одна точка. Отсюда вытекает возможность рассматривать эту информацию как информацию, заданную не в бесконечном, а лишь в конечном множестве точек.

Второе ограничение связано с чувствительностью прибора, воспринимающего информацию. Прибор может различать фактически лишь конечное число уровней величины, несущей информацию.

На основании описанных ограничений приходим к выводу, что прибор, вследствие своей неидеальности, может в каждый данный момент воспринимать лишь одну картину из конечного числа различных картин мгновенного распределения рассматриваемой информации в пространстве.

Третье ограничение обусловлено тем, что полоса пропускания прибора не позволяет ему различать слишком быстрые изменения воспринимаемых величин.

После введения дискретного времени информация, воспринимаемая прибором за любой конечный отрезок времени t, представляется в виде слова в некотором алфавите А. Наконец, одним из наиболее важных и существенных аспектов изучения деятельности самого человека является аспект, связанный с рассмотрением человека как весьма сложного и совершенного преобразователя информации.

В любом случае основой теории алфавитных операторов являются способы их задания.

В случае, если область определения алфавитного оператора конечна, оператор может быть задан простой таблицей соответствия. В левой части такой таблицы выписываются все слова, входящие в область определения рассматриваемого оператора, в правой части — выходные слова, получающиеся в результате применения оператора к каждому слову из левой части таблицы. В общем виде такая таблица имеет вид:

Входные слова	Выходные слова

В случае бесконечной области определения алфавитного оператора задание его с помощью таблицы принципиально невозможно. В этом случае оператор задается системой правил, позволяющей за конечное число шагов найти выходное слово, соответствующее любому наперед заданному входному слову из области определения рассматриваемого алфавитного оператора. Алфавитные операторы, задаваемые с помощью конечной системы правил, принято называть алгоритмами.

Отсюда легко понять, что всякий алфавитный оператор, который можно фактически задать, является непременно алгоритмом. Однако следует иметь в виду одно различие, существующее между понятиями алфавитного оператора и алгоритма. В понятии алфавитного оператора существенно лишь само соответствие, устанавливаемое между входными и выходными словами, а не способ, которым это соответствие устанавливается. В понятии алгоритма, наоборот, основным является способ задания соответствия, устанавливаемого алгоритмом. Таким образом, алгоритм — алфавитный оператор вместе с правилами, определяющими его действие.

Два алфавитных оператора считаются равными, если они имеют одну и ту же область определения и сопоставляют любому наперед заданному входному слову из этой области одинаковые выходные слова.

Два алгоритма считаются равными, если равны соответствующие им алфавитные операторы и совпадает система правил, задающих действие этих алгоритмов на выходные слова.

Два алгоритма считаются эквивалентными, если у них совпадают алфавитные операторы, но не совпадают способы их задания (система правил). Обычно в теории алгоритмов рассматриваются лишь такие алгоритмы, которым соответствуют однозначные алфавитные операторы.

Всякий алгоритм такого рода любому входному слову относит только одно выходное слово. Такие алгоритмы и соответствующие им алфавитные операторы будем называть детерминированными.

К числу других свойств алгоритмов относятся массовость и результативность.

Массовость алгоритма — свойство алгоритма быть применимым для множества строк. Если алгоритм применим для множества строк, то он обладает свойством массовости.

Результативность алгоритма — свойство алгоритма, обеспечивающее получение результата через конечное число шагов. Если для любой из строк данного множества алгоритм через

конечное число шагов приводит к выходу с конечным результатом, то он обладает свойством результативности.

Из свойства результативности вытекает понятие области применимости алгоритма. Областью применимости алгоритма называется множество строк, для которых алгоритм результативен.

Теперь эквивалентность алгоритмов может быть определена следующим образом: два алгоритма эквивалентны, если совпадают их области применимости и результаты переработки любого слова из этой области.

В общем случае различают еще случайные и самоизменяющиеся алгоритмы. Алгоритм называется случайным, если в системе правил, описывающих алгоритм, предусматривается возможность случайного выбора тех или иных слов или тех или иных правил. Им соответствуют многозначные алфавитные операторы.

Алгоритм называется самоизменяющимся, если он не только перерабатывает входные слова, но и сам изменяется в процессе такой переработки. Результат действия самоизменяющегося алгоритма на то или иное входное слово зависит не только от этого слова, но и от истории предыдущей работы алгоритма.

В теории алгоритмов большое внимание уделяется общим способам задания алгоритмов, характеризующимся свойством универсальности, т.е. таким способам, которые позволяют задать алгоритм, эквивалентный любому наперед заданному алгоритму. Всякий общий способ задания алгоритмов назовем алгоритмической системой. При описании алгоритмических систем используются специальные формализованные средства.

Основные формализмы прикладной теории алгоритмов можно разделить на два направления, условно называемые «алгебраическим» и «геометрическим».

«Алгебраическая» теория строится в некоторой конкретной символике, при которой алгоритмы рассматриваются в виде линейных текстов. В «геометрической» теории алгоритмы строятся в виде множеств, между которыми вводятся связи, носящие характер отображений или бинарных отношений. При этом значительное место занимает геометрическая интерпретация объектов в виде графов, вершины которых задают элементы множества, а ребра — отношения между ними. При этой интерпретации отображения задаются в виде разметки вершин или ребер графа.

К первому направлению относятся рекурсивные функции, машины Тьюринга, операторные системы Ван-Хао, А. А. Ляпунова, логические схемы алгоритмов Ю. И. Янова и др.

Ко второму направлению относятся представления нормальных алгоритмов А. А. Маркова в виде граф-схем, предложенных Л. А. Калужниным, блок-схемный метод алгоритмизации и др.

Упражнения:

1. Составить словесные алгоритмы вычитания из некоторого числа А последовательности n чисел b_1, b_2,..., b_n.

а) алгоритм вычисления по формуле

$$C= (\dots ((A - b_1-b_2)- \dots - b_n);$$

б) алгоритм вычисления по формуле

$$C = A - \sum_{i=1}^{n} b_i.$$

2. Составить алгоритм поэлементного вычисления из последовательности n чисел a_1, a_2,..., a_n, последовательности n чисел b_1, b_2..., b_n, т.е. алгоритм вычисления по формуле

$$C_i = a_i - b_i (1 \le i \le n).$$

3. Составить алгоритм вычисления по формуле

$$C_k = \sum_{i=1}^{n} a_i - b_k \ (1 \leqslant i \leqslant n,\ 1 \leqslant k \leqslant m).$$

4. Составить алгоритм вычисления по формуле

$$C_i = a_i \times b_i \ (1 \leqslant i \leqslant m).$$

5. Составить алгоритм вычисления по формуле

$$C = \prod_{k=1}^{m} a_k \ (1 \leqslant k \leqslant m).$$

6. Составить алгоритм вычисления по формуле

$$C_i = \prod_{k=1}^{m} a_k \times b_i \ (1 \leqslant i \leqslant m,\ 1 \leqslant k \leqslant m).$$

7. Составить алгоритм вычисления по формуле

$$\prod_i c_i = a_i : b_i \ (1 \leqslant i \leqslant e).$$

8. Составить алгоритм поиска максимального (минимального) элемента из множества M, заданного в виде M = {a_1, a_2, ...,a_k}.

9. Составить алгоритм определения количества одинаковых чисел в последовательности, заданной в виде одномерного массива: a_1, a_2,..., a_n.

10. Составить алгоритм определения количества одинаковых элементов в матрице B размерностью n x m.

11. Составить алгоритм умножения матрицы на вектор.

12. Составить алгоритм умножения матрицы на матрицу.

13. Составить алгоритм транспортирования матрицы.

14. Составить алгоритм, реализующий операцию объединения

следующих двух множеств: M_1 = {a_1, a_2, ..., a_k} и M_2 = {b_1, b_2,..., b_n).

15. Составить алгоритм, реализующий операцию пересечения двух произвольных множеств.

16. Составить алгоритм, реализующий операцию разности двух произвольных множеств.

1.2. Рекурсивные функции

Исторически первой алгоритмической системой была система, основанная на использовании конструктивно определяемых арифметических (целочисленных) функций, получивших специальное название *рекурсивных* функций.

Рекурсией называется способ задания функции, при котором значение определяемой функции для произвольных значений аргументов выражается известным образом через значения определяемой функции для меньших значений аргументов.

Численные функции, значение которых можно установить посредством некоторого алгоритма, называются вычислимыми функциями. Функция называется рекурсивной, если существует эффективная процедура для ее вычисления. Понятие эффективной процедуры является интуитивным. Говорят, что имеется эффективная процедура для выполнения определенных вычислений, если эти вычисления выполняются по механическим правилам, т.е. по определенному алгоритму.

Поскольку понятие алгоритма в этом определении берется в интуитивном смысле, то и понятие вычислимой функции оказывается интуитивным. Тем не менее при переходе от алгоритмов к вычислимым функциям возникает одно очень существенное обстоятельство: совокупность процессов, подпадающих под интуитивное понятие алгоритма, очень обширна и мало обозрима. Совокупность вычислимых функций для самых разных пониманий процессов оказалась одной и той же и притом легко описываемой в обычных математических терминах.

Совокупность числовых функций, совпадающая с совокупностью всех вычислимых функций при самом широком до сих пор известном понимании алгоритма, носит название совокупности рекурсивных функций.

Гёдель впервые описал класс всех рекурсивных функций как класс всех числовых функций, определимых в некоторой формальной системе. Исходя из совершенно других предпосылок, Черч в 1936 г. вывел тот же класс числовых функций, что и Гёдель. Черчем была сформулирована гипотеза о том, что класс рекурсивных функций тождествен с классом всюду определенных вычислимых функций. Эта гипотеза известна под именем тезиса Черча. Понятие вычислимой функции точно не определяется, поэтому тезис Черча доказать нельзя.

Если некоторым элементам множества X поставлены в соответствие однозначно определенные элементы множества У, то говорят, что задана частичная функция из X в У.

Совокупность тех элементов множества X, у которых есть соответствующие в У, называется областью определенности функции, а совокупность этих элементов множества У — совокупностью значений функции. Если область определенности функции из X в У совпадает с множеством X, то функция называется всюду определенной.

Клини ввел понятие частично рекурсивной функции и высказал гипотезу, что все частичные функции, вычислимые посредством алгоритмов, являются частично рекурсивными. Эта гипотеза также недоказуема, как и гипотеза Черча. Однако математические исследования последних 30 лет выявили полную целесообразность считать понятие частично рекурсивной функции научным эквивалентом интуитивного понятия вычислимой частичной функции.

В дальнейшем под тезисом Черча будем понимать гипотезу Черча в том расширенном виде, который был придан ей Клини.

Тезис Черча оказался достаточным, чтобы придать необходимую точность формулировке алгоритмических проблем и в ряде случаев сделать возможным доказательство их неразрешимости. В силу тезиса Черча вопрос о вычислимости функции равносилен вопросу о ее рекурсивности. Понятие рекурсивной функции строгое. Поэтому обычная математическая техника позволяет иногда непосредственно доказать, что решающая задачу функция не может быть рекурсивной, т.е. задача неразрешима.

Применение рекурсивных функций в теории алгоритмов основано на идее нумерации слов в произвольном алфавите последовательными натуральными числами. Наиболее просто такую нумерацию можно осуществить, располагая слова в порядке возрастания их длин, а слова, имеющие одинаковую длину, — в произвольном (лексикографическом) порядке.

После нумерации входных и выходных слов в произвольном алфавитном операторе этот оператор превращается в функцию $y = f(x)$, в которой аргумент x и функция y принимают неотрицательные целочисленные значения. Функция f(x) может быть определена не для всех значений x, а лишь для тех, которые составляют область определения этой функции.

Подобные частично определенные целочисленные и целозначные функции для краткости называются арифметическими функциями, среди них выделим наиболее простые и будем их называть *элементарными арифметическими функциями*:

1) функции, тождественно равные нулю:

$$O^n(x_1, x_2, \ldots, x_n) = 0,$$

определенные для всех неотрицательных значений аргументов;

2) тождественные функции, повторяющие значения своих аргументов:

$$I_i^n (x_1, x_2, \ldots, x_n) = x_i \ (1 \leqslant i \leqslant n;\ n = 1, 2, \ldots),$$

частный случай $I_1^1 (x) = x$;

3) функции непосредственного следования:

$$S^1(x) = x + 1$$

также определенные для всех целых неотрицательных значений своего аргумента.

Используя в качестве исходных функций перечисленные элементарные арифметические функции, можно с помощью небольшого числа общих конструктивных приемов строить все более и более сложные арифметические функции. В теории рекурсивных функций особо важное значение имеют три операции: суперпозиции, примитивной рекурсии и наименьшего корня. Рассмотрим каждую из них.

Операция суперпозиции функций заключается в подстановке одних арифметических функций вместо аргументов других арифметических функций.

Пусть заданы n функций $f_1, \ldots, f_n$ от m переменных каждая и функция $f(x_1,\ldots,x_n)$ от n переменных. Операция суперпозиции функций

$$g(x_1, \ldots, x_m) = f(f_1(x_1, \ldots, x_m), \ldots, f_n(x_1, \ldots, x_m)).$$

Например, осуществляя операцию суперпозиции функций $f(x) = 0$ и $g(x) = x + 1$, получим функцию

$$h(x) = g(f(x)) = 0 + 1 = 1.$$

При суперпозиции функции $g(x)$ с самой собой получим функцию $h(x) = x + 2$ и т. п.

Операция суперпозиции обозначается символом S^{n+l}, где индекс сверху означает число функций.

Операция примитивной рекурсии позволяет строить $n+1$ - местную арифметическую функцию (функцию от $n + 1$ аргумента) по двум заданным функциям, одна из которых является n-местной, а другая $n+2$ - местной.

Пусть заданы какие-нибудь числовые частичные функции: n-местная g и $n+2$ - местная h. Говорят, что $n+1$ - местная частичная функция f возникает из функций g и h примитивной рекурсией, если для всех натуральных значений $x_i,\ldots, x_n, y$ имеем:

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n); \quad (3)$$

$$f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)). \quad (4)$$

Для правильного понимания операции примитивной рекурсии необходимо заметить, что всякую функцию от меньшего числа переменных можно рассматривать как функцию от любого большего числа переменных. В частности, функции-константы, которые естественно рассматривать как функции от нуля аргументов, можно рассматривать как функции от любого конечного числа аргументов.

В соответствии с этим операцию примитивной рекурсии будем применять и для $n = 0$, говоря, что одноместная частичная функция f возникает примитивной рекурсией из постоянной одноместной функции, равной числу a, и двуместной частичной функции h, если

$$f(0) = a;$$
$$f(x+1) = h(x, f(x)).$$

Встает вопрос, для каждых ли g и h существует функция f и будет ли она единственной. Область определения функции — множество всех натуральных чисел, поэтому ответ на оба вопроса положительный.

Если f существует, то из (3) и (4) находим:

$$\begin{gathered} f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n); \qquad (5) \\ f(x_1, \ldots, x_n, 1) = h(x_1, \ldots, x_n, 0, g(x_1, \ldots, x_n)); \\ f(x_1, \ldots, x_n, 2) = h(x_1, \ldots, x_n, 1, f(x_1, \ldots, x_n, 1)); \\ \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \\ f(x_1, \ldots, x_n, m+1) = h(x_1, \ldots, x_n, m, f(x_1, \ldots, x_n, m)) \end{gathered}$$

и поэтому f определена однозначно. Таким образом, для любых частичных n-местной функции g и $n+2$ - местной функции $h/n = 0, 1, 2, \ldots$ существует только одна частичная $n+1$ – местная функция f, возникающая из g и h примитивной рекурсией. Символически пишут $f = R(g, h)$.

Из (5) видно, что если мы каким-то образом умеем находить значение g и h, то значение f можно вычислить при помощи процедуры вполне механического характера. Для нахождения значения $f(a_1, \ldots, a_n, m+1)$ достаточно последовательно найти числа:

$$\begin{gathered} b_0 = g(a_1, \ldots, a_n); \\ b_1 = h(a_1, \ldots, a_n, 0, b_0); \\ b_2 = h(a_1, \ldots, a_n, 1, b_1); \\ \ldots \ldots \ldots \ldots \\ b_{m+1} = h(a_1, \ldots, a_n, m, b_m). \end{gathered}$$

Полученное на $m+1$ шаге число b_{m+1} и будет искомым значением f в точке $(a_1, \ldots, a_n, m+1)$.

В качестве примера применения операции примитивной рекурсии покажем, как при помощи этой операции из элементарных функций можно построить двуместную функцию суммирования $f(x, y) = x + y$.

Эта функция определяется с помощью тождественной функции $g(x) = x$ и функции непосредственного следования:

$$h(x, y, z) = z + 1.$$

В самом деле, используя правила (3) и (4), имеем:

$$\begin{gathered} f(x, 0) = g(x) = x; \\ f(x, 1) = h(x, 0, x) = x + 1; \\ f(x, 2) = h(x, 1, x+1) = x + 2; \\ \ldots \ldots \ldots \ldots \ldots \\ f(x, y-1) = h(x, y-2, x+y-2) = x + y - 1; \\ f(x, y) = h(x, y-1, x+y-1) = x + y. \end{gathered}$$

Аналогично можно построить показательную, степенную и другие известные арифметические функции.

Функции, которые могут быть построены из элементарных арифметических функций с помощью операций суперпозиции и примитивной рекурсии, примененных любое конечное число раз в произвольной последовательности, называются *примитивно рекурсивными функциями*.

Операции суперпозиции и примитивной рекурсии, будучи применены к всюду определенным функциям, дают в результате снова всюду определенные функции.

В качестве примера докажем примитивную рекурсивность некоторых простых арифметических функций.

Двуместная функция $f(x, y) = x + y$, удовлетворяет соотношениям:

$$x + 0 = x = I^1_1(x) \text{ — тождественная функция;}$$

$$x + (y + 1) = (x + y) + 1 = S(x + y) \text{ — функция непосредственного следования.}$$

Следовательно, функция $x + y$ возникает из примитивно рекурсивных функций I^1_1 и $h(x, y, z) = z + 1$ операцией примитивной рекурсии и потому функция $x + y$ — примитивно рекурсивная. Двуместная функция $f(x, y) = xy$ удовлетворяет схеме примитивной рекурсии:

$$x \cdot 0 = 0(x);$$
$$x(y + 1) = xy + x,$$

с начальными примитивно рекурсивными функциями

$$g(x) = 0(x),\ h(x, y, z) = z + x,$$

поэтому функция xy примитивно рекурсивная.

В области натуральных чисел разность $x - y$ естественно считать частичной двуместной функцией от x и y, определенной лишь для $x \geqslant y$, так как отрицательные числа не входят в рассматриваемую область. Но примитивно рекурсивные функции всюду определенные. Поэтому в теории примитивно рекурсивных функций вместо обычной разности вводят усеченную разность, обозначаемую символом $\dot{-}$ и определяемую следующим образом:

$$x \dot{-} y = \begin{cases} x - y, & \text{если } x \geqslant y; \\ 0, & \text{если } x < y. \end{cases} \quad (*)$$

В отличие от простой разности усеченная разность в области натуральных чисел всюду определенная, и в то же время она очень просто связана с обычной разностью. Так, например, согласно определению (*):

$$5 \dot{-} 3 = 2;\ 3 \dot{-} 5 = 0,\ (x \dot{-} y) \dot{-} z = x \dot{-} (y + z).$$

Функция $x \dot{-} 1$ удовлетворяет примитивно рекурсивной схеме

$$0 \dot{-} 1 = 0;$$
$$(x + 1) \dot{-} 1 = x,$$

с примитивно рекурсивными начальными функциями O^1 и I^2_1. Поэтому функция $x \dot{-} 1$ — примитивно рекурсивная.

С другой стороны, из (*) следует, что для любых x, y

$$x \dot{-} 0 = x;$$
$$x \dot{-} (y + 1) = (x \dot{-} y) \dot{-} 1.$$

Эти тождества показывают, что двуместная функция $x \dot{-} y$ возникает примитивной рекурсией из функций I^1_1 и $h(x, y, z) = z \dot{-} 1$.

Обе последние функции примитивно рекурсивны, поэтому и функция $x \dot{-} y$ — примитивно рекурсивная.

Большинство арифметических функций относятся к примитивно рекурсивным. Тем не менее примитивно рекурсивные функции не охватывают всех арифметических функций, которые

могут быть определены конструктивно. При построении всех этих функций используются другие операции, в частности операция наименьшего корня.

Операция наименьшего корня, или операция минимизации, позволяет определить новую арифметическую функцию $f(x_1,..., x_n)$ от n переменных с помощью ранее построенной арифметической функции $g(x_1,... , x_n, y)$ от $n + 1$ переменных.

Для любого заданного набора значений переменных $x_1 = a_1,..., x_n = a_n$ в качестве соответствующего значения $f(a_1, ..., a_n)$ определяемой функции $f(x_1, ..., x_n)$ принимается наименьший целый неотрицательный корень $y = a$ уравнения $g(a_1, ..., a_n, y) = 0$.

В случае несуществования целых неотрицательных корней у этого уравнения функция $f(x_1, ..., x_n)$ считается неопределенной при соответствующем наборе значений переменных.

Арифметические функции, которые могут быть построены из элементарных арифметических функций с помощью операций суперпозиции, примитивной рекурсии и наименьшего корня, называются *частично рекурсивными функциями*. Если такие функции оказываются к тому же всюду определенными, то они называются *общерекурсивными функциями*.

В этом определении, как и в определении примитивно рекурсивных функций, предусматривается возможность выполнения всех допустимых операций в любой последовательности и любое конечное число раз.

Частично рекурсивные функции представляют собой наиболее общий класс конструктивно определяемых арифметических функций.

Понятие частично рекурсивной функции — одно из главных понятий теории алгоритмов. Значение его состоит в следующем.

1. Каждая стандартно заданная частично рекурсивная функция вычислима путем определенной процедуры механического характера.

2. Какие бы классы точно очерченных алгоритмов до сих пор фактически ни строились, во всех случаях неизменно оказывалось, что числовые функции, вычислимые посредством алгоритмов этих классов, были частично рекурсивными. Поэтому общепринятой является следующая естественнонаучная гипотеза, известная под именем тезиса Черча.

Класс алгоритмически (или машинно) вычислимых частичных числовых функций совпадает с классом всех частично рекурсивных функций. Этот тезис дает алгоритмическое толкование понятию частично рекурсивных функций.

Практически понятием частично рекурсивных функций пользуются для доказательства алгоритмической разрешимости или неразрешимости проблем.

Использование же частично рекурсивных функций для представления того или иного конкретного алгоритма практически нецелесообразно ввиду сложности такого процесса алгоритмизации.

Упражнения:

1. Построить функции сложения и расписать алгоритм в виде последовательных шагов с использованием рекурсивных функций:

а) трехместную функцию сложения;

б) n-местную функцию сложения.

2. Построить функции умножения и расписать алгоритм в виде последовательных шагов с использованием рекурсивных функций:

а) двухместную функцию умножения;

б) трехместную функцию умножения;

в) n-местную функцию умножения.

3. Построить функции возведения в степень и расписать алгоритм в виде последовательности шагов для:

а) двухместной функции;

б) трехместной функции;

в) *n*-местной функции.

4. Построить частичную двухместную функцию деления и расписать алгоритм в виде последовательности шагов.

1.3. Машины Тьюринга

1.3.1. Основные понятия

Точное описание класса частично рекурсивных функций вместе с тезисом Черча дает одно из возможных решений задачи об уточнении понятия алгоритма. Однако это решение не вполне прямое, так как понятие вычислимой функции является вторичным по отношению к понятию алгоритма. Нельзя ли уточнить непосредственно само понятие алгоритма, а затем при его помощи определить и класс вычислимых функций?

Это было сделано в 1936—1937 гг. Постом и Тьюрингом независимо друг от друга и почти одновременно с работами Черча и Клини. Основная мысль Поста и Тьюринга заключалась в том, что алгоритмические процессы — это процессы, которые может совершать подходяще устроенная «машина». В соответствии с этим ими с помощью точных математических терминов были описаны довольно узкие классы машин. На этих машинах оказалось возможным осуществить или имитировать все алгоритмические процессы, которые фактически когда-либо описывались математиками.

Машины, введенные Постом и Тьюрингом, отличались не очень существенно и в дальнейшем стали называться машинами Тьюринга. Рассмотрим алгоритмические системы, представленные этими машинами.

Под машинами Поста и Тьюринга понимается некоторая гипотетическая (условная) машина, состоящая из следующих частей:

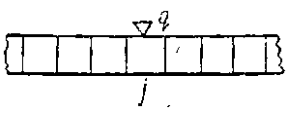
1) информационной ленты, представляющей собой бесконечную (неограниченную) память машины. В качестве информационной ленты может служить магнитная или бумажная бесконечная лента, разделенная на отдельные ячейки. В каждой ячейке можно поместить лишь один символ, в том числе и нуль;

2) «считывающей головки» — специального чувствительного элемента, способного обозревать содержимое ячеек. Вдоль головки информационная лента перемещается в обе стороны так, чтобы в каждый рассматриваемый момент времени головка находилась в одной определенной ячейке ленты;

3) управляющего устройства, которое в каждый рассматриваемый момент находится в некотором «состоянии». Предполагается, что устройство управления машины может находиться в некотором конечном числе состояний. Состояние устройства управления часто называют внутренним состоянием машины. Одно из этих состояний называется заключительным и в работе машины играет особую роль, так как оно управляет окончанием работы машины.

Схематически машина представлена на рис. 4.

Рассмотрим теперь алгоритмическую систему, предложенную Постом.



В алгоритмической системе Поста информация представляется в двоичном алфавите $A = \{1, 0\}$.

Рис. 4

Таким образом, в каждой ячейке информационной ленты можно поместить либо 0, либо 1. Алгоритм представляется в виде конечного упорядоченного набора правил, называемых приказами. Работа алгоритма начинается с некоторой начальной ячейки, соответствующей первому приказу алгоритма. Составляющие алгоритм приказы могут принадлежать к одному из 6 приказов, выполняемых устройством управления машины Поста:

1. Записать в рассматриваемую ячейку 1 и перейти к i-му приказу.

2. Записать в рассматриваемую ячейку 0 и перейти к *i*-му приказу.

3. Сдвинуть ленту вправо на одну ячейку и приступить к выполнению *i*-го приказа.

4. Сдвинуть ленту влево на одну ячейку и перейти к выполнению *i*-го приказа.

5. Если в рассматриваемой ячейке записана 1, то перейти к выполнению *j*-го приказа, а если записан 0, то перейти к выполнению
i-го приказа.

6. Окончание работы алгоритма, остановка.

Таким образом, алгоритмическая система Поста представляет собой машину с информационной лентой, содержащей в каждой ячейке либо 0, либо 1; с лентой, перемещающейся влево и вправо вдоль головки; с устройством управления, выполняющим 6 приказов (с внутренними состояниями).

Алгоритмы, составленные из любого конечного числа правил, представленных приказами машины Поста, называются *алгоритмами Поста.* Доказано, что алгоритмы Поста сводятся к алгоритмам, реализуемым с помощью частично рекурсивных функций, и наоборот, любая частично рекурсивная функция может быть представлена алгоритмом системы Поста.

В отличие от машины Поста в каждой ячейке ленты машины Тьюринга может находиться один из символов некоторого конечного алфавита, а устройство управления может быть в одном из конечных состояний. Другими словами, машина Тьюринга, работая в произвольном конечном алфавите, может выполнять некоторое конечное число приказов. При этом машины Тьюринга, как и машины Поста, могут сдвигать ленту на одну ячейку вправо или влево, оставляя содержимое ячеек неизменным, или могут изменять состояние воспринимаемой ячейки, оставляя ленту неподвижной.

Этот список операций можно расширить. Машина Тьюринга называется стандартной, если она при сдвиге ленты может предварительно изменять состояние воспринимаемой ячейки.

В дальнейшем будем рассматривать только стандартные машины Тьюринга.

Пусть алфавит машины Тьюринга задан в виде множества $A = \{s_0, s_1, \ldots, s_n\}$, где s_0 соответствует пустой ячейке, а число состояний устройства управления задано в виде множества $Q = \{q_0, q_1, \ldots, q_m\}$, где q_0 соответствует заключительному состоянию.

Конечная совокупность символов алфавита, с которой работает машина, называется внешним алфавитом, конечная совокупность состояний устройства управления — внутренним алфавитом.

Совокупность, образованная последовательностью состояний всех ячеек ленты и состоянием устройства управления, называется конфигурацией машины.

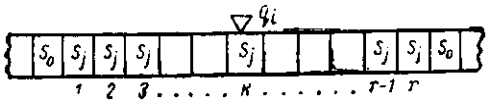


Рис. 5

Конфигурация задается в виде слова, описывающего конкретное состояние машины. Пусть в некоторый момент времени машина Тьюринга находится в состоянии, представленном на рис. 5. Конфигурация машины для данного случая будет представлена в виде слова

$$\ldots s_0 s_{j1} s_{j2} \ldots q_i s_{jk} \ldots s_{jr} s_0 \ldots , \qquad (6)$$

где s_0 — символ, обозначающий пустую ячейку; r — число заполненных ячеек на ленте; s_{j1} — состояние первой слева непустой ячейки; s_{jk} — состояние ячейки, просматриваемой в данный момент времени;

q_i — состояние устройства управления.

Каждая конфигурация содержит лишь одно вхождение символа q_i из внутреннего алфавита. Этот символ может быть в слове самым левым, но не может быть самым правым, так как справа от него должен помещаться символ состояния обозреваемой ячейки.

Если стандартная машина Тьюринга, находясь в состоянии q_i и воспринимая записанный на ленте символ s_k, переходит в новое состояние q_j, осуществляя при этом замену символа в рассматриваемой ячейке на символ s_m и сдвиг ленты влево на одну ячейку, то говорят, что машина выполняет команду $q_i s_k \rightarrow q_j s_m$Л. Если замены символа не происходит, s_m может в команде отсутствовать.

При манипуляциях с лентами примем следующие обозначения:

Л — движение ленты влево;

П — движение ленты вправо;

С — нет движения ленты.

Обычно команды стандартной машины Тьюринга задаются набором пятерок символов вида $q_i s_h q_i s_m$Л, т.е. стрелка опускается, или вида

$$q_i s_k \rightarrow q_j s_m \text{Л}. \qquad (7)$$

Рассмотрим пример машины Тьюринга с алфавитами $A = \{0, 1\}$, $Q = \{q_0, q_1\}$ и командами $q_i 1 q_i 1$Л, $q_1 0 q_0 1$С.

Пусть на ленте имеется слово 11100. Головка стоит над первой слева единицей. В результате работы машины Тьюринга это слово превращается в 11110. По окончании работы машины головка стоит над крайней правой единицей.

Стандартная машина Тьюринга с внешним алфавитом $A = \{0, 1\}$ называется нестирающей, если она способна выполнять лишь команды вида

$$\begin{aligned} q_\alpha 0 &\rightarrow q_\beta a T; \\ q_\alpha 1 &\rightarrow q_\beta 1 T, \end{aligned} \qquad (8)$$

где $a = \{0, 1\}$, $T = \{\text{С, Л, П}\}$, т.е. если она может вписать 1 в пустую ячейку, но не может удалить символ 1, если он уже вписан в ячейку.

Совокупность всех команд, которые может выполнять машина, называется ее программой.

Будем считать машину Тьюринга заданной, если заданы ее внешний и внутренний алфавиты, программа, начальная конфигурация и указано, какие из символов обозначают пустую ячейку и заключительное состояние.

Пусть машина Тьюринга задана внешним алфавитом $A = \{s^0, a, b, c, d\}$, внутренним алфавитом $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$ и совокупностью команд $q_0 a q_1 a$Л, $q_0 b q_0 b$Л, $q_2 a q_5 d$П, $q_0 c q_0 c$Л, $q_1 d q_2 c$П, $q_3 a q_4 d$П, $q_4 b q_2 c$П. Пустую ячейку обозначает символ s_0, а заключительное состояние — q_5.

Рассмотрим применение данной машины Тьюринга для переработки исходного слова *bcadc*, которое даст нам слово *bcdcc*. Начальная конфигурация имеет вид $q_0 bcadc$, при этом машиной будет порождена следующая последовательность конфигураций:

I шаг	$bq_0 cadc$	результат выполнения команды ($q_0 b \rightarrow q_0 b$Л)
II шаг	$bcq_0 adc$	то же ($q_0 c \rightarrow q_0 c$Л)
III шаг	$bcaq_1 dc$	-"- ($q_0 a \rightarrow q_1 a$Л)
IV шаг	$bcq \rightarrow cc$	результат выполнения команды ($q_1 d \rightarrow q_2 c$П)
V шаг	$bq_b cdcc$	-"- ($q_t a \rightarrow q_6 d$П)

Программа рассмотренной машины Тьюринга может быть представлена в виде таблицы соответствия:

A / Q	s^0	a	b	c	d
q_0	—	$q_1aЛ$	$q_0bЛ$	$q_0cЛ$	—
q_1	—	—	—	—	$q_2cЛ$
q_2	—	$q_3dП$	—	—	—
q_3	—	$q_4dП$	—	—	—
q_4	—	—	—	—	$q_2bП$
q_5		Останов			

Запись алгоритма, реализуемого машиной Тьюринга, производится с помощью программы работы этой машины, представляющей собой совокупность команд. В рассматриваемом примере алгоритм переработки входного слова *bcadc* в слово *bcdcc* представляется командами $q_0bq_0bЛ$, $q_0cq_0cЛ$, $q_0aq_1aЛ$, $q_1dq_2cП$, $q_2aq_5dП$.

Алгоритмы пишутся столбцами. Для контроля против некоторых команд указывается слово, в которое переходит первоначально заданное слово после выполнения всех предыдущих команд, включая и стоящую слева команду.

Рассмотрим в качестве примера алгоритм переноса нуля для машины Тьюринга, программа которой задана следующей таблицей соответствия:

A / Q	0	1
q_0	$q_00С$	$q_01С$
q_1	$q_2Л$	—
q_2	$q_31С$	—
q_3	$q_4П$	$q_3Л$

A / Q	0	1
q_4	—	$q_50С$
q_5	$q_6П$	—
q_6	$q_00С$	$q_6П$

Начальная конфигурация машины имеет вид q_10010, конечная конфигурация машины должна иметь вид q_00100. Таким образом, q_0 — заключительное состояние машины Тьюринга. Алгоритм будет иметь следующий вид:

Команды	Конфигурации
$q_10q_2Л$	$0q_2010$
$q_20q_31С$	$0q_3110$
$q_31q_3Л$	$01q_310 \rightarrow 011q_30$
$q_30q_4П$	$01q_410$
$q_41q_50С$	$01q_500$
$q_50q_6П$	$0q_6100$
$q_61q_6П$	q_60100
$q_60q_00С$	q_00100

Как было уже сказано, на машинах Тьюринга оказалось возможным осуществить или имитировать все алгоритмические процессы, которые когда-либо описывались математиками. Было доказано, что класс функций, вычислимых на этих машинах, точно совпадает с классом всех частично рекурсивных функций.

Вопрос о возможности или невозможности разрешающего алгоритма для задачи того или иного типа следует понимать как вопрос о существовании или не существовании машины Тьюринга, обладающей нужным свойством.

В зависимости от числа используемых лент, их назначения и числа состояний устройства управления определяются различные модификации машин Тьюринга. Рассмотрим основные из них.

1.3.2. Машины Тьюриига с двумя выходами

Предположим, мы расширили определение машины Тьюринга, добавив в устройство управления машины определенное состояние q^*. Будем говорить, что если устройство управления переходит в состояние q_0 для заданного входного слова x, то машина допускает x. Если устройство управления приходит в состояние, то машина запрещает x. Такое устройство будем называть машиной Тьюринга с двумя выходами.

Оказывается, что если заданы две машины Тьюринга T_1 и T_2, которые допускают непересекающиеся множества слов X_1 и X_2 соответственно, то всегда можно построить машину Тьюринга T_3 с двумя выходами, которая будет допускать X_1 и запрещать X_2. Эти машины Тьюринга будут нам полезны при рассмотрении, вопроса о разрешимости.

Множество разрешимо, если существует машина Тьюринга с двумя выходами, которая допускает все элементы этого множества и запрещает элементы, не принадлежащие ему.

1.3.3. Многоленточная машина Тьюринга

Схема многоленточной машины Тьюринга представлена на рис. 6. Она состоит из некоторого конечного числа лент, разбитых на ячейки. В каждой ячейке может находиться один из символов внешнего алфавита $A = \{s_0, s_1, \ldots, s_n\}$. Ячейки с символом s_0 будут считаться пустыми. Устройство управления машины в каждый момент времени находится в одном из конечного множества состояний $Q = \{q_0, q_1, \ldots, q_m\}$. Состояние q_0 считается заключительным состоянием машины.

Предположим, что в каждый момент времени машина «обозревает» одну ячейку каждой ленты.

Конфигурация машины считается заданной, если известносостояние устройства управления q, состояния всех ячеек всех лент и указаны ячейки,воспринимаемые машиной. Для k-ленточной машины конфигурация ее в i-й момент времени описывается системой *k-слов* вида:

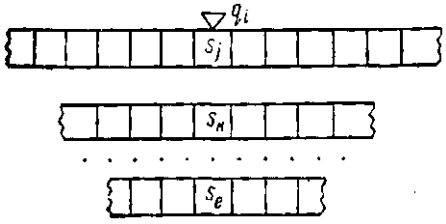


Рис. 6

$$\begin{aligned} &s_{i11} \ldots s_{i1e} q_i s_{i1e+1} \ldots s_{i1t}; \\ &\ldots\ldots\ldots\ldots \\ &s_{ik1} \ldots s_{ike} q_i s_{ike+1} \ldots s_{ikv}, \end{aligned} \qquad (9)$$

по одному слову для каждой ленты первый индекс соответствует моменту времени, второй — номеру ленты, третий — номеру ячейки на ленте, считая слева направо.

В зависимости от внутреннего состояния устройства управления q_i содержимого обозреваемых в данный момент ячеек лент $s_{i1e+1}\ldots s_{ike+1}$ машина переходит в следующее состояние посредством таких действий:

1) состояние каждой воспринимаемой ячейки заменяется новым состоянием (которое может совпадать со старым);

2) каждая лента передвигается на одну ячейку вправо, влево или остается неподвижной;

3) внутреннее состояние устройства управления q_i переводится в новое состояние q_j (которое может совпадать со старым).

Считается, что если устройство управления в какой-то момент приходит в заключительное состояние q_0, то машина останавливается и в следующие моменты времени конфигурация машины не меняется.

Говорят, что машина выполняет команду

$$q_i s_{\alpha 1} \ldots s_{\alpha k} \rightarrow q_j s_{\beta 1} T_1 \ldots s_{\beta k} T_k, \quad T = \{Л, С, П\}, \qquad (10)$$

если, находясь в состоянии q_i и воспринимая ячейки с символами $s_{\alpha 1}\ldots s_{\alpha k}$, машина переходит в состояние q_j заменяя содержимое ячеек соответственно символами $s_{\beta 1}\ldots s_{\beta k}$. После этого ленты соответственно сдвигаются в направлениях $T_1 \ldots T_k$.

Задать программу многоленточной машины Тьюринга — это значит для каждого слова

$$q_i s_{\alpha 1}\ldots s_{\alpha k} (i=1,\ldots,m;\ \alpha 1,\ldots,\ \alpha k=0,1,\ldots,\ n)$$

задать какую-нибудь команду вида (10). Таким образом, k-ленточная машина Тьюринга определяет алгоритм для переработки слов вида (9).

Заметим, что вычислимость на многоленточной машине Тьюринга частично рекурсивных функций есть прямое следствие вычислимости этих функций на одноленточной машине.

1.3.4. Универсальная машина Тьюринга

До сих пор мы придерживались той точки зрения, что различные алгоритмы осуществляются на различных машинах Тьюринга, отличающихся набором команд, внутренним и внешним алфавитами. Однако можно построить универсальную машину Тьюринга, способную в известном смысле выполнять любой алгоритм, а значит, способную выполнять работу любой машины Тьюринга.

В универсальной машине Тьюринга, как и во всякой тьюринговой машине, информация изображается символами, расположенными одновременно на магнитной ленте. При этом универсальная машина Тьюринга может располагать лишь фиксированным конечным внешним алфавитом. Между тем она должна быть приспособлена к приему в качестве исходной информации всевозможных состояний устройства управления и конфигураций, в которых могут встречаться буквы из разнообразных алфавитов со сколь угодно большим числом различных букв.

Это достигается путем кодирования конфигурации и программы любой данной машины Тьюринга в символах входного (внешнего) алфавита универсальной машины. Само кодирование должно выполняться следующим образом:

1) различные буквы должны заменяться различными кодовыми группами, но одна и та же буква должна заменяться всюду, где бы она ни встречалась, одной и той же кодовой группой;

2) строки кодовых записей должны однозначным образом разбиваться на отдельные кодовые группы;

3) должна иметь место возможность распознать, какие кодовые группы соответствуют различным сдвигам, т.е. каждой из букв Л, П, С в отдельности, и различать кодовые группы, соответствующие буквам внутреннего алфавита и буквам внешнего алфавита.

Рассмотрим пример такого кодирования для машины Тьюринга T, имеющей внешний алфавит $A = \{s_1, s_2, \ldots, s_k\}$ и внутренний алфавит $Q = \{q_1, q_2, \ldots, q_m\}$.

Если внешний алфавит универсальной машины Тьюринга состоит из символов $A = \{0, 1\}$, то эти условия будут наверняка соблюдены при следующем способе кодирования.

1. В качестве кодовых групп берутся $3 + k + m$ различных слов вида 100 ... 01 (между единицами — сплошь нули), где k — число символов внешнего алфавита;

m — число состояний устройства управления.

Тогда разбивка строк на кодовые группы производится однозначным образом путем выделения последовательностей нулей, заключенных между двумя единицами.

2. Сопоставление кодовых групп исходным символам внешнего и внутреннего алфавитов осуществляется согласно следующей таблице кодирования:

	Буква	Кодовая группа	
	Л	101	
	С	1001	
	П	10001	
Внешний алфавит	S_1	100001—4 нуля 10000001—6 нулей	Четное число нулей, большее чем 2
	S_2	. . . .	
	S_k	10 . . . 01 $2(k+1)$ нулей	
Внутренний алфавит (состояния)	q_1	1000001—5 нулей	Нечетное число нулей, большее чем 3
	q_2	100000001—7 нулей	
	q_m	10 . . . 01 $2(m+1)+1$ нулей	

Так, например, для машины Тьюринга, перерабатывающей слова bcadc в слово bcdcc, входное слово в универсальной машине Тьюринга с данным кодом будет представлено следующей строкой:

10000001 1000000001 100001 100000000001 1000000001,

где 100001 — a, 10000001 — b, 1000000001 — c, 100000000001 — d.

Программа же будет представлена следующими строками:

1000001 10000001 1000001 10000001 101 ($q_0bq_0bЛ$)
1000001 1000000001 1000001 1000000001 101 ($q_0cq_0cЛ$)
1000001 100001 100000001 100001 101 ($q_0aq_1aЛ$)
100000001 100000000001 10000000001 1000000001 10001 ($q_1d_2q_2cП$)
10000000001 100001 100000000000001 100000000001 10001 ($d_2aq_8dП$)

При кодировании программ, конфигураций и входных слов вместо нулей и единиц можно брать любые другие два символа, например a и b.

Таким образом, если какая-либо машина Тьюринга T решает некоторую задачу, то и универсальная машина Тьюринга способна решить эту задачу при условии, что кроме кодов исходных данных этой задачи на ее ленту будет подан код программы машины T. Задавая универсальной машине Тьюринга T_u изображение программы любой данной машины Тьюринга T_n и изображение любого ее входного слова x_n, получим изображение выходного слова y_n, в которое машина T_n переводит слово x_n.

Если же алгоритм, реализуемый машиной T_n, не применим к слову x_n, то алгоритм, реализуемый универсальной машиной T_u, также не применим к слову, образованному из изображения x_n и программы машины T_n.

Таким образом, машина Тьюринга T_n может рассматриваться как одна из программ для универсальной машины T_u.

В связи с существованием универсальной тьюринговой машины таблицы соответствия, описывающие различные состояния устройства управления машины, имеют двоякое назначение:

1) для описания состояний устройства управления специальной машины Тьюринга, реализующей соответствующий алгоритм;

2) для описания программы, подаваемой на ленту универсальной машины Тьюринга, при реализации соответствующего алгоритма.

Современные электронные вычислительные машины строятся как универсальные; в запоминающее устройство наряду с исходными данными поставленной задачи вводится также и программа ее решения.

Однако в отличие от машины Тьюринга, в которой внешняя память (лента) бесконечна, в любой реальной вычислительной машине она конечна.

Для сравнения структур различных машин и оценки их сложности необходимо иметь соответствующую меру величины или сложности машин. Шеннон предложил рассматривать в качестве такой меры произведение числа символов и числа состояний. Таким образом, «сложность» машины Тьюринга равна произведению числа символов внешнего алфавита на число внутренних состояний, отличных от заключительного. Значительный интерес вызвала задача построения универсальных машин Тьюринга минимальной сложности.

К началу 1963 г. последними результатами в этом направлении были теорема Ватанабе о существовании универсальной машины с 5-ю символами и 6-ю состояниями, теорема Минского о существовании машины с 4-мя символами и 7-ю состояниями и теорема Триттера о существовании универсальной машины с 4-мя символами и 6-ю состояниями.

1.3.5. Композиции машин Тьюринга

С математической точки зрения машина Тьюринга — просто определенный алгоритм для переработки слов.

Операции композиции, выполняемые над алгоритмами, позволяют образовывать новые, более сложные алгоритмы из ранее известных простых алгоритмов. Поскольку машина Тьюринга — алгоритм, то операции композиции применимы и к машинам Тьюринга. Рассмотрим основные из них, а именно: произведение, возведение, в степень, итерацию.

