Министерство образования и науки Российской Федерации
ГОУ ВПО "Алтайский государственный технический
университет им. И.И.Ползунова"

Е.Н. КРЮЧКОВА

ОСНОВЫ МАТЕМАТИЧЕСКОЙ ЛОГИКИ И ТЕОРИИ АЛГОРИТМОВ

Учебное пособие

Барнаул 2010

УДК 681.142.2:517

Крючкова Е.Н. Основы математической логики и теории алгоритмов: Учебное пособие / Алт. госуд. технич. ун-т им. И.И.Ползунова. Барнаул, 2010. — 277с.

Учебное пособие предназначено для студентов вуза с повышенным объемом подготовки по дискретной математике, в частности, для обучения специалистов по направлению 654600 — "Информатика и вычислительная техника".

Рекомендовано на заседании кафедры прикладной математики
протокол N 5 от 15 апреля 2010 г.

Рецензенты: профессор кафедры прикладной
математики АлтГТУ Л.И. Сучкова,
зав. кафедрой информатики АлтГУ Максимов А.В.

ВВЕДЕНИЕ

Данная книга представляет собой учебное пособие по курсу математической логики и теории алгоритмов. Цель этого учебника – выяснить и проанализировать некоторые фундаментальные понятия, алгоритмы и методы, лежащие в основе информатики. Несомненно, что важнейшая практическая задача — исследовать формальными средствами те понятия практического программирования, которые в той или иной степени относятся к проблемам, возникающим из самого вопроса о сущности программирования. Языки программирования занимают в программировании основное место. Однако необходимо понимать, что язык программирования — это не более чем средство для выражения алгоритмов, вид системы математических обозначений. Поэтому исследование систем, процессов и явлений, связанных с формальными системами вообще и теорией алгоритмов в частности, должны лежать в основе понимания самого процесса программирования.

В данном курсе можно выделить следующие основные разделы:

- формальные теории и математическая логика (глава 1),
- теория алгоритмов (главы 2, 3 и 4),
- теория сложности вычислений и эффективные алгоритмы (главы 5 и 6),
- теория формальных языков (глава 7),
- теория автоматов (главы 8 и 9).

Первый раздел посвящен теории формальных систем и, в частности, математической логике. Целю главы является систематическое построение формальной теории, включающее общее представление о конструировании формальных систем, методы логического вывода в исчислении высказываний и в исчислении предикатов, практическое приложении теории при построении систем логического программирования, моделировании сложных систем логическими методами.

Следующие разделы содержат информацию об основных теоретических вопросах теории алгоритмов, касающихся методов формализации понятия алгоритма, основных неразрешимых проблем теории алгоритмов, теории сложности вычислений. Знание неразрешимых проблем, понимание теории сложности алгоритмов и способов формирования эффективных алгоритмов необходимо профессиональному программисту, в какой бы области он не специализировался. По существу, оно дает понимание того, что можно и что нельзя выполнить с помощью вычислительных машин. Для профессионального программиста это особенно важно. Раздел, содержащий сведения по теории сложности, является теоретической базой для разработки эффективных программ. Анализ размера задачи и оценка временной сложности проектируемого алгоритма — один из фундаментальных принципов построения работоспособных программ. Излагаемые в качестве примеров задачи представляют интерес прежде всего с точки зрения эффективности проектируемых алгоритмов.

Важность результатов в этой области — не только в том, что они помогают оценить расход времени и памяти при решении задач на ЭВМ. Подобно тому, как общая

теория алгоритмов впервые показала, что бывают задачи неразрешимые, ее бурно развивающаяся ветвь — теория сложности — приводит к пониманию того, что бывают задачи объективно сложные (пример — так называемые переборные задачи), причем их сложность может оказаться в некотором смысле абсолютной, т.е. практически не устранимой никаким увеличением мощности вычислительных систем. Рассмотрение таких задач поучительно еще и потому, что почти все они несут характерный для дискретной математики эффект сочетания простоты формулировки со сложностью решения.

Последние два раздела посвящены формальным языковым системам — теории формальных языков и грамматик и теории автоматов. Именно теория языков и грамматик вместе с теорией автоматов являются теоретическими основами современной теории и практики построения компиляторов и других программ обработки текстов. Формальное определение таких систем — не самоцель. Его следует рассматривать лишь как метод, который необходим на пути от исходной идеи решения задачи к его реализации на конкретной машине. Примером шага на этом пути может служить переход от формальной модели языковой конструкции к ее реализации в виде программы на заданном языке программирования. Цель трех последних глав — рассмотреть общее определение формальных языковых систем посредством грамматик и автоматов, дать алгоритмы построения соответствующих формальных систем, рассмотреть алгоритмы их преобразования, описать синтаксические свойства языковых систем. С момента своего появления языки программирования, являясь средством общения человека с компьютером, представляют обширную и важную область прикладной математики. Реализация компиляторов или интерпретаторов для этих языков основана на теории формальных грамматик и автоматов. Фактически все методы построения компиляторов основаны на использовании автоматных и контекстно-свободных грамматик, а синтаксический анализатор, выполняющий грамматический разбор, является ядром любого компилятора. Алгоритмы и методы трансляции построены на основе формальных моделей. Если на пути от формальной модели к ее реализации на реальной машине выполнены все предписанные алгоритмом действия со всей требуемой аккуратностью, то полученная программа будет свободна от ошибок или по крайней мере от большого их количества.

Подводя итог всему вышесказанному следует отметить, что очень важно дать профессиональным программистам возможность осознать роль формализмов в их деятельности. Именно такого рода формализмы и рассматриваются в данном курсе.

Предполагается, что читатель знаком с основными понятиями теории множеств, началами математической логики в объеме булевой алгебры, теории графов, имеет хорошие навыки программирования на языке Си и опыт разработки алгоритмов. В связи с ограниченностью объема пособия, примеры, иллюстрирующие основные понятия предмета, вынесены большей частью в упражнения и задачи. Поэтому для более полного усвоения материала необходимо уделить время реализации помещенных в конце каждой главы заданий.

Каждая глава заканчивается тестами для самостоятельной оценки знаний студентами. Среди перечисленных вариантов возможных ответов или утверждений необходимо выбрать правильный ответ на вопрос или верное утверждение.

Студентам, желающим более подробно познакомиться с теорией, можно порекомендовать книги, список которых приведен в конце пособия.

Введем обозначения, которые будем использовать в книге.

Стандартный символ N обозначает множество всех натуральных чисел; $A, B, C, ...$ (первые прописные буквы латинского алфавита) обозначают подмножества множества N; $\emptyset$ — пустое множество; $a, b, ..., z$ (строчные буквы латинского алфавита) —

элементы множества N, т.е. числа.

Очевидно, что на множестве целых чисел операция деления не всегда определена, т.к. результат деления целых чисел не обязательно является целым числом. Поэтому будем использовать обозначения $\{a/b\}$ и $[a/b]$ соответственно для остатка и целой части от деления a на b. Например, $\{9/4\} = 1$, $[9/4] = 2$.

Будем использовать следующие теоретико-множественные обозначения: если A — множество натуральных чисел (т.е. $A \subseteq N$), то $\overline{A}$ обозначает дополнение A до N, т.е. $\overline{A} = N \setminus A$. $A = B$ означает, что A и B одинаковы как множества, т.е. A и B состоят из одних и тех же элементов; $x \in A$ означает, что x — элемент множества A. Обозначение $\{|\}$ указывает на образование множества: $\{x|...x...\}$ — это множество всех таких x, что выражение "$...x...$" верно для всех элементов этого множества. Знаком $\overset{x}{|}_y$ перед некоторым множеством A будем обозначать операцию подстановки элемента x вместо элемента y в множестве A. Например, если $A = \{1, 2, 3\}$, то множество $\overset{5}{|}_1 A$ равно $\{5, 2, 3\}$, а в качестве примера более сложной подстановки можно привести формулу

$$\overset{3}{|}_2 (\overset{3}{|}_1 A) = \{3\}.$$

Для заданных элементов x и y будем рассматривать упорядоченные пары $\langle x, y\rangle$, состоящие из элементов x и y, взятых именно в таком порядке. Аналогично будем использовать обозначение $\langle x_1, x_2, ..., x_n\rangle$ — для упорядоченной n–ки или кортежа длины n, состоящего из элементов x_1, x_2,..., x_n и именно в этом порядке. Через $A \times B$ обозначим декартово произведение множеств A и B, т.е. $A \times B = \{\langle x, y\rangle | x \in A \& y \in B\}$. Аналогично $A_1 \times A_2 \times ... \times A_n = \{\langle x_1, x_2, ..., x_n\rangle | x_1 \in A_1 \& x_2 \in A_2 \& ... \& x_n \in A_n\}$. Декартово произведение множества A на себя n раз обозначается A^n.

Будем использовать символы $P, Q, R, ...$ (прописные латинские буквы второй половины алфавита) для обозначения отношений на N. Если $R \subseteq N^n$, то R называется n–арным или n–местным отношением. Пусть R есть n–местное отношение. Будем говорить, что R однозначно, если для любого кортежа $\langle x_1, x_2, ..., x_{n-1}\rangle$ существует не более одного элемента z, такого, что $\langle x_1, x_2, ..., x_{n-1}, z\rangle \in R$. Очевидно, что однозначное n–местное отношение можно рассматривать как отображение его области определения в N. По этой причине, вместо того, чтобы говорить, что R — однозначное n-местное отношение, будем говорить, что R — функция от $n - 1$ аргументов. Для обозначения функций будем также использовать f, g, h и т.д. — строчные буквы латинского алфавита. Если функция $f(\overline{x})$ имеет n аргументов, то для явного указания числа аргументов, если это необходимо, будем указывать число аргументов в виде верхнего индекса: $f^n(\overline{x})$ или $f^n(x_1, x_2, \dots x_n)$.

Если область определения функции f от k аргументов может не совпадать с N^k, то f называется частичной функцией. В том случае, когда область определения функции f от k аргументов совпадает с N^k, функция f называется всюду определенной. Обычно область определения функции f обозначается $Dom(f)$, а область значения этой функции $Ran(f)$. В соответствии с указанным выше, будем рассматривать такие функции, у которых $Ran(f) \subseteq N$, а $Dom(f) \subseteq N^k$.

Если f и g — функции, то композиция $f \cdot g$ этих функций есть, по определению, такая функция, что $(f \cdot g)(x) = g(f(x))$. Функция $(f \cdot g)(x)$ определена тогда и только тогда, когда определены $f(x)$ и $g(f(x))$. Например, если $g(x) = x^2$ и $f(x) = x + 1$, то $(f \cdot g)(x) = g(f(x)) = (x + 1)^2$ и $(g \cdot f)(x) = f(g(x)) = x^2 + 1$.

Другие общие и специальные обозначения будут вводиться по мере необходимости.

Глава 1

ФОРМАЛЬНЫЕ ТЕОРИИ

1.1 Формальные модели

Математика — это взаимосвязанная совокупность математических дисциплин или *математических теорий*. Некоторые из них так и называются, например, теория чисел, теория множеств, теория графов и т.п. Другие называются иначе и не содержат в своем наименовании упоминания о теории или исчислении.

Математические теории построены с разной степенью формализации, но все теории возникли в результате операции абстрагирования явлений реального мира. Абстрагирование является первым шагом при построении математической теории, оно широко используется в науке для исследования различных аспектов некоторого явления. Реальные явления и объекты, как правило очень сложны и многообразны, и абстракция применяется для того, чтобы ограничить это многообразие, выделить принципиальные моменты исследуемого явления. Построенная *формальная модель* позволяет выполнить исследование модели на формальном уровне, доказать основные свойства в точных терминах математических определений. То, что выполнено по формальным правилам, легко поддается проверке. Если все этапы последовательных преобразований выполнены корректно, то не вызывает сомнения и достоверность связи между исходными условиями и полученными выводами.

Например, любая программа для компьютера представляет собой четкий и однозначный способ преобразования исходных данных. Без строжайшей формализации было бы невозможно создавать алгоритмы для их реализации в виде программ для ЭВМ, а, следовательно, было бы невозможно использовать вычислительную технику ни в одной области человеческой деятельности. Выяснение того, какие объекты и действия над ними следует считать точно определенными, какими свойствами и возможностями обладают комбинации элементарных действий — все это является предметом формальной теории.

Однако, при построении формальной модели вполне можно так абстрагироваться от существенных явлений реальности, что построенные теории будут неверными в существующей реальности. Построение и истолкование математической теории, когда каждое понятие более или менее соответствует некоторому явлению окружающей нас действительности, называется *содержательным истолкованием теории*. Соответствие законов, связей и отношений объектов формальной модели элементам реального мира называется *адекватностью*. Степень адекватности определяет, применимы ли полученные в результате формального вывода результаты к конкретным проблемам в реальном мире.

Любая формальная теория определяется заданием четырех ее элементов:

— алфавита,
— множества формул,
— множества аксиом,
— множества правил вывода.

Алфавит Σ *формальной теории* — это конечное множество символов. Если множество всевозможных цепочек над алфавитом Σ обозначить Σ^*, то множество F формул формальной теории — это некоторое подмножество Σ^*, т.е. $F \subseteq \Sigma^*$. Множество A аксиом — это некоторое подмножество множества ее формул. Аксиомы, если число их конечно, задаются перечислением. Если число аксиом бесконечно, то они должны быть заданы некоторыми конечными правилами, позволяющими эффективно распознавать аксиомы среди прочих формул.

Правила вывода должны давать возможность по некоторому набору формул B_1, B_2, ..., B_n и формуле C установить, выводима ли формула C из формул B_1, B_2, ..., B_n. Правила вывода обычно записываются в виде

$$B_1, B_2, ...B_n \vdash C$$

или

$$\frac{B_1, B_2, ...B_n}{C}$$

или

$$B_1, B_2, ...B_n \Rightarrow C.$$

В этом случае говорят, что формула C является *непосредственным следствием* формул B_1, B_2, ..., B_n или *непосредственно выводима* из них.

Доказательством в рамках формальной теории является последовательность утверждений, приводящих от исходных утверждений, принимаемых за истинные, к их логическим следствиям. *Выводом* формулы C в формальной теории называется конечная последовательность формул C_1, C_2, ..., C_n, где $C_n = C$, а каждая из формул C_i для $i = 1, ..., n$ — это либо аксиома формальной теории, либо непосредственное следствие каких–либо предыдущих формул этой последовательности в соответствиии с одним из правил вывода.

Особенно следует отметить, что при построении любой формальной теории мы имеем дело только с *конечными* множествами. Рассматриваемые нами теории будут строиться на основе *финитного метода.* Опыт парадоксов теории множеств научил математику крайне осторожно обращаться с бесконечностью и по возможности даже о бесконечности рассуждать с помощью конечных методов. Существо финитного подхода заключается в том, что он допускает только конечные действия над конечными объектами. Выяснение того, какие объекты и действия над ними следует считать точно определенными, какими свойствами и возможностями обладают комбинации элементарных действий — все это является предметом теории алгоритмов и формальных систем.

1.2 Исчисление высказываний

Математическая логика изучает формы мышления и способы проверки правильности рассуждений. Логика начинается тогда, когда относительно рассматриваемых объектов делаются некоторые утверждения или *высказывания.* Логика интересуется прежде всего истинностью или ложностью высказываний. Высказывания могут быть
— тождественно истинными,

— тождественно ложными,
— имеющими переменное значение истинности.

1.2.1 Алфавит исчисления высказываний

В алфавит исчисления высказываний входят большие латинские буквы (с возможными индексами) для обозначения высказываний. Эти символы будем называть переменными высказываниями.

Все тождественно истинные высказывания с точки зрения математической логики эквивалентны. Это же можно сказать и для тождественно ложных высказываний. Для тождественно истинного высказывания в математической логике вводится обозначение "истина" (или $true$ или T), а для тождественно ложного — обозначение "ложь" (или $false$ или F). Соответствующие обозначения входят в алфавит логической формальной теории — исчисления высказываний.

В алфавит входят также обозначения логических операций (или логических связок) и знаки круглых скобок "(" и ")".

Существуют различные варианты построения исчисления высказываний, которые по форме довольно сильно отличаются друг от друга, но практически совпадают по существу результатов, т.е. по своим основным свойствам. Рассмотрим вариант исчисления, который является простым и компактным как по форме определений, так и по доказательствам теорем о свойствах исчисления.

В качесте логических связок в исчислении высказываний рассмотрим четыре операции:

$$\&, \vee. \rightarrow, \neg.$$

Они носят следующие названия: & —конъюнкция или логическое умножение, $\vee$ — дизъюнкция или логическое сложение, $\rightarrow$ — импликация или логическое следование, $\neg$ — отрицание.

Иных символов, кроме указанных, в исчислении высказываний нет.

1.2.2 Множество формул исчисления высказываний

Формулы исчисления высказываний представляют собой конечные последовательности символов алфавита исчисления высказываний, записанные по определенным правилам. Определение формулы можно дать рекурсивно следующим образом.

Определение 1.1. Определение формулы

а) переменное высказывание есть формула;

б) если α и β есть формулы, то формулами являются также

$$(\alpha) \text{ и } (\beta),$$
$$\neg\alpha \text{ и } \neg\beta,$$
$$\alpha\&\beta,$$
$$\alpha \vee \beta,$$
$$\alpha \rightarrow \beta;$$

в) никаких других формул в исчислении высказываний нет.

Строго говоря, при определении формулы надо было бы для каждого знака бинарной операции, которым связывались две формулы, и для каждого знака унарной

операции, примененного к некоторой формуле, заключать в круглые скобки полученную в результате формулу. Однако, для уменьшения количества скобок в формулах вводятся правила приоритетов операций.

Итак, формулы исчисления высказываний — просто последовательности символов, удовлетворяющие определенныйм правилам записи. Например, слова

$$(\neg A \to \neg B) \to (B \to A),\ (((A))),\ A\&B\&C \vee D$$

являются формулами исчисления высказываний. Наоборот, следующие слова формулами не являются

$$\neg A \to \neg B)(B \to A),\ (((A,\ A\& \vee \&B\&C \vee .$$

Формулы исчисления высказываний иначе называются пропозициональными формами, а частный случай формулы — переменные высказывания и константы $true$, $false$ — пропозициональными переменными и пропозициональными константами соответственно.

Задачей логики является определение смысла каждой формулы. Если каждое высказываение в формуле может быть истинным или ложным и определены правила выполнения операции над соответствующими значениями, то можно вычислить значение каждой формулы для заданных значений переменных высказываний.

Существует определенная связь между алгеброй логики и исчислением высказываний. Во–первых, алгебра логики вполне удовлетворяет требованиям сторогости работы с бесконечностью, т.к. в алгебре рассматриваются конечные наборы символов и конечное число операций между ними. Вместо букв можно подставлять символы $true$ и $false$, а затем вычислять значение.

В отличие от алгебры логики исчисление высказываний — это аксиоматическая система, предназначенная для моделирования логического мышления. Формальная математическая логика решает проблемы проверки правильности рассуждений о реальном мире, строит логические модели и правила их преобразования. Каждому атому, входящему в формулу исчисления высказываний, можно приписать значение $true$ или $false$. Значения формул для всевозможных *интерпретаций* атомов определяются на основе таблиц истинности для основных операций:

x	y	$\neg x$	$x\&y$	$x \vee y$	$x \to y$
$false$	$false$	$true$	$false$	$false$	$true$
$false$	$true$	$true$	$false$	$true$	$true$
$true$	$false$	$false$	$false$	$true$	$false$
$true$	$true$	$false$	$true$	$true$	$true$

Аппарат алгебры логики весьма похож на аппарат исчисления высказываний, однако они решают разные задачи:

— алгебра логики занимается проблемами двоичного преобразования информации,

— логические исчисления (исчисление высказываний и исчисление предикатов) работают с абстракциями, построенными из предложений и рассуждений естественного языка, предназначены для формализации процессов мышления человека.

В формальной логике большую роль играют формулы, принимающие одно и то же значение "$true$" для всех значений переменных высказываний, входящих в формулу. Такие формулы называютмся *общезначимыми* или *тавтологиями.* Формулы, принимающие значение "$false$" для всех значений своих аргументов, называются *невыполнимыми.*

1.2.3 Множество аксиом исчисления высказываний

Множество аксиом исчисления высказываний зададим следующими тремя схемами аксиом:

$A1)$ $\alpha \to (\beta \to \alpha)$,
$A2)$ $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$,
$A3)$ $(\neg\alpha \to \neg\beta) \to (\beta \to \alpha)$.

Аксиомы $A1$ – $A3$ записаны только с использованием операций импликации и отрицания. Операции дизъюнкции и конъюнкции в аксиомах отсутствуют. Вообще говоря, можно построить исчисление высказываний и только на основе операций импликации и отрицания. Такое исчисление будет содержать в можестве аксиом только три аксиомы $A1$ – $A3$, а из алфавита исключаются знаки $\&$ и $\vee$.

В рассматриваемом нами исчислении высказываний с четырьмя логичекими операциями $\&, \vee, \to, \neg$ необходимо ввести соотношения между базовыми операциями импликации и отрицания и дополнительными операциями дизъюнкции и конъюнкции:

$A4)$ $\alpha \vee \beta$ означает $\beta \vee \alpha$ и означает $(\neg\alpha) \to \beta$,
$A5)$ $\alpha\&\beta$ означает $\neg((\neg\alpha) \vee (\neg\beta))$.

Часто в множество операций вводят еще одну операцию — эквивалентность, правила для которой имеют вид:

$$\alpha \Leftrightarrow \beta \text{ означает } (\alpha \to \beta)\&(\beta \to \alpha).$$

Можно ли какую–нибудь аксиому вывести из остальных, применяя правила вывода данной системы? Если оказывается, что некоторую аксиому можно таким образом вывести из остальных, то ее можно вычеркнуть из списка аксиом, и логическое исчисление при этом не изменится, т.е. множество его выводимых формул останется тем же.

Определение 1.2. Аксиома, не выводимая из остальных аксиом, называется независимой от этих аксиом, а система аксиом, в которой ни одна аксиома не выводится из остальных, называется *независимой системой аксиом*. В противном случае система аксиом называется зависимой.

Ясно, что зависимая система аксиом в некотором смысле менее совершенна, т.к. она содержит лишние аксиомы. Вопрос о независимости одной аксиомы некоторой системы от других аксиом часто бывает равносилен вопросу о возможности заменить без противоречия в рассматриваемой системе данную аксиому ее отрицанием.

Можно доказать, что система аксиом исчисления высказываний независима (доказательство можно найти, например, в [37], стр 112).

1.2.4 Правила вывода исчисления высказываний

В исчислении высказываний имеется единственное правило вывода, согласно которому формула β является непосредственным следствием формул α и $\alpha \to \beta$, каковы бы не являлись формулы α и β. Это правило записывается в виде схемы

$$\frac{\alpha, \alpha \to \beta}{\beta}.$$

Оно называется правилом заключения или правилом *modus ponens* и кратко обозначается MP. Термин modus ponens берет свое начало еще со времен Аристотеля и в

переводе на русский язык означает "способ спуска", т.к. правило вывода MP позволяет последовательно спускаться по операции импликации к правой части формулы.

Пример. Следующая последовательность из пяти формул является выводом формулы $A \to A$:

1) $A \to ((A \to A) \to A)$ (аксиома $A1$ при $\beta = A \to A$),
2) $(A \to ((A \to A) \to A)) \to ((A \to (A \to A)) \to (A \to A))$,
(аксиома $A2$ при $\beta = A \to A), \gamma = A$,
3) $((A \to (A \to A)) \to (A \to A))$ следует из 1 и 2 по MP,
4) $A \to (A \to A)$ (аксиома $A1$ при $\beta = A$),
5) $A \to A$ следует из 3 и 4 по MP.

Фактичести мы доказали теорему об общезначимости формулы $A \to A$ или, что то же самое, о выводимости в исчислении высказываний этой формулы. Приведем краткую формулировку этой теоремы.

Теорема 1.1. $\vdash A \to A$.

1.2.5 Теорема о дедукции исчисления высказываний

В математике часто требуется выполнить доказательство утверждений типа "из α следует β", где α и β — некоторые утверждения. Обычно такое доказательство проводим по следующей схеме. Предполагаем, что α справедливо. Посредством некоторой цепочки рассуждений устанавливаем, что и β справедливо. Тогда заключаем, что из α следует β. В рассуждениях может участвовать ряд дополнительных предположений. Обоснованием этого способа доказательства является следующая теорема, которая называется теоремой о дедукции для исчисления высказываний.

Теорема 1.2 (о дедукции). Пусть Γ — некоторый список формул и α — одна формула. Тогда если $\Gamma, \alpha \vdash \beta$, то $\Gamma \vdash \alpha \to \beta$.

Доказательство. Сначала сформулируем теорему подробнее следующим образом: если формула β выводима из списка гипотез Γ, дополненного формулой α, то формула $\alpha \to \beta$ выводима из списка гипотез Γ.

Теперь рассмотрим имеющийся по условию теоремы вывод β из Γ, α. Пусть это вывод формул

$$\beta_1, \beta_2, ..., \beta_n.$$

Индукцией по i покажем, что формула $\alpha \to \beta_i$ выводима из списка гипотез Γ. Рассмотрим всевозможные варианты вывода очередной формулы β_i. В соответствием определением выводимости существует всего четыре варианта вывода β_i:

а) β_i — аксиома,
б) β_i — гипотеза из Γ,
в) β_i — гипотеза α,
г) β_i — непосредственное следствие из некоторых β_m и β_k по правилу MP.

Заметим, что для первого шага вывода при $i = 1$ возможны только первые три варианта из вышеперечисленных. Рассмотрим общий случай — перечисленные четыре варианта.

а) Если β_i — аксиома, то продолжим вывод:

β_i (аксиома),
$\beta_i \to (\alpha \to \beta_i)$ (аксиома A1),
$\alpha \to \beta_i$ (правило MP).

При любом i в выводе не участвует формула α, таким образом, в соответствии с определением выводимости можем считать, что доказан вывод $\Gamma \vdash \alpha \to \beta_i$.

б) Если β_i — гипотеза из Γ, то вывод строится точно также, за тем исключением, что основанием для первого шага является принадлежность β_i множеству Γ.

в) Если β_i — гипотеза α, то вывод $\alpha \to \alpha$ мы выполнили при доказательстве теоремы 1.1. Таким образом, доказали $\vdash \alpha \to \beta_i$ и, следовательно, в соответствии с определением выводимости $\Gamma \vdash \alpha \to \beta_i$.

г) Если β_i — непосредственное следствие из некоторых β_m и β_k по правилу MP:

$$\begin{array}{l}\beta_m,\\ \beta_k = \beta_m \to \beta_i,\\ \beta_i.\end{array}$$

Воспользуемся индукцией: значения m и k меньше значения i, следовательно, в соответствии с индуктивным предположением из Γ выводимы формулы $\alpha \to \beta_m$ и $\alpha \to \beta_k$. Фактически, в силу равенства $\beta_k = \beta_m \to \beta_i$, которое существует по причине применимости правила MP в выводе, выводима формула $\alpha \to (\beta_m \to \beta_i)$. Продолжим этот вывод:

$$\begin{array}{l}\alpha \to \beta_m, \text{ (по индукции)},\\ \alpha \to (\beta_m \to \beta_i) \text{ (по индукции)},\\ (\alpha \to (\beta_m \to \beta_i)) \to ((\alpha \to \beta_m) \to (\alpha \to \beta_i)) \text{ (аксиома A2)},\\ (\alpha \to \beta_m) \to (\alpha \to \beta_i) \text{ (правило MP)},\\ \alpha \to \beta_i \text{ (правило MP)}.\end{array}$$

Следовательно, и в этом четвертом случае существует вывод $\Gamma \vdash \alpha \to \beta$.

Таким образом, рассмотрев всевозможные варианты вывода, мы установили, что для любого i для всех вариантов вывода β_i из Γ и α мы указали способ продолжения вывода для того, чтобы получить $\Gamma \vdash \alpha \to \beta_i$. Таким образом, из Γ выводима и формула $\alpha \to \beta_n$, совпадающая с $\alpha \to \beta$. □

Теорема 1.3. $\alpha \to \beta, \beta \to \gamma \vdash \alpha \to \gamma$.

Доказательство. В соответствии с формулировкой торемы у нас есть две гипотезы $\alpha \to \beta$ и $\beta \to \gamma$. Пусть эти две гипотезы составляют множество гипотез Γ. Рассмотрим вспомогательную гипотезу α и рассмотрим вывод:

$$\begin{array}{l}1)\ \alpha \to \beta, \text{ (гипотеза)},\\ 2)\ \beta \to \gamma, \text{ (гипотеза)},\\ 3)\ \alpha, \text{ (гипотеза)},\\ 4)\ \beta, \text{ (правило MP из 1 и 3)},\\ 5)\ \gamma, \text{ (правило MP из 2 и 4)},\end{array}$$

Таким образом, доказали

$$\alpha \to \beta, \beta \to \gamma, \alpha \vdash \gamma.$$

В соответствии с теоремой о дедукции $\alpha \to \beta, \beta \to \gamma \vdash \alpha \to \gamma$. □

Полученное при доказательстве теоремы 1.3 правило выводимости можно записать как

$$\frac{\alpha \to \beta, \beta \to \gamma}{\alpha \to \gamma}.$$

Это правило называется правилом силлогизма или правилом S. В качестве упражнения для самостоятельной работы Вам предлагается доказать следующие правила:

1) доказательство от противного (modus tollens)

$$\frac{\alpha \to \beta, \neg\beta}{\neg\alpha},$$

2) правило перестановки посылок

$$\frac{\alpha \to (\beta \to \gamma)}{\beta \to (\alpha \to \gamma)},$$

3) правило соединения посылок

$$\frac{\alpha \to (\beta \to \gamma)}{\alpha\&\beta \to \gamma},$$

4) правило разъединения посылок

$$\frac{\alpha\&\beta \to \gamma}{\alpha \to (\beta \to \gamma)},$$

5) правило исключения промежуточной посылки

$$\frac{\alpha \to (\beta \to \gamma), \beta}{\alpha \to \gamma},$$

6) правила конъюнкции

$$\frac{\alpha, \beta}{\alpha\&\beta}, \quad \frac{\alpha\&\beta}{\alpha, \beta},$$

7) простая конструктивная дилемма

$$\frac{\alpha \to \gamma, \beta \to \gamma, \alpha \vee \beta}{\gamma},$$

8) сложная конструктивная дилемма

$$\frac{\alpha \to \gamma, \beta \to \delta, \alpha \vee \beta}{\gamma \vee \delta},$$

9) простая деструктивная дилемма

$$\frac{\alpha \to \beta, \alpha \to \gamma, \neg\beta \vee \neg\gamma}{\neg\alpha},$$

10) сложная деструктивная дилемма

$$\frac{\alpha \to \beta, \delta \to \gamma, \neg\beta \vee \neg\gamma}{\neg\alpha \vee \neg\beta}.$$

11) правила двойного отрицания

$$\frac{\neg\neg\alpha}{\alpha}, \quad \frac{\alpha}{\neg\neg\alpha}.$$

Силлогизмом называется схема такая рассуждений, в которой заключение всегда верно. Впервые силлогизмы исследовались еще Аристотелем. Все вышеприведенные правила являются силлогизмами. Последовательное применение нескольких силлогизмов позволяет строить заключение.

1.3 Исчисление предикатов

1.3.1 Определение формальной теории исчисления предикатов

Перейдем к рассмотрению другой математической теории — исчисления предикатов, которое является развитием исчисления высказываний.

Каждое высказывание (или каждая формула исчисления высказываний) имеет значение "истина" или "ложь". Исчисление высказываний оперирует с отдельными высказываниями, представляющими собой аналоги повествовательных предложений на естественном языке, каждому из которых можно приписать одно из двух семантических значений: "истина" или "ложь". В естественном языке встречаются более сложные повествовательные предложения, истинность которых может меняться при изменении объектов и при неизменной структуре самого предложения. Например, фраза "*студент x отлично учится*" может иметь различные значения в зависимости от того, какое значение будет принимать переменная x. В математической логике предложения, смысл которых меняется в зависимости от параметров, определяются на формальном уровне с помощью предикатов.

Определение 1.3. Функция одной или нескольких переменных, которая принимает логическое значение "истина" или "ложь", называется предикатом. Переменные предиката называются предметными переменными.

1.3.2 Алфавит и множество формул исчисления предикатов

Алфавит исчисления высказываний полностью входит в алфавит исчисления предикатов. Большие латинские буквы получают в исчислении предикатов новый смысл: они могут обозначать как постоянные высказывания (например, A, B), так и переменные высказывания — предикаты (например, $F(x), G(x, y)$). Кроме того, в алфавит исчисления предикатов дополнительно по сравнению с исчислением высказываний входят

— маленькие латинские буквы с возмжными индексами, называемые предметными переменными (например, y, x_1, x_2); они обозначают некоторые объекты, но, в отличие от предметных констант, каждая из предметных переменных может обозначать любой, совершенно произвольный объект;

— квантор всеобщности $\forall$;

— квантор существования $\exists$.

Операции $\forall$ и $\exists$ выражают собой утверждения всеобщности и существования соответственно. Пусть $R(x)$ — некоторый предикат, принимающий значение "истина" или "ложь" для каждого элемента x в некоторой предметной области Ω. Тогда под выражением

$$\forall x \; R(x)$$

понимается: "*для каждого элемента x области* Ω *высказывание* $R(x)$ *истинно*". А под выражением

$$\exists x \; R(x)$$

понимается: "*существует элемент x области Ω, для которого высказывание $R(x)$ истинно*".

Переменная x в этих выражениях называется *связанной переменной*. Иначе говоря, переменная связана, если она находится в области действия этого квантора. Предметная переменная, не связанная никаким квантором, называется *свободной переменной*.

Правило построения формул исчисления предикатов дополним возможностью ставить перед любой формулой квантор существования или всеобщности.

1.3.3 Множество аксиом исчисления предикатов

Определим теперь множество аксиом:

$A1)$ $\alpha \to (\beta \to \alpha)$,
$A2)$ $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$,
$A3)$ $(\neg\alpha \to \neg\beta) \to (\beta \to \alpha)$,
$A4)$ $\forall x\ \alpha(x) \to \alpha(t)$
$A5)$ $\forall x\ (\alpha \to \beta) \to (\alpha \to \forall x\ \beta)$.

В аксиоме A4 переменная t должна быть свободной для предиката α. В аксиоме A5 предикат α не должен содержать предметной переменной x.

Аксиомы исчисления предикатов не содержат квантора существования. Для того, чтобы ввести в исчисление квантор существоания, свяжем его с квантором всеобщности определением:

$$\exists x\ R(x) \text{ эквивалентно } \neg(\forall x\ \neg R(x)).$$

1.3.4 Правила вывода исчисления предикатов

В исчислении предикатов имеется два правила вывода: знакомое нам правило MP

$$\frac{\alpha, \alpha \to \beta}{\beta}.$$

и правило обобщения (Gen)

$$\frac{\alpha}{\forall x\ \alpha}.$$

Это правило позволяет навешивать квантор всеобщности по произвольной переменной. Если формула $\forall x\ \alpha$ включена в некоторый вывод на основании правила Gen, то говорят, что в этом выводе использовано правило обобщения по переменной x.

1.3.5 Теорема о дедукции исчисления предикатов

Теорему о дедукции нельзя перенести на исчисление предикатов в том же виде, в каком она была сформулирована для исчисления высказываний. Причина этого кроется в наличии свободных и связанных переменных.

Теорема 1.4 (о дедукции исчисления предикатов). Пусть Γ — некоторый список формул и α — одна формула. Тогда если существует вывод $\Gamma, \alpha \vdash \beta$, в котором правило обобщения Gen не применяется ни по какой из переменных, свободных в

формуле α, то существует вывод $\Gamma \vdash \alpha \to \beta$, в котором правило Gen применяется по тем же переменным, по которым оно применялось в исходном выводе $\Gamma, \alpha \vdash \beta$.

Доказательство. Пусть построен вывод формулы β из совкупности гипотез Γ, α:

$$\beta_1, \beta_2, ..., \beta_n.$$

Так же, как и при доказательстве теоремы о дедукции для исчисления высказываний воспользуемся индукцией по i в выводе β из Γ, α. Дополнительно к вариантам, рассмотренным при доказательстве теоремы для исчисления высказываний, в исчислении предикатов имеется еще только один вариант вывода: формула β_i получена в результате применения правила Gen к некоторой предшествующей формуле β_j, $j < i$. Тогда формула β_i должна иметь вид $\forall x\ \beta_j$ и по условию теоремы переменная x не должна входить в α свободно. По индуктивному предположению $\Gamma \vdash \alpha \to \beta_j$. Рассмотрим вывод

$\forall x\ (\alpha \to \beta_j)$ (Gen),
$\forall x\ (\alpha \to \beta_j) \to (\alpha \to \forall x\ \beta_j)$ (аксиома A5),
$\alpha \to \forall x\ \beta_j$ (правило MP).

Так как $\beta_i = \forall x\ \beta_j$, то из Γ получили доказательство $\alpha \to \beta_i$. $\square$

Пример. Покажем $\alpha(t) \to \exists x\ \alpha(x)$:

1) $\alpha(t)$ (посылка),
2) $\forall x\ \neg\alpha(x) \to \neg\alpha(t)$, (аксиома A4)
3) $(\forall x\ \neg\alpha(x) \to \neg\alpha(t)) \to (\alpha(t) \to \neg\forall x\ \neg\alpha(x))$, (аксиома A3) ,
4)$\alpha(t) \to \neg\forall x\ \neg\alpha(x)$, (MP 2,3)
5)$\neg\forall x\ \neg\alpha(x)$, (MP 4,1)
6)$\neg\forall x\ \neg\alpha(x) = \exists x\ \alpha(x)$, (определение)
7)$\exists x\ \alpha(x)$, (подстановка 6 в 5)
8)$\alpha(t) \to \exists x\ \alpha(x)$, (теорема о дедукции)..

Получили дополнительное правило вывода в исчислении предикатов

$$\frac{\alpha(t)}{\exists x\ \alpha(x)}.$$

1.3.6 Теоремы о полноте

Мы уже упоминали понятие общезначимости формул, понимая под ними такие формулы, которые принимают значение "истина" при всех значениях входящих в эту формулу переменных высказываний. Любая математическая теория интересна не только сама по себе, но и с точки зрения ее приложения в практических целях. Формулы исчисления высказываний могут иметь некоторый смысл и обозначать некоторые высказывания естественного языка, если существует какая–либо *интерпретация* математической теории. Говоря неформально, интерпретировать математическую теорию — это значит связать с ней некоторую предметную область и указать соответствие формальных объектов математической теории и объектов данной предметной области. Дадим определение интерпретации на формальном уровне.

Определение 1.4. Интерпретация I математической теории T задается следующим образом:

1) указано некоторое множество D — предметная область интерпретации I; $D \neq \emptyset$;

2) каждой предметной константе (если они есть) из T сопоставлен некоторый элемент D;

3) каждой функциональной букве (если они есть) из T сопоставлена операция над элементами D, которая совокупности объектов области D ставит в соответствие один элемент этой же области; число аргументов функциональной буквы теории T и число аргументов операции над элементами D должны совпадать;

4) каждой n–местной предикатной букве (если они есть) из T сопоставлено конкретное отношение над n элементами области D;

5) каждой пропозициональной букве (если они есть) из T сопоставлено одно из двух логических значений "истина" или "ложь".