Пусть заданы машины Тьюринга T_1 и T_2, имеющие какой-то общий внешний алфавит $A = \{a_0, a_1,..., a_m\}$ и внутренние алфавиты $Q_1 = \{q_0, q_1,..., q_n\}$ и соответственно $Q_2 = \{q_0, q_1 ,..., q'_t\}$. *Композитом*, или *произведением*, машины T_1 на машину T_2 будем называть машину T с тем же внешним алфавитом $A = \{a_0, a_1,..., a_m\}$, внутренним алфавитом $Q = \{q_0, q_1,..., q_n, q_{n+1}, ..., q_n+t\}$ и программой, получающейся следующим образом. Во всех командах T_1, содержащих заключительный символ q_0, заменяем его на символ q_{n+1}. Все остальные символы в командах T_1 оставляем неизменными. В командах T_2, напротив, символ q_0 оставляем неизменным, но зато каждый из остальных символов $q_j^1 \{j = 1, 2, ...,t)$ заменяем символом q_{n+j}. Совокупность всех команд T_1 и T_2, измененных указанным способом, и будет программой композита или произведения машин T_1 и T_2.

Произведение машины T_1 на машину T_2 обозначается через $T = T_1 \cdot T_2$, или $T = T_1 * T_2$.

Таким образом, машина T есть произведение машин T_1 и T_2, если последовательная работа этих двух машин эквивалентна работе одной машины T.

Таблица соответствия такой машины строится по следующему правилу: если управляющее устройство машин T_1 и T_2 имеют k_1 и k_2 состояний соответственно (без учета заключительных состояний), то управляющее устройство машины T имеет $k_1 + k_2$ состояний, причем начальным состоянием машины T является q_1^1 (начальное состояние T_1), а заключительным — q_0^2 (заключительное состояние машины T_2). Таблица соответствия машины T состоит из двух частей: верхняя описывает машину T_1, нижняя — машину T_2. При этом заключительное состояние машины T_1 отождествляется с начальным состоянием машины T_2.

Пусть машина T_1 задана таблицей соответствия:

Q \ A	0	1
q_1	$q_2$0П	$q_1$1П
q_2	$q_3$0П	$q_0$1С

Эта машина отыскивает ближайшую слева группу единиц после группы нулей.

Машина T_2 задана таблицей соответствия:

Q \ A	0	1
q_1	$q_0$0С	$q_1$0П

Машина стирает все единицы слева, если они есть, до ближайшего слева нуля.

В исходном состоянии считывающие головки машин T_1 и T_2 находятся у крайней правой заполненной ячейки. Тогда машина T, являющаяся их произведением, будет иметь таблицу с 2+1=3 состояниями, $q_0 = q_0^2$ — заключительное состояние машины T_2:

Q \ A	0	1
q_1^1	$q_2^1 0П$	$q_1^1 1П$
q_2^1	$q_2^1 0П$	$q_1^2 1С$
q_1^2	$q_0^2 0С$	$q_1^2 0П$

Если для машины Т применить новую нумерацию состояний, то таблица соответствия примет вид:

Q \ A	0	1
q_1	$q_2 0П$	$q_1 1П$
q_2	$q_2 0П$	$q_3 1С$
q_3	$q_0 0С$	$q_3 0П$

Машина Т отыскивает ближайшую группу единиц и стирает их.

Таким же образом определяется *операция возведения в степень*: *n-й* степенью машины T называется произведение $T...T$ с n сомножителями.

В том случае, когда одна из перемножаемых машин имеет два или несколько заключительных состояний, умножение определяется совершенно аналогично, но только обязательно должно присутствовать указание, какое из заключительных состояний предыдущего сомножителя отождествляется с начальным состоянием последующего.

Так, например, если машина T_1 имеет два заключительных состояния, то произведение T_1 и T_2 будем обозначать, через

$$T = T_1 \begin{cases} (1) T_2 \\ (2) \end{cases} \text{ или } T = T_1 \begin{cases} (1) \\ (2)\, T_2, \end{cases}$$

в зависимости от того, отождествляется ли начальное состояние машины T_2,с первым или со вторым заключительным состоянием машины T_1

Машина T в этом случае также имеет два заключительных состояния: первое — одно из заключительных состояний машины T_1 второе — заключительное состояние T_2.

В примере

$$T = T_1 \begin{cases} (1)\, T_2 T_3 \\ (2)\, T_4 \end{cases}$$

умножение производится независимо по двум каналам, связанным с первым и вторым заключительными состояниями машины T_1.

Операция итерации применима к одной машине.

Суть этой операции состоит в следующем. Пусть машина T_1 имеет несколько заключительных состояний. Выберем какое-либо r-е заключительное состояние и отождествим его в основной таблице машины T_1 с ее начальным состоянием. Полученная машина обозначается через

$$T = \dot{T}_1 \begin{cases} (1) \\ \vdots \\ (r) \\ \vdots \\ (s) \end{cases}$$

и является результатом итерации машины T_1. Здесь точка указывает на отождествление заключительного состояния с начальным состоянием машины T_1.

Если T_1 имеет только одно заключительное состояние, то после выполнения итерации получается машина, не имеющая заключительного состояния вовсе.

Если итерация производится над машиной, которая сама является результатом умножения и итерации других машин, то соответствующее число точек ставится над теми машинами, чьи заключительные и начальные состояния отождествляются.

Например, запись

$$T = \dot{T}_1 \ddot{T}_2 \begin{cases} (1)\,\dot{T}_3 \\ (2)\,T_4 \ddot{T}_5 \\ (3)\,T_6 \end{cases}$$

означает, что заключительное состояние T_3 отождествляется с начальным состоянием T_1, а заключительное состояние T_5 отождествляется с начальным состоянием T_2.

Рассмотрим пример построения машины с помощью операций умножения и итерации.

Пусть исходными являются машина T_2, рассмотренная в композиции произведения машин, и машины T_3, T_4 и T_5, имеющие следующие таблицы соответствия.

Машина T_3:

Q \ A	0	1
q_1	$q_2$0Л	$q_1$1Л
q_2	$q_2$0Л	$q_3$1Л
q_3	$q_0$0Л	$q_3$1Л

Машина отыскивает первую после группы нулей группу единиц справа от начальной ячейки и останавливается около последней из этих единиц.

Машина T_4:

Q \ A	0	1
q_1	$q_0$0П	$q_1$1П

Начиная работу с заполненной ячейки, машина T_4 идет налево я останавливается левее группы единиц на две ячейки.

Машина T_5:

Q \ A	0	1
q_1	q_0'0С	q_0''1С

Имея два заключительных состояния — q_0' и q_0'', машина «распознает», напротив какой ячейки находится считывающая головка, и в зависимости от этого переходит в первое или второе заключительное состояние.

Тогда машина, определяемая формулой

$$N = \dot{T}_4 T_5 \begin{cases} (1)\,T_3 \\ (2)\,\dot{T}_2, \end{cases}$$

имеет следующую таблицу соответствия:

Q \ A	0	1
q_1	$q_2$0П	$q_1$1П
q_2	$q_3$0С	$q_4$1С
q_3	$q_5$0Л	$q_3$1Л

Q \ A	0	1
q_4	$q_1$0С	$q_4$0П
q_5	$q_5$0Л	$q_6$1Л
q_6	$q_0$0Л	$q_6$1Л

Машина N, находясь в начальный момент времени в состоянии q_1, напротив заполненной ячейки стирает все единицы слева от начального положения до того места, где впервые встречаются две пустые ячейки подряд. После этого лента машины возвращается назад и останавливается у крайней правой заполненной ячейки той группы расположенных подряд единиц, из которой машина начинала работать. Машина повторяет операцию «стирания» групп единиц до тех пор, пока левее некоторой группы единиц не окажутся две пустые ячейки. Таким образом, применение операции итерации позволяет составлять машины, повторяющие определенную последовательность операций на ленте.

Прежде чем рассмотреть другие примеры композиций машин Тьюринга, приведем ряд машин, из которых будут строиться эти композиции.

Машина T_6:

Q \ A	0	1
q_1	$q_0$1С	$q_1$1Л

Если в начальный момент машина находится в состоянии q_1 и считывающая головка находится над ячейкой, в которой записана «1», то она «отыскивает» на ленте первую пустую ячейку, т.е. ячейку с нулем справа от той, над которой находится головка, записывает в ней «1» и останавливается. Если же вначале считывающая головка находилась напротив пустой ячейки, то машина записывает в ней «1» останавливается, не передвигая ленты.

Машина T_7:

Q \ A	0	1
q_1	$q_1$0П	$q_0$0П

Эта машина «стирает» единицу в той ячейке, где находится головка, если ячейка не пуста, передвигает ленту вправо на одну ячейку и останавливается. Если головка в начальный момент времени находилась над пустой ячейкой, то лента передвигается вправо до тех пор, пока не окажется над ячейкой, содержащей «1», и работает так, как описано выше.

Машина T_8:

Q \ A	0	1	Q \ A	0	1
q_1 q_2	— $q_3$1Л	$q_2$1Л $q_2$1Л	q_3 q_4	$q_3$1Л —	$q_4$1Л $q_0$0Л

Машина заполняет первый промежуток справа между двумя группами единиц, оставляя всего одну незаполненную ячейку.

Машина T_9 имеет два заключительных состояния — q_0' и q_0'':

Q \ A	0	1
q_1 q_2	— q_0'0Л	$q_2$1П q_0''1Л

Находясь в начальном состоянии всегда против заполненной ячейки, машина обнаруживает, что записано в соседней ячейке слева: 0 или 1. В зависимости от этого она переходит в состояние q_0' или q_0'' и останавливается в исходной ячейке. Машина T_{10}:

Q \ A	0	1
q_1 q_2	$q_0$1С $q_1$0Л	$q_2$1Л $q_2$1Л

Она заносит единицу слева от группы единиц через одну пустую, ячейку.

Машина T_{11}:

Q \ A	0	1
q_1	—	$q_2 0П$
q_2	$q_0 1Л$	$q_2 1П$

Начиная работу с заполненной ячейки, машина стирает в ней единицу и «переносит» ее в ближайшую пустую ячейку слева.

Машина T_{12}:

Q \ A	0	1
q_1	$q_2 0Л$	$q_2 1Л$
q_2	$q_0 0С$	$q_0 0С$

Машина стирает одну единицу в ячейке, расположенной правее исходной (если там есть «1»).

Рассмотрим пример построения машины P из машин T_3, T_5, T_6, T_{11}, T_{12} с помощью композиций итерации и произведения по следующей формуле:

$$P = \dot{T}_{11} T_5 \begin{cases} (1)\ \dot{T}_3 \\ (2)\ T_{12} T_3 T_6. \end{cases}$$

Машина P начинает работу, воспринимая самую правую заполненную ячейку представления какого-либо числа. Результатом ее работы является ситуация, в которой это число «перегоняется» налево и ставится рядом с ближайшим слева числом.

Q \ A	0	1	Q \ A	0	1
q_1	—	$q_2 0Л$	q_7	$q_8 0Л$	$q_8 1Л$
q_2	$q_3 1Л$	$q_2 1П$	q_8	$q_9 0С$	$q_9 0С$
q_3	$q_4 0С$	$q_7 1С$	q_9	$q_{10} 0Л$	$q_9 1Л$
q_4	$q_5 0Л$	$q_4 1Л$	q_{10}	$q_{10} 0Л$	$q_{11} 1Л$
q_5	$q_5 0Л$	$q_6 1Л$	q_{11}	$q_{12} 0Л$	$q_{11} 1Л$
q_6	$q_1 0Л$	$q_6 1Л$	q_{12}	$q_0 1С$	$q_{12} 1Л$

К такому же результату может привести машина с более простой таблицей соответствия:

Q \ A	0	1	Q \ A	0	1
q_1	—	$q_2 0П$	q_4	$q_1 0П$	$q_4 1Л$
q_2	$q_3 1П$	$q_2 1П$	q_5	—	$q_6 0Л$
q_3	$q_4 0Л$	$q_5 1Л$	q_6	$q_0 1С$	$q_6 1Л$

Машина R_m строится из машин T_1, T_3, T_6, T_7, T_8, T_9, T_{10} по формуле

$$R_m = T_{10} \dot{T}_1^m T_9 \begin{cases} (1)\ T_8 T_3^m \\ (2)\ T_7 T_3^m \dot{T}_6, \end{cases}$$

где m — степень соответствующей машины.

В частности,

$$R_1 = T_{10} \dot{T}_1 T_9 \begin{cases} (1)\ T_8 T_3 \\ (2)\ T_7 T_3 \dot{T}_6. \end{cases}$$

Воспринимая в начальный момент времени систему чисел $x_1, \ldots, x_m$ в стандартном положении, машина R_m печатает справа от представления чисел $x_1, \ldots, x_m$ первое из этих чисел (т.е. x_1) и останавливается, воспринимая в стандартном положении систему $m + 1$ чисел ($x_1, \ldots,$

x_m, x_1). Если же машина R_m воспринимает в начальный момент времени в стандартном положении систему чисел $x_1,\ldots, x_n$, где $n > m$, то она отпечатает справа от этой системы чисел число $x_{n-m} + 1$ и остановится, воспринимая систему ($x_1, x_2,\ldots, x_{n-m} + 1$).

Легко можно убедиться, что m-я степень машины R_m — R_m^m копирует систему чисел ($x_1,\ldots, x_m$), воспринятую в стандартном положении справа так, что результатом работы будет система $2m$ чисел ($x_1,\ldots, x_m, x_1,\ldots, x_m$), разделенных одним промежутком.

Если начальная конфигурация 0011110000, то заключительная — 001111011110000.

Машина S_m строится по формуле:

$$S_m = T_6 R_m^m T_7 T_1^m T_7 T_3^m.$$

Машина S_m делает почти то же, что и машина R_m^m, т.е. копирует систему чисел ($x_1,\ldots, x_m$) справа, но не с одним промежутком, а с двумя (т.е. между числами x_m и x_1 расположены две пустые ячейки).

Пусть имеется система чисел (x_1, x_2), где $x_1 = 2$, $x_2 = 4$. Тогда если начальная конфигурация была 011011110000, то конечная конфигурация будет иметь вид: 011011110011011111000.

Условимся теперь, как будут записываться и считываться с ленты аргументы и значения вычисляемых функций. Будем говорить, что считывающая и записывающая головка воспринимает систему чисел ($x_1,\ldots, x_m$) в стандартном положении, если, во-первых, эти числа записаны подряд и, во-вторых, головка расположена напротив самой правой ячейки в представлении последнего из чисел (x_n).

Дадим определение понятию вычисления на машине Тьюринга. Пусть задана машина T и пусть в начальный момент времени выполнены следующие условия:

1) машина T находится в начальном состоянии q_1;

2) на ленте представлена система n чисел ($x_1,\ldots, x_n$), которую головка воспринимает в стандартном положении;

3) все ячейки, расположенные правее той, напротив которой находится головка, пусты.

Будем считать, что T вычисляет функцию $x = \varphi(x_1,\ldots, x_n)$ от n переменных, если каковы бы ни были числа $x_1,\ldots, x_n$ существует такой момент времени, в который:

1) машина T достигает заключительного состояния q_0;

2) на ленте будет представлена система $n + 1$ чисел $x_1,\ldots, x_n, x,$ где $x = \varphi(x_1,\ldots, x_n)$, которую головка машины воспринимает в стандартном положении;

3) все ячейки, расположенные правее той, напротив которой находится головка, пусты.

Если же для какой-либо системы чисел ($x_1,\ldots, x_n$) машина никогда не останавливается или останавливается, но при этом не выполнено 2 или 3 условие, то такая машина не вычисляет значения функции φ при этих значениях аргументов.

Рассмотрим пример. Пусть $n = 3$, $\varphi(x_1, x_2, x_3) = x_1 + x_2 + x_3$. Тогда при $x_1 = 3$, $x_2 = 3$, $x_3 = 4$ начальная конфигурация на ленте будет иметь вид 00 111 0 11 0 1111 0, а заключительная ситуация запишется так 00 111 0 11 0 11110 1111111110. Считывающая головка при этом расположена над крайней правой единицей.

Рассмотрим машины Тьюринга, вычисляющие элементарные арифметические функции, из которых строятся рекурсивные функции.

Функцию непосредственного следования $S(x) = x + 1$ вычисляет машина $T = R_1 T_6$. Действительно, воспринимая в начальный момент число x в стандартном положении, машина запишет $S(x) = x + 1$ правее числа x и остановится в стандартном положении $(x, x + 1)$, т.е. вычислит функцию непосредственного следования.

Константную функцию $O^n(x_1,\ldots, x_n) = q$ вычисляет машина $T = T_{10} T_6^q$, которая правее системы чисел $(x_1,\ldots, x_n)$ записывает q и останавливается.

Тождественную функцию $I_i^n(x_1,\ldots, x_n) = x_i$ вычисляет машина $T = R_{n-i+i}$. Действительно, эта машина правее системы чисел $(x_1,\ldots, x_n)$ записывает $(n — i + 1)$-е число, т.е. x_i.

Можно доказать, что любая рекурсивная функция может быть вычислена на машине Тьюринга и наоборот, что на машине Тьюринга вычисляются только рекурсивные функции.

Имеет место следующая *теорема Тьюринга* (1). Для каждой частично рекурсивной словарной функции $f(x_1,\ldots, x_g)$, определенной в некотором алфавите $a_1,\ldots, a_m$, существует машина Тьюринга с символами $a_0, a_1,\ldots, a_m$ и подходящими внутренними состояниями, которая вычисляет функцию $f(x_1,\ldots, x_g)$.

Каждая частично рекурсивная функция получается из простейших арифметических функций с помощью операций суперпозиции, примитивной рекурсии и минимизации. Поэтому теорема Тьюринга будет доказана, если нам удастся решить следующие частичные задачи:

1. Построить машины Тьюринга, вычисляющие простейшие арифметические функции.

2. Имея машины Тьюринга, вычисляющие функции $f, f_1, \ldots, f_s$, построить машину, вычисляющую суперпозицию этих функций.

3. Имея машины Тьюринга, вычисляющие какие-то функции g и h, построить машину, вычисляющую функцию, возникающую из g и h примитивной рекурсией.

4. Построить машину Тьюринга, осуществляющую обращение функции.

Первая из этих задач нами уже решена, остальные предлагается решить читателю.

Упражнения:

1. Машина Тьюринга задана внешним алфавитом $A = \{0, 1, *\}$, внутренним алфавитом $Q = \{q_0, q_1, q_2, q_3\}$, где 0 — пустой символ, а q_3 — заключительное состояние. Программа машины Тьюринга задана в виде следующей таблицы соответствия:

Q \ A	1	0	*
q_0	$q_2$0Л	q_0Л	$q_3$0
q_1	q_1П	q_0Л	q_1П
q_2	q_2Л	$q_1$1	q_2Л
q_3	Останов		

Записать алгоритм сложения двух чисел с указанием против каждой команды соответствующих конфигураций, заданных на ленте в виде:

а)

∇q_0						
1	1	1	*	1	1	

б)

∇q_0								
1	1	1	1	1	*	1	1	1

в)

∇q_0								
0	0	1	1	*	0	0	1	1

2. Задана машина Тьюринга:

$$A = \{0,1\},\ Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}\},$$

где 0 — пустой символ, а q_{12} — заключительное состояние.

Программа задана в виде:

Q \ A	0	1
q_0	$q_7 0Л$	$q_1 0Л$
q_1	$q_2 0Л$	$q_1 1Л$
q_2	$q_3 0Л$	$q_2 1Л$
q_3	$q_4 1П$	$q_3 1Л$
q_4	$q_5 0П$	$q_4 1П$
q_5	$q_6 0П$	$q_5 1П$
q_6	$q_0 1Л$	$q_6 1П$

Q \ A	0	1
q_7	$q_{12} 0Л$	$q_8 0Л$
q_8	$q_9 0Л$	$q_8 1Л$
q_9	$q_{10} 1П$	$q_9 1Л$
q_{10}	$q_{11} 0П$	$q_{10} 1П$
q_{11}	$q_7 Л$	$q_{11} 1П$
q_{12}	Останов	

Записать алгоритм сложения чисел, заданных на ленте в виде

1	1	1	1	1	0	1	1	1

Машина начинает просмотр с самой левой единицы, находясь в состоянии q_0.

Результат суммирования должен быть представлен в виде

1	1	1	1	1	0	1	1	1	0	1	1	1	1	1	1	1	1

3. Машина Тьюринга задана следующей таблицей:

Q \ A	1	0	*	α
q_0	$q_2 \alpha Л$	$q_0 Л$	q_3	$q_0 1П$
q_1	$q_1 П$	$q_0 Л$	$q_1 П$	$q_0 Л$
q_2	$q_2 Л$	$q_1 1$	$q_2 Л$	$q_0 1$
q_3	$q_3 П$	$q_3 \alpha$	$q_0 П$	$q_0 0Л$

Записать алгоритм повторного сложения чисел m и n, заданных в виде

1	1	1	1	*	1	1	1

Процесс сложения состоит в следующем: левое число m прибавляется к первому числу n, потом m прибавляется к полученной сумме $n + m$ и т. д.

Просмотр ленты начинается с самой левой единицы. Машина находится в состоянии q_0.

4. Составить таблицу соответствия для машины Тьюринга, выполняющей алгоритм умножения двух чисел, и записать алгоритм. За основу взять таблицу соответствия для повторного сложения (3 упражнение) и изменить ее так, чтобы процесс повторного суммирования выполнялся столько раз, сколько единиц во множителе, после чего следует останов машины.

5. Составить таблицу соответствия для машины Тьюринга, выполняющей алгоритм умножения двух чисел, записанных на ленте в виде

▽ q_3								
1	1	1	·	1	1	*	1	1

Результат умножения должен быть представлен в виде:

▽													
0	0	0	0	·	1	1	*	1	1	1	1	1	1

Записать алгоритм с указанием соответствующих конфигураций.

6. Составить таблицу соответствия для машины Тьюринга, выполняющей алгоритм умножения двух чисел.

Записать алгоритм с указанием для каждой команды соответствующей конфигурации.

7. Составить таблицу соответствия для машины Тьюринга, выполняющей алгоритм вычитания, записать алгоритм и соответствующие каждой команде конфигурации.

8. Составить таблицу соответствия для машины Тьюринга, выполняющей алгоритм деления, записать алгоритм с соответствующими конфигурациями.

9. Пользуясь понятием машины Поста, записать алгоритмы:

а) сложения двух чисел;

б) вычитания двух чисел;

в) умножения двух чисел;

г) деление одного числа на другое.

10. Машина Тьюринга задана следующей таблицей соответствия:

A / a	S	0	1	2	3	4	5	6	7	8	9	a	A
q_0	q_1SП	—	—	—	—	—	—	—	—	—	—	—	—
q_1	q_6SП	$q_1$0П	q_2aЛ	—	—	—	—	—	—	—	—	—	—
q_2	q_3SП	$q_2$0Л	$q_2$1Л	—	—	—	—	—	—	—	—	—	—
q_3	—	$q_4$1П	$q_4$2П	$q_4$3П	$q_4$4П	$q_4$5П	$q_4$6П	$q_4$7П	$q_4$8П	$q_4$9П	$q_4$0Л	—	$q_4$1П
q_4	q_5SП	$q_4$0П	—	—	—	—	—	—	—	—	—	—	—
q_5	—	$q_5$0П	$q_5$1П	—	—	—	—	—	—	—	—	$q_1$1П	—
q_6	Останов												

Записать алгоритм счетчика единиц с указанием соответствующих конфигураций.

11. Составить таблицу соответствия для машины Тьюринга, выполняющей алгоритмы сложения и вычитания. Записать алгоритм одного примера с указанием конфигураций, соответствующих каждой команде.

12. Составить таблицу соответствия для машины Тьюринга, выполняющей алгоритмы сложения, вычитания, умножения, деления. Записать алгоритм одного из примеров с указанием соответствующих конфигураций.

1.4. Нормальные алгоритмы А.А.Маркова

Как мы уже условились, всякий общий способ задания алгоритмов называется алгоритмической системой. Алгоритмическая система, основанная на соответствии между словами в абстрактном алфавите, включает в себя объекты двоякой природы: элементарные операторы и элементарные распознаватели.

Элементарные операторы — достаточно просто задаваемые алфавитные операторы, с помощью последовательного выполнения которых реализуются любые алгоритмы в рассматриваемой алгоритмической системе.

Элементарные распознаватели служат для распознавания наличия тех или иных свойств перерабатываемой алгоритмом информации и для изменения, в зависимости от результатов распознавания, последовательности, в которой следуют друг за другом элементарные операторы.

Для указания набора элементарных операторов и порядка их следования при задании конкретного алгоритма удобно пользоваться ориентированными графами особого ряда, называемыми граф-схемами соответствующих алгоритмов.

Граф-схема алгоритма представляет собой конечное множество соединенных между собой вершин (или геометрических фигур), называемых узлами граф-схемы. Каждому узлу, кроме особых узлов, называемых входом и выходом, сопоставляется какой-либо элементарный оператор или распознаватель. Из каждого узла, которому сопоставлен оператор, а также из входного узла исходит точно по одной дуге. Из каждого узла, которому сопоставлен распознаватель, исходит точно по две дуги. Из выходного узла не исходит ни одной дуги. Число дуг, входящих в узел граф-схемы, может быть любым.

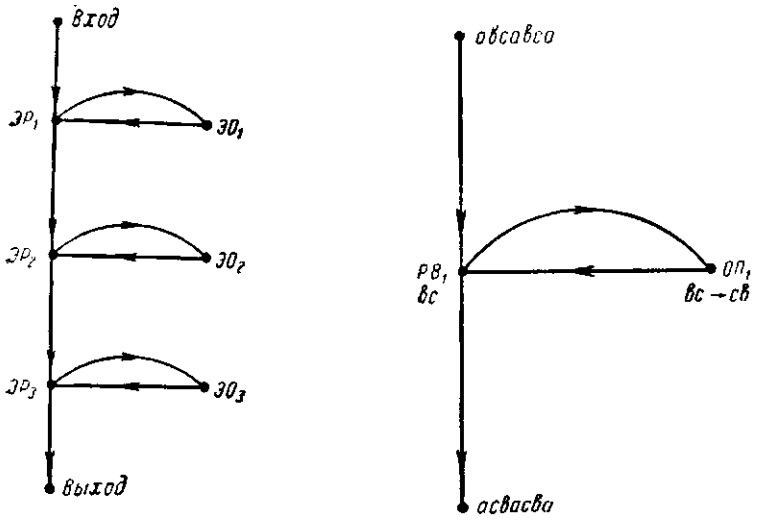
Рассмотрим на примере общий вид такой граф-схемы (стр. 46).

В рассматриваемом случае для вершины, соответствующей входу, имеем $\overset{+}{P}$ или $P^{+}(x) = 0$, $\overset{-}{P}$ или $P^{-}(x) = 1$; для вершин, соответствующих элементарным операторам, $-\overset{+}{P}$ или $P^{+}(x) = 1$, $\overset{-}{P}$ или $P^{-}(x) = 1$; для вершин, соответствующих элементарным распознавателям, $-\overset{+}{P}$ или $P^{+}(x) = 2$, $\overset{-}{P}$ или $P^{-}(x) = 2$; для вершины, соответствующей выходу, $-\overset{+}{P}$ или $\mathrm{P}^{+}(x) = 1$, $\overset{-}{P}$ или $P^{-}(x) = 0$.

В общем случае число дуг, входящих в вершины-распознаватели, может быть любым.

Алгоритм, определенный граф-схемой, работает следующим образом. Входное слово поступает на вход и двигается по направлениям, указанным стрелками. При попадании слова в распознавательный узел осуществляется проверка условия, сопоставленного этому узлу. При выполнении условия слово направляется в операторный узел, при невыполнении — к следующему распознавателю (иногда эти стрелки соответственно обозначают знаками «+» и «–»).

Если входное слово *p*, поданное на вход граф-схемы, проходя через узлы схемы и преобразуясь, попадает через конечное число шагов на выход, то считается, что алгоритм применим к слову *p* (слово *p* входит в область определения этого алгоритма). Результатом воздействия на слово *p* будет то слово, которое оказывается на выходе схемы. Если после подачи слова *p* на вход графа его преобразование и движение по граф-схеме продолжается бесконечно долго, не приводя на выход, считается, что алгоритм не применим к слову *p*, т.е. слово *p* не входит в область определения алгоритма. В нормальных алгоритмах в качестве элементарного оператора используется оператор подстановки, а в качестве элементарного распознавателя — распознаватель вхождения.



Распознаватель вхождения проверяет условие — имеет ли место вхождение рассматриваемого слова p_i в качестве полслова некоторого заданного слова q.

Оператор подстановки заменяет первое слева вхождение слова p_1 в слово q на некоторое заданное слово p_2. Оператор подстановки задается обычно в виде двух слов, соединенных стрелой $p_1 \rightarrow p_2$.

Например, для слова *abcabca* применение подстановки $bc \rightarrow cb$ через два шага приводит к слову *acbacba*.

$$abcabc \rightarrow acbabca \rightarrow acbacba.$$

Последовательность слов p_1, p_2, p_n, получаемых в процессе реализации алгоритма, называется дедуктивной цепочкой, ведущей от слова p_1 к слову p_n.

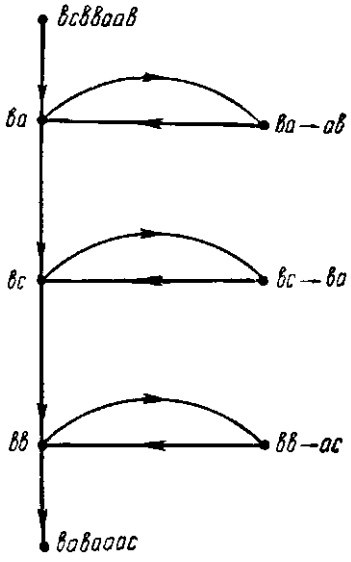
Алгоритмы, которые задаются графами, составленными исключительно из распознавателей вхождения слов и операторов подстановки, назовем *обобщенными нормальными алгоритмами*. При этом предполагается, что к каждому оператору подстановки ведет только одна дуга, исходящая из соответствующего распознавателя. Таким образом, для вершин, соответствующих операторам подстановки $\overset{+}{P}(x) = 1$.

Рассмотрим пример обобщенного нормального алгоритма, заданного подстановками

$$ba \rightarrow ab,\ bc \rightarrow ba,\ bb \rightarrow ac.$$

Работу алгоритма, заданного таким графом, рассмотрим для входного слова *bcbaab*.

$$bcbaab \rightarrow bcabab \rightarrow bcaabb \rightarrow baaabb — baaaac.$$



Рассмотрев обобщенные нормальные алгоритмы, перейдем к характеристике собственно нормальных алгоритмов.

Нормальными алгоритмами называются такие обобщенные нормальные алгоритмы, граф-схемы которых удовлетворяют следующим условиям:

1. Все узлы, соответствующие распознавателям, упорядочиваются с помощью их нумерации от 1до n.

2. Дуги, исходящие из узлов, соответствующих операторам подстановки, подсоединяются либо к узлу, соответствующему первому распознавателю, либо к выходному узлу. В первом случае подстановка называется обычной, во втором — заключительной.

3. Входной узел подсоединяется дугой к первому распознавателю.

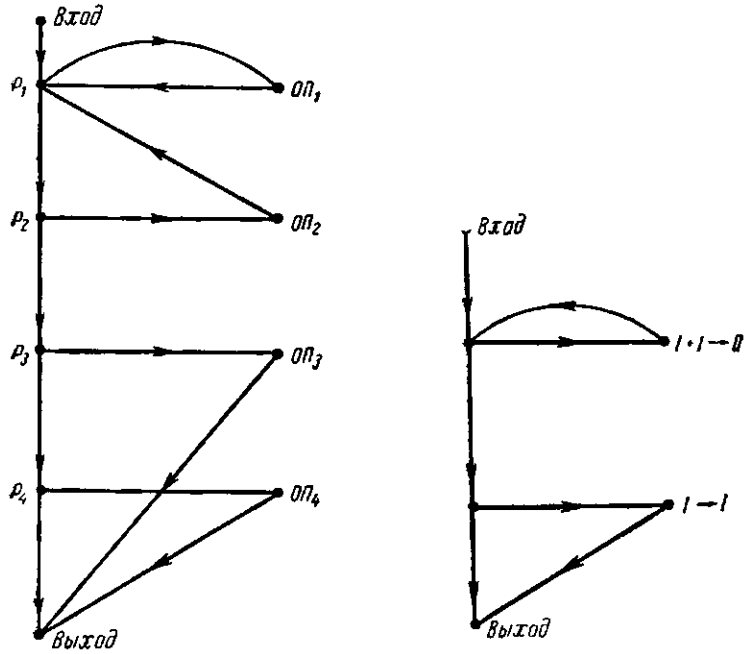
Нормальные алгоритмы принято задавать упорядоченным множеством подстановок всех

операторов данного алгоритма, называемым схемой алгоритма. При этом обычные подстановки записываются, как и в обобщенных алгоритмах, в виде двух слов, соединенных стрелкой ($p_1 \to p_2$), а заключительные подстановки обозначаются стрелкой с точкой ($p_1 \to\bullet\ p_2$). Процесс выполнения подстановок заканчивается лишь тогда, когда ни одна из подстановок схемы не применима к полученному слову или когда выполнена (первый раз) какая-либо заключительная подстановка.

Граф-схема нормального алгоритма в общем виде может быть представлена на стр. 43.

В качестве примера рассмотрим работу нормального алгоритма А.А.Маркова, заданного алфавитом А = {+, 1} и системой подстановок ‘1+1’→‘11’, ‘1’→•‘1’.

Пусть на входе задана строка ‘11+11+1’. Она перерабатывается алгоритмом в строку ‘11+11+1’→‘1111+1’→‘11111’→‘11111’. Алгоритм реализует сложение единиц.



Наличие в нормальных алгоритмах подстановок двух видов — заключительной и обычной — необходимое условие универсальности нормальных алгоритмов, т.е. возможности построения нормального алгоритма, эквивалентного любому наперед заданному алгоритму. Универсальность нормальных алгоритмов формулируется следующим *принципом нормализации*. Для любого алгоритма (конструктивно задаваемого алфавитного отображения) в произвольном конечном алфавите A можно построить эквивалентный ему нормальный алгоритм над алфавитом A.

Понятие нормального алгоритма над алфавитом означает следующее. В ряде случаев не удается построить нормальный алгоритм, эквивалентный данному алгоритму в алфавите A, если использовать в подстановках алгоритма только буквы этого алфавита. Однако можно построить требуемый нормальный алгоритм, производя расширение алфавита A, т.е. добавляя к A некоторое количество новых букв. В этом случае принято говорить, что построенный алгоритм является алгоритмом над алфавитом A, хотя алгоритм по-прежнему будет применяться лишь к словам в исходном алфавите A.

Если алгоритм N задан в некотором расширении алфавита A, то говорят, что N есть нормальный алгоритм над алфавитом A.

Одноместная частичная словарная функция $F(p)$, заданная в алфавите A, называется нормально вычислимой, если существует нормальный алгоритм N над алфавитом A такой, что для каждого слова p в алфавите A выполнено равенство $F(p) — N(p)$.

В качестве примера рассмотрим словарную функцию в алфавите {0, 1}, заданную формулой $F(p) = pa$. Пусть N — нормальный алгоритм в более широком алфавите {0, 1, 2}, имеющий схему:

$$20\to02,\ 21\to12,\ 2\to\bullet a,\ \Lambda\to2.$$

Возьмем какое-нибудь слово в алфавите {0, 1}, например слово 0110 = *p*. После первого шага мы получим слово 20110. Каждый следующий шаг переработки будет сдвигать символ 2 на одно место вправо, и после пятого шага мы будем иметь слово 01102, которое после использования заключительной формулы даст слово 0110*a* = *pa*. Таким образом, алгоритм над алфавитом {0,1} с указанной схемой вычисляет функцию *pa*, заданную в алфавите {0,1}.

В частности, алгоритм со схемой $\Lambda\to\bullet\Lambda$ вычисляет функцию $F(p)=p$, а алгоритм со схемой $\Lambda\to\Lambda$ вычисляет нигде не определенную функцию.

Математически доказать принцип нормализации невозможно, поскольку понятие произвольного алгоритма не является строго определенным математическим понятием.

Условимся называть тот или иной алгоритм нормализуемым, если можно построить эквивалентный ему нормальный алгоритм, и ненормализуемым — в противном случае. Принцип нормализации теперь может быть высказан в следующей видоизмененной форме: все алгоритмы нормализуемы.

Справедливость этого принципа основана на том, что все известные в настоящее время алгоритмы являются нормализуемыми, а способы композиции алгоритмов, позволяющие строить новые алгоритмы из уже известных, не выводят за пределы класса нормализуемых алгоритмов. Рассмотрим основные способы композиции нормальных алгоритмов.

Новые алгоритмы могут быть построены из уже известных алгоритмов путем применения различных способов композиции алгоритмов. Одной из наиболее распространенных видов композиции является суперпозиция алгоритмов.

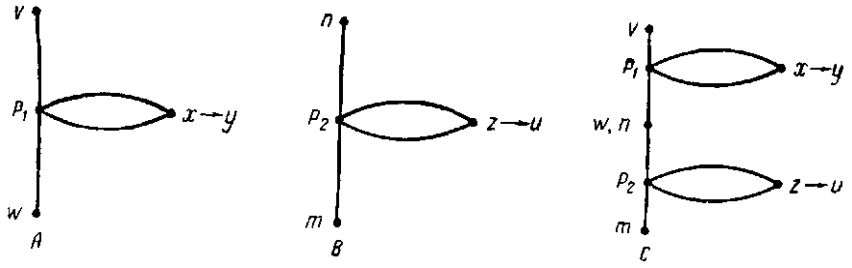
I. При суперпозиции двух алгоритмов *A* и *B* выходное слово первого алгоритма *A* рассматривается как входное слово второго алгоритма *B*. Результат суперпозиции алгоритмов *A* и *B* можно представить в виде $C(p)=B(A(p))$. Суперпозиция может выполняться для любого конечного числа алгоритмов.

Суперпозицию обобщенных нормальных алгоритмов можно рассматривать как обобщенный нормальный алгоритм.

II. Объединением алгоритмов *A* и *B* в одном и том же алфавите *X* называется алгоритм *C* в том же алфавите, преобразующий любое слово *p*, содержащееся в пересечении областей определения алгоритмов *A* и *B* в записанные рядом слова *A*(*p*) и *B*(*p*), на всех остальных входных словах этот алгоритм считается неопределенным.

Заданы *X* = {*a*, *b*), *A* = {*ab*→*ba*), *B* = {*ba*→*ab*} и слово из области их пересечения *aba*, тогда *A*(*aba*) = *baa*, *B*(*aba*) = *aab*, *C*(*aba*) = *baaaab*.

III. Разветвление алгоритмов представляет собой композицию трех алгоритмов *A*, *B* и *C*. Обозначая результат этой композиции буквой *D*, будем считать, что область определения алгоритма *D* совпадает с пересечением областей определения всех трех алгоритмов *A*, *B* и *C*, а для любого слова *p* из этого пересечения $D(p)=A(p)$, если $C(p)=e$, $D(p)=B(p)$ и $C(p)\neq e$, где *e* — пустая строка.



Заданы алгоритмы *A* = {*ab*→*ba*), *B* = {*ba*→*ab*), *C* = {*ab*→*a*, *ba*→*e*} в алфавите *X*= {*a*, *b*}. Рассмотрим действие алгоритма *D* на строки *aba*, *bab*.

$A\,(aba) = baa \qquad A\,(bab) = bba$
$B\,(aba) = aab \qquad B\,(bab) = abb$
$C\,(aba) = aa \qquad C\,(bab) = e$
$D\,(aba) = aab \qquad D\,(bab) = bba$

IV. Повторение (итерация) представляет собой композицию двух алгоритмов A и B. Обозначая результат этой композиции через C, определим, что для любого входного слова p соответствующее ему выходное слово $C(p)$ получается в результате последовательного многократного применения алгоритма A до тех пор, пока не получится слово, перерабатываемое алгоритмом B в некоторое фиксированное слово.

Например, заданы $X=\{a, b\}$, $A = \{ab \to ba\}$, $B = \{bbbaa \to ab\}$. Тогда $C(ababb) = ab$, так как $ababb \to baabb \to babab \to bbaab \to bbaba \to bbbaa \to ab$. Все рассмотренные композиции нормальных алгоритмов приводят к нормализуемым алгоритмам.

Очень важное значение для нормальных алгоритмов, как и для всякой универсальной алгоритмической системы, имеет задача построения *универсального алгоритма*, который должен выполнять работу любого нормального алгоритма, если задана его схема (система подстановок).

Для задания универсального нормального алгоритма U может быть применена, например, следующая схема. Фиксируется некоторый стандартный алфавит C (допустим двоичный). Для всех других возможных алфавитов задается некоторый способ их кодирования в алфавите C. Для символов, употребляемых в схемах нормальных алгоритмов (ввод, вывод, знак подстановки, разделитель операторов подстановки), отводятся отдельные коды.

Тогда, если задан некоторый алгоритм N, его кодируют одним словом N^U, называемым изображением алгоритма N, в стандартном алфавите. Входное слово p кодируют в слове p^U, называемое изображением слова.

Например, задан алгоритм $N\{xy \to x,\ y \to y\}$ и задано входное слово $p = xxyx$. Изображение этого алгоритма в стандартном алфавите, состоящем из десятичных цифр при кодировании: $\Gamma x = 010$, $\Gamma y = 020$, $\Gamma = 030$, Г<знак раздела> = 040. Г<ввод, вывод> = 50 будет следующим: $N^U = 050\ 010\ 020\ 030\ 010\ 040\ 020\ 030\ 020\ 050$; $p^U = 010\ 010\ 020\ 010$.

Справедлива следующая теорема Маркова об универсальном нормальном алгоритме. Существует такой универсальный нормальный алгоритм U, который для любого нормального алгоритма N и любого входного слова p из области определения алгоритма N переводит слово $N^U p^U$, полученное приписыванием изображения слова $p(p^U)$ к изображению алгоритма $N(N^U)$, в слово R^U, являющееся изображением соответствующего входного слова $R = N(p)$, в которое алгоритм N перерабатывает слово p. Если же слово p выбирается так, что алгоритм N к нему не применим, то и универсальный алгоритм U не применим к слову $N^U p^U$.

Эта теорема имеет большое значение, так как из нее вытекает возможность построения машины, которая может выполнять работу любого нормального алгоритма, а значит, в силу принципа нормализации — работу произвольного алгоритма. Программой машины в этом случае должно быть слово N^U (изображение данного алгоритма), а исходными данными — слово p^U.

Принцип нормализации хотя и доказан для всех известных в настоящее время алгоритмов, однако фактическая его реализация является очень нелегким делом. Поэтому на практике машины строятся на основании использования других алгоритмических систем.

После осуществления нумерации входных и выходных слов любой нормальный алгоритм может быть реализован в виде частично рекурсивной функции. Наоборот, любой алгоритм, реализуемый с помощью частично рекурсивной функции, оказывается эквивалентным некоторому нормальному алгоритму.

Имеет место следующая *теорема Детловса* (2). Класс нормально вычислимых частичных функций, заданных в произвольном алфавите A, совпадает с классом всех одноместных частично рекурсивных словарных функций в алфавите A.

Для доказательства достаточно сравнить определение нормального алгоритма с определением алгоритма Тьюринга при помощи программы подстановок. В самом деле, пусть *F(p)* — произвольная, частично рекурсивная словарная функция, заданная в алфавите *A*.

Согласно теореме Тьюринга, существует машина Тьюринга *T,* имеющая внешний алфавит a_0, A (a_0— новый, не входящий в *A* символ), и вычисляющая функция *F(p)*.

Пусть $q_0, q_1, ..., q_n$ — внутренние состояния Т и П — программа подстановок для машины *T*. Заменяя входящие в П формулы вида $q_i a_j \to q_0 a_k$ формулами $q_i a_j \to \bullet q_0 a_k$, получим схему некоторого нормального алгоритма *N* в алфавите $\{a_0, A, q_0, q_1, \ldots, q_n\}$. Сравнивая описания работы машины *T* и операций, предписываемых алгоритмом *N,* непосредственно видим, что для каждого слова *p* в алфавите *A* П*(p) =N(p)*.

Таким образом, каждая частично рекурсивная словесная функция нормально вычислима.

Упражнения:

1. Построить граф реализации алгоритма в алфавите $A = \{+, \backsim, ?, Q\}$, заданного подстановками ‘?$\backsim$’→‘$\backsim$’, ‘+$\backsim$’→‘?’, ‘?Q’→‘+’:

а) определить, к какому виду нормальных алгоритмов он относится;

б) рассмотреть на нем примеры дедуктивных цепочек, задавая исходное слово длиною не менее трех символов.

2. Задать нормальный алгоритм Маркова, реализующий вычитание *A–B*, где значениями *A* и *B* являются натуральные числа, представленные строками, состоящими из символов 1 (например, для *A* = 4, *B* = 3, *A–B* = 1 слово ‘1111–111’ должно быть переработано алгоритмом в слово ‘1’).

Проверить работу алгоритма для случаев:

а) *A* = 6, *B* = 2;

б) *A* = 3, *B* = 5.

3. Нормальный алгоритм Маркова, реализующий операцию умножения, задан алфавитом *A* = {1, ✳ , Т, Ф} и последовательностью подставок:

$$✳11 \to T✳1;\ ✳1 \to T;\ 1T \to T1Ф;\ ФT \to TФ;$$
$$Ф1 \to 1Ф;\ T1 \to T;\ TФ \to Ф;\ Ф \to 1;\ 1 \to \cdot 1.$$

Построить дедуктивную цепочку от слова ‘111✳1111’ к слову ‘111111111111’.

4. Задать нормальный алгоритм Маркова, реализующий операцию умножения чисел, представленных в виде последовательности единиц (алгоритм должен быть отличен от алгоритма п.3). Построить дедуктивную цепочку для одного из входных слов.

5. Задать нормальный алгоритм, реализующий операцию сравнения призначной части двух чисел, заданных в виде:

а) ∨∨∨111✳∨∨∨111 или ∨∨11✳∨∨∨∨1111
б) ∨∨∨✳111▭∨∨✳111 или ∨∨▭111✳∨∨∨▭11;
в) 111▭11✳11▭1111.

В пунктах *а* и *б* призначная часть числа представлена символами ∨, а в пункте *в* она представлена в тех же символах, что и само число, т.е. единицами.

Построить соответствующие дедуктивные цепочки по каждому алгоритму.

6. Выполнить суперпозицию нормальных алгоритмов, полученных в пп. 3 и 4.

Построить дедуктивную цепочку для одного произвольного примера.

1.5. Операторные алгоритмические системы

1.5.1. Общие замечания

Одной из особенностей абстрактной теории алгоритмов является неизменность набора допустимых средств, используемых для построения записи алгоритма или при его выполнении. Например, все частично рекурсивные функции получаются из некоторого фиксированного набора базисных функций с помощью трех операторов — оператора подстановки, оператора примитивной рекурсии, оператора наименьшего корня.

Элементарные акты при вычислениях на машине Тьюринга ограничиваются фиксированным набором операций, которые выполняет описываемый класс машин и т. д.

В то же время при изучении конкретных алгоритмов желательно, чтобы каждый алгоритм мог изучаться в терминах тех элементарных средств, которые наиболее удобны для его описания. Например, алгоритмы линейной алгебры удобнее всего описывать с помощью четырех арифметических действий, а алгоритмы вычисления функций алгебры логики — с помощью тех базисных логических операций, в терминах которых эти функции записаны.

Поэтому одним из требований, которое предъявляется к определению алгоритма при его практическом использовании, является то, чтобы как вид перерабатываемой информации, так и средства ее переработки выбирались в зависимости от класса рассматриваемых алгоритмов.

В каждой классической алгоритмической системе тем или иным способом вводится понятие выполнения алгоритма, т.е. осуществления последовательности актов, производящих постепенный переход от исходных данных к конечному результату. При этом характерно, что в каждом алгоритме список предписаний о выполнении элементарных актов — «команд» алгоритма фиксируется заранее и явно указывается в записи алгоритма. Например, в нормальных алгоритмах все применяемые формулы подстановки заранее указаны в записи алгоритма. Список допустимых действий машины Тьюринга при ее работе остается неизменным. Частично рекурсивная функция задается фиксированной последовательностью уравнений.

В то же время допущение формирования команд алгоритма в процессе ее реализации приводит к существенному сокращению и упрощению записи алгоритма. В связи с этим следующим требованием, предъявляемым к понятию алгоритма, является допущение формирования команд алгоритма в процессе его

Рассмотрение реальных алгоритмов показывает, что все элементарные операции, производимые в процессе выполнения алгоритмов, распадаются на две группы операций, которые обычно называют арифметическими и логическими. Арифметические операции осуществляют непосредственное преобразование информации. Логические операции определяют дальнейшее направление счета, т.е. последовательность выполнения арифметических операций. При этом во многих сложных алгоритмах преобладают логические операции, в то время как преобразование информации носит иногда очень простой характер. К числу таких задач относится и большинство экономических задач.

Элементарные акты, аналогичные логическим операциям, в явном или неявном виде вводятся и при определении понятия выполнения алгоритма в абстрактной теории алгоритмов.

Например, при выполнении нормального алгоритма после рассмотрения очередной формулы подстановки осуществляется либо переход к следующей формуле подстановки, либо остановка, либо переход к первой формуле подстановки схемы алгоритма.

В машине Тьюринга логической операцией можно считать движение ленты вправо или влево в зависимости от состояния машины и прочитанного символа. Однако, как правило, логические операции в классических алгоритмических системах носят тривиальный характер. Во многих случаях такая тривиальность, логических операций сильно усложняет построение конкретных алгоритмов.

В связи с этим следующим требованием, которое предъявляется к определению алгоритма, является допущение более универсальных логических элементарных операций, нежели те, которые имеются в абстрактной теории алгоритмов.

В той или иной мере этим требованиям отвечают так называемые операторные алгоритмические системы, к рассмотрению которых мы и переходим.

1.5.2. Операторные алгоритмы Ван Хао

Операторный алгоритм Ван Хао задается последовательностью приказов специального вида. Каждый приказ имеет определенный номер и содержит указания: какую операцию следует выполнить над заданным объектом и приказ с каким номером следует далее выполнить над результатом данной операции. Общий вид такого приказа:

$$i : \boxed{\omega \mid \alpha \mid \beta} \qquad (11)$$

где i — номер приказа,
ω — элементарная операция над объектом,
α, β — номера некоторых приказов.

В терминах машин Тьюринга номера приказов соответствуют номерам конфигураций машины, а элементарная операция над объектом — элементарному действию над конфигурацией при некотором состоянии.

Выполнить приказ (11) над числом x в операторном алгоритме — значит найти число $\omega(x)$ и далее перейти к выполнению над $\omega(x)$ приказа с номером α. Если же $\omega(x)$ не определено, то перейти к выполнению над числом x приказа с номером β.

Например, выполнив над числом 24 приказ

$$i : \boxed{6 \mid 7 \mid 13},$$

получим число 4 и указание выполнять далее над 4 приказ с номером 7. Выполнив этот приказ над числом 23, получим снова число 23 и указание выполнять над 23 приказ с номером 13.

Заключительному состоянию машины Тьюринга в операторных алгоритмах соответствует приказ вида

$$i : \boxed{\text{стоп}},$$

который означает, что вычисления следует остановить. Число, над которым следует выполнить этот приказ, есть результат вычислений.

В общем случае результат выполнения приказа (11) над числом x будем изображать парой (z, γ), где z — полученное число, γ — номер приказа, который должен выполняться далее над z.

Программой операторного алгоритма называется последовательность приказов вида:

$$\begin{aligned} i &: \boxed{\omega_i \mid \alpha_i \mid \beta_i} \\ &\cdots\cdots \\ i+s &: \boxed{\omega_{i+s} \mid \alpha_{i+s} \mid \beta_{i+s}} \\ &\cdots\cdots \\ i+n &: \boxed{\text{стоп}} \end{aligned} \qquad (12)$$

где $\omega(x), \ldots, \omega_{i+s}(x)$ — заданные одноместные частичные функции, определяющие элементарные операции над объектом x;
$\alpha_i, \beta_i, \ldots, \alpha_{i+s}, \beta_{i+s} \ldots$ — какие-то натуральные числа из последовательности $i, i+1, \ldots, i+n$.

Переработать x согласно данному операторному алгоритму — это значит выполнить над x последовательность следующих действий.

i — шаг. Находим $\omega_i(x)$. Если $\omega_i(x)$ определено, то результатом будет пара чисел $[\omega_i(x), \alpha_i]$. Если $\omega(x)$ не определено, то результатом i-го шага будет пара (x, β_i).

$i + 1$ — шаг, если результат предыдущего шага x, β_i, где $\beta_i = i + 1$. Находим $\omega_{i+1}(x)$. Если она определена, то результатом будет пара $[\omega_{i+1}(x), \alpha_{i+1}]$, если не определена, то результатом будет пара (x, β_{i+1}) и т. д.

$i + n - 1$ — шаг. Если в качестве результата на данном шаге получится пара, указывающая на заключительный оператор (z, $i + n$), то процесс обрывается и число z является результатом переработки числа x согласно заданному алгоритму ($z = \omega_{i+n-1}(x)$).

Если же в процессе выполнения алгоритма не возникает пары, указывающей на заключительный оператор, то результатом переработки x будет «неопределенное значение».

Если функция со всюду определенная, то символ β не оказывает влияние на процесс вычислений и поэтому обычно опускается. В этом случае приказ имеет вид i: | ω | α |.

Говорят, что операторный алгоритм A с программой (x) вычисляет частичную функцию $f(x)$, если алгоритм A перерабатывает каждое натуральное число x в $f(x)$. В частности, если $f(x)$ не определено, то процесс переработки x согласно программе (12) должен быть бесконечным.

Природа функций, вычислимых посредством операторных алгоритмов Ван-Хао, зависит от того, какие функции ω_1 входят в записи приказов. Имеет место следующая теорема, определяющая природу функций ω_i.

Теорема (3). Для того чтобы частичная функция $f(x)$ была вычислимой с помощью операторного алгоритма, программа которого содержит лишь частично рекурсивные функции $\omega_i(x)$ с рекурсивной областью определенности, необходимо и достаточно, чтобы $f(x)$ была частично рекурсивной.

Необходимость условий очевидна, и мы ограничимся лишь доказательством их достаточности.

Прежде всего, введем понятие одной важной для доказательства функции. Эту функцию будем обозначать через $ex(x, y)$ или сокращенно $ex_x y$ и называть экспонентой числа p_x в числе y. При $y \neq 0$ полагаем $ex_x y$ равным показателю наивысшей степени простого числа p_x, на которую делится y. Для y = 0 полагаем по определению $ex_x 0 = 0$ для всех значений x.

Например,

$$ex_3 8 = 3,\ ex_1 8 = 0,\ ex_0 0 = 0.$$

Ясно, что алгоритм с программой

0: | стоп |; 1: | + α | 1 |

вычисляет нигде не определенную функцию $f(x)$.

Пусть частично рекурсивная функция $f(x)$ имеет непустой график. Тогда этот график можно представить в виде совокупности пар $\langle\alpha(t), \beta(t)\rangle$, ($t$ = 0, 1, …), где α и β — подходящие примитивно рекурсивные функции. Обозначим через $\upsilon(x)$ выражение 2^x и введем частичную функцию $\omega(x)$, полагая

$$\omega(x) = \begin{cases} 3^x, \text{ если } ex_0 x \neq \alpha(ex_1 x), \\ \text{не определена, если } ex_0 x = \alpha(ex_1 x). \end{cases}$$

Функция $\omega(x)$ частично рекурсивна с рекурсивной областью определенности. Легко убедиться, что операторный алгоритм, имеющий программу

0: | стоп |; 1: | υ | 2 |; 2: | ω | 2 | 3 |;
3: | ex_1 | 4 |; 4: | β | 0 |

вычисляет как раз функцию $f(x)$. В самом деле, выполнив над произвольным числом x приказ 1, получим пару (2^x, 2). Если $x \neq \alpha(0)$, то, выполнив над 2^x приказ 2, получим пару ($2^x 3^1$, 2). Если $x \neq \alpha(1)$, то, выполнив над $2^x 3^1$ приказ 2, получим пару ($2^x 3^2$, 2) и т. д. Процесс продолжается до тех пор, пока, наконец, получится пара ($2^x 3^n$, 2), для которой $x = \alpha(n)$. Так как $\omega(2^x 3^n)$ не определено, то над числом $2^x 3^n$ теперь надо выполнить приказ 3, в результате которого получим пару (n, 4). Выполнив над n приказ 4, получим пару ($\beta(n)$, 0). Так как $x = \alpha(n)$, то $\beta(n) = f(x)$, и, следовательно, алгоритм перерабатывает x в $f(x)$, что и требовалось.

Построенная программа для вычисления функции $f(x)$ оказалась довольно тривиальной вследствие того, что запас функций, допустимых в приказах, был очень большой. Естественно, чем меньше запас, тем труднее строить нужные программы. Поэтому несомненный интерес представляет следующая теорема, которую мы приводим без доказательства.

Теорема Минского (4). Для каждой частично рекурсивной функций $f(x)$ существует операторный алгоритм, программа которого состоит из приказов вида

для любого x, перерабатывающий 2^x в $2^{f(x)}$.

Иными словами, любая частично рекурсивная функция $f(x)$ вычислима при помощи подходящего алгоритма, программа которого состоит из приказов приведенного вида, при условии, что значения аргумента и функции кодируются числами 2, $2^{f(x)}$.

1.5.3. Операторные алгоритмы А. А. Ляпунова

Алгоритмическая система советского ученого А.А.Ляпунова, предложенная им в 1953 г., является одной из первых, учитывающих все требования, предъявляемые к конкретным алгоритмам. Она возникла в связи с реализацией алгоритмов различных задач на ЭВМ.

Для описания строения алгоритмов Ляпунов использует специальный математический аппарат — так называемые «логические схемы алгоритмов», в которых заглавными логическими буквами обозначаются отдельные акты алгоритма, перерабатывающие информацию. Их называют операторами. Малыми буквами обозначаются проверяемые логические условия, при этом используется символика, принятая в математической логике. Например, символом $p(x<y)$ обозначают логическое условие, выполненное в том случае, когда неравенство, стоящее в скобках, истинно. В противном случае это условие ложно.

Последовательное выполнение нескольких операторов обозначается как произведение, причем сомножитель, стоящий справа, действует после сомножителя, стоящего слева.

Логическими схемами алгоритма называются выражения, составленные из операторов и логических условий, следующих один за другим. От каждого логического условия начинается стрелка, которая оканчивается у какого-либо из операторов. Например, из операторов A, B, C и логических условий p и q можно составить следующие логические схемы:

$$\downarrow^{2} A p^{1} B \downarrow q' \uparrow^{2} C$$

или

$$p^{\uparrow 1} A \downarrow^{2} B \downarrow^{1} C q \uparrow^{2}.$$

Знак $\uparrow_i$ обозначает начало стрелки, знак $\downarrow_i$ — ее конец. Одинаковыми номерами помечаются начало и конец одной и той же стрелки. Стрелки могут начинаться и у любого нелогического оператора.

Логическая схема алгоритма определяет порядок работы операторов в зависимости от значения входящих в нее логических условий. Работа алгоритма начинается с того, что выполняется самый левый оператор схемы. После того, как некоторый элемент схемы выполнен, определяется, какой оператор схемы должен выполняться следующим. Если это был оператор, то следующий за ним должен выполняться тот элемент, который стоит непосредственно справа, или тот, который указан соответствующей стрелкой. Если последним было логическое условие, то возможны два случая. Если проверяемое условие выполнено, то должен работать элемент, стоящий справа. Если оно нарушено, должен работать тот оператор, к которому ведет стрелка, начинающаяся после данного условия. Работа алгоритма оканчивается либо тогда, когда последний из работающих операторов содержит указание о ее прекращении, либо когда на некотором этапе не оказывается такого элемента схемы, какой должен был бы работать.

Для записи алгоритмов используются следующие основные типы операторов: 1) арифметические операторы; 2) операторы проверки логических условий; 3) операторы переадресации; 4) операторы переноса; 5) операторы формирования.

1. Арифметические операторы служат для записи различных арифметических действий и обозначаются начальными буквами латинского алфавита.

Например, оператор A вычисляет величину $a = a_{jk} - ca_{ik}$, оператор B — величину $c = a_{ij} : a_{ii}$. Выбор букв для обозначения операций произвольный.

2. Операторы проверки логических условий служат для определения порядка работы алгоритма и обозначаются малыми буквами латинского алфавита p, q и др.

3. Операторы переадресации служат для изменения адресов в приказах; для изменения различных параметров, от которых зависят операторы программы; для восстановления значений параметров и адресов, которые были изменены в процессе работы алгоритма (программы).

Операторы переадресации обозначаются буквой F с указанием в скобках изменяемого адреса или параметра. Так, оператор, изменяющий параметр i на единицу, будет обозначаться $F(i)$. Оператор, увеличивающий параметр i на n единиц, будет обозначаться $F^{n}(i)$. Оператор, уменьшающий параметр i на n единиц, будет обозначаться $F^{-n}(i)$.

4. Оператор переноса служит для «переноса» одного параметра на «место» другого, или, другими словами, для замены одного параметра другим. Операторы переноса обозначаются $[a \rightarrow b]$, где a означает, чем заменяется (что переносится), b — что заменяется (вместо чего переносится).

5. Операторы формирования служат для формирования начальных значений некоторых операторов алгоритма. Они переносят некоторые заранее запасенные приказы в определенные места алгоритма.

Операторы формирования можно использовать вместо операторов переадресации. Это особенно удобно, когда число переадресаций может быть различным, а начальное значение параметра — всегда одно и то же. Например, если начальное значение параметра i равно l, то оператор формирования может быть записан в виде $\{l \rightarrow i\}$.

Рассмотрим пример записи операторного алгоритма Ляпунова для суммирования пяти чисел: a_1, a_2, a_3, a_4, a_5.

Пусть c — параметр, представляющий собой $\sum_{i=1}^{5} a_i$, оператор A_i вычисляет величину $c_i = c_{i-1} + a_i$. Вычисление суммы начинаем с $i = 1$. Алгоритм имеет вид:

$$[1 \rightarrow i]\,\{0 \rightarrow c_{i-1}\}\downarrow\ A_i F(i)\, p\,(i = 6)\uparrow \text{ останов}$$

Операторные алгоритмы Ляпунова для решения определенной задачи допускают некоторые эквивалентные преобразования. Программа, построенная по любому из эквивалентных между собой алгоритмов, служит для решения одной и той же задачи.

Такие преобразования алгоритмов полезны в том отношении, что они могут быть использованы для поисков рациональной программы при реализации данного алгоритма на машине.

Преобразование схем можно рассматривать с двух точек зрения. Первый путь — исследовать формальные тождественные преобразования схем алгоритмов, предлагается советским математиком Ю.И.Яновым и будет рассмотрен далее.

Второй путь — исследовать содержательные преобразования схем алгоритмов, учитывающие индивидуальные особенности решаемой задачи.

1.5.4. Блок-схемный метод алгоритмизации

При блок-схемном методе алгоритмизации весь процесс решения задачи расчленяется на отдельные этапы-блоки. Каждый блок изображается на бумаге в виде простейших геометрических фигур (прямоугольника, ромба, круга и т. п.), блоку присваивается номер (метка), и он снабжается пояснительным текстом. Направление процесса обработки в блок-схеме указывается путем соединения отдельных элементов блок-схемы стрелками. Если один блок передает управление другим блокам в зависимости от выполнения определенных условий, то на стрелках связи указывается условие, при котором вычислительный процесс разветвляется.

При построении блок-схемы сложных вычислительных процессов не всегда целесообразно дробить весь процесс решения задачи на мелкие блоки. Иногда для большей обозримости возможно соединить в один блок целую группу этапов, т.е. в зависимости от сложности задачи составлять блок-схемы с различной степенью детализации.

Рассмотрим применение блок-схемного метода алгоритмизации при таксировке первичных документов. Эта типичная учетная задача, и ее используют для обработки таких первичных документов бухгалтерского учета, которые содержат в своем составе количественные и нормативно-расценочные показатели.

К таким задачам может быть отнесена и таксировка индивидуальных рабочих нарядов. Рассмотрим блок-схему обработки индивидуальных рабочих нарядов для единичных и мелкосерийных цехов:

Индивидуальный рабочий наряд		Цех	Участок	Заказ	Табельный номер	Вид оплаты
Деталь	**Операция**	**Количество**	**Норма времени**	**Расценка**	**Время нормированное**	**Заработная плата**
Подписи				**Время нормированное по документу**		**Заработная плата по документу**

Цель задачи сводится к определению времени нормированного (ВНД) и заработной платы (ЗПД) по каждому документу, участвующему в обработке.

Если обрабатывается п строк каждого наряда, то процесс получения ВНД и ЗПД можно представить как

$$\text{ВНД} = \sum_{i=1}^{n} k_i \times t_i;$$

$$\text{ЗПД} = \sum_{i=1}^{n} k_i \times p_i,$$

где k — количество деталеопераций;
t — норма времени;
p — расценка;
n — количество строк в одном документе;
i — номер строки в одном документе.

Логика решения задачи сводится к последовательному получению времени нормированного (ВН) и заработной платы (ЗП) по каждой строке. Для упрощения расчета принимаем, что количество заполненных строк в каждом документе (n) равно пяти. Затем ВН и

ЗП по пяти строкам каждого документа суммируются с получением соответственно ВНД и ЗПД.

Результатная информация выводится в виде следующих сообщений:

Номер документа	Цех	Участок	Заказ	Табельный номер	Вид оплаты	Время нормированное по документу	Заработная плата по документу

Блок-схема решения данной задачи представлена на рис. 7, где обозначения имеют следующие значения:

j — порядковый номер наряда (всего их 100);
k — номер строки в наряде;
n — количество строк в наряде.

Приведенный пример позволяет сформулировать выводы о некоторых основных достоинствах и недостатках блок-схемного метода алгоритмизации.

Блок-схема является формой представления алгоритма решения задачи в общем виде. Преимущества его сводятся к следующему:

1. Обеспечивается обмен методами решения между специалистами, использующими различные машины.

2. Облегчается работа по составлению машинной программы, так как заранее составлено укрупненное описание всего процесса решения задачи.

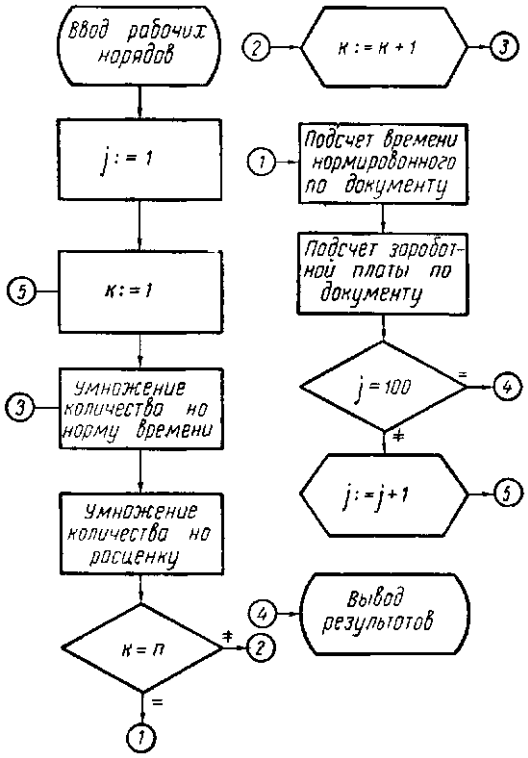
3. Создается возможность отдельно программировать каждый блок, а программу решения всей задачи получать путем объединения таких программ.

4. Облегчается чтение и понимание программ.

5. Уменьшается количество ошибок при программировании.

6. Упрощается проверка и отладка готовых программ.

Недостатки этого метода связаны, прежде всего, с невозможностью использовать его при автоматизации программирования, так как для более сложных задач словесное описание блок-схем оказывается громоздким для ввода в машину.



Другой серьезный недостаток состоит в том, что при составлении блок-схемы не устанавливается определенная степень детализации, которая впоследствии необходима при разработке программы. Причем четкость и полнота описания отдельных блоков зачастую зависят от квалификации составителей. И получается, что некоторые блоки описаны с излишней подробностью, а другие — в общих чертах.

Эти недостатки приводят к большим трудностям при переводе блок-схемы в рабочую программу машины и не позволяют сформулировать четкие правила такого перевода, которые необходимы при автоматизации программирования.

Таким образом, из изложенного выше следует общий вывод о том, что блок-схемный метод алгоритмизации может быть рекомендован при ручном методе программирования как этап, облегчающий составление рабочей программы.

Кроме того, блок-схемным методом удобно пользоваться при составлении сложных алгоритмов, в качестве предварительной проверки и наглядного представления логики решения задачи.

Если абстрактные алгоритмические системы служат для решения таких проблем, как алгоритмическая разрешимость или неразрешимость задач, то остальные системы используются практически для записи реальных алгоритмов. Они служат тем аппаратом, который используется для формализации алгоритмов (записи в формулах и терминах той или иной алгоритмической системы).

Однако в связи с возникновением ЭВМ эти алгоритмические системы оказались не совсем удобными, так как для решения задач, представленных аппаратом той или иной алгоритмической системы на ЭВМ, необходимо алгоритмы перевести в команды конкретной машины. Этот перевод выполнялся вручную и требовал много труда и времени.

Для ускорения процесса перевода алгоритмов задач с языка алгоритмической системы в язык машины были разработаны специальные алгоритмические системы, получившие название *алгоритмические языки*.

Строгая формализация и однозначность описания алгоритмов в алгоритмическом языке позволила переложить на машину и сам перевод с алгоязыка на язык машины, т.е. автоматизировать процесс подготовки задач для выполнения их на ЭВМ, автоматизировать программирование. Для этого необходимо снабдить машину так называемой «программирующей программой» — транслятором, которая, анализируя описание, сделанное

на алгоритмическом языке, перерабатывает его в программу, состоящую только из машинных команд.

В настоящее время разработано и используется более 700 раз личных алгоритмических языков. Наиболее распространенные из них АЛГОЛ, КОБОЛ, ФОРТРАН. У нас в СССР разработаны АЛГЭК, АЛГЭМ, АЛМО и др.

Детальным изучением алгоязыков занимается курс «Алгоритмические языки». В данном курсе изучаются общие принципы построения таких языков — формальная грамматика.

Упражнения:

В операторной системе Ляпунова

1. Составить алгоритм вычисления по формуле

$$c_i = a_i - b_i \, (1 \leqslant i \leqslant n).$$

2. Составить алгоритм вычисления по формуле

$$c_k = \sum_{i=1}^{n} A_i - b_k \, (1 \leqslant i \leqslant n, \; 1 \leqslant k \leqslant m).$$

3. Составить алгоритм вычисления по формуле

$$c_i = a_i \times b_i \, (1 \leqslant i \leqslant n).$$

4. Составить алгоритм вычисления по формуле

$$c_i = \prod_{k=1}^{m} A_k \times b_i \, (1 \leqslant i \leqslant n, \; 1 \leqslant k \leqslant m).$$

5. Составить алгоритм вычисления по формуле

$$\prod_{i=1}^{n} c_i = a_i : b_i.$$

6. Составить алгоритм определения количества одинаковых чисел в последовательности a_1, a_2, ..., a_n.

7. Составить алгоритм умножения матрицы на матрицу.

8. Составить блок-схему алгоритма поразрядного сложения двух чисел произвольной разрядности.

9. Составить блок-схему алгоритма поразрядного умножения двух чисел произвольной разрядности.

10. Составить блок-схему алгоритма упорядочения массива из n чисел в порядке возрастания элементов.

11. Составить блок-схему алгоритма поиска минимального элемента из массива n чисел.

12. Составить блок-схему алгоритма умножения матрицы на вектор,

13. Составить блок-схему алгоритма умножения матрицы на матрицу.

14. Составить блок-схему алгоритма транспонирования матрицы.

15. Составить блок-схему алгоритма поиска минимального (максимального) элемента матрицы.

16. Составить блок-схему алгоритма определения количества положительных (отрицательных) элементов матрицы.

17. Составить блок-схему алгоритма вычислений по формуле

$$c_k = \prod_{k=1}^{m} a_k \times b_i \, (1 \leqslant i \leqslant n, \; 1 \leqslant k \leqslant m).$$

18. Составить блок-схему алгоритма вычислений по формуле

$$c_k \doteq \sum_{i=1}^{n} a_i - \boldsymbol{b}_k \, (1 \leqslant k \leqslant e).$$

1.6. Методы оценки алгоритмов

В теории алгоритмов понятие «алгоритм» обычно уточняется посредством описания «математической модели» вычислительной машины. Здесь возможны два подхода в зависимости от того, оценивается ли сложность алгоритма (машины, программы) или способность вычислительного процесса, протекающего в соответствии с алгоритмом.

В качестве меры сложности алгоритмов рассматривается функционал, соотносящий каждому алгоритму A некоторое число $\mu(A)$, характеризующее (в подходящем смысле) его громоздкость, например: число команд в A, длина записи A, или какой-нибудь другой числовой параметр, характеризующий объем информации, содержащийся в A. Подобные функционалы уже давно применяются в теоретико-кибернетических исследованиях схем, реализующих функции алгебры логики (Шеннон, Яблонский, Ляпунов и др.), а также в вычислительной математике, где мощность схемы, по которой вычисляется многочлен, измеряется числом арифметических операций, фигурирующих в схеме. Различие заключается лишь в том, что в указанных работах рассматриваются специальные узкие классы функций и специальные способы описания алгоритмов. Мы же заинтересованы в разработке и применении аналогичных понятий в более общей ситуации (любые вычислимые функции, общие концепции алгоритма). Первые публикации, в которых такие меры сложности нашли применения, принадлежат А.А.Маркову и А.Н.Колмогорову.

В качестве меры сложности вычислений рассматривается функционал, соотносящий каждой паре (A, α), где A — алгоритм, α — индивидуальная задача из того класса задач, которые алгоритм A решает, некоторое число $\sigma(A, \alpha)$. Это число характеризует сложность работы алгоритма A применительно к исходным данным α до выдачи соответствующего результата. Например, в качестве $\sigma(A, \alpha)$ можно брать число элементарных шагов, из которых складывается эта работа (иначе говоря, длительность процесса вычисления) или объем памяти, который может понадобиться для реализации всех выкладок по ходу данного процесса, и т. д. Поскольку для каждого алгоритма A однозначно определен тот класс задач Ω, который он решает (например, та функция f, значение которой он вычисляет), то можно считать, что в рассматриваемой ситуации каждый алгоритм A характеризуется функцией переменного α $\sigma_A(\alpha) \underset{df}{=} \sigma(A, \alpha)$. Иными словами, мерой сложности работы алгоритма (вычисления) является оператор, сопоставляющий с каждым алгоритмом A соответствующую функцию $\sigma_A(\alpha)$.

Такой подход к оценке сложности вычислений содержался в работе советского математика Г.С.Цейтина, сделанной им в 1959 г., в которой сложность работы нормального алгоритма измерялась функцией, указывающей зависимость числа шагов алгоритма от слова, к которому он применяется. Примерно в то же время и независимо от Г.С.Цейтина Б.А.Трахтенброт ввел аналогичные функции, измеряющие объем памяти, необходимой для рекурсивных вычислений, и назвал их сигнализирующими функциями.

Определяя некоторую меру сложности для алгоритмов (или вычислений), мы тем самым надеемся получить удобный инструмент для сравнения алгоритмов, а также для оценки объективной трудности, присущей различным вычислимым функциям. В связи с этим можно отметить различие в осуществлении такого замысла в зависимости от того, рассматривается ли мера сложности алгоритма или мера сложности его работы. Поскольку сложность алгоритма измеряется действительным числом, то любые два алгоритма сравнимы по сложности.

Обычно считают, что мера сложности алгоритма принимает лишь натуральное значение, поэтому для каждой вычислимой функции существует вычисляющий ее алгоритм с минимальной сложностью. Эту минимальную сложность естественно рассматривать как сложность самой функции, тем самым и множество всех зависимых функций упорядочено по степени сложности этих функций.

Если же исходной является мера сложности вычислений, то получается иная картина. Две сигнализирующие функции могут оказаться несравнимыми, даже если считать, как обычно это принято, что сигнализирующая σ_A меньше сигнализирующей σ_B, если для всех α, за исключением, быть может, конечного их числа $\sigma_A(\alpha) < \sigma_B(\alpha)$. Поэтому коль скоро

зафиксирована некоторая функция f, то априори не ясно, существует ли для нее наилучшее вычисление. Здесь приходится по существу ограничиваться более слабой характеристикой сложности самой функции f, а именно отыскиваются две функции φ_1 (нижняя оценка) и φ_2 (верхняя оценка), такие, что:

1) существует алгоритм A_1 вычисляющий f с сигнализирующей, не превосходящей φ_2;

2) каков бы ни был алгоритм A, вычисляющий f, его сигнализирующая не меньше φ_1.

Разумеется, чем ближе друг к другу верхняя и нижняя оценки φ_1 и φ_2, тем точнее охарактеризована сложность самой функции f.

Итак, в отличие от иерархий, основанных на сложности алгоритмов, иерархии, основанные на мере сложности вычислений, являются частично упорядоченными. Это усложняет их изучение, но вместе с тем позволяет, по-видимому, более тонко улавливать сущность того, что мы интуитивно понимаем под сложностью вычислений функции.

Теория сложности зависит от положенных в ее основу концепций алгоритма (рекурсивные функции, машины Тьюринга и т. п.), а также от выбранной меры сложности.

Поэтому априори можно предположить, что при переходе от одних концепций алгоритма к другим необходимо заново строить теорию сложности. Однако идея моделирования одних алгоритмов другими избавляет нас от этого.

Рассмотрим оценки сложности алгоритма применительно к машинам Тьюринга.

С каждой конфигурацией k, к которой применима машина Тьюринга, можно ассоциировать число, характеризующее сложность процесса $M(k)$ в том или ином смысле.

Варьируя k, получим функцию от нее, определенную на множестве всех конфигураций, к которым данная машина применима. Функции такого рода мы называем *сигнализирующими функциями.*

Работу машины M будем характеризовать следующими сигнализирующими функциями:

Сигнализирующая времени $t_M(k)$ равна длительности процесса $M(k)$ (т.е. числу его конфигураций), если М применима к k, и $t_M(k)$ не определена, если M не применима к k.

Активной зоной процесса называется минимальная часть ленты, объемлющая активные зоны всех его конфигураций. Рабочей (активной) зоной данной конфигурации называется минимальная связная часть ленты, содержащая обозреваемую ячейку, а также все ячейки, в которых записаны значащие буквы.

Сигнализирующая емкости $s_M(k)$ равна длине активной зоны процесса $M(k)$, если M применима к k, и не определена в противном случае.

Сигнализирующая колебаний $\omega_M(k)$ равна числу колебаний (изменения направлений) головки за время $t_M(k)$, если M применима к k, и не определена в противном случае.

Сигнализирующая режима $r_M(k)$ равна максимальному числу прохождений головки через край ячейки за время $t_M(k)$, если M применима к k, и не определена в противном случае.

В частности, когда в качестве конфигураций k взяты начальные конфигурации для слова p, то процесс будет характеризоваться сигнализирующими функциями $t_M(p)$, $s_M(p)$, $\omega_M(p)$, $r_M(p)$. Наряду с ними рассматриваются и функции типа

$$t_M^{(n)} = \max_{|p|=n} t_M(p), \quad \omega_M^{(n)} = \max_{|p|=n} \omega_M(p);$$
$$s_M^{(n)} = \max_{|p|=n} s_M(p), \quad r_M^{(n)} = \max_{|p|=n} r_M(p),$$

где $|p|$ — длина слова p, а также функции типа

$$t_M^*(n) = \max_{\nu \leqslant n} t_M(\nu), \quad \omega_M^*(n) = \max_{\nu \leqslant n} \omega_M(\nu);$$
$$s_M^*(n) = \max_{\nu \leqslant n} s_M(\nu), \quad r_M^*(n) = \max_{\nu \leqslant n} r_M(\nu),$$

где ν — наибольшее из длин слов p_i.

Построение машины дает лишь верхние оценки сигнализирующих. Нахождение же нижних оценок является более сложным делом и требует специальной теории.

Имеет место теорема, показывающая, что при наличии верхней оценки, для какой-нибудь одной из сигнализирующих t, s, ω, r и заданных параметрах m и n (m — число символов внешнего алфавита, n — число символов внутреннего алфавита) можно получить верхние оценки для остальных.

Теорема. Для любой машины M и для любого слова p, к которому она применима, справедливы следующие неравенства:

$$\omega(p) \leqslant t(p);$$
$$r(p) \leqslant \omega(p) + 1;$$
$$s(p) \leqslant |p| + 2n^{r(p)+1};$$
$$t(p) \leqslant m^{s(p)} \cdot s(p) \cdot n.$$

Сигнализирующий оператор T ставит в соответствие каждой одноместной функции $\varphi_i(x)$ ее сигнализирующую $\Phi_i(x)$ ($\Phi = t, s, \omega, r$).

Нумерационный функционал $\mu(i)$ называется критерием сложности, а для фиксированного i число $\mu(i)$ называется сложностью описания функции φ_i, если выполнены условия:

1) $\mu(i)$ — всюду определенная эффективная функция;

2) при любом фиксированном a уравнение $\mu(i) = a$ имеет конечное число решений, причем существует алгоритм, посредством которого для любого a все решения могут быть эффективно найдены.

1.7. Формальные преобразования алгоритмов

Одним из основных вопросов, возникающих в процессе преобразования алгоритмов, является их эквивалентность. Напомним, что два алгоритма считаются эквивалентными, если они имеют одну и ту же область определения, реализуемые ими функции совпадают, а система правил различна.

Можно определить сильную и слабую эквивалентность алгоритмов. Два алгоритма будем называть слабо эквивалентными, если они имеют одну и ту же область определения и результаты переработки одинаковых слов из этой области совпадают.

Два алгоритма будем называть сильно эквивалентными, если они имеют одинаковую область определения и совпадают не только результаты переработки слов из этой области, но и сам процесс их переработки.

В «чистом виде» понятие эквивалентности двух алгоритмов U_1 и U_2 будем определять следующим образом. Для каждого алгоритма вводится понятие «входа» и «выхода». Для каждого входа, который имеет смысл для данного алгоритма, выполнение алгоритма может приводить к некоторому выходу. Пусть x_1 и x_2 — входы алгоритмов U_1 и U_2 соответственно. Алгоритмы U_1 и U_2 считаются эквивалентными, если из условия $x_1 = x_2$ следует, что если хотя бы один алгоритм, например U_1, имеет выход y_1 то и другой алгоритм также имеет выход y_2, причем $y_1 = y_2$.

Таким образом, эквивалентность «в чистом виде» имеет место тогда, когда равенство входов приводит к равенству выходов. На практике большое значение имеет более общий способ введения эквивалентности, а именно определение эквивалентности с точностью до изоморфизма. В этом случае для выяснения вопроса об эквивалентности сопоставляются друг с другом не равные входы и выходы, а лишь находящиеся в соответствии, задаваемом изоморфизмом.

В этом случае между некоторыми входами x_1 алгоритма U_1 и входами x_2 алгоритма U_2 конструктивно задается некоторый изоморфизм I (в данном случае понимаемый просто как однозначное соответствие). Аналогично задается некоторый изоморфизм J между выходами y_1 и y_2 алгоритмов U_1 и U_2 соответственно.

Предикат от двух конструктивных объектов A и B «быть в соответствии, заданном изоморфизмом L» будем обозначать через $A_L \sim B$.

Эквивалентность теперь будет определяться следующим образом. Алгоритмы U_1 и U_2 считаются эквивалентными, если из условия $x_1 \underset{I}{\sim} x_2$ следует, что если хотя бы один алгоритм имеет выход y_1, то и другой алгоритм также имеет выход y_2, причем $y_1 \underset{I}{\sim} y_2$.

Разные виды эквивалентности отличаются тем, как определяются «входы» и «выходы» алгоритма, а также тем, как фактически выбираются изоморфизмы I и J.

Между различными видами эквивалентности можно ввести частичное отношение порядка, выражающееся словами «сильнее» и «слабее». Будем считать, что отношение эквивалентности $Э_1$ слабее отношения эквивалентности $Э_2$, если любые алгоритмы U_I и U_2, эквивалентные в смысле $Э_2$, эквивалентны и в смысле $Э_1$ и в то же время есть хотя бы одна пара алгоритмов $\tilde{U}_1$, $\tilde{U}_2$, таких, что $\tilde{U}_1$, $\tilde{U}_2$, которые эквивалентны в смысле $Э_1$ и не эквивалентны в смысле $Э_2$.

Очевидно, чем слабее отношение эквивалентности, тем шире классы алгоритмов, эквивалентных согласно этому отношению. Естественным является стремление расширить классы эквивалентных алгоритмов, вводя более слабое определение эквивалентности. Однако при слишком слабом определении эквивалентности массовые проблемы, возникающие в теории алгоритмов, и среди них основная проблема распознавания эквивалентности алгоритмов, могут оказаться неразрешимыми. С другой стороны, слишком сильное определение эквивалентности чрезмерно сужает классы эквивалентных алгоритмов.

Правильный выбор понятия эквивалентности играет большую роль как с точки зрения возможности получения содержательных теорем, так и с точки зрения их практической применимости.

На степень широты понятия эквивалентности наиболее существенно влияет выбор определения «выхода» алгоритма. Не во всех случаях нас могут интересовать в качестве выхода только те конструктивные объекты, которые формально объявляются результатами по окончании выполнения алгоритма. В некоторых случаях для нас является важной информация о каких-либо промежуточных результатах или о том, в какой последовательности выполнялись элементарные акты алгоритма и т. д. Поэтому в самом общем смысле под выходом алгоритма следует понимать какую-то запись всей той информации, которую можно получить, наблюдая процесс выполнения алгоритма.

Таким образом, чем больше информации, полученной в ходе выполнения алгоритма, несет в себе выход, тем более сильным оказывается отношение эквивалентности, основанное на таком определении выхода. Другими словами, все разнообразие эквивалентных алгоритмов состоит в том, что один и тот же конечный результат получается разными путями. Следовательно, чем больше истории о том, как выполнялся алгоритм, содержит в себе выход, тем к более сильному понятию эквивалентности он приводит. Это объясняется тем, что в один класс эквивалентных алгоритмов попадают только те из них, история выполнения которых соответствует истории, зафиксированной в данном выходе.

Исчерпывающую информацию о том, как происходило выполнение алгоритма, можно получить, рассматривая в качестве выхода алгоритма всю последовательность выполнявшихся операторов — значение операторного алгоритма. В этом случае алгоритмы считаются эквивалентными, если их значения совпадают при одинаковых входах. Такое определение эквивалентности рассматривалось Ю.И.Яновым, для операторных алгоритмов Ляпунова. В этом случае им была доказана разрешимость проблемы эквивалентности. Однако такое определение эквивалентности является слишком сильным и многие алгоритмы, которые разумно считать эквивалентными, при таком определении оказываются неэквивалентными.

Исходя из этого в качестве выхода операторного алгоритма целесообразно рассматривать нечто такое, что по количеству информации о ходе выполнения алгоритма было бы чем-то средним между значением алгоритма и значением результативного переменного по окончании выполнения алгоритма. С этой точки зрения будем рассматривать в качестве выхода *S*-представления тех переменных, значения которых нас интересуют в качестве результатов выполнения операторного алгоритма.

Если ввести определения эквивалентности, базирующееся на использовании *S*-представления в качестве выхода, то тогда можно сказать, что два алгоритма эквивалентны по отношению к выделенным переменным в том случае, если соответствующие переменные при совпадающих входах вычисляются по одинаковым формулам. *S*-представление переменного есть не что иное, как явное выражение формулы, по которой вычислялось результирующее значение переменного для данных исходных значений функциональных переменных.

Как известно, в программировании важную роль играют именно такие преобразования алгоритмов, которые оставляют расчетные формулы (*S*-представления) неизменными. Действительно, такие основные приемы программирования, как разбиение задачи на части, расчленение формул, выделение промежуточных результатов, преобразование логических операторов, экономия команд и ячеек памяти, приводят к таким преобразованиям алгоритмов, которые сохраняют *S*-представления результативных переменных.

Для определения эквивалентности операторных алгоритмов рассмотрим два алгоритма — U_1 и U_2 из некоторого класса операторных алгоритмов. Для каждого из алгоритмов выделим те переменные, которые нас интересуют в качестве результатов выполнения этих алгоритмов. Выделенные переменные могут быть различными, у алгоритма U_1 и алгоритма U_2, но число их должно быть одинаковым и между ними должно быть установлено взаимнооднозначное соответствие. То же самое относится к функциональным переменным алгоритмов U_1, U_2 и к параметрам этих алгоритмов, которые могут входить в *S*-представление выделенных

переменных. Обозначим функциональные переменные, параметры и выделенные переменные алгоритма U_1 буквами $x_1, \ldots, x_S$, $p_1, \ldots, p_r$ и $y_1, \ldots, y_n$ соответственно. Соответствующие им переменные, играющие аналогичную роль для U_2, обозначим буквами $\tilde{x}_1, \ldots, \tilde{x}_s$;

$$\tilde{p}_1, \ldots, \tilde{p}_r \text{ и } \tilde{y}_1, \ldots, \tilde{y}_n.$$

Алгоритмы U_1 и U_2 эквивалентны по отношению к выделенным переменным, $y_1, \ldots, y_n$ и $\tilde{y}_1, \ldots, \tilde{y}_n$, если для любого набора исходных данных $x_1, \ldots, x_S$ имеет место следующее утверждение: если какой-либо из этих алгоритмов, например U_1, имеет значение для исходных данных $x_1, \ldots, x_S$ и S-представления выделенных переменных имеют вид:

$$\begin{aligned}
&T_1(x_{i11}, \ldots, x_{i1m1}, p_{j11}, \ldots, p_{j1e1}) \Rightarrow y_1;\\
&T_2(x_{i21}, \ldots, x_{i2m2}, p_{j21}, \ldots, p_{j1e1}) \Rightarrow y_2;\\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\\
&T_n(x_{in1}, \ldots, x_{inmn}, p_{jn1}, \ldots, p_{jnen}) \Rightarrow y_n,
\end{aligned}$$

то другой алгоритм — U_2 также имеет значение для исходных данных $\tilde{x}_1, \ldots, \tilde{x}_S$ и S-представления выделенных переменных имеют вид:

$$\begin{aligned}
&T_1(\tilde{x}_{i11}, \ldots, \tilde{x}_{i1m1}, \tilde{p}_{j11}, \ldots, \tilde{p}_{j1e1}) \Rightarrow \tilde{y}_1;\\
&T_2(\tilde{x}_{i21}, \ldots, \tilde{x}_{i2m2}, \tilde{p}_{j21}, \ldots, \tilde{p}_{j2e2}) \Rightarrow \tilde{y}_2;\\
&\ldots\ldots\ldots\ldots\ldots\ldots\ldots\\
&T_n(\tilde{x}_{in1}, \ldots, \tilde{x}_{inmn}, \tilde{p}_{jn1}, \ldots, \tilde{p}_{jmen}) \Rightarrow \tilde{y}_n.
\end{aligned}$$

Можно также определить эквивалентность между двумя алгоритмами U_1 и U_2 из разных классов $U_I(\gamma_1, \varphi_1)$ и $U_2(\gamma_2, \varphi_2)$. Соответствие между функциональными переменными, параметрами и выделенными переменными устанавливаются так же, как и выше. Однако может оказаться, что списки операций φ_1 и φ_2, а также значения функциональных переменных не совпадают буквально. В этом случае между операциями из списков φ_1 и φ_2, а также между возможными значениями функциональных переменных из множеств γ_1 и γ_2 необходимо дополнительно задать определенные изоморфизмы и считать алгоритмы U_1 и U_2 эквивалентными в том случае, если изоморфные наборы исходных данных приводят к S-представлениям, совпадающим с точностью до изоморфизма образующих их операций.

Примером изоморфизма операций могут служить следующие представления арифметических операций:

$$\begin{aligned}
&(\;)+(\;) \quad \text{и} \quad (\;)+(\;);\\
&(\;)-(\;) \quad \text{и} \quad (\;)-(\;);\\
&(\;)\times(\;) \quad \text{и} \quad (\;)\times(\;);\\
&(\;)/(\;) \quad \text{и} \quad (\;)/(\;);
\end{aligned}$$

Следует отметить, что если вводится понятие эквивалентности для алгоритмов из разных классов, то может оказаться, что алгоритмы, эквивалентные в смысле S-представлений, будут неэквивалентны в смысле совпадения значений выделенных переменных. Это может произойти в том случае, если изоморфные операции не будут совпадать функционально. Например, в одном случае арифметические действия выполняются в двоичной системе, а в другом — в десятичной.