В этом определении ничего не говорится о том, что может быть сопоставлено предметным переменным. Предполагается, что предметная переменная может обозначать любой элемент области D.

Определение 1.5. Формула называется общезначимой, если она истинна при любой интерпретации.

Логическая формальная теория называется полной, если любая формула выводима в этой теории тогда и только тогда, когда она общезначима. Полнота теории гарантирует, что во–первых, теория содержит все необходимое для формального вывода, а во–вторых, что ничего лишнего и неверного в этой теории вывести нельзя. Если теория полна, достаточно просто перебирать все варианты выводов в этой теории и получать различные общезначимые формулы (теоремы). Однако, история математики показывает, что существуют очень трудные теоремы, поиск доказательства которых требует больших творческих усилий и временных затрат. Это наводит на мысль, что полнота маьематических теорий — достаточно редкое свойство. Так оно и есть на самом деле. Исчисление высказываний является одной из весьма редких теорий, для которых выполняется свойство полноты.

Сначала отметим, что формулы исчисления высказываний можно интерпретировать как формулы алгебры высказываний. Для этого будем трактовать свободные переменные исчисления высказываний как переменные алгебры высказываний, т.е. переменные в содержательном смысле, принимающие значения $true$, $false$. Если логические операции определим так же, как в алгебре высказываний, то всякая формула при любых значениях перемнных сама будет принимать некоторое значение $true$ или $false$, вычисленное по правилам алгебры высказываний.

Пусть α — некоторая формула исчисления высказываний, I — интерпретация, в которой определены значения всех пропозициональных букв, входящих в формулу α, и, следовательно, определено значение $V(\alpha)$ этой формулы. Обозначим через $\tilde{\alpha}$ формулу, совпадающую с формулой α, если $V(\alpha) = true$, и имеющую изображение $\neg\alpha$, если $V(\alpha) = false$. Таким образом, всегда $V(\tilde{\alpha}) = true$.

Аналогичный смысл имеют обозначения $\tilde{\beta}_1$, $\tilde{\beta}_2$,...,$\tilde{\beta}_n$ для формул β_1, β_2,...,β_n — любых подформул формулы α.

Теорема 1.5. Во введенных обозначениях для произвольной интерпретации I имеет место

$$\tilde{\beta}_1, \tilde{\beta}_2, ..., \tilde{\beta}_n \vdash \tilde{\alpha}.$$

Доказательство. Проведем доказательство по индукции по n — числу логических связок в формуле α.

Если $n = 0$, то формула α не содержит логических связок, т.е. является элементарной формулой и состоит из одной пропозициональной буквы B_i. Тогда значение α совпадает со значением B_i, поэтому $\tilde{\alpha}$ совпадает со значением $\tilde{\beta}$, следовательно, вывод формулы $\tilde{\alpha}$ из $\tilde{\beta}_1$, $\tilde{\beta}_2$,...,$\tilde{\beta}_n$ состоит только из одного шага — гипотезы $\tilde{\beta}$.

Допустим теперь, что для любого $n \geq 0$ для всех формул, содержащих не более n связок, утверждение теоремы справедливо. Докажем, что это утверждение справедливо и для $n+1$. В соответствии с определением формулы исчисления высказываний формула α может быть получена из одной или двух формул, каждая из которых содержит не более n логических связок, с помощью логической связки отрицания, импликации, конъюнкции или дизъюнкции. Для доказательства надо рассмотреть все четыре варианта образования формулы α с помощью четырех логических связок.

Пусть, например, $\alpha = \neg\beta$. По индуктивному предположению

$$\tilde{\beta}_1, \tilde{\beta}_2, ..., \tilde{\beta}_n \vdash \tilde{\beta}.$$

Если $V(\beta) = true$, то $\tilde{\beta} = \beta$ и указанный вывод $\tilde{\beta}$ превращается в

$$\tilde{\beta}_1, \tilde{\beta}_2, ..., \tilde{\beta}_n \vdash \beta.$$

При этом $V(\alpha) = V(\neg\beta) = false$ и, следовательно,

$$\tilde{\alpha} = \neg\alpha = \neg\neg\beta.$$

Мы ранее доказали для любого высказывания γ справедливость вывода (см. правила двойного отрицания)

$$\vdash \gamma \to \neg\neg\gamma,$$

поэтому к выводу $\tilde{\beta}_1, \tilde{\beta}_2, ..., \tilde{\beta}_n \vdash \beta$ можно приписать вывод формулы $\vdash \beta \to \neg\neg\beta$, получив тем самым вывод $\vdash \alpha$.

Если же $V(\beta) = false$, то $\tilde{\beta} = \neg\beta$, $V(\alpha) = V(\neg\beta) = true$, $\tilde{\alpha} = \alpha = \neg\beta$, и, таким образом, вывод $\tilde{\beta}_1, \tilde{\beta}_2, ..., \tilde{\beta}_n \vdash \tilde{\beta}$ одновременно является и выводом $\tilde{\alpha}$.

Доказательство для остальных случаев оставляется читателю в качестве упражнения. □

Теорема 1.6 (о полноте исчисления высказываний). Формула α выводима в исчислении высказываний тогда и только тогда, когда она общезначима.

Доказательство.

1. Пусть формула исчисления высказываний α общезначима и $B_1, B_2, ..., B_k$ — список всех входящих в нее букв. Возможны 2^k различных интерпретаций, которые ставят в соответствие этим буквам различные комбинации значений. На каждой из этих интерпретаций α имеет значение $V(\alpha)$, истинное в силу общезначимости α. Поскольку формула α истинна в любой из этих 2^k интерпретаций, $\tilde{\alpha}$ совпадает с α и из теоремы 1.5

$$\tilde{B}_1, \tilde{B}_2, ..., \tilde{B}_n \vdash \alpha.$$

Будем последовательно сокращать список гипотез, применяя следующий прием. Сгруппируем интерпретации попарно так, чтобы в каждой паре они отличались только противоположным значением пропозициональной буквы B_k. Тогда утверждения о выводимости в каждой паре интерпретаций имеют вид:

$$\tilde{B}_1, \tilde{B}_2, ..., \tilde{B}_{k-1}, B_k \vdash \alpha, \tilde{B}_1, \tilde{B}_2, ..., \tilde{B}_{k-1}, \neg B_k \vdash \alpha.$$

По теореме о дедукции перепишем эти утверждения в виде

$$\tilde{B}_1, \tilde{B}_2, ..., \tilde{B}_{k-1} \vdash B_k \to \alpha, \tilde{B}_1, \tilde{B}_2, ..., \tilde{B}_{k-1} \vdash \neg B_k \to \alpha.$$

По правилу простой конструктивной дилеммы (см. стр. 13) имеем

$$\tilde{B}_1, \tilde{B}_2, ..., \tilde{B}_{k-1} \vdash \alpha.$$

Повторяя этот прием, получим

$$\vdash \alpha.$$

2. Пусть формула α выводима в исчислении высказываний. Покажем, что она общезначима. Рассмотрим вывод α:

$$\beta_1, \beta_2, ..., \beta_n,$$
$$\beta_n = \alpha$$

Покажем по индукции, что все β_i общезначимы. Если β_i — аксиома, то она общезначима. Если β_i получена по правилу из общезначимых формул, она общезначима. Отсюда и общезначима α. □

Логическое исчисление называется *непротиворечивым*, если в нем нельзя одновременно вывести формулу $\neg\alpha$ и формулу α.

Следствие. Исчисление высказываний непротиворечиво.

Доказательство. Пусть в исчислении высказываний имеется такая формула α, что $\vdash \alpha$ и $\vdash \neg\alpha$. Тогда по теореме о полноте формулы α и $\neg\alpha$ являются общезначимыми, что невозможно. □

В исчислении предикатов нельзя говорить о полноте в таком же смысле, что и в исчислении высказываний. Поэтому говорят о полноте в широком и в узком смысле. Логическая система *полна в широком смысле*, если всякая тождественно истинная формула выводима в этой системе. Логическая система называется *полной в узком смысле*, если нельзя без противоречия присоединить к ее аксиомам в качестве новой аксиомы никакую не выводимую в ней формулу так, чтобы полученная при этом система была непротиворечива. В отличие от исчисления высказываний для исчисления предикатов существует недоказуемая там формула, которую без противоречия можно присоединить к системе аксиом:

$$\exists x(F(x)) \to \forall x(F(x)).$$

Поэтому для исчисления предикатов теорема о полноте выглядит иначе.

Теорема 1.6. (Теорема о полноте исчисления предикатов, теорема Геделя) Всякая тождественно истинная формула α выводима в исчислении предикатов.

Доказательство Вы можете найти в [37]. Таким образом, исчисление высказываний полно как в широком, так и в узком смысле, а исчисление предикатов полно только в широком смысле.

1.3.7 Логическое следствие

Неполнота исчисления предикатов заставляет искать способ получения следствий в этой теории, который можно было бы применить на практике.

Определение 1.6. Формула β называется логическим следствием формулы α (или β логически следует из α) тогда и только тогда, когда для всякой интерпретации, на которой формула α истинна, β тоже истинна.

Из определения можно сделать вывод: если формула β называется логическим следствием формулы α, то можно построить таблицу истинности:

α	β	β является логическим следствием α
ложь	ложь	истина (если α ложь, то значение β может быть любым)
ложь	истина	истина
истина	ложь	ложь
истина	истина	истина

Полученная таблица совпадает с таблицей истинности логической операции "$\rightarrow$", поэтому если формула β называется логическим следствием формулы α, то можно записать $\alpha \rightarrow \beta = true$.

Очевидно, что любая формула является логическим следствием самой себя, причем наиболее сильным следствием. Наоборот, слабым следствием любой формулы является тавтология — тождественно истинная формула (или частный случай такой формулы — константа $true$). С другой стороны, из ложных фактов можно вывести любое утверждение.

На основании определения можно также сделать вывод, что логическим следствием формулы α является $false$ тогда и только тогда, когда формула невыполнима. При этом, если формула α невыполнима, то кроме $false$ из нее можно получить любые следствия.

Рассмотрим определение логического следствия не из одной формулы, а из множества формул. Естественно считать, что на истинность следствия должна оказывать влияние истинность всех исходных формул одновременно. Поэтому имеет смысл в качестве оценки истинности рассмотреть конъюнкцию посылок.

Определение 1.7. Формула β называется логическим следствием формул α_1, α_2,..., α_n (или β логически следует из α_1, α_2,..., α_n) тогда и только тогда, когда для всякой интерпретации, на которой формула $\alpha_1 \& \alpha_2 \& ... \& \alpha_n$ истинна, β тоже истинна.

Понятие логического следствия лежит в основе машинных алгоритмов построения выводов. Одним из способов получения выводов является метод резолюций. Предложенный в 1965 году Дж. Робинсоном этот метод позволяет получить максимально сильное следствие множества формул с помощью систематической процедуры последовательного построения логических следствий. Метод использует тот факт, что если некоторая формула является невыполнимой, то наиболее сильное следствие этой формулы — константа $false$.

Теорема 1.7. Пусть β — логическое следствие формулы α. Тогда $\alpha \& \beta = \alpha$.

Доказательство. Пусть условие теоремы выполнено, тогда

$$\alpha \rightarrow \beta = true,$$

следовательно

$$\alpha = \alpha \& true = \alpha \& (\alpha \rightarrow \beta) = \alpha \& (\neg\alpha \vee \beta) = \alpha \& \neg\alpha \vee \alpha \& \beta =$$
$$= false \vee \alpha \& \beta = \alpha \& \beta$$

□

Определение 1.8. Резольвентой двух дизъюнктов $D_1 \vee L$ и $D_2 \vee \neg L$ называется дизъюнкт $D_1 \vee D_2$.

Теорема 1.8. Резольвента является логическим следствием порождающих ее дизъюнктов.

Доказательство. Пусть даны два дизъюнктов $D_1 \vee L$ и $D_2 \vee \neg L$. В соответствии с определением логического следствия из множества формул рассмотрим формулу $(D_1 \vee L) \& (D_2 \vee \neg L)$, логическое следствие из которой нам нужно рассмотреть. По определению $D_1 \vee D_2$ является логическим следствием дизъюнктов, если при истинности $(D_1 \vee L) \& (D_2 \vee \neg L)$ формула $D_1 \vee D_2$ истинна. Проверим этот факт, воспользовавшись проверкой общезначимости выражения при любой интерпретации

$$((D_1 \vee L) \& (D_2 \vee \neg L)) \rightarrow (D_1 \vee D_2) =$$
$$= (D_1 \& D_2 \vee D_1(\neg L) \vee D_2 \& L \vee L \& (\neg L) \rightarrow (D_1 \vee D_2) =$$
$$= (D_1 \& D_2 \vee D_1(\neg L) \vee D_2 \& L) \rightarrow (D_1 \vee D_2) =$$
$$= \neg(D_1 \& D_2 \vee D_1(\neg L) \vee D_2 \& L) \vee (D_1 \vee D_2).$$

Общезначимость указанного выражения эквивалента невыполнимости его отрицания. Проверим это:

$$\begin{aligned}
&\neg(\neg(D_1\&D_2 \vee D_1(\neg L) \vee D_2\&L) \vee (D_1 \vee D_2)) = \\
&= (D_1\&D_2 \vee D_1(\neg L) \vee D_2\&L)\&\neg(D_1 \vee D_2) = \\
&= (D_1\&D_2 \vee D_1(\neg L) \vee D_2\&L)\&(\neg D_1\&\neg D_2) = \\
&= D_1\&D_2\&\neg D_1\&\neg D_2 \vee D_1(\neg L)\&\neg D_1\&\neg D_2 \vee D_2\&L\&\neg D_1\&\neg D_2 = \\
&= false \vee false \vee false = false.
\end{aligned}$$

Тождественная ложность выражения доказывает теорему. □

Резолюция — это правило вывода, говорящее о том, как одна формула может получена из другой. Рассмотрим классический пример — одни из первых примеров в истории математики, который иллюстрирует применение принципа резолюции.

Дано: 1) Каждый человек смертен. 2) Сократ — человек.

Доказать: Сократ смертен.

Доказательство проводим от противного: предположим, неверно утверждение о смертности Сократа. Тогда имеем два дизъюнкта:

$$\forall x(\text{ Человек}(x) \to \text{Смертен}(x)) \text{ или } \forall x(\neg \text{ Человек}(x) \vee \text{Смертен}(x))$$
$$\neg\text{Смертен(Сократ) или } false \vee \neg\text{Смертен(Сократ)}.$$

В соответствии с обозначениями в определении резольвенты

$$\begin{aligned}
D_1 &= \neg \text{ Человек(Сократ)} \\
D_2 &= false \\
L &= \text{Смертен(Сократ)}
\end{aligned}$$

На основе принципа резолюции заключаем, что логическим следствием является

$$\neg \text{ Человек(Сократ)} \vee false = \neg \text{ Человек(Сократ)},$$

что противоречит условию 2.

Метод резолюций есть фактически правило присоединения к рассуждению, в состав которого входят два утверждения $A \to B$ и $\neg A \to C$, их следствия — утверждения $B \vee C$.

На основе принципа резолюции можно автоматизировать процесс доказательства. Важно только иметь формулы в той форме, к какой применим принцип резолюции наиболее эффективно и просто. Нужно формулу представить в конъюнктивной форме и конъюнктивно присоединять к ней всевозможные резольвенты ее дизъюнктов, а затем и получаемые в процессе доказательства резольвенты. Можно показать, что число резольвент конечно. Если, перебрав все возможные резольвенты, мы получили следствие $false$, то исходная формула невыполнима. В противном случае формула не является невыполнимой. Именно этот случай и демонстрирует пример о Сократе.

Метод резолюций имеет одно важное формальное свойство: он является полным для доказательства несовместности множества дизъюнктов. Это значит, что если множество дизъюнктов несовместно, то используя метод резолюций *всегда* можно вывести из данного множества дизъюнктов пустой дизъюнкт.

Метод резолюций разрабатывался применительно к формулам в некоторой стандартной форме. Рассмотрим эту форму.

1.3.8 Сколемовская нормальная форма и метод резолюций

Из Булевой алгебры известно, что любую формулу алгебры логики можно привести к дизъюнктивной нормальной форме, то есть представить ее в виде дизъюнкции конъюнкций

$$K_1 \vee K_2 \vee ... \vee K_m.$$

Это означает, что любую формулу исчисления высказываний можно представить в виде дизъюнкта.

Рассмотрим алгоритм приведения любой формулы исчисления предиктов к специальной стандартной форме, в которой использованы только операции конъюнкции, дизъюнкции и отрицания, а кванторы стоят в определенном порядке. Процесс приведения предиката к стандартному виду состит из шести основных этапов.

Этап 1 — исключение импликаций.

Процедура реализуется путем замены всех вхождений "$\rightarrow$" по правилу

$$\alpha \rightarrow \beta = \neg\alpha \vee \beta.$$

Этап 2 — перенос отрицаний внутъ формулы.

На этом этапе обрабатываются все случаи применения отрицания к формулам, не являющимися атомарными. Правила переноса отрицания внуть имеют вид

$$\begin{aligned}
&\neg(\alpha\&\beta) = (\neg\alpha) \vee (\neg\beta),\\
&\neg(\alpha \vee \beta) = (\neg\alpha)\&(\neg\beta),\\
&\neg(\forall x\alpha) = \exists x(\neg\alpha),\\
&\neg(\exists x\alpha) = \forall x(\neg\alpha).
\end{aligned}$$

После выполнения второго этапа каждое отрицание будет относиться лишь к атомарным формулам. Будем называть атомарную формулу или ее отриципие *литералом.* На всех последующих этапах литералы обрабатываются как единый элемент, а то, какие литералы представлены отрицинием, будет существенно лишь в самом конце.

Этап 3 — сколемизиция.

Удаляются кванторы существования. Это делается путем введения новых констант — *сколемовских констант* — вместо перемнных, связанных квантором существования. Вместо того, чтобы говорить, что существует объект, обладающий некоторым множеством свойств, можно ввести имя для такого объекта и просто сказать, что он обладает данными свойствами. Эти соображения лежат в основе введения сколемовских констант. Сколемизация более существенно изменяет логические свойства формулы, чем все обсуждавшиеся ранее преобразования. Тем не менее, она обладает следующим важным свойством. *Если имеется формула, то интерпретация, в которой эта формула истинна, существует тогда и только тогда, когда существует интерпретация, в которой истинна формула, полученная из первой в результате сколемизации.* Такая форма вполне достаточна для целей получения следствий.

Однако, простой прием замены переменной, связанной квантором существования, на обычную простую констану не всегда дает верный результат.

С целю иллюстрации возникающих при простой замене сложностей рассмотрим два примера. Пусть предикат $W(x)$ означает "x — женщина", предикат $A(x)$ означает "x — мужчина", а предикат $M(x, Y)$ — человек x является матерью человека Y. Тогда справедлива следующая формула

$$\exists x\ (W(x)\&M(x,\ \text{Иван})).$$

При сколемизации формулы получим

$$W(t_1)\&M(t, \text{Иван}),$$

где t_1 — некоторая константа, не использовавшаяся ранее. Полученная формула означает, что t_1 —некоторая женщина, являющаяся матерью Ивана.

Рассмотрим теперь другой пример:

$$\forall x\ (A(x) \to \exists y\ M(y, x)).$$

Эта формула означает, что у каждого мужчины существует мать. Заменим теперь вхождение связанной квантором существования переменной y на константу t_2 и получим формулу

$$\forall x\ (A(x) \to M(t_2, x)).$$

В результате получили, что у всех мужчин одна и та же мать t_2. Поэтому при сколемизации при наличии в формуле переменных, связанных квантором общности, необходимо вводить не константы, а *составные термы* — функциональные символы с множеством переменных аргументов. Это делается для того, чтобы отразить зависимость объекта, о существовании которого идет речь, от этих связанных квантором всеобщности переменных. В нашем примере должно получиться

$$\forall x\ (A(x) \to M(t_2(x), x)).$$

В этом случае функциональный символ t_2 соответствует функции, которая каждому мужчине ставит в соответствие его мать.

Константы и функции, которые использовались для замены переменных квантора существования, называются *сколемовскими константами и функциями.*

Этап 4 — Вынесение кванторов всеобщности в начало формулы.

Этот этап очень прост. каждый квантор всеобщности просто выносится в начало формулы, что не влияет на значение формулы.

Так как теперь каждая переменная в этой формуле вводится посредством квантора общности, находящегося в начале формулы, то кванторы сами по себе больше не несут никакой дополнительной информации. Поэтому можно сократить длину формулы, опустив кванторы. Необходимо только помнить, что каждая переменная вводится посредством указанного неявно квантора всеобщности, который опущен при записи формулы.

Этап 5 — использование дистрибутивных законов для & *и* $\vee$.

На этом этапе исходная формула уже претерпела много изменений: формула больше не содержит в явном виде кванторов, а из логических операций в ней остались лишь & и $\vee$ (знаки отрицания "$\neg$" мы считаем входящими в состав литералов). Теперь формулу преобразуем к конъюнктивной нормальной форме, характерной тем, что дизъюнктивные члены не содержат внутри себя коньъюнкцию. Из алгебры логики известно, что это всегда можно сделать. Напомним, что для этого достаточно использовать два тождества:

$$(A\&B) \vee C \text{ эквивалентно } (A \vee C)\&(B \vee C),$$
$$A \vee (B\&C) \text{ эквивалентно } (A \vee B)\&(A \vee C).$$

Этап 6 — выделение множества дизъюнктов.

После выполнения предыдущих этапов формула представляет собой совокупность конъюнктивных членов, каждый из которых — либо один литерал, либо литералы,

соединенные дизъюнкциями. Очевидно, что в этом выражении можно опустить все лишние скобки, оставив в скобках только дизъюнктивные элементы. Например, если в результате преобразований 1 — 5 получилось выражение

$$(A \vee B)\&((C \vee D \vee A)\&((\neg A \vee F \vee D)\&(D \vee A))),$$

то это выражение можно переписать без лишних скобок

$$(A \vee B)\&(C \vee D \vee A)\&(\neg A \vee F \vee D)\&(D \vee A).$$

Более того, теперь нет необходимости указывать знак конъюнкции, достаточно перечислить дизъюнкты:

$$(A \vee B)$$
$$(C \vee D \vee A)$$
$$(\neg A \vee F \vee D)$$
$$(D \vee A).$$

Если внутри дизъюнктов имеются скобки, их также можно опустить.

После всех вышеперечисленных преобразований формула представлена в записи, которая соответствует *сколемовской нормальной форме.* Если теперь по списку дизъюнктов восстановить формулу в правильной записи (т. е. восстановить знаки коньюнкции между дизъюнктамии, в начале формулы указать все кванторы всеобщности, которые там были), то полученная формула и будет формулой в сколемовской нормальной форме. Мы доказали, что к сколемовской нормальной форме приводима любая формула исчисления предикатов. Еще раз перечислим требования, которым удовлетворяет сколемовская нормальная форма:

— кванторы всеобщности стоят в начале формулы, а кванторы существования отсутствуют;

— часть формула, стоящая после кванторов существования, представляет собой конъюнкцию дизъюнктов, каждый из которых представляет дизъюнкцию литералов.

Теперь, когда мы имеем способ, позволяющий представлять предикаты в такой привлекательной форме, можно исследовать вопрос о логический следствиях. Точнее, выяснить, что следует из заданной совокупности предикатов, к каким следствиям они приводят.

1.3.9 Логическое программирование

Высказыания, которые исходно считаются истинными, называются гипотезами, а высказывания, которые следуют из гипотез, можно считать теоремами. Отсюда возникает путь к автоматическому построению вывода. Именно эта область научной деятельности дала жизнь идеям, положенным в основу логического программирования вообще и языка Пролог в частности.

Рассмотрим предикат в сколемовской нормальной форме и запишем множество дизъюнктов, каждый из которых представляет собой множество литералов. Договоримся записывать дизъюнкты последовательно один за другим, помня при этом, что парядок записи не имеет значения.

Литерал — либо атомарная формула, либо отрицание атомарной формулы. Примем соглашение записывать сначала литералы без отрицания, а затем литералы с отрицанием. Эти две группы литералов будем разделять знаком ":–". Литералы будем разделять запятыми. Будем считать, что в каждом дизъюнкте не более одного литерала без отрицания. Такие дизъюнкты называются *хорновскими дизъюнктами.*

При записи хорновских дизъюнктов количество атомарных формул слева от знака ":–" не может превышать единицу. И описанный выше метод описания хорновских дизъюнктов фактически представляет собой программу на Прологе:

1) вопрос в Прологе соответствует хорновскосу дизъюнкту без заголовка;

2) утверждения программы на Прологе соответствуют хорновским дизъюнктам с заголовком.

Пролог–система основывается на процедуре доказательства методом резолюций для хорновских дизъюнктов. Процедура начинается с применения правила резолюции к целевому дизъюнкту и к одной из гипотез, что дает новый дизъюнкт. Далее процесс продолжается и на каждом шаге правило резолюций применяется к последнему дизъюнкту из вновь полученных и к одной из исходных гипотез. Если необходимо, происходит конкретизация переменных.

1.4 Другие логические теории

1.4.1 Пороговая логика

Рассмотрим формальную модель нейрона — нервной клетки мозга — как переключающую функцию $\{0,1\}^n \to \{0,1\}$ в виде логической схемы, которая имеет один выход и n входов. Каждый вход x_i учитывается в нейроне с некоторым весом w_i. Нейрон возбуждается, если суммарное возбуждение его входов не меньше некоторого порога срабатывания h. Обозначим выходной сигнал нейрона y, тогда:

$$y = \begin{cases} 1, & \sum_i w_i x_i \geq h \\ 0, & \sum_i w_i x_i < h. \end{cases}$$

Изменение порога весов приводит к изменению функции, которую вычисляет нейрон. Рассмотрим, например, дизъюнкцию и конъюнкцию двух переменных x_1, x_2. Пусть $w_1 = w_2 = 1$, тогда со значением порога h сравнится значение выражения $x_1 + x_2$. Если $h = 1$, то нейрон реализует дизъюнкцию, а при $h = 2$ реализует конюнкцию:

x_1	x_2	значение при $h = 1$	значение при $h = 2$
0	0	$0 + 0 < 1 \to y = 0$	$0 + 0 < 2 \to y = 0$
0	1	$0 + 1 \geq 1 \to y = 1$	$0 + 1 < 2 \to y = 0$
1	0	$1 + 0 \geq 1 \to y = 1$	$1 + 0 < 2 \to y = 0$
1	1	$1 + 1 \geq 1 \to y = 1$	$1 + 1 \geq 2 \to y = 1$

На практике возникает обратная задача: для заданной функции найти значения весов входов и порог его срабатывания. Очевидно, что для решения этой задачи достаточно просто решить систему неравенств

$$\begin{cases} \sum_i w_i x_i \geq h, & \text{если значение функции равно 1} \\ \sum_i w_i x_i < h, & \text{если значение функции равно 0.} \end{cases}$$

Здесь каждое неравенство получается подстановкой значения аргументов и значения функции в соответствующее выражение. Рассмотрим, например, вычисление весов

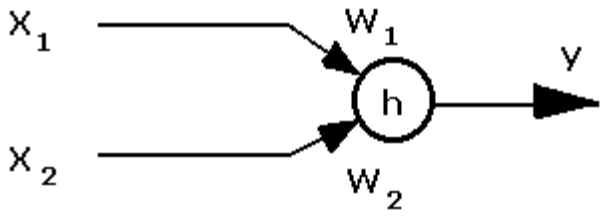


Рис. 1.1: Пример нейрона.

входов и порога для конъюнкции двух аргументов. Получим систему неравенств

$$\begin{cases} 0 * w_i + 0 * w_2 = 0 < h, & \text{так как } 0\&0 = 0 \\ 0 * w_i + 1 * w_2 = w_2 < h, & \text{так как } 0\&1 = 0 \\ 1 * w_i + 0 * w_2 = w_1 < h, & \text{так как } 1\&0 = 0 \\ 1 * w_i + 1 * w_2 = w_1 + w_2 \geq h, & \text{так как } 1\&1 = 1 \end{cases}$$

или

$$\begin{cases} 0 < h \\ w_2 < h \\ w_1 < h \\ w_1 + w_2 \geq h. \end{cases}$$

Решение этой системы неравенств: $h = 2$, $w_1 = 1$, $w_2 = 1$.

Если заданы все значения функции с n аргументами, то получим соответствующую систему из 2^n неравенств с $n + 1$ неизвестными. Такая система не всегда имеет решение, что означает отсутствие однокаскадного элемента с n входами. Рассмотрим, например, функцию $f(x_1, x_2) = x_1 \oplus x_2$ — функцию сложения по модулю 2. Система неравенств для этой функции имеет вид:

$$\begin{cases} 0 < h \\ w_2 \geq h \\ w_1 \geq h \\ w_1 + w_2 < h. \end{cases}$$

Система несовместна, следовательно, нейрон с двумя входами, реализующий эту функцию, не существует. Выходом в таком случае является построение многокаскадных схем.

1.4.2 K–значные логики

Конечнозначные логики вводятся как обобщение двузначной логики. В силу этого последующее изложение будет кратким, будут опускаться аналогичные определения и доказательства.

В k–значных логике аргумены функций определены на множестве $E_k = \{0, 1, ..., k-1\}$, значениями функций являются также элементы множества E_k. Очевидно, что функция полностью определена, если задана ее таблица

$x_1, x_2, ..., x_m$	$f(x_1, x_2, ..., x_m)$
$0, 0, ..., 0, 0$	$f(0, 0, ..., 0, 0)$
$0, 0, ..., 0, 1$	$f(0, 0, ..., 0, 1)$
...	...
$0, 0, ..., 0, k-1$	$f(0, 0, ..., 0, k-1)$
...	...
$k-1, k-1, ..., k-1, k-1$	$f(k-1, k-1, ..., k-1, k-1)$

Очевидно, что в k–значной логике существует k^{k^m} различных m–местных функций, а это значит, что при $k > 2$ существенно возрастают сложности эффективного задания функций с помощью таблиц. Можно выделить несколько функций, которые в k–значной логике можно считать элементарными:

1) $y = x + 1(mod\ k)$. Здесь y представляет собой обобщение отрицания в смысле циклического сдвига значений.

2) $\tilde{x} = k-1-x$ — другое обобщение отрицания в смысле зеркального отображения значений. функцию часто называют *отрицанием Лукашевича*.

3) Обобщение некоторых свойств отрицания при $i \neq k-1$:

$$J_i(x) = \begin{cases} k-1, \text{ при } x = i \\ 0, \text{ при } x \neq i \end{cases}$$

4) Еще один вид функции обобщение отрицания при $i \neq k-1$:

$$j_i(x) = \begin{cases} 1, \text{ при } x = i \\ 0, \text{ при } x \neq i \end{cases}$$

5) $min(x_1, x_2)$ — обобщение свойств конъюнкции.

6) $x_1 \cdot x_2(mod\ k)$— второе обобщение конъюнкции.

7) $max(x_1, x_2)$ — обобщение свойств дизъюнкции.

8) $x_1 + x_2\ (mod\ k)$.

Функции алгебры логики имеют в k–значной логике при $k > 2$ несколько аналогов, каждый из которых обобщает соответствующие свойства функции. Функции $min(x_1, x_2)$ и $max(x_1, x_2)$ будем иногда обозначать соответственно через $x_1 \& x_2$ и $x_1 \vee x_2$. Так же, как в алгебре логики, можно ввести понятие формулы над множеством функций. В силу свойства ассоциативности функций $min(x_1, x_2)$ и $max(x_1, x_2)$, а также соглашения о том, что операция & выполняется раньше операции $\vee$, можно при записи формул опускать некоторые скобки.

Во многом k–значные логики похожи на двузначную логику. В них сохраняются многие результаты, имеющие место в двузначной логике, правда рост значности все–таки приводит к усложнениям формулировок и доказательств. Однако, в k–значных логике при $k > 2$ наблюдаются явления, обнаруживающие принципиальное их отличие от алгебры логики. В связи с этим некоторые задачи не имеют такого исчерпывающего решения, как в алгебре логики, а другие вовсе не решены.

1.4.3 Нечеткая логика и приближенные рассуждения

В свое время появление формальной логики было шагом вперед в борьбе с неопределенностью, расплывчатостью представления человеческих знаний и рассуждений. По мере развития математики возникла необходимость создания теории, позволяющей формально описывать нестрогие понятия. Основой нечеткой логики является

отказ от основного утверждения классической теории множеств о том, что некотрый элемент может либо принадлежать, либо не принадлежать множеству. При этом вводится специальная характеристическая функция множества — *функция принадлежности*, которая принимает значения из интервала $[0,1]$. Этот способ приводит к континуальной логике, т.к. двухэлементное множество значений истинности $\{0,1\}$ двузначной логики расширяется до континуума $[0,1]$.

Нечеткое множество

$$A = \{< x, \mu_A(x) >\}$$

определяется как совокупность упорядоченных пар, составленных из элементов x множества X и соответствующих степеней принадлежности $\mu_A(x)$. Например, множество невысоких зданий мы можем определить с помощью функции принадлежности

$$\mu_A(x) = \frac{1}{h(x)},$$

где $h(x)$ — количество этажей здания x. Таким образом, в зависимости от значения функции принадлежности $\mu_A(x)$ можно говорить о "совершенно невысоком", "невысоком", "более или менее невысоком" или "не очень невысоком" здании.

Подобно нечеткому множеству можно ввести *нечеткое отношение* как функцию $f(x_i, x_2, ..., x_n)$, отображающую декартово произведение множеств $X_1 \times X_2 \times ... \times X_n$ в $[0,1]$:

$$f : X_1 \times X_2 \times ... \times X_n \to [0,1].$$

Пусть x_i — переменные со значениями из $[0,1]$. Можно ввести следующие нечеткие функции:

$$\neg x_i = 1 - x_i,$$
$$x_i \vee x_j = max\{x_i, x_j\},$$
$$x_i \& x_j = min\{x_i, x_j\}.$$

Нечеткая логическая формула определяет функцию, задающее отображение $[0,1]^*$ в $[0,1]$. Нечеткие формулы являются обобщением булевых функций, удовлетворяют всем аксиомам двузначного исчисления высказываний, кроме закона о дополнении: $x\&\neg x \neq 1$, $x \vee \neg x \neq 0$.

В качестве общезначимых (непротиворечивых) рассматривают такие формулы $f(x_1, x_2, ..., x_n)$, для которых для всех значений логических нечетких переменных x_i значение $T(f)$ функции f не меньше 0.5. Например, формула $x \vee \neg x$ общезначима:

$$T(x \vee \neg x) = max\{T(x), 1 - T(x)\} = \begin{cases} T(x), \text{ если } T(x) \geq 0.5 \\ 1 - T(x), \text{ если } T(x) < 0.5. \end{cases}$$

Таким образом, $T(x \vee \neg x)$ всегда не меньше 0.5 и формула $x \vee \neg x$ общезначима.

На нечеткую логику естественным образом распространяется метод резолюции, что позволяет проводить нечеткие рассуждения. Трактовка истинности приводит к возможности оперировать, например, значениями "совершенно истинный", "сомнительный", "более или менее истинный", "не очень истинный", "ложный" и тому подобными. Под приближенными рассуждениями понимается процесс, при котором из нечетких посылок с помощью правил вывода получаются некоторые следствия, тоже нечеткие. Приближенные вычисления лежат в основе способности человека понимать естественный язык, разбирать рукописный текст, играть в игры, принимать решения в сложной и неполностью определенной среде.

1.4.4 Темпоральная логика

Формулы обычной логики истинны или ложны в статическом мире и не зависят от времени. Многие дискретные системы меняют свое поведение во времени. Например, в программе переменная $factor$ перед циклом равна единице, а после его завершения равна 10!:

```
int n=10, factor = 1;
for (int i=1; i <= n; i++)
   factor *= i;
```

Зависимость от времени неявно имеется в фразах на естественном языке. Например, фраза

Васе стало страшно, и он убежал

не означает то же самое, что фраза

Вася убежал, и ему стало страшно.

В классической логике утверждения понимаются как истинные или ложные независимо от времени. Приведенный выше пример демонстрирует некоммутативность операции конъюнкции в ситуации, когда мы имеем дело с действиями во времени. Анализ подобных выражений в классической логике невозможен.

В темпоральных логиках истинность логических формул зависит от того момента времени, в который вычисляется значение этих формул. Если кванторы стали основой для расширения исчисления предикатов до исчисления высказываний, то темпоральные операторы являются основой для построения темпоральной логики. Рассмотрим основные определения современной линейной темпоральной логики Linear Time Logic (LTL) . Все формулы темпоральной логики интерпретируются на бесконечной дискретной направленной в будущее дискретной последовательности "миров", в каждом из которых существует своя интерпретация логических формул. В последовательности миров время можно считать изоморфным натуральному ряду 0, 1, 2..., т.е. все миры в цепочке можно считать пронумерованными, и в каждом мире логические переменные и предикаты принимают конкретные стабильные значения $true$ или $false$. Например, может так оказаться, что сегодня Вася является студентом, а завтра - уже нет, так как его отчислили за неуспеваемость. Рассмотрим базовые определения: рекурсивное определение атомарного предиката и определения темпоральных операторов.

Определение 1.9. Атомарным предикатом называется

- Элементарное утверждение;
- Атомарные предикаты, связанные логическими операторами $\neg$, $\rightarrow$, $\vee$, &;
- Атомарные предикаты, связанные темпоральными операторами.

Определение 1.10. Основными темпоральными операторами являются два оператора U и X. Оператор X — $NextTime$ — имеет один аргумент и означает, что утверждение Xq истинно в момент времени t, если утверждение q истинно в следующий момент. Оператор U — $Until$ — имеет два аргумента, и выражение pUq истинно в момент времени t, если q истинно в некоторый будущий момент t_1, а на всем промежутке от t до t_1 истинно p. Другими словами, смысл этого оператора можно представить словами

в будущем будет истинно q, а до него непрерывно будет истинно p.

Рассмотрим примеры:

• Завтра будет день $\Rightarrow X(\text{ДеньЗавтра})$

• Я не получу диплом, пока не сдам все экзамены

$\Rightarrow (\neg\text{ПолучитьДиплом})U(\text{СдатьВсеЭкзамены})$

Для оператора X существует его аналог в прошлом X^{-1}, соответствующий предыдущему моменту времени: утверждение $X^{-1}q$ истинно в момент времени t, если утверждение q было истинно в предшествующий момент. Для оператора U также существует аналог в прошлом — оператор $S = U^{-1}$ ($Since$ — с тех пор, как). Можно рассмотреть вспомогательные операторы G и F, а также их композицию и отрицание, которые выражаются через основные темпоральные операторы X и U:

• Gq – всегда в будущем,

• Fq – хотя бы раз в будущем,

• $\neg Fq$ – никогда в будущем,

• GFq – бесконечно много раз в будущем,

• FGq – с какого-то момента постоянно.

Очевидно, что эти операторы являются зависимыми от операторов X и U, так как можно записать:

$$Fq = trueUq$$

$$Gq = \neg(F(\neg q)) = \neg(trueU(\neg q))$$

Несмотря на тот факт, что вспомогательные темпоральные операторы выражаются через базовые, зависимые операторы F и G часто используются при записи выражений, так как их использование делает выражение более наглядным. Например:

• "Мы будем бороться, пока не победим (и победим обязательно)"

$\Rightarrow (\text{МыБоремся})U(\text{МыПобедили})$

• "Мы будем бороться, пока не победим (может быть, победа никогда не будет достигнута)"

$\Rightarrow (\text{МыБоремся})U(\text{МыПобедили}) \vee G(\text{МыБоремся})$

Интуитивно понятное определение выполнимости формулы темпоральной логики можно записать формально.

Определение 1.10. Формула α линейной темпоральной логики выполняется на последовательности миров $w0$, $w1$, $w2$..., если α истинна в начальном мире этой последовательности.

Привязка формулы к начальному моменту важна для технических систем, поскольку нам важно знать, будет ли справедливо в будущем некоторое свойство системы, которая в настоящий момент стартовала из начального состояния. Приведенное определение иногда заменяют другим, в соответствии с которым формула α выполняется на последовательности миров $w0$, $w1$, $w2$,..., если α истинна на всех мирах этой последовательности. Такое определение является более узким, потому что его легко можно выразить через определение (1.10): достаточно использовать темпоральный оператор $G\alpha$.

1.5 Контрольные вопросы к разделу

1. Зачем нужны формальные теории?
2. Как задается формальная теория?
3. Укажите различие между аксиомами и формулами математической теории.
4. Поясните суть финитного метода.

5. Дайте определение доказательства в математической теории.

6. Перечислите аксиомы исчисления высказываний.

7. Перечислите аксиомы исчисления предикатов.

8. Чем отличаютмя формулы исчисления высказываний от формул исчисления предикатов?

9. Докажите теорему о дедукции для исчисления предикатов.

10. Докажите теорему о дедукции для исчисления высказываний.

11. Как связаны кванторы существования и всеобщности?

12. Перечислите правила вывода исчисления предикатов. Чем эти правила отличаются от правил вывода исчисления высказываний?

13. Поясните понятие интерпретации математической теории.

14. Сформулируйте и докажите теорему о полноте исчисления высказываний.

15. Укажите правила преобразования предиката к сколемовсокой нормальной форме.

16. Приведите правила, которым должен удовлетворять предикат в сколемовсокой нормальной форме.

17. Что называется резольвентой?

18. Сформулируйте и докажите теорему о резольвенте.

19. Объясните метод резолюций и приведите пример его применения.

20. Как связан язык Пролог с исчислением предикатов?

21. Как можно вычислить веса входов нейрона, если заданы значении функции, которую должен вычислять нейрон?

22. Какие обобщения двузначной логики Вы знаете?

23. Как определяется конъюнкция в k - значной логике?

24. Что означает функция принадлежности в нечеткой логике?

25. Дайте определение выполнимости формулы в темпоральной логике.

26. Какие темпоральные операторы логики LTL Вы знаете?

27. Приведите пример атомарного предиката в линейной темпоральной логике.

1.6 Тесты для самоконтроля к разделу

1. Укажите определение вывода в формальной теории.

Варианты ответов:

1) Выводом формулы C в формальной теории называется произвольная последовательность формул C_1, C_2, C_3,..., где каждая из формул C_i — это либо аксиома формальной теории, либо непосредственное следствие каких–либо предыдущих формул этой последовательности в соответствиии с одним из правил вывода.

2) Выводом формулы C в формальной теории называется конечная последовательность формул C_1, C_2, ..., C_n, где $C_n = C$, а каждая из формул C_i для $i = 1, ..., n$ — непосредственное следствие каких–либо предыдущих формул этой последовательности в соответствиии с одним из правил вывода.

3) Выводом формулы C в формальной теории называется конечная последовательность формул C_1, C_2, ..., C_n, где $C_n = C$, а каждая из формул C_i для $i = 1, ..., n$ — это либо аксиома формальной теории, либо формула формальной теории, либо непосредственное следствие каких–либо предыдущих формул этой последовательности в соответствиии с одним из правил вывода.

4) Выводом формулы C в формальной теории называется конечная последовательность формул C_1, C_2, ..., C_n, где $C_n = C$, а каждая из формул C_i для $i = 1, ..., n$ —

это либо аксиома формальной теории, либо непосредственное следствие каких–либо предыдущих формул этой последовательности в соответствиии с одним из правил вывода.

5) Выводом формулы C в формальной теории называется произвольная последовательность формул C_1, C_2, C_3,..., где $C_n = C$, а каждая из формул C_i для $i = 1, ..., n$ — это либо аксиома формальной теории, либо формула формальной теории, либо непосредственное следствие каких–либо предыдущих формул этой последовательности в соответствиии с одним из правил вывода.

Правильный ответ: 4.

2. Как в нечеткой логике определяется $x \vee y$?

Варианты ответов:

1) $min\{x, y\}$,
2) $max\{x, y\}$,
3) $1 - xy$,
4) $x + y$,
5) xy.

Правильный ответ: 2.

3. Перечислите аксиомы исчисления высказываний, построенного с помощью операций импликации и отрицания.

Варианты ответов:

1) $A1)$ $\alpha \to (\beta \to \alpha)$,
$A2)$ $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$,
$A3)$ $(\neg\alpha \to \neg\beta) \to (\beta \to \alpha)$.

2) $A1)$ $\alpha \to \alpha$,
$A2)$ $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$,
$A3)$ $\neg(\alpha \to \beta) \to (\neg\beta \to \neg\alpha)$.

3) $A1)$ $\alpha \to (\beta \to \alpha)$,
$A2)$ $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$,
$A3)$ $(\neg\alpha \to \neg\beta) \to (\beta \to \alpha)$.
$A4)$ $\alpha \to \alpha$.

4) $A1)$ $\alpha \to \alpha$,
$A2)$ $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$,
$A3)$ $(\neg\alpha \to \neg\beta) \to (\beta \to \alpha)$.
$A4)$ $\alpha \to (\alpha \to \beta)$.

5) $A1)$ $\alpha \to \alpha$,
$A2)$ $(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$,
$A3)$ $\neg(\alpha \to \beta) \to (\neg\beta \to \neg\alpha)$.

Правильный ответ: 1.

4. Какие из перечисленных ниже утверждений являются правильными?

Варианты ответов:

1) В пороговой логике выходной сигнал нейрона определяется суммой входных сигналов.

2) Изменение порога весов приводит к изменению функции, которую вычисляет нейрон.

3) Для любой функции можно построить нейрон, который вычисляет эту функцию.

4) Если для функции трех аргументов веса равны $\{2, 2, 2\}$, а порог равен 3, то нейрон вычисляет функцию $x_1 \vee x_2 \vee x_3$.

5) Нейрон с пороговым значением 7 и весами $\{4, 5\}$ вычисляет функцию $x_1 \& x_2$.

Правильные ответы: 2, 5.

5. Сформулируйте теорему о дедукции исчисления высказываний.

Варианты ответов:

1) Пусть Γ — некоторый список формул и α — одна формула. Тогда если существует вывод $\Gamma, \alpha \vdash \beta$, в котором правило обобщения Gen не применяется ни по какой из переменных, свободных в формуле α, то существует вывод $\Gamma \vdash \alpha \to \beta$, в котором правило Gen применяется по тем же переменным, по которым оно применялось в исходном выводе $\Gamma, \alpha \vdash \beta$.

2) $\alpha \to \forall x\ \alpha$.

3) $\alpha(t) \to \exists x\ \alpha(x)$.

4) Формула α выводима в исчислении высказываний тогда и только тогда, когда она общезначима.

5) Пусть Γ — некоторый список формул и α — одна формула. Тогда если $\Gamma, \alpha \vdash \beta$, то $\Gamma \vdash \alpha \to \beta$.

Правильный ответ: 5.

Глава 2

ЧАСТИЧНО – РЕКУРСИВНЫЕ ФУНКЦИИ

2.1 Свойства алгоритмов

Под алгоритмом решения задач некоторого класса обычно понимается общее правило, с помощью которого решение любой конкретной проблемы этого класса может быть найдено чисто механически и без всяких дополнительных размышлений о процессе решения. Среди известных примеров – алгоритм Евклида нахождения наибольшего общего делителя для двух натуральных чисел, алгоритм деления двух целых чисел, алгоритм перевода целого десятичного числа в двоичную систему счисления и т.п. Для любой задачи алгоритм ее решения должен быть доказан. Даже если на некотором наборе тестов соответствующая программа для ЭВМ выдала правильный результат, принципиальное отсутствие доказательства правильности алгоритма может означать только одно: предлагаемый алгоритм решает какую–то другую задачу, возможно, являющуюся частным случаем поставленной задачи.

Слово "алгоритм" происходит от имени арабского математика Мохаммеда ибн Муса Альхваризми, который в IX столетии внес значительный вклад в разработку и практическое применение методов вычислений. Прогресс в развитии таких методов породил представление о том, что можно найти алгоритм решения любой поставленной математической и даже философской проблемы. Вера в универсальность алгоритмических методов была впервые подорвана работой Геделя, в которой была доказана алгоритмическая неразрешимость некоторых математических проблем. Точнее, было доказано, что известные математические проблемы не могут быть решены с помощью алгоритмов из некоторого, точно определенного класса алгоритмов. Значение результата Геделя при этом зависит от степени совпадения этого алгоритмического класса и класса всех алгоритмов в интуитивном смысле. Тем самым возникла совершенно новая ситуация. До тех пор, пока мы верили в возможность того, что все поставленные математические задачи могут быть алгоритмически решены, у нас не было повода уточнять понятие алгоритма, вводить его математическое определение и рассматривать строгие свойства алгоритмов. Когда для решения какого-то класса проблем предлагался алгоритм, возникало соглашение считать указанный алгоритм действительно алгоритмом. Только утверждение об алгоритмической неразрешимости требует строго определения, т.к. для такого доказательства требуется утверждение обо всех мыслимых алгоритмах.

С появлением ЭВМ проблема теоретического анализа алгоритмов стала еще актуальнее. Программа для ЭВМ представляет собой запись алгоритма решения задачи на каком–либо языке программирования. Для программиста программа — это после-

довательность действий, которые нужно выполнить, чтобы из исходных данных получить результат. Однако такое определение алгоритма не является математически строгим и, следовательно, не позволяет рассмотреть свойства алгоритмов как свойства некоторых формальных объектов. Практическая реализация алгоритмов в виде программы привела к необходимости сравнения качества алгоритмов. Стало важным не просто показать разрешимость или неразрешимость какой-либо проблемы, но и оценить возможности компьютера для практического решения поставленных задач.

Таким образом, необходимость рассмотрения математического определения алгоритма определяется несколькоми причинами.

Во–первых, *только при наличии формального определения алгоритма можно ставить задачу о разрешимости или неразрешимости каких–либо проблем.* Если мы хотим доказать, что та или иная функция не является эффективно вычислимой, то мы прежде всего должны сформулировать точное математическое понятие эффективной вычислимости. Главным результатом теории алгоритмов, как будет показано в дальнейшем, является доказательство существованиия некоторых неразрешимых проблем.

Во–вторых, *необходимо иметь математически точный инструмент для сравнения различных алгоритмов решения одних и тех же задач, а также для сравнения различных проблем по сложности алгоритмов их решения.* Такое сравнение невозможно без введения средств исследования алгоритмов как математических объектов и использования точного языка математики для их описания. Иначе говоря, сами алгоритмы должны стать такими же предметами точного исследования, как те объекты, для работы с которыми они предназначены.

Прежде, чем ввести математически строгое определение алгоритма, необходимо рассмотреть свойства тех объектов, которые мы считаем алгоритмами. С точки зрения современной практики алгоритм — это программа, а критерием алгоритмичности процесса является возможность его запрограммировать. Именно благодаря этой реальности алгоритма, а также потому, что подход программиста к математическим методам всегда основан на возможности их практической реализации, можно выделить следующие характерные свойства, которым удовлетворяет любой алгоритм.

1. *"Конечность."*

Любой алгоритм задается последовательностью инструкций конечных размеров. Например, конечную длину имеет любая программа для ЭВМ, любой математический алгоритм можно описать с помощью конечного числа русских слов.

2. *"Детерминированность."*

Алгоритм выполняется детерминированно, т.е. представляющая алгоритм последовательность действий на одних и тех же данных всегда выполняется одинаково. Вычисления выполняются детерминированно без обращения к случайным устройствам (например, игральным костям). Здесь следует отметить, что все программно реализованные датчики случайных чисел генерируют на самом деле псевдослучайную последовательность с достаточно большим периодом.

3. *"Дискретность."*

Отдельные инструкции алгоритма выполняются дискретно, т.е. последовательно по отдельным шагам, без использования каких–либо аналоговых устройств непрерывного принципа действия или соответствующих непрерывных методов.

4. *"Вычислимость" или "эффективная вычислимость."*

Должен существовать вычислитель, способный выполнить указанные в алгоритме инструкции. Здесь под вычислителем может пониматься любое существующее или реализуемое устройство, способное понимать инструкции алгоритма. Частным случаем такого устройства может являться компьютер или человек. Понятие "эф-

фективная вычислимость" означает практическую выполнимость алгоритма: "эффективный" означает "практически выполнимый". Этот термин часто сокращают до термина "вычислимость предполагая "по умолчанию" наличие определения "эффективная". В частности, свойство конечности алгоритма непосредственным образом связано с требованием вычислимости. Никакой вычислитель (человек — исполнитель алгоритма, компьютер или какое–либо другое устройство) не может использовать более чем конечное количество информации.

5. *"Конечная память."*

Вычислитель должен иметь средства для хранения и отображения информации. Память вычислителя может быть сколь угодно большой, но при каждом выполнении алгоритма требуется конечный объем памати.

6. *"Результативность."*

Алгоритм должен быть результативным, т.е. завершающимся с некоторым результатом после некоторого конечного числа шагов.

Таким образом, говоря неформально, алгоритм – это эффективная детерминированная конечная процедура, которую можно применять к любому элементу некоторого класса входов и которая для каждого такого входа дает в конце концов соответствующий результат. Анализ свойств алгоритмов показывает, что следует различать описание алгоритма и процесс выполнения алгоритма. Несмотрия на неточную формулировку, практически все программисты согласились бы с тем, что определение понятия алгоритма включает первые пять пунктов. Эти первые пять свойств должны выполняться для любого алгоритма и характеризуют его описание. В отличие от этих свойств последнее свойство — свойство результативности — характеризует не описание алгоритма, а процесс его выполнения. Легко можно написать работоспособную программу, содержащую бесконечный цикл. Таким образом, описание алгоритма всегда конечно, но процесс выполнения может быть бесконечным, поэтому свойство результативности алгоритма является только желательным, но не обязательным свойством алгоритмов. Более того, как будет показанов в дальнейшем, общего метода проверки результативности алгоритма, применимого к любому алгоритму A и любым исходным данным x, вообще не существует.

Можно выделить три основных типа универсальных алгоритмических моделей, различающихся исходными эвристическими соображениями относительно того, что такое алгоритм. В зависимости от того, что берется за основу формализации, рассмотрим следующие подходы к определению понятия алгоритма.

1) *Частично–рекурсивные фунции.* Модель основана на функциональном подходе и рассматривает понятие алгоритма с точки зрения того, что можно вычислить с помощью алгоритма. Эта наиболее развитая и изученная модель является исторически первой формализацией понятия алгоритма и связывает определение алгоритма с наиболее традиционными понятиями математики — вычислениями и числовыми функциями.

2) *Машины Тьюринга.* Эта автоматная модель имеет в своей основе анализ процесса выполнения алгоритма и рассматривает алгоритм в виде набора инструкций для некоторой формальной модели вычислителя. Данный тип основан на представлении об алгоритме как о некотором детерминированном устройстве, способном выполнять в каждый отдельный момент лишь весьма примитивные операции. Такое представление не оставляет сомнений в однозначности алгоритма и элементарности его шагов. Кроме того, теоретическая основа этих моделей очень близка к ЭВМ.

3) *Алгоритмы Маркова.* Представляет собой языковый подход к понятию алгоритма, в основе которого лежит формализация процесса преобразования записей исходных данных в запись результатов. Всякие исходные данные для некоторой

частной проблемы из какого–нибудь общего класса проблем могут быть сформулированы в виде некоторого выражения подходящего языка. Всякое решение этой проблемы в свою очередь также является выражением того или иного языка. Тогда алгоритм можно понимать как способ преобразования записи исходных данных в запись результатов. Преимущества такого типа формализации алгоритма — в его максимальной абстрактности и возможности применить понятие алгоритма к объектам произвольной природы, а не обязательно числовой, как это требуется для функциональной модели. Впрочем, как мы увидим в дальнейшем, модели второго и третьего типа довольно близки на формальном уровне и различаются в основном только эвристическими признаками. На протяжении всей книги мы будем иметь дело с неотрицательными целыми числами, множествами неотрицательных целых чисел и отображениями неотрицательных целых чисел в неотрицательные целые числа. Если только не оговорено противное, мы используем термин "натуральное число" или просто "число" для обозначения неотрицательного целого числа.

Будем рассматривать формальное определение алгоритмов, выполняющих действия на множестве целых неотрицательных чисел — на самом простом множестве, на котором можно исследовать поведение алгоритмов. Как мы увидим, это ограничение числовыми функциями не приводит к потере общности.

Такой подход не снижает общности понятия алгоритма при условии, что любой алгоритм обрабатывает исходные данные конечной длины, причем эти данные представлены в конечном алфавите. Ограничение числовыми функциями также не приводит к потере общности, т.к. при условии конечности алфавита и длины исходной информации эту информацию всегда можно представить некоторым натуральным числом — ее номером. Например, любая программа для ЭВМ получает на вход последовательность байтов, которую можно считать одним очень длинным натуральным числом.

Теория алгоритмов строится на основе *финитного метода.* Опыт парадоксов теории множеств научил математику крайне осторожно обращаться с бесконечностью и по возможности даже о бесконечности рассуждать с помощью конечных методов. Существо финитного подхода заключается в том, что он допускает только конечные действия над конечными объектами. Выяснение того, какие объекты и действия над ними следует считать точно определенными, какими свойствами и возможностями обладают комбинации элементарных действий — все это является предметом теории алгоритмов и формальных систем.

2.2 Примитивно–рекурсивные функции

Всякий алгоритм для любых исходных данных однозначно определяет некоторый результат, если, конечно, этот результат существует для них. Поэтому каждому алгоритму соответствует функция, которая вычисляет этот результат. Поставим цель дать формальное определение произвольного алгоритма как функции, принадлежащей некоторому специальным образом построенному классу функций. Для того, чтобы определение такого класса функций было строго формальным, необходимо выбрать способ его задания. Воспользуемся конструктивным методом определения и опишем некоторый класс функций с помощью рекурсивных определений. Основным свойством конструктивного подхода является то, что все множество исследуемых объектов (в данном случае функций) строится из конечного числа исходных объектов — базиса — с помощью простых операций, эффективная выполнимость которых

достаточно очевидна. Операции над функциями в дальнейшем будем называть операторами.

Рассмотрим сначала набор *простейших функций,* для которых с очевидностью существуют алгоритмы их вычисления, а затем введем специальные операторы над числовыми функциями, с помощью которых можно будет из имеющихся функций получать новые функции.

В качестве простейших функций будем использовать следующие три функции:

1) $0(x_1, \ldots, x_n) = 0$ — нуль–функция,
2) $s(x) = x + 1$ — функция следования,
3) $I_m(x_1, \ldots, x_n) = x_m$ — функция выбора (или тождества).

Сразу отметим, что нуль–функция и функция выбора могут иметь разное число аргументов: $0^n(x_1, \ldots, x_n)$ и $I_m^n(x_1, \ldots, x_n)$.

Рассмотрим операторы, позволяющие конструировать новые функции из уже имеющихся. Введем в рассмотрение два оператора – оператор суперпозиции и оператор примитивной рекурсии. *Оператор суперпозиции* из функций

$$f(x_1, \ldots, x_m),$$
$$f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n)$$

строит новую функцию

$$g(x_1, \ldots, x_n) = f(f_1(x_1, \ldots, x_n), \ldots f_m(x_1, \ldots, x_n)). \tag{2.1}$$

Обозначим оператор суперпозиции через $S(f, f_1, \ldots, f_m)$ или, с явным указанием числа аргументов функций, $S_m^n(f, f_1, \ldots, f_m)$. Благодаря наличию функций выбора, стандартное представление суперпозиции с точным определением числа аргументов у всех функций f_1, f_2,..., f_m не уменьшает возможностей самого оператора суперпозиции, т.к. любую подстановку функции в функцию можно выразить через оператор S и функцию I. Наприер, для трех функций

$$h(x, y) = x + y, \;\; f(x) = 3x - 1, \;\; g(x, y, z) = \frac{x}{y + z}$$

выражение

$$h(f(y), g(x, y, z)) = 3y - 1 + \frac{x}{y + z}$$

можно представить в виде стандартной суперпозиции $h(f(I_2^3(x, y, z)), g(x, y, z))$.

Следует отметить, что при наличии нуль–функции и функции следования, а также оператора суперпозиции можно построить все множество натуральных чисел:

$$0 = 0(x), 1 = s(0(x)), 2 = s(s(0(x))), \ldots.$$

Рассмотрим *оператор примитивной рекурсии.* Известно, что рекурсия — это способ вычисления значения функции в текущей точке через значение этой же функции в некоторых предыдущих точках. В зависимости от того, какие предшествующие точки используются при вычислении значения в текущей точке, рассматривают различные схемы рекурсии.

Определение 2.1. Функция $f(x_1, ..., x_n, y)$ получается оператором примитивной рекурсии из функций $g(x_1, \ldots, x_n)$ и $h(x_1, \ldots, x_n, y, z)$, если

$$\begin{aligned} &f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n), \\ &f(x_1, \ldots, x_n, y + 1) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)). \end{aligned} \tag{2.2}$$

Обозначим оператор примитивной рекурсии как $R(g,h)$. Схема (2.2) называется схемой примитивной рекурсии. Например, при задании примитивно–рекурсивного описания функции одной переменной, схема примитивной рекурсии имеет вид:

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

где a — константа, принадлежащая множеству N.

Для того, чтобы в некоторой точке $(x_1, \ldots, x_n, y)$ вычислить значение функции, заданной оператором примитивной рекурсии, можно выполнить следующую конечную последовательность вычислений:

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n),$$
$$f(x_1, \ldots, x_n, 1) = h(x_1, \ldots, x_n, 0, f(x_1, \ldots, x_n, 0)),$$
$$\ldots$$
$$f(x_1, \ldots, x_n, y) = h(x_1, \ldots, x_n, y-1, f(x_1, \ldots, x_n, y-1)).$$

Существенной характеристикой оператора примитивной рекурсии является такое его важнейшее свойство, что независимо от числа аргуменов функции f рекурсия ведется только по одному аргументу, остальные аргументы зафиксированы на момент применения рекурсии. Очевидно, что реализовать программное вычисление функции, заданной оператором примитивной рекурсии, можно рекурсивным способом. Программа вычисления функции на языке Паскаль имеет вид:

```
function   f(x1,...xn,y: integer): integer;
begin if y=0 then f:= g(x1,..., xn)
             else f:= h(x1,..., xn,y-1,f(x1,...,xn,y-1))
end;
```

Или на языке Си:

```
int  f(int x1,..., int xn,int y)
{if (y==0) return  g(x1,..., xn);
return h(x1,..., xn,y-1,f(x1,...,xn,y-1));
}
```

Можно использовать итерационный метод вычисления этой же функции. Для этого в программе достаточно организовать цикл, который выполняется столько раз, какова глубина вызова рекурсивной функции.

```
function   f(x1,...xn,y: integer): integer;
var t,i: integer;
begin t:= g(x1,..., xn);
      for i:=0 to y-1 do  t:= h(x1,..., xn, i, t);
      f:=t;
end;
```

Или соответственно на языке Си:

```
int   f(int x1,...,int xn,int y)
{int t;
t= g(x1,..., xn);
for ( int i=0; i<y; i++ ) t= h(x1,..., xn, i, t);
return t;
}
```

Обратите внимание, что цикл выполняется для переменной i, значение которой равно значению связанного рекурсией аргумента в *предшествующей точке*. Эти значения равны 0, 1, ..., $y-1$.

Определение 2.2. Функция называется примитивно – рекурсивной, если она может быть получена из простейших функций с помощью конечного числа операторов суперпозиции и примитивной рекурсии.

Данное определение эквивалентно рекурсивному заданию множества функций. Действительно, простейшие функции являются примивно–рекурсивными по определению. Если некоторые функции являются примитивно–рекурсивными, то в результате применения к ним одного из операторов суперпозиции или примитивной рекурсии получаем новые примитивно–рекурсивные функции. Конечное число операторов суперпозиции или примитивной рекурсии, которые применяются при построении примитивно–рекурсивных функций, гарантируют завершение указанного рекурсивного процесса. Таким образом, можно сформулировать рекурсивный аналог определения 2.2.

Рекурсивный вариант определения 2.2. Функция называется примитивно–рекурсивной, если

а) она является простейшей,

б) она получена из примитивно–рекурсивных функций с помощью оператора суперпозиции;

в) она получена из примитивно–рекурсивных функций с помощью оператора примитивной рекурсии.

Других примитивно–рекурсивных функций нет.

Теорема 2.1. Примитивно рекурсивные функции всюду определены.

Доказательство. В соответствии с рекурсивным вариантом определения примитивно–рекурсивной функции достаточно рассмотреть три шага доказательства. Очевидно, что простейшие функции всюду определены. Из определенных всюду функций с помощью оператора суперпозиции можно получить только всюду определенные функции. Из определенных всюду функций в соответствии с алгоритмом вычисления функций, заданных оператором примитивной рекурсии, также получаем всюду определенные функции. □

Для доказательства примитивной рекурсивности некоторой заданной функции можно воспользоваться определением примитивно–рекурсивных функций, следовательно, существуют три направления поиска такого доказательства:

– показать, что заданная функция является простейшей,

– или показать, что заданная функция построена из примитивно–рекурсивных функций с помощью оператора суперпозиции,

– или показать, что заданная функция построена из примитивно–рекурсивных функций с помощью оператора примитивной рекурсии.

Пример. Доказать примитивную рекурсивность функции

$$f(x,y) = x + y.$$

Рассмотрим способ рекурсивного определения этой функции:

$$\begin{aligned} &f(x,0) = x, \\ &f(x,y+1) = x + y + 1 = f(x,y) + 1. \end{aligned}$$

Для того, чтобы показать соответствие полученной для данной функции рекурсивной формулы схеме примивной рекурсии, достаточно воспользоваться функцией

выбора и функцией следования:

$$f(x,0) = x = I(x),$$
$$f(x,y+1) = f(x,y) + 1 = s(f(x,y)) = s(I_3^3(x,y,f(x,y))).$$

Аналогично можно доказать примитивную рекурсивность функций xy, $x!$, $sg(x)$, $\overline{sg}(x)$, $x \dot{-} y$ и т.д. Здесь символ $\dot{-}$ используется для обозначения усеченной разности:

$$x \dot{-} y = \begin{cases} 0, & \text{если } x < y \\ x - y, & \text{если } x \geq y \end{cases}$$

Функция знака $sg(x)$ равна нулю при $x = 0$ и единице во всех остальных точках $x \in N$:

$$sg(x) = \begin{cases} 0, & x = 0 \\ 1, & x > 0 \end{cases}$$

Другая знаковая функция $\overline{sg}(x)$ возвращает отрицание знака:

$$\overline{sg}(x) = \begin{cases} 1, & x = 0 \\ 0, & x > 0. \end{cases}$$

Как уже отмечалось, суперпозицию функции $f(\overline{x}, y)$ и функций произвольного вида, не содержащих $f(\overline{x}, y)$, всегда можно представить в стандартной форме (2.2), если воспользоваться функцией выбора. Поэтому, если для некоторой функции $f(\overline{x}, y)$ мы провели доказательство ее примитивной рекурсивности с помощью оператора примитивной рекурсии и для выражения $f(\overline{x}, y+1)$ получили не стандартную форму суперпозиции функции $f(\overline{x}, y)$ и известных примитивно–рекурсивных функций, то мы знаем, что получить стандартную форму (2.2) мы всегда можем. Поэтому в дальнейшем мы не всегда будем доводить процесс доказательства до конечной суперпозиции в стандартном виде (2.2).

2.3 Оператор минимизации

Вернемся к начальной цели, которую мы поставили — дать формальное определение алгоритма как функции, принадлежащей некоторому конструктивно определенному классу функций. Возможно, что определение примитивно–рекурсивных функций будет удовлетворять этой цели. Однако сначала надо практически убедиться в том, что частный случай алгоритмов — программы для ЭВМ — можно представить примитивно–рекурсивными функциями. Заданные в определении примитивно–рекурсивных функций средства их конструирования позволяют использовать для написания алгоритмов простейшие функции, оператор суперпозиции и оператор примитивной рекурсии. Если рассматривать программную реализацию этих конструктивных элементов, то простейшие функции соответствуют некоторому базовому набору операций языка программирования, оператор суперпозиции — последовательному выполнению действий, а оператор примитивной рекурсии — рекурсивной процедуре или оператору цикла типа "for". Рассмотрим достаточность этих средств для конструирования любого алгоритма. Сначала введем в рассмотрение новый оператор, представляющий аналог оператора цикла типа "while". Поскольку такой цикл выполняется многократно при истинности некоторого условия, необходимо рассматривать предикаты, определяющие такие условия. Любой предикат принимает значения "ложь" и "истина с которыми нельзя производить арифметических действий. Для

того, чтобы представить значение предиката натуральным числом, введем понятие характеристической функции предиката

$$\chi_p(x_1,\ldots,x_n)=\begin{cases}1, & P(x_1,\ldots,x_n)= \quad \text{истина}\\ 0, & P(x_1,\ldots,x_n)= \quad \text{ложь}\end{cases}$$

Кстати, при реализации логических переменных в программах для ЭВМ также обычно используются именно числа 0 и 1 для машинного представления значений "ложь" и "истина".

Теперь можно дать определение специального оператора, выполняющего цикл вычислений по некоторому условию.

Определение 2.3. Функция $f(x_1,\ldots,x_n)$ получается оператором минимизации из предиката $P(x_1,\ldots,x_n,z)$, если в любой точке $(x_1,x_2,...,x_n)$ значением функции $f(x_1,\ldots,x_n)$ является минимальное значение z, обращающее предикат $P(x_1,\ldots,x_n,z)$ в истину. Оператор минимизации имеет обозначение

$$f(x_1,\ldots,x_n)=\mu_z(P(x_1,\ldots,x_n,z)). \tag{2.3}$$

Для того, чтобы для любого предиката $P(x_1,\ldots,x_n,z)$ найти минимальное значение z, обращающее $P(x_1,\ldots,x_n,z)$ в истину, нужно последовательно перебирать значения $z=0,1,2,....$ Вычисление значения функции, заданной с помощью оператора минимизации, можно программно реализовать следующим образом:

```
int f(int x1,..., int xn)
{
int z=0;
while (P(x1,... xn,z)==0)  z++;
return z;
}
```

Пример. Пусть $f(x,y)=\mu_z(2{*}x{+}z=y{+}1)$. Предикат $P(x,y,z)=(2{*}x{+}z=y{+}1)$ является примитивно–рекурсивным, т.к. его характеристическая функция является суперпозицией примитивно–рекурсивных функций и равна

$$sg(sg((2x+z)\dot{-}(y+1))+sg((y+1)\dot{-}(2x+z))).$$

Вычислим значение $f(1,5)$:

$$\begin{array}{ll} z=0, & P(1,5,0)= \quad \text{Ложь},\\ z=1, & P(1,5,1)= \quad \text{Ложь},\\ \ldots & \\ z=4, & P(1,5,4)= \quad \text{Истина}.\end{array}$$

Тогда $f(1,5)=4$. Рассмотрим вычисление $f(5,1)$. В этой точке функция не определена, т.к. $P(5,1,z)=$ "Ложь"для любого значения $z\in N$.

Полученный пример показывает, что с помощью оператора минимизации из примитивно–рекурсивных функций можно получить не всюду определенные функции, поэтому оператор минимизации выводит из класса примитивно–рекурсивных функций. Попробуем сделать так, чтобы остаться внутри класса примитивно–рекурсивных функций, но оставить возможность использования циклов типа "while". Для этого достаточно при определении оператора минимизации ввести ограничение на изменение переменной z так, чтобы цикл всегда завершался после некоторого числа шагов. В общем случае такое ограничение должно являться функцией свободных переменных x_1, x_2,...,x_n.

2.4 Ограниченный оператор минимизации

Определение 2.4. Функция $f(x_1, \ldots, x_n)$ получается ограниченным оператором минимизации из предиката $P(x_1, \ldots, x_n, z)$ и граничной функции $U(x_1, \ldots, x_n)$, если в любой точке значение этой функции определяется следующим образом:

а) существует $z < U(x_1, x_2, \ldots, x_n)$ такое, что $P(x_1, \ldots, x_n, z) =$ "истина тогда $f(x_1, \ldots, x_n) = \mu_z(P(x_1, \ldots, x_n, z))$;

б) не существует $z < U(x_1, \ldots, x_n)$ такое, что $P(x_1, \ldots, x_n, z) =$ "истина тогда в качестве значения функции принимается значение граничной функции:

$$f(x_1, \ldots, x_n) = U(x_1, \ldots, x_n)$$

.

Сразу заметим, что в определении (2.4) отношение $z < U(x_1, x_2, \ldots, x_n)$ можно было бы заменить на $z \leq U(x_1, x_2, \ldots, x_n)$ — значение функции не изменилось бы. С учетом эквивалентности таких определений иногда будем использовать второе из этих отношений, если это окажется удобным. Записывается определение функции $f(x_1, \ldots, x_n)$, полученной с помощью ограниченного оператора минимизации, как

$$f(x_1, \ldots, x_n) = \mu_{z<U(x_1,\ldots,x_n)}(P(x_1, \ldots, x_n, z)). \tag{2.4}$$

Ясно, что алгоритм вычисления функции (2.4) получен из алгоритма вычисления (2.3) дополнением условия так, чтобы имелась гарантия завершения цикла:

```
int   f(int x1, ..., int xn)
{
int z=0;
while ( !P(x1,..., xn,z) && (z<U(x1,...,xn) ) z++;
return z;
}
```

Докажем, что ограниченный оператор минимизации является примитивно–рекурсивным. Рассмотрим сначала вспомогательные утверждения.

Теорема 2.2. Функция

$$g(x_1, \ldots, x_n, y) = \sum_{i=0}^{y} f(x_1, \ldots, x_n, i)$$

является примитивно–рекурсивной при условии примитивной рекурсивности функции $f(x_1, \ldots, x_n, i)$.

Доказательство. Представим $g(x_1, \ldots, x_n, y)$ с помощью схемы примитивной рекурсии от примитивно–рекурсивных функций:

$$g(x_1, \ldots, x_n, 0) = \sum_{i=0}^{0} f(x_1, \ldots, x_n, i) = f(x_1, \ldots, x_n, 0)$$
$$g(x_1, \ldots, x_n, y+1) = \sum_{i=0}^{y+1} f(x_1, \ldots, x_n, i) =$$
$$= g(x_1, \ldots, x_n, y) + f(x_1, \ldots, x_n, y+1).$$

□

Теорема 2.3. Функция

$$g(x_1,\ldots,x_n,z,y)=\sum_{i=z}^{y} f(x_1,\ldots,x_n,i)$$

является примитивно–рекурсивной, если функция $f(x_1,\ldots,x_n,i)$ примитивно–рекурсивна.

Доказательство. Запишем функцию $g(x_1,\ldots,x_n,z,y)$ в виде суперпозиции примитивно–рекурсивных функций:

$$\begin{aligned} &g(x_1,\ldots,x_n,z,y)=\\ &=\left(\sum_{i=0}^{y} f(x_1,\ldots,x_n,i)\dot{-}\sum_{i=0}^{z} f(x_1,\ldots,x_n,i)+f(x_1,\ldots,x_n,z)\right)\cdot\chi(z\le y). \end{aligned}$$

Здесь $\chi(z\le y)$ —характеристическая функция предиката $(z\le y)$, которая является примитивно–рекурсивной, т.к. может быть представлена в виде суперпозиции примитивно – рекурсивных функций сложения, усеченной разности и знаковой функции: $\chi(z\le y)=sg(y+1\dot{-}z)$. □

Теорема 2.4. Функция

$$g(x_1,\ldots,x_n)=\sum_{i=V(x_1,\ldots,x_n)}^{U(x_1,\ldots,x_n)} f(x_1,\ldots,x_n,i)$$

является примитивно–рекурсивной, если примитивно–рекурсивны функции

$$U(x_1,\ldots,x_n),\ f(x_1,\ldots,x_n,i),\ V(x_1,\ldots,x_n).$$

Доказательство очевидно, т.к. функция $g(x_1,\ldots,x_n)$ является суперпозицией заданных примитивно–рекурсивних функций и функции, примитивно–рекурсивной по теореме 2.3. □

Аналогичные теоремы можно доказать для произведения.

Теорема 2.5. Функция

$$g(x_1,\ldots,x_n,y)=\prod_{i=0}^{y} f(x_1,\ldots,x_n,i)$$

является примитивно–рекурсивной, если функция $f(x_1,\ldots,x_n,i)$ примитивно–рекурсивна.

Теорема 2.6. Функция

$$g(x_1,\ldots,x_n,z,y)=\prod_{i=z}^{y} f(x_1,\ldots,x_n,i)$$

является примитивно–рекурсивной, если примитивно–рекурсивна $f(x_1,\ldots,x_n,i)$.

Теорема 2.7. Функция

$$g(x_1,\ldots,x_n)=\prod_{i=V(x_1,\ldots,x_n)}^{U(x_1,\ldots,x_n)} f(x_1,\ldots,x_n,i)$$

является примитивно–рекурсивной, если примитивно–рекурсивны функции

$$U(x_1,\ldots,x_n),f(x_1,\ldots,x_n,i),V(x_1,\ldots,x_n).$$

Теорема 2.8. Функция

$$g(x_1,\dots,x_n)=\mu_{z<U(x_1,x_2,\dots x_n)}(P(x_1,\dots,x_n,z)),$$

построенная с помощью ограниченного оператора минимизации из примитивно–рекурсивного предиката $P(x_1,\dots,x_n,z)$ и примитивно–рекурсивной функции $U(x_1,\dots,x_n)$, является примитивно–рекурсивной.

Доказательство. Представим функцию $g(x_1,\dots,x_n)$ в виде вспомогательной примитивно–рекурсивной функции и покажем, что в любой точке эти функции равны.

Рассмотрим функцию

$$\tilde{g}(x_1,\dots,x_n)=\sum_{i=0}^{U(x_1,\dots,x_n)\dot{-}1}\left(\prod_{j=0}^{i}\left(1\dot{-}\chi_p(x_1,\dots,x_n,j)\right)\right). \tag{2.5}$$

В соответствии с определением функции $g(x_1,\dots,x_n)$ рассмотрим два случая вычисления функции $\tilde{g}(x_1,\dots,x_n)$.

Первый вариант вычисления (2.5) соответствует случаю, когда не существует $z<U(x_1,\dots,x_n)$, такое, что $\chi_p(x_1,\dots,x_n,z)=1$. Тогда для любого $z<U(x_1,\dots,x_n)$ имеем $\left(1\dot{-}\chi_p(x_1,\dots,x_n)z\right)=1$, отсюда для любого $i<U(x_1,\dots,x_n)$

$$\prod_{j=0}^{i}\left(1\dot{-}\chi_p(x_1,\dots,x_n,z)\right)=1,$$

следовательно, в соответствии с (2.5)

$$\sum_{i=0}^{U(x_1,\dots,x_n)\dot{-}1}\left(\prod_{j=0}^{i}(1\dot{-}\chi_p(x_1,\dots,x_n,j)\right)=U(x_1,\dots,x_n).$$

Функция $g(x_1,\dots,x_n)$ по определению в этом случае также равна $U(x_1,\dots,x_n)$.