Оказывается, что проблемы эквивалентности и распознавания принадлежности алгоритма к некоторому классу алгоритмов в своей полной постановке алгоритмически неразрешимы. Поэтому до сих пор их решали только для некоторых видов алгоритмических систем при довольно узком определении эквивалентности. Основные результаты, полученные в этой области, относятся к операторным системам алгоритмизации, в частности к операторным схемам А.А.Ляпунова, называемым часто схемами программ. Характерные черты операторных схем состоят в следующем:

совокупность операторов, образующих схему алгоритма, изображается в схеме явно;

для каждого оператора явно указываются его приемники и предшественники по выполнению, а также его аргументы и результаты (входы и выходы оператора);

при построении реализации приемник оператора обычно выбирается произвольно, без учета истории движения к данному оператору;

если в рассмотрение вовлекается некоторая величина, «вырабатываемая» некоторым оператором, то она трактуется как независимая переменная, т.е. считается, что после выполнения данного оператора она может принимать любое значение независимо от предыдущей истории;

если аргументом или результатом оператора оказывается некоторая компонента массива, указываемая индексом, то конкретное значение индекса обычно игнорируется и считается, что аргументом или результатом оператора является весь массив.

Важное место в процессе преобразования алгоритмов занимают различные меры качества. Обычно они носят характер числовых функционалов, но в общем случае ими могут быть любые объекты, допускающие отношение порядка.

Меры качества определяются конкретными условиями постановки задачи, но они всегда связаны с «пространственно-временными» характеристиками алгоритма, т.е. со временем его выполнения и объемом памяти, требуемым для хранения перерабатываемой информации.

Первой работой, посвященной общей теории преобразования алгоритмов, явилась работа Ю.И.Янова «О логических схемах алгоритмов». В ней были сформулированы основные компоненты теории преобразования алгоритмов. К этим компонентам относятся: формализация понятия схемы алгоритма, задание отношения эквивалентности, нахождение алгоритма, распознающего эквивалентность схем, построение системы преобразований, полной в том смысле, что любую пару эквивалентных алгоритмов можно трансформировать один в другой последовательным применением этих преобразований, сохраняющих эквивалентность.

Всякий алгоритм при переработке конкретного объекта предписывает однозначно определенную последовательность элементарных действий. Такая последовательность, вообще говоря, различна для различных объектов, к которым данный алгоритм может быть применен. Однако всегда найдется конечное множество предикатов, характеризующих некоторые свойства перерабатываемых объектов, такое, что для данного алгоритма зависимость порядка выполнения элементарных операций от перерабатываемых объектов будет однозначной функцией этих предикатов. Такая функция может быть записана при помощи конечной строки, составленной из символов элементарных действий $A_1, A_2, \ldots, A_n$ (называемых операторами), предикатов и некоторых вспомогательных символов:

$$\underset{i}{\lfloor};\ \underset{i}{\rfloor}\ (i = 1, 2, \ldots),$$

называемых соответственно левой и правой полускобками. При этом строчка

$$A_{i1}A_{i2}\ \ldots\ A_{is}$$

означает, что должны быть выполнены последовательно операторы

$$A_{i1},\ A_{i2},\ \ldots,\ A_{is}.$$

Строчка

$$A_{i1}\alpha\ \underset{m}{\lfloor}\ A_{i2}\ \ldots\ \underset{m}{\rfloor}\ A_{i3},$$

где α — некоторый предикат, означает, что после выполнения оператора A_{i1} в случае, если $\alpha = 1$, должен быть выполнен оператор A_{i2}, стоящий непосредственно правее $\alpha\underset{m}{\lfloor}$, а если $\alpha = 0$, то оператор A_{i3}, стоящий справа от полускобки $\underset{m}{\rfloor}$.

Строчки такого вида будем называть схемными записями алгоритмов. Рассмотрим пример схемной записи нормального алгоритма. Пусть схема нормального алгоритма в некотором алфавите A имеет вид:

$$\begin{cases} P_1 \to Q_1; \\ \dots\dots\dots \\ P_s \to \cdot Q_s; \\ \dots\dots\dots \\ P_n \to Q_n, \end{cases}$$

где формула

$$P_s \to \cdot Q_s$$

является заключительной подстановкой. Обозначим через p_i предикат: «слово P_i входит в перерабатываемое слово», а через A_i — операцию замены первого вхождения слова P_i в перерабатываемом слове на слово $Q_i (i = 1, 2, \dots, n)$. Тогда строчка

$$\overline{\underset{n+1}{}} \dots \overline{\underset{n+s-1}{}} \overline{\underset{n+s+1}{}} \dots \underset{2n}{\lrcorner}\, p_1 \underset{1}{\llcorner}\, A_1 0 \overline{\underset{n+1}{}} \underset{1}{\lrcorner}\, p_2 \underset{2}{\llcorner}\, A_2 0 \overline{\underset{n+2}{}} \dots$$
$$\dots \underset{-1}{\lrcorner}\, p_s \underset{s}{\llcorner}\, A_s 0 \overline{\underset{n+s}{}} \dots \underset{n-1}{\lrcorner}\, p_n \underset{n}{\llcorner}\, A_n 0 \underset{2n}{\llcorner}\, \underset{n}{\lrcorner}\, \overline{\underset{n+s}{}}$$

есть схемная запись рассматриваемого нормального алгоритма. (0 означает тождественно ложный предикат).

Ясно, что для всякого слова R в алфавите A последовательность операций

$$A_{i_1},\ A_{i_2},\ \dots\ (1 \leqslant is \leqslant n;\ s = 1,\ 2,\ \dots)$$

над этим словом, которую предписывает схемная запись, совпадает с последовательностью тех преобразований слова R, которая возникает при применении к нему нормального алгоритма с данной схемой.

Аналогичным образом можно говорить о схемных записях алгоритмов, задаваемых программами универсальных вычислительных машин, которые в этом случае называются схемами программ.

Если в схемных записях алгоритмов отвлечься от содержательного смысла операторов, считая их элементарными символами, а предикаты соответственно независимыми логическими переменными, то полученные выражения мы будем называть *логическими схемами алгоритмов*.

Нетрудно убедиться, что один и тот же алгоритм при фиксированном множестве элементарных операций и предикатов может иметь различные логические схемы. Например, выражения

$$p_1 \vee \bar{p}_2 \underset{1}{\llcorner}\, \underset{2}{\lrcorner}\, A_1 p_2 \underset{2}{\llcorner}\, 0 \underset{3}{\llcorner}\, \underset{1}{\lrcorner}\, A_2 \underset{3}{\lrcorner}\, \text{и}$$
$$\bar{p}_1 \cdot p_2 \underset{1}{\llcorner}\, A_2 0 \underset{2}{\llcorner}\, \underset{1}{\lrcorner}\, \underset{3}{\lrcorner}\, A_1 p_2 \underset{3}{\llcorner}\, \underset{2}{\lrcorner}$$

являются схемными записями одного и того же алгоритма при любой подстановке конкретных операторов и предикатов соответственно вместо A_1, A_2 и p_1, p_2.

Будем считать, что значения логических переменных могут изменяться только в моменты выполнения операторов. При этом поскольку в логических схемах алгоритмов не содержится никакой информации о поведении значений логических переменных, то некоторые ограничения на возможности их изменения мы будем задавать извне с помощью так называемых распределений сдвигов, т.е. соответствий между операторами и множествами логических переменных.

Если задано какое-либо распределение сдвигов A_i — P_i, где A_i — операторы, а P_i — множества логических переменных ($i = 1, 2, \dots$), то будем считать, что в момент выполнения оператора A_i могут изменяться значения только тех логических переменных, которые входят в P_i.

Пусть имеется некоторая схема. Зададим для нее последовательность Δ_{s1}, Δ_{s2}, ..., Δ_{sm}, ... наборов значений логических переменных. Если считать, что при выполнении схемы они изменяются согласно этой последовательности, т.е. после выполнения m-го оператора логические переменные принимают набор значений Δs_{m+1}, то получим определенную последовательность выполненных операторов, которую будем называть значением схемы для данной последовательности наборов. Задание распределения сдвигов выделяет для каждой схемы из всех возможных последовательностей наборов множество допустимых последовательностей. Таким образом, допустимые последовательности отличаются значениями только тех переменных, которые данным распределением сдвигов постав лены в соответствие оператору, выполненному при предыдущем наборе.

Понятие равносильности схем при заданном распределении сдвигов требует совпадения значений равносильных схем для любой допустимой последовательности наборов. Это означает, что множество всех равносильных схем при любой подстановке конкретных операторов и предикатов (связь которых соответствует дан ному распределению сдвигов) переходит во множество схемных записей одного и того же алгоритма.

Ю.И.Яновым вводится система аксиом и правил вывода, описывающая понятие равносильности логических схем алгоритмов при данном распределении сдвигов. А.П. Ершовым теория логических схем алгоритмов Янова была перенесена на язык граф-схем, что позволило упростить аксиоматику преобразований. Вместо 14 аксиом Янова Ершов использует только 7 аксиом. Н.А.Криницкий расширил систему преобразований Янова, вводя в рассмотрение информационные связи между операторами. Оператор рассматривается как система функций, берущих свои аргументы из не которых входных величин и записывающих результаты в выходные величины оператора.

Дальнейшее расширение системы преобразований Янова шло в направлении преобразования не алгоритмов, а программ. Преобразование программ рассматривается в работах Н.А.Криницкого, Р.И. Подловченко и др.

Поскольку преобразование алгоритмов в изложении Янова является довольно трудным, то мы рассмотрим упрошенную систему преобразований, предложенную Ершовым.

Операторной схемой Янова называется конечный ориентированный граф, обладающий следующими свойствами. Вершины графа имеют не более двух выходящих на них стрелок. Вершины, из которых выходят по две стрелки (одна из стрелок некоторым образом помечается), называются распознавателями, остальные называются операторами. К одной из вершин ведет специальная входная стрелка.

Помеченные стрелки называются плюс-стрелками, а непомеченные — минус-стрелками. В графе имеется только одна вершина, не имеющая выходной стрелки, называемая конечным оператором.

Каждому распознавателю сопоставляется логическое условие α — произвольная функция алгебры-логики, зависящая от k логических переменных $p_1, \ldots, p_k$. Распознаватель R с сопоставленным ему условием а обозначается $R(\alpha)$.

Каждому оператору A, кроме конечного, приписывается некоторый набор

$$P \subseteq \{p_1, \ldots, p_k\}$$

логических переменных, называемый сдвигом по логическим переменным. Оператор A с приписанным ему сдвигом обозначается $A(P)$.

Совокупность сдвигов $P_1, \ldots, P_n$ всех операторов $A_1, \ldots, A_n$ операторной схемы Янова называется распределением сдвигов, приписанным схеме G.

Если для любого i $(1 \le i \le n)$ $P_i = \Lambda$ (пустое множество), то распределение сдвигов называется пустым; при $P_i = \{p_i, \ldots, p_n\}$ распределение сдвигов называется универсальным.

Все операторы одной операторной схемы считаются попарно различными. Операторная схема Янова G с операторами $A_1, \ldots, A_n$, логическими переменными $p_1, \ldots, p_k$ обозначается $G(A_1,$

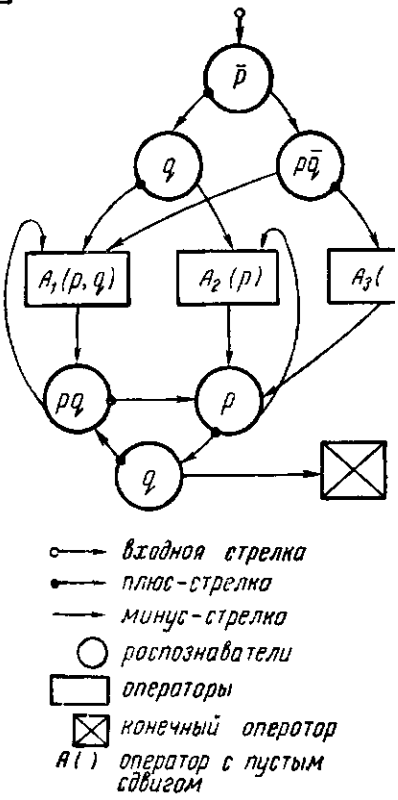


Рис.8

…, A_n, p_1, …, p_k). Пример изображения операторных схем Янова приведен на рис. 8. Распознаватель с логическим условием $p \wedge \bar{q}$ обозначается на схеме просто $p\bar{q}$.

Определим понятие эквивалентности операторных схем Янова. Пусть дана схема $G(A_1, \ldots, A_n, p_1, \ldots, p_k)$. Будем обозначать буквой A с индексом или без него упорядоченный двоичный набор значений переменных $p_1, \ldots, p_k$.

Для операторной схемы G определим понятие конфигурации, строящейся с помощью порождающего процесса блуждения по схеме G, задаваемого по индукции. Сама конфигурация будет получаться в виде двойной последовательности наборов значений логических переменных $p_1, \ldots, p_k$ и символов операторов $A_1, \ldots, A_n$, записываемых в виде двух строк: верхней для наборов и нижней для операторов.

При построении конфигурации основываемся на некотором базисе индукции, от которого осуществляются последующие шаги индукции. Их построение следующее.

Базис индукции. В верхнюю строку записывается произвольный набор Δ_1, нижняя строка пуста. Движение по схеме G начинается в направлении, указанном входной стрелкой.

Шаг индукции. Пусть происходит движение по схеме с набором Δ. Если стрелка приводит к распознавателю $R(\alpha)$, то вычисляется $\alpha(\Delta)$. Если $\alpha(\Delta) = 1$, то с тем же набором движение продолжается по плюс-стрелке, выходящей из распознавателя R. Если $\alpha(\Delta) = 0$, то движение происходит по минус-стрелке. Если стрелка приводит к оператору $A(P)$, то выписываем символ оператора A в нижнюю строку конфигурации и преобразуем набор Δ в набор Δ' с таким условием, чтобы набор Δ' отличался от Δ значениями только, может быть, тех переменных из $p_1, \ldots, p_k$, которые входят в сдвиг P оператора A.

Набор Δ' выписывается в верхнюю строку конфигурации, и движение продолжается по стрелке, выходящей из A.

Если стрелка приводит к конечному оператору, то его символ выписывается в нижнюю строку и построение конфигурации обрывается.

При движении по распознавателям возможен случай, когда, не попав ни на один из операторов, мы придем снова к недавно пройденному распознавателю. В этом случае движение по распознавателю станет бесконечным. При обнаружении такого цикла движение искусственно прерывается и в нижнюю строку ставится символ пустого периода ( ).

Таким образом, любая конкретная конфигурация имеет вид:

$$\begin{array}{cccc} \Delta_1 & \Delta_2 & \ldots & \\ A_{j1} & A_{j2} & \ldots, & \end{array}$$

при этом она либо является бесконечной, либо обрывается (на нижней строке) символами ⊠ или ( ).

Любая пара соседних наборов Δ_m, Δ_{m+1} может отличаться значениями только тех переменных, которые входят в сдвиг оператора A_{jm}.

Верхняя последовательность наборов называется допустимой последовательностью s наборов для схемы G, нижняя последовательность операторов называется значением z схемы G, определяемым допустимой последовательностью s.

Поскольку при построении конфигурации возможен произвол в выборе Δ_1, при переходе от Δ_m к Δ_{m+1} очевидно, что правило ставит в соответствие каждой схеме G некоторое множество конфигураций $K(G)$, являющееся в общем случае бесконечным.

Выпишем несколько конфигураций для приведенного примера операторной схемы Янова. Пусть $\Delta = 10$ (где первая переменная $p = 1$, а вторая $q = 0$), тогда примеры конфигураций имеют вид:

При $\Delta = 10$

$$\begin{matrix} 10 & & 10 \\ 1. \quad A_3(\ ) & & |\overline{\times}| \end{matrix}$$

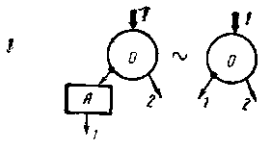
2. При $\Delta = 11$

$$\begin{matrix} 11 & 00 & 11 \\ A_1(p,\ q) & A_1(p,\ q) & |\overline{\times}| \end{matrix}$$

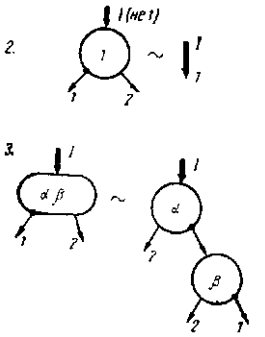
Конфигурацию, в которой при ее построении символы распознавателей выписываются в нижней строке так же, как и операторы, будем называть точной конфигурацией. Таким образом, точная конфигурация регистрирует прохождение всех вершин при движении по схеме. Очевидно, что между конфигурациями и точными конфигурациями одной схемы существует взаимно однозначное соответствие. Пример точной конфигурации для примера 1 имеет вид:

$$\begin{matrix} 10 & 10 & 10 & & 10 & 10 & 10 \\ \bar{p} & p\bar{q} & A_3(\ ) & & p & q & |\overline{\times}| \end{matrix}$$

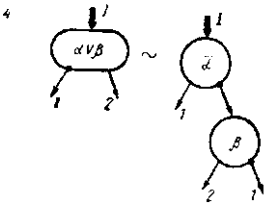
Отношение эквивалентности определяется для любой пары операторных схем G_1 и G_2 с одними и теми же логическими переменными $p_1, \ldots, p_k$. Схемы G_1 и G_2 являются эквивалентными ($G_1 \simeq G_2$), если $K(G_1) = K(G_2)$, т.е. если совпадают их множества конфигураций.



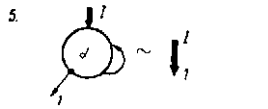
Эквивалентность схем обладает свойствами рефлексивности ($G_1 \simeq G_1$) и симметричности (если $G_1 \simeq G_2$, то $G_2 \simeq G_1$). Как указывалось, система правил преобразований операторных схем Янова построена в виде логического исчисления, т.е. совокупности аксиом и правил вывода.



Аксиомы в этом исчислении постулируют безусловную возможность замены некоторого фрагмента операторной схемы на другой фрагмент, предписываемый данной аксиомой.



Правила вывода постулируют возможность замены одного фрагмента на другой при выполнении некоторых условий, содержащихся в посылке данного правила вывода.



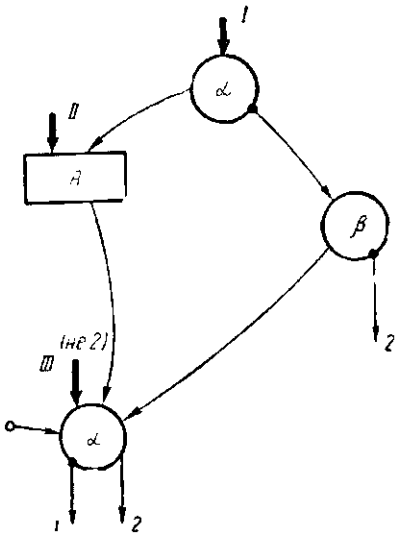
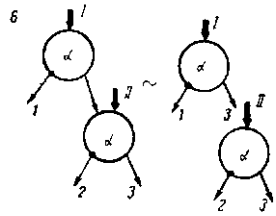
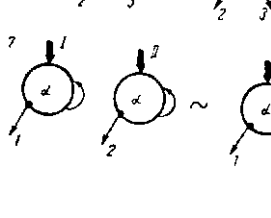


Рис.9



Фрагментом операторной схемы G называется некоторая часть (подграф) схемы G, для которой указана ее связь с остальными вершинами схемы G. Эта связь указывается в виде входов и выходов фрагмента.



Входы фрагмента — это различные стрелки, показывающие, к каким вершинам фрагмента ведут стрелки от остальных вершин схемы. Входы фрагмента могут быть указаны также в виде обобщенного входа, который указывает, к какой вершине может вести произвольная совокупность стрелок, выходящих либо от остальных вершин схемы, либо (если это не запрещается специальной оговоркой) от выходов этого же фрагмента. Обобщенные входы указываются жирными стрелками. Выходы фрагмента — это различные стрелки, которые показывают, какие вершины фрагмента

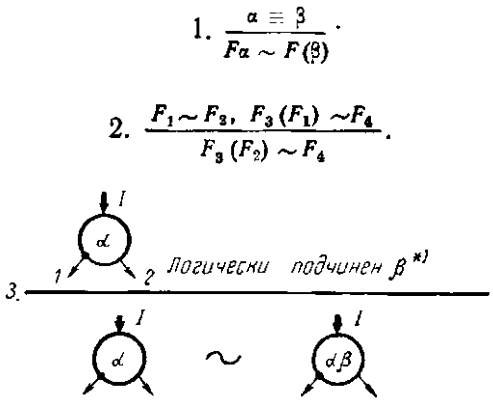
соединены с остальными вершинами схемы. Входы метятся римскими цифрами, выходы — арабскими.

Поясним сделанные определения на примере фрагмента, представленного на рис. 9.

Как видим, к верхнему распознавателю обязательно подходит некоторая стрелка. В оператор А извне могут входить любые стрелки. В нижний распознаватель могут вести извне любые стрелки, но среди них не может быть ни одна из выходных стрелок фрагмента, помеченных Рис.10 цифрой 2. Пометка двух выходных стрелок одной и той же цифрой означает, что обе эти стрелки должны вести к одной и той же вершине схемы.

Сами аксиомы и утверждение правил вывода имеют вид соотношений равносильности, т.е. двух фрагментов, разделенных знаками равносильности ~. Аксиомы и правила вывода представлены на рис. 10.

Правила имеют вид:



В правилах вывода над чертой записывается посылка, а под чертой утверждение. Буквой F обозначаются произвольные фрагменты, Λ — пустые фрагменты; вместо конъюнкции в условиях ставится подразумеваемая точка.

Дальнейшее усовершенствование аксиоматики связано с третьим правилом вывода. Проанализируем его и уточним его содержание.

Полным условием работы некоторой вершины V в схеме $G(p_1, \ldots, p_k)$ называется логическая функция U_V, равная единице на тех и только тех наборах значений $p_1, p_2, \ldots, p_k$, которые являются допустимыми для вершины V. Набор Δ является допустимым для V, если для схемы G существует такая точная конфигурация, в которую входит пара (Δ_V). Логическая функция β логически подчиняет распознаватель $R(\alpha)$, если для β тождественно выполняется импликация $U_R \rightarrow \beta$. Выполненные условия логической подчиненности гарантируют, что при любом возможном выполнении схемы G логическое условие α будет вычисляться только для таких Δ, для которых $\beta(\Delta) = 1$, что и позволяет заменить а на β*.

Применение любого правила вывода предполагает предварительную проверку его посылки. «Простота» этой проверки является основным требованием при построении элементарных преобразований. Понятие простоты не уточняется, поскольку проверка посылок обычно производится содержательными средствами. Обычно считается, что критерий «простоты» выдержан, если процесс проверки посылки сводится к выяснению графического совпадения или несовпадения конструктивных объектов, упоминаемых в правиле вывода. Правила вывода *1* и *2* удовлетворяют этому интуитивному представлению о простоте, поскольку при попытке применить правило *2* к двум отношениям равносильности $F_1 \sim F_2$ и $F_3 \sim F_4$ нужно только обнаружить, содержится ли F_1 в F_3,

---

* Некоторый распознаватель R (α) подчинен логической функции β, если проверка Rα при выполнении схемы производится только для тех значений логических переменных, на которых β истина.

а для применения правила *1* нужно обнаружить тождественность функций α и β , что легко сделать, если считать, что α и β представлены в совершенной нормальной форме. Если же отказаться от ограничений на форму представления α и β, то все же этот процесс может быть формализован с помощью аксиоматики.

Для применения правила *3* необходимо знать для любого распознавателя *R* полное условие его работы U_R. Для нахождения U_R Яновым предложен соответствующий алгоритм. Таким образом, хотя применение правила *3* является в принципе эффективным, тем не менее оно в некотором смысле некорректно, так как явно не удовлетворяет критерию «простоты» его применения.

Для применения правила *3* необходимо знать для любого распознавателя *R* полное условие его работы U_R. Для нахождения U_R Яновым предложен соответствующий алгоритм. Таким образом, хотя применение правила *3* является в принципе эффективным, тем не менее оно в некотором смысле некорректно, так как явно не удовлетворяет критерию «простоты» его применения.

Поскольку знание полных условий работы распознавателей является необходимым этапом преобразований операторных схем Янова, единственной возможностью усовершенствовать правило *3* является формализация процесса построения полных условий работы с тем, чтобы эти условия появлялись в преобразуемой схеме как результат применения аксиом и правил вывода. Для этого правило *3* заменяется четырьмя новыми аксиомами и одним правилом вывода. Правила *1* и *2* остаются без изменений, аксиомы I—VII подвергаются незначительной стилистической переработке. Окончательный вид аксиом и правил вывода приводится на рис. 11.

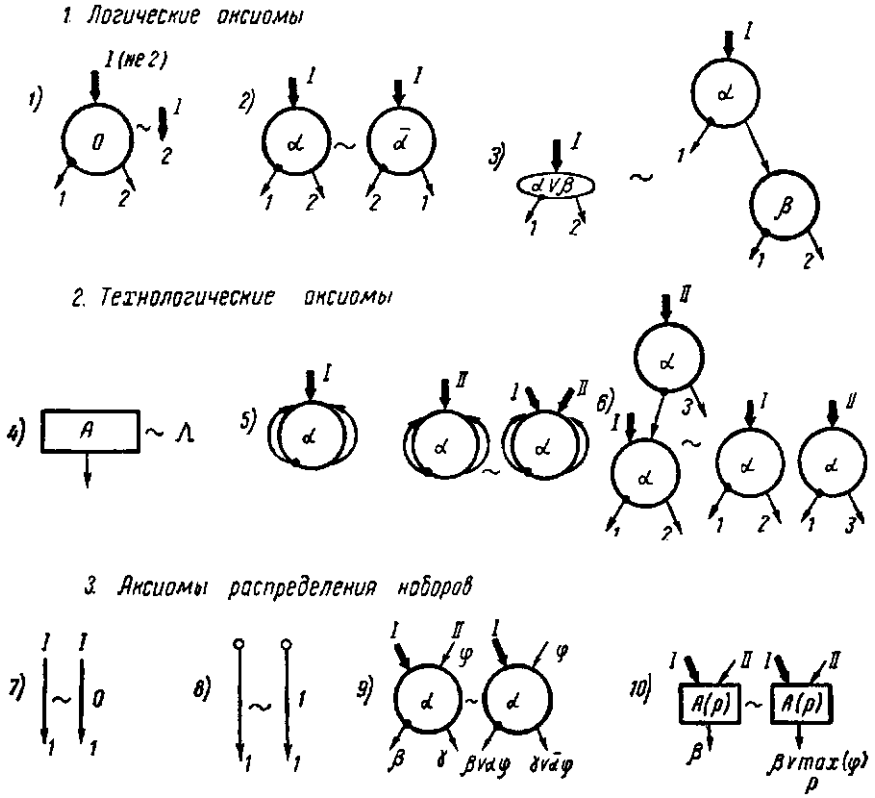


Рис. 11

Правила:

$$1.\ \frac{\alpha \equiv \beta}{F(\alpha) \sim F(\beta)}.$$

$$2.\ \frac{F_1 \sim F_2,\ F_3(F_1) \sim F_4}{F_3(F_2) \sim F_4}.$$

3. $$\frac{\forall \left(\alpha : \varphi \to \beta \vee \gamma\right),\ \forall \left(\beta(\rho) : \varphi \to \gamma\right)(\max_{\rho}(\varphi) \to \gamma)}{\forall(\alpha : 1 \to 1,2)\ (\alpha : 1 \to 1,2 \sim \alpha\beta : 1 \to 1,2)}$$

Исследуем содержание этой аксиоматики. Логические аксиомы постулируют связь базисных логических функций (нуль, отрицание и дизъюнкция) с правилами выбора стрелок при выполнении распознавателей. Их справедливость очевидна. Топологические аксиомы постулируют возможность определенных топологических изменений схемы (т.е. изменений соединений между вершинами схемы и устранения вершин). Аксиома 4 утверждает, что оператор без входа может быть устранен, аксиома 5 постулирует эквивалентность пустых периодов, аксиома 6 постулирует сохранение значений логических переменных при движении по распознавателям схемы. Аксиомы распределения наборов формализуют понятие допустимости набора для распознавателя или преобразователя схемы. Будем трактовать логические функции $f(\Delta)$ как множества тех наборов Δ, на которых $f(\Delta) = 1$. Аксиомы группы III задают правила пометки стрелок операторной схемы некоторыми логическими функциями, представляющими те наборы, которые могут «проходить» вдоль стрелок при построении конфигураций схемы. Аксиома 7 носит служебный характер, задавая «начальное значение» метящих функций в виде пустого множества наборов. Аксиома 8 постулирует тот факт, что на входную стрелку операторной схемы может быть подан любой набор. Аксиома 9 утверждает, что если на вход распознавателя с условием α попадает некоторое множество наборов φ, то оно полностью проносится на выходы распознавателя; при этом те наборы из φ, на которых $\alpha = 1$, проносятся на плюс-стрелку, а остальные — на минус-стрелку. Аксиома 10 утверждает, что если на вход оператора A со сдвигом P попадает множество наборов φ, то оно порождает на выходе A множество всех наборов, которое может отличаться от наборов из φ только значениями переменных, входящих в сдвиг P.

Из аксиом распределения наборов видно, что при пометке стрелок схемы единственным безусловным источником наборов значений логических переменных является входная стрелка. Аксиома 9 переносит на выходные стрелки лишь те наборы, которые поступают на вход распознавателя. Аксиома 10 порождает лишь те наборы, которые могут получиться из наборов, поступающих на вход оператора, применением операции взятия максимума (заметим, что $\max_{p}(0) = 0$ для любого P). Таким образом, если в результате применения аксиом 7 — 10 на стрелке s, ведущей к некоторой вершине операторной схемы, появилось множество наборов φ, то это значит, что каждый из этих наборов мог появиться на стрелке s лишь в результате применения аксиом 7 — 10 к цепочке фрагментов схемы, образующих путь от входной стрелки до вершины V.

Поскольку операторная схема в целом является частным случаем понятия фрагмента, рассмотренная аксиоматика позволяет выводить соотношения равносильности для операторных схем. Свойства равносильности как некоторого бинарного соотношения выражает следующая лемма. Отношение равносильности является рефлексивным, симметричным и транзитивным.

Доказательство. Рефлексивность следует из правила *1* при одинаковых α и β. Симметричность следует из правила *2*, если в качестве посылки взять следующие соотношения: $F_1 \sim F_2,\ F_1 \sim F_2$. Доказательство транзитивности требует в качестве посылки соотношения $F_1 \sim F_2,\ F_2 \sim F_3$. По симметричности эту посылку можно переписать в виде $F_2 \sim F_1,\ F_2 \sim F_3$, откуда по правилу *2* вытекает, что $F_1 \sim F_3$.

Формальную возможность трактовать рассмотренную аксиоматику как правила преобразования операторных схем представляет следующая лемма (правило подстановки): если $F_1 \sim F_2$, то $F(F_1) \sim F(F_2)$.

Доказательство. Взяв в правиле *2* в качестве посылки $F_1 \sim F_2$, $F(F_1) \sim F(F_1)$, получим, что $F(F_2) \sim F(F_1)$, откуда в силу симметричности вытекает утверждение леммы.

Таким образом, справедлива следующая теорема: любые две равносильные операторные схемы Янова эквивалентны.

1.8. Алгоритмически неразрешимые проблемы

Одним из свойств алгоритма является его массовость. Это означает, что алгоритм представляет собой способ решения некоторой массовой проблемы, формулируемой в виде проблемы отображения не одного, а целого множества входных слов в соответствующие им выходные слова. Таким образом, всякий алгоритм можно рассматривать как некоторое универсальное средство для решения целого класса задач.

Оказывается, что существуют такие классы задач, для решения которых нет и не может быть единого универсального приема. Проблемы решения такого рода задач называют алгоритмически неразрешимыми проблемами. Однако алгоритмическая неразрешимость проблемы решения задач того или иного класса вовсе не означает невозможность решения любой конкретной задачи из этого класса. Речь идет о невозможности решения всех задач данного класса одним и тем же приёмом.

Таким образом, задачи (проблемы) можно разделить на алгоритмически разрешимые и алгоритмически неразрешимые. Примером алгоритмически разрешимой проблемы является проблема доказательства тождеств в обычной алгебре. Существует единый конструктивный прием (раскрытие скобок, приведение подобных членов), позволяющий за конечное число шагов решить, является ли любое заданное соотношение тождественным.

Переход от интуитивного понятия алгоритма к точному понятию рекурсивной функции (машины Тьюринга, нормального алгоритма) позволил доказать алгоритмическую неразрешимость ряда проблем.

Одним из первых результатов такого типа является доказательство неразрешимости проблемы распознавания выводимости в математической логике, выполненной Черчем в 1936г. Результат этого доказательства формулируется как теорема Черча (5). Проблема распознавания выводимости алгоритмически неразрешима. Тем самым не только выясняется причина безуспешности всех прошлых попыток создания соответствующего алгоритма, но и обнаруживается полная бессмысленность таких попыток.

Доказательство данной теоремы Черча рассматривать не будем ввиду его сложности. Отметим лишь, что суть его сводится к доказательству нерекурсивности функции, решающей данную задачу.

В качестве примера доказательства алгоритмической неразрешимости рассмотрим проблему распознавания самоприменимости. Существуют как самоприменимые, так и несамоприменимые алгоритмы. Примером самоприменимого алгоритма является так называемый тождественный алгоритм в любом алфавите A, содержащем две или более двух букв. Этот алгоритм применим к любому слову p в алфавите A и перерабатывает любое входное слово в себя.

Примером несамоприменимого алгоритма является так называемый нулевой алгоритм в любом конечном алфавите B. Этот алгоритм задается схемой, содержащей единственную подстановку $\rightarrow y$ (где y — любая буква алфавита B). По своему определению он не применим ни к какому входному слову, а значит, и к своему изображению.

Проблема распознавания самоприменимости алгоритмов состоит в том, чтобы найти единый конструктивный прием, позволяющий за конечное число шагов по схеме любого данного алгоритма узнать, является алгоритм самоприменимым или несамоприменимым.

Доказательство алгоритмической неразрешимости данной проблемы будем проводить, используя алгоритмические системы Тьюринга.

Для этого сформулируем проблему самоприменимости в терминах машины Тьюринга. Пусть в машине Тьюринга зафиксирована какая-нибудь конфигурация. Возможны два случая:

1) машина применима к этой конфигурации, т. е. после конечного числа тактов она останавливается в заключительной конфигурации;

2) машина не применима к этой конфигурации. Это означает, что машина никогда не переходит в заключительную конфигурацию, она впадает в бесконечный процесс.

Предположим теперь, что на ленте машины Тьюринга изображен ее собственный шифр (т.е. шифр таблицы соответствия и исходной конфигурации), записанный в алфавите машины. Если машина применима к такой конфигурации, то будем называть ее самоприменимой, в противном случае — несамоприменимой.

Тогда проблема распознавания самоприменимости состоит в следующем: по любому заданному шифру установить, к какому классу относится машина, зашифрованная им, — к классу самоприменимых или несамоприменимых?

Оказывается, можно доказать следующую теорему (6). Проблема распознавания самоприменимости алгоритмически неразрешима.

Доказательство. Предположим, напротив, что такая машина M существует. Тогда в M всякий самоприменимый шифр перерабатывается в какой-то символ ν (имеющий смысл утвердительного ответа на поставленный вопрос о самоприменимости), а всякий несамоприменимый шифр — в символ τ (имеющий смысл отрицательного ответа на поставленный вопрос).

В таком случае можно было бы построить и такую машину M_1, которая по-прежнему перерабатывает несамоприменимые шифры в τ, в то время как к самоприменимым шифрам M_1 уже не применима. Этого можно добиться путем такого изменения схемы машины (таблицы соответствия) M, чтобы после появления символа ν вместо остановки машина стала бы неограниченно повторять этот же символ.

Итак, M_1 применима ко всякому несамоприменимому шифру (вырабатывая при этом символ τ) и не применима к самоприменимым шифрам. Однако это приводит к противоречию.

Действительно:

1) пусть машина M_1 самоприменима, тогда она применима к своему шифру M_1' и перерабатывает его в символ τ, но появление этого символа как раз и должно означать, что машина несамоприменима;

2) пусть M_1 несамоприменима, тогда она применима к M_1', что должно означать, что M_1 самоприменима.

Полученное противоречие доказывает теорему, т.е. наше предположение о машине M неверно.

Из этой теоремы вытекает алгоритмическая неразрешимость более общей проблемы — проблемы распознавания применимости. Суть этой проблемы состоит в следующем: установить применимость любой заданной машины к любой заданной конфигурации.

Доказательства теоремы о самоприменимости алгоритма можно выполнить, используя нормальные алгоритмы. В терминах нормальных алгоритмов проблема самоприменимости формулируется следующим образом.

Проблема распознавания самоприменимости алгоритмов состоит в том, чтобы найти единый конструктивный прием, позволяющий за конечное число шагов по схеме любого данного алгоритма A узнать, является алгоритм A самоприменимым или несамоприменимым.

Если считать справедливым принцип нормализации, можно предполагать, что единый конструктивный прием — это не что иное, как нормальный алгоритм B, определенный на любом слове p, которое является изображением произвольного нормального алгоритма A, и переводящим это слово в два различных фиксированных слова q_1 и q_2 в зависимости от того, будет алгоритм A самоприменимым или несамоприменимым (слово q_i соответствует самоприменимости, а q_2 — несамоприменимости).

На любом входном слове l, не являющемся изображением какого-либо (нормального) алгоритма, алгоритм B также должен быть определен. Действительно, в противном случае, не получив никакого результата после достаточно большого числа шагов работы алгоритма,

неизвестно было бы, изображением какого алгоритма — самоприменимого или несамоприменимого — является слово *l*. Очевидно, что результат применения алгоритма *B* к любому слову, не являющемуся изображением алгоритма, должен быть отличным как от слова q_1, так и от слова q_2.

Предположим, что алгоритм *B* с указанными свойствами существует. В таком случае существует нормальный алгоритм *C* в том же самом алфавите *X,* что и алгоритм *B,* определенный на всех тех и только тех словах в алфавите *X,* которые являются изображениями несамоприменимых алгоритмов (напомним, что по самому определению алгоритма *B* алфавит *X* включает в себя стандартный двоичный алфавит).

Действительно, построим нормальный алгоритм *D* в алфавите *X,* область определения которого состоит из одного лишь слова q_2. Такой алгоритм может быть задан, например, в виде суперпозиции двух нормальных алгоритмов D_1 и D_2, первый из которых задается схемой, состоящей из одной подстановки $q_2 \rightarrow \cdot$, а второй — схемой, состоящей из подстановок вида $x_i \rightarrow x_i$, где x_i последовательно принимает значения всех букв алфавита *X*. Ясно, что первый алгоритм переводит в пустое слово только слово q_2, а область определения второго алгоритма состоит только из пустого слова. Поэтому область определения суперпозиции *D* алгоритмов D_1 и D_2 будет состоять только из слова q_2, что и требуется.

Построив алгоритм *D,* образуя суперпозицию с ним алгоритма *B* и нормализуя эту суперпозицию, придем к нормальному алгоритму *C* в алфавите *X,* область определения которого состоит из всех тех и только тех слов в алфавите *X,* которые являются записями несамоприменимых алгоритмов. Однако подобное свойство алгоритма *C* внутренне противоречиво, поскольку к своему собственному изображению C^U алгоритм *C* не может быть применим и неприменимым.

В самом деле, в первом случае алгоритм *C* был бы применим к своему изображению и являлся бы поэтому самоприменимым. Но это противоречило бы тому, что алгоритм *C* в силу своего построения должен быть применимым только к несамоприменимым алгоритмам. Во втором случае, будучи неприменимым к своему изображению, алгоритм *C* принадлежал бы к числу несамоприменимых алгоритмов. Но тогда по определению алгоритм *C* должен был бы быть применимым к своему изображению, поскольку он применим к изображению всех несамоприменимых алгоритмов. Следовательно, алгоритм *C* является самоприменимым.

Таким образом, предположение об алгоритмической разрешимости проблемы распознавания самоприменимости приводит к логическому противоречию и поэтому неверно. Тем самым показана алгоритмическая неразрешимость этой проблемы.

Вывод обоснован нами лишь при условии, что принцип нормализации алгоритмов справедлив. Однако природа противоречия, использованного при доказательстве алгоритмической неразрешимости проблемы распознавания самоприменимости алгоритмов, является в действительности более глубокой. Читатель, знакомый с парадоксами теории множеств и математической логики, легко заметит, что указанное противоречие имеет ту же природу, что и противоречие в известном парадоксе Рассела, устанавливающем внутреннюю противоречивость понятия «множество всех множеств, не содержащее себя в качестве своего элемента».

Это обстоятельство приводит к выводу, что алгоритмическая неразрешимость проблемы распознавания самоприменимости не является следствием узости современного точного понятия алгоритма. Если бы удалось построить точное понятие алгоритма, включающее в себя некоторые ненормализуемые алгоритмы, то проблема распознавания самоприменимости алгоритмов по-прежнему оставалась бы алгоритмически неразрешимой.

При доказательстве алгоритмической неразрешимости проблемы распознавания самоприменимости был использован метод «от противного». С помощью этого метода доказана алгоритмическая неразрешимость множества различных проблем, в том числе и проблема эквивалентности слов для ассоциативных исчислений.

Рассмотрим эту проблему более детально.

Условимся называть ассоциативным исчислением совокупность всех слов в некотором алфавите вместе с какой-нибудь конечной системой подстановок. Чтобы задать ассоциативное исчисление, достаточно указать соответствующие алфавит и систему подстановок.

Мы будем рассматривать преобразование одних слов в другие посредством некоторых допустимых подстановок, которые задаются в виде: p — q или $p \to q$, где p и q — слова в одном и том же алфавите.

Первая из этих подстановок называется неориентированной, вторая — ориентированной.

Ориентированная подстановка означает замену p на q, т. е. вместо левой части подставляется правая часть. Неориентированная подстановка означает допустимость замены левой части правой, и наоборот, правой части — левой (т. е. p на q или q на p).

Рассмотрим пример ассоциативного исчисления, заданного алфавитом $A = \{a, b, c\}$ и подстановкой ab — bcb. Рассмотрим слово $abcbcbab$. Замена каждого из двух вхождений ab дает слово $abcbcbab \to bcbcbcbab \to bcbcbcbbcb$. Замена же bcb дает слово $abcbcbab \to aabcbab \to aaabab$.

Если слово r может быть преобразовано в слово s посредством однократного применения допустимой подстановки, то и s может быть преобразовано в r таким же путем. Такие слова r и s будем называть смежными.

Два слова r и s называются смежными, если одно из них может быть преобразовано в другое посредством однократного применения допустимой подстановки.

Последовательность смежных слов, ведущую от r_1 к r_n, будем называть дедуктивной цепочкой от r_1 к r_n.

Слова r и s называются эквивалентными, если существует дедуктивная цепочка, ведущая от слова r к слову s, и наоборот, от слова s к слову r.

Эквивалентность будем обозначать как $r \sim s$. Эквивалентность слов а) рефлексивна, т. е. $r \sim r$; б) симметрична, т. е. если $r \sim s$, то $s \sim r$; в) транзитивна, т. е. если $s \sim r$, $r \sim t$, то $s \sim t$.

Для каждого ассоциативного исчисления возникает своя специальная проблема эквивалентности слов. Она заключается в следующем.

Для любых двух слов в данном исчислении требуется узнать, эквивалентны они или нет.

Поскольку различных слов в любом исчислении может быть бесчисленное множество, то фактически здесь имеется бесконечная серия однотипных задач. Требуется найти алгоритм, распознающий эквивалентность или неэквивалентность любой пары слов.

Проблема эквивалентности слов для ассоциативных исчислений была сформулирована еще в 1914г. норвежским математиком Туэ. Им же был предложен алгоритм для распознавания эквивалентности слов в некоторых ассоциативных исчислениях специального вида. С тех пор предпринимались многие попытки построить такой общий алгоритм, который для любого ассоциативного исчисления и для любой пары слов в нем позволил установить, эквивалентны эти слова или нет.

В 1946 и 1947 гг. А.А. Марков и Э. Пост, независимо один от другого, построили конкретные примеры ассоциативных исчислений, для каждого из которых проблема эквивалентности слов алгоритмически неразрешима. Тем более не существует алгоритма для распознавания эквивалентности слов в любом ассоциативном исчислении.

В 1955г. советский математик П.С. Новиков доказал алгоритмическую неразрешимость проблемы распознаваемости слов для ассоциативных исчислений специального вида называемых теорией групп.

Первые примеры, построенные А.А. Марковым и П.С. Новиковым для опровержения алгоритмической разрешимости проблемы эквивалентности, были весьма громоздкими и насчитывали сотни допустимых подстановок.

Позднее советским математиком Г.С. Цейтиным был построен пример ассоциативного исчисления, насчитывающего всего лишь семь допустимых подстановок, для которого проблема эквивалентности слов была также алгоритмически неразрешима.

Пример этого ассоциативного исчисления мы и рассмотрим. Заданы: алфавит *A = {a, b, c, d, e}* и система допустимых подстановок: *ac— ca*, *ad — da*, *bc — cb*, *bd — db*, *abac — abacc*, *eca — ae*, *edb — be*.

Если взять слово *abcde* в этом исчислении, то к нему применима только третья подстановка и оно имеет только одно смежное слово *acbde*. Далее имеет место эквивалентность *abcde~cadedb*, что видно из следующей дедуктивной цепочки:

abcde→acbde→cabde→cadbe→cadedb.

Если же взять слово *aaabb*, то к нему неприменима ни одна из подстановок, и поэтому оно не имеет никаких смежных слов. Не существует и слов, отличных от *aaabb*, которые были бы ему эквивалентны.

Цейтин доказывает, что для данного ассоциативного исчисления не может быть единого алгоритма для распознавания эквивалентности любой пары слов.