Рассмотрим второй вариант вычисления функции (2.5): пусть найдется такое $z<U((x_1,\dots,x_n)$, что $\chi_p(x_1,\dots,x_n,z)=1$, тогда

$$\begin{aligned}
&1\dot{-}\chi_p(x_1,\dots,x_n,0)=1,\\
&1\dot{-}\chi_p(x_1,\dots,x_n,1)=1,\\
&\dots\\
&1\dot{-}\chi_p(x_1,\dots,x_n,z-1)=1,\\
&1\dot{-}\chi_p(x_1,\dots,x_n,z)=0.
\end{aligned}$$

Следовательно

$$\begin{aligned}
&\prod_{j=0}^{0}\left(1\dot{-}\chi_p(x_1,\dots,x_n,j)\right)=1,\\
&\dots\\
&\prod_{j=0}^{z-1}\left(1\dot{-}\chi_p(x_1,\dots,x_n,j)\right)=1,\\
&\prod_{j=0}^{z}\left(1\dot{-}\chi_p(x_1,\dots,x_n,j)\right)=0,\\
&\prod_{j=0}^{z+1}\left(1\dot{-}\chi_p(x_1,\dots,x_n,j)\right)=0,\\
&\dots
\end{aligned}$$

Тогда сумма (2.5) равна z и в соответствии с определением ограниченного μ – оператора значение $g(x_1,\ldots,x_n)$ также равно z. Таким образом, в любом случае $g(x_1,\ldots,x_n)$ можно представить выражением (2.5): функции $\tilde{g}(x_1,\ldots,x_n)$ и $g(x_1,\ldots,x_n)$ эквивалентны. Но функция $\tilde{g}(x_1,\ldots,x_n)$ является суперпозицией примитивно–рекурсивных функций, следовательно, она примитивно–рекурсивна, а, значит, примитивно–рекурсивна и функция $g(x_1,\ldots,x_n)$. $\square$

Пример. Показать, что функция $[x/(y+1)]$ является примитивно–рекурсивной.

Очевидно, что $[x/(y+1)] = \mu_{z<x+1}\ ([x/(y+1)]=z)$. Рассмотрим два случая. Пусть x нацело делится на $y+1$, тогда

$$[x/(y+1)] = \mu_{z<x+1}\ (x = z(y+1))\,.$$

Пусть x нацело не делится на $y+1$, тогда в соответствии с определением ограниченного оператора минимизации подставляя в качестве z значения 0,1,2,.. в предикат $x = z(y+1)$ будем получать неравенство $x > z(y+1)$ до тех пор, пока значение z не достигнет $[x/(y+1)]$. Тогда при подстановке значения $z+1$ получим отношение $x < (z+1)\cdot(y+1)$. Объединяя оба варианта, получим $[x/(y+1)] = \mu_{z<x+1}\ (x<(z+1)\cdot(y+1))$. Характеристическая функция $sg((z+1)\cdot(y+1)\dot{-}x)$ предиката и граница $s(x)$ — примитивно – рекурсивные функции, тогда и $[x/(y+1)]$ примитивно–рекурсивна.

Пример. Показать, что $\tau(x)$ — число делителей числа x — является примитивно–рекурсивной функцией.

Очевидно, что $\tau(x) = \sum\limits_{i=1}^{x} \overline{sg}\{x/i\}$. Выполняя подстановку для $i = y+1$, получим $\tau(x) = \sum\limits_{y=0}^{x\dot{-}1} \overline{sg}\{x/(y+1)\}$. Остаток от деления $\{x/(y+1)\}$ равен $x\dot{-}[x/(y+1)]$. Тогда $\tau(x)$ является суперпозицией примитивно–рекурсивных функций.

Таким образом, ограниченным оператором минимизации можно пользоваться при построении примитивно–рекурсивных функций. Возникает вопрос: достаточно ли класса примитивно–рекурсивных функций для построения определения любого алгоритма. Чтобы показать недостаточную порождающую силу данного класса функций, необходимо привести пример функции, для которой существует алгоритм ее вычисления, но которая не является примитивно–рекурсивной. Этим мы и займемся в следующем параграфе.

2.5 Быстро растущие функции

Чтобы показать существование функций вычислимых, но не примитивно–рекурсивных, построим такую функцию, которая растет быстрее любой примитивно–рекурсивной функции. Сначала рассмотрим известные функции сложения, умножения, возведения в степень, зная, что каждая последующая из них растет быстрее предыдущей.

$$\begin{aligned} &P_0(a,x) = a + x;\\ &P_1(a,x) = a\cdot x;\\ &P_2(a,x) = a^x. \end{aligned} \tag{2.6}$$

Рассмотрим рекурсивное определение каждой из этих функций через предше-

ствующие функции:

$$P_1(a,0) = 0;$$
$$P_1(a,x+1) = a \cdot (x+1) = a + P_1(a,x) = P_0(a, P_1(a,x));$$
$$P_2(a,0) = 1;$$
$$P_2(a,x+1) = a^{x+1} = a^x \cdot a = P_1(a, P_2(a,x)).$$

Продолжим последовательность функций (2.6), для чего введем в рассмотрение множество функций $P_n(a,x)$:

$$\begin{aligned} &P_0(a,x) = a + x; \\ &P_{n+1}(a,0) = sg(n); \\ &P_{n+1}(a,x+1) = P_n(a, P_{n+1}(a,x)). \end{aligned} \tag{2.7}$$

Очквидно, что характер функций $P_n(a,x)$ существенно зависит от n и x, значение переменной a при фиксированных n и x принципиально не меняет тип возрастания функции. Поэтому для анализа поведения функций $P_n(a,x)$ зафиксируем $a = 2$ и введем в рассмотрение функцию Аккермана:

$$B(n,x) = P_n(2,x) \tag{2.8}$$

или в соответствии с определением (2.7)

$$\begin{aligned} &B(0,x) = 2 + x; \\ &B(n+1,0) = sg(n); \\ &B(n+1,x+1) = B(n, B(n+1,x)). \end{aligned} \tag{2.9}$$

Рассмотрим диагональную функцию Аккермана:

$$A(x) = B(x,x).$$

Можно доказать, что функция Аккермана обладает следующими свойствами:

1) $B(n,x) \geq 2^x$ при $n \geq 2,\ x \geq 1$;
2) $B(n,x+1) \geq B(n,x)$ при $n, x \geq 1$;
3) $B(n+1,x) \geq B(n,x+1)$ при $n \geq 1,\ x \geq 2$;
4) $B(n+1,x) \geq B(n,x)$ при $n \geq 2,\ x \geq 2$.

Доказательство свойства 1.

Применим индукцию по n. При $n = 2$ в соответствии с (2.7) и (2.8) выполняется равенство

$$B(2,x) = P_2(2,x) = 2^x.$$

Пусть свойство справедливо для $n \geq 2$, докажем его для $n+1$. Для этого применим индукцию по x. При $x = 1$ справедливо

$$\begin{aligned} B(n+1,1) = \ & B(n, B(n+1,0)) = (\text{по 2.7 и условию } n \geq 2) \\ & = B(n,1) = \ (\text{по индуктивному предположению}) \\ & = 2^1. \end{aligned}$$

Пусть при некотором $n \geq 2$ свойство справедливо для $x \geq 1$, докажем его для $x+1$:

$$\begin{aligned} B(n+1,x+1) = \ & B(n, B(n+1,x)) \geq \ (\text{индукция по n}) \\ & \geq 2^{B(n+1,x)} \geq \ (\text{индукция по x}) \\ & \geq 2^{2^x} \geq \ (\text{ограничение на x}) \\ & \geq 2^{x+1}, \end{aligned}$$

что и требовалось доказать.□

Доказательство свойства 2.

При $n = 1$ в соответствии с (2.7) и (2.8) $B(1, x) = P_1(2, x) = 2x$, $B(1, x + 1) = P_1(2, x + 1) = 2(x + 1)$, следовательно при $n = 1$ свойство справедливо. Рассмотрим $n > 1$, тогда

$$\begin{aligned} B(n+1, x+1) = \quad & B(n, B(n+1, x)) \geq \text{ (свойство 1)} \\ & \geq 2^{B(n+1,x)} \geq B(n+1, x). \end{aligned}$$

□

Доказательство свойства 3.

$$\begin{aligned} B(n+1, x+1) = \quad & B(n, B(n+1, x)) \geq \text{ (свойства 1 и 2)} \\ & \geq B(n, 2^x) \quad \geq \text{ (свойство 2)} \\ & \geq B(n, x+2). \end{aligned}$$

□

Доказательство свойства 4.

$$\begin{aligned} B(n+1, x) \quad & \geq \text{ (свойство 3)} \\ & \geq B(n, x+1) \quad \geq \text{ (свойство 2)} \\ & \geq B(n, x). \end{aligned}$$

□

Определение 2.5. Функция $f(x_1, \ldots, x_n)$ называется B–мажорируемой, если существует такое $N \geq 0$, что

$$f(x_1, \ldots, x_n) < B(N, max\{x_1, \ldots, x_n\})$$

в любой точке $(x_1, \ldots, x_n)$, где все $x_i > 2$.

Теорема 2.9. Любая примитивно–рекурсивная функция B–мажорируема.

Доказательство. В соответствии с определением примитивно–рекурсивных функций доказательство проведем в три шага. Сначала покажем, что простейшие функции B–мажорируемы:

$$s(x) = x + 1 < x + 2 = P_0(2, x) = B(0, x),$$

$$\begin{aligned} I_m(x_1, \ldots, x_n) = x_m \leq max\{x_1, \ldots, x_n\} < max\{x_1, \ldots, x_n\} + 2 \\ = B(0, max\{x_1, \ldots, x_n\}), \end{aligned}$$

$$0(x_1, \ldots, x_n) = 0 < max\{x_1, \ldots, x_n\} + 2 = B(0, max\{x_1, \ldots, x_n\}).$$

Затем покажем, что оператор суперпозиции позволяет получать из B–мажорируемых функций B–мажорируемые функции. Для простоты рассмотрим функции одного аргумента. Пусть $f(x) = g(h(x))$, где $h(x)$ и $g(x)$ — B–мажорируемы:

$$\begin{aligned} &\exists_{n \geq 0} \forall_{x>2} g(x) < B(n, x), \\ &\exists_{m \geq 0} \forall_{x>2} h(x) < B(m, x). \end{aligned}$$

Тогда

$$\begin{aligned} f(x) = \quad & g(h(x)) < B(n, h(x)) < \text{ (св. 2)} < \\ & \leq B(n, B(m, x)) < \text{ (определение и свойство 3)} \\ & \leq B(max\{n, m\}, B(max\{n, m\} + 1, x)) = \\ & = B(max\{n, m\} + 1, x + 1) \leq \text{ (свойство 4)} \\ & \leq B(max\{n, m\} + 2, x). \end{aligned}$$

Следовательно, функция $f(x)$ B–мажорируема. Наконец, покажем, что из B–мажорируемых функций с помощью оператора примитивной рекурсии можно получить только B–мажорируемую функцию. Пусть

$$\begin{array}{l} f(0) = const \\ f(x+1) = h(x, f(x)), \text{ где} \\ h(x,y) < B(n, max\{x,y\}). \end{array}$$

Рассмотрим начальную точку $x = 3$, соответствующую определению B–мажорируемой функции:

$$\begin{array}{rl} f(3) & = h(2, h(1, h(0, f(0)))) = k < k+2 = \\ & = B(0,k) \leq \text{ (свойство 4)} \\ & \leq B(l,3), \text{ где } l = k \dot{-} 3. \end{array}$$

Теперь воспользуемся математической индукцией по x:

$$\begin{array}{l} f(x+1) = h(x, f(x)) < B(n, max\{x, f(x)\}) \leq \\ \leq B(max\{n,l\}, max\{x, f(x)\}). \end{array}$$

Рассмотрим варианты вычисления $max\{x, f(x)\}$.

Пусть $max\{x, f(x)\} = x$, тогда

$$\begin{array}{rl} f(x+1) & < B(max\{n,l\}, x) \leq \\ & \leq B(max\{n,l\}+1, x+1). \end{array}$$

Пусть $max\{x, f(x)\} = f(x)$, тогда

$$\begin{array}{rl} f(x+1) & < B(max\{n,l\}, f(x)) \leq \text{ (св. 2, индукция)} \\ & \leq B(max\{n,l\}, B(max\{n,l\}+1, x)) = \\ & = B(max\{n,l\}+1, x+1). \end{array}$$

□

Теорема 2.10. Диагональная функция Аккермана растет быстрее любой примитивно–рекурсивной функции.

Доказательство. Пусть $f(x)$ — любая примитивно–рекурсивная функция, тогда по теореме (2.9) она B–мажорируема, т.е. найдется такое $n \geq 0$, что для всех $x > 2$ справедливо неравенство $f(x) < B(n,x)$. Тогда

$$f(x+n) < B(n, x+n) \leq B(x+n, x+n) = A(x+n).$$

Указанное неравенство справедливо для произвольной примитивно–рекурсивной функции. В частности, если функция $A(x)$ является примитивно–рекурсивной, то для нее должно выполняться неравенство

$$A(x+n) < A(x+n).$$

Полученное противоречие означает, что функция $A(x)$ не является примитивно–рекурсивной функцей. □

Следствие. Примитивно–рекурсивные функции нельзя использовать для определения понятия алгоритма. Это следует из того факта, что существует алгоритм вычисления функции $A(x)$, но примитивно–рекурсивной эта функция не является.

2.6 Частично–рекурсивные функции и тезис Черча

В силу узости класса примитивно–рекурсивных функций для определения алгоритма необходимо выйти из этого класса. Мы рассмотрели оператор, применение которого выводит из класса примитивно–рекурсивных функций, — это оператор минимизации. Будем использовать этот оператор как дополнительное конструктивное средство при определении нового класса функций.

Определение 2.6. Частично–рекурсивной называется функция, построенная из простейших с помощью конечного числа операторов суперпозиции, примитивной рекурсии и минимизации.

Определение 2.7. Всюду определенная частично–рекурсивная функция называется общерекурсивной или просто рекурсивной функцией.

В силу теоремы (2.10) примером рекурсивной, но не примитивно–рекурсивной функции является диагональная функция Аккермана $A(x)$.

Частично–рекурсивные функции используются для определения понятия алгоритма, которое вводится с помощью тезиса Черча.

Тезис Черча: всякий алгоритм может быть реализован частично–рекурсивной функцией.

Понятие частично–рекурсивной функции — одно из главных в теории алгоритмов. *Тезис Черча — это не теорема*, а утверждение, которое связывает понятие алгоритма и строгое математическое понятие частично–рекурсивной функции. Тезис Черча является достаточным для того, чтобы придать необходимую точность формулировкам алгоритмических проблем и сделать принципиально возможным доказательство их неразрешимости. Утверждение о несуществовании частично–рекурсивной функции эквивалентно факту несуществованию алгоритма решения соответствующей задачи.

В силу тезиса Черча вопрос о вычислимости функции или, что то же самое, о существовании алгоритма ее вычисления, равносилен вопросу о ее частичной рекурсивности. Понятие частично–рекурсивной функции — строгое математическое, поэтому обычные математические методы и приемы позволяют непосредственно доказать, что решающая задачу функция не может быть рекурсивной и тем самым доказать неразрешимость проблемы.

Точное описание класса частично–рекурсивных функций вместе с тезисом Черча дает одно из возможных решений задачи о формальном определении алгоритма. Вместе с тем, это решение не вполне прямое, так как понятие вычислимой функции является вторичным по отношению к понятию алгоритма: значение функции в каждой точке, соответствующей исходным данным алгоритма — это результат работы алгоритма на этих данных. Спрашивается, нельзя ли уточнить само понятие алгоритма и уже затем при его помощи определить точно класс вычислимых функций? Это было сделано Постом и Тьюрингом. Основная мысль, заложенная в такой формализации алгоритма, заключается в том, что алгоритмические процессы — это процессы, которые может совершать некоторая подходящее устроенная машина. В соответствии с этой мыслью в следующей главе мы рассмотрим описание в точных математических терминах довольно узких классов машин, но на этих машинах оказывается возможным осуществить все алгоритмы. Алгоритмы, осуществимые на этих формально определенных машинах, можно рассматривать как математические точно определенные алгоритмы. В главе 3 мы покажем, что класс функций, вычислимых на этих машинах, в точности совпадает с классом всех частично–рекурсивных функций. Тем самым мы получим еще одно фундаментальное подтверждение тезиса

Черча.

Заметим, что хотя рассмотренный класс частично–рекурсивных функций содержит только функции, определенные на множестве натуральных чисел, это не снижает общности представления об алгоритме как о частично–рекурсивной функции. Как мы рассмотрим в дальнейшем, существуют способы нумерации объектов, не являющихся по сути своей числами. В качестве простейшего примера приведем пока пример нумерации n–ок натуральных чисел $\langle a_1, a_2, \ldots, a_n \rangle$. Для простоты рассмотрим сначала случай $n = 2$. В двумерном пространстве расположим пары натуральных чиел в виде следующей бесконечной матрицы:

$$
\begin{array}{ccccccccccc}
< 0,0 > & & < 0,1 > & \rightarrow & < 0,2 > & & < 0,3 > & \rightarrow & < 0,4 > & \ldots \\
\downarrow & \nearrow & & \swarrow & & \nearrow & & \swarrow & & \\
< 1,0 > & & < 1,1 > & & < 1,2 > & & < 1,3 > & & < 1,4 > & \ldots \\
& \swarrow & & \nearrow & & \swarrow & & & & \\
< 2,0 > & & < 2,1 > & & < 2,2 > & & < 2,3 > & & < 2,4 > & \ldots \\
\downarrow & \nearrow & & \swarrow & & & & & & \\
< 3,0 > & & < 3,1 > & & < 3,2 > & & < 3,3 > & & < 3,4 > & \ldots \\
& \swarrow & & & & & & & & \\
< 4,0 > & & < 4,1 > & & < 4,2 > & & < 4,3 > & & < 4,4 > & \ldots \\
\downarrow & & & & & & & & & \\
\ldots & & \ldots & & \ldots & & \ldots & & \ldots & \ldots
\end{array}
$$

Аналогично можно ввести нумерацию в n–мерном пространстве, нумеруя кортежи $\langle a_1, a_2, \ldots, a_n \rangle$.

Другой метод нумерации — метод цифр в некоторой системе счисления — основан на представлении объектов (не обязательно числовых) в виде натурального числа в некоторой системе счисления с конечным основанием. Метод нумерации методом цифр мы рассмотрим в главе 3.

2.7 Рекурсивные и рекурсивно перечислимые множества

До сих пор понятия примитивной рекурсивности и частичной рекурсивности были определены лишь для функций. Теперь эти понятия будут перенесены на подмножества натуральных чисел, а также на множества некоторых других объектов.

Определение 2.8. Подмножество A множества всех натуральных чисел N называется рекурсивным (примитивно–рекурсивным), если характеристическая функция множества A частично–рекурсивна (соответственно, примитивно–рекурсивна).

Так как все примитивно–рекурсивные функции являются частично–рекурсивными, то каждое примитивно–рекурсивное множество является рекурсивным. Обратное неверно.

С точки зрения теории алгоритмов из двух введенных понятий — рекурсивного множества и примитивно–рекурсивного множества — основным следует считать первое благодаря следующему обстоятельству. Проблемой вхождения числового множества A называется задача отыскания алгоритма, который по стандартной записи числа a в некоторой системе счисления позволяет узнать, входит число a в A или нет, т.е. указанный алгоритм позволяет вычислять значение характеристической функции множества A. В силу тезиса Черча существование такого алгоритма равносильно

рекурсивности характеристической функции. Поэтому можно сказать, что рекурсивные множества — это множества с алгоритмически разрешимой проблемой вхождения.

Введем теперь несколько основных свойств рекурсивных и примитивно–рекурсивных множеств.

Характеристическими функциями пустого множества $\emptyset$ и множества натуральных чисел N являются функции–константы 0 и 1 соответственно. Эти функции примитивно–рекурсивны. Поэтому множества $\emptyset$ и N также примитивно–рекурсивны.

Характеристической функцией для конечного множества чисел $\{a_1, a_2, ..., a_n\}$ является примитивно–рекурсивная функция

$$sg(|x - a_1| \cdot |x - a_2| \cdot ... \cdot |x - a_n|).$$

Поэтому каждое конечное множество натуральных чисел примитивно–рекурсивно.

В упражнениях к данной главе будет показано, что характеристические функции свойств "быть простым числом", "быть четным числом", "быть точным квадратом" примитивно–рекурсивны. Поэтому примитивно–рекурсивными являются и множества всех простых чисел, всех четных чисел, всех точных квадратов. Аналогично можно убедиться, что примитивно–рекурсивными будут и многие другие часто встречающиеся множества чисел.

Теорема 2.11. Дополнение рекурсивного (примитивно–рекурсивного) множества, а также объединение и пересечение любой конечной системы рекурсивных (примитивно–рекурсивных) множеств есть множества рекурсивные (примитивно–рекурсивные).

Доказательство. Пусть $f_1(x)$, $f_2(x)$, ... , $f_n(x)$ — характеристические функции множеств A_1, A_2,...,A_n. Тогда функции

$$\begin{aligned} f(x) &= \overline{sg}(f_1(x)), \\ h(x) &= sg(f_1(x) + f_2(x) + ... + f_n(x)) \\ g(x) &= f_1(x) \cdot f_2(x) \cdot ... \cdot f_n(x), \end{aligned}$$

будут характеристическими множествами соответственно для дополнения множества A_1, объединения и пересечения множеств A_1, A_2,...,A_n. Если $f_1(x)$, $f_2(x)$,...,$f_n(x)$ — частично–рекурсивные (соответственно, примитивно–рекурсивные) функции, то такими же будут и функции $f(x)$, $g(x)$, $h(x)$. □

Теорема 2.12. Если всюду определенная функция $f(x)$ частично–рекурсивна (примитивно–рекурсивна), то множество A решений уравнения

$$f(x) = 0$$

рекурсивно (примитивно–рекурсивно).

Доказательство. Характеристической функцией множества A служит функция $sg(f(x))$, рекурсивная или примитивно-рекурсивная вместе с функцией $f(x)$. □

Совокупность всех значений, принимаемых некоторой примитивно–рекурсивной функцией, в общем случае не будет ни примитивно–рекурсивным, ни даже рекурсивным множеством. Однако, существует ряд простых признаков для того, чтобы совокупность значений примитивно–рекурсивной (рекурсивной) функции была примитивно–рекурсивной (рекурсивной). Один из примеров такого признака формулируется в следующей теореме.

Теорема 2.13. Если примитивно–рекурсивная (общерекурсивная) функция $f(x)$ удовлетворяет условию

$$\forall_{x \in N} f(x) \geq x,$$

то совокупность значений M этой функции есть множество примитивно–рекурсивное (рекурсивное). В частности, условие теоремы выполняется для возрастающей функции.

Доказательство. По условию теоремы $\forall_{x\in N}(f(x) \geq x)$, следовательно, для любого заданного x можно рассмотреть все натуральные числа $i \leq x$, вычислить значение функции $f(i)$, а затем сравнить его с x. Равенство $x = f(i)$ выполняется тогда и только тогда, когда x является значением функции $f(i)$. Из этого равенства следует, что $sg|f(i) - x| = 0$. Следовательно, в процессе проверки для всех $0 \leq i \leq x$ получим $\prod\limits_{i=0}^{x} sg|f(i) - x| = 0$ тогда и только тогда, когда x — значение $f(i)$. Тогда характеристической функцией множества M служит функция

$$g(x) = 1\dot{-}\prod_{i=0}^{x} sg|f(i) - x|$$

рекурсивная или примитивно-рекурсивная вместе с функцией $f(x)$. □

Фундаментальную роль в теории рекурсивных функций играют рекурсивно перечислимые множества.

Определение 2.9. Множество чисел A называется рекурсивно перечислимым, если существует такая рекурсивная функция $f(x, y)$, что уравнение

$$f(a, y) = 0$$

имеет решение тогда и только тогда, когда $a \in A$.

Таким образом, множество натуральных чисел является рекурсивно перечислимым, если оно либо пустое множество, либо есть область значений некоторой рекурсивной функции. Если мы принимаем тезис Черча, то, говоря неформально, можно считать, что рекурсивно перечислимым является всякое множество натуральных чисел, порождаемое каким–либо алгоритмическим процессом.

Легко заметить, что каждое примитивно–рекурсивное множество рекурсивно перечислимо. Действительно, согласно определению, характеристическая функция $f(x)$ произвольного примитивно–рекурсивного множества A является примитивно–рекурсивной. Но тогда примитивно–рекурсивное уравнение

$$\overline{sg}(f(a)) + x = 0$$

имеет решение $x = 0$ тогда и только тогда, когда $a \in A$. Действительно, если $a \in A$, то $f(a) = 1$, $\overline{sg}(f(a)) = \overline{sg}(1) = 0$ и уравнение $0 + x = 0$ имеет решение $x = 0$. Иначе, если $a \notin A$, то $f(a) = 0$ и уравнение $1 + x = 0$ решения не имеет.

Пример. Множество квадратов натуральных чисел $M = \{a|a = x^2\}$ рекурсивно перечислимо, т.к. оно просто задано уравнением $a = x^2$.

Важность введенных понятий рекурсивности и рекурсивной перечислимости множеств важны потому, что благодаря им становятся точными такие понятия, как "конструктивный способ задания множеств" и "множество, заданное эффективно". В соответствии с определением, множество является рекурсивно перечислимым, если оно является областью значений некоторой общерекурсивной функции $f(a, y)$. Функция $f(a, y)$ называется перечисляющей функцией. Тогда для рекурсивно перечислимого множества существует перечисляющая процедура, и, соответственно, алгоритм вычисления элементов $a \in A$. Такой алгоритм называется перечисляющим или порождающим.

Пример. В большинстве языков программирования используется понятие идентификатора - последовательности букв или цифр, начинающейся с буквы. Чтобы

построить порождающую процедуру для идентификаторов, можно ввести лексико–графический порядок перечисления букв и цифр, а затем на основе рекурсивного определения идентификатора построить перечисляющую процедуру для множества всех идентификаторов. Подробнее такие порождающие процедуры мы рассмотрим в главе 6, когда введем понятие порождающей грамматики.

2.8 Контрольные вопросы к разделу

1. Перечислите свойства алгоритма. Какие из этих свойств являются обязательными? Выполняются ли эти свойства для программ для ЭВМ?

2. Перечислите простейшие функции.

3. Дайте общее определение оператора примитивной рекурсии.

4. Запишите схему примитивной рекурсии для функции трех аргументов.

5. Дайте определение оператора суперпозиции.

6. Как устранить рекурсию в программе вычисления функции, если функция определена с помощью оператора примитивной рекурсии?

7. Может ли не всюду определенная функция быть примитивно-рекурсивной?

8. Дайте определение оператора минимизации.

9. В чем состоит отличие оператора минимизации от ограниченного оператора минимизации?

10. Какие цели преследовались при использовании ограничения в ограниченном операторе минимизации?

11. Пусть некоторая функция построена из примитивно-рекурсивных функций с помощью оператора минимизации. Является ли эта функция примитивно-рекурсивной?

12. Пусть некоторая функция построена из примитивно–рекурсивных функций с помощью ограниченного оператора минимизации. Является ли эта функция примитивно–рекурсивной?

13. Как строятся быстро растущие функции $P_i(a, x)$?

14. Поставьте знак отношения между $B(n, x)$ и $B(n + 1, x)$.

15. Поставьте знак отношения между $B(n, x)$ и 2^x.

16. Укажите зависимость между диагональной функцией Аккермана $A(x)$ и функцией Аккермана $B(n, x)$.

17. Какое отношение существует между множеством всех примитивно-рекурсивных функций и множеством всех частично-рекурсивных функций?

18. Является ли диагональная функция Аккермана примитивно-рекурсивной? Почему?

19. Сформулируйте тезис Черча.

20. Приведите формальное определение алгоритма.

2.9 Упражнения к разделу

Задачей выполнения упражнений является приобретение навыков даказательства примитивной или частичной рекурсивности заданной функции, усвоение принципов построения рекурсивного определения простых функций.

2.9.1 Задача 1

Доказать примитивную рекурсивность функции

$$f(x,y) = [log_2(x^{y+2\cdot x} + 2)].$$

Решение. Рассмотрим следующие вспомогательные функции:

$$\begin{array}{l} g_1(x,y) = x + y, \\ g_2(x,y) = x \cdot y, \\ g_3(x,y) = x^y, \\ g_4(x) = [log_2(x+1)]. \end{array} \qquad l$$

Примитивную рекурсивность функции $g_1(x,y) = x + y$ мы доказали в 1.2. Докажем примитивную рекурсивность остальных функций. Рассмотрим сначала $g_2(x,y) = x \cdot y$. В соответствии с определением, некоторая функция является примитивно-рекурсивной, если выполняется одно из следующих условий:

а) эта функция является простейшей;

б) эта функция получена с помощью оператора примитивной рекурсии из функций, примивная рекурсивность которых уже доказана;

в) эта функция получена с помощью оператора суперпозиции из функций, примивная рекурсивность которых уже доказана.

Попробуем построить схему примитивной рекурсии для функции $g_2(x,y)$. В соответствии с определением этой функции:

$$\begin{array}{l} g_2(x,0) = x \cdot 0 = 0 = 0^1(x); \\ g_2(x,y+1) = x \cdot (y+1) = x \cdot y + x = g_2(x,y) + x = g_1(g_2(x,y),x). \end{array}$$

Покажем соответствие полученного рекурсивного определения функции $g_2(x,y) = x \cdot y$ определению оператора примитивной рекурсии R и докажем примитивную рекурсивность тех функций, из которых с помощью оператора R получена искомая функция. В соответсвии с определением, функция двух аргументов $t(x,y)$ построена с помощью оператора $R^2(r,h)$, если она имеет вид:

$$\begin{array}{l} t(x,0) = r(x); \\ t(x,y+1) = h(x,y,t(x,y)). \end{array} \qquad (2.10)$$

Рассмотрим вид функций $r(x)$ и $h(x,y,z)$ в нашем случае:

$$\begin{array}{l} g_2(x,0) = 0^1(x); \\ g_2(x,y+1) = g_1(g_2(x,y),x) = g_1(I_3^3(x,y,g_2(x,y)), I_1^3(x,y,g_2(x,y))). \end{array}$$

Таким образом, имеем

$$\begin{array}{l} r(x) = 0^1(x), \\ h(x,y,z) = g_1(I_3^3(x,y,g_2(x,y)), I_1^3(x,y,g_2(x,y))). \end{array}$$

Суперпозиция примитивно–рекурсивных функций $g_1(x,y)$, $I_3^3(x,y,z)$, $I_1^3(x,y,z)$ примитивно рекурсивна, следовательно и функция $g_2(x,y)$ является примитивно–рекурсивной.

Аналогично можно доказать примитивную рекурсивность функции $g_3(x,y) = x^y$. Заметим только, что примитивно–рекурсивная по определению функция выбора $I_m^n(x_1,x_2,...x_n)$ всегда позволяет ввести в выражение нужные аргументы и записать

аргументы в том порядке, который соответствует схеме примитивной рекурсии. Поэтому в дальнейшем не будем представлять функцию в форме, строго согласованной с порядком аргументов функции h(x,y,z) в (2.10). Например, для функции $g_3(x,y)$ имеем

$$g_3(x,0) = x^0 = 1 = s(0(x));$$
$$g_3(x,y+1) = x^{(}y+1) = x^y \cdot y = g_2(y, g_3(x,y)).$$

Функция $g_3(x,y)$ получена оператором примитивной рекурсии из примитивно-рекурсивных функций $s(0(x))$, $g_2(x,y)$ и, следовательно, примитивно-рекурсивна.

Доказательство для функции $g_4(x) = [log_2(x+1)]$ выполняется с помощью ограниченного оператора минимизации. Действительно, $g_4(x) = [log_2(x+1)] = \mu_{z\le x+1}(2^{z+1} > x+1)$, причем ограничивающая функция $x+1 = s(x)$ и характеристическая функция предиката

$$\chi(x,z) = \begin{cases} 0, & 2^{z+1} \le x+1 \\ 1, & 2^{z+1} > x \end{cases} = \begin{cases} 0, & 2^{z+1} \dot{-} (x+1) = 0 \\ 1, & 2^{z+1} \dot{-} (x+1) > 0 \end{cases} = sg(2^{z+1} \dot{-} (x+1))$$

примитивно–рекурсивны. Следовательно, и $g_4(x)$ — примитивно–рекурсивная функция.

Вернемся теперь к поставленной задаче доказательства примитивной рекурсивности функции

$$\begin{aligned} f(x,y) &= [log_2(x^{y+2\cdot x}+2)] = g_4(x^{y+2\cdot x}+2-1) = \\ &= g_4(x^{y+2\cdot x}+1) = g_4(s(x^{y+2\cdot x})) = \\ &= g_4(s(g_2(x, y+2\cdot x))) = g_4(s(g_2(x, g_1(y, 2\cdot x)))) = \\ &= g_4(s(g_2(x, g_1(y, g_1(x,x))))). \end{aligned}$$

Функция $f(x,y)$ является примитивно-рекурсивной как суперпозиция примитивно-рекурсивных функций.

2.9.2 Варианты заданий

1. $f(x,y) = [log_2((x+1)\cdot(y^{y+2x}) + 2^x] + 2 + \left[\sqrt{y}\right]$.

2. $f(x,y) = \left[x\sqrt{3}\right] + \left[\sqrt[1+x+y]{xy+2x}\right] + 2^{2x+y^2}$.

3. $f(x,y) = x^{y^{y+1^{\cdots^{2y+2}}}}$.

4. $f(x,y) = x^{(y+1)^{(y+2)^{\cdots^{2y+3}}}}$.

5. $f(x) = \left[(x^2+x)\sqrt{4+x^3+2x}\right] + \left[\sqrt[5]{x^3+2x+1}\right] + 8^{2x+2}$.

6. $f(x) = \sum\limits_{i=1}^{2^x+3} \left[\sqrt{x^i+3^i+i^x} + \sqrt{2x+2+i}\right]$.

7. $f(x,y,z) = \left[log_{2y+x+1}((y+3)^{(y+2\cdot x)!})\right] + \{\frac{x}{y+2}\}$.

8. $f(x,y,z) = \{\frac{x\sqrt{3}}{3}\} + \left[\sqrt[1+x+y]{xy+2zy^2}\right] + 2^{2x+z^2}$.

9. $f(x,y) = \left[x\sqrt{4+x^2}\right] + \left[\sqrt[5]{x^y+2x+1}\right] + 8^{2x+y^3}$.

10. $f(x,y,z) = \left[(z^2+y)\sqrt{4+x^3+2x=z^4}\right] + \left[\sqrt[3]{x^z+2x+1}\right]$.

11. $f(x,y) = [log_5(y^{y+2\cdot x}) + x!] + [\sqrt{x}] + \left[\sqrt{y}\right]$.

12. $f(x) = \sum\limits_{i=0}^{x^x+4} \{\frac{x+2^i+i^3}{4x+1+i}\}$.

13. $f(x,y) = \prod\limits_{i=1}^{2^x+3} \left[\sqrt{x^i+2^i+2^y} + \sqrt{3y+i}\right]$.

14. $f(x,y,z) = \prod\limits_{i=0}^{x+z} \sum\limits_{j=0}^{iy} \left[\sqrt{i+2^j+2^{z+x+ij}} + \sqrt{3j+i}\right]$.

15. $f(x) = \left[(x^2+y)\sqrt{4+x^3+2x}\right] + \left[\sqrt[5]{x^3+2x+1}\right] + 8^{2x+2}$.

16. $f(x) = \left[5x\sqrt{2}\right] + \left[\sqrt[x+2]{x+2x^4}\right] + 2^{2x+2}$.

17. $f(x,y) = [1 + log_3(5^{2y+2+x}) + 2^x] + \{\frac{x+y}{y^3+3}\}$.

18. $f(x,y) = \left[xy\sqrt{x^2+y^2}\right] + \left[\sqrt[1+x]{xy+2^x}\right] + x^{2+y}$.

19. $f(x) = \sum\limits_{i=0}^{2x+3} \left[\frac{x^i+3^i+i^x}{2x+2+i}\right]$.

20. $f(x,y) = [lg(y(y+2^{x+y+2})) + 2^x] + \{\frac{x}{y^2+2^x}\}$.

2.9.3 Задача 2

Доказать примитивную рекурсивность функции $p(x)$ – простое число с номером x. В соответствии с доказательством написать программу вычисления этой функции.

Решение. Рассмотрим сначала вспомогательную функцию $h(x)$, равную числу делителей числа x. Очевидная формула

$$h(x) = \sum_{i=1}^{x} (\overline{sg}(\{x/i\})),$$

(где выражение $\{x/i\}$ означает остаток от деления x на i) доказывает примитивную рекурсивность функции $h(x)$, являющейся суперпозицией примитивно-рекурсивных функций. Простые числа и только они имеют ровно два делителя — единицу и самого себя. Числа 0 и 1 простыми не являются. Все числа, не являющиеся простыми, кроме 0 и 1, имеют число делителей, большее двух. Тогда характеристическая функция $\chi_p(x)$ предиката $\langle x$ — простое число при $x > 1\rangle$ равна

$$\chi_p(x) = \overline{sg}(h(x) \dot{-} 2)$$

и является примитивно–рекурсивной.

Одной из наиболее известных арифметических функций является функция $\pi(x)$, равная числу простых чисел, не превосходящих x. Используя уже известную нам примитивно–рекурсивную функцию $\chi_p(x)$, получим

$$\pi(x) = \sum_{i=1}^{x} (\chi_p(i).$$

Отсюда следует, что искомая функция $p(x)$ – простое число с номером x – представляет собой минимальное решение уравнения

$$\pi(z) = x.$$

Таким образом,

$$p(x) = \mu_z(\pi(z) = x).$$

Для доказательства примитивной рекурсивности функции $p(x)$ достаточно ввести верхнюю границу изменения z и доказать примитивную рекурсивность как этой границы, так и характеристической функции предиката $p(x) = x$. Последнее легко сделать, т.к. эта функция равна

$$sg((\pi(z)\dot{-}x) + (x\dot{-}\pi(z))).$$

Осталось рассмотреть границу изменения z. Из теории чисел хорошо известно, что $p_n < 2^{2^n}$. В самом деле, требующееся нам неравенство заведомо истинно при $n = 0$. По индукции далее предполагаем, что это неравенство истинно для всех значений n, докажем его для $n + 1$.