Можно было привести еще много примеров алгоритмически неразрешимых проблем, с ними мы еще встретимся при изучении теории грамматик алгоритмических языков. К типу алгоритмических проблем относится и теорема Гёделя о неполноте арифметической логики. Прежде всего уточним основные понятия, используемые в теореме Гёделя.

Под арифметической логикой понимается совокупность аксиом и правил вывода теорем в элементарной теории чисел (теории положительных целых чисел). Логика называется полной, если в ней можно доказать истинность или ложность каждого утверждения. Логика называется непротиворечивой, если она свободна от противоречий.

Теорема Гёделя (7). Каждая адекватная ω — непротиворечивая арифметическая логика неполна. Теорема утверждает, что любая адекватная непротиворечивая арифметическая логика неполна, т.е. существует истинное утверждение о целых числах, которое нельзя доказать в такой логике.

Более того, Гёдель показал, что невозможно доказать непротиворечивость арифметической логики (даже неполной) теми методами, которые выразимы в самой этой логике.

Чтобы было понятно значение результатов, полученных Гёделем, вспомним, что первой аксиоматической системой была геометрия Евклида (III в. до н. э.). В основе евклидовой геометрии лежит совокупность определений и аксиом, отражающих простейшие геометрические свойства, подтвержденные многовековым человеческим опытом.

В число аксиом, которые в геометрии Евклида принимаются без доказательств, входит и аксиома о параллельных линиях. Она формулировалась Евклидом так: «Если две прямые, лежащие в одной плоскости, при пересечении их какой-нибудь третьей образуют внутренние односторонние углы, сумма которых меньше двух прямых, то эти прямые пересекаются по ту сторону от третьей прямой, на которой сумма указанных углов меньше двух прямых».

Эта аксиома по многим причинам смущала математиков. Она значительно сложнее других аксиом и по утверждаемому ею факту и по своей формулировке. Ведь даже для осознания ее смысла надо предварительно овладеть рядом сведений.

Аксиому о параллельных линиях нередко называли темным пятком в гениальном труде Евклида. Не была также доказана полнота и непротиворечивость евклидовой геометрии. И очень рано начались упорные попытки математиков освободить евклидову геометрию от всяких пятен.

С этой целью стремились доказать аксиому о параллельных. Пытались логически вывести ее утверждение из остальных аксиом Евклида. Такие попытки продолжались на протяжении двух с лишним тысячелетий. Почти все выдающиеся математики испытывали свои силы на решении этой проблемы (Декарт, Лейбниц, Гаусе и др).

Полную бесплодность попыток доказать аксиому о параллельных впервые строго научно установил великий русский математик Н.И. Лобачевский в XVIII веке. Он доказал, что утверждение этой аксиомы нельзя вывести из остальных аксиом Евклида.

Лобачевский постулировал особую неевклидову геометрию, названную его именем. Она отвергала истинность следующей аксиомы Евклида: «Если даны прямая и точка, не лежащая на ней, то существует только одна прямая, которая проходит через данную точку параллельно

данной прямой». В геометрии Лобачевского параллельных линий не существует. По отношению к этой новой геометрии евклидова геометрия является только частным случаем.

Риман доказал, что неевклидова геометрия Лобачевского непротиворечива, если непротиворечива евклидова геометрия, а Гильберт показал, что евклидова геометрия непротиворечива, если непротиворечива арифметика, т.е. элементарная теория положительных целых чисел.

Для доказательства непротиворечивости искали арифметическую логику, которая была бы полна, т.е. такую, в которой можно было бы вывести (как теоремы) все истинные утверждения о целых числах.

Гёдель же показал тщетность таких поисков и доказал алгоритмическую неразрешимость проблемы доказательства непротиворечивости арифметической логики. Таким образом, в своей теореме о неполноте Гёдель показал, что существует бесконечное число проблем элементарной теории чисел, решение которых невозможно никаким данным аксиоматическим методом.

Для доказательства теоремы Гёделя можно использовать машины Тьюринга. Для этого введем следующие определения.

Множество будем называть рекурсивно перечислимым, если существует машина Тьюринга — МТ, которая допускает все цепочки их этого множества — и только их. Множество называется рекурсивным, если Оно и его дополнение рекурсивно перечислимы.

Оказывается, существуют рекурсивно перечислимые множества, которые не рекурсивны. К числу таких множеств относится аксиоматическая система обычной арифметики. Известно, что множество теорем будет хотя и рекурсивно перечислимо, но не рекурсивно.

Действительно, формальная арифметика обладает следующим свойством: существует механическая процедура *f*, такая, что если M_i представляет собой какую-либо МТ с двумя выходами, которая допускает все теоремы Σ, выводимые из некоторого непротиворечивого множества аксиом этой теории, и запрещает отрицание всех этих теорем $(\overline{\sum})$, то *f(i)* есть формула формальной арифметики, которая не допускается и не отвергается машиной M_i.

Таким образом, не существует МТ с двумя выходами, которая допускала бы множество Σ и запрещала бы его дополнение $(\overline{\sum})$.

Глава 2.ОСНОВЫ ТЕОРИИ ФОРМАЛЬНЫХ ГРАММАТИК

2.1. Основные понятия грамматик

За последние десять лет появилось большое количество работ по общей теории языков и грамматик.

Можно выделить четыре научных направления, связанных с четырьмя постановками различных задач, которые до этого развивались самостоятельно и, казалось, были мало связаны между собой. В последние годы общность этих задач и методов их исследования стала очевидной, их удалось объединить и понять как одну общую задачу теории языков.

Первое из этих направлений связано с построением формальной или математической лингвистики. Возникнув первоначально в связи с формальным изучением структуры литературного языка, математическая лингвистика начала особенно быстро развиваться, когда были сформулированы, например, такие вопросы машинного перевода: всякая ли фраза может быть переведена с одного языка на другой при заданном запасе правил в памяти машины; как описать множество текстов, доступных для перевода при заданных условиях. Попытка не только ответить, но даже точно поставить вопросы такого рода сразу же требовали формализации понятий «словарь», «грамматика», «язык», подразделения и классификации этих понятий и умения относить конкретные словари, грамматики, языки к тому или иному классу. Это послужило стимулом для большого числа работ, исследующих указанные понятия с позиций формальной лингвистики.

Появление искусственных машинных языков программирования еще более усилило интерес к задачам такого рода. Однако принятый ранее в лингвистике изобретательский (эвристический) подход к новым языкам, при котором оценка удобства языка являлась делом интуиции и опыта, оказался явно недостаточным для машинных языков. Положение усугубилось появлением трансляторов, которые сделали проблему перевода центральной, во многом определяющей построение общей теории вычислительных машин. Сами же машинные языки стали все более приближаться к формально построенным математическим конструкциям.

Совершенно независимо развивалось направление науки, связанное с построением формальных моделей динамических систем. Эти модели должны были быть, с одной стороны, достаточно узкими, чтобы допускать построение продуктивной теории, с другой — достаточно широкими, чтобы охватить некоторый общий класс прикладных задач.

Типичным примером такого рода является модель «конечный автомат». Охватывая многие процессы, заданные на конечных множествах и развивающихся в счетном времени, конечные автоматы оказались вместе с тем настолько узкой моделью, что для них удалось создать продуктивную теорию. Однако как только в эту модель вводилась бесконечность где-либо, кроме шкалы времени, это немедленно приводило к слишком общему классу систем (машины Тьюринга), эквивалентному столь широкому понятию, как произвольный алгоритм.

Это породило многочисленные попытки построить промежуточные модели динамических систем, более широкие, чем конечный автомат, но более узкие, чем машины Тьюринга, и как-либо классифицировать эти модели. Была очевидна глубокая связь всех этих вопросов с теорией алгоритмов. Сама же постановка вопроса шла не от математической абстракции «алгоритм», а скорее от механического формализма «динамическая система».

Независимо и параллельно развивалась общая теория алгоритмов как ветвь современной математики. Была установлена эквивалентность понятий «нормальный алгоритм Маркова», «общерекурсивная функция» и «машина Тьюринга», а тезис Черча связал эти три понятия с интуитивным представлением об алгоритме. Развитие вычислительной техники поставило перед математической теорией алгоритмов новую задачу: стало необходимо классифицировать алгоритмы, например, по вычислительной сложности. Эквивалентность понятий «алгоритм» и «машина Тьюринга» сделала естественным предположение о том, что поиски классификации алгоритмов окажутся в известной мере связанными с поисками промежуточных моделей между моделями конечного автомата и машиной Тьюринга.

Таким образом, перечисленные четыре направления оказались тесно связанными. Естественно поэтому, что теория языков, порожденная чисто лингвистическими задачами,

оказалась в центре интересов математиков, занимающихся теорией алгоритмов, и ученых, занимающихся абстрактными моделями динамических систем, а в последние годы — в центре интересов ученых, разрабатывающих теоретические основы автоматики.

Теория формальных грамматик и языков является основным разделом математической лингвистики — специфической математической дисциплины, ориентированной на изучение структуры естественных и искусственных языков.

Эта теория возникла в 50-е годы в работах американского лингвиста Н. Хомского (хотя основы аппарата, на котором она базируется, были разработаны в математической лингвистике значительно раньше). При этом Хомский исходил исключительно из потребностей лингвистики, по традиции, понимаемой как наука о строении естественных языков. Однако очень скоро обнаружилось, что методы его теории в не меньшей степени приложимы к искусственным специализированным языкам, и в частности к алгоритмическим. Это объясняется тем, что алгоритмические языки проще устроены и лучше поддаются формализации, чем естественные языки.

По характеру используемого математического аппарата теория формальных грамматик и языков близка к теории алгоритмов и к теории автоматов.

Под грамматикой понимается некоторая система правил, за дающая множество цепочек (конечных последовательностей) символов языка.

Эти цепочки могут интерпретироваться как языковые объекты разных уравнений, например как словоформы, словосочетания и предложения.

Словоформа или просто слово — (последовательность) цепочка морфем (морф). Морфема — мельчайшая грамматически значимая часть слова (словоформы).

Так, например, словоформа *ведший* состоит из морфем *вед* + *ш* + *ий* (+ — граница морфем). Где *вед* — корень, *ш* — суффикс, *ий* — окончание.

Словосочетание и предложение — цепочка словоформ.

Таким образом, грамматика языка — конечное множество правил, определяющих этот язык. Грамматику языка можно рассматривать как теорию структуры этого языка, т.е. теорию повторяющихся закономерностей построения предложений, называемых синтаксической структурой языка.

Синтаксис языка — правила построения предложений в языке, или правила построения конструкций языка. Семантика языка — толкование этих правил, правила использования синтаксиса. Итак, грамматика языка это конечное множество правил, рекурсивно задающих язык как синтаксическую структуру.

Грамматика играет роль повторяющихся закономерностей построения предложений языка, которые отражают синтаксическую структуру его. Отсюда первое и основное требование к формализуемой грамматике — чтобы она приписывала каждому предложению языка его структурное описание. Это правило определяет, из каких элементов построено предложение, каков их порядок, расположение.

Структурное описание будет тогда и только тогда однозначно, когда в языке через грамматику однозначно определены его синтаксические единицы и иерархическая взаимосвязь между ними.

Второе требование к грамматике языка — грамматика должна быть конечной, т. е. если допустить грамматики с неопределенным множеством правил, то сама проблема построения грамматик снимается.

Формальные грамматики имеют дело с абстракциями, возникающими путем обобщения понятий словоформа, словосочетание, предложение.

Если обычные грамматики позволяют задавать множества правил построения предложений, то формальные грамматики — это не который способ изучать и описывать множества правил.

Между обычными и формальными грамматиками имеется существенное различие. В формальных грамматиках все утверждения формулируются в терминах небольшого числа четко определенных и весьма элементарных символов и операций. Это делает формальные грамматики сравнительно простыми с точки зрения их логического строения и облегчает изучение их свойств.

Однако формальные грамматики оказываются весьма громоздкими для описания естественного языка и поэтому предназначаются сугубо для научного, теоретического исследования наиболее общих свойств языка.

Различают распознающие, порождающие и преобразующие формальные грамматики.

Формальная грамматика называется распознающей, если для любой рассматриваемой цепочки она умеет решить, является эта цепочка правильной или нет, и в случае положительного ответа дать указания о строении этой цепочки.

Формальная грамматика называется порождающей, если умеет построить любую правильную цепочку, давая при этом указания о ее строении, и не строит ни одной неправильной цепочки.

Формальная грамматика называется преобразующей, если для любой правильно построенной цепочки она умеет построить ее отображения опять же в виде правильной цепочки, задавая при этом указания о порядке проведения отображений.

Рассмотрим класс порождающих грамматик. Порождающей грамматикой будем называть упорядоченную систему $G = (V_T, V_H, P, S)$, где V_T — конечное не пустое множество символов, называемое терминальным (основным) словарем G.

Терминальный словарь — набор исходных элементов, из которых строятся цепочки, порождаемые грамматикой, или словарь основных слов языка, из которых строятся предложения. V_H — конечное непустое множество символов, называется нетерминальным (вспомогательным) словарем G.

Нетерминальный словарь — набор символов, которыми обозначаются классы или цепочки исходных элементов, или словарь синтаксических типов.

Элементы V_T и V_H называются соответственно терминальными или нетерминальными символами. $V = V_T \cup V_H$ — конечное множество символов, называемое словарем грамматики G. Произвольную конечную последовательность ω элементов V будем называть цепочкой в словаре V. Пустая цепочка обозначается Λ, таким образом ωΛ = Λω = ω. Число членов этой конечной последовательности назовем длиной цепочки и будем обозначать |ω|.

Цепочки символов словаря V получаются с помощью операции соединения (конкатенации) $\frown$. Например, $\omega = \widehat{a_1 a_2 a_3}$. Если не возникает неоднозначности, символ $\frown$ может опускаться. Операция соединения ассоциативна, но не коммутативна.

$$(\widehat{a_1 a_2})\, \widehat{a_3} \sim \widehat{a_1 (a_2 a_3)}.$$

S — начальный символ. Это выделенный нетерминальный символ, обозначающий класс всех тех языковых объектов, для описания которых предназначается данная грамматика. В литературе символ S называется еще аксиомой, или целью грамматики.

P — правила грамматики или конечное множество цепочек вида φ→ψ, где φ, ψ — слова в словаре V, и цепочка φ содержит по крайней мере один символ из словаря V_H. Конечное, двуместное отношение → интерпретируется как «заменить φ на ψ», или «поставить ψ вместо φ». Это отношение обладает свойствами асимметричности и иррефлексивности. Цепочки вида φ→ψ называются правилами подстановки, или просто правилами грамматики. A множество P — схемой грамматики.

Если задана грамматика $G = (V_T, V_H, P, S)$, то будем говорить:

1. Цепочка ω получается (непосредственно) из цепочки ω' применением правила φ→ψ, если $\omega = \xi_1\varphi\xi_2$, $\omega' = \xi_1\psi\xi_2$, и $\{\varphi\to\psi\} \in P$.

2. Последовательность цепочек $\varphi = \varphi_0, \varphi_1, \varphi_2, \ldots, \varphi_n = \psi$ ($n \geq 1$) есть φ — вывод цепочки ψ, если для каждого i φ_{i+1} следует из φ_i, где $0 \leq i \leq n$. Наличие φ вывода цепочки ψ будем обозначать $\varphi \Rightarrow \psi$.

Количество применений правил в данном выводе называется его длиной. Длина вывода равна числу цепочек в нем, не считая начального символа.

3. Цепочка ψ выводима из φ, если она получается из φ применением некоторых правил грамматики G.

4. Вывод цепочки ψ считается законченным, если не существует цепочки, которая следует из ψ.

5. Цепочка, состоящая только из терминальных символов, называется терминальной цепочкой (каждая терминальная цепочка есть цепочка в словаре V_T, и ни к одному из символов V_T не применимо ни одно из правил грамматики G).

Множество цепочек, выводимых в грамматике G, называется языком, порожденным этой грамматикой, и обозначается $L(G)$.

Язык $L(G)$ называется терминальным, если L — множество терминальных цепочек грамматики G.

Условимся при написании конкретных грамматик, если это не оговорено особо, первыми строчными латинскими буквами *a, b, ...* обозначать элементы терминального словаря V_T, элементы нетерминального словаря V_H — прописными латинскими буквами *A, B, ...* а элементы общего словаря V — строчными греческими буквами α, β, ... Предложения, составленные из этих символов, будем обозначать последними буквами соответствующих алфавитов, т. е. *X, Y, ...* — предложения, составленные из элементов терминального словаря, *X, У, ...* — предложения из элементов нетерминального словаря, ω, φ, ψ, ... — предложения из элементов общего словаря.

Далее, везде, если к обозначению какого-либо множества добавлена сверху звездочка, например вместо V написано V^*, это означает, что имеется в виду множество всех цепочек, которые могут быть получены из символов множества V.

Рассмотрим введенные понятия на примере.

Пусть задана грамматика $G = (V_T, V_h, P, S)$, где

$$V_T = \{a, b\};$$
$$V_H = \{A, B, C\};$$
$$S = C;$$
$$P = \{C \rightarrow ab,\ C \rightarrow aCb\}.$$

Пусть

$$\omega = aCb,\ \omega' = aaCbb.$$

Цепочка ω' непосредственно выводится из цепочки ω в результате применения одного правила.

φ — вывод: *aCb, aaCbb, aaaCbbb, aaaabbbb*. Последняя цепочка является терминальной. Длина вывода равняется трем.

Из приведенного примера видно, что порождающая грамматика не является алгоритмом.

Правила подстановки грамматики — это не последовательность предписаний, а совокупность разрешений. Это означает:

1. Правило вида φ→ψ понимается в грамматике как «φ можно заменить на ψ» (а можно и не заменять), тогда как в алгоритме оно означало бы «φ следует заменить на ψ» (нельзя не заменить).

2. Порядок применения правил в грамматике произволен, тогда как в алгоритме был бы задан жесткий порядок применения от дельных инструкций.

Две грамматики — G_1 и G_2 — называются слабо эквивалентными, если они порождают один и тот же язык, $L_{G1} = L_{G2}$, или две грамматики называются слабо эквивалентными, если совпадает множество порождаемых ими фраз.

Две грамматики называются сильно эквивалентными, если они не только порождают одни и те же цепочки, но и приписывают одинаковым цепочкам одинаковые описания структуры. Другими словами, под сильной эквивалентностью понимается совпадение множества фраз вместе со способами их описания.

Основным объектом применения теории грамматик являются непроизвольные грамматики, а грамматики некоторых специальных типов, наиболее важных как в теоретическом, так и в практическом аспекте. Выделение этих типов производится по виду правил.

В теории Хомского выделяются и изучаются четыре типа языков, порождаемых соответственно четырьмя типами грамматик. Эти грамматики выделяются путем наложения последовательно усиливающихся ограничений на систему правил P.

Грамматика называется *грамматикой типа 0* в тех случаях, когда не накладывается никаких ограничений на правила $\varphi \to \psi$, где φ и ψ могут быть любыми цепочками из словаря V.

Грамматика называется *грамматикой типа 1*, если в системе P правила $\varphi \to \psi$ удовлетворяют условию $\varphi = \varphi_1 A \varphi_2$, $\psi = \varphi_1 \omega \varphi_2$, где A — нетерминальный символ, а φ, ψ, ω — цепочка из словаря V. Таким образом, в грамматиках типа *1* отдельный нетерминальный символ A переходит в непустую цепочку ω в контексте φ_1 и φ_2 (φ_1 и φ_2 могут быть пустыми цепочками). Грамматики типа *1* называют контекстными грамматиками.

Грамматика называется *грамматикой типа 2*— бесконтекстной, если в системе правил P допустимы лишь правила вида $A \to \omega$, где A — нетерминальный символ, а ω — непустая цепочка из V.

Согласно правилам этой грамматики, замена A на ω происходит независимо от контекста.

Грамматика называется грамматикой типа *3*, когда допустимы лишь правила вида $A \to \omega$,где $\omega = aB$, либо $\omega = a$.

Введенные здесь классы могут быть разбиты на подклассы. Такое разделение будет рассмотрено далее.

Упражнения:

1. Пусть $G = (V_T, V_H, P, S)$ — порождающая грамматика, где $V_T = \{a, d, e\}$, $V_B = \{B, C, S\}$, $P = \{S \to aB, B \to Cd, C \to e\}$. Выписать терминальные цепочки, порождаемые данной грамматикой, и определить длину их вывода.

2. Пусть $G = (V_T, V_B, P, S)$, где $V_T = \{a, d, e\}$, $V_H = \{B, C, S\}$, $P = \{S \to aB, B \to Cd, B \to dC, C \to e\}$. Определить терминальные цепочки, порождаемые данной грамматикой, и длину их вывода.

3. Для грамматики G известны ее общий словарь $V = \{A, B, C, D, E\}$ и схема правил $P = \{E \to DCD, E \to A, D \to BC, D \to C, A \to BB\}$. Определить состав терминального и нетерминального словарей, цель грамматики, построить язык $L(G)$ и определить длину выводов для каждой терминальной цепочки.

4. Определить, являются ли порождающими следующие грамматики:

a) $G = (\{A, B\}, \{S, D\}, P = \{S \to AB, S \to \to ASD, SD \to B, B \to AS\}, S)$;

b) $G = (\{A, B\}, \{S\}, P = \{S \to ASBAS, S \to AB, AS \to B\}, S)$;

c) $G = (\{B, C\}, \{A, S\}, P = \{S \to A, A \to B, A \to CA\}, S)$;

d) $G = (\{B, C\}, \{A, S\}, P = \{S \to A, A \to B, A \to CAC\}, S)$.

5. Дана грамматика $G = (V_T, V_H, P, S)$, где $V_T = \{A, B\}$, $V_H = \{S, D\}$, $P = \{S \to AB, S \to ADSB, D \to BSB, DS \to B, D \to \Lambda\}$. Показать, что цепочка *ABABBAB* принадлежит множеству $L(G)$.

6. Дана грамматика $G = (\{a, b, c, d, e\}, \{A, B, C, D, E\}, P = \{A \to ed, B \to Ab, C \to Bc, C \to dD, D \to aE, E \to bc\}, C)$. Определить, принадлежит ли множеству $L(G)$ цепочка *eadabcbc*.

7. Дано $V = \{C, S, a, b\}$ и $V_T = \{a, b\}$. Определить, является ли грамматикой четверка (V_T, V_H, P, S) для следующих множеств правил грамматики:

a) $P = \{C \to b, S \to aCb\}$;

b) $P = \{b \to a, C \to Sb\}$;

c) $P = \{C \to bCaC, CS \to \Lambda\}$;

d) $P = \{C \to bC, CS \to aS, S \to a\}$.

8. Пусть для каждого $n \geq 1 G = (\{a\}, \{S \to S^n\}, S)$. Показать, что каково бы ни было n, $L(G_n) = \varnothing$.

9. Задан терминальный словарь грамматики. Определить грамматики, порождающие следующие языки:

a) язык $a^n b^n a^n$ для $n \geqslant 1$;

b) язык a^{n^2} для $n \geqslant 1$;

c) язык $a^n b^{n^2}$ для $n \geqslant 1$.

2.2. Грамматики непосредственно составляющих

Неукорачивающие грамматики — грамматики, у которых для любого правила $\varphi \to \psi$ справедливо соотношение $|\varphi| \le |\psi|$. Правила такого вида также называются неукорачивающими.

Рассмотрим пример неукорачивающей грамматики. Пусть за дана $G = (V_T, V_H, P, S)$, где

$$V_т = \{a, b\},\ V_н = \{S\},\ P = \{F_1, F_2, F_3, F_4\}$$
$$F_1 : S \to aa, \quad F_3 : S \to aSa,$$
$$F_2 : S \to bb, \quad F_4 : S \to bSb.$$

Дано слово *bSb*. В результате подстановки любого из четырех правил мы получаем новое слово, длина которого не меньше длины исходного слова: *baab, bbbb, baSab, bbSbb*.

Грамматики непосредственно составляющих — грамматики с правилами вида: $\varphi A \psi \to \varphi \omega \psi$ *или* $A \to \omega$, где A — нетерминальный символ; ω — произвольная непустая цепочка.

Таким образом, в грамматиках непосредственно составляющих на каждом шаге вывода разрешается заменять только один символ. Очевидно, что НС-грамматика является неукорачивающей грамматикой. Рассмотренный пример — НС-грамматика.

Шаг вывода в неукорачивающей грамматике, состоящий в одновременной замене нескольких символов, может быть разбит на несколько шагов, каждый из которых осуществляет замену только одного символа, т.е. для любой неукорачивающей грамматики может быть построена эквивалентная ей НС-грамматика.

Поясним это на примере. Допустим, что в неукорачивающей грамматике имеет место правило вида $AB \to BA$, где A и B — нетерминальные символы.

Такое правило может быть заменено четырьмя правилами грамматики непосредственно составляющих:

$$AB \to 1B;$$
$$1B \to 12;$$
$$12 \to B2;$$
$$B2 \to BA,$$

где *1*, *2* новые нетерминальные символы, которые не встречаются ни в каких старых правилах. Последовательное применение этих правил равносильно применению правила $AB \to BA$, причем такая замена не может привести к появлению «лишних» выводов, поскольку символы *1*, *2* — новые.

Языки, порождаемые НС-грамматиками, называются НС-языками. Среди НС-грамматик различают левоконтекстные и правоконтекстные грамматики.

Левоконтекстной НС-грамматикой будем называть грамматику *с* правилами вида:

$$\varphi A \to \varphi \omega,$$

а *правоконтекстной* — грамматику с правилами вида:

$$A\varphi \to \omega\varphi,$$

где A — нетерминальный символ,
φ и ω — цепочки в алфавите $V = V_T \cup V_H$.

Язык, порождаемый левоконтекстной грамматикой G, будем называть левоконтекстным языком $L(G)$; язык, порождаемый правоконтекстной грамматикой G', — правоконтекстным языком и обозначать $L(G')$.

Относительно этих языков доказаны следующие теоремы:

Теорема (8). Класс левоконтекстных языков не совпадает с классом контекстно-свободных языков.

Теорема (9). Класс правоконтекстных языков не совпадает с классом контекстно-свободных языков.

Грамматика непосредственно-составляющих называется НС-грамматикой, *отмеченной справа* (НС — ОП-грамматикой), если каждое правило имеет вид:

$$\varphi A \psi \to \varphi\omega\psi = \varphi\omega' a\psi,$$

где $a \in V$.

НС-грамматика называется НС-грамматикой, отмеченной слева (НС — ОЛ-грамматика), если каждое ее правило имеет вид:

$$\varphi A\psi \to \varphi\omega\psi = \varphi a\omega'\psi,\ a \in V.$$

Имеют место теоремы:

Теорема (10). Для любой НС — ОП-грамматики можно построить слабо эквивалентную КС-грамматику.

Теорема (11). Для любой НС — ОЛ-грамматики можно построить слабо эквивалентную КС-грамматику.

Для изучения свойств НС-грамматик рассмотрим маленький фрагмент грамматики русского языка, заданного следующими правилами:

F_1 : $\widetilde{\text{П}} \longrightarrow \widetilde{\text{С}}\widetilde{\text{Г}}$;
F_2 : $\widetilde{\text{С}} \longrightarrow$ ПС;
F_3 : $\widetilde{\text{Г}} \longrightarrow$ Г$\widetilde{\text{С}}$;
F_4 : П $\longrightarrow$ маленький, шаловливый, красивый;
F_5 : С $\longrightarrow$ мяч, мальчик, девочка;
F_6 : Г $\longrightarrow$ потерял, ударил, бросил.

Правила F_4—F_6 — это в действительности группа правил, поскольку они указывают несколько возможностей для каждого символа П, С, Г.

В рассматриваемой грамматике терминальными символами будут слова, перечисленные в правилах: V_T = {маленький, шаловливый, красивый, мяч, мальчик, девочка, потерял, ударил, бросил}.

Нетерминальными символами будут синтаксические категории

($\tilde{\Gamma}$ — группа глагола, $\tilde{C}$ — группа существительного, Г — глагол, С — существительное, П — прилагательное, $\tilde{\Pi}$ — предложение):

$$V_{\text{н}} = \{\widetilde{\text{П}}, \widetilde{\text{С}}, \widetilde{\text{Г}}, \text{Г}, \text{С}, \text{П}\}.$$

Начальным символом будет понятие предложения. Выводимые терминальные цепочки — правильные предложения данного языка.

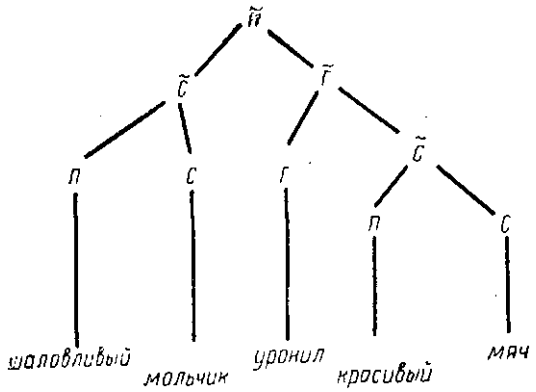


Рис. 12

В этой простой грамматике все терминальные цепочки имеют одну и ту же синтаксическую структуру фраз, что может быть выражено с помощью структурного дерева — маркера структуры составляющих (С-маркера). *Структурное дерево* — это помеченный граф (помеченные ребра и узлы), где узлам соответствуют грамматические типы или синтаксические единицы, а ребра различаются своим порядковым номером. Цепь дерева — последовательность ребер, каждое из которых связано с предшествующим. В лингвистике слова или последовательности, которые функционируют в качестве элементов другой конструкции, называются составляющими. Отсюда название маркера структуры составляющих.

Каждый С-маркер содержит в виде меток при конечных узлах перечень слов, из которых составлено данное предложение (рис. 12).

В рассматриваемом предложении составляющими являются — «мяч», «красивый мяч», «уронил красивый мяч», а «уронил красивый» — не является составляющей. Каждая

составляющая возводится к некоторому узлу дерева. Если этот узел, допустим, помечен С, то говорят, что составляющая принадлежит к типу С.

Те составляющие, из которых конструкция непосредственно образована, являются непосредственными составляющими.

Например, $\tilde{C}$ и $\tilde{\Gamma}$ — непосредственные составляющие предложения. Есть непосредственные составляющие для $\tilde{C}$ и $\tilde{\Gamma}$ — это соответственно П и С, и Г и $\tilde{C}$ и т. д. Очевидно, грамматика не может считаться удовлетворительной, если не дает порождаемым предложениям структурного описания — хотя бы в виде разложения на непосредственные составляющие.

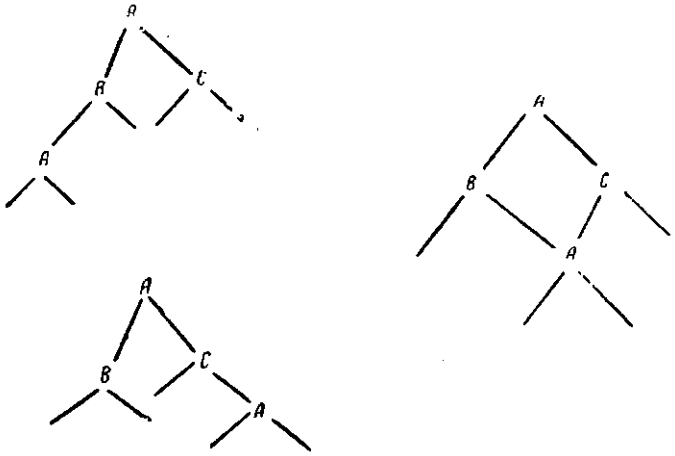
Таким образом, грамматика должна обеспечивать С-маркером каждое из бесконечного числа предложений.

Два С-маркера считаются тождественными, если они имеют одинаковую структуру ветвей и одинаковые метки при соответствующих узлах. Дерево С-маркера характеризуется определенным упорядочением ветвей слева направо в соответствии с порядком элементов в цепочке.

Поскольку число правил конечно, а число маркеров может быть бесконечно, то могут найтись такие символы грамматического словаря, которые повторяются в С-маркерах сколько угодно раз. Более того, могут найтись такие цепи дерева, которые содержат некоторый символ более чем *n* раз для любого фиксированного *n.*

Синтаксический элемент называется рекурсивным элементом, если для некоторого фиксированного *n* найдется некоторое структурное дерево, цепь которого содержит этот символ как наименование узла более чем *n* раз.

Выделим три вида рекурсивных элементов.



Элемент *A* называется леворекурсивным, если подчиненное ему дерево содержит *A* только в самой левой цепи. (Дерево, подчиненное *A*, — это дерево, которое может быть выведено из *A*.)

Элемент *A* называется праворекурсивным, если подчиненное ему дерево содержит *A* только в крайней правой цепи.

Рекурсивный элемент *A* называется самовставленным, если подчиненное ему дерево содержит А в некоторой внутренней цепи. Два структурных дерева тождественны, если имеют одинаковую структуру ветвей, т. е. восходящих линий от морфем языка к синтаксическим типам, и одинаковую разметку ребер и узлов дерева.

Грамматика $G= (V_T, V_H, S, P)$ называется грамматикой с самовставлением, если для некоторого $A \in V_H$ существует вывод $A = >\varphi A \psi$, где φ и ψ — не пустые слова в *V*.

Грамматика *G* называется *однозначной*, если для каждого слова $x \in L_G$ все его выводы имеют одно и то же дерево.

Хорошо известно, что в живых языках могут быть предложения, совпадающие по написанию, но допускающие неоднозначную синтаксическую трактовку. Примером такого рода может служить русская фраза «пальто испачкало окно», в которой остается невыясненным, что является подлежащим и что служит дополнением.

В примерах такого рода существенно не «статистическое» окончательное начертание, а «динамика» процесса грамматического вывода. Подобного рода неоднозначность играет существенную роль для языков программирования. Поэтому возникает вопрос об однозначности различных типов грамматик и возможности найти однозначную грамматику определенного типа.

Грамматика G называется *неоднозначной*, если имеется цепочка $x \in L_G$, которая может быть выведена двумя существенно различными способами. При этом под существенно различными выводами понимается следующее:

Грамматике $G = (V_T, V_H, S, P)$ с правилами вида $\varphi_1 \to \psi_1$, $\varphi_2 \to \psi_2$, ..., $\varphi_k \to \psi_k$ поставим в соответствие грамматику $G' = (V'_T, V'_H, S', P'$, у которой $V'_H = V_H$, $S' = S$, $V'_T = V_T U \{(\varphi_1, (\varphi 2, \ldots, (\varphi_k)\}$, т.е. для каждого правила $\varphi_i — \psi_i$ системы P к терминальному словарю добавляется скобка $(\varphi_i (i = 1, 2, \ldots, k)$.

Система P' получается из системы P заменой каждого правила вида $\varphi_i \to \psi_i$ правилом $\varphi_i \to (_{\varphi i} \psi_i)$.

Очевидно, что каждому выводу цепочки x в Lg однозначно соответствует вывод цепочки x' в L_G'. Эта цепочка x' совпадает с x, если в ней опустить все скобки. Скобки, таким образом, сохраняют «динамику» вывода — их расстановка позволяет восстановить процесс вывода.

Цепочка x' называется структурным описанием цепочки x.

Пример. Пусть $G = (\{0,1\}, S, S, P)$; $P = \{S \to 0, S \to SlS\}$, тогда $G' = (\{0,1\ (S)\}, S, S, P)$; $P = \{S \to (_S 0), S \to (_S S1S)\}$, цепочка 01010 языка L_G может быть выведена различными способами:

1) $S \to S1S \to 01S \to 01S1S \to 0101S \to 01010$;

2) $S \to S1S \to S10 \to S1S10 \to 01S10 \to 01010$.

Им соответствуют различные структурные описания, полученные в L_G':

1) $(s\,(s\,0)\,1\,(s\,(s\,0)\,1\,(s\,0)))$;

2) $(s\,(s\,(s\,0)\,1\,(s\,0))\,1\,(s\,0))$.

Выводы цепочки x в некотором языке L_G называются существенно различными, если структурные описания x', соответствующие этим выводам, отличаются друг от друга.

Если заранее известно, что грамматика не является неоднозначной, то, очевидно, по любой порождаемой ею цепочке можно однозначно восстановить вывод этой цепочки.

Если задана конкретная грамматика G одного из типов и нужно установить, является ли она неоднозначной, то алгоритмическая разрешимость такой задачи устанавливается следующей теоремой:

Теорема (12). Для языков типа *3* существует алгоритм, позволяющий по заданной грамматике определить, является ли она однозначной. Для остальных типов языков эта задача алгоритмически неразрешима.

Язык данного типа называется существенно неоднозначным, если любая грамматика G этого типа, порождающая этот язык, неоднозначна, т. е. если в заданном классе грамматик нельзя найти грамматику без неоднозначностей, эквивалентную заданной. В противном случае язык L называется *языком без неоднозначности* или однозначно выводимым.

Следующие две теоремы относятся к вопросу о существовании существенно неоднозначных языков различных типов.

Теорема 13 (Хомского и Миллера, Шютценберже). Не существует существенно неоднозначных языков типов 0, 2^D*, 3; все эти языки однозначно выводимы.

Теорема 14 (Парика). Существуют существенно неоднозначные бесконтекстные языки.

Для языков типа *1* (контекстных) этот вопрос не решен.

Парик доказал, что бесконтекстные языки $L = \{a^n b^m a^P \mid n = m$ или $n = P\}$ и $L = \{x | x = a^n b^m a^{n'} b^{m'} \cdot Vx = a^n b^m a^n b^{m'}$; $n, n', m, m' \geq 1\}$ являются существенно неоднозначными.

---

* 2^D — подкласс языков типа 2, называемый «детерминированные бесконтекстные языки».

Второй из этих языков порождается грамматикой $G = (\{a, b\}, \{S, A, B, C, D\}, S, P)$, где P включает следующие правила:

$$S \to AB \qquad S \to DC$$
$$A \to aAa \qquad A \to aBa$$
$$B \to b \qquad B \to bB$$
$$C \to bCb \qquad C \to bDb$$
$$D \to a \qquad D \to aD$$

Следующая теорема рассматривает алгоритмическую разрешимость распознавания существенной неоднозначности:

Теорема 15 (Гладкого). Не существует алгоритма для решения задачи о том, является ли заданный язык типа 2^D существенно не однозначным. Однако в некоторых случаях этот вопрос может быть решен.

Следует заметить, что произвольная неукорачивающая грамма тика (без требования заменять сразу только один символ) уже не обладает свойством сопоставлять фразам их структуры непосредственно составляющих. Поскольку в такой грамматике, вообще говоря, каждый раз заменяется не один символ, а целая группа их, то в выводе невозможно однозначно указать для каждого сим вола узла, непосредственно его включающего (его «предка»).

Содержательный лингвистический анализ предложений может включать в себя данные о предложении трех типов: о системе составляющих, об отношении управления, о характеристике составляющих и словоформ, т. е. о принадлежности к тому или иному классу.

В соответствии с этим в общем случае можно рассматривать различные типы маркеров, содержащие ту или иную информацию.

Упражнения:

1. Три грамматики — G_1, G_2, G_3 — заданы соответствующими правилами: $P_1 = \{A \to x, A \to Ax\}$, $P_2 = \{A \to x, A \to xA\}$, $P_3 = \{A \to x, A \to xAx\}$, где A — цель грамматики.

Определить тип эквивалентности грамматик G_1, G_2, G_3.

2. Пусть $G = (V_m, V_н, P, S)$ — порождающая грамматика, где $V_T=\{a, b\}$, $V_H=\{A, S\}$, $P=\{A \to ab, A \to aSAb, S \to bAb, SA \to b, S \to \Lambda\}$. Определить тип рекурсивности правил грамматики и показать, что цепочка *ababbabb* принадлежит множеству $L(G)$.

3. Построить языки $L(G_j)$, где $j = 1, 2, 3$, если схемы правил грамматик G_1, G_2 и G_3 имеют следующий вид:

$$P_1 = \{S \to AB,\ S \to ASB\};$$

$$P_2 = \{S \to AA,\ S \to BB,\ S \to ASA,\ S \to BSB\};$$

$$P_3 = \{S \to AS,\ S \to BS,\ CABSC \to CABABC\}.$$

Для каждой грамматики определить V_m, V_H, длину выводов, тип правил, вид грамматики.

4. Дана грамматика $G = (\{A, B\}, \{S\}, P, S)$, где $P = \{S \to SA, SS \to ABA\}$. Определить, является ли G-грамматика грамматикой непосредственно составляющих.

5. Пусть $V_m = \{A, B\}$. Показать, что $V^*_T = \{A^nB^nA^n | n \geq 1\}$ есть НС-язык.

6. Грамматики G_1, G_2, G_3, G_4 заданы правилами вида:

$$P_1 = \{A \to xB,\ B \to y,\ B \to zB\};$$
$$P_2 = \{A \to By,\ A \to CB,\ B \to Bx,\ B \to x,\ C \to x,\ C \to xC\};$$
$$P_3 = \{A \to Cx,\ B \to Ay,\ C \to Bz,\ C \to zD,\ D \to xE,\ E \to yx\};$$
$$P_4 = \{A \to Cx,\ B \to Ay,\ C \to Bz,\ C \to xD,\ D \to xE,\ E \to yz\}.$$

Определить для каждой из заданных грамматик:

а) терминальный и нетерминальный словарь;

б) однозначность грамматики;

в) вид рекурсии синтаксических элементов;

г) тождественность С-маркеров для терминальных цепочек, выводимых в грамматике.

7. Дана грамматика $G= (V_m, V_H, P, S)$, где $P = \{A \to aB, A \to b, B \to Aa, B \to b$. Построить C-маркер для цепочек *aaba, aabaa, aaabaa.*

8. Пусть $V_T = \{a, b, c\}$. Построить C-маркер, соответствующий выводу $S \to abAbB \to abBaAbB \to abSbaAbB \to abcbaAbB \to abcbacabbB \to abcbacabbaa$.

9. Определить, является ли однозначной грамматика $G = (\{a, b, c, d, e\} \{S, A, B, C\}, P = \{S \to ABB, A \to aC, C \to bc, B \to d\ e\}, S)$.

10. Пусть $V_T = \{a, b, c\}$. Для каких из приводимых ниже множеств P грамматика $G = (V_T, V_H, P, S)$ является неопределенной:

а) $P = \{S \to aSbc,\ S \to c,\ S \to bA,\ A \to bA,\ A \to a\}$;

б) $P = \{S \to a^2Sa,\ S \to aSa,\ S \to aS,\ S \to \Lambda\}$;

в) $P = \{S \to aBc,\ S \to bS,\ B \to aB,\ B \to Ba,\ B \to c\}$;

г) $P = \{S \to aS,\ S \to Sa,\ S \to bBb,\ B \to aBb,\ B \to acb\}$.

11. Грамматики G_1 и G_2 заданы правилами вида:

$$P_1\{A \to AA,\ A \to a^s\};$$

$$P_2\{B \to a^sB,\ B \to a^s\},$$

Определить терминальные и нетерминальные словари грамматик, их однозначность и тип эквивалентности грамматик.

2.3. Контекстно-свободные грамматики

Обратим внимание на то, что в правилах НС-грамматики заменяется только один символ, левая же часть правила не обязательно состоит только из этого символа: $A \to \omega$. В правилах могут присутствовать и другие символы — контекст: $\varphi A \psi \to \varphi \omega \psi$. Такие правила означают разрешение заменять символ A на ω только в контексте φ и ψ. Сам контекст при этой замене переписывается без изменения.

Правила, использующие контекст, назовем контекстно-связанными, а правила, не использующие контекста, — контекстно-свободными.

НС-грамматики, содержащие только контекстно-свободные правила вида $A \to \omega$, называются контекстно-свободными (КС-грамматики), или бесконтекстными грамматиками.

НС-грамматики, содержащие контекстно-связанные правила, называются контекстно-связанными грамматиками.

Языки, порождаемые КС-грамматиками, называются КС-языками.

Заметим, что связанными контекстом, или свободными, являются только правила, а не элементы в терминальной цепочке.

КС-грамматики представляют собой важный частный случай НС-грамматик. Их ценность обусловлена следующими двумя обстоятельствами:

во-первых, отказ от контекста, т.е. требование, чтобы в левой части правила был ровно один символ, делает структуру грамматик еще более простой, что облегчает ее изучение;

во-вторых, хотя в естественных языках замена одних единиц другими часто допустима лишь в определенных контекстах, целесообразно исследовать возможность описывать языки, отвлекаясь от указанного факта. В естественных языках возможны ситуации, когда явления, которые представляются существенно зависимыми от контекста, могут описываться и как независящие от контекста, т.е. в терминах КС-грамматики. При этом, разумеется, описание может усложняться в других отношениях. Например, может потребоваться много новых категорий, правил или того и другого.

В самых общих чертах такая замена делается так: пусть имеется класс элементов X; в соседстве с элементами некоторого класса Y элементы X ведут себя иначе, чем в соседстве с элементами класса Z, так что имеют место правила

$$YX \to YAB;$$
$$ZX \to ZCD$$

(правила используют контекст).

Введем два новых символа — X_1 и X_2. Элемент X в позиции после Y обозначим через X_1, а в позиции после Z — через X_2.

Тогда приходим к правилам, не использующим контекста

$$X_1 \to AB;$$
$$X_2 \to CD.$$

Не следует, однако, думать, что всякая контекстно-связанная НС-грамматика может быть заменена эквивалентной ей КС-грамматикой. Известно, что существуют НС-языки, не являющиеся КС-языками, например, язык, состоящий из всевозможных цепочек вида $a^n b^n a^n$ *(aba, aabbaa,...)* или из цепочек вида $a^n в^n c^n$. От контекста нельзя отказаться, если правило должно обеспечивать перестановку символов, так как перестановка по своему существу является многомерной операцией. Следовательно, КС-грамматика не может порождать язык, содержащий цепочки, которые не могут быть построены без применения перестановок.

Почти все имеющиеся примеры НС-языков, не являющихся КС-языками, носят абстрактный характер и не имеют интерпретаций в естественных языках.

До сих пор мы занимались введением все новых и новых ограничений на классы рассматриваемых грамматик. Сначала мы потребовали, чтобы число символов в правой части правил было не меньше, чем в левой, и получили неукорачивающие грамматики. Затем потребовали, чтобы замене подвергался только один символ, и получили НС-грамматики. Наконец, мы потребовали, чтобы в левой части правила вообще был только один символ, и получили КС-грамматики.