По предположению имеем

$$p_0 < 2^{2^0},$$
$$p_1 < 2^{2^1},$$
$$\dots$$
$$p_n < 2^{2^n}.$$

Тогда

$$p_0 \cdot p_1 \cdot \dots \cdot p_n + 1 < 2^{2^0} \cdot 2^{2^1} \cdot \dots \cdot 2^{2^n} + 1 = 2^{2^0+2^1+\dots+2^n} + 1 < 2^{2^{n+1}}.$$

Число $a = p_0 \cdot p_1 \cdot \dots \cdot p_n + 1$ больше единицы и поэтому должно иметь какой-то простой делитель p_r. Этот делитель не может совпадать ни с одним из простых чисел $p_0, p_1, \dots p_n$, так как при делении числа a на любое из чисел $p_0, p_1, \dots p_n$ получается остаток 1. Но все простые числа, не превосходящие p_n, содержатся в последовательности p_0 , p_1 , ... p_n. Число p_r в нее не входит и, следовательно, $p_{n+1} \le p_r$. Так как $p_r \le a$, то

$$p_{n+1} \le a < 2^{2^{n+1}},$$

что и требовалось доказать.

Итак, неравенство доказано, а вместе с ним доказана и примитивная рекурсивность функции $p(x)$, имеющей своим значением простое число с номером x.

Напишем теперь программу вычисления этой функции, соответствующую построенному доказательству. Сразу отметим, что доказательство примитивной рекурсивности функции не ставит своей целью построение алгоритма, обладающего хорошими временными характеристиками. Такое доказательство обосновывает лишь существование алгоритма решения поставленной задачи, причем свойство определенности всюду для примитивно–рекурсивных функций означает, что для любых исходных данных соответствующая программа получит результат через некоторое конечное (может быть, очень большое) время.

Приведенному выше доказательству соответствует следующая программа.

```
#include <stdio.h>
#include <STDLIB.H>
```

```
int sg(int x) { return x>0; }

int h(long int x)
{
long int i; int s=0;
for (i=1; i<=x; i++ )   s+=1-sg(x%i);
return s;
}

int Xi(long int x)
{
if (x==1) return 1;
if (x==0) return 0;
return 1-sg(h(x)-2);
}

long int Pi( long int x)
{
long int i,s=0;
for (i=1; i<=x; i++ )s+=Xi(i);
return s;
}

long int p( long int x)
{
long int z=0;
while (Pi(z)!=x)  z++;
return z;
}

void main(void)
{
int  x; scanf("%d",&x);
unsigned long int z=p(x);
printf("простое число с номером %d равно %ld\n",x,z);
}
```

Следует отметить, что мы доказали алгоритм для любого $x \in N$. Однако, ограничение разрядной сетки компьютера не позволяет на уровне команд выполнить операции над сколь угодно большими числами. В нашей программе мы ограничили получаемые значения типом long int. Если программа должна работать с большими числами, необходимо реализовать так называемую длинную арифметику.

2.9.4 Варианты заданий

1. $f(x)$ – количество ненулевых цифр в десятичной записи числа x.

2. $f(x)$ – числитель следующего непрерывного многочлена x – – ой степени

$$1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+...\cfrac{1}{1+\cfrac{1}{x}}}}}.$$

Если в результате вычислений получается сократимая дробь, деление числителя и знаменателя на их наибольший общий делитель не выполнять.

3. $f(x,y)$ – наименьшее общее кратное чисел x и y.

4. $f(x,y)$ – минимальное простое число, принадлежащее отрезку $[x,y]$.

5. $f(x)$ – простое число с номером x; простые числа нумеровать, начиная от 0.

6. $f(x)$ – количество ненулевых цифр в троичном представлении числа x.

7. $f(x,y)$ – максимальное простое число, принадлежащее отрезку $[x,y]$. Если такие числа отсутствуют, значение функции равно 0.

8. $f(x)$ – номер наибольшего простого делителя числа x, где $f(0)=0$.

9. $f(x)$ – количество единиц в шестнадцатиричном представлении числа x.

10. $f(x,y)$ – число простых чисел, не превосходящих $x+y$.

11. $f(x)$ – количество ненулевых цифр в шестнадцатиричном представлении заданного числа x.

12. $f(x)$ – произведение удвоенных делителей числа x.

13. $f(x,y)$ – число простых чисел, не превосходящих $x+y$.

14. $f(x)$ – нечетное число Фибоначчи с номером x. Числа Фибоначчи определяются следующим соотношением

$$F(0)=0, F(1)=1,$$
$$F(n)=F(n-1)+F(n-2) \quad \text{для} \quad n>1.$$

Например, последовательность первых чисел Фибоначчи имеет вид:

$$0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,...$$

15. $f(x)$ – четное число Фибоначчи с номером n (см. пояснение к предшествующему упражнению).

16. $f(x)$ – число Стирлинга второго рода с номером x. Число Стирлинга первого рода $\left\{ {n \atop k} \right\}$ равно количеству способов разбиения множества из n элементов на k непустых подмножеств. Так, например, $\left\{ {4 \atop 2} \right\}=7$ и определяет число способов разбиения четырехэлементного множества на две части:

$$\{1,2,3\}\cup\{4\},\{1,2,4\}\cup\{3\},\{1,3,4\}\cup\{2\},\{2,3,4\}\cup\{1\},$$
$$\{1,2\}\cup\{3,4\},\{1,3\}\cup\{2,4\},\{1,4\}\cup\{2.3\}.$$

Обратите внимание, что фигурные скобки используются для обозначения как множеств, так и чисел $\left\{ {n \atop k} \right\}$. Подобное сходство помогает понять и запомнить смысл обозначения $\left\{ {n \atop k} \right\}$, которое может быть прочитано как "k подмножеств из n". Для $k=0$ принимается $\left\{ {0 \atop 0} \right\}=1$, $\left\{ {n \atop 0} \right\}=0$ при $n>0$.

17. $f(x)$ – число Стирлинга первого рода с номером x. Числом Стирлинга первого рода $\left[{n \atop k} \right]$ отчасти похожи на числа Стирлинга второго рода (см. предшествующее задание) с тем отличием, что число $\left[{n \atop k} \right]$ подсчитывает число способов представления n объектов в виде k циклов вместо представления в виде подмножеств. Обозначение $\left[{n \atop k} \right]$

произносится как "k циклов из n". Цикл $[A, B, C, ..., D]$ равен циклу $[B, C, ..., D, A]$, поскольку конец цикла соединен с его началом. Таким образом, для цикла из четырех элементов выполняется равенство

$$[A, B, C, D] = [B, C, D, A] = [C, D, A, B] = [D, A, B, C] .$$

Существует одиннадцать различных способов составить два цикла из четырех элементов:

$$[1, 2, 3] [4] , [1, 2, 4] [3] , [1, 3, 4] [2] , [2, 3, 4] [1] ,$$
$$[1, 3, 2] [4] , [1, 4, 2] [3] , [1, 4, 3] [2] , [2, 4, 3] [1] ,$$
$$[1, 2] [3, 4] , [1, 3] [2, 4] , [1, 4] [2.3] .$$

Следовательно, $\begin{bmatrix} 4 \\ 2 \end{bmatrix}$ =11.

18. $f(x)$ – число Эйлера с номером x. Число Эйлера $\left\langle {n \atop k} \right\rangle$ равно числу перестановок множества $\{1, 2, 3..., n\}$, имеющих k участков подъема, т.е. k мест в перестановке $a_1 a_2 ... a_n$, где $a_j < a_{j+1}$. Например, $\left\{ {4 \atop 2} \right\} = 11$, т.к. одиннадцать перестановок множества $\{1, 2, 3, 4\}$ (из всех $4! = 34$ перестановок) содержат по два участка подъема:

$$1324, 1423, 2314, 2413, 3412,$$
$$1243, 1342, 2341;$$
$$2134, 3124, 4123.$$

В первой строке перечислены перестановки с отношениями $a_1 < a_2 > a_3 < a_4$, во второй строке перечислены перестановки с отношениями $a_1 < a_2 < a_3 > a_4$, и в третьей – с отношениями $a_1 > a_2 < a_3 < a_4$.

19. $f(x)$ – числитель несократимого гармонического числа с номером x. Гармоническим числом называется

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n} = \sum_{k=1}^{n} \frac{1}{k}$$

20. $f(x)$ – знаменатель несократимого гармонического числа с номером x (см. предшествующее задание).

2.10 Тесты для самоконтроля к разделу

1. Функцию $(2n + 1)!$ необходимо представить рекурсивно. Постройте схему примитивной рекурсии для этой функции.

Варианты ответов:

1) $f(0) = 1;$
   $f(n + 1) = 2 * f(n) * (n + 1) * (2n + 3);$
2) $f(0) = 1;$
   $f(1) = 6;$
   $f(n) = (2n + 1)!;$
3) $f(0) = 1;$
   $f(2n + 1) = f(2n) * (2n + 1);$
4) $f(n) = 2n * f(n - 1) * (2n + 1);$
5) $f(0) = 1;$
   $f(n) = 2n * f(n - 1) * (2n + 1).$

Правильный ответ: 1.

2. Что означает свойство детерминированности алгоритма?

Варианты ответов:

1) алгоритм всегда останавливается после фиксированного числа шагов;

2) алгоритм всегда останавливается, решая задачу на некоторых данных, но число выполненных шагов зависит от конкретных исходных данных;

3) в каждый фиксированный момент времени вычислитель, выполняющмй алгоритм, имеет отображенными в своей памяти все данные, полученные в процессе вычислений;

4) последовательность действий, соответствующая алгоритму, всегда выполняется одинаково на одних и тех же данных;

5) последовательность инструкций, представляющая алгоритм, имеет конечные размеры.

Правильный ответ: 4.

3. Какая функция получена с помощью оператора суперпозиции $S(f_0, f_1, f_2)$ из функций $f_0(x, y) = xy + x^2, f_1(x) = 2x + 1, f_2(x) = 2^(x + 2)$?

Варианты ответов:

1) $f(x, y) = (2x + 1) \cdot 2^{y+2}$;
2) $f(x) = (2x + 1) \cdot (2x + 2^{x+2} + 1)$;
3) $f(x, y) = xy + x^2 + (2x + 1) + 2^{y+2}$;
4) $f(x) = 2^{x+2} \cdot (2x + 1)$;
5) $f(x) = x \cdot (2x + 1) + (2^{x+2})^2$;
6) $f(x, y) = (xy + x^2) \cdot (2x + 1) \cdot 2^{x+2}$.

Правильный ответ: 2.

4. Какие из следующих утверждений истинны?

1) Если функция f построена с помощью оператора минимизации из примитивно–рекурсивных функций, то f может быть как примитивно–рекурсивной, так и частично–рекурсивной функцией.

2) Если функция f построена с помощью ограниченного оператора минимизации из примитивно-рекурсивных функций, то f может не являться примитивно-рекурсивной, но она всегда является частично-рекурсивной функцией.

3) Понятие примитивно-рекурсивной функции эквивалентно определению алгоритма.

4) Если функция частично-рекурсивна, то она и примитивно-рекурсивна.

5) Если функция примитивно-рекурсивна, то она и частично-рекурсивна.

Правильный ответ: 1 и 5.

5. Поясните назначение тезиса Черча и дайте информацию о его доказательстве.

Варианты ответов:

а) Тезис Черча является определением частично-рекурсивной функции;

б) Тезис Черча является определением примитивно-рекурсивной функции;

в) Тезис Черча доказан Черчем;

г) Тезис Черча — это предположение, высказанное Черчем, еще не доказанное к настоящему моменту, но ведутся интенсивные поиски такого доказательства;

д) Тезис Черча не может быть доказан, так как является определением алгоритма.

Правильный ответ: д.

Глава 3

МАШИНЫ ТЬЮРИНГА

В предшествующей главе мы рассмотрели определение алгоритма с функциональной точки зрения. Рассмотрим теперь сам *процесс выполнения алгоритма* и в основу формального определения алгоритма положим вычислительный процесс, который выполняется на некотором абстрактном вычислителе. Стремясь найти точное определение понятия "эффективной вычислимости Тьюринг выделил некоторый класс абстрактных машин, о которых высказал предположение, что они пригодны для осуществления любой "механической " вычислительной процедуры. Эти машины называют теперь в честь их автора машинами Тьюринга.

3.1 Неформальное определение машины Тьюринга

Машина Тьюринга (МТ) — это автомат, который имеет потенциально бесконечную в обе стороны ленту, считывающую головку и управляющее устройство.

Лента разделена на *ячейки*, в одной ячейке или записан один символ некоторого алфавита или она пуста. Потенциальная бесконечность ленты понимается в том смысле, что в каждый данный момент времени она имеет конечную заполненную часть, и вместе с тем к этой заполненной части всегда можно добавить как слева, так и справа любые заполненные ячейки. Таким образом, в каждый момент функционирования машины Тьюринга может быть заполнено только конечное число ячеек. Имеется некоторое конечное множество символов ленты, которое называется *алфавитом* машины. В каждый момент времени каждая ячейка ленты может быть заполнена не более чем одним символом. Наконец, имеется *читающая головка*, которая в каждый данный момент времени обозревает одну из ячеек ленты. Машина действует не непрерывно, а по *тактам*, в дискретные моменты времени. На каждом такте работы машина Тьюринга может считать символ, записать вместо него новый и сдвинуть головку на одну ячейку или влево, или вправо или оставить ее на месте. Для того, чтобы с пустой ячейкой машина Тьюринга могла действовать так же, как и с заполненной некоторым символом ячейкой, вводится специальный символ алфавита, обозначающий пустую ячейку. Таким образом, всегда можно считать, что в любой момент времени читающая головка обозревает ячейку с некоторым символом алфавита.

Машина обладает некоторым конечным *множеством внутренних состояний*. В каждый момент времени машина находится в точности только в одном из этих состояний. Управляющее устройство ответственно за функционирование машины Тьюринга по тактам.

Таким образом, на каждом такте управляющее устройство находится в каком—либо состоянии из конечного множества состояний и обозревает символ из конечного алфавита ленты. В зависимости от текущего состояния и прочитанного символа записывает вместо прочитанного символа новый, сдвигает головку на одну позицию или оставляет ее на месте и переходит в новое состояние. В множестве состояний выделяются два специальных состояния: начальное и заключительное. Машина Тьюринга начинает работу из начального состояния и завершает ее, когда переходит в заключительное состояние.

3.2 Формальное определение машины Тьюринга

Введем сначала некоторые общие определения, связанные с представлением информации для машин Тьюринга.

Определение 3.1. Алфавит —некоторое конечное множество символов.

Определение 3.2. Цепочка над алфавитом —это последовательность символов алфавита.

Например, над алфавитом $\Sigma = \{0, 1\}$, содержащим два символа 0 и 1 одна из цепочек имеет вид 00010.

Определение 3.3. Длина цепочки —это число символов в цепочке.

Длина пустой цепочки равна нулю. Обычно пустая цепочка обозначается ε. Таким образом, $|\varepsilon| = 0$, а $|00010| = 5$.

Множество всех цепочек над алфавитом Σ обозначается Σ^*.

Следует отметить, что пустая цепочка ε является цепочкой над любым алфавитом. Например, множество всех цепочек над алфавитом $\{0, 1\}$ — это множество $\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, ...\}$.

Определение 3.4. Языком над алфавитом Σ называется некоторое подмножество множества Σ^*.

Определение 3.5. Конкатенацией цепочек x и y называется цепочка, полученная приписыванием символов цепочки y к цепочке x справа.

Для произвольной цепочки x и пустой цепочки ε справедливо равенство $x\varepsilon = \varepsilon x = x$. Таким образом, цепочка ε играет роль единицы для операции конкатенации.

Определение 3.6. Машиной Тьюринга называется семерка вида:

$$T = (K, \Sigma, \delta, p_0, f, a_0, a_1), \text{ где}$$

K — конечное множество состояний,

Σ — алфавит ленты,

p_0 — начальное состояние,

f — заключительное состояние,

a_0 — символ для обозначения пустой ячейки (если это не будет вызывать неоднозначности, будем обозначать его ε), $a_0 \in \Sigma$,

a_1 — специальный символ — разделитель цепочек на ленте (обычно для этой цели будем использовать знак "*"), $a_1 \in \Sigma$,

δ — функция переходов, которая описывает поведение машины Тьюринга и представляет собой отображение вида

$$\delta : K \times \Sigma \to K \times \Sigma \times S,$$

где $S = \{R, L, E\}$ — направления сдвига головки по ленте.

Вводя определение машаны Тьюринга мы преследовали одну цель — дать формальное определение алгоритма. Рассмотрим соответствие введенного определения неформальным требованиям к алгоритму, сформулированным в 1.1. В соответствии с этими требованиями в определении машины Тьюринга можно выделить следующие характерные черты:

1) имеется вычислитель — сама машина Тьюринга;

2) машина Тьюринга работает по тактам, что соответствует дискретности выполнения алгоритма;

3) соблюдается требование конечности алгоритма, т.к. множества K и Σ конечны, а, следовательно, конечно и отображение δ;

4) требование детерминированности Машины Тьюринга соответствует детерминированности отображения δ (это значит, что множество команд машины Тьюринга не содержит различных команд с одинаковыми левыми частями).

Таким образом, можно сделать следующий вывод: для согласования определения машины Тьюринга с требованиями к алгоритму требуется рассматривать только детерминированные машины Тьюринга. Кстати, это требование уже учтено в определении функции переходов как

$$\delta : \ K \times \Sigma \to K \times \Sigma \times S.$$

Функция переходов недетерминированной машины Тьюринга могла бы иметь вид

$$\delta : \ K \times \Sigma \to K \times 2^{\Sigma \times S}$$

или

$$\delta : \ K \times \Sigma \to 2^{K \times \Sigma \times S}.$$

Полное состояние машины Тьюринга, по которому однозначно можно определить ее дальнейшее поведение, определяется ее внутренним состоянием, словом, записанным на ленте, и положением головки на ленте.

Определение 3.7. Конфигурацией машины Тьюринга $T = (K, \Sigma, \delta, p_0, f, a_0, a_1)$ называется

$$t = \langle \alpha q a \beta \rangle, \text{ где}$$

α — цепочка слева от головки, $\alpha \in \Sigma^*$;

q — состояние машины Тьюринга, $q \in K$;

a — символ под головкой, $a \in \Sigma$;

β — цепочка справа от головки, $\beta \in \Sigma^*$.

Например, конфигурация $\langle 11 * 1 q_6 * * \rangle$ означает, что машина Тьюринга находится в состоянии q_6, имеет на ленте цепочку $11 * 1 * *$, причем, слева от головки находится часть этой цепочки $11 * 1$, читающая головка обозревает символ $*$, справа от головки находится цепочка $*$.

Элемент функции переходов $qa \to pbr$ (где $q, p \in K$, $a, b \in \Sigma$, $r \in S$) называется командой машины Тьюринга и описывает один такт ее функционирования. Один такт работы означает следующее: если в состоянии q машина Тьюринга обозревает на ленте символ a, то она переходит в состояние p, записывает вместо a новый символ b и сдвигает головку в направлении r. Дадим формальное определение одного такта работы машины Тьюринга в терминах конфигураций.

Определение 3.8. Конфигурация $\langle \alpha q a \beta \rangle$ непосредственно переходит в конфигурацию $\langle \alpha_n q_n a_n \beta_n \rangle$, если новая конфигурация получилась в результате применения одной команды к исходной конфигурации:

1) команда $qa \to q_n a_n E$, тогда $\alpha = \alpha_n, \beta = \beta_n$,

2) команда $qa \to q_n bR$, тогда

а)$\beta \neq \varepsilon, \beta = a_n\beta_n, \alpha_n = \alpha b$,

б) $\beta = \varepsilon, a_n = a_0$ (или ε), $\beta_n = \varepsilon, \alpha_n = \alpha b$,

3) команда $qa \to q_n bL$, тогда

а) $\alpha \neq \varepsilon, \alpha = \alpha_n a_n, \beta_n = b\beta$,

б) $\alpha = \varepsilon, a_n = a_0$ (или ε), $A_n = \varepsilon, \beta_n = b\beta$,

4) если в множестве команд отсутствует команда с левой частью qa, то машина Тьюринга оказывается блокированной, т.е. она никак не реагирует на символ a и до бесконечности продолжает оставаться в одной и той же конфигурации.

Обозначим непосредственный переход

$$\langle \alpha q a \beta \rangle \Rightarrow \langle \alpha_n q_n a_n \beta_n \rangle.$$

Имея формальное определение одного такта работы машины Тьюринга, можно теперь определить ее поведение в целом.

Определение 3.9. Конфигурация t переходит в конфигурацию p (обозначается $t \overset{*}{\Rightarrow} p$), если существуют такие конфигурации $t_1, t_2, \ldots, t_k$, что

$$t = t_1 \Rightarrow t_2 \Rightarrow t_3 \Rightarrow \ldots \Rightarrow t_k = p.$$

Определение 3.10. Конфигурация

$$\langle \alpha p_0 a \beta \rangle$$

машины Тьюринга $T = (K, \Sigma, \delta, p_0, f, a_0, a_1)$, содержащая начальное состояние p_0, называется начальной, а конфигурация

$$\langle \alpha f a \beta \rangle,$$

содержащая заключительное состояние f, называется заключительной, причем если в данной конфигурации цепочка α слева от головки пустая ($\alpha = \varepsilon$), то соответственно начальная конфигурация называется стандартной начальной конфигурацией, а заключительная — стандартной заключительной.

Определение 3.11. Машина Тьюринга $T = (K, \Sigma, \delta, p_0, f, a_0, a_1)$ перерабатывает цепочку φ в цепочку ϕ, если, действуя из начальной конфигурации, имея цепочку φ на ленте, машина Тьюринга переходит в заключительную конфигурацию, имея цепочку ϕ на ленте: $\varphi_1 p_0 \varphi_2 \overset{*}{\Rightarrow} \phi_1 f \phi_2$, где $\varphi_1\varphi_2 = \varphi$, $\phi_1\phi_2 = \phi$.

Если начальная и заключительная конфигурации стандартные, то процесс переработки цепочки φ в цепочку ϕ называется правильной переработкой: $p_0\varphi \overset{*}{\Rightarrow} f\phi$.

3.3 Способы представления машины Тьюринга

Машину Тьюринга можно задать перечислением семи объектов в соответствии с определением 3.6. Для наглядного представления машин Тьюринга на практике используются следующие способы: перечисление множества команд, задание машины Тьюринга в виде графа, формирование таблицы переходов.

Рассмотрим представление машины Тьюринга $T = (K, \Sigma, \delta, q_0, f, a_0, a_1)$ с помощью множества команд. Допустим, мы описали алгоритм на неформальном уровне и представили его в виде упорядоченной последовательности некоторых выполняющихся пунктов "1 "2 "3 ... Тогда для того, чтобы этот алгоритм записать в виде множества команд машины Тьюринга, можно воспользоваться следующими правилами формирования команд:

а) начальному пункту алгоритма ставится в соответствие начальное состояние q_0 машины Тьюринга;

б) циклы в алгоритме реализуются так, чтобы последнее действие цикла соответствовало переходу в то состояние, которое соответствует началу цикла;

в) последовательное выполнение пунктов алгоритма обеспечивается переходом в соответствующие этим пунктам смежные состояния;

г) последний пункт алгоритма вызывает переход в заключительное состояние f.

Пример. Построим машину Тьюринга, которая для заданной цепочки из 0 и 1 стороит ее инверсию, т.е. заменяет в цепочке все знаки 0 на 1 и знаки 1 на 0. Будем предполагать, что в начальный момент машина Тьюринга обозревает первый символ исходной цепочки, а после преобразования цепочки головка возвращается к начальному символу результата. Тогда алгоритм можно представить в словесной форме.

а) Пока не ε, читаем 0 или 1, записывая инверсию прочитанных символов — символы 1 или 0 соответственно и сдвигаем головку право. Тем самым мы последовательно инвертируем каждый символ и в результате сдвигаем головку в позицию, непоспедственно следующую за преобразованной цепочкой. За преобразованной цепочкой находится ячейка с "пустым" символом.

б) Читаем ε, пишем ε, сдвигаем головку влево. Следовательно, головка обозревает последний символ результирующей цепочки. Теперь необходимо вернуться к началу цепочки.

в) Пока не ε, читаем 0 или 1 с восстановлением и сдвигаем головку влево (термин "читать с восстановлением" означает, что прочитанный символ записывается на ленту без изменения).

г) Читаем ε с восстановлением и сдвигаем головку вправо; тем самым установили головку на первый символ результирующей цепочки.

Опишем эти действия с помощью команд машины Тьюринга. Начальный пункт алгоритма (а) соответствует начальному состоянтю q_0. После выполнения пункта (г) машина Тьюринга должна перейти в заключительное состояние f. Цикл (а) реализуется так, что машина Тьюринга находится в одном и том же состоянии p_0. Это же касается и пункта (в): цикл выполняется в состоянии, соответствующем пункту (в). Таким образом, получим следующий набор команд:

$$
\begin{aligned}
&q_0 1 \to q_0 0 R,\\
&q_0 0 \to q_0 1 R,\\
&q_0 \varepsilon \to q_1 \varepsilon L,\\
&q_1 0 \to q_1 0 L,\\
&q_1 1 \to q_1 1 L,\\
&q_1 \varepsilon \to f \varepsilon R.
\end{aligned}
$$

Здесь алфавит ленты $\Sigma = \{0, 1, \varepsilon\}$, множество состояний $K = \{q_0, q_1, f\}$.

Теперь рассмотрим представление машины Тьюринга графом. Для того, чтобы представить машину Тьюринга в виде графа, надо каждое состояние сделать вершиной этого графа, а каждой команде поставить в соответствие помеченную дугу. Таким образом, команде $pa \to gbr$ соответствует дуга из вершины p в вершину q с меткой $a \to br$. Для того, чтобы обозначить начальную и конечную вершины, их обычно выделяют специальным образом.

Например, машинаТьюринга, инвертирующая цепочку из 0 и 1, может быть предствлена графом рис. 3.1.

Рассмотрим представление машины Тьюринга таблицей. Такая таблица также должна полностью отображать все множество команд. Если каждому состоянию по-

ставим в соответствие строку таблицы, а каждому символу — столбец, то для одной команды $pa \to qbr$ на пересечении строки p (исходное состояние p) и столбца a (читаемый символ a) указывается выполняемое действие qbr (записать симол b, перейти в состояние q и сдвинуть головку в направлении r).

Таблица переходов машины Тьюринга, инвертирующей цепочку из 0 и 1, имеет вид:

	ε	0	1
q_0	$q_1\varepsilon L$	$q_0 1R$	$q_0 0R$
q_1	$f\varepsilon R$	$q_1 0L$	$q_1 1L$

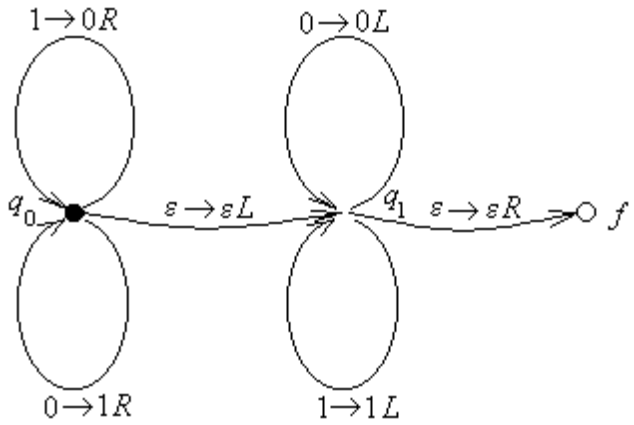


Рис. 3.1: Машина Тьюринга, инвертирующая цепочку из 0 и 1.

3.4 Функции, вычислимые по Тьюрингу

Мы можем с каждой машиной Тьюринга связать некоторый алгоритм. Возьмем произвольную цепочку над алфавитом Σ, запишем ее на ленте и запустим машину Тьюринга $T = (K, \Sigma, \delta, q_0, q_z, a_0, a_1)$ из начального состояния q_0. Если машина Тьюринга когда–нибудь остановится, то появившуюся на ленте цепочку можно рассматривать как результат работы алгоритма. Можно на любом алфавите рассматривать машины Тьюринга, которые:

а) никогда не прекращают работу, например:

$$\Sigma = \{1, \varepsilon\},$$
$$q_0 1 \to q_0 1E, \qquad q_0\varepsilon \to q_0\varepsilon E;$$

б) на любых исходных данных машина Тьюринга всегда закончит свою работу через конечное число шагов, например:

$$\Sigma = \{1, \varepsilon\}$$
$$q_0 1 \to q_z 1E, \qquad q_0\varepsilon \to q_z\varepsilon E;$$

в) на некоторых исходных данных машина Тьюринга работает бесконечно, а на некоторых завершает работу, например:

$$\Sigma = \{1, \varepsilon\},$$
$$q_0 1 \to q_0 1E,$$
$$q_0\varepsilon \to q_z\varepsilon E.$$

Определения, связанные с работой машины Тьюринга, даны в таком виде, что они допускают переработку любых нечисловых объектов, описанных в терминах алфавита. Перейдем теперь к анализу числовых функций, точнее, функций, отображающих $N \times N \times ... \times N$ в N. Рассматривая машины Тьюринга и функции, постараемся установить соответствие между зацикливанием машины Тьюринга и неопределенностью функции в некоторой точке, между завершением работы машины Тьюринга и определенностью функции.

Для простоты будем записывать натуральные числа в единичном (*унарном*) коде, в котором число k представляется словом

$$\underbrace{11...1}_{\text{k}} = 1^k,$$

состоящим из k единиц. Для разделения числовых аргументов 1^{x_1}, 1^{x_2},..., 1^{x_n} функции $f(x_1, x_2, ..., x_n)$ будем использовать символ – разделитель a_1. Используя знак "*" в качестве символа a_1, получим, что аргументы функции $f(x_1, x_2, ..., x_n)$ всегда имеют вид $1^{x_1} * 1^{x_2} * ... * 1^{x_n}$.

Пример. Пусть дана функция $f(x, y, z) = x+y+z$, тогда запись аргументов этой функции на ленте машины Тьюринга имеет вид $1^x * 1^y * 1^z$, а значение функции — вид 1^{x+y+z}.

Определение 3.12. Машина Тьюринга

$$T = (K, \Sigma, \delta, q_0, q_z, a_0, a_1)$$

вычисляет функцию $f(x_1, \ldots, x_n)$, если выполняются следующие условия:

1) для любых $x_1, \ldots, x_n$, принадлежащих области определения $Dom(f)$ функции f, машина Тьюринга из начальной конфигурации, имея на ленте представление аргументов $x_1, \ldots, x_n$, переходит в заключительную конфигурацию, имея на ленте представление значения функции:

$$\begin{aligned} &A_1 q_0 A_2 \stackrel{*}{\Rightarrow} B_1 q_z B_2, \\ &A_1 A_2 = 1^{x_1} * \cdots * 1^{x_n}, \\ &B_1 B_2 = 1^{f(x_1,\ldots,x_n)}; \end{aligned}$$

2) для любых $x_1, \ldots, x_n$, не принадлежащих $Dom(f)$, машина Тьюринга T из начальной конфигурации, имея на ленте представление аргументов $x_1, \ldots, x_n$, работает бесконечно:

$$A_1 q_0 A_2 \stackrel{*}{\Rightarrow} \infty.$$

Определение 3.13. Если начальная конфигурация является стандартной начальной кофигурацией, а заключительная — стандартной заключительной, то говорят, что машина Тьюринга правильно вычисляет функцию f:

$$\begin{aligned} &1) q_0 1^{x_1} * \ldots * 1^{x_n} \stackrel{*}{\Rightarrow} q_z 1^{f(x_1,\ldots,x_n)}, \\ &2) q_0 1^{x_1} * \ldots * 1^{x_n} \stackrel{*}{\Rightarrow} \infty. \end{aligned}$$

Определение 3.14. Функция называется вычислимой (правильно вычислимой), если существует машина Тьюринга, вычисляющая (правильно вычисляющая) эту функцию.

Обычно мы будем рассматривать правильную вычислимость функций, поэтому везде в дальнейшем, если не оговорено противное, под вычислимостью будем понимать правильную вычислимость.

Таким образом, для того, чтобы доказать вычислимость функции, а в перспективе и существование алгоритма, необходимо построить соответствующую машину Тьюринга. Непосредственные построения трудоемки, а часто представляют практически невыполнимый процесс в силу громоздкости и необозримости такого построения. Поэтому необходимо рассмотреть операции над машинами Тьюринга, чтобы получить инструменты для построения сложных машин Тьюринга из простых машин.

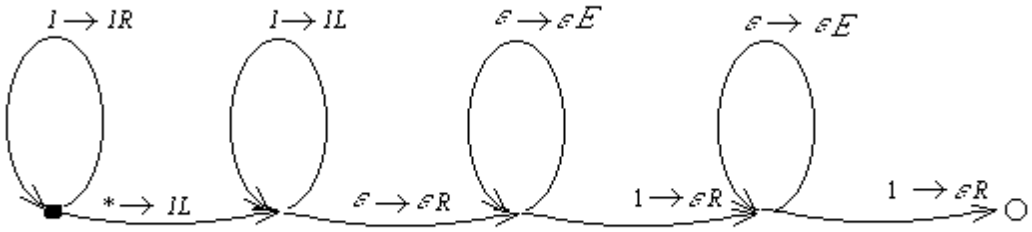
Пример. Построить машину Тьюринга, правильно вычисляющую функцию

$$f(x, y) = x + y - 1.$$

В соответствии с определением требуемая машина Тьюринга должна выполнять следующие действия:

$$q_0 \overset{*}{\Rightarrow} \begin{cases} \infty, & x + y = 0 \\ q_z 1^{x+y-1}, & x + y - 1 > 0 \end{cases}$$

Указанные действия выполняет следующая машина Тьюринга:



3.5 Машина Тьюринга с полулентой

Рассмотрим сначала ряд вспомогательных утверждений.

Теорема 3.1. Композиция двух вычислимых функций есть функция вычислимая.

Доказательство. Пусть

$$g(x) = f_2(f_1(x)), \text{ где}$$

f_1, f_2 — правильно вычислимые функции. Тогда существуют две машины Тьюринга, правильно вычисляющие f_1 и f_2:

$$T_1 = (K_1, \Sigma_1, \delta_1, p_{01}, p_{z1}, a_{01}, a_{11}), \tag{3.1}$$

$$T_2 = (K_2, \Sigma_2, \delta_2, p_{02}, p_{z2}, a_{02}, a_{12}). \tag{3.2}$$

Если нужно, переобозначим символы пустой ячейки и разделителя так, чтобы они совпадали: $a_{01} = a_{02}$, $a_{11} = a_{12}$, а также переобозначим K_1 и K_2 так, чтобы они не пересекались.

Построим машину Тьюринга:

$$T = (K_1 \cup K_2 \setminus \{p_{z1}\}, \Sigma_1 \cup \Sigma_2, \Big|_{p_{z1}}^{p_{02}} (\delta_1 \cup \delta_2), p_{01}, p_{z2}, a_0, a_1). \tag{3.3}$$

Построенная машина Тьюринга T выполняет следующие действия:

$$p_{01}1^x \stackrel{*}{\Rightarrow}_{(T_1)} p_{z1}1^{f_1(x)} = p_{02}1^{f_1(x)} \stackrel{*}{\Rightarrow}_{(T_2)} p_{z2}1^{f_2(f_1(x))}.$$

Если $g(x)$ не определена в точке x — это значит, что не определена либо $f_1(x)$, либо $f_2(t)$, где $t = f_1(x)$. В этом случае T зациклится соответственно либо на первом участке, работая как T_1, либо на втором, работая как T_2. □

Определение 3.15. Машина Тьюринга T называется композицией машин Тьюринга T_1 и T_2, если она построена по правилам (3.1), (3.3), (3.3).

Таким образом, мы получили алгоритм построения из двух машин Тьюринга такой новой машины Тьюринга, которая последовательно выполняет действия двух исходных машин.

Теорема 3.2. Композиция n правильно вычислимых функций $f_1(x)$, $f_2(x)$,...,$f_n(x)$, есть правильно вычислимая функция $f_1(f_2(\dots f_n(x)\dots))$.

Доказательсво. Воспользуемся принципом математической индукции. Для $n = 2$ теорема доказана — это теорема 3.1. Пусть теорема справедлива для некоторого $n \geq 2$, докажем ее для $n+1$. Имеется композиция $g(x) = f_1(f_2(...f_{n+1}(x)))$. Функция $f_2(...f_{n+1}(x))$ является композицией n вычислимых функций и, следовательно, вычислима по индуктивному предположению. Тогда композиция $f_1(g(x))$ двух вычислимых функций $f_1(x)$ и $g(x)$ является вычислимой по теореме 3.1. □

Рассмотренные нами определения машины Тьюринга использовали бесконечную ленту в обе стороны. Это значит, что на ленте нельзя оставить какие–нибудь данные, которые машина Тьюринга не будет использовать при движении влево или вправо. Ограничим ленту с одной стороны и покажем, что машина Тьюринга с полулентой (левой или правой) эквивалентна машина Тьюринга с бесконечной в обе стороны лентой.

Теорема 3.3. Функция, правильно вычислимая на машине Тьюринга с обычной лентой, правильно вычислима на машине Тьюринга с правой полулентой.

Доказательство. Главная идея доказательства основана на следующих положениях:

– ограничим рабочую область ленты двумя маркерами — неподвижным левым маркером $\triangle$ и подвижным правым $\otimes$;

– на внутренней части ограниченной области машина Тьюринга должна работать так, как обычная машина Тьюринга, а при выходе на маркеры она должна освобождать рабочее пространство, для чего правый маркер надо сдвинуть вправо, а при выходе на левый маркер придется сдвинуть всю цепочку;

– полученный результат, который находится где–то между маркерами, в конце работы необходимо сдвинуть вплотную к левому маркеру.

Итак, пусть $f(x_1, \dots, x_n)$ — правильно вычислимая функция, для которой существует машина Тьюринга

$$T_f : q_0 1^{x_1} * \dots * 1^{x_n} \stackrel{*}{\Rightarrow} q_z 1^{f(x_1,\dots,x_n)}.$$

Построим вспомогательную машину Тьюринга T_1, которая получает на вход такую же цепочку, что и T_f, и ограничивает эту цепочку двумя маркерами $\triangle$ и $\otimes$ слева и справа соответственно:

$$\begin{array}{lllll} T_1: & p_0 1 \to p_1 1 L, & p_0 * \to p_1 * L, & p_0 \varepsilon \to p_1 \varepsilon L, \\ & p_1 \varepsilon \to p_2 \triangle R, & & \\ & p_2 1 \to p_2 1 R, & p_2 * \to p_2 * R, & p_2 \varepsilon \to p_3 \otimes L \\ & p_3 1 \to p_3 1 L, & p_3 * \to p_3 * L, & p_3 \triangle \to p_z \triangle R. \end{array}$$

Построенная машина Тьюринга T_1 выполняет следующие действия:

$$T_1 : p_0 1^{x_1} * \ldots * 1^{x_n} \stackrel{*}{\Rightarrow} \triangle\ p_z 1^{x_1} * \cdots * 1^{x_n} \otimes .$$

Теперь можно считать, что машина Тьюринга T_f получает на вход цепочку, ограниченную маркерами. Преобразуем T_f так, чтобы на участке, ограниченном неподвижным и подвижным маркерами, новая машина Тьюринга T_{fn} выполняла те же действия, что и исходная машина Тьюринга.

Так как новая машина Тьюринга должна работать внутри ограниченной области так же, как T_f, то она должна содержать все команды этой машины Тьюринга. Разница в функционировании исходной T_f и конструируемой T_{fn} будет при выходе за границы обрабатываемых данных.