Ясно, что никаких дальнейших ограничений на левые части правил наложить уже нельзя. Поэтому, если мы хотим выделить еще более узкие классы грамматик, придется накладывать ограничения на правые части.

Начнем с числа символов в правой части. В зависимости от числа символов в правой части правил КС-грамматики можно разделить на бинарные и небинарные.

КС-грамматики будем называть бинарными, если правая часть любого правила содержит не более двух символов.

Например, правила вида $A \to BC$, или $A \to b$B, $A \to$B,
где $A, B, C \in V_H$, $b \in V_T$.

КС-грамматики будем называть небинарными, если правая часть любого правила содержит более двух символов. Например, грамматика с правилами вида $A \to Aab$, $B \to ABC$, $A \to \omega$, где $A, B, C \in V_H$, $a, b \in V_T$ ω не пустая цепочка в этой грамматике, содержащая более двух символов.

Бинарные КС-грамматики обладают той особенностью, что в соответствующих им деревьях структуры составляющих — *С*-маркерах — из каждой вершины исходит не более двух ветвей. Это значит, что любая сложная составляющая всегда состоит ровно из двух непосредственно вложенных в нее составляющих.

В свою очередь каждый из выделенных классов КС-грамматик в соответствии с дальнейшими ограничениями, налагаемыми на правые части правил, подразделяются на подклассы. Так, в зависимости от числа вхождений в правую часть правил нетерминальных символов КС-грамматики подразделяются на линейные и нелинейные:

Линейные грамматики — такие КС-грамматики, правые части правил которых содержат не более чем по одному вхождению не терминального символа. Таким образом, для бинарных КС-грамматик это правила вида

$$A \to aB,$$

где B, $A \in V_H$, $a \in V_т$,
для небинарных КС-грамматик правила вида

$$A \to aBab, \quad A \to acB;$$

$$A, B \in V_н, a, b, c \in V_т.$$

Язык, порождаемый линейными грамматиками, называется линейным языком. КС-грамматики, правые части правил которых содержат более одного нетерминального символа, будем называть *нелинейными* КС-*грамматиками.*

КС-грамматика называется металинейной, если правые части ее правил не содержат цели грамматики и все правила, левые части которых отличны от цели, имеют такой же вид, как правило линейной грамматики.

Примером металинейной грамматики может служить следующая грамматика $G = (\{a, b, c\}, \{S, T\}, \{S \to TT, T \to aTa, T \to bTB, T \to c\}, S)$.

Язык называется металинейным, если существует порождающая его металинейная грамматика.

Накладывая ограничения на состав символов правой части правил КС-грамматик, Флойд выделил следующие подклассы небинарных, нелинейных грамматик.

Операционные грамматики — правые части правил которых не могут содержать двух рядом стоящих нетерминальных символов. Примерами правил такого вида могут служить

$$A \to BbC;$$

$$A \to BabC,$$

где $A, B, C \in V_H$, $a, b \in V_T$.

Грамматики предшествий — правые части правил которых могут содержать два рядом стоящих терминальных символа. При этом имеется возможность указать, какой из этих терминальных символов возникает в словообразовании первым, имеющим больший приоритет.

Примерами правил такой грамматики могут служить следующие:

$$A \to BaaB,$$

где $A, B \in V_H$, $a \in V_T$. Имеются и другие подклассы.

Подклассом линейных КС-грамматик являются односторонние линейные грамматики.

Односторонние линейные грамматики — такие грамматики, правые части правил которых содержат терминальные символы, только с одной стороны от нетерминального символа.

Односторонние линейные грамматики подразделяются на левосторонние — с правилами вида $A \to xB$ и $A \to x$ и правосторонние — с правилами вида $A \to Bx$ и $A \to x$. В обоих случаях A, B - нетерминальные символы, а x — непустая цепочка терминальных символов.

Односторонние линейные грамматики, у которых в каждом правиле цепочка *x* состоит только из одного символа, называются *автоматными грамматиками*, или А-*грамматиками*, а языки, порождаемые этими грамматиками, называются *автоматными языками*, или *А-языками*.

Из данных определений ясно, что каждый следующий класс грамматик содержится в предыдущем.

Взаимосвязь рассмотренных классов грамматик можно представить в виде графа, изображенного на рис. *13.* Таким образом, КС-грамматики представляют собой наиболее важный подкласс НС-грамматик. Это объясняется следующими четырьмя основными причинами:

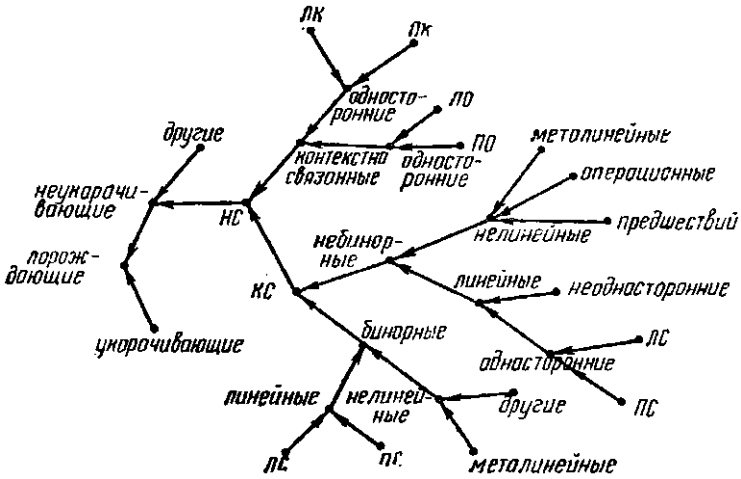


Рис. 13

1. КС-грамматики являются основой определения почти всех употребительных языков программирования.

2. Все действия системы синтаксического анализа для естественных языков основаны на КС-грамматиках.

3. Это единственный тип грамматики, теория которой изучена и практически проверена.

4. Все трансформационные грамматики построены на КС-грамматиках.

Все рассмотренные типы грамматик порождают четыре типа языков: НС-язык, КС-язык; линейный язык, А-язык, не считая языка, порождаемого самым общим типом грамматики с неограниченными правилами вывода грамматикой типа 0.

Взаимосвязь между языками, порождаемыми рассмотренными типами грамматик, будет следующей:

$$L(\text{О}) \supset L(\text{НС}) \supset L(\text{КС}) \supset L(\text{Л}) \supset L(\text{А}).$$

Упражнения:

1. Пусть *VT = {a, b},* построить грамматики, порождающие следующие языки:

а) язык $L = \{a^n b^n a^n | n \geqslant 1\}$;

б) язык $L = \{a^{n^2} | n \geqslant 1\}$;

в) язык $L = \{a^n b^{n^2} | n \geqslant 1\}$.

2. Грамматики *G1* и *G2* заданы правилами:

$$P_1 = \{A \to aAbB,\ A \to bB,\ A \to A,\ A \to b,$$
$$B \to bAbb,\ B \to B,\ B \to bAb,\ B \to ab\};$$
$$P_2 = P_1 - \{A \to A,\ B \to B\}.$$

Показать, что *L(G1) = L(G2).*

3. Дана Лграмматика *G{VT, VH, P, S),* где *Vm = {a, b, c}; VH = {S, B, C}, P={S→aSBC, S→aBC, CB→BC, aB→ab, bB→bb, bC→bc, cB→cc).* Определить тип грамматики и язык, порождаемый ею.

4. Доказать, что каждый линейный язык порождается грамматикой, в которой каждое правило является либо леволинейным, либо праволинейным.

5. Заданы две грамматики — *G1* и *G2* с правилами вида:

$$P_1 = \{S \to abS,\ S \to cA,\ A \to baA,\ A \to a\};$$
$$P_2 = \{S \to AB,\ A \to aAb,\ B \to aBb,\ A \to c,\ B \to c\}.$$

Определить тип грамматик.

6. Доказать, что язык *L = {ambmanbn}* не является линейным.

7. Построить пример КС-языка, не являющегося А-языком и порождаемого КС-грамматикой, каждое правило которой является либо леволинейным, либо праволинейным.

8. Пусть *G = (VT, VH, P, S); VT = {a, b, c}; P = {S→AB, A→aAb, B→aBb, A→c, B→c}*. Показать, что КС-язык является металинейным, но не линейным.

9. Грамматика *G* задана правилами *P = {S→aSa, S→bSb, S→aaSaa, S→c}*. Показать, что она не является существенно неоднозначной.

10. Язык $L = \{a^n b^m c^r | n + m \geq r\}$ является КС-языком. Выписать КС-грамматику, порождающую этот язык. Определить, к какому более узкому классу языков он принадлежит.

2.4. Классы, промежуточные между бесконтекстными и контекстными грамматиками

Неоднозначность грамматик является важным понятием теории языков. Существенно, например, что грамматика, служащая для порождения языка программирования, неоднозначна. Это означает, что программа в этом языке может иметь более чем одну возможную интерпретацию.

Давно существует мнение, что для полного описания естественных или программных языков необходима более тонкая структура, чем та, которую дают бесконтекстные грамматики. Однако контекстные языки слишком сложны, чтобы их использовать при описании. Необходим компромисс. Недавно появились два обобщения бесконтекстных грамматик: программные и индексные грамматики. Рассмотрим каждую из них.

2.4.1. Программные грамматики

Программной грамматикой является грамматика вида $G = (N, T, I, P, S)$, где N, T, P — конечные множества нетерминальных символов, терминальных символов и правил грамматики соответственно; S — начальный символ из N; I — множество меток правил. Правила грамматики имеют вид:

$$(r)A \to \alpha S(V)F(W),$$

где $A \to \alpha$ называется ядром, оно имеет бесконтекстную форму, $A \in N$, $\alpha \in (NUT)^*$, (r) является меткой и $r \in I$. Никакие два правила из P не могут иметь одну и ту же ячейку. Далее V называется областью удачи, W — областью неудачи, V и W — подмножества множества I. Таким образом, если пытаются применить правило r к строке ω и если имеется вхождение A в ω, то самое левое A заменяется на α, а следующее используемое правило должно иметь метку из V. Если ω не содержит A, то со не меняется и следующее используемое правило должно иметь метку из W.

Формально отношение => определено для пар вида (α, r), где $\alpha \in (N \cup T)^*$ и $r \in I$. Предположим, что правило с меткой r есть $A \to \omega\ S(V)F(W)$. Если $\alpha = \alpha_1 A \alpha_2$, $\alpha \in (N \cup T - \{A\})^*$ и s произвольная метка из V,то мы можем написать $(\alpha, r) => (\alpha, \omega\alpha, s)$. Если же $a \in (N \cup T - \{A\})^*$ и s — метка из W, то мы можем написать $(\alpha, r) => (\alpha, s)$. Языком $L(G)$, порожденным грамматикой G, будет $\{y | y \in \overset{G}{T}{}^*\}$ и для некоторого r из I имеет место $(s, 1) => (y, r)$.

Пример. Пусть $G = (\{S, A, B, C\}, \{a, b, c\}, \{1, 2, ..., 8\}, P, S)$, где P состоит из следующих правил:

Метка	Ядро	Удача	Неудача	Метка	Ядро	Удача	Неудача
(1)	$S \to ABC$	$S(\{2,5\})$	$F(\Phi)$	(5)	$A \to a$	$S(\{6\})$	$F(\Phi)$
(2)	$A \to aA$	$S(\{3\})$	$F(\Phi)$	(6)	$B \to b$	$S(\{7\})$	$F(\Phi)$
(3)	$B \to bB$	$S(\{4\})$	$F(\Phi)$	(7)	$C \to c$	$S(\{8\})$	$F(\Phi)$
(4)	$c \to cC$	$S(\{2,5\})$	$F(\Phi)$	(8)	$S \to S$	$S(\{\Phi\})$	$F(\Phi)$

Тогда имеем $L(G) = \{a^n b^n c^n | n \geq 1\}$. Заметим, что язык $L(G)$ не является бесконтекстным. С помощью правила (1) мы имеем $(S,1) => (ABC, 2)$ и $(S, 1) => (ABC, 5)$. В последнем случае можно использовать только правила с (5) по (7) и тогда закончить вывод.

В этом случае мы имеем $(ABC, 5) => (abC, 6) => (abC, 7) => (abc, 8)$. К паре $(ABC, 2)$ можно применять цикл правил (3), (4), (2) до тех пор, пока после (4) будет выбрано правило (5). Таким образом, имеем

$$(S,\ 1) \Rightarrow (a^{n-1}Ab^{n-1}Bc^{n-1}C,\ 5) \text{ для всех } n \geqslant 1.$$

Вывод заканчивается применением правил (5) — (7). В результате получим

$$(a^{n-1}Ab^{n-1}Bc^{n-1}C,\ 5) \Rightarrow (a^n,\ b^n,\ c^n,\ 8).$$

Теорема (16). Любой язык типа 0 порождается программной грамматикой.

Бесконтекстной программной грамматикой называется программная грамматика $G = (N, T, I, P, S)$, в которой никакое правило из P не имеет ядра вида $A \to \Lambda$.

Теорема (17). Если $L = L(G)$ для бесконтекстной программной грамматики G, то L является контекстным языком.

2.4.2. Индексные грамматики

Индексной грамматикой называется грамматика $G = (N, T, F, P, S)$, где N, T, F и P — конечные множества нетерминальных символов, терминальных символов указателей и правил соответственно; S из N— начальный символ. Указатели из F суть конечные множества индексных правил вида $A \to \alpha$, где $A \in N$ и $\alpha \in (NUT)^*$. Правила из P имеют вид:

$$A \to X_1\psi_1 X_2\psi_2 \ldots X_m\psi_m,$$

где $X1, X2, \ldots X_m$ принадлежат N или T, а $\psi_1, \psi_2, \ldots, \psi_m$ принадлежат F^*. Если $X_i \in T$, то $\psi_i = \Lambda$.

Таким образом, за нетерминалами в строке могут следовать произвольные списки указателей. Если нетерминал, за которым следуют указатели, заменяется одним или более нетерминалами, то возникают указатели, следующие за каждым нетерминалом. Если нетерминал замещается терминалом, то указатели исчезают. Если за нетерминалом A следуют указатели $f_1, f_2, \ldots, f_n$ и f_1 содержит индексное правило, вида $A \to \alpha$, то можно убрать $f1$, заменив A на α и написав вслед за каждым нетерминалом и α список указателей $f2, f3, \ldots, fn$.

Формально пусть $G = (N, T, F, P, S)$ — индексная грамматика. Определим отношение => на строчках из $(NF^* \cup T)^*$ следующим образом.

1. Если αAΘβ — строчка, такая, что α и β принадлежат

$$(NF^* \cup T)^*, \quad A \in N, \quad \Theta \in F^* \text{ и}$$
$$A \to X_1\psi_1 X_2\psi_2 \ldots X_m\psi_m \in P,$$

где $X_i \in N \cup T$ и $\psi_i \in F^*$ для всех i, то мы можем написать

$$\alpha A\Theta\beta \underset{G}{\Rightarrow} \alpha X_1\varphi_1 X_2\varphi_2 \ldots X_m\varphi_m\beta.$$

Здесь $\varphi_i = \Lambda$, если $X_i \in T$, и $\varphi_i = \psi_i\Theta$, если $X_i \in N$.

2. Если αAΘβ — строчка, такая, что α и β принадлежат $(NF^* \cup T)^*$, $A \in N$, $f \in F$, $\Theta \in F^*$, а $A \to X_1X_2 \ldots X_m$ является индексным правилом из f, то мы можем написать

$$\alpha Af\Theta\beta \underset{G}{\Rightarrow} \alpha X_1\varphi_1 X_2\varphi_2 \ldots X_m\varphi_m\beta,$$

где $\varphi_i = \Lambda$, если $X_i \in T$, и $\varphi_i = \Theta$, если $X_i \in N$.

Пусть $\overset{*}{\underset{G}{\Rightarrow}}$ есть рефлексивное, транзитивное замыкание $\underset{G}{\Rightarrow}$.

Язык, порождаемый грамматикой G, обозначается $L(G)$ и есть $\{y \mid y \in T \text{ и } S \overset{*}{\underset{G}{\Rightarrow}} y\}$.

Пример. Пусть $G = (\{S, T, A, B, C\}, \{a, b, c\}, \{f, q\}, P, S)$, где P состоит из правил

$$S \to Tq, \quad T \to Tf, \quad T \to ABC;$$
$$f = \{A \to aA, \quad B \to bB, \quad C \to cC\};$$
$$q = \{A \to a, \quad B \to b, \quad C \to c\}.$$

Имеем $L(G) = \{a^n b^n c^n \mid n \geq 1\}$. Применяя первое правило из P один раз, затем второе правило n—1 раз и, наконец, третье правило один раз, находим

$$S \underset{G}{\Rightarrow} Tq \underset{G}{\Rightarrow} Tfq \underset{G}{\Rightarrow} \ldots \underset{G}{\Rightarrow} Tf^{n-1}q \underset{G}{\Rightarrow} Af^{n-1}qBf^{n-1}qCf^{n-1}q.$$

Затем, используя указатели, получаем

$$Af^{n-1}q \underset{G}{\Rightarrow} aAf^{n-2}q \underset{G}{\Rightarrow} \ldots \underset{G}{\Rightarrow} a^{n-1}Aq \underset{G}{\Rightarrow} a^n.$$

Аналогично

$$Bf^{n-1}q \overset{*}{\underset{G}{\Rightarrow}} b^n \text{ и } Cf^{n-1}q \overset{*}{\underset{G}{\Rightarrow}} c^n.$$

Таким образом,

$$S \overset{*}{\underset{G}{\Rightarrow}} a^n b^n c^n.$$

Теорема (18). Любой индексный язык L порождается индексной грамматикой $G = (N, T, F, P, S)$, обладающей следующими свойствами:

1.Индексные правила из F имеют вид $A \to B$, где A и B принадлежат N и $B \neq S$.

2.Если $A \neq S$, то $A \to \Lambda$ не принадлежит P.

3.Правила из P (отличные от $S \to \Lambda$, если такое правило содержится в P) имеют вид $A \to BC$, $A \to Bf$ или $A \to a$ для A, B, $C \in N$, $f \in F$, $a \in T$.

Теорема (19). Если L — индексный язык, то L — контекстный язык.

2.5.Трансформационные грамматики

До сих пор рассматривались различные классы и подклассы порождающих грамматик. Но как было указано ранее, кроме них в математической лингвистике выделяются распознающие и преобразующие грамматики. Эти грамматики менее изучены, чем порождающие, и правила их недостаточно формализованы.

Очевидно, любая порождающая грамматика содержит только правила образования языковых выражений.

Однако в логике рассматривается еще и другой вид правил — правила преобразования, которые определяют соотношения между правильными фразами (по существу это смысловые отношения). Так, в элементарной алгебре — это правило тождественных преобразований, в алгебре логики — правило вывода, позволяющие из одних истинных выражений получить другие истинные выражения. Введение правил преобразования всегда означает переход к более высокому уровню рассмотрения языка, а именно — к семантическому уровню.

Совершенно очевидно, что правила преобразования необходимы и при описании естественных языков. Владение языком обязательно предполагает умение не только построить правильную фразу, но и перейти от одной фразы к другим, либо полностью синонимичным ей, либо отличающимся от нее по смыслу на определенную «величину». Эти возможности, которые обязательно должны учитываться в описании языка, не могут быть изложены в терминах грамматик, и поэтому встает вопрос о разработке формального аппарата для правил преобразования применительно к естественным языкам.

Соответствующая задача была впервые четко сформулирована в работах Н. Хомского. Выдвинутая им концепция быстро приобрела широкую популярность под названием «трансформационная грамматика».

В общем случае правила грамматики всегда могут быть сведены.к виду

$$\varphi_i, \ldots, \varphi_n \to \psi, \qquad (13)$$

где φ_i, ..., φ_n — любые синтаксические элементы или структуры произвольного вида. А символ $\to$ означает, что если φ_i, ...,φ_n были порождены в процессе вывода, то на определенном шаге может быть порождено ψ.

Правила грамматик непосредственно составляющих представляют собой специальную разновидность правил (13), а именно случай, когда $n = 1$ и каждая из компонент φ и ψ являются просто цепочкой терминальных и нетерминальных символов. Кроме того, правила НС-грамматики должны иметь вид

$$A \to \omega;$$

$$\varphi A \psi \to \varphi \omega \psi.$$

Порождающие грамматики, частным случаем которых являются НС-грамматики, обладают некоторыми недостатками. К числу таких недостатков относятся следующие:

1. НС-грамматика, как и любая порождающая грамматика, содержит только правила образования языковых выражений и не содержит правил преобразования, которые задают определенные соотношения между правильными фразами (образование из утвердительного предложения, вопросительного или отрицательного, или из активной конструкции, пассивной).

2. НС-грамматика строит предложения с точно определенным порядком слов. В синтаксической структуре, определяемой НС-грамматиками, не расчленены два совершенно различных по своей природе, хотя и связанных между собой отношения: синтаксическое подчинение и линейное взаиморасположение.

Порождаемому предложению сопоставляется синтаксическая структура в форме упорядоченного дерева, т. е. дерева, где между узлами, кроме отношения подчинения, задаваемого самим деревом, имеется еще и отношение линейного порядка (правее — левее).

Таким образом, порождающие грамматики имеют дело только с синтаксисом языка, т.е. с законами построения правильных словоформ и предложений.

Однако у языка есть и другой, не менее важный аспект — семантика. Для описания семантики необходимо ввести в рассмотрение правило перехода от одних выражений языка к некоторым другим выражениям, несущим ту же информацию, т.е. правила преобразования языковых выражений — правила грамматической трансформации.

Грамматика, использующая правила трансформации языковых выражений, называется трансформационной и представляет собой класс грамматик преобразования.

Рассмотрим класс грамматик, состоящих из двух компонент:

1. Из НС-компоненты, представляющей собой НС-грамматику справилами вида $\varphi A\psi \rightarrow \varphi\omega\psi$, где А—единичный символ, ω — непустая цепочка;

2. Из Т — компоненты, представленной правилами вида $\varphi 1, ..., \varphi_n \rightarrow \psi$, где $n > 1$, φ_1, φ_2, φ_n, ψ представляют собой не цепочки символов, а структурные деревья — *С*-маркеры цепочек.

Такие грамматики будем называть трансформационными или *Т*-грамматиками.

Множество правил трансформационной грамматики удовлетворяет следующим условиям. Прежде всего имеется грамматика НС, порождающая конечное число *С*-терминальных цепочек, с каждой из которых мы можем ассоциировать размеченное дерево, *С*-маркер, представляющее ее структуру составляющих. Затем мы добавляем к грамматике множество операций, называемых грамматическими трансформациями, каждая из которых отображает набор *n* С-маркеров ($n \geq 1$) в новый *С*-маркер.

Среди трансформаций различают обязательные, которые должны применяться к любому выводу, и факультативные, применяемые в соответствии с конкретными условиями.

Цепочка, полученная в результате применения всех обязательных и в случае необходимости некоторых факультативных трансформаций, называется *Т*-терминальной цепочкой.

Т-терминальная цепочка, полученная в результате применения только обязательных трансформаций, называется ядерной цепочкой. Если язык содержит только ядерные цепочки, то эти цепочки будут соответствовать только наиболее простым предложениям языка.

Трансформации, применимые к одному С-маркеру, будем называть сингулярными трансформациями. Трансформации, применимые к паре (или более) *С*-маркеров, будем называть обобщенными трансформациями.

Сингулярные трансформации часто сводятся к простым перестановкам элементов исходной цепочки (*С*-маркера), к добавлению константных цепочек в заданное место или к устранению некоторых элементов исходной цепочки. Сингулярные трансформации являются элементарными трансформациями.

Обобщенные трансформации основываются на элементарных трансформациях, которые подставляют трансформированный вариант второй (*n*-й) компоненты пары исходных терминальных цепочек вместо некоторого элемента первой (n — i) компоненты пары.

Так, например, если трансформация заменяет символ α некоторого структурного дерева σ_1 на структурное дерево σ_2, то *С*-маркер результирующей цепочки есть *С*-маркер σ_1, где α заменен *С*-маркером σ_2.

Другим видом обобщенной трансформации может быть трансформация присоединения к одному C_1-маркеру другого C_2-маркера.

Таким образом, *Т*-грамматика основана на правилах НС-грамматики, к которым добавлены конечное число трансформаций определенного типа, а также ограничений относительно порядка применения трансформаций. Структура, полученная в результате применения некоторой трансформации, в общем случае может быть подвергнута новым трансформациям, так что путем повторного применения трансформаций можно получить бесконечное множество *С*-маркеров самых различных типов.

Структурное описание любой *Т*-терминальной цепочки должно состоять не только из собственного *С*-маркера, ассоциированного с ней, но и из *С*-маркеров, лежащих в основе трансформационной истории данной цепочки.

Т-грамматика — это упорядоченная семерка:

$$\mathrm{T} = (V^T V_T^T C^T S^T P T \tau),$$

где V^T — множество всех синтаксических и терминальных символов, к которым возможно применять трансформацию;

V^T_T — множество символов, образующих *Т*-терминальные цепочки;

C^T — множество *С*-маркеров, представленных в соответствие всем *Т*-терминальным цепочкам из V^T_T;

S^T — нетерминальный символ, подчиняющий общий *С*-маркер вывода *Т*-грамматики;

P — совокупность правил вывода или НС-компонента *T*-грамматики, порождающая *T*-терминальную цепочку, к которой применима грамматическая трансформацияΛ;

T — совокупность правил грамматической трансформации, применимая к элементам из C^T множества;

τ — совокупность заданных соотношений на множестве $V^T \cup C^T$, определяющая связь между структурным описанием (*C*-маркером) и *T*-терминальной цепочкой из $V^T{}_T$.

Теперь можно определить понятие грамматической трансформации следующим образом.

Пусть C_t есть *C*-маркер некоторой *T*-терминальной цепочки *t*, где $t \in V^T$, $C_t \in C^T$. И пусть *t* имеет своими непосредственными составляющими терминалы $t_1, ..., t_n$, каждый из которых связан с некоторым узлом *C*-маркера C_t помеченным символом φ_i. В таком случае говорят, что цепочка *t* может быть проанализирована относительно C_t как $(t_1, ..., t_n, \varphi_1, ..., \varphi_n)$.

Цепочка *t* с *C*-маркером C_t входит в область применения трансформации *T*, если *t* может быть проанализирована относительно C_t как $(t_1, ..., t_n, \varphi_1, ..., \varphi_n)$.

В этом случае $(t_1, ..., t_n)$ есть собственный анализ относительно $<C_tT>$, а $(\varphi_1, ..., \varphi_n)$ — *структурный индекс*.

Отсюда *T*-трансформация применима к элементам собственного анализа. Действие трансформации при этом заключается в перестановке или выбрасывании некоторых элементов собственного анализа *t*, в замене их на некоторые другие, в постановке на заданное место некоторых постоянных цепочек.

Структурный индекс $(\varphi_1, ... \varphi_n)$ определяет возможность *t* — цепочки, т. е. разложения *T*-терминальных цепочек на элементы собственного анализа, которые конкретно задают область применения трансформации.

Значение *T*-грамматики состоит в том, что она позволяет решить проблему анализа структур различной сложности и выступает как объяснительная теория грамматик вообще.

Как мы уже знаем, порождающие грамматики рассматриваются в рамках строго формализованной теории. Что же касается *T*-грамматик, то здесь подобный уровень формализации пока не достигнут. Предлагавшиеся до сих пор трансформационные правила не сформулированы в терминах одной простой операции как правила порождающих грамматик с их операцией подстановки. Задача формализации трансформаций является весьма актуальной.

2.6. Категориальная грамматика

Рассмотрим один из классов, распознающих грамматик-категориальные грамматики (*К*-грамматики).

Понятие *К*-грамматики было введено Бар-Хиллелом в 1960г. с целью построения механической процедуры синтаксического анализа предложений. Для этого в грамматиках должна быть разработана такая система кодирования синтаксических классов (словоформ и словосочетаний), чтобы комбинирование двух классов, а также выделение класса результирующего сочетания можно было определить непосредственно из исходных классов.

Сущность *К*-грамматик состоит в том, что для реализации синтаксического кода все синтаксические классы делятся на два типа: элементарные синтаксические классы и операторы.

Например, имеются элементарные синтаксические классы $S_{ИМ}$ (существительное в именит, падеже) и ПРЕДЛ — (предложение). Тогда синтаксический класс, присоединяемый к $S_{ИМ}$ справа и дающий предложение, есть оператор, действующий на $S_{ИМ}$ справа и переводящий $S_{ИМ}$ в ПРЕДЛ. Такой оператор обозначается [$S_{ИМ}$ \ПРЕДЛ], или {$S_{ИМ}$ \ ПРЕДЛ}.

Теперь перейдем к определению *К*-грамматики. *К*-грамматикой называется упорядоченная четверка

$$K = <T, H, I, R>,$$

где *Т* — конечное множество символов, называемое терминальным или основным словарем (словарь словоформ);

Н — конечный набор элементарных категорий (ЭК), из которых строятся категории в *Н* следующим образом:

1) всякая ЭК есть категория в *Н*;

2) если φ и ψ — категории в *Н*, то слова {φ\ψ} и {φ/ψ} — категории в *Н*.

Запись {φ\ψ} читается как «φ под ψ», а запись {φ/ψ} -как «φ над ψ»;

I — «отмеченная», или «главная», категория. Это конечный символ ∈H, роль которого в некотором смысле противоположна роли начального символа *S* в порождающей грамматике. Из *S* развертывается порождающая цепочка, а к *I* свертывается распознаваемая цепочка;

R — конечное множество синтаксических правил грамматики. Каждому основному символу из *Т* множество *R* ставит в соответствие конечное число категорий. Содержательно эти правила можно представлять себе как задание в словаре при словах их синтаксических классов (нескольких в случае омонимии: течь — существительное или глагол).

Таким образом, элементы словарей *T, H, I* представляют соответственно словоформы, элементарные категории, отмеченные категории, а цепочки *R* — синтаксические правила грамматики *К*.

Укажем, как работает такая грамматика, введя еще одно понятие: сокращение цепочек категории.

Под непосредственным сокращением понимается одна из двух операций:

либо некоторое вхождение цепочки {φ/ψ}ψ заменяется на φ (правое сокращение),

либо некоторое вхождение цепочки φ{φ\ψ} заменяется на ψ (левое сокращение).

Категория {φ/ψ} приписывается такой цепочке, которая, находясь слева от цепочки с категорией ψ, образует с нею цепочку, имеющую в целом категорию φ:

$$\overbrace{\{\varphi/\psi\}\ \psi}^{\varphi}.$$

Иначе говоря, {φ/ψ} есть оператор, который, действуя на ψ слева, превращает его в φ.

Совершенно аналогично интерпретируется категория {φ\ψ}, но только речь идет здесь о присоединении не к ψ, а к φ, и не слева, а справа.

Цепочка категорий α сокращается до цепочки категорий β, если β получается из α последовательностью непосредственных сокращений. Такой процесс называется сокращением.

Например, цепочка

$$\alpha = \{\{\{x \backslash y\}/x\}/z\}\ z\ \{y/y\}\ \{\{y/y\} \backslash x\}\ \{z/x\}\ x$$

сокращается до цепочки $\beta = \{x \backslash y\}z$ путем четырех непосредственных сокращений:

$$\alpha = \underbrace{\{\{\{x \backslash y\}/x\}/z\}}_{\gamma}\ z\ \{y/y\}\ \{\{y/y\} \backslash x\}\ \{z/x\}\ x.$$

$$1.\ \{\gamma/z\}\ z \to \gamma;$$
$$\{\underbrace{\{x\backslash y\}/x}_{\gamma}\}\ \underbrace{\{y/y\}}_{\delta}\ \{\underbrace{\{y/y\}}_{\delta}\backslash x\}\ \{z/x\}\ x.$$
$$2.\ \delta\ \{\delta\backslash x\} \to x;$$
$$\{\{\underbrace{x\backslash y}_{E}\}/x\}\ x\ \{z/x\}\ x.$$
$$3.\ \{E/x\}\ x \to E \quad \{\underbrace{x\backslash y}_{E}\}\ \{z/x\}\ x.$$
$$4.\ \{z/x\}\ x \to z \quad \beta = \{x\backslash y\}\ z.$$

Одну и ту же цепочку категорий можно, вообще говоря, сокращать разными способами, применяя непосредственные сокращения в разном порядке.

Рассмотрим принцип работы *К*-грамматик. Пусть имеется цепочка $x = a_1a_2a_3 \ldots a_k$ из символов основного словаря *Т* некоторой *К*-грамматики G_1. Система правил *R* этой грамматики позволяет сопоставить цепочке *x* цепочку категорий $\xi = R(a_1)R(a_2)R(a_3) \ldots R(a_k)$. Если эта цепочка сокращается до одной категории φ, то мы будем говорить, что грамматика *G1* приписывает цепочке *x* категорию φ.

Для произвольной цепочки (из символов словаря *Т*) *К*-грамматика позволяет узнать, какими категориями характеризуется эта цепочка и характеризуется ли она вообще какими бы то ни было категориями. В частности, для каждой цепочки *К*-грамматика позволяет узнать, является ли эта цепочка правильным предложением, и установить грамматическую правильность. Более того, если цепочка оказывается грамматически правильным предложением, то *К*-грамматика выделяет в нем словосочетания, т.е. составляющие.

В случае синтаксически неоднозначного предложения *К*-грамматика может сопоставить ему разные системы составляющих, т.е. давать разные анализы.

С каждой К-грамматикой естественно связывается множество тех цепочек, которые эта грамматика признает предложениями, т.е. приписывает им категорию ПРЕДЛ. Это множество называется языком, определяемым К-грамматикой, или К-языком.

Возникает вопрос: каково соотношение между языками, определяемыми К-грамматиками, и языками, порождаемыми порождающими грамматиками? Легко видеть, что всякий *К*-язык есть КС-язык. Более того, для всякой *К*-грамматики можно построить эквивалентную ей КС-грамматику (т.е. порождающую в точности тот же язык, который определяется исходной *К*-грамматикой).

Идея доказательства этого факта состоит в следующем: если $G = <T, H, ПРЕДЛ, R>$ есть *К*-грамматика, то соответствующая КС-грамматика Г строится так:

1. Основным словарем в Г будет *Т*.
2. Вспомогательным словарем в Г будет множество всех категорий, являющихся значением *R*.
3. Начальным символом служит ПРЕДЛ.
4. Правила грамматики Г будут двух типов:

а) правило вида $\psi \to \varphi\{\varphi\backslash\psi\}$ и $\varphi \to \{\varphi/\psi\}\psi$, где $\{\varphi\backslash\psi\}$ и $\{\varphi/\psi\}$ — произвольные составные категории из вспомогательного словаря грамматики Г;

б) правило вида $R(a) \to a$, где *a* — терминальный символ, а $R(a)$ — значение соответствующего правила.

Оказывается, что верно и обратное: всякой КС-язык есть *К*-язык, причем для всякой КС-грамматики можно построить эквивалентную ей *К*-грамматику.

Из сказанного непосредственно следует, что класс *К*-языков в точности совпадает с классом КС-языков. К тому же *К*-грамматики имеют три безусловных достоинства:

1. *К*-грамматики фактически не имеют правил (за исключением общих для всех грамматик правил сокращения, которых всего два). Все необходимые сведения о синтаксических свойствах слов должны содержаться в «словарных статьях», т. е. приписываться *R*.
2. *К*-грамматика предполагает весьма тонкий анализ синтаксических свойств отдельных словоформ, т. е. детальное различение синтаксических функций.
3. *К*-грамматики позволяют получать для фразы не только систему (дерево) составляющих, но и систему (дерево) зависимостей. Это свойство делает *К*-грамматики интересными с точки зрения автоматического анализа и перевода текстов.

Что же касается практического удобства, то в двух очень существенных отношениях *К*-грамматики явно неудобны.

1. Применение их к естественным языкам, в особенности к языкам с развитой морфологией (например, русскому), требует введения огромного количества составных категорий и притом очень громоздких.

2. В *К*-грамматиках синтаксические категории слишком тесносвязаны с порядком слов: левое и правое подлежащее, левое и правое дополнение — совершенно различные категории.

В то же время, учитывая указанные недостатки, *К*-грамматики в чистом виде, по-видимому, мало пригодны для подобных практических целей.

Теперь дерево общей классификации грамматик имеет вид, представленный на рис. 14.

Заканчивая разбор основных типов грамматик, можно высказать следующее соображение относительно их классификации. В литературе по математической лингвистике грамматики всегда подразделяют на порождающие и распознавающие. Этой принятой классификации придерживались и мы, хотя она в значительной мере условна.

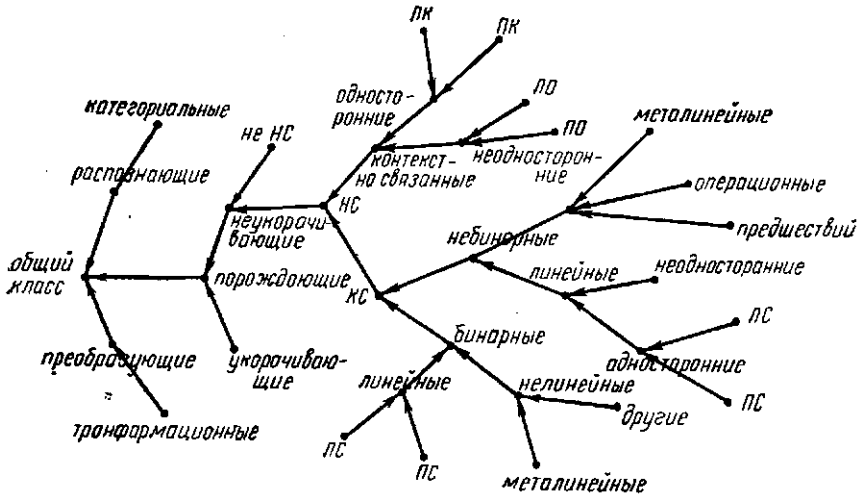


Рис 14

В самом деле, легко видеть, что порождающие грамматики наиболее важного типа, а именно неукорачивающие (и, в частности, НС-грамматики), могут быть использованы также и для распознавания, т.е. для отличия грамматически правильных (выводимых) предложений от неправильных (невыводимых). Для произвольных грамматик такая ситуация не имеет места: существуют грамматики, для которых алгоритм распознавания выводимости цепочек невозможен.

Однако суть дела в том, что порождению естественно противопоставляется не распознавание, а, так сказать, «допускание». Именно естественно говорить, что некоторая грамматика *G* допускает язык *L*, если *G* дает процедуру, способную для любой цепочки $x \in L$ рано или поздно установить это. Если же $x \notin L$, то от этой процедуры ничего не требуется, от процедуры же распознавания требуется больше: она должна давать результат в любом случае — положительный, если $x \in L$, и отрицательный, если $x \notin L$.

Если вместо распознавания рассматривать «допускание», то тогда все порождающие грамматики могут трактоваться и как допускающие. При этом допускающая процедура состоит в том, что правила грамматики применяются к данной цепочке наоборот — справа налево. В цепочке отыскивается вхождение правой части некоторого правила и заменяется левой частью, и этот процесс продолжается, пока можно. Допускаемыми цепочками будут в точности те, которые могут быть свернуты указанным процессом к начальному символу. Это как раз те самые цепочки, которые при «обычном» использовании грамматики выводятся из начального символа.

Чтобы с помощью категориальных грамматик можно было порождать цепочки, достаточно переформулировать правила сокращения как правила развертывания (т.е. фактически прочитать их наоборот):

Всякую категорию ψ можно развернуть в $\varphi[\varphi\backslash\psi]$, где φ — произвольная категория (левое развертывание).

Всякую категорию φ можно развернуть в [φ/ψ]ψ, где ψ — произвольная категория (правое развертывание). Тогда легко сообразить, как будет осуществляться процесс порождения.

Таким образом, формальные грамматики по существу нейтральны по отношению к процессам порождения и допускания. Обычное деление грамматик на порождающие и распознающие имеет естественное историческое объяснение. Те грамматики, которые называют порождающими (соответственно распознающими), разрабатывались с целью использования их как раз для порождения (распознавания). Однако независимо от цели создания грамматики, она может использоваться в «обе стороны».

2.7. Основные свойства языков

2.7.1. Свойства языков

Приведем ряд теорем, характеризующих основные свойства языков, порождаемых четырьмя основными типами грамматик.

Теорема 20 (Поста). Любой язык типа *0* является рекурсивно-перечисленным (хотя, возможно, и нерекурсивным) множеством цепочек. Любое рекурсивно-перечислимое множество цепочек является языком типа *0*.

В силу этой теоремы теория языков типа *0* охватывается общей теорией рекурсивных функций и поэтому обычно в теории языков языки типа *0* не рассматриваются.

Теорема 21 (Хомского). Языки типа *1, 2, 3* являются рекурсивными множествами цепочек, т.е. для каждого языка из указанных типов существует алгоритм, позволяющий по заданной грамматике этого языка распознавать принадлежность любой цепочки языку. Обратное утверждение неверно, т.е. существуют рекурсивные множества, не являющиеся языками. Это дало основание для проведения специальных исследований этих языков.

Теорема 22 (Хомского). Языки типа *3* являются регулярными множествами цепочек. Поэтому их называют иногда автоматными.

Таким образом, если класс языков типа *0* оказался столь широким, что теория его совпала с общей теорией рекурсивных функций (теорией алгоритмов), то класс языков типа *3* оказался, наоборот, чрезмерно узким, совпадающим с хорошо изученным классом регулярных множеств (в теории конечных автоматов). Поэтому особое внимание при построении теории языков уделялось языкам типа *1* и *2*.

Теорема (23). Существует язык типа *0*, не являющийся языком типа *1*. Эта теорема следует из теории рекурсивных функций.

Теорема (24). Существует язык типа *1*, не являющийся языком типа *2*.
Примером такого языка может служить язык $L\{a^n b^m a^n b^m c^3\}$.

Теорема (25). Существует язык типа *2*, не являющийся языком типа *3*.
Примерами таких языков являются языки

$$\{a^n b^n\} \text{ и } \{xx^{\tau}\}.$$

Приведенный ряд теорем определяет следующее отношение: между различными типами языков:

$$\{L \text{ тип } 3\} \subseteq \{L \text{ тип } 2\} \subseteq \{L \text{ тип } 1\} \subseteq \{L \text{ тип } 0\}.$$

Языки типов *0—3* образуют систему, грамматики их представляют систему правил единого типа с последовательно возрастающими ограничениями. Однако они не исчерпывают возможностей построения грамматик такого же вида, но с другими ограничениями, •которые порождают языки, отличные от уже рассмотренных.

В некоторых работах выделяются подклассы языков, для которых могут быть установлены интересные законы и свойства, не имеющие места для класса в целом.

Так, в грамматиках типа *1* рассматривается язык, называемый языком Парика, грамматика которой имеет дополнительные ограничения: запрещены правила вида $\varphi_1 A \varphi_2 \rightarrow \varphi_1 B \varphi_2$, т.е. правила, включающие замену одного нетерминального символа другим. Хотя этот язык представляет интерес для математической лингвистики, он изучен недостаточно.

Из всех четырех типов языков наиболее интересными являются языки типа *2* — бесконтекстные.

Во многом это определяется возможностью их использования для исследования языков программирования. Как известно, программа цифровой вычислительной машины может рассматриваться как цепочка символов в некотором алфавите. Тогда некоторый язык программирования представляет собой бесконечное множество таких цепочек. Этот язык имеет грамматику и конечный набор правил, определяющих построение программ. Кроме того, языки типа *2* совпадают с алголоподобными языками, т.е. языками, которые могут быть заданы в нормальной форме Бэкуса — общепринятой формализации языков программирования.

В то же время при исследовании естественных языков используются так называемые категориальные грамматики, которые позволяют отбирать из числа всех возможных

предложений — правильно построенные. Класс языков, определяемых категориальными грамматиками, совпадает с классом языков, порождаемых грамматиками типа *2*.

Языки типа *2* наиболее близки к языкам регулярных событий (типа *3*), с исчерпывающей полнотой исследованным в теории конечных автоматов. Кроме того, языки типа *2* получили адекватное представление с помощью математической модели автомата с магазинной памятью.

Наконец, языки этого типа более доступны для математического изучения, чем, например, языки типа *1* или язык Парика, и поэтому больше всего используются при исследовании распространенных языков программирования и в математической лингвистике. Из сказанного становится ясным, почему для языков типа *2* получено наибольшее количество результатов и исследовано большое число их разновидностей, каждая из которых является частным случаем бесконтекстного языка.

Рассмотрим еще один подкласс бесконтекстного языка, который определяется с помощью дополнительных ограничений на систему правил.

Пусть имеется терминальный словарь $V_T = \{a_1, \ldots, a_n\}$. Обозначим через V'_T расширенный словарь, в котором каждому терминальному символу a_i, сопоставлен символ a'_i: $V'_m = \{a_1, a'_1, \ldots, a_n, a'_n\}$. Язык, порождаемый грамматикой $G = (V'_T, \{S\}, P, S)$,правила которой имеют вид

$$P: \begin{matrix} S \to \Lambda, \\ S \to Sa_iSa'_iS, \end{matrix}$$

называется языком Дика и обозначается буквой Д.

Предложения со этого языка обладают следующим свойством: если каждую пару $a_1a'_i$ $(i = 1, 2, \ldots, n)$ соседних символов, содержащихся в ω, разрешить заменять пустым символом Λ, то для каждого предложения ω найдется такая последовательность этих замен, с помощью которой ω сведется к пустой цепочке. Интерес к языкам Дика объясняется тем, что они очевидным образом связаны со скобочными структурами, обычными для естественных и искусственных языков.

Представим себе набор формул некоторого математического вычисления или программу, записанную в языке типа АЛГОЛ. Это будет текст, в котором будут встречаться знаки, всегда употребляемые только попарно: левые и правые скобки всех видов (круглые, квадратные, фигурные, ломаные) или операторные скобки, состоящие из слов «начало» и «конец». Устраним из текста все, кроме знаков указанного типа. Получится новый текст, построенный по некоторым строгим правилам. Подобные тексты дают представление о языках Дика.

2.7.2. Операции над языками

К языкам, как и ко всяким множествам, могут быть применены различные операции. Прежде чем рассматривать операции над языками, определим свойство замкнутости множества. Говорят, что множество замкнуто относительно некоторой операции, если результат применения ее к любому элементу множества или к любой паре элементов содержится в этом множестве.

Для языков обычным образом определяются операции объединения, пересечения и операции дополнения относительно фиксированного словаря V_T.

Объединением языков L_1 и L_2 (обозначение: $L_1 \cup L_2$) называется множество всех слов, принадлежащих хотя бы одному из языков.

Эта операция представляет собой обычное теоретико-множественное объединение; она коммутативна и ассоциативна:

$$L_1 \cup L_2 = L_2 \cup L_1$$
$$(L_1 \cup L_2) \cup L_3 = L_1 \cup (L_2 \cup L_3).$$

При тех же условиях пересечением двух языков (обозначение: $L_1 \cap L_2$) называется множество всех слов, принадлежащих одновременно обоим языкам.

Эта операция представляет собой обычное теоретико-множественное пересечение; она тоже коммутативна и ассоциативна.

Дополнением языка L до V_T называется множество всех слов, принадлежащих V^*_T, но не принадлежащих L.

Само множество V^*_T есть язык, дополнением которого до V^*_T является пустой язык.

Подчеркнем, что язык, содержащий пустое слово e, не является пустым.

Операцию дополнения рассмотрим на примере

$$V_T = \{a, b\};$$
$$L = \{a^m b^n \mid m \geqslant 1, \ n \geqslant 1\};$$

$V^*_T \backslash L = (L_1 \cup L_2 \cup L_3)$, где L_1 — множество всех слов, начинающихся с b, L_2 — множество всех слов, начинающихся с $a^m b^n a$ и $L_3 = \{a\}^*$, т. е. L_3 есть множество всех слов, состоящих только из a.

Отношение языков типов *0, 1, 2, 3* к булевским операциям определяют следующие четыре теоремы, которые мы приводим без доказательства.

Теорема (26). Класс языков типа *0* замкнут относительно операций объединения и пересечения. Алгоритмически неразрешима, задача определения того, является ли дополнение языка типа *0* относительно фиксированного словаря также языком типа *0*.

Теорема (27). Класс языков типа *1* замкнут относительно операций объединения и пересечения.

Вопрос о том, какому классу принадлежит дополнение языков типа 1 по отношению к фиксированному словарю, остается открытым.

Теорема 28 (Бар-Хиллела, Перльса и Шамира). Класс языков типа *2* замкнут относительно операций объединения, но не замкнут относительно операции дополнения по отношению к словарю, содержащему не менее двух символов, а также по отношению к операции пересечения.

Теорема 29 (Клини). Класс языков типа *3* замкнут относительно всех булевских операций.

Кроме булевских операций над языками рассматриваются так же операции умножения или конкатенации, итерации, транспозиции (зеркального отображения языка), гомоморфизма и некоторые другие.

Произведением (конкатенацией) двух языков (обозначение: $L_1 \cdot L_2$) называется множество всех слов, которые можно получить следующим способом: берется некоторое слово из L_1 и к нему при соединяется конкатенацией справа некоторое слово из L_2, т. е.

$$L_1 L_2 = \{X_1 X_2 \mid X_1 \in L_1, \ X_2 \in L_2\}.$$

Эта операция (называемая умножением языков) не совпадает с декартовым умножением; она ассоциативна, но не коммутативна.

Пусть $V_T = \{a, b, c\}$. Рассмотрим язык $\{a\}$, состоящий из одного однобуквенного слова a. Тогда произведение

$$L = V^*_T$$

есть множество всех слов, начинающихся с *a*.

Операция итерации (операция Клини). Поскольку операция умножения языков ассоциативна, мы можем возводить данный язык в степень:

$$L^2 = LL,\ L^3 = (LL)L = L(LL),\ \ldots$$

Клини предложил рассматривать объединение

$$L^* = E \cup L \cup L^2 \cup L^3 \cup \ldots \cup L^n \cup \ldots$$

всех последовательных степеней языка *L*. Это объединение обозначается *L** и называется итерацией языка *L*.

Например, мы можем рассматривать алфавит

$$V = \{a,\ b,\ \ldots\}$$

как язык, состоящий из однобуквенных слов. Тогда V^2 — множество всех двубуквенных слов; V^3 — множество всех трехбуквенных и т.д. Поэтому $EU\ V\ U\ V^2\ U\ V^3U\ \ldots$ — есть множество всех слов над *V*, т.е. *V**.

Доказана замкнутость классов регулярных и бесконтекстных языков относительно умножения и итерации. Язык *L* называется регулярным, если существует конечный автомат *A*, такой, что $L = L(A)$.

Относительно языков *0, 1, 2, 3* имеет место следующая теорема.

Теорема (30). Классы языков типов *0, 1, 2, 3* замкнуты относительно операции зеркального отображения, определенной следующим образом.

Пусть дан язык $L \subset V^*_T$; через L^T обозначается язык, состоящий из обращений всех слов языка *L*:

$$L^T = \{X^T \mid X \in L\}.$$

Эта операция является инвалютивной (т. е. совпадает со своей обратной операцией):

$$(L^T)^T = L.$$

Кроме того, она связана с умножением языков следующим соотношением:

$$(LM)^T = M^T L^T.$$

Например, язык $V^*_T(V^*_T)^T$ совпадает с V^*_T, поскольку $(V^*_T)^T = V^*_T$ и $E \in V^*_T$.

Введем теперь понятие об операции гомоморфизма.

Поставим в соответствие элементу a словаря V_T конечный словарь $V_{та}$. Обозначим через $V^*_{та}$ множество всех цепочек из словаря $V_{та}$, а через $\tau(a)$ — какую-либо цепочку из $V^*_{та}$. Таким образом, функция τ определена на отдельных символах *a* из словаря V_T. Определим теперь функцию τ на предложениях из словаря V_T следующим образом: если $x = ai_1ai_2 \ldots ai_s$, то $\tau(x) = \tau(ai_1)\tau(ai_2) \ldots \tau(ai_s)$.

Так, определенная функция τ отображает подмножество цепочек *L* из V^*_T, в некоторое подмножество цепочек из $(\bigcup_a V_{та})^*$, т.е. $\tau(L) = (\bigcup_a V_{та})^*$. Операция $\tau(L)$ отображения языка *L* с помощью функции τ называется операцией гомоморфизма.

Теорема 31 (Бар-Хиллела, Перльса и Шамира). Классы языков типа *0, 2* и *3* замкнуты относительно операции гомоморфизма.

Для языков же типа *1* (контекстных) имеет место следующая теорема.

Теорема 32 (Гинсбурга и Роуза). Если V_T содержит хотя бы два элемента, то класс контекстных языков не замкнут относительно операции гомоморфизма.

Проекция языка. Для каждого слова в алфавите *X*×*Y*

$$\langle x(1)\,y(1)\rangle\langle x(2)\,y(2)\rangle \ldots \langle x(t)\,y(t)\rangle$$

его проекциями в *X* и *Y* называются соответственно слова

$$x(1) \ldots x(t);$$
$$y(1) \ldots y(t).$$

Другими словами, если задан язык *L* в алфавите *X*×*Y*, то проекцией языка *L* в *X* называется язык, состоящий в точности из проекций в *X* слов языка *L*.

Цилиндр языка. Пусть заданы алфавит *Y* и язык *L* в алфавите *X*. *Y*-цилиндром языка *L* называется язык *L'*, состоящий из всех слов в алфавите *X*×*Y*, *X* проекции которых принадлежат *L*.

Свойства языков по отношению к рассмотренным операциям сведены в табл. 1.

Таблица 1

№ п/п	Операция	Тип языка			
		0	1	2	3
1	Объединение	1	1	1	1
2	Пересечение	1	1	0	1
3	Дополнение	0		0	1
4	Транспозиция (зеркальное отображение)	1	1	1	1
5	Произведение (конкатенация)	1		1	1
6	Итерация	1		1	1
7	Гомоморфизм	1	0	1	1

В этой таблице единица обозначает замкнутость относительно соответствующей операции класса языков определенного типа, нуль — незамкнутость. Пустая клетка означает, что вопрос пока не решен.

Упражнения:

1. Дан язык $L = \{aab, aaaab, aaaaaab\}$.

Выполнить операции умножения, итерации и транспозиции над ним.

2. Задан словарь терминальных символов $V = \{c, b\}$ и язык $L = \{c^n b c^n | n \geq 1\}$.

Определить дополнение языка.

3. Заданы языки $L_1 = \{abbc, ab, abcc, ac\}$ и $L_2 = \{xy, xyz, xz, xzy, yz, yzx\}$. Выполнить операцию умножения этих языков.

4. Задан язык $L = \{a^n b^n | n \geq 1\}$. Выполнить операции транспозиции, умножения, итерации.

5. Пусть $V_T = \{a, b\}$, $L = \{a^i b^j a^j | i, j \geq 1\}$, $M = \{a^j b^j a^i | i, j \geq 1\}$, $L_1 = \{a^i b^j a^k | i, j, k \geq 1\}$. Показать, что язык $L_2 = L \cap M$ не является контекстно-свободным, а язык $L_3 = L_1 — (L \cap M)$ является контекстно-свободным.

6. Задан язык $L = \{a^n c b^n | n \geq 0\}$. Выполнить все возможные операции над данным языком.

2.8. Методы анализа грамматики языков

2.8.1. Общие положения

Основная проблема теории языков заключается в том, чтобы формально анализировать классы языков с целью разработки возможных методов как моделирования, так и эффективной обработки языков машинными средствами.

В основном это сводится к определению логической структуры языков, т.е. системы правил, определяющих синтаксис грамматики. Если форма правил (т.е. синтаксическая часть грамматики) точно установлена, то возможно проведение следующего ряда исследований по языку:

установление связи между видами языков с их структурными деревьями и формой синтаксических правил;

изучение структурных свойств языков, порождаемых некоторой формой правил *G*-грамматики;

определение относительного богатства или бедности различных форм грамматик, порождающих *L* языки.

установление разного рода проблем разрешимости *L* языка относительно заданной *G*-грамматики и класса *G*-грамматик;

установление мощности *G*-грамматики, порождающей моделирующий *L* язык в зависимости от контекста применения последнего;

оценка порождающей способности *G*-грамматик, а следовательно, определение эквивалентности порождаемого разными грамматиками множества *L* языков;

оценка меры сложности, порожденных *G*-грамматиками предложений *L*-языка;

установление сводимости *G*-грамматик различной сложности к более простым *G*-грамматикам;

определение возможных методов построения распознающих *G*-грамматик для множества *L*-языков и ряд других не менее важных исследований, связанных с изучением свойств класса формальных грамматик в целях их практического использования.

Наибольшее развитие получила проблема синтаксического анализа грамматик и контроля соответствующих им языков, состоящая в том, что для любой цепочки $\omega \in L$ определяется ее формальная правильность (или неправильность) и производится синтаксический разбор.

Проблема синтаксического анализа и контроля языка возникла в связи с потребностями трансляции — разработкой специализированной программирующей программы, с помощью которой машина переводит введенную в нее программу на машинном языке.

Каждый из трансляторов построен в основном для некоторой конкретной пары языков: одного — входного и второго — выходного. Упрощенно говоря, такие трансляторы работают по принципу словаря: для каждой конкретной комбинации символов, имеющей смысл в данном входном языке, этот транслятор выдает определенную последовательность символов выходного языка.

Построение таких трансляторов представляет собой весьма трудоемкую работу, так как нужно рассмотреть все возможные (имеющие смысл) выражения входного языка и каждому из них сопоставить соответствующие выражения на выходном языке.

Определение имеющих смысл выражений входного языка осуществляется в результате синтаксического анализа этого языка. Суть его состоит в том, что на основании синтаксических правил грамматики осуществляется проверка грамматической правильности заданных предложений или слов языка путем грамматического разбора. Таким образом, задачей грамматического разбора является анализ предложений с точки зрения установления их грамматической правильности.

Под грамматическим разбором понимается процесс определения структуры предложения или слова α $\alpha \in G$ в соответствии с правилами, определяемыми *G*.

Установление того факта, что предложение или слово является грамматически правильным, может быть выполнено не одним способом. Грамматический разбор может в некоторых случаях выполняться более чем одним способом.

В качестве примера языка, для которого грамматический разбор может быть осуществлен не единственным образом, рассмотрим язык, задаваемый грамматикой *G* с правилами.

$$HBC \to HBc;$$
$$HBC \to hBC;$$
$$BC \to bc;$$
$$HB \to hb.$$

Рассмотрим слово *hbc*. Это слово может быть разобрано двумя различными способами. Деревья грамматического разбора этого слова приведены на рис. 15.

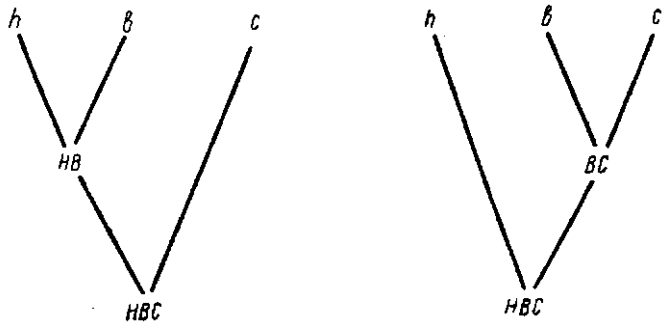


Рис. 15

В простейшем виде грамматический разбор состоит в том, что бы, начав с первого правила, просматривать список правил сверху вниз, пока не будет найдено применимое правило, применить его и повторить этот процесс столько раз, сколько нужно. Эта методика в применении к нашим правилам дала бы грамматический разбор, показанный на правом рисунке.

Рассмотрим еще один метод грамматического разбора, основанный на порядке просмотра разбираемого слова или предложения. Правила грамматики языка имеют вид:

$$A \to BC;$$
$$B \to DE;$$
$$C \to FH.$$

Необходимо выполнить грамматический разбор слова *DEFH*.

Разбор данного слова можно проводить *слева направо* или *справа налево*. В первом случае в слове выбирается первая доступная для замещения совокупность символов *DE* в соответствии с заданными грамматиками. Вместо нее подставляются символы из некоторого правила. После этого полученное слово вновь просматривается, начиная слева, с целью поиска совокупности символов для замещения по правилам грамматики.

Для нашего примера просмотр слева направо дает дерево грамматического разбора, представленное на рис. 16а.

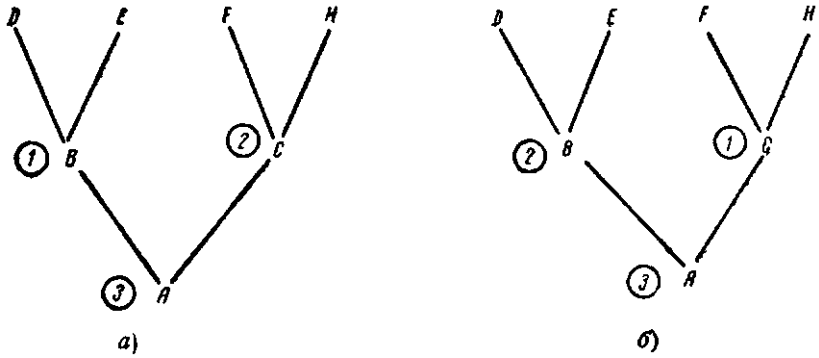


Рис. 16

Цифры в кружках означают порядок разбора. Результативные деревья идентичны. Разница состоит лишь в процессе разбора: для *а* — слева по строке, для *б* — справа по строке.

Эта разница в методе разбора может быть исключена введением понятия канонически *упорядоченного грамматического разбора.*

Канонический вид грамматического разбора — это разбор, который применяется слева направо по строке. При этом в первую очередь разбирается крайне левая часть предложения, если это возможно, прежде чем продвинуться по строке вправо для поиска доступной разбору ситуации. На рис. *а* представлен канонический грамматический разбор предложения. Однако

канонически упорядоченный грамматический разбор не всегда может быть использован. Рассмотрим примеры:

Пример 1. Задана грамматика

$$\begin{matrix} A \to x \\ A \to Ax \end{matrix} \quad \text{(левая рекурсия)}$$

и строка *xxxx*. Канонический разбор приведен на рис. 17.

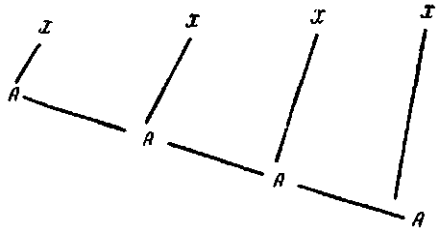


Рис. 17

Пример 2. Задана грамматика

$$A \to x$$

(правая рекурсия)

$$A \to xA$$

и слово *xxxx*. Грамматический разбор для данного слова не может быть каноническим, так как он не может быть выполнен (приводит в тупик). Грамматический разбор в этом случае может быть выполнен только при разборе справа по строке (рис. 18).

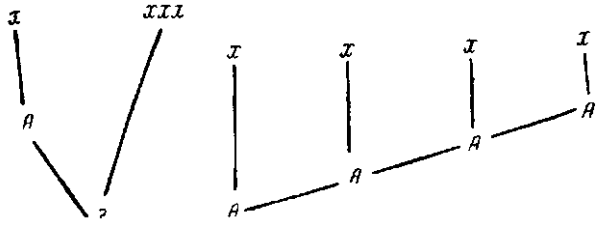


Рис. 18

Пример 3. Задана грамматика

$$\begin{matrix} A \to x \\ A \to xAx \end{matrix} \quad \text{(центральная рекурсия)}$$

и слово *xxxxx*. Выполнить грамматический разбор. В этом случае он не может быть успешно завершен ни при просмотре предложения слева направо (канонический разбор), ни при просмотре справа налево. В этом случае разбор на каждом этапе начинается с середины разбираемого предложения (рис. 19).

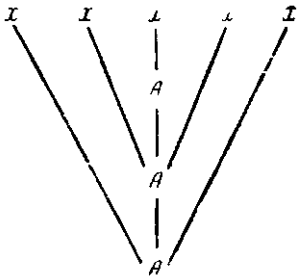


Рис. 19

Из рассмотренных примеров *1, 2, 3* можно сделать вывод, что путь разбора предложения определяется типом рекурсии в правилах вывода грамматики. Правая рекурсия предопределяет грамматический разбор, начинающийся справа. Левая рекурсия предопределяет успешный канонический вид грамматического разбора.

Центральная рекурсия предопределяет грамматический разбор, начинающийся с середины предложения. Но не всегда путь грамматического разбора может быть легко установлен по правилам грамматики, как это было в приведенных примерах. Рассмотрим следующий пример.

Пример 4. Заданы правила грамматики в виде

$$A \to wx;$$
$$B \to Ay;$$
$$C \to Bz \mid wD;$$
$$D \to xE;$$
$$E \to yv.$$

Необходимо произвести грамматический разбор строк *wxyz* и *wxyv*.

Разбор первой строки может быть успешно выполнен в результате применения канонического вида разбора.

Результат такого разбора приведен на рис. 20а. Использование же канонического разбора для второй строки заводит в тупик (рис.20б).

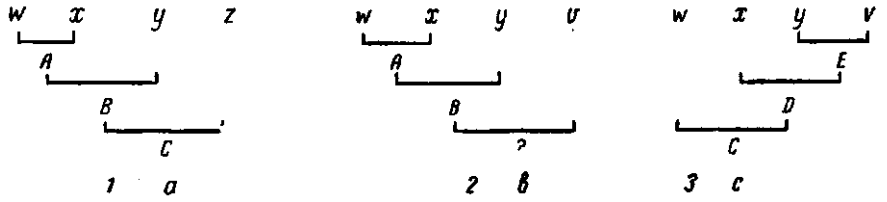


Рис. 20

Успешное выполнение грамматического разбора второй строки осуществляется при условии, если разбор начат со следующей, начиная слева, подстроки, т.е. с *yv*. Результаты такого разбора приведены на рис. 20в. Таким образом, задача грамматического разбора — успешное проведение анализа предложений. Порядок же грамматического разбора зависит от правил вывода (синтаксиса) и вида анализируемых строк или предложений.

Используя формализацию как критерий классификации, все существующие методы грамматического разбора можно разделить на *эвристические* и *формализованные*.

Формализация методов заключается в систематизации правил, определяющих правильность или ошибочность исходной строки относительно заданной грамматики при применении данного метода на каждом шаге (этапе) грамматического разбора.

Эвристические методы не систематизированы по отношению к каждому шагу, т.е. они говорят только, правильна ли данная строка, лишь после окончательного прохождения анализируемого текста.

Эвристический метод известен еще под названием метода проб и ошибок, поиска и подстановок, поскольку правильный путь порождений находится после проверки всех возможных путей решения (разбора). Ограниченность использования эвристических методов заключается в том, что:

1) правильность или ошибочность строки определяется не сразу на каждом шаге разбора, а лишь после окончания анализируемого текста;

2) правильный путь порождения находится после проверки всех возможных путей разбора;

3) выбор ложного пути требует обратного возвращения к последнему, правильно определенному состоянию;

4) при реализации на машине очень трудно осуществляется связка семантических правил с синтаксическими.

Все это отрицательно сказывается на методе с точки зрения потери времени.

Вместе с тем эвристическим методам присущи и некоторые преимущества: возможность применения ко всем языкам, что делает их универсальными; ориентация либо «на цель», либо «от цели», что определяет два направления эвристического метода — «сверху вниз» и «снизу вверх».

Грамматический разбор методом «снизу вверх» строки s языка L, порождаемого грамматикой $G = <V_T, V_N, S, P>$, начинается со строки s и состоит в просмотре последовательностей, получаемых в результате разбора, ведущего к S (к цели). Формализованно цель такого разбора можно представить так: $s \Rightarrow S$, т.е. в результате грамматического разбора определяется, является ли данная строка предложением.

Все рассмотренные ранее примеры грамматического разбора неявно демонстрировали этот метод.

Грамматический разбор методом «сверху вниз», называемый методом «спуска» (или «рекурсивного спуска»), начинается от цели *S* (т.е. от отправного правила порождения или вывода), и далее рассматривается последовательность таких порождений, которые бы привели к *s*. Формализованное представление такого разбора — *S* => *s*. В результате грамматического разбора методом «сверху вниз» определяется состав предложений языка.

Оба эти метода можно представить одними и теми же деревьями порождений, только в одном случае корень дерева внизу, в другом — вверху. Так, разбор слова *wxyz* для примера 4, приведенный обоими методами, представлен на рис. 21.

Формализованные методы появились и появляются в связи с разработкой трансляторов. При этом в новейших методах иногда находят отражение хорошие черты одних и устраняются недостатки других ранее описанных методов.

Разработка каждого метода привязана к определенным классам машин (малого, среднего или большего типа) и к определенным классам языков (в зависимости от того, с какими языками работает машина). Поскольку все языки программирования относятся ко второму классу по классификации Хомского, то и методы в основном ориентированы на второй класс языков.

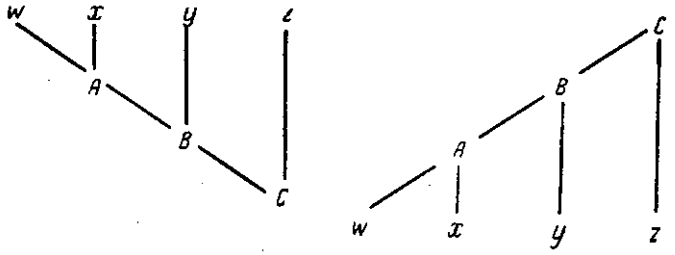


Рис. 21

Некоторые методы контроля вносят ограничения в грамматику языка. Обычно ограничения накладываются на правую часть правил: либо по числу символов, либо по их составу.

2.8.2. Метод Эйкела, Паула, Бауера, Замелзона

По сравнению с эвристическим этот метод является формализованным, но может управлять алгоритмом грамматического разбора, построенным только для языков определенного класса. Метод применим к языкам второго класса по классификации Хомского, т.е. к КС-языкам, правая часть которых содержит не более двух символов: $A \rightarrow a$ и $A \rightarrow aB$.

Ограничение длины правой части порождений, которая равна 1 или 2, не сужает общности метода, поскольку любое порождение вида $A::=B_1B_2B_3...B_kB_{k+1}...B_iB_{i+1}...B_n$ возможно свести к множеству порождений: $A_0::=B_1A_1$, $A_1::=B_2A_2$, $A_2::=B_3A_3...A_{n-t}::=B_n$, где $A_1=B_2B_3...B_kB_{k+1}...B_iB_{i+1}...B_n$; $A_2 = B_3...B_kB_{k+1}...B_iB_{i+1}...B_n$ и т. д.

В этом случае для запоминания начальных символов B_1, B_2, B_3, ..., B_n и сформированных промежуточных величин A_1, A_2, A_3, ..., A_{n-1} используется специальное устройство, которое называют либо магазином, либо стэком. Самое распространенное название этого устройства — стэк. Стэк устроен так, что, когда в него вводится некоторый символ, все символы, бывшие там ранее, опускаются вниз, а когда верхний символ стирается, остальные символы автоматически поднимаются на шаг вверх.

Грамматический разбор методом Бауэра-Замелзона осуществляется механизмом четырех таблиц: 1) стэк символов разбираемой строки; 2) синтаксическая таблица состояний n; 3) таблица действий N; 4) таблица семантических правил r.

Синтаксическая таблица является управляющей: состоянию n соответствует определенное действие N — порождение или свертка. В момент определения состояния синтаксической таблицы и соответствующих ей действий N подключается таблица семантических правил. При этом применение правил семантики $r \in E^*$ обеспечивает образование кодов по входной анализируемой строке — s.

Осуществление грамматического разбора можно представить последовательностью нескольких шагов:

Шаг 1. Занесение символов входной строки в стэк. Если строка имеет вид $B_1B_2B_3... B_kB_{k+1}... B_i... B_n$, то в стэке символы можно расположить так: $B_iB_{i-1}B_{i-2}... B_1$.

Шаг 2. Выявление возможного соответствия двух верхних символов B_iB_{i-1} стэка и текущего символа входной строки B_{i+1} состоянию n_i синтаксической таблицы. При поиске просматриваются на соответствие только два верхних элемента стэка B_i и B_{i-1}.

Шаг 3. Если поиск положительный, то происходит свертка подстроки, т.е. замена либо B_{i-1} на A_{i-2}, либо $B_{i-1}B_i$ на A_{i-1}, либо B_iB_{i+1} на A_i и занесение в стэк B_{i+1}.

Шаг 4. Если поиск отрицателен, то содержание стэка B_iB_{i-1} сдвигается вниз и заносится B_{i+1}.

Шаг 5. Производится чтение следующего символа входной строки.

2.8.3. Метод Флойда

Метод Флойда основан на использовании отношения «предшествия» между двумя смежными терминальными символами, т.е. он применим также к языкам второго класса по классификации Хомского, но с известным ограничением: порождения не должны иметь вид $c \to xW_1W_{2y}$, где W_1W_2 — смежные нетерминалы — $W_1W2 \in (V - V_T)$, а x, y — строки (возможно и пустые).

Отношения предшествия определяют состояние терминальных символов при их свертке. Возможные отношения предшествия по Вирту и Веберу записываются такими знаками: $\doteq F$, $\bullet>$, $<\bullet$.

Пусть при грамматическом разборе пары $T_iT_j \in V$, тогда:

1) отношение предшествия со знаком $>\bullet$ будет иметь место тогда и только тогда, если T_j, при свертке строки s является крайне левым символом подстроки $s' \in s$, или:

$$\ldots T_i \,|\, T_j \xrightarrow{\qquad} s' \ldots$$

2) отношение предшествия со знаком имеет место тогда и только тогда, если T_i является крайне правым символом в s', или:

$$s' \xleftarrow{\qquad} \ldots\ldots T_i \,|\, T_j \ldots\ldots$$

3) и, наконец, предшествие со знаком $\doteq$ имеет место тогда и только тогда, если оба символа являются частью редукцируемой подстроки s':

$$\ldots\ldots \underbrace{T_iT_j}_{s'} \ldots\ldots$$

Если между парой символов T_iT_j нельзя поставить ни одного из отношений предшествия $|T_i \doteq T_j,\ T_i \bullet > T_j,\ T_i < \bullet Tj|$, то это говорит о синтаксической ошибке. Знаки 1/$\doteq$; 2/$\bullet>$; 3/$<\bullet$ между символами T_i и T_j можно поставить тогда и только тогда, если соответственно:

$$1)\ c \to xT_iT_jy;$$

$$2)\ c \to xT_iC_ey \text{ и } C_e \Rightarrow T_jz;$$

$$3)\ c \to xC_kT_jy \text{ и } C_k \Rightarrow zT_i, \text{ или}$$

$$c \to xC_kC_ey \text{ и } C_k \Rightarrow zT_i,\ C_e \Rightarrow T_jw,$$

где w, z, x, y — любые строки.

Отношения предшествия, которые могут быть между парой символов в строке и которые определяются правилами вывода, представляются в виде матрицы предшествий. Рассмотрим матрицу предшествий для одного примера.

Исходная матрица и правила вывода для некоторой грамматики заданы в следующем виде:
Строка: 'if false then 0'

Правила вывода:

$$P: \begin{cases} <\text{prog}> \to <\text{state}><\text{string}> \\ <\text{state}> \to \text{if false} \\ <\text{string}> \to \text{then 0} \end{cases}$$

№ п/п	Отношение предшествий символов	Основание
1	if $\doteq$ false	<state> → if false
2	then $\doteq$ 0	<string> → then 0
3	<state> $\doteq$ <string>	<prog> → <state><string>
4	false $\bullet>$ <string>	<prog> → if false <string>
5	false $\bullet>$ then	<prog> → if false then 0
6	<state> $<\bullet$ then	<prog → <state> then 0
7		ошибка

Матрица предшествий будет иметь вид:

Символ	<prog>	<state>	<string>	if	false	then	0
<prog>							
<state>			$\doteq$			$<\cdot$	
<string>							
if					$\doteq$		
false			$\cdot>$			$\cdot>$	
then							$\doteq$
0							

Число элементов такой матрицы очень велико, поэтому Вирт и Вебер обобщили метод Флойда и ввели новое понятие — «функция предшествия». Так как в установлении соотношения предшествия принимают участие два символа, то требуется по крайней мере две функции — f и l, чтобы для любой пары символов T_iT_j выполнялись соотношения:

$$\begin{aligned} f(T_i) &\doteq l(T_j) \sim T_i \doteq T_j; \\ f(T_i) &\cdot> l(T_j) \sim T_i \cdot> T_j; \\ f(T_i) &<\cdot l(T_j) \sim T_i <\cdot T_j. \end{aligned} \qquad (14)$$

Для существования функций предшествия f и t необходимым условием является перекомпоновка матрицы предшествий таким образом, чтобы выявились три независимые области, в каждой из которых определено одно и только одно отношение из отношений предшествия:

$\cdot> \in$ I (область); $\doteq \in$ II (область); $<\cdot \in$ III (область).

Построим матрицу предшествий для грамматики с правилами вывода:

$$<\text{expr}> \to <\text{string}><\text{string}>;$$
$$<\text{expr}> \to \underline{\text{if}}\ \underline{\text{if}};$$
$$<\text{string}> \to <\text{state}><\text{expr}>\text{d};$$
$$<\text{state}> \to \underline{\text{else}};$$
$$<\text{expr}> \to <\text{string}>\underline{\text{if}};$$
$$<\text{expr}> \to \underline{\text{if}}<\text{string}>.$$

Символ	else	<state>	if	<string>	a	<expr>
else	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$		$\cdot>$
a	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	$\cdot>$	
if	$<\cdot$	$<\cdot$	$\doteq$	$\doteq$	$\cdot>$	III
<string>	$<\cdot$	$<\cdot$	$\doteq$	$\doteq$	$\cdot>$	
<state>	$<\cdot$	III $<\cdot$	$<\cdot$	$<\cdot$	II	$\doteq$
<expr>					$\doteq$	

$\cdot> \in$ I {область};
$\doteq \in$ II {область};
$<\cdot \in$ III {область}.

В обоих случаях в матрицах не имеют места два вида отношений предшествия для любой пары (т.е. нет клетки с символами $\bullet>$ и $\doteq$ или $<\bullet$ и $\doteq$.

Если такие порождения не определяются ни одним из правил грамматики, то грамматику называют простой грамматикой предшествия.

2.8.4. Метод Наура

Основная идея разбора заключается в следующем: фраза обрабатывается терминал за терминалом в одном направлении — слева направо (канонический метод разбора). Для каждого очередного терминала формируется определенное синтаксическое состояние, т.е. предсказывается, какая конструкция может следовать в данной фразе за этим терминалом. Если следующий терминал удовлетворяет этому состоянию, анализ продолжается, в противном случае — прекращается. Поскольку терминал может предсказывать много разных конструкций, то образуется много вариантов анализа, причем в случае синтаксически однозначной фразы анализ доводится до конца только по одному варианту. В результате анализа фразы составляется система синтаксических состояний, одно из которых помечает границу последнего терминала, входящего в определенную языковую конструкцию (границу конструкции можно определить с помощью скобок).

Логику действия грамматического разбора можно описать таблицей (табл. 2), которая включает: 1) основную строку, где помещены основные символы (терминалы) языка; 2) основной столбец, где перечисляются синтаксические состояния. Столбец является специфической особенностью данного метода — его можно изобразить:

а) либо в общей таблице, где сначала записываются синтаксические состояния, и на пересечении с определенным терминалом записан текущий номер состояния; б) либо рассматривать как две синтаксические таблицы, одна из которых по данному синтаксическому состоянию определяет «начало», а другая «конец» возможных конструкций; 3) как рабочую зону, в которой записываются но мера новых состояний, полученных из комбинации: элемент столбца «состояние» — элемент строки «терминал».

Таблица 2

Входиая строка; a5c615 : = ;

№ п/п	Символы / Состояние	±	;	Буква	Цифра	:=
1	После ;		1	2	3	
2	После буквы				3	
3	Идентификатор			2	3	4
4	Левая часть					

Поскольку содержимое основной строки и основного столбца в процессе разбора не меняется, то терминалы и синтаксические состояния используются как справочники.

В отличие от них рабочая зона (называемая еще накопитель синтаксических состояний) — это место, где происходит переработка всех промежуточных результатов. Ее можно сравнить и рассматривать как неограниченное число стэков. В каждом стэке ввод осуществляется по одному терминалу и только с одного конца зоны. Так что терминал, введенный в стэк последним, вынимается из него первым.

Изложенная идея синтаксического анализа методом Наура имеет некоторое сходство с принципом работы машины Тьюринга с тремя лентами.

Вкратце изложим этот принцип. Машина Тьюринга имеет три ленты, ограниченные слева и неограниченные справа: а) входная — на ней расположен словарь языка (основные символы); б) рабочая — содержит набор синтаксических состояний; в) выходная — содержит набор левых и правых скобок, помеченных терминалами типов конструкций, и три головки (по одной на каждой ленте).

На каждом шаге машина работает только на одной ленте, выполняя одно из следующих действий:

1) на входной ленте читается символ за символом, сдвигая головку на одну ячейку вправо;

2) на рабочей ленте либо пишется символ и головка сдвигается на одну ячейку вправо, либо на одну ячейку влево и стирается записанный в ней символ, но после такого действия всегда влево от головки остаются непустые ячейки, а вправо — пустые;

3) на выходной ленте только записываются символы, и головка сдвигается на одну ячейку вправо.

Работа машины начинается при пустых рабочей и выходной лентах. Непустой остается только входная лента, содержащая основные символы языка. Головки всех лент при этом находятся в крайне левом положении.

Машина заканчивает свою работу, если головка входной ленты дошла до пустой ячейки, а рабочая лента в этот момент пуста.

Отличие метода Наура от принципа работы машины Тьюринга заключается в следующем:

1) состояния, которые могут встретиться в процессе грамматического разбора, были явно перечислены, т.е. помещение синтаксических состояний в стэк всегда непосредственно обусловливается прочтением очередного терминала на рабочую ленту;

2) в машине Тьюринга разрешается писать на рабочей ленте за один шаг только один символ, тогда как при грамматическом разборе иногда читается сразу несколько символов (терминалов).

Синтаксические состояния Наур рассматривает как эквивалентные, при этом каждое состояние характеризуется своим номером. Текущий номер должен давать описание ситуации достаточное для того, чтобы полностью определить действия при выборе следующего терминала на входе, включая определение нового текущего состояния.

Каждый шаг анализа состоит в обработке одного терминала. При этом выполняется: проверка, может ли данный терминал начинать предсказанную синтаксическую конструкцию. Это делается путем сравнения номера терминала с рабочим состоянием в стэке, т.е. ищется соответствующая пара «терминал-состояние» в общей таблице (если представлены две таблицы в таблице «начало»). Если ответ отрицательный, продолжение анализа невозможно. Если ответ положительный, то эта конструкция определена и дальше можно делать одно из двух:

а) в предположении, что начатая конструкция продолжается, выбирается любое другое синтаксическое состояние из состояния рабочего терминала, и помещается в верхнюю часть стэка, после чего осуществляется переход к следующему шагу (т.е. к следующему терминалу);

б) в предположении, что начатая конструкция заканчивается этим рабочим терминалом, в общей таблице отыскивается соответствующее состояние и тем самым проверяется, может ли данный терминал заканчивать эту конструкцию. Если нет, продолжение анализа невозможно, если да, то это означает конец конструкции, и соответствующее состояние убирается из стэка. Эту операцию можно повторить любое количество раз в зависимости от начальных вариантов.

Далее следует переход к очередному шагу анализа. Анализ считается законченным, а результат его правильным, если после обработки последнего терминала фразы стэк оказывается простым.

В некоторых вариантах возможна ситуация, когда на некотором шаге не оказывается ни одного допустимого продолжения, а обработка фразы еще не окончена. Это означает, что либо анализируемая фраза синтаксически неправильна, либо, если она правильна, на одном из предыдущих шагов было выбрано «нежелательное» продолжение, что завело в тупик. Таким образом, правильность варианта анализа сигнализируется достижением конца фразы, неправильность — невозможностью дойти до конца фразы.

Все рассмотренные методы грамматического разбора (синтаксического анализа) зависят от конкретных особенностей языков.

Была высказана идея построения универсальных алгоритмов синтаксического анализа и контроля не зависящих от конкретных особенностей языков, а на их основе — построение универсальных так называемых синтаксических трансляторов.

Один из вариантов универсального алгоритма синтаксического анализа и контроля ТС-М (ориентированного на «класс алгоритмических языков, задаваемых рекурсивно полными грамматиками) разработан в Академии наук Молдавской ССР и реализован на ЭВМ «Минск-22».

Упражнения:

1. Организовать грамматический разбор приведенных ниже терминальных цепочек, если грамматики, порождающие языки, заданы схемами правил Р.

а) терминальные цепочки: *AED*, *ADE*,

$$P = \{S \to AB,\ B \to CD,\ B \to DC,\ C \to E\};$$

б) терминальные цепочки: *CCB, CCCCB, CCCCCCB,*

$$P = \{S \to A,\ A \to B,\ A \to CA\};$$

в) терминальные цепочки: *CBC, CCBCC, CCCCBCCCC,*

$$P = \{S \to A,\ A \to B,\ A \to CAC\};$$

г) терминальные цепочки: *xzzy, yyzxx, yx, xzzzzy,*

$$P = \{A \to xB,\ B \to y,\ B \to zB\};$$

д) терминальные цепочки: *xxxxxxy, xxxxy, xxxxxy, xxxz, xxxxxz, xxxxxxz, xxyxy,*

$$P = \{A \to By,\ A \to Cz,\ B \to x,\ B \to Bx,\ C \to x,\ C \to xC\};$$

е) терминальные цепочки: *wxyz, wxyu, wxxyz, xwuy, uxyz,*

$$P = \{C \to Bz,\ C \to wD,\ B \to Ay,\ A \to wx,\ D \to xE,\ E \to yu\};$$

ж) терминальные цепочки: *(xx, (((x, (((x (xx(x(xx, (((x(xx(x(xx((xx((xx, (((xx((xx((xx((xx*

$$P = \{T \to PB,\ P \to (,\ B \to xx,\ B \to xT,\ B \to Tx,\ B \to TT\}.$$

2. Определить, имеют ли смысл терминальные цепочки

$$(((xx\ ((xx\ ((xx\ (x, (xx,\ (((xx\ (x\ (x\ ((x$$

в языке, порождаемом КС-грамматикой со схемой правил вывода

$$P = \{T \to PB,\ P \to (,\ B \to xx,\ B \to xT,\ B \to Tx,\ B \to TT\}.$$

методом Эйкеля, Паула, Бауэра, Замелзона.

3. Используя метод грамматического разбора Эйкеля, Паула, Бауэра и Замелзона, проконтролировать грамматическую правильность следующих терминальных цепочек, определяя способ чтения цепочки в процессе построения алгоритма:

а) *xxxy, xxxz* при

$$P = \{A \to By,\ A \to Cz,\ B \to x,\ B \to Bx,\ C \to x,\ C \to xC\};$$

б) *xxzy, xzzzzy* при

$$P = \{A \to xB,\ B \to y,\ B \to zB\};$$

в) (x·x + x + x)·x·x·(x), x + x + x + x, x·x + (x + (x + x)) ·x + x при

$$P = \{E \to T,\ E \to T + E,\ T \to F,\ T \to F \cdot T,\ F \to x,\ F \to (E)\}.$$

4. Организовать грамматический разбор цепочек любым возможным методом:

а) ⊥<начало> <начало>" ⊥
⊥'x'xx'"⊥ при

Р = {<целая строка> : : = ⊥ <строка> ⊥ <строка> : : = <начало>' <начало> : : ='| <начало> x|<начало> <строка>};

б) x : = x + x·x при

$$P = \{S : : = x : = E, E : : = T|E + T, T : : = F|T \cdot F, F : : = x|(x)\}.$$

в) начало *D; D; D; S* конец,
начало *D; D; D; S; S* конец при
P = {П : : = начало *D*; *S* конец

$$D_1 : : = D\,|\,D_1;\ D$$
$$S_1 : : = S\,|\,S_1;\ S$$

5. Реализовать алгоритм синтаксического контроля следующих предложений:

а) A x B + C;
б) A + B x C — D;
в) A x B + C — D;
г) A x (B + C)/D↑E;
д) A + B — C x D/F↑F;
е) ((A x B + C) x D + E) x F + G;
ж) A + (B+ (C+ (D+ (E + F))));

если схема правил вывода P^2G — грамматики, порождающей приведенные выше предложения, имеет вид:

Р = {<выражение> : : = <терм> | <выражение> +<терм> | <выражение> — <терм>
<терм> : : = <множитель> | <терм> x <множитель>/ <терм>/<множитель>,
<множитель> : : = <простое выражение> | <множитель> ↑<простое выражение>,
<простое выражение> : : = <буква> | (<выражение>),
<буква> : : = A|B|C|D|E|F|G|H|I|J|K|L|M|N|0|P|R|S|T|U|V|W|X|Y|Z}.

2.9. Формальные свойства грамматик

До сих пор мы рассматривали грамматики только с точки зрения содержательной интерпретации. Теперь остановимся на некоторых чисто формальных свойствах грамматик, т.е. на математической теории порождающих грамматик, которая в настоящее время довольно хорошо развита.

В теории грамматик часто возникают вопросы о наличии или отсутствии того или иного алгоритма, т.е. об алгоритмической разрешимости или неразрешимости проблем. Например:

1) существует ли алгоритм, позволяющий по любой данной КС-грамматике узнать, является ли порождаемый ею язык конечным (т.е. конечно ли порождаемое ею множество терминальных цепочек)? Аналогичный вопрос возникает относительно НС-грамматик;

2) существует ли алгоритм, позволяющий узнать относительно любой КС-грамматики, приписывает ли она каждой порождаемой терминальной цепочке только одну синтаксическую структуру, т.е. имеются ли терминальные цепочки, которые могут быть выведены в данной грамматике более чем одним способом?

Подобные вопросы и образуют круг алгоритмических проблем теории грамматик.

Представим некоторые результаты, относящиеся к проблемам разрешения для основных классов грамматик в виде табл. 3, где указано, для каких проблем доказана их разрешимость или неразрешимость.

Буква Р означает, что данная проблема разрешима, т.е. существует алгоритм, дающий ответ на вопрос. Буква Н означает, что проблема неразрешима. Знак ? соответствует тому, что неизвестно, существует ли разрешающий алгоритм.

Как видно из таблицы, в классе КС-грамматик оказываются распознаваемыми некоторые из тех свойств, которые нераспознаваемы в классе НС-грамматик (разумеется, свойства, распознаваемые для НС-грамматик, распознаваемы и для КС-грамматик). К ним относятся:

а) свойство порождать пустой язык, состоящее в том, что существует алгоритм, позволяющий для любой КС-грамматики узнать, порождает ли она хотя бы одну терминальную цепочку (п. 1);

б) свойство порождать конечный язык (п. 2);

в) свойство порождать хотя бы одну цепочку, содержащую вхождение данной цепочки ψ. Это свойство означает, что по КС-грамматике для любого словосочетания можно определить, входит ли оно хотя бы в одну фразу языка. Таким образом, в отличие от НС-грамматик, где распознаваема правильность целых фраз, но не частей фраз, для КС-грамматик распознаваемо и то и другое (п. 10).