Исходная машина Тьюринга T_f при выходе за границу участка требует пустую ячейку, новая машина Тьюринга T_{fn} в этой же ситуации выходит на маркеры. Освободим место на ленте для пустой ячейки. После этого для обеспечения эквивалентности переработки цепочки машина Тьюринга T_{fn} должна действовать точно так же, как и T_f, следовательно, она должна вернуться в то состояние q_i, в котором ей потребовалась пустая ячейка.

Освободить ячейку в сторону подвижной границы очень просто — надо просто перенести подвижный маркер на одну ячейку вправо. Передвинуть левый маркер нельзя, придется передвинуть на одну ячейку вправо всю цепочку. Для этого введем дополнительные состояния q_0, q_1,...,q_k ($k = |\Sigma|$) и проведем сдвиг цепочки в процессе передвижения головки слева от неподвижного маркера к подвижному. На каждом такте прочитанный символ заменяется тем символом, который был прочитан на предшествующем шаге. Исключение составляет только первый шаг — на нем надо освободить пустую ячейку.

Схематически указанные действия машины тьюринга T_{fn} представлены на рис. 3.2.

Каждое из дополнительных состояний q_0, q_1,...,q_k новой машины Тьюринга T_{fn} соответствует символу, прочитанному на предыдущем шаге. Этот символ необходимо записать в обозреваемую ячейку вместо находящегося там символа и перейти в то состояние, которое соответствует этому символу.

Построенная машина Тьюринга выполняет следующие действия:

$$T_{fn} : \triangle\ p_0 1^{x_1} * \ldots * 1^{x_n} \otimes\ \stackrel{*}{\Rightarrow} \triangle\ a_0^r p_z 1^{f(x_1,\ldots,x_n)} a_0^m \otimes .$$

Построим вспомогательную машину Тьюринга T_2, которая начинает действовать из заключительной конфигурации T_{fn} и сдвигает последовательность единиц вплотную к левому маркеру.

$$\begin{array}{lll} T_2 : & p_0\varepsilon \to p_0\varepsilon R, & p_0 1 \to p_1\varepsilon L, \\ & p_1\varepsilon \to p_1\varepsilon L, & p_1 1 \to p_2 1 R, \quad p_1\triangle\ \to p_2\triangle\ R \\ & p_2\varepsilon \to p_0 1 R & \\ & p_0\otimes\ \to p_3\varepsilon L & \\ & p_3\varepsilon \to p_3\varepsilon L, & p_3 1 \to p_3 1 L, \quad p_3\triangle\ \to p_z\triangle\ R. \end{array}$$

Построенная машина Тьюринга выполняет следующие действия:

$$T_2 : \triangle\ a_0^r p_0 1^z a_0^m \otimes\ \stackrel{*}{\Rightarrow} \triangle\ p_z 1^z.$$

Рассмотрим композицию построенных машин Тьюринга: $T = T_1 \cdot T_{fn} \cdot T_2$. Она выполняет действия, соответствующие условию теоремы, следовательно функция f вычислима на машине Тьюринга с правой полулентой.□

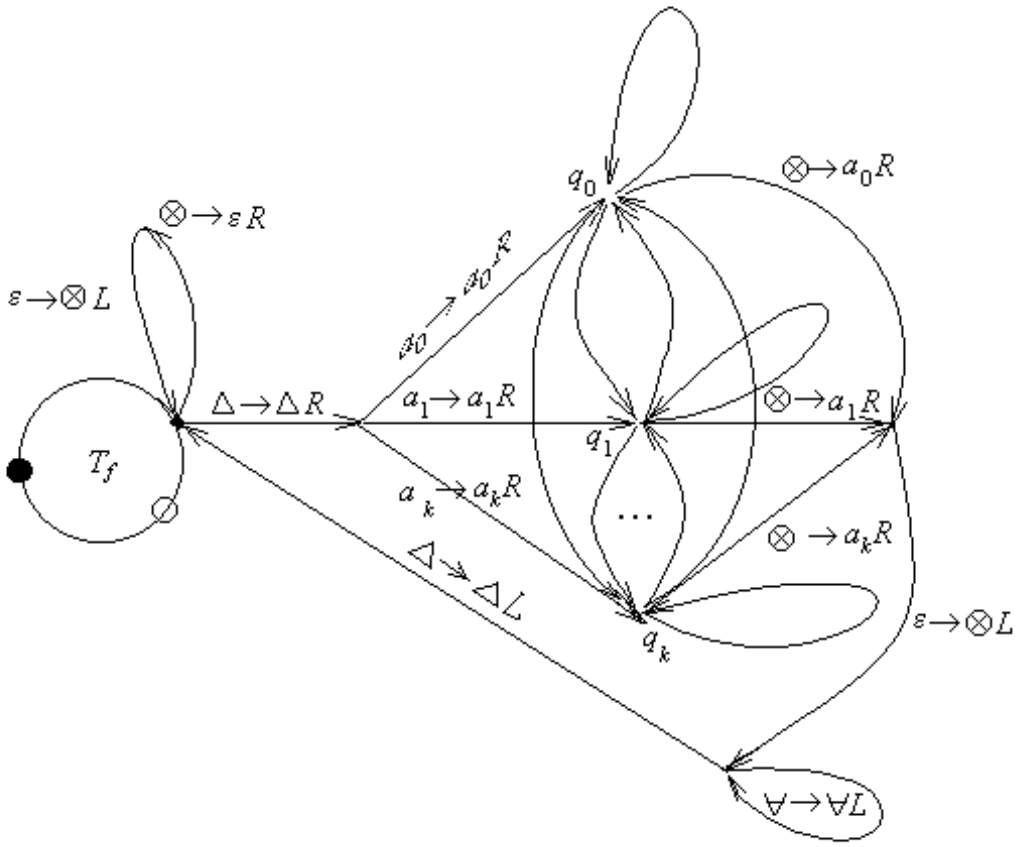


Рис. 3.2: Машина Тьюринга с полулентой.

Определение 3.16. Машина Тьюринга вычисляет функцию f с восстановлением, если:

$$p_0 1^{x_1} * \ldots * 1^{x_n} \stackrel{*}{\Rightarrow} p_z 1^{f(x_1,\ldots,x_n)} * 1^{x_1} * \ldots * 1^{x_n}.$$

Вычисление функции с восстановлением означает работу машины Тьюринга с сохранением исходных данных. Приведенное определение позволяет получать на ленте сначала результат, а затем исходные данные. Иногда бывает удобно сделать наоборот:

$$p_0 1^{x_1} * \ldots * 1^{x_n} \stackrel{*}{\Rightarrow} p_z 1^{x_1} * \ldots * 1^{x_n} * 1^{f(x_1,\ldots,x_n)}.$$

Аналогично теореме о правой полуленте можно доказать следующую теорему о левой полуленте.

Теорема 3.4. Функция, правильно вычислимая на машине Тьюринга с обычной лентой, правильно вычислима на машине Тьюринга с левой полулентой.

Теорема 3.5. Всякая правильно вычислимая функция правильно вычислима с восстановлением.

Доказательство. Пусть $f(\overline{x})$ — правильно вычислимая функция, тогда существует машина Тьюринга

$$T_f : p_0 1^{x_1} * \ldots * 1^{x_n} \stackrel{*}{\Rightarrow} p_z 1^{f(x_1,\ldots,x_n)}.$$

Тогда по теореме о левой полуленте существует машина Тьюринга T_f^{left}, вычисляющая функцию $f(\overline{x})$ на левой полуленте.

Построим вспомогательную машину Тьюринга, которая копирует исходные данные на ленте:

$$T_{copy} : p_0 A \stackrel{*}{\Rightarrow} p_0 A \triangle \ A, \text{ где A } \in \{1, *\}^*.$$

Очевидно, что T_{copy} легко построить.

Теперь рассмотрим композицию машин Тьюринга T_{copy} и T_f^{left}:

$$\begin{aligned} T_{copy} \cdot T^{left} : \ & p_0 1^{x_1} * \ldots * 1^{x_n} \stackrel{*}{\Rightarrow} p_z 1^{x_1} * \ldots * 1^{x_n} \triangle \ 1^{x_n} * \ldots * 1^{x_n} \\ & \Rightarrow p_z 1^{f(x_1,\ldots,x_n)} \triangle \ 1^{x_1} * \ldots * 1^{x_n} \end{aligned}$$

В заключение построим машину Тьюринга T_1, которая заменяет маркер на знак разделителя $*$. Композиция $T_{copy} \cdot T_f^{left} \cdot T_1$ выполняет требуемые действия в соответствии с определением 3.16. □

Теорема 3.6. Суперпозиция вычислимых функций — вычислимая функция.

Доказательство. Пусть дана суперпозиция вычислимых функций

$$f(\overline{x}) = h(g_1(\overline{x}), g_2(\overline{x}), \ldots g_m(\overline{x}))$$

и $\overline{x} = \langle x_1, x_2, ..., x_k \rangle$.

Тогда существуют машины Тьюринга T_h, T_1, T_2, ..., T_m, правильно вычисляющие функции $g_1(\overline{y})$, $g_2(\overline{x})$, ..., $g_m(\overline{x})$ соответственно. Тогда композиция машин Тьюринга

$$T = T_{copy} \cdot T_{mark} \cdot T_1 \cdot T_{shift} \cdot (\prod_{i=2}^{m-1} (T_{copy,right} \cdot T_{mark} \cdot T_{i,right} \cdot T_{shift})) \cdot T_{m,right} \cdot T_{shift} \cdot T_{back} \cdot T_h$$

вычисляет функцию $f(\overline{x})$. Здесь вспомогательные машины Тьюринга T_{copy}, T_{mark}, T_{shift}, T_{back} предназначены соответственно для копирования исходной цепочки $1^{x_1} * 1^{x_2} * ... * 1^{x_m}$, разделения ленты на левую и правую, так, что на правой полуленте остается скопированная цепочка $1^{x_1} * 1^{x_2} * ... * 1^{x_m}$, сдвига головки к началу цепочки $1^{x_1} * 1^{x_2} * ... * 1^{x_m}$, возврата головки к началу ленты с заменой всех маркеров на знаки разделителя $*$. Индекс $right$ означает, что машина Тьюринга работает на правой полуленте. □

3.6 Разветвление и повторение

Любая программа для ЭВМ, как правило, содержит операторы цикла и условные операторы. Поэтому в процессе формализации понятия алгоритма необходимо ввести в рассмотрение операторы разветвления и повторения.

Теорема 3.7. Разветвление к правильно вычислимым функциям по правильно вычислимому предикату является правильно вычислимым.

Доказательство. Пусть заданы предикат $P(x_1, \ldots, x_n)$, функции $g_1(x_1, \ldots, x_n)$ и $g_2(x_1, \ldots, x_n)$. Как известно, функция $f(x_1, \ldots, x_n)$ получена из

$$P(x_1, \ldots, x_n),\ g_1(x_1, \ldots, x_n) \text{ и } g_1(x_1, \ldots, x_n)$$

оператором разветвления, если

$$f(x_1, \ldots, x_n) = \begin{cases} g_1(x_1, \ldots, x_n), & \text{если } P(x_1, \ldots, x_n) \\ g_2(x_1, \ldots, x_n), & \text{если } \neg P(x_1, \ldots, x_n) \end{cases}$$

Можно рассмотреть два варианта доказательства, используя два различных подхода:

– функциональный,

– автоматный.

Функциональный подход базируется на операторе суперпозиции над вычислимыми функциями. Ранее мы доказали две теоремы: теорему 3.2 о вычислимости композиции — частного случая суперпозиции — вычислимых функций и теорему 3.6 о вычислимости суперпозиции вычислимых функций. Тогда можно представить требуемую функцию в виде суперпозиции известных вычислимых функций и тем самым показать ее вычислимость:

$$\begin{aligned} f(x_1, \ldots, x_n) \quad &= g_1(x_1, \ldots, x_n) \cdot \chi_P(x_1, \ldots, x_n) + \\ &+ g_2(x_1, \ldots, x_n) \cdot (1 - \chi_P(x_1, \ldots, x_n)). \end{aligned}$$

Как правило, функциональный метод доказательства является простым и наглядным при условии, что функцию можно получить с помощью вычислимых операторов над вычислимыми функциями. Иногда требуются дополнительные построения на уровне машин Тьюринга, поэтому для иллюстрации автоматного метода доказательства рассмотрим такое доказательство данной теоремы с тем, чтобы в дальнейшем при более сложных доказательствах его можно было использовать без особых затруднений.

Итак, рассмотрим автоматный подход к доказательству теоремы. Если характеристическая функция предиката $P(x_1, \ldots, x_n)$ и функции $g_1(x_1, \ldots, x_n)$, $g_2(x_1, \ldots, x_n)$ являются вычислимыми, то существуют соответственно три машины Тьюринга:

$$T_1 : 1^{x_1} * 1^{x_2} * \ldots 1^{x_n} \stackrel{*}{\Rightarrow} 1^{g_1(\overline{x})},$$

$$T_2 : 1^{x_1} * 1^{x_2} * \ldots 1^{x_n} \stackrel{*}{\Rightarrow} 1^{g_2(\overline{x})},$$

$$T_p : 1^{x_1} * 1^{x_2} * \ldots 1^{x_n} \Rightarrow 1^{\chi_p(\overline{x})}.$$

Если существует T_p, то в соответствии с ранее доказанной теоремой о вычислимости с восстановлением существует T_p^{reset}, вычисляющая с восстановлением характеристическую функцию предиката $P(x_1, \ldots, x_n)$. Построим машину Тьюринга $\tilde{T}$ с командами

$$\begin{aligned} &\delta = \delta_p^{reset} \cup \{p_{zp}1 \to p_1 \varepsilon R, p_1 * \to p_{01} \varepsilon R, p_{zp} * \to p_{02} \varepsilon R\} \cup \\ &\cup p_{z1} \cup \delta_1 \cup \delta_2. \end{aligned} \tag{3.4}$$

Если в качестве начального состоянием $\tilde{T}$ выбрать начальное состояние машины Тьюринга T_p^{reset}, а в качестве заключительного — объединенные в одно состояние p_z заключительные состояния машин Тьюринга T_1 и T_2, то $\tilde{T}$ выполняет следующие действия:

$$p_0 1^{x_1} * \ldots * 1^{x_n} \stackrel{*}{\Rightarrow} p_z 1^{\chi_p(x_1,\ldots,x_n)} * 1^{x_1} * \ldots * 1^{x_n} =$$
$$= \begin{cases} p_z * 1^{x_1} * \ldots * 1^{x_n}, & \text{если } P(\overline{x}) = \text{ Ложь} \\ p_z 1 * 1^{x_1} * \ldots * 1^{x_n}, & \text{если } P(\overline{x}) = \text{ Истина} \end{cases}$$

$$\stackrel{*}{\Rightarrow} \begin{cases} p_{01} 1^{x_1} * \ldots * 1^{x_n}, & \text{если } P(\overline{x}) = \text{ Ложь} \\ p_{02} 1^{x_1} * \ldots * 1^{x_n}, & \text{если } P(\overline{x}) = \text{ Истина} \end{cases}$$

$$\stackrel{*}{\Rightarrow} \begin{cases} p_z 1^{g_1(x_1,\ldots,x_n)}, & \text{если } P(\overline{x}) = \text{ Ложь} \\ p_z 1^{g_2(x_1,\ldots,x_n)}, & \text{если } P(\overline{x}) = \text{ Истина} \end{cases}$$

Следовательно, $\tilde{T}$ вычисляет функцию $f(x_1, \ldots, x_n)$. $\square$

Перейдем теперь к реализации циклов. Для простоты мы пока ограничимся функцией одного аргумента и рассмотрим $g(x)$, которая любому значению x ставит в соответствие результат выполнения следующей последовательности действий:

"пока $P(x)$ вычислять $x = f(x)$. "

Такая функция $g(x)$ называется повторением $f(x)$ по предикату $P(x)$.

Теорема 3.8. Повторение правильно вычислимой функции по правильно вычислимому предикату правильно вычислимо.

Доказательство. Если характеристическая функция предиката $P(x)$ и функция $f(x)$ являются правильно вычислимыми, то существуют машины Тьюринга, вычисляющие их:

$$T_p : 1^{x_1} * 1^{x_2} * \ldots 1^{x_n} \stackrel{*}{\Rightarrow} 1^{\chi_p(x)},$$
$$T_f : 1^{x_1} * 1^{x_2} * \ldots 1^{x_n} \stackrel{*}{\Rightarrow} 1^{f(x)},$$

и, следовательно, существует машина Тьюринга T_p^{reset}, вычисляющая с восстановлением характеристическую функцию предиката.

Пусть

$$T_p^{reset} = (K_1, \Sigma_1, \delta_p^{reset}, p_0, p_z, a_0, a_1),$$
$$T_f = (K_2, \Sigma_2, \delta_f, t_0, t_z, a_0, a_1).$$

Построим новую машину Тьюринга $\tilde{T}$ со следующими командами

$$\begin{array}{l} \delta = \delta_p^{reset} \cup \{p_z 1 \to q\varepsilon R,\ q* \to t_0\varepsilon R,\ p_z* \to p\varepsilon R\} \cup \\ \cup \delta_f \cup \{t_z 1 \to p_0 1 E, t_z\varepsilon \to p_0\varepsilon E.\} \end{array} \tag{3.5}$$

Состояние p_0 возьмем в качестве начального состояния этой машины Тьюринга, а состояние p — в качестве заключительного.

Функционирование построенной машины Тьюринга $\tilde{T}$ можно представить следующей последовательностью конфигураций:

$$p_0 1^x \Rightarrow p_z 1^{\chi_p(x)} * 1^x \Rightarrow \begin{cases} p_z 1 * 1^x, & \text{если } P(x) = \text{ истина} \\ p_z * 1^x, & \text{если } P(x) = \text{ ложь} \end{cases}$$
$$\Rightarrow \begin{cases} t_o 1^x \Rightarrow p_0 1^{f(x)} 1^x \Rightarrow p_z 1^{\chi_p(x)} * 1^{f(x)} \Rightarrow \ldots \\ p 1^x \end{cases}.$$

До тех пор, пока $P(x)$ ="истина построенная машина Тьюринга $\tilde{T}$ будет последовательно выполнять действия T_p^{reset} и T_f, т.е. результатом работы будет искомая функция g(x).$\square$

Следствие. Функция, заданная оператором примитивной рекурсии над вычислимыми функциями, вычислима.

Доказательство этого факта непосредственно следует из алгоритма вычисления функции, заданной оператором примитивной рекурсии, с помощью циклического процесса.

В общем случае можно рассматривать повторение вычислений для набора аргументов $x_1, x_2, \ldots, x_n$: *пока* $P(x)$ *вычислять*

$$x_1 = f_1(x_1, x_2, \ldots, x_n)$$

$$x_2 = f_2(x_1, x_2, \ldots, x_n)$$

$$\ldots$$

$$x_n = f_n(x_1, x_2, \ldots, x_n). \tag{3.6}$$

Можно доказать вычислимость повторения и в общем случае. Для такого доказательства достаточно вместо машины Тьюринга T_f использовать композицию машин Тьюринга, которая на текущем наборе аргументов выполняет последовательное вычисление функций $f_1(\overline{x})$, $f_2(\overline{x})$, …, $f_n(\overline{x})$ с одновременной заменой в цепочке

$$1^{f(x_1,\ldots,x_n)} * 1^{x_1} * 1^{x_2} * \ldots * 1^{x_n}$$

старого значения аргумента x_i новым вычисленным значением. Для выполнения такой подстановки можно воспользоваться вспомогательными машинами Тьюринга:

– циклического сдвига

$$T_c : p_0 1^{x_1} * 1^{x_2} * \ldots * 1^{x_i} \stackrel{*}{\Rightarrow} p_z 1^{x_2} * 1^{x_3} * \ldots * 1^{x_i} * 1^{x_1};$$

– удаления первого аргумента

$$T_{del} : p_0 1^{x_1} * 1^{x_2} * \ldots * 1^{x_i} \stackrel{*}{\Rightarrow} p_z 1^{x_2} * 1^{x_3} * \ldots * 1^{x_i}.$$

Поскольку эти действия требуется выполнять на полуленте (чтобы не оказывать воздействия на неиспользуемые элементы цепочки), необходимо использовать вспомогательные машины Тьюринга, вставляющие и удаляющие маркеры — разделители ленты на левую и правую полуленты. Полное построение машины Тьюринга, вычисляющей повторение (3.6), оставляется в качестве упражнения.

Анализ доказательства вычислимости разветвления и повторения позволяет ввести определение таких новых операций над машинами Тьюринга, которые позволяют строить новые машины Тьюринга из имеющихся.

Пусть даны три машины Тьюринга $\acute{T}$, T_1, T_2, причем у первой из них выделены два состояния p_i и p_j. Тогда можно дать определение *разветвления машины Тьюринга к двум машинам Тьюринга по двум состояниям*:

$$\acute{T} = T \begin{array}{ll} p_i \nearrow & T_1 \\ p_j \searrow & T_2. \end{array}$$

Машину T можно считать имеющей два выходных состояния p_i и p_j, к которым подсоединяются начальные состояния машин Тьюринга T_1 и T_2 соответственно. Как только $\acute{T}$ попадет в состояние p_i, она должна работать как T_1, аналогично для состояния p_j и машины Тьюринга T_2. Однако, исходная машина Тьюринга T имела

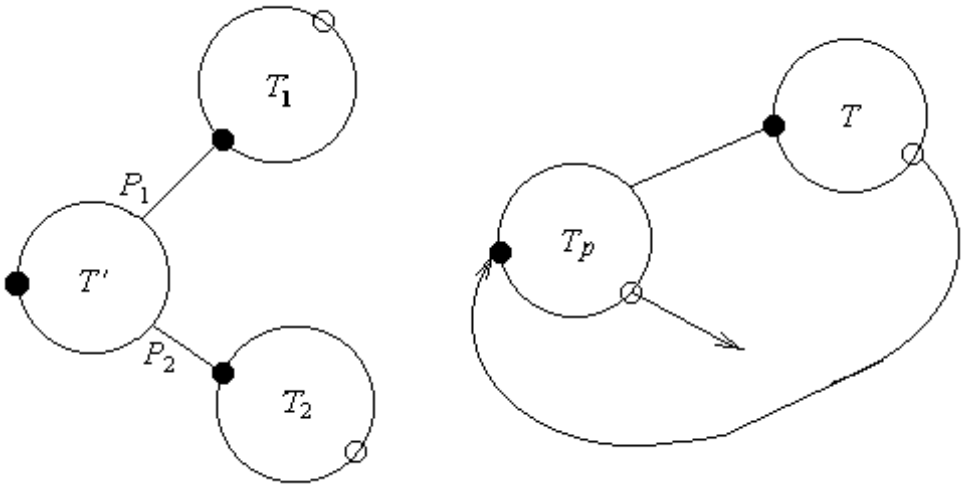


Рис. 3.3: Разветвление и повторение машин Тьюринга.

команды продолжения работы из состояний p_i и p_j. При определении множества команд $\acute{T}$ эти команды необходимо удалить из множества команд исходной машины Тьюринга T. В результате таких построений машина Тьюринга, представляющая собой разветвление T к T_1 и T_2 по состояниям p_i и p_j, имеет следующую систему команд:

$$\delta_p = \delta \setminus \{k|k \in \delta_t \wedge k = p_1 a \to qbr\} \setminus \{k|k \in \delta_t \wedge k = p_2 a \to qbr\},$$

где δ — команды, построенные по правилам 3.4.

Аналогично можно дать определение для операции повторения. *Повторением* некоторой машины Тьюринга T по состоянию q второй машины Тьюринга T_p называется машина Тьюринга с командами:

$$\delta_l = \delta \setminus \{k|k \in \delta_{T_p} \wedge k = qa \to pbr\},$$

где δ —команды, построенные по правилам 3.5.

Схематическое изображение алгоритма построения разветвления и повторения для произвольных машин Тьюринга представлено на рис. 3.3.

3.7 Тезис Тьюринга

Тезис Тьюринга дает определение алгоритма и формулируется следующим образом: *всякий алгоритм может быть реализован машиной Тьюринга.*

Смысл тезиса Тьюринга аналогичен смыслу тезиса Черча: машина Тьюринга вводится в виде определения алгоритма. Это не теорема, а именно тезис: в нем предлагается отождествить несколько расплывчатое интиутивное понятие алгоритма с понятием, сформулированным в точных математических терминах, и потому *доказать тезис Тьюринга невозможно.* Но в поддержку этого тезиса можно привести очень веские доводы.

По аналогии с тезисом Черча появление такого определения делает возможной постановку задачи о разрешимости и неразрешимости любой заданной проблемы. Доказать *разрешимость проблемы* — это значит доказать существование алгоритма. И наоборот, доказать *неразрешимость проблемы* — доказать отсутствие машины Тьюринга, решающей указанную проблему.

Понятие машины Тьюринга возникло в результате прямой попытки разложить интуитивно известные нам вычислительные процедуры на элементарные операции. Тьюринг описал некоторого рода теоретическую вычислительную машину. От существующих ЭВМ она отличается в двух отношениях, идеализируя ЭВМ и отвлекаясь от имеющихся практических ограничений. Во–первых, машина Тьюринга принципиально свободна от сбоев, т.е. она всегда без всяких отклонений выполняет правила, установленные для ее работы. Практически эта идеализированная характеристика машины Тьюринга не столь существенна, т.к. программы для реальных ЭВМ пишутся в расчете на то, что *в момент выполнения текущей операции* компьютер работает верно, хотя какие–то ошибки и могли иметь место на предшествующих шагах выполнения этой программы. Во–вторых, что более важно, машина Тьюринга снабжена потенциально бесконечной памятью. Это означает, что хотя в каждый момент количество накопленной ею информации конечно, для него нет никакой верхней грани. По этой причине иногда говорят, что машина Тьюринга — теоретически более мощный вычислитель, чем современные ЭВМ, обладающие памятью фиксированного объема. Это утверждение не верно, т.к. в каждый момент времени рассматривается не вся бесконечная память машины Тьюринга, а *потенциально бесконечная память,* которая может неограниченно расти в процессе выполнения программы, как и память на сменных внешних носителях, которую может использовать компьютер.

3.8 Контрольные вопросы к разделу

1. Чему равно число ячеек ленты машины Тьюринга?

2. Какие действия выполняет машина Тьюринга за один такт?

3. Может ли машина Тьюринга иметь бесконечное множество состояний?

4. Дайте определение конфигурации машины Тьюринга.

5. Чем отличается непосредственный переход конфигурации в конфигурацию от произвольного перехода конфигурации в конфигурацию?

6. Как по представлению машины Тьюринга в виде графа построить табличное ее представление?

7. Как по представлению машины Тьюринга в виде таблицы переходов построить ее представление в виде графа?

8. Чем отличается начальная конфигурация от стандартной начальной конфигурации?

9. Пусть машина Тьюринга T вычисляет функцию $f(x, y) = x/y$. Что будет делать T, имея на ленте цепочку $111*$? Поясните действия T при наличии на ленте исходной цепочки $*111$, $1111 * 11$, $1111 * 111$.

10. Чем отличается машина Тьюринга с полулентой от обычной машины Тьюринга?

11. Сколько дополнительных состояний появится у машины Тьюринга с полулентой, эквивалентной произвольной машине Тьюринга $T = (K, \Sigma, \delta, p_0, f, a_0, a_1)$ с обычной лентой?

12. Как построить машину Тьюринга, вычисляющую функцию $sg(x)$ с восстановлением? Как это сделать для произвольной функции $f(x)$?

13. Как построить машину Тьюринга, вычисляющую повторение функции одного аргумента?

14. Как построить машину Тьюринга, вычисляющую разветвление к вычислимым функциям по вычислимому предикату?

15. Пусть $f(x)$ — некоторая всюду определенная функция, $g(x)$ —- нигде не определенная функция. Как будет функционировать машина Тьюринга, вычисляющая функцию $f(g(x))$?

16. Приведите пример использования вычисления функции с восстановлением.

17. Зачем при работе машины Тьюринга с правой полулентой используется правый маркер?

18. Приведите пример вычислимой функции, которая определена только в одной точке и постройте соответствующую машину Тьюринга.

19. Приведите пример всюду определенной функции и постройте соответствующую машину Тьюринга.

20. Сформулируйте тезис Тьюринга. Поясните его смысл.

3.9 Упражнения к разделу

Задание. По словесному описанию машины Тьюринга построить ее программу. Представить построенную машину Тьюринга набором команд и графом переходов.

3.9.1 Задача

Машина Тьюринга получает на вход цепочку, состоящую из символов a, b, c. Если исходная цепочка имеет вид $a^n(bc)^n$, тогда заменить ее на цепочку из n единиц. В противном случае стереть исходную цепочку и оставить пустую ленту.

Решение. Запишем сначала действия машины Тьюринга в терминах конфигураций. Обозначим p_n и p_z соответственно начальное и заключительное состояния. Исходная цепочка задана над алфавитом $\Sigma = \{a, b, c\}$, тогда начальная конфигурация имеет вид $p_n\varphi$, где $\varphi \in \{a, b, c\}^*$. Заключительная конфигурация — либо p_z1^n, если $\varphi = a^n(bc)^n$, либо $p_z\varepsilon$ в противном случае. Тогда работа машины Тьюринга представима следующим образом:

$$p_n\varphi \stackrel{*}{\Rightarrow} \begin{cases} p_z1^n, & \text{если } \varphi = a^n(bc)^n, \\ p_z\varepsilon, & \text{если } \varphi \neq a^n(bc)^n \end{cases}$$

Для того, чтобы проверить соответствие подцепочек a^n и $(bc)^n$, будем последовательно заменять символы a на 1, а совокупность символов bc на 00. Эту подстановку выполним до тех пор, пока либо не закончится цепочка — и это означает, что она имеет требуемый вид, либо пока не обнаружим нарушение последовательности символов. В первом случае достаточно стереть все нули, во втором случае надо стереть всю цепочку. Изложим наши действия в виде последовательности пунктов словесного алгоритма.

1. Если читаем символ ε, пишем ε, оставляем головку на месте и переходим в заключительное состояние. Это означает, что исходная цепочка пустая, что соответствует значению $n = 0$. В противном случае переходим к п. 2.

2. Пока читаем символ a, выполняем действия, контролирующие соответствие символов a и bc:

а) читаем a, пишем 1, сдвигаем головку вправо;

б) пока читаем $a, 0$, пишем соответственно те же символы $a, 0$, сдвигаем головку вправо;

в) если читаем b, пишем 0, сдвигаем головку вправо; в противном случае перейти на 8;

г) если читаем c, пишем 0, сдвигаем головку влево; в противном случае перейти на 8;

д) пока не 1 читаем символы a или 0 с восстановлением и сдвигаем головку влево;

е) читаем 1 с восстановлением и сдвигаем головку вправо.

3. После выполнения пункта 2 прочитана начальная часть исходной цепочки, состоящая из последовательности символов a и выполнена замена a^n на 1^n, а следующих за a^n символов $(bc)^n$ на 0^{2n}. Головка обозревает первый символ за 1^n. Этим символом является первый символ цепочки 0^{2n}. Если окажется, что за 0^{2n} больше нет символов, то цепочка имеет требуемый вид. Поэтому, если читаем символ 0, то восстанавливаем его и оставляем головку на месте (фактически ничего не делаем и только переходим к следующему пункту алгоритма). Если читаем какой-либо другой символ, то переходим к пункту 8, который соответствует действиям по уничтожению содержимого ленты.

4. Пока читаем символы 0, восстанавливаем их и сдвигаем головку вправо. Тем самым головка дошла до конца начальной подцепочки вида $a^n(bc)^n$

5. Если читаем ε, пишем ε, сдвигаем головку влево. Это означает, что исходная цепочка имела требуемый вид. Теперь осталось только стереть все 0 и перейти в начало цепочки. При чтении какого–либо другого символа нужно перейти на 8.

6. Пока читаем $0, 1$ пишем соответственно $\varepsilon, 1$ и сдвигаем головку влево.

7. Читаем символ ε, пишем ε, сдвигаем головку вправо на начало результирующей цепочки, переходим в заключительное состояние. Процесс преобразования закончен.

8. В соответствии с указанными ранее действиями машина Тьюринга переходит в данное состояние только в том случае, когда нарушена структура цепочки и требуется затереть содержимое ленты. В общем случае непустые символы находятся как слева, так и справа от головки, следовательно, надо организовать проход по непустым символам как вправо, так и влево. Поэтому сначала установим головку, например, на начало цепочки, а потом организуем стирание символов при движении головки вправо. Итак, пока не ε, читаем символы $0, 1, a, b, c$ с восстановлением и передвигаем голвку влево.

9. Читаем символ ε, пишем ε, сдвигаем головку вправо.

10. Пока не ε, читаем символы $0, 1, a, b, c$, пишем ε и передвигаем голвку вправо.

11. Читаем символ ε, пишем ε, оставляем головку на месте и переходим в заключительное состояние.

Расммотрим теперь способ построения команд машины Тьюринга. Начальному пункту 1 поставим в соответсвие начальное состояние p_n. Обозначим p_z — заключительное состояние.

Цикл 2 должен обеспечиваться переходом из последнего пункта цикла (2.e) в начальное состояние p_0, в котором начинал выполняться этот цикл. Этот принцип возврата в то состояние, в котором начинался цикл, должен выполняться для каждого цикла алгоритма.

Для реализации перехода между последовательными пунктами 2 и 4 мы вставили "ничего не выполняющий " пункт 3 алгоритма. На уровне команд машины Тьюринга такая пустая последовательность действий реализуется переходом в другое состояние без изменения читаемого символа и сдвига головки:

$$p_0 \Rightarrow p_4 0 E.$$

Указанные действия можно представить в виде следующей последовательности команд (команды, соответствующие рассмотренным пунктам алгоритма, записаны

под соответствующим номером):

1. $p_n\varepsilon \to p_z\varepsilon E$, $p_n a \to p_0 a E$,
   $p_n b \to p_0 b E$, $p_n c \to p_0 c E$.
2. $p_0 a \to p_1 1 R$,
   $p_0 0 \to p_4 0 E$, $p_0 b \to p_6 b E$, $p_0 c \to p_6 c E$,
   $p_1 a \to p_1 a R$, $p_1 0 \to p_1 0 R$, $p_1 b \to p_2 0 R$,
   $p_1 c \to p_6 c E$, $p_1\varepsilon \to p_6\varepsilon L$,
   $p_2 c \to p_3 0 L$,
   $p_2 b \to p_6 b E$, $p_2 a \to p_6 a E$, $p_2\varepsilon \to p_6\varepsilon L$,
   $p_3 0 \to p_3 0 L$, $p_3 a \to p_3 a L$,
   $p_3 1 \to p_0 1 R$.
3. $p_0 0 \Rightarrow p_4 0 E$.
4. $p_4 0 \Rightarrow p_4 0 R$.
5. $p_4\varepsilon \Rightarrow p_5\varepsilon L$, $p_4 a \Rightarrow p_6 a E$,
   $p_4 b \Rightarrow p_6 b E$, $p_4 c \Rightarrow p_6 c E$.
6. $p_5 0 \Rightarrow p_5\varepsilon L$, $p_5 1 \Rightarrow p_5 1 L$.
7. $p_5\varepsilon \Rightarrow p_z\varepsilon R$.
8. $p_6 a \Rightarrow p_6 a L$, $p_6 b \Rightarrow p_6 b L$, $p_6 c \Rightarrow p_6 c L$,
   $p_6 0 \Rightarrow p_6 0 L$, $p_6 1 \Rightarrow p_6 1 L$.
9. $p_6\varepsilon \Rightarrow p_7\varepsilon R$.
10. $p_7 a \Rightarrow p_7\varepsilon R$, $p_7 b \Rightarrow p_7\varepsilon R$, $p_7 c \Rightarrow p_7\varepsilon R$,
    $p_7 0 \Rightarrow p_7\varepsilon R$, $p_7 1 \Rightarrow p_7\varepsilon R$.
11. $p_7\varepsilon \Rightarrow p_z\varepsilon E$.

Граф переходов построенной машины Тьюринга представлен на рис. 3.4.

3.9.2 Варианты заданий

1. Машина Тьюринга получает на вход цепочку, состоящую из двух двоичных чисел, разделенных знаком +. Вычислить результат сложения этих двоичных чисел. Результат представить двоичным числом.

2. Машина Тьюринга получает на вход цепочку, состоящую из символов 0, 1, +. Если исходная цепочка не содержит символов +, тогда оставить ее без изменения. В противном случае стереть в исходной цепочке все символы + и "стянуть" цепочку так, чтобы все оставшиеся символы 0 и 1 были записаны на ленте без пробелов.

3. Машина Тьюринга получает на вход цепочку, состоящую из n единиц. Заметить эту цепочку двоичным представлением числа n.

4. Машина Тьюринга получает на вход цепочку, состоящую из символов a, b, c. Если исходная цепочка имеет вид $(abc)^n$, тогда заменить ее на цепочку из n единиц. В противном случае стереть исходную цепочку и оставить пустую ленту.

5. Машина Тьюринга получает на вход цепочку, состоящую из символов $*$, 0, 1 и представляющую собой два двоичных числа, разделенных знаком $*$. Оставить на ленте минимальное из этих чисел.

6. Машина Тьюринга получает на вход цепочку, состоящую из символов a, b, c. Если исходная цепочка имеет вид $(ab)^n(c)^n$, тогда заменить ее на цепочку из $3n$ единиц. В противном случае стереть исходную цепочку и оставить пустую ленту.

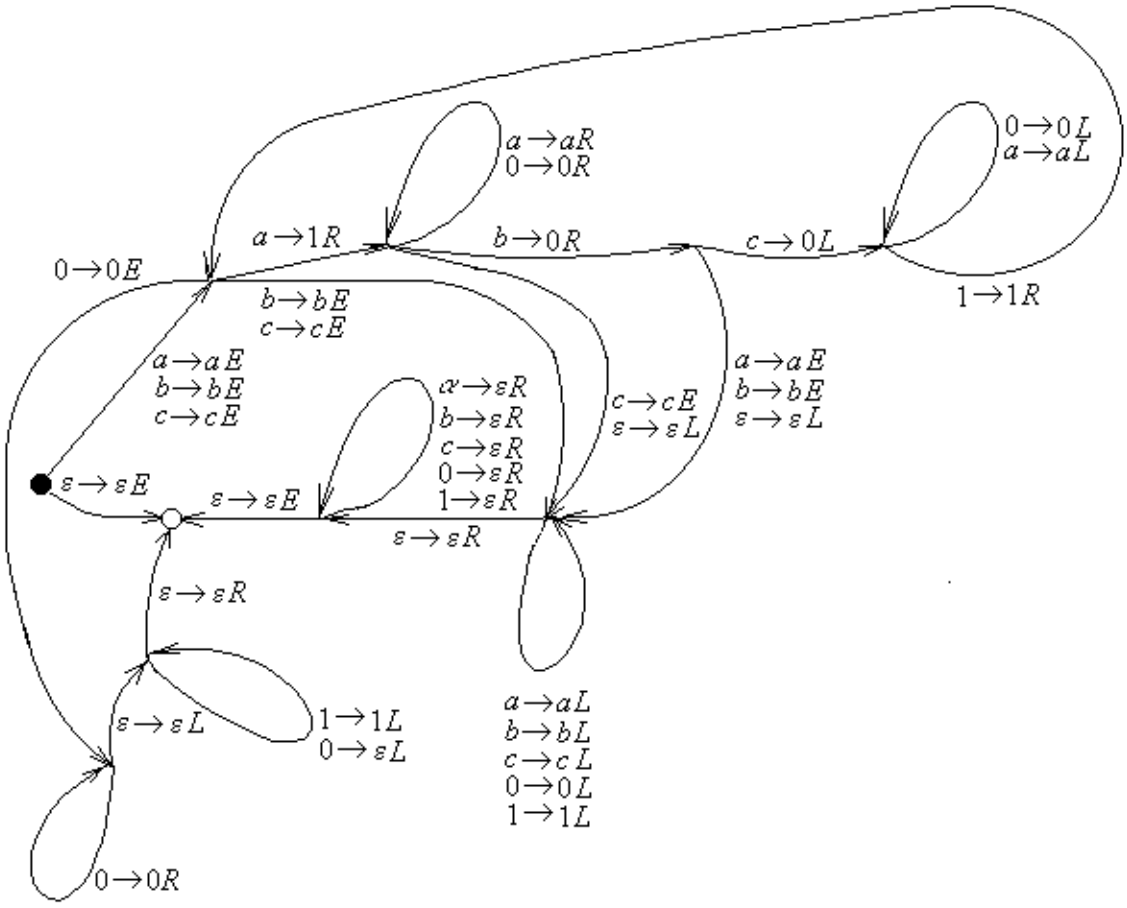


Рис. 3.4: Машина Тьюринга преобразования цепочек над алфавитом $\{a, b, c\}$.