Таблица 3

№ п/п	Проблема	Класс грамматик			
		А	КС	НС	О
1	2	3	4	5	6
1	Пуст ли язык, порождаемый данной грамматикой ($L_G = Ø$)?	Р	Р	Н	Н
2	Бесконечен ли язык, порождаемый дайной грамматикой ($L_G = \infty$)?	Р	Р	Н	Н
3	Состоит ли язык, порождаемый данной грамматикой, из всех слов в V_t($L_G = V^*_T$)?	Р	Н	Н	Н
4	Порождают ли две грамматики один и тот же язык ($H = L_{G2}$)?	Р	Н	Н	Н
5	Составляет ли язык, порождаемый одной грамматикой, подмножества языка, порождаемого другой грамматикой ($L_{G1} \subseteq L_{G2}$)?	Р	Н	Н	Н
6	Пусто ли пересечение языков, порождаемых двумя данными грамматиками ($L_{G1} \cap L_{G2} = Ø$)?	Р	Н	Н	Н

7	Является ли пересечение языков, порождаемых данными грамматиками, языком того же класса ($L_{G1} \cap L_{G2}$ — язык того же класса)?	Да	Н	Да	Да
8	Является ли дополнение языка, порождаемого данной грамматикой, языком того же класса ($V^*_T \backslash L_G$ — язык того же класса)?	Да	Н	?	Н
9	Для любых слов φ и ψ выводимо ли ψ из φ в данной грамматике (φ=>ψ)?	Р	Р	Р	Н
10	Для любых φ и ψ выводимо ли слово, содержащее ψ из φ в данной грамматике ($\varphi \overset{*}{\Rightarrow} ? \Rightarrow \psi$)?	Р	Р	Н	Н
11	Есть ли в языке грамматики предложение, выводимое более чем одним способом (*G* — неоднозначна)?	Р	Н	Н	Н
12	Есть ли однозначная грамматика того же класса, порождающая тот же язык?	Да	Н	?	Да

Тем не менее многие важные свойства нераспознаваемы и для КС-грамматик. В частности, нераспознаваемы следующие свойства:

порождать полный язык (содержащий всевозможные цепочки, составленные из терминальных символов, п. 3);

иметь эквивалентную КС-грамматику, приписывающую каждой терминальной цепочке только одну синтаксическую структуру (п. 11);

эквивалентность произвольных двух грамматик, т. е. не существует алгоритма, позволяющего по любой паре КС-грамматик узнать, являются ли они эквивалентными (п. 12).

В классе *A*-грамматик распознаваемы все свойства, перечисленные в таблице.

Из проблем, относящихся к алгоритмам не для классов грамматик, а для конкретных фиксированных грамматик, мы рассмотрим только одну: так называемую проблему распознавания замещаемости. Она состоит в том, чтобы для данной грамматики *G* найти алгоритм, позволяющий по любой паре цепочек φ, ψ узнать, замещена ли φ на φ в языке L_G. Для некоторых грамматик такие алгоритмы существуют, в частности, для всех *A*-грамматик. Однако имеются примеры КС-грамматик, для которых подобного алгоритма нет.

Кроме алгоритмических проблем к математической теории грамматик относятся также проблемы оценки сложности вывода в грамматиках.

Сложность вывода естественно измерять либо числом шагов, т.е. числом промежуточных цепочек, либо необходимым объемом «памяти». При этом под объемом памяти может пониматься, например, длина промежуточных цепочек, число вспомогательных символов в промежуточных цепочках, расстояние от первого вспомогательного символа до конца цепочки и т. д.

Для оценки сложности по числу шагов предложена мера

$$\tau_G(n) = \max_{l(\omega) \leq n} \tau_G, \ \omega,$$

где *G* —данная грамматика;

ω — произвольная цепочка, выводимая в *G*;

l(ω) — длина цепочки ω (число символов в ω);

$\tau_G\omega$ — число шагов («время») самого короткого вывода цепочки ω в *G*;

n — произвольное натуральное число.

Чтобы найти $\tau_G(n)$, нужно, как видно из формулы, найти для каждой выводимой цепочки, по длине, не превосходящей *n*, наименьшее число шагов — «время» вывода этой цепочки в *G* (цепочка ω может иметь в *G* много выводов разной длины). Затем среди всех этих «времен» берется максимальное. Это и будет значением функции $\tau_G(n)$— так называемой *временной сигнализирующей функции*.

Другими словами, $\tau_G(n)$ — это такое число шагов, которое, с одной стороны, заведомо достаточно для вывода любой цепочки не длиннее *n*, а с другой стороны, необходимо. Условие

необходимости вытекает из того, что среди выводимых цепочек не длиннее n имеется хотя бы одна цепочка, которую нельзя вывести меньше чем за $\tau_G(n)$ шагов.

Исходя из понятия временной сигнализирующей функции, можно получать оценки сложности вывода в разных грамматиках. Так, для любой неукорачивающей грамматики G имеет место неравенство

$$\tau_G(n) \leqslant P^{n+1},$$

где P — общее число терминальных и нетерминальных символов. Очевидно, что это неравенство имеет силу и для НС-грамматики и для КС-грамматики.

Однако для КС-грамматик указанную оценку можно значительно улучшить, т.е. понизить. Для любой КС-грамматики G_1,выполняется неравенство

$$\tau_{G1}(n) \leq 2ln,$$

где l — число нетерминальных символов в G_1.

Эта оценка получается из следующих несложных соображений: любой КС-вывод можно провести так, чтобы на каждом шаге заменялся самый левый вспомогательный (нетерминальный) символ. Таким образом, на каждом шаге длина цепочки либо не изменяется, либо увеличивается, причем возможны шаги трех типов:

1. $A \to B$ («неудлиняющий нетерминальный»), где A и B нетерминальные символы;
2. $A \to a$ («неудлиняющий терминальный), где A — нетерминальный символ, a — терминальный символ;
3. $A \to d_1 d_2 \ldots d_m$ («удлиняющий»), при $m > 1$.

Если вывод не содержит «петель» (повторений промежуточных цепочек), то в нем нигде не может быть больше чем с «неудлиняющих нетерминальных» шагов подряд. В самом деле, если бы было произведено подряд K таких шагов, где $K \geq c$, то соответствующие цепочки вывода имели бы вид

Исходная цепочка xA_0Y.
1-й шаг xA_1Y;
2-й шаг xA_2Y;
c-й шаг xA_cY;
K-й шаг xA_kY.

Здесь x — цепочка терминальных или основных символов, $A_0, A_1, \ldots, A_k$ — вспомогательные символы, Y — цепочка произвольного вида. Уже с шагов дают $(c+1)$ цепочек. Поскольку эти цепочки различаются только символами $A_0, A_1, \ldots, A_c$, а среди этих символов не более с различных, две из этих цепочек обязательно совпадут, т.е. получится «петля». Таким образом, в любом КС-выводе без «петель», а тем более в кратчайшем выводе не бывает более $c-1$ неудлиняющих нетерминальных шагов подряд. Между двумя сериями таких шагов обязательно должен вклиниться хотя бы один шаг типа 2 или типа 3, а таких шагов всего не более $2n$ (не более n терминальных, поскольку вся цепочка состоит из n терминальных символов, и не более n «удлиняющих», поскольку каждый удлиняющий шаг увеличивает длину цепочки, а увеличиться более чем на n эта длина не может). Стало быть, мы имеем не больше чем $2n$ серий шагов типа 1 (не более чем по $c-1$ шагов в серии) и не более чем $2n$ шагов типа 2 и 3. Общее число шагов не превосходит $2n(c-1) + 2n = 2cn$, что и требовалось доказать.

Мы рассмотрели примеры формул верхних оценок временной сигнализирующей функции. Процесс получения нижних оценок оказывается более сложным, и пока известна одна такая оценка для частного вида НС-грамматики, которую мы приведем без доказательства.

Язык, состоящий из всевозможных цепочек вида $\omega q \omega^1$, не может быть порожден никакой такой НС-грамматикой G, у которой временная сигнализирующая функция $\tau_G(n)$ по порядку меньше (~ много меньше), чем n^2 (ω и ω^1 — цепочки, элементы которых определенным образом согласованы между собой).

Аналогичным образом может быть введена и *емкостная сигнализирующая функция* $\sigma_G(n)$, характеризующая необходимый для вывода объем памяти. Именно

$$\sigma_G(n) = \max_{l(\omega) \leq n} \sigma_G \omega,$$

где G, ω, n имеют тот же смысл, что и в определении временной сигнализирующей функции, а $\sigma_G\omega$ — есть емкость («объема памяти») наименее емкого вывода цепочки ω в G. Емкостью вывода называем длину самой длинной цепочки, входящей в этот вывод.

Упражнения:

1. Определить емкостную и временную сигнализирующие функции для грамматик *G1, G2, G3*, заданных правилами:

$$P_1 = \{A \to aB,\ A \to BA,\ B \to ab,\ B \to Ab,\ A \to a\};$$
$$P_2 = \{A \to aB,\ A \to CB,\ C \to ab,\ B \to Ca,\ B \to b\};$$
$$P_3 = \{A \to BC,\ B \to a,\ C \to bb,\ C \to bA,\ C \to Ab,\ C \to AA\},$$

где A — цель грамматики. Длина терминальных цепочек $n \leq 5$.

2. Для грамматики с правилами $P = \{A \to BC,\ A \to C,\ xA \to B,\ B \to y,\ B \to Ay, B \to Cx,\ C \to xx,\ C \to AAx,\ C \to xAA\}$ определить сложность вывода терминальных цепочек длиной 3, 4 и 10 символов.

3. Для грамматики $G = (\{A\}, \{S\}, P, S)$ с правилами вида $P = \{S \to SS,\ S \to SaS,\ S \to a^3\}$ определить сложность вывода терминальных цепочек, длина которых не превышает 15 знаков.

4. Рассматривая терминальные цепочки с длиной $n \leq 6$, определить временную и емкостную сигнализирующие функции для грамматик *G1, G2, G3, G4*, заданных правилами:

$$P_1 = \{S \to ASBC,\ S \to C,\ C \to BD,\ D \to BD,\ D \to A\};$$
$$P_2 = \{S \to A^2SA,\ S \to ASA,\ S \to AS,\ S \to \Lambda\};$$
$$P_3 = \{S \to ATC,\ S \to BS,\ T \to AT,\ T \to TA,\ T \to C\};$$
$$P_4 = \{S \to AS,\ S \to SA,\ S \to BTB,\ T \to ATB,\ T \to ACB\}.$$

5. Определить временную и емкостную сигнализирующие функции (по длине вывода и по длине промежуточной цепочки) для заданных терминальных цепочек с длиной $n \leq 7$.

$$S \to Sa \to Sab \to aab;$$
$$S \to SS \to SSa \to Sab \to aab;$$
$$S \to Sab \to Saab \to aaSaab \to aabSaab \to aabaaab;$$
$$S \to abS \to abSb \to abSab \to abSaab \to abSaaab \to abbaaab;$$
$$S \to SS \to Sab \to Saab \to Saaab \to aSaaab \to aaSaaab \to aabaaab.$$

6. Дана КС-грамматика $G = (V_T, V_H, P, S)$ с правилами

$$P = \{S \to AC, S \to SB,\ S \to B,\ C \to SEC,\ C \to BC,\ C \to A\}.$$

Определить сложность вывода терминальных цепочек, длина которых не превышает 10.

2.10. Абстрактные автоматы и их связь с языками и грамматиками

2.10.1. Основные понятия теории автоматов

Рассмотрим другой метод определения языка, основанный на использовании множества строк приемлемых некоторым распознающим устройствам, автоматам.

Все автоматы, которыми мы будем пользоваться, суть частные случаи машины Тьюринга с входной лентой, которая может быть описана следующим образом. Машина имеет две ленты — входную и рабочую. Каждой ленте сопоставлены алфавит с выделенным в нем пустым символом и головка, также называемые соответственно входными и выходными.

Каждая лента ограничена слева, и крайняя левая ячейка всегда занята символом, не входящим в соответствующий алфавит.

Машина имеет конечное число внутренних состояний, среди которых выделены начальное и заключительное.

Элементарный акт машины представляет собой либо чтение символа на входной ленте с одновременным сдвигом входной головки на одну ячейку вправо и переходом машины в другое состояние, либо одно из следующих действий на рабочей ленте:

а) замена обозреваемого символа другим с одновременным сдвигом головки вправо и переходом машины в другое состояние;

б) то же, но со сдвигом влево;

в) то же, но без сдвига.

Когда машина находится в заключительном состоянии, никакой элементарный акт невозможен.

Будем говорить, что машина находится в начальной ситуации (конфигурации), если на ее входной ленте непосредственно за ограничивающим символом записана цепочка, не содержащая пустого символа, рабочая лента пуста (кроме ограничивающей ячейки), обе головки обозревают ограничивающие символы и машина находится в начальном состоянии.

Машина находится в заключительной ситуации, если на входной ленте обозревается первая слева пустая ячейка, рабочая лента пуста, машина находится в заключительном состоянии.

Машина допускает цепочку со, если начав работу в начальной ситуации с цепочкой со на входной ленте через конечное число шагов, она может оказаться в заключительной ситуации.

Языком, допускаемым машиной M, называется множество допускаемых его цепочек. Автомат в общей форме представлен на рис. 22.

Он содержит входную ленту с отметками концов, на которой записана строка. Входная головка считывает по одному входному символу. Имеется конечное управление, которое может находиться в конечном числе состояний. Имеется конечная память, возможно, с некоторой структурой. Автомат делает элементарные шаги, зависящие от состояния конечного управления, символа, считываемого входной головкой, и конечного объема информации из бесконечной памяти. За один шаг он может сделать следующее:

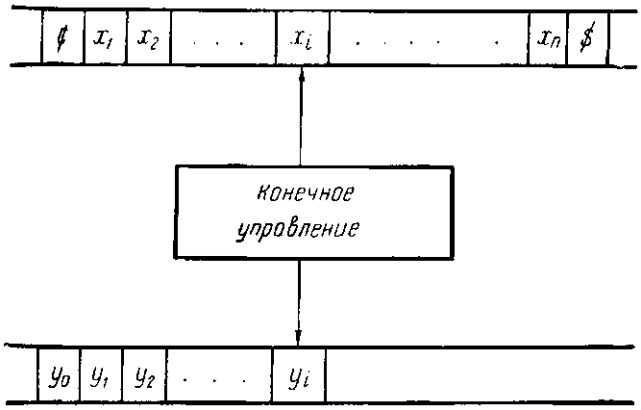


Рис. 22

1) изменить состояние конечного управления;
2) передвинуть входную головку на одну клетку в любом направлении или оставить ее на месте;
3) изменить бесконечную память.

Автомат может иметь не более одного допустимого элементарного шага в каждой ситуации, тогда он является детерминированным. Он может иметь в некоторых ситуациях произвольное конечное множество допустимых шагов, тогда он является недетерминированным. Если нет ограничений на движение входной головки, то автомат называется двусторонним. Если головке позволяется передвигаться только в одном направлении, то автомат будет односторонним.

В начале конечное управление автомата находится в некотором начальном состоянии при некотором начальном содержании бесконечной памяти и с входной головкой у метки левого конца.

Условимся знаками $\overline{C}$ и $\overline{S}$ обозначать левую и правую метки концов соответственно и будем предполагать, что они не содержатся ни в каком алфавите, если это не оговорено специально.

Множество автоматов с «одинаковой» структурой бесконечной памяти образует семейство автоматов. Семейство имеет четыре класса автоматов: односторонних детерминированных, односторонних недетерминированных, двусторонних детерминированных и двусторонних недетерминированных. Сокращенно эти классы обозначаются соответственно *1D, 1N, 2D, 2N*.

По способу функционирования различают два вида автоматов: синхронные и асинхронные. В синхронных автоматах переходы из одних состояний в другие осуществляются через равные временные промежутки, в то время как в асинхронных автоматах эти переходы совершаются через неравные между собой промежутки времени.

Остановимся более подробно на законах функционирования автоматов. Состояние *s(t)* автомата в момент времени *t* однозначно определяется предыдущим состоянием *s(t—1)* и входным сигналом *x(t)*. Поэтому можно записать

$$s(t) = \varphi(s(t-1),\ x(t)),$$

где φ — функция, определяющая последующее состояние автомата, которая обозначается просто φ(*s, x*) и называется функцией переходов. Выходной сигнал *y(t)* реального автомата всегда появляется после входного сигнала *x(t)*. Однако относительно момента времени *t* перехода автомата из состояния *s(t—1)* в состояние *s(t)* выходной сигнал *y(t)* может появиться либо раньше, либо позже этого момента времени. Поэтому справедливы выражения

$$(*)\quad y(t) = \psi(s(t-1),\ x(t));$$

$$(**)\quad y(t) = \psi(s(t),\ x(t)),$$

где ψ — функция выходов обычная (*) или сдвинутая (**). Она однозначно определяет выходную букву автомата в зависимости от состояния *s(t—1)* в предыдущий момент времени и входного сигнала *x(t)*, если это обычная функция выходов, а в случае сдвинутой — от состояния *s(t)*, в которое автомат переходит в текущий момент времени, и входного сигнала *x(t)*.

Учитывая работу реальных автоматов, различают два рода абстрактных автоматов. Автоматами первого рода называют автоматы, описываемые следующими уравнениями:

$$s(t) = \varphi(s(t-1),\ x(t));$$

$$y(t) = \psi(s(t-1),\ x(t)),$$

которые определяют закон функционирования этих автоматов, а автоматами второго рода называют автоматы, закон функционирования которых задается уравнениями.

$$s(t) = \varphi(s(t-1),\ x(t));$$

$$y(t) = \psi(s(t),\ x(t)),$$

где *t = 1, 2, ...* .

Абстрактные автоматы любого рода называются правильными, если выходной сигнал *y(t)* не зависит явно от входного сигнала *x(t)*, а определяется лишь состоянием *s(t—1)* или *s(t)*.

Существует несколько эквивалентных способов задания абстрактных автоматов, среди которых по аналогии с графами можно назвать три: аналитический, геометрический и матричный.

Говорят, что задан абстрактный автомат *A*, если задана совокупность пяти объектов: конечного множества $X=\{x_i\}$, $i \in I = \{1, 2, ..., m\}$, называемого входным алфавитом автомата,

конечного множества $Y=\{y_i\}$, $j \in I=\{1, 2, ..., n\}$, называемого выходным алфавитом автомата, произвольного множества S, называемого алфавитом состояний, элемента $s_0 \in S$, называемого начальным состоянием автомата, и отображения δ множества S в себя, которое любому $s \in S$ и каждой входной букве $x \in X$ сопоставляет состояние $S_k \in S$, определяющее функцию переходов $\varphi(s, x)$, и выходную букву $y \in Y$, определяющую обычную или сдвинутую функцию выходов $\psi(s, x)$. Следовательно, запись $A = \{X, S, Y, s_0, F\}$ обозначает произвольный абстрактный автомат. Если A — автомат первого рода, то $\psi(s, x)$— обычная функция выходов, если же A — автомат второго рода, то $\psi(s, x)$ — сдвинутая функция выходов.

Если множество S конечно, то автомат A называется конечным автоматом, в противном случае A — бесконечный автомат. Все реальные автоматы являются конечными.

Покажем теперь, каким образом определить отображение, индуцируемое заданным конечным автоматом A. Как известно, автомат функционирует в дискретном времени $t = 0, 1, 2, ...$. Предполагается, что в начальный момент времени $t = 0$ автомат всегда находится в начальном состоянии $s_0 \in S$. В каждый момент времени, отличный от начального, на вход автомата подается входной сигнал $x(t)$ — произвольная буква входного алфавита X, а на выходе возникает некоторый выходной сигнал $y(t)$— буква выходного алфавита Y. Пусть $G(X)$ и $G(Y)$ — соответственно множества входных и выходных слов автомата A и $p = x_{i1}x_{i2} ... x_{ik}$ — произвольное входное слово, т.е. $p \in G(X)$. Подача на вход автомата A, установленного в начальное состояние, конечной последовательности $x(1) = x_{i1}, x(2) = x_{i2}, ..., x(k) = X_{ik}$ на основании известного отображения δ вызывает появление однозначно определенной конечной последовательности $y(1)=y_{j1}, y(2) = y_{j2}, ..., y(k) = y_{jk}$ на выходе автомата, которая соответствует выходному слову $r = y_{j1}y_{j2}... y_{jk}$ из $G(Y)$. Поэтому $r = \delta(p)$. Относя каждому входному слову $p \in G(X)$ соответствующее ему входное слово $r \in G(Y)$, получим искомое отображение δ, которое и является отображением, индицируемым конечным автоматом A.

Очевидно, что отображение δ множества S в S однозначно задает функции $\varphi(s_1x)$ и $\psi(s_1x)$, определяющие закон функционирования автомата A, и наоборот.

Опишем классы автоматов, которые связаны с типами грамматик в иерархии Хомского.

Машина Тьюринга с двумя лентами, изображенная на рис. 23, представляет собой устройство, бесконечной памятью которого является линейная лента клеток со считывающей и записывающей ленточной головкой. За один шаг, зависящий от ее состояния и символов, считываемых входной и ленточной головками, машина Тьюринга может:

1) изменить состояние;
2) изменить символ считываемой ленточной головкой;
3) передвинуть входную и ленточные головки на одну клетку независимо в любом направлении.

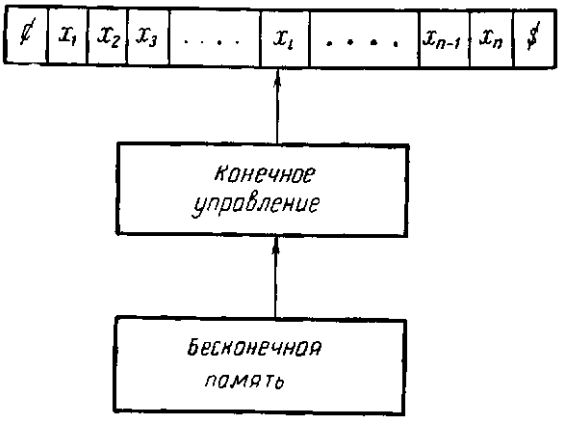


Рис. 23

Формально двусторонняя недетерминированная машина Тьюринга (или просто машина Тьюринга) обозначается так:

$$M=(S, X, Y, \delta, s_0, B, F),$$

где S, X, Y и F — конечное множество состояний, входных символов, ленточных символов и заключительных состояний соответственно: $F \subseteq S$, $s_0 \in S$ — начальное состояние; $B \in F$ — пустой символ; δ — это отображение $S \times (X, U \{\overline{C}, \overline{S}\}) \times Y$ в классе подмножеств множества

Sx(Y — {B}x{—1, 0, +1}x{—1, 0, +1}. Функция ? осуществляет отображение вида $C \times X \times Y \to S \times Y \times d_1 \times d_2$. Это означает, что машина Тьюринга *M* находится в состоянии *s*, считывает *x* на входе и *y* на ленте, то она имеет право перейти в состояние s_i напечатать непустой символ y_i вместо *y* и передвинуть свои входную и ленточные головки в направлениях d_1 и d_2 соответственно (—1 обозначает сдвиг влево, 0 — отсутствие движений, +1 — сдвиг вправо).

Конфигурация машины Тьюринга *M* будет обозначаться $(s, x_1, x_2, ..., \overset{\frown}{x}_i, ..., x_n, y_1, y_2, ..., \overset{\frown}{y}_j, ..., y_m)$. Здесь *s* — состояние $X_1 = \overline{C}$, $x_n = \overline{S}$, $x_2, x_3, ..., x_{n-1}$ — вход, где каждое x_k, $2 \le k \le n-1$ принадлежит *X*. Символы y_1 и y_m пустые, а $y_2, y_3, ..., y_{m-1}$ — непустая часть ленты машины *M*. Входная и ленточная головки считывают x_i и y_j соответственно.

Если $M = (S, X, Y, \delta, s_0, B, F)$ и для любых *s* из *S*, *x*, из $X \cup \{\overline{C}, \overline{S}\}$ и *y* из *Y* множество $\delta(s, x, y)$ содержит не более одного элемента, то *M* является детерминированной. Если ни для каких *s*, *x*, *y* множество $\delta(s, x, y)$ не содержит элементов вида (*p, Y, —1, d*), то *M* является односторонней. Таким образом, определены четыре класса машин Тьюринга: *1D, 1N, 2D, 2N*.

Два класса автоматов называются эквивалентными, если они распознают одинаковые языки.

Теорема (33). Четыре класса машин Тьюринга (*1D, 1N, 2D, 2N*) попарно эквивалентны, и каждый точно распознает все языки типа 0.

2.10.2. Линейноограниченные автоматы

1. Машина Тьюринга с входной лентой называется линейноограниченной, если существует такое число c, что в процессе «допускания» цепочки ω машина не может использовать более чем $c \cdot |\omega|$ рабочих ячеек.

Таким образом, объем памяти такого автомата не является неограниченным, как в машинах Тьюринга, а определяется длиной последовательности входных символов. Линейная зависимость между объемом памяти и длиной входной последовательности определила название этого класса автоматов — линейноограниченные автоматы.

Линейноограниченный автомат представляет собой двусторонний автомат, управляющее устройство которого имеет конечное число состояний. Входная лента его может перемещаться в обе стороны, причем головка управляющего устройства может не только считывать входные символы с ленты, но и записывать на ней новые символы.

Детерминированным линейноограниченный автоматом с одной лентой называется упорядоченная шестерка

$$A = (X, S, s_0\ \delta, F, d),$$

где X, S, s_0, F имеют тот же смысл, что и в универсальном автомате, $d = \{—1, 0, +1\}$, как и в двустороннем автомате, а отображение δ имеет вид: $S \times X \rightarrow S \times X \times d$. Этот автомат работает следующим образом. В начальный момент на входной ленте записана последовательность входных символов, так что левее крайнего левого символа последовательности и правее крайнего правого символа лента пуста, т.е. клетки заполнены символом Λ. Автомат находится в состоянии s_0 и считывает крайний левый символ входной последовательности, записанный на ленте.

Далее, если управляющее устройство автомата находится в состоянии s_i и считывает с ленты символ x_j и если имеет место $\delta(s_i, x_j) = (s_t, x_k, d_q)$, то автомат вместо x_j печатает на ленте символ x_k, переходит в состояние s_t и лента продвигается на одну клетку вправо при $d_q = —1$, влево — при $d_q = +1$ и остается на месте — при $d_q = 0$.

Условие допустимости или приемлемости входной последовательности для детерминированного линейноограниченного автомата те же, что и для двустороннего. Последовательность считается приемлемой, если считывающая головка сходит с правого конца заполненной части ленты и при этом управляющее устройство находится в одном из состояний множества F.

Таким образом, в линейноограниченных автоматах входная лента используется также и в качестве ленты памяти.

Для детерминированных линейноограниченных автоматов доказана следующая теорема.

Теорема 34 (Ландвебера). Множество цепочек, представимых линейноограниченным автоматом, есть язык типа 1, т. е. контекстно связанный.

Эта теорема, однако, не утверждает, что для любого языка типа 1 существует детерминированный линейноограниченный автомат, который представляет этот язык. Вопрос о том, существует ли язык типа 1, не представимый детерминированным линейноограниченным автоматом, остается пока открытым. Доказана лишь подобная теорема для бесконтекстных языков.

Теорема 35 (Курода). Любой язык типа 2 представим в детерминированном линейноограниченном автомате.

Недетерминированным линейноограниченным автоматом с одной лентой называется линейноограниченный автомат с одной ленной, функция отображения которого неоднозначна, а начальным состоянием может быть любое из состояний множества S_0. Для недетерминированного линейноограниченного автомата входная последовательность считается приемлемой, если существует такой вариант работы автомата, что автомат сходит с правого конца заполненной ленты в одном из состояний множества F.

Область представимости недетерминированных линейноограниченных автоматов определяется следующей теоремой.

Теорема 36 (Курода). Множество входных последовательностей является языком типа 1 тогда и только тогда, когда оно является множеством приемлемых последовательностей

какого-либо недетерминированного линейноограниченного автомата, т.е. когда это множество представимо таким автоматом.

Из этой теоремы и из того факта, что детерминированный линейноограниченный автомат заведомо не может делать больше, чем недетерминированный автомат такого же типа, следует, что класс линейноограниченных автоматов обоих этих типов заведомо `уже, чем класс машин Тьюринга, в том смысле, что область представимости этих автоматов является точным включением области представимости машин Тьюринга. В то же время класс линейноограниченных автоматов заведомо шире, чем класс конечных автоматов, которые, как мы увидим далее, представляют только языки типа 3. Следовательно, линейноограниченные автоматы являются промежуточными между конечными автоматами и машинами Тьюринга и притом адекватными определенному классу языков типа 1 — контекстным языкам.

Теорема (37). Классы *2N* и *1N* линейноограниченных автоматов эквивалентны, и каждый распознает язык *L* тогда и только тогда, когда *L— {l}* является контекстным.

Теорема (38). Классы *2D* и *1D* линейноограниченных автоматов эквивалентны.

2.10.3. Автоматы с магазинной памятью

У автоматов с магазинной памятью, кроме бесконечной вправо входной ленты, есть специальная вспомогательная лента, также бесконечная вправо. При движении вспомогательной ленты влево на ней записываются вспомогательные символы, а при движении этой ленты вправо эти символы считываются и стираются. Первым всегда считывается последний записанный символ. В таком автомате невозможно считать символ с ленты памяти, не стерев его.

Таким образом, в автоматах с магазинной памятью ограничение накладывается не на объем памяти, а на доступ к информации, записанной на ленте памяти — в «магазине».

Автомат с магазинной памятью определяется как упорядоченная восьмерка

$$A = (X,\ S,\ s_0,\ Z,\ \sigma,\ \delta,\ F,\ d),$$

где X, S, s_0, F — соответственно конечные непустые множества входных символов и состояний управляющего устройства, начальное состояние и выделенное подмножество состояний, Z — конечное множество вспомогательных магазинных символов σ — специально выделенный символ «стирание» ($\sigma \in Z$), $d=\{-1,0,+1\}$, δ-функция, осуществляющая отображение множества $S \times (X \cup \Lambda) \times Z$ в множестве $S \times Z^* \times d$, если автомат детерминированный, или в множество конечных подмножеств множества $S \times Z^* \times d$, если недетерминированный.

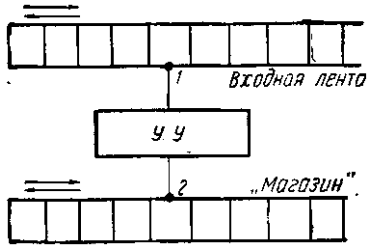


Рис. 24

Детерминированный автомат с магазинной памятью работает следующим образом (рис. 24).

В начальный момент на входной ленте записана цепочка u и считывающая головка 1 воспринимает крайний левый символ этой цепочки. Управляющее устройство находится в состоянии s_0, лента памяти пуста и головка 2 расположена против крайней левой ее клетки. Если в некоторый момент времени имеет место $\delta(s_i, x_t, z_i) = (s_j, v_1, d_q)$, то это означает, что автомат, находящийся в состоянии s_i и воспринимающий символ xt на ленте памяти, переходит в состояние s_j, а на ленте магазинной памяти печатается цепочка v, первый символ которой печатается в той клетке, где было записано z_l, и лента сдвигается на $|v| - 1$ клеток влево ($|v|$ — длина цепочки v). После этого входная лента сдвигается на одну клетку влево, если $d_q = +1$, на одну клетку вправо, если $d_q = -1$, или остается на месте, если $d_q = 0$.

Если цепочка $v = \Lambda$, т.е. пустая, то лента памяти не движется и содержимое ее не меняется. Если же $v = \sigma$ — стирающий символ, то символ zi на ленте памяти, против которого стояла головка 2, стирается и лента памяти сдвигается на одну клетку вправо.

Входная последовательность считается приемлемой, если после того, как головка 1 считывает последний символ входной последовательности, управляющее устройство переходит в одно из состояний выделенного подмножества F состояний *.

Для теории автоматов с магазинной памятью основной является следующая теорема.

Теорема 39 (Хомского). Язык L является языком типа 2 тогда и только тогда, когда он представим в некотором недетерминированном автомате с магазинной памятью.

Таким образом, область представимости для класса недетерминированных автоматов с магазинной памятью совпадает с классом языков типа 2, т.е. бесконтекстных языков.

Множество, представимое в детерминированном автомате с магазинной памятью, носит название детерминированного бесконтекстного языка, т.е. языка типа *2D*. Имеет место следующая теорема.

Теорема 40 (Хейнс-Шютценберже). Если A — детерминированный автомат с магазинной памятью, то множество $L(A)$ последовательностей, которые в нем представимы, являются бесконтекстным языком без неоднозначности.

Однако обратное неверно, так как существуют языки типа 2 без неоднозначности, не являющиеся детерминированными.

2.10.4. Конечные автоматы

Наиболее простой моделью является автомат с конечным числом состояний (с конечной памятью) — конечный автомат.

Детерминированным конечным автоматом называется упорядоченная система из пяти символов — «упорядоченная пятерка».

$$A = (X,\ S,\ s_0,\ \delta,\ F),$$

где $X = \{x_1, x_2 \ldots, x_r\}$ — множество входных символов (входной алфавит), $S = \{s_1, s_2, \ldots, s_n\}$ — множество внутренних состояний, r, n конечны, $s_0 \in S$ — начальное состояние, δ — функция, отображающая $S \times X$ в S, что обычно записывается $\delta: S \times X \to S$. Эта функция однозначно ставит в соответствие паре символов (s_i, x_j) некоторый символ $s_k \in S$, $F \subseteq S$ - некоторое выделенное подмножество состояний автомата (заключительные состояния).

Этот автомат можно интерпретировать так, как это показано на рис. 25.

Автомат состоит из управляющего устройства, считывающей головки и бесконечной вправо входной ленты, разделенной на клетки. Вначале на ленте записана входная цепочка так, что в каждой клетке ленты содержится по одному символу цепочки. Начальный символ записан в крайней левой клетке, а все клетки той части ленты, которая расположена правее последнего символа записи входной цепочки, пусты, т.е. в каждой из этих клеток записан «пустой» символ Λ. Головка в начальный момент расположена против крайней левой клетки ленты, а управляющее устройство находится в начальном состоянии.

В каждый очередной такт головка воспринимает символ на ленте, управляющее устройство

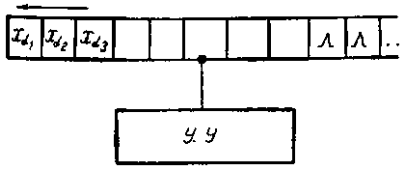


рис 25

изменяет свое состояние в соответствии с отображением δ, а лента сдвигается на одну клетку влево. Если головка воспринимает пустой символ Λ, который записан правее последнего символа входной последовательности, то это будет означать прекращение работы автомата, т.е. состояние управляющего устройства больше не изменяется.

Допустимой, или приемлемой, называется входная последовательность, обладающая следующим свойством: когда головка воспринимает последний символ этой входной последовательности, автомат переходит в одно из состояний множества F.

Множество допустимых входных последовательностей автомата A, или множество последовательностей, представимых в автомате A множеством состояний F, обозначается $L(A)$. Такое множество, являясь множеством цепочек символов из конечного алфавита X, будет с точки зрения теории языков некоторым языкам над терминальным словарем $V^T = X$. Таким образом, естественно возникает способ сопоставления автоматов, языков и грамматик.

Для конечных автоматов множество $L(A)$ выделяется известной теоремой Клини.

Теорема 41 (Клини). Для любого конечного автомата A множество $L(A)$ допустимых последовательностей является регулярным, т.е. языком типа 3. Эта теорема объясняет, почему языки типа 3 называются языками с конечным числом состояний.

Детерминированным конечным автоматом с двумя лентами называется упорядоченная шестерка

$$A = (X,\ Y,\ S,\ s_0,\ \delta,\ \lambda),$$

где X, S, s_0 и δ имеют тот же смысл, что и для конечного автомата с одной лентой, $Y = \{y_1, y_2, \ldots, y_l\}$ множество выходных символов (выходной алфавит), l — конечно, а λ — функция, осуществляющая отображение $S \times X$ в Y, т.е. $\lambda : S \times X \to Y$.

Интерпретация этого автомата представлена на рис. 26.

Кроме входной ленты автомат имеет бесконечную вправо выходную ленту, которая может перемещаться только в одну сторону — справа налево. Каждый очередной такт в клетке ленты

---

* Определение приемлемости в различных работах трактуется по-разному.

печатается символ, и лента сдвигается на одну клетку.

Этот автомат называется последовательной машиной, или автоматом Мили. Автомат Мили *M* каждой цепочке входных символов *u* однозначно ставит в соответствие цепочку выходных символов ω, что записывается *M(u) = ω*. Очевидно, что *M(Λ) = Λ*.

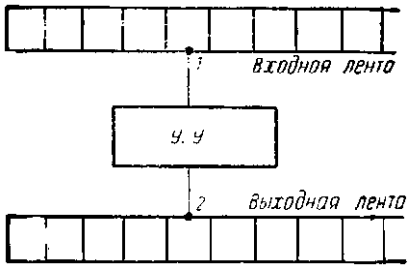


Рис 26

Недетерминированным конечным автоматом с одной лентой называется упорядоченная пятерка

$$A=(X, S, S_0, \delta, F),$$

где *X, S, F* имеют тот же смысл, что и в определении детерминированного автомата, δ — отображение множества *S*x*X* в множество всех подмножеств состояний *S*, а S_0 — некоторое выделенное (начальное) подмножество множества *S*.

Представление множества входных последовательностей в недетерминированном автомате понимается следующим образом. Множество входных последовательностей *U* допустимо в недетерминированном конечном автомате, если для каждой последовательности этого множества найдутся два таких состояния $s_i \in S_0$ и $s_i \in F$ и такой вариант работы (т.е. такие конкретные значения функции δ), что последовательность *U* переводит автомат из состояния s_i в состояние s_j.

Доказана следующая теорема.

Теорема 42 (Рабина и Скотта). В недетерминированных одноленточных конечных автоматах, так же как и в детерминированных, допустимы регулярные множества цепочек и только они. С помощью недетерминированных автоматов нельзя расширить класс представимых множеств. Однако недетерминированный конечный автомат обычно имеет меньшее число состояний по сравнению с детерминированным автоматом, представляющим то же множество.

Конечный автомат с двумя лентами является недетерминированным, если при одной и той же ситуации, характеризуемой парой (s_i, x_j), возможно конечное число вариантов его действия. Следовательно, такой автомат может одной входной цепочке поставить в соответствие конечное множество выходных цепочек.

Двусторонний конечный автомат отличается от обычного детерминированного автомата тем, что входная лента бесконечная в обе стороны и может двигаться не только влево, но и вправо, а также может оставаться неподвижной. При этом отображающая функция δ:*S*x*X*→*S* заменяется функцией δ:*S*x*X*→*S*x*d*, где *d={—1, 0, +1}*. Символ *d* показывает, в какую сторону должна сдвинуться входная лента: -1 соответствует сдвигу на одну клетку вправо, +1 — сдвигу влево на одну клетку, 0 — лента остается неподвижной.

Множество *L(A)* всех допустимых последовательностей такого автомата определяется следующей теоремой.

Теорема 43 (Рабина, Скотта, Шепердсона). Для каждого двустороннего автомата *A* можно эффективно определить такой автомат *B,* который имеет то же множество приемлемых входных последовательностей, что и автомат *A*. Поэтому множество *L(A)* регулярно, т.е. является языком типа 3.

Таким образом, двусторонний автомат не расширяет возможности конечных автоматов. Соответствие между основными видами автоматов и представимыми в них языками сведены в табл. 4.

Таблица 4

№ п/п	Наименование автомата	Тип предста-вимого языка
1	Детерминированный и недетерминированный конечный автомат	↔ 3
2	Детерминированный автомат с магазинной памятью	↔ 2
3	Недетерминированный автомат с магазинной памятью	↔ 2
4	Детерминированный линейноограниченный автомат	→ 2 ← 1
5	Недетерминированный линейноограниченный автомат	1↔
6	Детерминированная и недетерминированная машина Тьюринга	0↔

В этой таблице использованы следующие обозначения. Стрелка ↔ означает, что язык тогда и только тогда типа *i (i=0, 1, 2, 2^D, 3)*, когда он представим в автомате типа A_k *(k = **1**, 2, ..., 6)*. Стрелка → означает, что если язык представим в автомате A_k, то он является языком типа ***i***. Стрелка ← означает, что если язык является языком типа *i*, то найдется автомат типа A_k, в котором он представим.

Таблица 5

Обозначение	Общая теория языков	Математическая лингвистика	Программирование	Математические модели
V_T	Терминальный словарь (алфавит)	Основной словарь языка	Выходные символы	Выходной алфавит
V_H	Нетерминальный словарь (алфавит)	Вспомогательные грамматические термины	Набор команд	Множество состояний управляющеего устройства
S	Начальный нетерминальный символ	Предложения	Программа	Начальное состояние
P	Система продукций	Синтаксические правила	Операции	Отображающая функция

В заключение рассмотрим возможную трактовку понятий и результатов общей теории языков применительно к трем основным областям, где эти результаты используются: математической лингвистике, языкам программирования и теории автоматов.

В табл. 5 собраны аналогичные понятия из этих трех областей и указаны соответствующие им понятия общей теории языков.

Упражнения:

1. По каждому из приводимых ниже конечных автоматов *A* построить праволинейную КС-грамматику, порождающую множество *T(A)*.

а) $A = (S, X, p, \delta, \{p_3\})$, δ задается табл. 6.

б) $A = (S, X, p_1, \delta, \{p_4, p_5\}$, δ задается табл. 7.

Таблица 6

	a	в
P1	P2	P5
P2	P3	P4
P3	P2	P5
P4	P3	P1
P5	P2	P4

Таблица7

	в	в	с
P1	P2	P3	P1
P2	P1	P6	P6
p3	P4	P4	P2
P4	P2	P3	P1
P5	P4	P6	P3
P6	P1	P2	P6

На пересечении строки p_i и столбца x указано значение $\delta(p_i, x)$.

2. Построить МП-автомат M_1 такой, что $X = \{a, b\}$ и $T(M)$ — множество всех цепочек, содержащих одинаковое число входящих символов a и b.

3. Построить МП-автомат M, такой, что $X = \{a, b\}$ и $T(M)=\{a^n b^n a^i | n\geq 1, i\geq 1\} \cup \{a^n b^{2n} a^i b^{3i} | n\geq 1, i\geq 1\}$.

4. Построить МП-автомат, допускающий язык, порождаемый грамматикой с правилами $P = \{s\to TT, T\to Ta, T\to bT, T\to c\}$.

5. Задано множество команд конечного автомата, допускающего язык $\{a^m b^n | n, m\geq 0\}$

$$(a_1 S_0) \to S_1;$$
$$(a_1 S_1) \to S_1;$$
$$(b_1 S_1) \to S_2;$$
$$(b_1 S_2) \to S_2;$$
$$(b_1 S_2) \to S_0.$$

Построить грамматику, порождающую этот язык, и определить ее тип.

6. Построить линейноограниченные автоматы, допускающие языки:

$$L_1 = \{a^n b^n a^n \mid n \geqslant 0\};$$
$$L_2 = \{xcx \mid x \in \{a, b\}^*\}.$$

7. Язык $L = \{a^p b^q c^r | p + q\geq r\}$ является КС-языком.

Выписать КС-грамматику, порождающую этот язык. Определить, к какому более узкому классу языков он принадлежит. Построить детерминированный МП-автомат, допускающий этот язык.

ЛИТЕРАТУРА

1. Айзерман М. А., Гусев Л. А. и др. Логика. Автоматы, алгоритмы. М., Физматгиз, 1963.
2. Асташов В. В. Элементы теории алгоритмов. Изд-во Пермского высшего командно-инженерного училища.
3. Г л у ш к о в В. М. Введение в кибернетику. Изд-во АН УССР, 1964.
4. Г л а д к и й А. В., Мельчук И. А. Элементы математической логики. М., «Наука», 1969.
5. Гросс М., Л а н т е н А. Теория формальных грамматик. М., «Мир», 1972.
6. Гинсбург С. Математическая теория контекстно-свободных языков. М., «Мнр», 1970.
7. Г у с е в Л. А., Смирнова И. М. Языки, грамматики н абстрактные автоматные модели. — «Автоматика и телемеханика», № 4, 5, М., «Наука», 1968.
8. Дятлов В. К. Нормальные алгоритмы и рекурсивные функции, ДАН СССР, т. 90, № 5, 1953.
9. Е р ш о в А. П. Операторные алгоритмы П. — В сб.: Проблемы кибернетики. Вып. 8. М., Физматгиз, 1962.
10. Ершов А. П. Об операторных схемах Янова. — В сб.: Проблемы кибернетики. Вып. 20. М., Физматгиз, 1968.

П.Колмогоров А. Н., Успенский В. А. К определению алгоритма, УМН, т. 13, вып. 4/82, 1958; Колмогоров А. Н. Три подхода к определению понятия количества информации. — «Проблемы передачи информации». Вып. 1, 1965.

12. Л ет и ч е в с к и й А. А. Синтаксис н семантика формальных языков. — «Кибернетика». Вып. 4, 1968.
13. М а р к о в А. А. Теория алгоритмов. Труды математического института им. В. А. Стеклова. М., Изд-во АН СССР, т. 42, 1954; Марков А. А. О нормальных алгоритмах, вычисляющих булевы функции, ДАН СССР, т. 157, № 2, 1964.
14. Мальцев А. И. Алгоритмы и рекурсивные функции. М., «Наука», 1960.
15. Мендельсон Э. Введение в математическую логику. М., «Наука», 1971.
16. Мелихов А. Н. Ориентированные графы и конечные автоматы. М., «Наука», 1971.
17. Минский М. Вычисления и автоматы. М., «Мир», 1971.
18. Новиков П. С. Об алгоритмической неразрешимости проблемы тождества слов в теории групп. Труды математического института им. В. А. Стеклова. М., Изд-во АН СССР, т. 44, 1955.
19. Трахтенброт Б. А. Алгоритмы н машинное решение задач. М., Физматгиз, 1960.
20. Трахтенброт Б. А. Сложность алгоритмов и вычислений. Новосибирск, 1967; Трахтенброт Б. А. Сигнализирующие функции и табличные операторы. — «Записки Пензенского ГПИ». Вып. 4, 1956.
21. Трахтенброт Б. А., Барздинь Я. М. Конечные автоматы. М., «Наука», 1970.
22. Тьюринг А. Может ли машина мыслить? М., Физматгиз, 1960.
23. Хомский Н. Введение в формальный анализ естественных языков.— «Кибернетический сборник». Вып. 1, 1965.
24. Хомский Н. Три модели для описания языка. — «Кибернетический сборник». Вып. 2, 1961.
25. Хомский Н. Формальные свойства грамматик. — «Кибернетический сборник». Вып. 2, 1966.
26. Цейтин Г. С. Оценка числа шагов при применении нормального алгоритма. Математика в СССР за 40 лет, т. 1, М., 1959.
27. Шенон К. Э. Универсальная машина Тьюринга с двумя внутренними состояниями. — В сб.: Автоматы. М., «Наука», 1956.
28. Шиханович Ю. А. Введение в современную математику. М., «Наука»,
29. Янов Ю.И. О логических схемах алгоритмов. — В сб.: Проблемы кибернетики. Вып. 1, М., Физматгиз, 1958.