7. Машина Тьюринга получает на вход цепочку, состоящую из символов $*$,0, 1 и представляющую два двоичных числа, разделенных знаком $*$. Если оба числа четные, то оставить на ленте среднее арифметическое из этих чисел. Результат представить двоичным числом. В противном случае очистить ленту.

8. Машина Тьюринга получает на вход цепочку, состоящую из символов a, b, c. Если исходная цепочка имеет вид $x\tilde{x}$, где $\tilde{x}$ обозначает зеркальное отображение цепочки x (например, если $x = abcc$, то $\tilde{x} = ccba$), то оставить на ленте x. В противном случае заменить каждый символ цепочки на единицу.

9. Машина Тьюринга получает на вход цепочку, состоящую из символов 0, 1 и представляющую некоторое двоичное число n. Преобразовать исходную цепочку в цепочку 1^n.

10. Машина Тьюринга получает на вход цепочку, состоящую из символов + 0, 1. Если исходная цепочка имеет вид $x + \tilde{x}$, где $\tilde{x}$ обозначает зеркальное отображение цепочки x (например, если $x = 011$, то $\tilde{x} = 110$), то стереть ее. В противном случае каждый символ цепочки заменить нулем.

11. Машина Тьюринга получает на вход цепочку, состоящую из двух чисел в системе счисления с основанием 3, разделенных знаком +. Вычислить результат сложения этих троичных чисел. Результат представить троичным числом.

12. Машина Тьюринга получает на вход цепочку, состоящую из символов 0, 1, +. Если исходная цепочка имеет вид $1^n + (0)^n$, тогда заменить ее на цепочку из n единиц. В противном случае стереть исходную цепочку и оставить пустую ленту.

13. Машина Тьюринга получает на вход цепочку, состоящую из двух двоичных чисел, разделенных знаком $-$. Вычислить модуль разности двоичных чисел. Результат представить двоичным числом.

14. Машина Тьюринга получает на вход цепочку, состоящую из символов a, b, c. Если исходная цепочка читается одинаково справа налева и слева направо, то записать на ленту 1. В противном случае записать на ленту 0.

15. Машина Тьюринга получает на вход цепочку, состоящую из символов 0, 1 и представляющую некоторое двоичное число n. Преобразовать исходную цепочку в цепочку 1^{2n+1}.

16. Машина Тьюринга получает на вход цепочку, состоящую из двух двоичных чисел, разделенных знаком $*$. Если одно из них в два раза больше второго, записать на ленту 1. В противном случае записать на ленту 0.

17. Машина Тьюринга получает на вход цепочку, состоящую из символов a, b, c. Если исходная цепочка не содержит символв b, тогда оставить ее без изменения. В противном случае стереть в исходной цепочке все символы b и "стянуть" цепочку так, чтобы все оставшиеся символы a и c были записаны на ленте без пробелов.

18. Машина Тьюринга получает на вход цепочку, состоящую из символов a и b. Если исходная цепочка x состоит из чередующихся символов a и b, то заменить ее зеркальным отображением $\tilde{x}$. Например, если $x = ababab$, то $\tilde{x} = bababa$. В противном случае оставить на ленте столько подряд идуцих символов c, сколько символов b содержит цепочка. Например, если $x = aaabbab$,то результатом является ccc.

19. Машина Тьюринга получает на вход цепочку, состоящую из символов 0, 1. Если исходная цепочка имеет вид $x\tilde{x}$, где $\tilde{x}$ обозначает зеркальное отображение цепочки x (например, если $x = 011$, то $\tilde{x} = 110$), то стереть ее. В противном случае оставить на ленте x.

20. Машина Тьюринга получает на вход цепочку, состоящую из символов $*$,0, 1 и представляющую два двоичных числа, разделенных знаком $*$. Оставить на ленте максимальное из этих чисел.

3.10 Тесты для самоконтроля к разделу

1. Дана функция $x\dot{-}1$. Необходимо построить машину Тьюринга, вычисляющую эту функцию. Обозначим g_0 – начальное состояние, q_z – заключительное состояние. Машину Тьюринга представить в виде последовательности каманд.

Варианты ответов:

1) $q_0\varepsilon \longrightarrow g_0\varepsilon E, \quad q_0 1 \longrightarrow g_0\varepsilon R;$
2) $q_0\varepsilon \longrightarrow g_z\varepsilon E, \quad q_0 1 \longrightarrow g_z\varepsilon R;$
3) $q_0\varepsilon \longrightarrow g_0\varepsilon E$
3) $q_0 1 \longrightarrow g_z\varepsilon R, \quad q_0\varepsilon \longrightarrow g_0\varepsilon R;$
4) $q_0 1 \longrightarrow g_z\varepsilon R;$
5) $q_0 1 \longrightarrow g_0\varepsilon R.$

Правильный ответ: 2.

2. Какое из следующих утверждений является правильным:

1) для любой машины Тьюринга с N состояниями можно построить эквивалентную, работающую на правой полуленте и имеющую не более $2N$ состояний;

2) композиция двух машин Тьюринга получается в результате объединения начальных состояний этих машин;

3) суперпозиция вычислимых функций не всегда является вычислимой функцией;

4) для любой машины Тьюринга можно построить эквивалентную, работающую на левой полуленте;

5) для любой машины Тьюринга можно построить эквивалентную, работающую на ленте ограниченного размера.

Правильный ответ: 4.

3. Какие из следующих утверждений истинны?

1) Тезис Тьюринга доказывает существование алгоритмов.

2) Тезис Тьюринга представляет собой формальное определение алгоритма.

3) Тезис Тьюринга пока еще не доказан, но по мере развития теории алгоритмов ожидается получение его доказательства.

4) Тезис Тьюринга доказать нельзя, т.к. он представляет собой определение.

5) Тезисом Тьюринга называется теорема о существовании алгоритмов.

Варианты ответов:

а) 1 и 5;
б) 2 и 5;
в) 1 и 4;
г) 2, 3 и 4;
д) 2 и 4.

Правильный ответ: д.

4. Дана машина Тьюринга с командами:

$$\begin{array}{ll} T: & q_0\varepsilon \longrightarrow g_0\varepsilon E, \\ & q_0 1 \longrightarrow g_0\varepsilon R, \\ & q_0 * \longrightarrow g_0 1 E. \end{array}$$

Состояние q_0 является начальным и состояние q_z — заключительным. Какую функцию одного аргумента вычисляет эта машина Тьюринга?

Варианты ответов:

а) $f(x) = 0$;

б) $f(x) = 1$;

в) $f(\overline{x})$ — нигде не определенная функция;

г) $f(x) = x$;

д) $f(x, y) = y$;

е) $f(x)$ определена в единственной точке $x = 1$ и $f(1) = 1$.

Правильный ответ: в.

5. Даны две машины Тьюринга:

$$\begin{aligned} T_1: \quad & q_0\varepsilon \longrightarrow g_0\varepsilon E, \\ & q_0 1 \longrightarrow g_z\varepsilon R; \end{aligned}$$

$$\begin{aligned} T_2: \quad & q_0 1 \longrightarrow g_0 1 E, \\ & q_0\varepsilon \longrightarrow g_z 1 E. \end{aligned}$$

Какую функцию одного аргумента вычисляет композиция машин Тьюринга $T_1 \cdot T_2$?

Варианты ответов:

а) $f(x) = 0$;

б) $f(x) = 1$;

в) $f(x)$ — нигде не определенная функция;

г) $f(x) = x$;

д) $f(x)$ определена в единственной точке $x = 1$ и $f(1) = 1$.

Правильный ответ: д.

Глава 4

ОБЩАЯ ТЕОРИЯ АЛГОРИТМОВ

4.1 Геделевский номер машины Тьюринга

Для того, чтобы понятие рекурсивности и рекурсивной перечислимости перенести из области натуральных чисел в область более сложных объектов — n–ок чисел, в главе 1 мы занумеровали все n–ки натуральными числами. В данной главе метод нумерации будет рассмотрен в более общем виде. Он позволяет глубже вскрыть природу алгоритмических процессов и прямым путем приводит к решению ряда интересных проблем. Необходимость нумерации произвольных объектов вызвана, прежде всего, необходимостью анализа различных задач, которые должны обрабатывать алгоритмы в качестве исходной информации. Рассматривая определение алгоритма в виде частично–рекурсивной функции или машины Тьюринга, мы приходим к пониманию того, что исходная информация для этих алгоритмических моделей представима в одном и том же виде — в виде натуральных чисел. Следовательно, для того, чтобы рассматривать алгоритмы над алгоритмами, необходимо представлять алгоритм (в данной главе — машину Тьюринга) в виде натуральных чисел.

Рассмотрим Геделевскую нумерацию объектов. Считается, что введена *система Геделевской нумерации* для всех объектов A, принадлежащих некоторому множеству M, если выполняются следующие два требования:

1) существует натуральное число $ng(A)$, которое однозначно определяется по A;

2) для всех n, принадлежащих множеству натуральных чисел N, выполняется одно из двух условий:

– либо не существует объекта A, принадлежащего множеству M, такого, что $n = ng(A)$;

– либо существует единственный объект A, принадлежащий M, такой, что $n = ng(A)$ и этот объект однозначно восстанавливается по n.

Рассмотрим Геделевскую нумерацию машин Тьюринга. Известно, что любая машина Тьюринга $T = (K, \Sigma, \delta, p_0, p_z, a_0, a_1)$ задается множеством команд вида

$$p_i a_j \to p_k a_l r, \text{ где}$$

$p_i, p_k \in K$ — состояния, $a_j, a_l \in \Sigma$ — символы алфавита ленты, $r \in \{R, L, E\}$ — направление движения головки. Занумеруем все состояния и символы алфавита натуральными числами: $K = \{p_0, p_1, \ldots, p_z\}$ и $\Sigma = \{a_0, a_1, \ldots, a_s\}$. Будем считать, что состояние p_0 с нулевым номером — начальное и состояние p_z с максимальным номером z — заключительное.

Рассмотрим сначала одну команду $p_i a_j \to p_k a_l r$. Индексы i, j, k, l — это натуральные числа, которые можно записать в какой–либо системе счисления. Выберем

двоичную систему счисления, тогда в записи каждой команды используются только следующие символы: p, a, 0, 1, R, L, E.

Поставим в соответствие каждому символу десятичную цифру:

$$p \to 1, a \to 2, 0 \to 3, 1 \to 4, R \to 5, L \to 6, E \to 7.$$

(Вообще говоря, эти цифры можно рассматривать и в восьмеричной системе счисления, но в силу привычки будем считать их десятичными.) Тогда каждой команде ставится в соответствие целое число в десятичной системе счисления. Например, команде с десятичными индексами

$$p_3 a_0 \to p_5 a_2 R$$

или в двоичном эквиваленте

$$p_{11} a_0 p_{101} a_{10} R$$

соответствует число 1442314342435.

Поскольку любая машина Тьюринга задается набором команд, то каждой команде ставится в соответствие одно число, а всему набору команд — одно длинное число, полученное последовательной записью соответствующих каждой команде чисел. Для однозначного определения такого длинного числа будем формировать его из отдельных чисел в возрастающей последовательности. Очевидно, что оба требования определения Геделевской нумерации выполняются при предложенном способе кодирования машин Тьюринга в виде натурального числа.

Определив нумерацию Геделя для машин Тьюринга, мы можем ставить вопрос о том, какие свойства алгоритмов можно распознать по номерам машин Тьюринга. Например, естественно спросить, существует ли алгоритм, позволяющий для произвольного номера n машины Тьюринга узнать, остановится ли соответствующий алгоритм, обрабатывая заданные исходные данные, или он зациклится. Частным случаем этой проблемы является анализ функции, которую вычисляет заданная машина Тьюринга: необходимо определить, примитивно–рекурсивна эта функция или нет. Не менее интересным вопросом является анализ заданного своим номером алгоритма с целью определить функцию, которую вычисляет этот алгоритм. Все указанные вопросы и многие другие аналогичные появляются в процессе написания, отладки и анализа программ для ЭВМ. Например, для имеющейся программы крайне важно знать, зациклится эта программа на данных или нет, какую функцию выполняет эта программа. При тестировании возникает вопрос о том, правильно ли программа перерабатывает данные и получает ли требуемый результат.

По тезису Тьюринга каждый алгоритм может быть реализован машиной Тьюринга. Возможность нумерации машин Тьюринга означает, что любой вычислимой функции можно поставить в соответствие ее номер. Возникает вопрос: все ли функции вычислимы? Другими словами, необходимо выяснить существование функций, не вычислимых никакими алгоритмами. Известно, что любой кортеж $\langle a_1, a_2, \ldots, a_n \rangle$ можно представить одним натуральным числом, используя соответствующую нумерацию. Поэтому можем рассмотреть только функции одного аргумента. Допустим, что все одноместные функции на множестве натуральных чисел вычислимы. Тогда каждой вычислимой функции можно поставить в соответствие натуральное число — геделевский номер машины Тьюринга, вычисляющей эту функцию. Пусть M — множество всех вычислимых функций, $M = \{f_0(x), f_1(x), f_2(x), \ldots\}$. Построим одноместную функцию $h(x)$, отличную от всех функций множества M. Тем самым мы докажем существование функций, не являющихся вычислимыми.

Чтобы сделать построение функции $h(x)$ более наглядным, составим бесконечную матрицу, строками которой будут служить последовательности значений функций $f_0(x), f_1(x), f_2(x), \ldots$, а столбцами — натуральные числа $0, 1, 2, \ldots$, на которых вычисляются значения этих функций:

	0	1	2	...
$f_0(x)$	$f_0(0)$	$f_0(1)$	$f_0(2)$	...
$f_1(x)$	$f_1(0)$	$f_1(1)$	$f_1(2)$	...
$f_2(x)$	$f_2(0)$	$f_2(1)$	$f_2(2)$	...
...	...	...	...	...

Определим теперь функцию $h(x)$ как функцию, последовательность значений которой получается из последовательности значений, стоящих в нашей таблице на диагонали, увеличением каждого из них, скажем, на единицу, т.е. $h(a) = f_a(a) + 1$. Эта функция существует в силу нашего конструктивного построения, но функция $h(x)$ не принадлежит множеству M, т.к. она отличается от $f_0(x)$ своим значением для аргумента 0, от $f_1(x)$ своим значением для аргумента 1 и т.д. Другими словами, если $h(x) \in M$, то существует такой натуральный номер m, что для всех x справедливо $h(x) = f_m(x)$. Тогда подставляя вместо переменной x число m получим $h(m) = f_m(m) = f_m(m) + 1$, что невозможно. Полученное противоречие доказывает существование функций, не принадлежащих множеству вычислимых функций. Тем самым *появляется новый вопрос — вопрос о существовании или несуществовании алгоритмов, решающих некоторую поставленну проблему.* Классический пример неразрешимой проблемы — проблема остановки машины Тьюринга.

4.2 Проблема остановки машины Тьюринга

Проблема остановки алгоритма заключается в определении для произвольного алгоритма и произвольных исходных данных, поступающих на вход этому алгоритму, принципа обработки указанным алгоритмом предложенных исходных данных:

– остановится алгоритм через некоторое конечное число шагов с полученным в процессе работы результатом;

– не остановится никогда.

Теорема 4.1. Проблема остановки неразрешима.

Доказательство. Допустим, что проблема разрешима. Это значит, что существует алгоритм решения данной проблемы, т.е. существует машина Тьюринга, решающая эту проблему. Эта машина Тьюринга должна получать на вход алгоритм и исходные данные для него и выдавать на выход "да" или "нет" в зависимости от того, остановится или зациклится алгоритм на этих данных. Такая машина Тьюринга должна действовать следующим образом:

$$T_0 : p_0 x * ng(A) \stackrel{*}{\Rightarrow} \begin{cases} p_z 1, & \text{если } A \text{ остановится на } x; \\ p_z \varepsilon, & \text{если } A \text{ не остановится на } x. \end{cases}$$

Легко построить машину Тьюринга, которая копирует исходные данные:

$$T_{copy} : q_0 u \stackrel{*}{\Rightarrow} q_z u * u,$$

где u — цепочка из символов "1".

Поскольку x и $ng(A)$ — натуральные числа, то цепочки u и $ng(A)$ в унарном коде состоят из одних и тех же символов "1". Тогда на вход машины Тьюринга копирования подадим Геделевский номер $ng(A)$ и рассмотрим композицию машин Тьюринга

$$T_{copy} \cdot T_0 : q_0 ng(A) \stackrel{*}{\Rightarrow} q_z ng(A) * ng(A) \stackrel{*}{\Rightarrow}$$
$$\stackrel{*}{\Rightarrow} \begin{cases} p_z 1, & \text{если } A \text{ остановится на } ng(A); \\ p_z \varepsilon, & \text{если } A \text{ не остановится на } ng(A). \end{cases}$$

Теперь построим другую машину Тьюринга:

$$T_1 : t_0 1^n \stackrel{*}{\Rightarrow} \begin{cases} t_z \varepsilon, & \text{если } n = 0; \\ \infty, & \text{если } n > 0. \end{cases}$$

Тогда

$$T_{copy} \cdot T_0 \cdot T_1 : q_0 ng(A) \stackrel{*}{\Rightarrow} q_z ng(A) * ng(A) \stackrel{*}{\Rightarrow}$$
$$\stackrel{*}{\Rightarrow} \begin{cases} \infty, & \text{если } A \text{ остановится на } ng(A); \\ t_z \varepsilon, & \text{если } A \text{ не остановится на } ng(A). \end{cases}$$

Обозначим $T_{res} = T_{copy} \cdot T_0 \cdot T_1$. Подадим на вход машины T_{res} ее собственный Геделевский номер $ng(T_{res})$ и получим противоречие:

$$T_{res} : q_0 ng(T_{res}) \stackrel{*}{\Rightarrow}$$
$$\stackrel{*}{\Rightarrow} \begin{cases} \infty, & \text{если } T_{res} \text{ остановится на } ng(T_{res}); \\ t_z \varepsilon, & \text{если } T_{res} \text{ не остановится на } ng(T_{res}). \end{cases}$$

Машина Тьюринга T_{res} существовать не может, но $T_{res} = T_{copy} \cdot T_0 \cdot T_1$, причем T_{copy} и T_1 существуют. Следовательно, не существует T_0, и наше предположение о разрешимости проблемы остановки было неверным. □

В силу тезиса Тьюринга невозможность построения машины Тьюринга означает отсутствие алгоритма решения данной проблемы. Поэтому полученная теорема дает первый пример алгоритмически неразрешимой проблемы.

Следствие. Проблема результативности программы неразрешима. Иными словами, не существует общего алгоритма, который для любой программы определил бы, остановится программа при обработке каких–нибудь данных или нет.

Алгоритмическая неразрешимость означает отсутствие единого алгоритма, решающего данную проблему. При этом вовсе не исключается возможность решения проблемы в каждом частном случае. Например, неразрешимость проблемы остановки не исключает того, что для отдельных классов машин Тьюринга она может быть решена. Неразрешимость проблемы остановки можно интерпретировать как несуществование общего алгоритма для отладки программ, точнее, алгоритма, который по тексту любой программы и данным определял бы, зациклится программа на этих данных или нет. Если учесть сделанное ранее замечание, такая интерпретация не противоречит тому эмпирическому факту, что большинство программ в конце концов все же удается отладить, т.е. установить наличие зацикливания, найти его причину и устранить ее. При этом решающую роль играют опыт и интуиция программиста.

При доказательстве теоремы мы рассматривали машину Тьюринга $T_{copy} \cdot T_0$, которая получает на вход Геделевский номер некоторой машины и определяет, зациклится машина или нет. Алгоритм, который получает на вход свое собственное описание

(Геделевский номер) и перерабатывает его, не зацикливаясь, называется самоприменимым. Самоприменимыми должны быть трансляторы и интерпретаторы, программы печати текстов программ, и тому подобные программы, исходной информацией для которых также являются программы. Из доказательства теоремы об остановке следует, что машина $T_{copy} \cdot T_0$ существовать не может, следовательно проблема самоприменимости также является неразрешимой.

Еще один пример неразрешимой проблемы дает теорема Райса.

Теорема 4.2 (Райс). Никакое нетривиальное свойство вычислимых функций не является алгоритмически разрешимым.

Доказательство. Для доказательства эту теорему удобнее сформулировать в несколько иной форме. Пусть C — любой класс вычислимых функций одной переменной, нетривиальный в том смысле, что имеются как функции, принадлежащие C, так и функции, не принадлежащие C. Тогда не существует алгоритма, который бы по номеру n функции f_n определял бы, принадлежит f_n классу C или нет.

Допустим, множество $M = \{\, x \mid f_x \in C \,\}$ разрешимо, тогда вычислима его характеристическая функция

$$\chi(x) = \begin{cases} 1, & \text{если } f_x \in C, \\ 0, & \text{если } f_x \notin C. \end{cases}$$

По предположению класс C — нетривиальный клвсс функций, поэтому существуют нигде не определенные функции, как принадлежащие C, так и не принадлежащие C. Возьмем нигде не определенную функцию f_0, не принадлежащую С, и одну вычислимую функцию с некоторым фиксированным номером f_m из C. Определим новую функцию

$$F(x,y) = \begin{cases} f_m(y), & \text{если } f_x(x) \text{ определена}, \\ f_0(y), & \text{если } f_x(x) \text{ не определена}. \end{cases}$$

В зависимости от значения x функция $F(x,y)$ равна либо вычислимой функции $f_m(y)$, либо нигде не определенной функции $f_0(y)$, следовательно, функцию $F(x,y)$ можно представить как функцию от y с номером $g(x)$, которая имеет значения из двухэлементного множества {0,m}:

$$f_{g(x)}(y) = \begin{cases} f_m(y), & \text{если } f_x(x) \text{ определена, , } f_m(y) \in C \\ f_0(y), & \text{если } f_x(x) \text{ не определена,} f_0(y) \notin C \end{cases}$$

Тогда номер $g(x)$ функции $f_{g(x)}(y)$ равен

$$g(x) = \begin{cases} m, & \text{если } f_x(x) \text{ определена}, \\ 0, & \text{если } f_x(x) \text{ не определена}. \end{cases}$$

Вычислим новую функцию $r(x) = sg(g(x))$:

$$r(x) = \begin{cases} 1, & \text{если } f_x(x) \text{ определена}, \\ 0, & \text{если } f_x(x) \text{ не определена}. \end{cases}$$

Вычисление $r(x)$ эквивалентно решению проблемы самоприменимости, следовательно наше предположение о разрешимости множества M неверно. □

Следствием из этой теоремы является такой хорошо известный каждому опытному программисту факт, что по тексту программы, не снабженной комментариями о ее назначении и выполняемых действиях, очень сложно и иногда практически невозможно выяснить, что эта программа делает. Опытный "хакер" может "взломать"

программу со сложной системой защиты, высококлассный программист может разобраться в чужой большой и сложной программе. Но применяемые при этом методы не поддаются универсальной алгоритмизации. Нельзя написать универсальную программу, которая будет выполнять указанные действия и перерабатывать текст любой заданной программы.

4.3 Метод сводимости и доказательство неразрешимости

Метод сводимости одной проблемы к другой заключается в следующем. Пусть имеется некоторая проблема A, которая является задачей о решении множества (обычно бесконечного) некоторых единичных задач: $A = \{a_i\}$. Пусть имеется некоторая другая проблема: $B = \{b_j\}$.

Определение 4.1. Если для каждой частной задачи a_i проблемы A можно найти такую задачу b_j проблемы B , что из решения задачи b_j можно получить решение задачи a_i, тогда проблема A сводится к проблеме B.

Если для всех j частная задача b_j разрешима, то для всех i задача a_i разрешима. Тогда можно записать два правила разрешимости:

B разрешима $\longrightarrow$ разрешима A ;

A неразрешима $\longrightarrow$ неразрешима B.

Допустим, имеется некоторая проблема C. Необходимо рассмотреть эту проблему с двух сторон: а) C разрешима, б) C неразрешима.

Пусть необходимо доказать разрешимость C. Тогда надо найти такую разрешимую проблему B, чтобы C сводилось к B. В этом случае из алгоритма B получим алгоритм C.

Пусть необходимо доказать неразрешимость C. Тогда надо найти такую неразрешимую проблему A, чтобы она сводилась к C.

В качестве примера применения метода сводимости рассмотрим проблему переводимости, связанную с анализом переработки исходной цепочки в результирующую. Применительно к практическому программированию проблема переводимости выглядит так. При отладке программы решается следующая задача: даны исходные данные x и результат y. Необходимо проверить правильность работы программы на этом тесте. В терминах машины Тьюринга проблема переводимости заключается в проверке истинности выражения

$$x \stackrel{*}{\Rightarrow}_A y.$$

Решение этой проблемы означает существование машины Тьюринга:

$$T_n : P_0\hat{x} * ng(A) * \hat{y} \stackrel{*}{\Rightarrow} \begin{cases} P_z1, & A \text{ работает правильно,} \\ P_z\varepsilon, & A \text{ работает неправильно.} \end{cases}$$

Здесь $\hat{x}$ и $\hat{y}$ — представление на ленте машины Тьюринга цепочек x и y соответственно. Заметим, что результаты работы машины Тьюринга $T = (K, \Sigma, \delta, p_0, f, a_0, a_1)$ всегда представляют собой цепочку над алфавитом Σ.

Допустим, что проблема переводимости разрешима и машина Тьюринга T_n, решающая ее, существует. Тогда разрешима следующая проблема: $\hat{x} * ng(A) * \hat{Y}$, где $\hat{Y}$ — произвольная цепочка над алфавитом Σ. Это лзначает, что проблема остановки сводится к проблеме переводимости, т.к. получение в качестве результата любой цепочки $\hat{Y}$ означает просто переход в заключительное состояние, т.е. остановку алгоритма. В соответствии с методом сводимости и в силу неразрешимости проблемы остановки неразрешимой является и проблема переводимости.

4.4 Алгоритмы Маркова

Рассмотрим теперь другую точку зрения на алгоритм: каждый алгоритм преобразует исходные данные в результаты, причем исходные данные и результаты каким–то способом записываются в качестве текстовых строк. Тогда всякий алгоритм преобразует запись данных в запись результатов. Поэтому можно сказать, что всякий алгоритм преобразует цепочку над некоторым алфавитом в другие цепочки. Рассмотрим процесс преобразования слов над заданным алфавитом.

Пусть $\sum$ — алфавит, а P, Q — цепочки над этим алфавитом. Рассмотрим правила подстановки вида

$$P \to Q$$

$$P \to Q_\bullet$$

Первое из этих правил называется обычным правилом подстановки, а второе — заключительным. Оба эти правила применяются ко входным цепочкам одинаково, разница заключается лишь в том, что после заключительного правила работа алгоритма заканчивается, а после обычного может быть продолжена. Применение правила $P \to Q[\bullet]$ к цепочке x заключается в том, что самое левое вхождение P в x заменяется на Q:

$$x = yPz \to yQz.$$

Определение 4.2. Алгоритмом Маркова над алфавитом Σ называется упорядоченная конечная последовательность правил подстановки:

$$\begin{aligned} P_1 &\to Q_1[\bullet] \\ &\dots \\ P_k &\to Q_k[\bullet] \end{aligned}$$

Алгоритм Маркова работает следующим образом: на каждом шаге применяется правило подстановки с минимально возможным номером. Алгоритм заканчивает работу либо после принятия заключительного правила, либо при отсутствии применимых правил.

Пример. Построить алгоритм Маркова, вычисляющий функцию

$$f(x,y) = \begin{cases} x+y+1, & x \le 2, \\ 0, & x > 2. \end{cases}$$

Очевидно, что для вычисления функции необходимо выполнить анализ условия $x > 2$. Если аргументы, как и ранее, будем записывать в унарном коде, то наличие хотя бы трех единиц перед знаком "*" означает справедливость $x > 2$. Тогда функцию $f(x)$ вычисляет следующий алгоритм Маркова:

$$\begin{aligned} U_f: \quad & 111* && \to \sharp \\ & * && \to 1_\bullet \\ & \sharp 1 && \to \sharp \\ & 1\sharp && \to \sharp \\ & \sharp && \to \bullet \end{aligned}$$

Следует обратить внимание на порядок правил в алгоритме U_f. Если поменять местами первое и второе правило, то алгоритм будет вычислять функцию $x+y+1$ для любых аргументов.

Иногда алгоритм Маркова должен выполнять различные действия на участках цепочки, разных по местоположению, но одинаковых по структуре. Для различения таких участков удобно применять бегущий маркер, который ставится в начале цепочки с помощью правила $\varepsilon \to \sharp$. Это правило может быть только самым последним в последовательности правил, т.к. в противном случае оно запретит возможность принятия всех стоящих за ним правил по той причине, что ε есть в любой цепочке на любом месте. Например, алгоритм Маркова

$$U : \varepsilon \to 1$$

никогда не остановится на любой цепочке, вставляя до бесконечности перед ней символы 1.

Принцип Маркова, как и рассмотренные ранее тезисы Тьюринга и Черча, дает определение алгоритма: *всякий алгоритм может быть реализован алгоритмом Маркова.*

4.5 Эквивалентность алгоритмических моделей

Ввиду неточности интуитивных понятий алгоритма и эффективно вычислимой функции невозможно *доказать* верность тезиса Черча или Тьюринга, истинность принципа Маркова. Не располагаем мы и какими–либо априорными доводами в пользу этих гипотез. Здесь можно рассчитывать лишь на неполное подтверждение правильности подходов при введении математического определения алгоритма в форме машины Тьюринга, частично–рекурсивной функции или алгоритма Маркова, а не на строгое доказательство. Однако, есть некоторое дополнительное обстоятельство, говорящее в пользу этих тезисов, а именно тот замечательный факт, что все три определения алгоритма оказываются эквивалентными друг другу.

Попробуем доказать эквивалентность рассмотренных трех алгоритмических моделей. С учетом того факта, что машины Тьюринга предназначены для выполнения алгоритмической процедуры, будем использовать машины Тьюринга в качестве основной модели и докажем эквивалентность всех трех определений алгоритма на основе доказательства четырех теорем:

а) для любой машины Тьюринга можно построить эквивалентную частично–рекурсивную функцию;

б) для любой частично–рекурсивной функции можно построить эквивалентную машину Тьюринга;

в) для любой машины Тьюринга можно построить эквивалентный алгоритм Маркова;

г) для любого алгоритма Маркова можно построить эквивалентную машины Тьюринга.

Следствием из этих теорем, которые мы рассмотрим в следующих параграфах, является эквивалентность алгоритмов Маркова и частично–рекурсивных функций.

Можно отметить, что рассмотренными нами алгоритмическими моделями не исчерпываются попытки дать математически строгое определение алгоритма. К тому же самому результату ведут и другие подходы — теория λ—вычислимости Черча, теория нормальных систем Поста, вычислимость по Эрбрану и т.п.

4.6 Теорема об эквивалентности Машин Тьюринга и частично–рекурсивных функций

Мы рассмотрели три алгоритмические модели: машины Тьюринга, частично–рекурсивные функции, алгоритмы Маркова. Если покажем эквивалентность этих формальных определений алгоритма, то все свойства, доказанные для какой–либо алгоритмической модели, можем распространить на общее понятие алгоритма. При этом особое значение имеют такие принципиально важные положения как выводы о неразрешимости некоторых проблем. Покажем эквивалентность частично–рекурсивных функций и функций, вычислимых по Тьюрингу.

Теорема 4.3. Всякая частично–рекурсивная функция вычислима по Тьюрингу.

Доказательство. В соответствии с рекурсивным определением частично–рекурсивной функции для доказательства теоремы достаточно доказать следующие четыре утверждения:

– простейшие функции вычислимы по Тьюрингу;

– с помощью оператора суперпозиции из вычислимых по Тьюрингу функций получаем вычислимые функции;

– с помощью оператора минимизации из вычислимых по Тьюрингу функций получаем вычислимые функции;

– с помощью оператора примитивной рекурсии из вычислимых по Тьюрингу функций получаем вычислимые функции.

Вычислимость простейших функций очевидна, так как легко построить соответствующие машины Тьюринга. Теорема о вычислимости суперпозиции быда доказана выше (см. теорему 2.6). Рассмотрим оператор минимизации

$$f(\overline{x}) = \mu_z(P(\overline{x}, t)).$$

Вычисление этой функции можно реализовать с помощью алгоритма

1. $t := 0$;
2. пока $X_p(\overline{x}, t) = 0$ вычисляется $t := t + 1$;
3. $f(\overline{x}) = t$.

Последовательное выполнение пунктов алгоритма эквивалентно выполнению всего алгоритма при условии существования этих отдельных алгоритмов. Очевидно существование машин Тьюринга T_1 и T_3, где T_1 вычисляет значение $t = 0$ и T_3 возвращает значение t в качестве значения функции $f(\overline{x})$. Второй пункт этого алгоритма представляет собой повторение вычислимых функций по вычислимому предикату. Следовательно, машина Тьюринга T_2, выполняющая такое вычисление, также существует. Тогда существует и композиция $T_1 \cdot T_2 \cdot T_3$.

Вычислимость оператора примитивной рекурсии доказывается аналогично. □

Теорема 4.4. Любая вычислимая по Тьюрингу функция является частично–рекурсивной функцией.

Доказательство. Пусть дана произвольная машина Тьюринга

$$T = (K, \Sigma, \delta, p_0, p_z, a_0, a_1).$$

В основу доказательства теоремы положим принцип нумерации поведения машины Тьюринга, заменяя числами как все данные на ленте, так и состояния управляющего устройства, а затем определяя числовые функции, которые характеризуют процессы изменения таких чисел.

Будем считать, что каждый символ $a \in \Sigma$ на ленте — это цифра в некоторой системе счисления. Выполняя действия, машина Тьюринга заменяет одни такие цифры на другие и, следовательно, преобразует одно число в другое. Кроме изменения содержимого ленты, машина Тьюринга переходит из одного состояния в другое. Номер этого состояния также можно считать числом. Для простоты возьмем в качестве системы счисления число

$$S = max\{|\Sigma|, |K|\},$$

где Σ, K — соответственно входной алфавит и множество состояний машины Тьюринга. Известно, что на любом такте конфигурация $\langle \alpha, q, a, \beta \rangle$ машины Тьюринга T однозначно определяет ее состояние, положение головки и содержимое ленты. Заменим символьное определение конфигурации набором натуральных чисел. Левую часть α цепочки на ленте будем читать как число слева направо, правую часть β цепочки — справа налево. Бесконечное число нулей — пустых символов a_0 — в этом случае являются незначащими лидирующими нулями. Тогда в любой момент конфигурация $\langle \alpha, q, a, \beta \rangle$ машины Тьюринга T однозначно определяется четырьмя числами $\langle \alpha, q, a, \beta \rangle$:

$...a_0$	a_0	α	a	β	a_0	$a_0...$
			$\Uparrow$			
			q			

Построим функции, которые в соответствии с командами машины Тьюринга определяют записываемый символ, новое состояние и направление движения, причем знакам направления движения поставим в соответствие, например, следующие числа:

$$L \to 2,\ R \to 1,\ E \to 0.$$

Машина Тьюринга на каждом такте выполняет одну какую–либо команду вида $qa \to pbr$. Тогда построим функции $v_q(q,a)$, $v_r(q,a)$, $v_a(q,a)$, которые по текущему состоянию и обозреваемому символу определяют новое состояние, записываемый символ и направление сдвига головки. Каждой команде $qa \to pbr$ можно поставить в соответствие следующее определение

$$v_q(q,a) = p, \quad v_r(q,a) = r = \begin{cases} 0,\ r = E \\ 1,\ r = R \\ 2,\ r = L, \end{cases} \quad v_a(q,a) = b.$$

Множество команд машины Тьюринга T конечно по определению, следовательно, данные функции являются примитивно–рекурсивными как функции, полученные с помощью разветвления к примитивно–рекурсивным функциям.

Определим функции на множестве $N \times N \times N \times N$ — множестве четверок чисел (α, q, a, β), которые являются числовым эквивалентом конфигураций машины Тьюринга. Эти функции определят элементы новой четверки после применения команды на одном такте работы машины Тьюринга :

$$f_\alpha(\alpha, q, a, \beta) = \begin{cases} \alpha,\ \text{если}\ \ v_r(q,a) = 0 \\ \alpha \cdot S + v_a(q,a),\ \text{если}\ \ v_r(q,a) = 1 \\ [\alpha/S]\,,\ \text{если}\ \ v_r(q,a) = 2, \end{cases}$$

$$f_q(\alpha, q, a, \beta) = v_q(q,a),$$

$$f_a(\alpha, q, a, \beta) = \begin{cases} v_a(q, a), \text{ если } v_r(q, a) = 0 \\ \{\beta/S\}, \text{ если } v_r(q, a) = 1 \\ \{\alpha/S\}, \text{ если } v_r(q, a) = 2, \end{cases}$$

$$f_\beta(\alpha, q, a, \beta) = \begin{cases} \beta, \text{ если } v_r(q, a) = 0 \\ [\beta/S], \text{ если } v_r(q, a) = 1 \\ \beta \cdot S + v_a(q, a), \text{ если } v_r(q, a) = 2. \end{cases}$$

Все эти функции, определяющие каждое из новых четырех характеризующих конфигурацию чисел на последующем шаге, — примитивно–рекурсивные. Рассмотрим теперь поведение машины Тьюринга T по тактам и введем функции, являющиеся (кроме уже известных элементов четверки) еще и функцией номера такта t:

$$F_\alpha(t, \alpha, q, a, \beta),$$

$$F_a(t, \alpha, q, a, \beta),$$

$$F_q(t, \alpha, q, a, \beta),$$

$$F_\beta(t, \alpha, q, a, \beta).$$

Все эти функции можно определить рекурсивно.

$$F_\alpha(0, \alpha, q, a, \beta) = \alpha,$$
$$F_\alpha(t+1, \alpha, q, a, \beta) =$$
$$= f_\alpha(F_\alpha(t, \alpha, q, a, \beta)), F_q(t, \alpha, q, a, \beta), F_a(t, \alpha, q, a, \beta), F_\beta(t, \alpha, q, a, \beta))$$
$$F_q(0, \alpha, q, a, \beta) = q,$$
$$F_q(t+1, \alpha, q, a, \beta) =$$
$$= f_q(F_\alpha(t, \alpha, q, a, \beta)), F_q(t, \alpha, q, a, \beta), F_a(t, \alpha, q, a, \beta), F_\beta(t, \alpha, q, a, \beta)),$$
$$F_a(0, \alpha, q, a, \beta) = a,$$
$$F_a(t+1, \alpha, q, a, \beta) =$$
$$= f_a(F_\alpha(t, \alpha, q, a, \beta)), F_q(t, \alpha, q, a, \beta), F_a(t, \alpha, q, a, \beta), F_\beta(t, \alpha, q, a, \beta)),$$
$$F_\beta(0, \alpha, q, a, \beta) = \beta,$$
$$F_\beta(t+1, \alpha, q, a, \beta) =$$
$$= f_\beta(F_\alpha(t, \alpha, q, a, \beta)), F_q(t, \alpha, q, a, \beta), F_a(t, \alpha, q, a, \beta), F_\beta(t, \alpha, q, a, \beta)).$$

Приведенные рекурсивные схемы соответствуют так называемой параллельной рекурсии. Можно доказать, что схема параллельной рекурсии сводится к примитивной рекурсии (доказательство оставляется в качестве упражнения). Следовательно, каждая из этих функций определяется схемой примитивной рекурсии с использованием примитивно–рекурсивных функций и, значит, все они являются примитивно–рекурсивными функциями.

Пусть в начальный момент на ленте записана цепочка x. В конечный момент результатом является некоторая цепочка y, которая представляет собой функцию от исходных данных: $y = f(x)$.

Рассмотрим эту функцию. В начальный момент при условии правильной работы машины Тьюринга имеем

$\alpha = 0$;
$q = 0$;
$\beta = [z(x)/S]$, где $z(x)$ — зеркальное отображение цепочки x;
$a = \{z(x)/S.\}$

Функция зеркального отображения числа является примитивно–рекурсивной в силу следующего определения:

$$\begin{aligned}
&z(0) = 0,\\
&z(1) = 1,\\
&\dots\\
&z(S-1) = S-1,\\
&z(x) = \{x/S\} \cdot S[\log_S x] + z[x/S].
\end{aligned}$$

По определению правильной вычислимости перед началом работы машина Тьюринга находится в начальном сотоянии ($q = 0$), на ленте имеется исходная цепочка x, головка обозревает первый символ цепочки x ($\alpha = 0$, $a = \{z(x)/S\}$, $\beta = [z(x)/S]$). Тогда значение $y = f(x)$ определяется при условии правильной вычислимости по формуле:

$$y = z(\beta \cdot S + a) = z(S \cdot F_\beta(t_{end}, 0, 0, \{z(x)/S\}, [z(x)/S]) + F_a(t_{end}, 0, 0, \{z(x)/S\}, [z(x)/S])$$

Машина Тьюринга переходит в заключительное состояние и завершает работу на таком шаге t_{end}, когда ее текущее состояние окажется заключительным:

$$t_{end} = \mu_t(q_{end} = F_q(t, 0, 0, \{z(x)/S\}, z[x/S])).$$

Тогда $y = f(x)$ — суперпозиция частично–рекурсивных и примитивно–рекурсивных функций, следовательно, сама является частично–рекурсивной функцией .□

4.7 Теорема об эквивалентности машин Тьюринга и алгоритмов Маркова

Теорема 4.5. Для произвольной машины Тьюринга можно построить эквивалентный алгоритм Маркова.

Доказательство. Пусть $T = (K, \Sigma, \delta, p_0, f, a_0, a_1)$ — произвольная машина Тьюринга. Построим алгоритм Маркова M, выполняющий те же действия, что и T. В алфавит алгоритма Маркова M включим все символы из множества $K \cup \Sigma$. Построим правила M. Каждой команде $p_i a_j \to p_k a_m r \in \delta$ машины Тьюринга T сопоставим правило или множество правил алгоритма Маркова следующим образом:

а) если $r = E$, то правило $p_i a_j \to p_k a_m$;

б) если $r = L$, то для всех $b \in \Sigma$ включим в множество правил соответствующее правило $bp_i a_j \to p_k b a_m$, отмечая тем самым движение головки влево;

в) если $r = R$, то для всех $b \in \Sigma$ включим в множество правил соответствующее правило $p_i a_j b \to a_m p_k b$, отмечая тем самым движение головки вправо.

Для того, чтобы начать выполнение алгоритма Маркова в соответствии с правилом, указывающим момент начала работы машины Тьюринга из начального состояния, в котором головка обозревает первый символ исходной уепочки, добавим к множеству правил алгоритма Маркова правило $\varepsilon \to p_0$ и сделаем это правило последним в списке правил M.

Чтобы отметить момент завершения работы машины Тьюринга, добавим к множеству правил M заключительное правило с точкой $f \to \varepsilon_\bullet$.

Осталось реализовать условие "зависания" машины Тьюринга в том случае, когда ее состояние p и обозреваемый символ a таковы, что в множестве ее команд нет

команды с левой частью pa. Это условие также легко реализуется, если множество правил алгоритма Маркова дополнить правилами $pa \to pa$ для всех $p \in K$ и $a \in \Sigma$, для которых в δ отсутствует команда с левой частью pa.

Эквивалентность исходной машины Тьюринга T и построенного алгоритма Маркова M очевидна. □

Теорема 4.6. Для произвольного алгоритма Маркова можно построить эквивалентную машину Тьюринга.

Доказательство. Пусть $M = \{U\}$ — алгоритм Маркова над заданным алфавитом Σ. Рассмотрим следующие вспомогательные машины Тьюринга.

$T_{\alpha i}$ — поиск цепочки α_i в текущей цепочке, которая имеется на ленте. Если цепочка найдена, машина Тьюринга $T_{\alpha i}$ переходит в состояние p_i и обозревает последний символ найденной цепочки α_i, в противном случае она переходит в состояние f_i. Эти два состояния нам в дальныйшем понадобятся для реализации разветвления к различным машинам Тьюринга.

$T^l_{\beta i}$ — заменить на левой полуленте цепочку α_i на цепочку β_i. Перед началом работы машина Тьюринга обозревает последний символ цепочки α_i. Заменив цепочку α_i на β_i, машина Тьюринга переходит в заключительное состояние и обозревает последний символ цепочки β_i.

T_{begin} — перейти к началу цепочки, записанной на ленте.

T_{mar} — разделить ленту на полуленты. Неподвижный маркер ставится справа от обозреваемой ячейки.

T_{del} — удалить маркер, разделяющий ленту на полуленты и слить содержимое полулент.

T_{end} — закончить работу, для чего необходимо вернуть головку к началу цепочки и прейти в заключительное состояние.

Построение указанных машин Тьюринга достаточно просто и не должно вызвать затруднений. Построим теперь из вспомогательных машин Тьюринга несколько новых:

а) если правило $\alpha \to \beta$ алгоритма Маркова была не заключительным, то

$$T_{i,1} = T_{mar} \cdot T_{\beta i} \cdot T_{del} \cdot T_{begin} \ -\!-$$

в соответствии с тем, как были определены вспомогательные машины Тьюринга, эта машина Тьюринга заменит цепочку α на β и вернется к началу данных на ленте; если правило было заключительныйм, то

$$T_{i,1} = T_{mar} \cdot T_{\beta} \cdot T_{del} \cdot T_{begin} \cdot T_{end} \ -\!-$$

в отличие от предшествующего варианта данная машина Тьюринга выполнит те же действия и закончит работу;

а) рассмотрим машину Тьюринга $T_{i,2}$:

$$T_{i,2} = T_{\alpha i} \begin{array}{ll} & p_i \nearrow \ T_{i,1} \\ & \\ & f_i \searrow \ T_{(i+1),2}. \end{array}$$

Осталось теперь определить машину Тьюринга $T_{n+1,2}$, где $n = |U|$ — число правил алгоритма Маркова $M = \{U\}$. Машина $T_{(n+1,2)}$ начнет работу только при условии, что ни одно из правил множества U не найдено на ленте. Это означает, что алгоритм Маркова должен закончить свою работу. Эквивалентное певедение построенной машины Тьюринга будет при условии, что $T_{(n+1,2)} = T_{end}$.

Машины Тьюринга $T_{i,1}$ и $T_{i,2}$ существуют, т.к. они построены с помощью композиции и разветвления из существующих машин Тьюринга. Очевидно, что построенная нами машина Тьюринга $T_{1,2}$ выполняет те же преобразования исходной цепочки, что и алгоритм Маркова M. $\square$

Следствие. Определения алгоритма в виде машины Тьюринга, алгоритма Маркова и частично–рекурсивной функции эквивалентны.

Эквивалентность рассмотренных алгоритмических моделей означает, что всякий алгоритм, описанный средствами одной модели, может быть описан средствами другой. Благодаря взаимной сводимости можно выработать инвариантную по отношению к модели систему понятий, позволяющую рассматривать свойства алгоритмов независимо от того, какая формализация алгоритма выбрана. Эта система понятий основана на вычислимой функции, т.е. функции, для вычисления которой существует алгоритм. Все свойства, рассмотренные в какой–либо одной модели справедливы для понятия алгоритма независимо от расматриваемой модели.

Сравнивая обычное определение частично–рекурсивных функций с определением тех же функций как функций, вычислимых на машинах Тьюринга, программисту легко заметить следующие свойства этих определений. Обычное определение частично–рекурсивных функций настолько широко и приближено к программе вычисления этих функций на каком–либо алгоритмическом языке, что из него почти непосредственна видна частичная рекурсивность функций, вычислимых посредством программ для ЭВМ. Напротив, определение с помощью машин Тьюринга — очень специальное. Его цель — показать, как самые сложные процессы можно моделировать на простых устройствах.

4.8 Контрольные вопросы к разделу

1. Приведите пример Геделевской нумерации программ, написанных на языке Си.

2. Что называется Геделевской нумерацией?

3. Постройте Геделевский номер машины Тьюринга с двумя командами

$$p_0 1 \to p_0 1 L, \quad p_0 \varepsilon \to p_1 1 E.$$

Какую функцию одного аргумента вычисляет эта машина Тьюринга?

4. Зачем вводится понятие Геделевской нумерации?

5. Сформулируйте проблему остановки машины Тьюринга.

6. Разрешима ли проблема остановки машины Тьюринга?

7. Сформулируйте проблему переводимости.

8. Какой практический смысл связян с проблемой остановки машины Тьюринга?

9. Как для произвольной машины Тьюринга построить эквивалентный алгоритм Маркова?

10. Какую функцию одного аргумента вычисляет алгоритм Маркова с правилами

$$\begin{aligned} *111 &\longrightarrow 1* \\ * &\longrightarrow * \\ \varepsilon &\longrightarrow *1 \end{aligned}$$

11. Как изменится вычисляемая функция одного аргумента, если в предшествующем задании переставить местами первое и последнее правило?

12. Как для произвольной машины Тьюринга построить эквивалентную частично–рекурсивную функцию?

13. Как для произволной частично–рекурсивной функции построить эквивалентный алгоритм Маркова?

14. Перечислите известные Вам неразрешимые проблемы, имеющие непосредственное отношение к практике программирования.

15. Сформулируйте теорему Райса.

16. Как используется метод сводимости для доказательства неразрешимости?

17. Как для произвольного алгоритма Маркова построить эквивалентную машину Тьюринга?

18. Какие способы доказательства неразрешимости проблем Вы знаете?

19. Дайте формальное определение алгоритма.

20. Какие эквивалентные определения алгоритма Вы знаете?

4.9 Упражнения к разделу

4.9.1 Задача

Построить машину Тьюринга и алгоритм Маркова, вычисляющие функцию

$$f(x,y) = \frac{2x+3y}{4}.$$

Решение. Очевидно, что на множестве натуральных чисел функция определена не всюду, поэтому как для машины Тьюринга, так и для алгоритма Маркова необходимо предусмотреть зацикливание в случае, когда $2x+3y$ не кратно 4.

Рассмотрим сначала алгоритм Маркова. Очевидно, что умножение на некоторую константу c в унарном коде реализуется с помощью замены каждой единицы на c единиц. Для того, чтобы организовать умножение сначала на 2, а потом на 3, нужно использовать разный маркер. Можно сначала перед цепочкой поставить, например, маркер $\triangle$, выполнить с этим маркером умножение на 2, а затем, когда закончатся знаки "1" первого аргумента, заменить его на другой маркер, например, $\oplus$. Для реализации деления полученной суммы на 4 потребуется еще один маркер. Используем для этой цели знак Ω. Поставим перед исходной цепочкой два маркера $\triangle$ и Ω, используя правило

$$\varepsilon \to \Omega\triangle$$

Эта правило должно стоять последним в последовательности правил алгоритма. Умножение последовательно на 2 и 3 реализуется правилами

$$\begin{array}{l}\triangle 1 \to 11\triangle \\ \triangle * \to \oplus \\ \oplus 1 \to 111 \oplus .\end{array}$$

Завершаем умножение, удаляя уже не нужный маркер:

$$\oplus \to \varepsilon.$$

Теперь осталось с помощью маркера Ω реализовать деление:

$$\Omega 1111 \to 1\Omega.$$

Если имеется остаток, организуем зацикливание:

$$\Omega 1 \to \Omega 1.$$

Эта команда может стоять только после предыдущей команды, т.к. по правилам выполнения алгоритма Маркова выполняется первая применимая к цепочке команда. Если остатка нет, завершаем вычисление функции:

$$\Omega \to \varepsilon_{\bullet}.$$

Таким образом, получили следующий алгоритм Маркова:

$$\begin{array}{l}
\triangle 1 \to 11\triangle \\
\triangle * \to \oplus \\
\oplus 1 \to 111\oplus \\
\oplus \to \varepsilon \\
\Omega 1111 \to 1\Omega \\
\Omega 1 \to \Omega 1 \\
\Omega \to \varepsilon_{\bullet} \\
\varepsilon \to \Omega \triangle .
\end{array}$$

Рассмотрим теперь машину Тьюринга, которая вычисляет эту же функцию. Если ввести обозначения для вспомогательных функций

$$f_1(x) = 2x,\ f_2(x) = 3x,\ f_3(x) = \frac{x}{4},\ f_4(x,y) = x + y,$$

то

$$f(x,y) = f_3(f_4(f_1(x), f_2(y))).$$

Теперь достаточно построить машины Тьюринга T_1, T_2, T_3, T_4, вычисляющие соответственно функции $f_1(x), f_2(x), f_3(x), f_4(x,y)$, а затем с помощью композиции

$$T_1^{left} \cdot T_{second} \cdot T_2^{right} \cdot T_{back} \cdot T_3 \cdot T_4$$

вычислить искомую функцию $f(x,y)$. Здесь T_1^{left} и T_2^{right} — T_1 и T_2, работающие соответственно на левой и правой полуленте и воспринимажщие символ $*$ в качестве разделителя ленты на полуленты. Вспомогательные машины Тьюринга T_{second} и T_{back} перемещают головку соответственно к началу второго аргумента и к началу всей цепочки.

Построить указанные машины Тьюринга просто. Рассмотрим в качестве примера построение T_4. На вход в соответствии с определением машины Тьюринга, которая вычисляет функцию двух аргументов, поступает цепочка $1^x * 1^y$. Функция всегда определена, поэтому для любых исходных данных после завершения работы на ленте должна остаться цепочка 1^{x+y}. Для этого достаточно заменить символ $*$ на 1, а затем вернуть головку к началу цепочки и стереть первую единицу из получившейся последовательности 1^{x+y+1}. Указанные действия выполняет машина Тьюринга с командами:

$$\begin{array}{ll}
p_0 1 \to p_0 1R, & p_0 * \to p_1 1L, \\
p_1 1 \to p_1 1L, & p_1 \varepsilon \to p_2 \varepsilon R, \\
p_2 1 \to p_z \varepsilon R. &
\end{array}$$

Эта машина Тьюринга представлена в виде графа на рис. 4.1.

Функция

$$f_3(x) = \frac{x}{4}$$

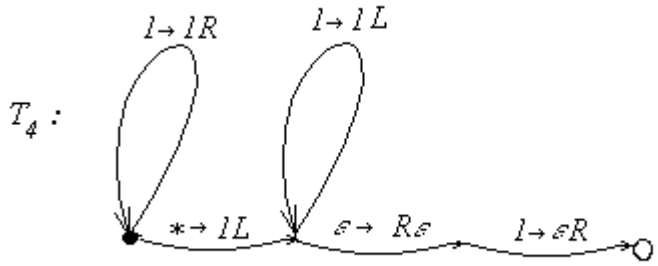


Рис. 4.1: Машина Тьюринга T_4, вычисляющая функцию $f_4(x, y) = x + y$.

определена только при условии, что значение переменной x кратно 4. Деление можно реализовать с помощью переноса одной единицы за специально поставленный маркер $*$ после чтения последовательности из четырех единиц. При чтении эти единицы стираются. Если единицы закончились, то частное вычислено и остается только стереть маркер. Если при чтении единиц на ленте остается одна, две или три единицы, надо организовать зацикливание. Указанные действия выполняет машина Тьюринга с командами:

$p_0\varepsilon \to p_z\varepsilon E$, $p_0 1 \to p_1 1 L$,
$p_1\varepsilon \to p_2 * R$,
$p_2 1 \to p_2 1 R$, $p_2 * \to p_2 * R$,
$p_2\varepsilon \to p_3\varepsilon L$,
$p_3 1 \to p_4\varepsilon L$, $p_3 * \to p_8\varepsilon L$,
$p_4 1 \to p_5\varepsilon L$, $p_5 1 \to p_6\varepsilon L$, $p_6 1 \to p_7\varepsilon L$,
$p_7 1 \to p_7 1 L$, $p_7 * \to p_7 * L$,
$p_7\varepsilon \to p_2 1 R$,
$p_8 1 \to p_8 1 L$, $p_8\varepsilon \to p_z\varepsilon R$.

В множестве команд мы не предусмотрели команды зацикливания, т.к. по определению машина Тьюринга остается в неопределенном состоянии, если она оказалась в такой конфигурации, в которой нет применимой команды. Такое неопределенное состояние означает зацикливание машины Тьюринга. Явное указание зацикливания можно указать, если в множество каманд включить команды

$$p_4 * \to p_4 * E,\ p_5 * \to p_5 * E,\ p_6 * \to p_6 * E.$$

Эта же машина Тьюринга, представленная в виде графа, имеет вид, представленный на рис. 4.2.

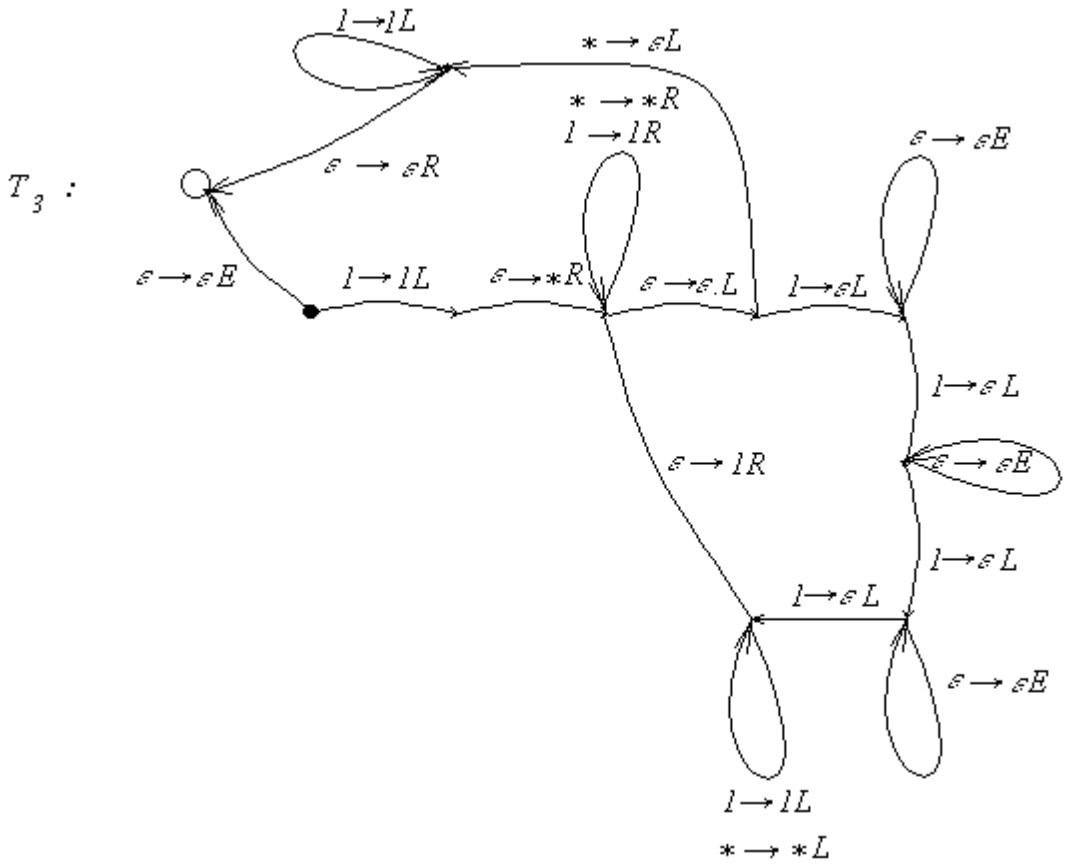


Рис. 4.2: Машина Тьюринга, вычисляющая функцию $f_3(x) = \frac{x}{4}$.

4.9.2 Варианты заданий

1. $f(x,y) = \begin{cases} \frac{x+2}{y}, & \text{если } y < 3 \\ xy, & \text{если } y \geq 3 \end{cases}$

2. $f(x,y) = \begin{cases} \frac{x-2}{y^2}, & \text{если } y < 3 \\ x+y, & \text{если } y \geq 3 \end{cases}$

3. $f(x,y,z) = \begin{cases} \frac{x-z}{y^2}, & \text{если } y < 3 \\ x+z, & \text{если } y \geq 3 \end{cases}$

4. $f(x,y,z) = \begin{cases} \frac{x-z}{3}, & \text{если } x+y < 4 \\ x+y+z, & \text{если } x+y \geq 4 \end{cases}$

5. $f(x,y,z) = \begin{cases} y-x-z, & \text{если } z < 4 \\ \frac{x+y}{z}, & \text{если } z \geq 4 \end{cases}$

6. $f(x,y,z) = \begin{cases} y-\frac{x}{z}, & \text{если } z < 3 \\ \frac{x+y+z}{3}, & \text{если } z \geq 3 \end{cases}$

7. $f(x,y,z) = \begin{cases} log_2 z, & \text{если } z < 5 \\ \frac{x+y}{2}, & \text{если } z \geq 5 \end{cases}$

7. $f(x,y) = \begin{cases} log_2 x, & \text{если } x < 8 \\ \frac{x-y}{2}, & \text{если } z \geq 8 \end{cases}$

7. $f(x,y) = \begin{cases} \sqrt{x}, & \text{если } x < 7 \\ \frac{x-3}{y}, & \text{если } x \geq 7 \end{cases}$

8. $f(x,y) = \begin{cases} \sqrt{x-y}, & \text{если } |x-y| < 5 \\ \frac{x-3}{y}, & \text{если } |x-y| \geq 5 \end{cases}$

9. $f(x,y) = \begin{cases} \frac{x-2}{y-1}, & \text{если } y < 3 \\ 3x-y, & \text{если } y \geq 3 \end{cases}$

10. $f(x,y) = \begin{cases} \frac{x-2}{y-1}, & \text{если } y < 3 \\ 3x, & \text{если } 7 > y \geq 3 \\ 2y, & \text{если } y \geq 7 \end{cases}$

11. $f(x,y,z) = \begin{cases} \frac{z-2}{y-1}, & \text{если } x+y < 3 \\ 3+x, & \text{если } 7 > x+y \geq 3 \\ 2+y, & \text{если } x+y \geq 7 \end{cases}$

12. $f(x,y,z) = \begin{cases} \sqrt{x+y-2}, & \text{если } x+y < 8 \\ x, & \text{если } 10 > x+y \geq 8 \\ 2x-z, & \text{если } x+y \geq 10 \end{cases}$

13. $f(x,y) = \begin{cases} log_3 x+y+2, & \text{если } x+y < 6 \\ x-y, & \text{если } x+y \geq 6 \end{cases}$

14. $f(x,y) = \begin{cases} log_2 x + y + 2, & \text{если } x + y < 8 \\ (x-6)^{x-5}, & \text{если } x + y \geq 8 \end{cases}$

15. $f(x,y,z) = \begin{cases} log_3 \frac{x-y}{3z}, & \text{если } x + y < 7 \\ z + (x-5)^{x-6}, & \text{если } x + y \geq 7 \end{cases}$

16. $f(x,y,z) = \begin{cases} \sqrt{\frac{x-y}{3-z}}, & \text{если } z < 7 \\ z + 2, & \text{если } z \geq 7 \end{cases}$

16. $f(x,y,z) = \begin{cases} \sqrt{\frac{x-y}{3-z}}, & \text{если } z < 7 \\ 2, & \text{если } z \geq 7 \& x > 0 \\ \frac{z}{2}, & \text{если } z \geq 7 \& x = 0 \end{cases}$

17. $f(x,y,z) = \begin{cases} \sqrt{l} log_2 z, & \text{если } z < 7 \\ 2 + z + x - y, & \text{если } z \geq 7 \& x > 0 \\ \frac{y}{2}, & \text{если } z \geq 7 \& x = 0 \end{cases}$

18. $f(x,y) = \begin{cases} \sqrt{x + \sqrt{x-2}}, & \text{если } x < 5 \\ \frac{y}{2x}, & \text{если } x \geq 5 \end{cases}$

19. $f(x,y,z) = \begin{cases} \sqrt{1 + \sqrt{x}}, & \text{если } x < 4 \\ z, & \text{если } x = 4 \\ \frac{y}{2+x}, & \text{если } x > 4 \end{cases}$

20. $f(x,y) = \begin{cases} \frac{x}{x-2}, & \text{если } x < 4 \\ x - y, & \text{если } x = 4 \\ \frac{x+1}{2}, & \text{если } x > 4 \end{cases}$

4.10 Тесты для самоконтроля к разделу

1. Какую функцию двух аргументов вычисляет алгоритм Маркова со следующими правилами:

$$*1 \longrightarrow 11\bullet$$

$$* \longrightarrow 11*$$

Варианты ответов:
а) $f(x,y) = x + y + 1$.
б) $f(x,y) = \begin{cases} x + 1, & y > 0 \\ 0, & y = 0 \end{cases}$
в) $f(x,y) = \begin{cases} x + 1, & y > 0 \\ \infty, & y = 0 \end{cases}$
г) $f(x,y) = \infty$.
д) $f(x,y) = \{y + 1, x = 0 \infty, x > 0$
Правильный ответ: в.

2. Задать Геделевскую нумерацию объектов множества M — это значит выполнить следующие действия:
1) указать алгоритм, который для каждого элемента $x \in M$ однозначно вычисляет целое число – номер элемента x;

2) указать алгоритм, который для каждого натурального числа n либо выдает восстановленный по номеру n элемент $x \in M$, либо зацикливается (при устовии, что число n не является номером какого-либо элемента множества M);

3) указать алгоритм, который для каждого натурального числа n выдает одно из двух сообщений: либо восстановленный но n элемента $x \in M$, либо сообщение о том, что число n не является номером какого-либо элемента множества M.

Правильный ответ: 1 и 3.

3. Какие проблемы из перечисленных ниже проблем неразрешимы?

1) Проблема эквивалентности алгоритмов Маркова и частично–рекурсивных функций.

2) Проблема переводимости.

3) Проблема остановки машины Тьюринга.

Правильный ответ: 2 и 3.

4. Какие из следующих утверждений истинны?

1) Если функция f построена с помощью ограниченного оператора минимизации из примитивно-рекурсивных функций, то f может быть вычислена на машине Тьюринга.

2) Если существует машина Тьюринга, правильно вычисляющая функцию f, то эта функция может быть построена с помощью ограниченного оператора минимизации из примитивно-рекурсивных функций.

3) Если функция вычислима алгоритмом Маркова, то она является примитивно—рекурсивной.

Правильный ответ: только 1.

5. Теорема Райса — это теорема

а) о неразрешимости проблемы остановки машины тьюринга;

б) об эквивалентности алгоритмов Маркова и машин Тьюринга;

в) об эквивалентности алгоритмов Маркова и примитивно—рекурсивных функций;

г) о неразрешимости проблемы эквивалентности частично–рекурсивных функций и машин Тьюринга;

д) о неразрешимости проблемы распознавания нетривиальных свойств множества.

Правильный ответ: д.

Глава 5

ТЕОРИЯ СЛОЖНОСТИ АЛГОРИТМОВ

5.1 Понятие временной и емкостной сложности алгоритмов

Для того, чтобы оценить сложность алгоритма, необходимо сначала выбрать характеристику, которая определяет величину исходных данных или их количество. Такая характеристика называется *размером задачи*. Выбор размера задачи главным образом определяется формулировкой этой задачи. Например, для любой машины Тьюринга в качестве размера задачи удобно выбрать длину исходной цепочки, а в задаче, алгоритм решения которой построен на основе обработки графа, для этой цели обычно используется число вершин соответствующего графа.

Время работы алгоритма и используемую алгоритмом память можно рассматривать как функции размера задачи n. Обычно рассматривают следующие *функции сложности* алгоритма:

$T(n)$ — временная сложность,

$C(n)$ — емкостная сложность.

Единицы измерения $T(n)$ и $C(n)$ зависят от типа исследуемой алгоритмической модели.

Обычно при выполнении алгоритма над разными исходными данными, имеющими один размер, время работы и используемая память зависят еще и от значения этих данных. Поэтому, чтобы сделать функции сложности независимыми от конкретных значений данных, как правило, рассматривают не точную аналитическую зависимость T или C от n, а оценки сложности: максимальную $T_{max}(n)$ или $C_{max}(n)$, среднюю $T_{mid}(n)$ или $C_{mid}(n)$. Причем часто представляют интерес даже не точное аналитическое представление функциональной зависимости, например $T_{max}(n)$, а просто *порядок сложности* алгоритма.

Определение 5.1.

Функция $f(n)$ есть $O(g(n))$, если существует константа C такая, что $|f(n)| < C|g(n)|$ для всех $n > 0$.

Запись $f(n) = O(g(n))$ читается: "функция $f(n)$ имеет порядок $g(n)$". Полиномиальным алгоритмом (или алгоритмом полиномиальной временной сложности) называется алгоритм, у которого

$$T(n) = O(p(n)),$$

где $p(n)$ — некоторая полиномиальная функция. Алгоритмы, временная сложность которых не поддается подобной оценке, называются экспоненциальными. Различие

Таблица 5.1

Зависимость времени работы программы
от сложности задачи

Функция временной сложности	$n = 10$	$n = 30$	$n = 60$
n	0.00001 сек.	0.00003 сек.	0.00006 сек.
n^2	0.0001 сек.	0.0009 сек.	0.0036 сек.
n^3	0.001 сек.	0.027 сек.	0.216 сек.
n^5	0.1 сек.	24.3 сек.	13.0 мин.
2^n	0.001 сек.	17.9 мин.	366 столетий
3^n	0.059 сек.	6.5 лет	$13 \cdot 10^{13}$ столетий

между двумя указанными типами алгоритмов становится особенно заметным при решении задач большого размера. В таблице 5.1 приведены скорости роста некоторых типичных полиномиальных и экспоненциальных функций.

Разные алгоритмы имеют разную временную сложность, и выяснение того, какие алгоритмы "достаточно эффективны а какие "совершенно неэффективны всегда будет зависеть от конкретной ситуации. Однако различие между полиномиальными и экспоненциальными алгоритмами становится настолько заметным при решении задач большого размера, что становятся ясными причины, по которым понятие "труднорешаемости" отождествляется с экспоненциальным характером функции временной сложности. Именно временная сложность алгоритма определяет в итоге размер задач, которые можно решить этим алгоритмом. Разные алгоритмы имеют различную временную сложность $T(n)$ и влияние того, какие алгоритмы достаточно эффективны, а какие нет, всегда зависит как от размера задачи, так и от порядка временной сложности, а при небольших размерах еще и от коэффициентов в выражении $T(n)$.

Можно было бы подумать, что колоссальный рост скорости вычислений, вызванный появлением нового поколения компьютеров, уменьшит значение эффективных алгоритмов. Однако происходит в точности противоположное. Так как вычислительные машины работают все быстрее и мы можем решать все большие задачи, именно сложность алгоритма определяет то увеличение размера задачи, которое можно достичь с увеличением скорости машины. Различие между полиномиальными и экспоненциальными алгоритмами становится еще более убедительным, если проанализировать влияние увеличения быстродействия ЭВМ на время работы алгоритмов.

Пусть n — размер задачи при использовании старой техники, m — размер задачи при использовании новой техники. Тогда из условия, что задача решается за одно и то же время на старой и новой технике, следует равенство функций временной сложности для разных размеров задачи:

$$T_{old}(n) = T_{new}(m).$$

Таблица 5.2

Зависимость размеров задач от быстродействия ЭВМ

Функция временной сложности	На современной ЭВМ	На ЭВМ в 100 раз более быстрых	На ЭВМ в 1000 раз более быстрых
n	n_1	$100n_1$	$1000n_1$
n^2	n_2	$10n_2$	$31.6n_2$
n^3	n_3	$4.64n_3$	$10n_3$
n^5	n_4	$2.5n_4$	$3.98n_4$
2^n	n_5	$n_5 + 6.64$	$n_5 + 9.97$
3^n	n_6	$n_6 + 4.19$	$n_6 + 6.29$

Но скорость новой машины в k раз выше, следовательно $T_{new}(m) = \frac{1}{k}T_{old}(m)$. Отсюда

$$T_{old}(n) = \frac{1}{k}T_{new}(m),$$

откуда можно получить соотношение $m = f(n)$ для определения нового размера задачи, решаемой за то же время на новой технике.

Примеры роста размеров задач при увеличении скорости компьютера для некоторых полиномиальных и экспоненциальных зависимостей функции временной сложности приведены в таблице 5.2. Эти данные получены для задач, решаемых за один час машинного времени, если быстродействие ЭВМ возрастает в 100 или 1000 раз по сравнению с современными компьютерами.

Таким образом, таблицы 5.1, 5.2 наглядно демонстрируют некоторые причины, по которым полиномиальные алгоритмы считаются более предпочтительными по сравнению с экспоненциальными.

Сколько вычислений должна потребовать задача, чтобы мы сочли ее труднорешаемой? Общепринято, что если задачу нельзя решить быстрее, чем за полиномиальное время, то ее следует рассматривать как труднорешаемую. Тогда при такой схеме классификации задачи, решаемые алгоритмами полиномиальной сложности, будут легко решаемыми. Но нужно иметь в виду, что хотя экспоненциальная функция (например, 2^n) растет быстрее любой полиномиальной функции от n, для небольших значений n алгоритм, требующий $O(2^n)$ времени, может оказаться эффективнее многих алгоритмов с полиномиально ограниченным временем работы. Например, функция 2^n не превосходит n^{10} до значения n, равного 59. Тем не менее, скорость роста экспоненциальной функции столь стремительна, что обычно задача называется труднорешаемой, если у всех решающих ее алгоритмов сложность по меньшей мере экспоненциальна.

5.2 Практическая оценка временной сложности

Любая программа состоит из элементов трех типов: последовательно выполняющихся участков, циклов и условных операторов, каждый из которых, в свою очередь, может иметь сложную структуру и представлять собой такие же элементы. Очевидно, что время работы последовательного участка равно сумме времени выполнения

всех его элементов. Время работы цикла любого типа можно оценить по формуле

$$T_{while} = T_{begin} + \sum_i (T_{body} + T_{next}),$$

где T_{begin} и T_{next} предназначены для выполнения начальных действий подготовки цикла и перехода к очередному шагу цикла и зависят от типа цикла, а i — условие выполнения цикла. Время T_{body} — это время выполнения тела цикла.

Время работы условного оператора вычисляется как сумма времени $T_{expression}$ вычисления условного выражения и максимального времени, которое может потребоваться для вычисления одной из ветвей:

$$T_{if} = T_{expression} + max\{T_{then} + 1,\ T_{else}\}.$$

В выражении "$T_{then} + 1$ " одна дополнительная операция означает выполнение одной команды перехода после реализации ветви $< then >$.

Рассмотрим, например, два программных фрагмента, реализующих вычисление суммы элементов матрицы $A[100][3]$. Первый из них имеет вид

```
for (i=0; i< 100; i++)
    for (j=0; j<3; j++ ) S=S+A[i][j];
```

Второй реализуется последовательностью операторов

```
for (j=0; j<3; j++)
     for (i=0; i<100; i++) S=S+A[i][j];
```

Цикл типа

$$for(i = V1;\ i <= V2;\ i++)\ O;$$

требует при выполнении число операций

$$T_{for} = T_{V_1} + 1 + \sum_{i=V_1}^{V_2} (T_O + 4 * T_{V_2}).$$

Одна операция перед циклом соответствует начальному присваиванию, а дополнительные четыре операции в цикле — это сравнение i и $V2$, условный переход, увеличение i и безусловный переход на начало цикла.

Тогда первый алгоритм потребует

$$1 + \sum_{i=1}^{100}(4 + 1 + \sum_{j=1}^{3}(4 + 2)) = 2301$$

операций, а второй

$$1 + \sum_{j=1}^{3}(4 + 1 + \sum_{i=1}^{100}(4 + 2)) = 1816$$

операций, что существенно меньше.

Анализ временной сложности рекурсивных алгоритмов приводит к рекурсивному определению этой функции:

a) $T(n_0) = const$, т.к. начальном значении $n = n_0$ нет рекурсивного хода;

b) $T(n) = f(T(g(n)))$ при рекурсивном вызове.

В зависимости от вида рекурсивной схемы можно либо попытаться подобрать вид точного аналитического выражения для $T(n)$, либо воспользоваться грубой оценкой функций. Если сложность рекурсивного алгоритма представляется следующей рекурсивной функцией

$$T(1) = d,$$

$$T(n) = aT(\frac{n}{c}) + bn, n > 1,$$

то в зависимости от a и c выражение для сложности имеет вид

$$T(n) \leq \begin{cases} O(n), & a < c \\ O(n \log_2 n), & a = c \\ O(n^{log_c a}), & a > c. \end{cases} \tag{5.1}$$

Например, для функции, заданной рекурсивной схемой

$$T(0) = 10$$

$$T(k+1) = 8T(\frac{k+1}{2}) + 41$$

грубая оценка имеет вид $T(k) = O(k^{log_2 8}) = O(k^3)$.

Не всегда следует пользоваться предложенной формулой вычисления оценки порядка сложности. Иногда можно вывести по индукции более точную формулу. Пусть, например, алгоритм имеет формулу сложности

$$T(0) = 1,$$

$$T(n+1) = (n+1)T(n).$$

В соответствии с формулой грубой оценки имеем

$$a = n+1,\ c = \frac{n+1}{n},\ T(n) = O(n^{log_{\frac{n}{n-1}}(n)} \leq O(n^n).$$

Однако, анализ рекурсивной зависимости позволяет получить более точную формулу $O(n!)$. Действительно, допустим, соотношение $T(n) = n!$ справедливо для любого n. Докажем справедливость этого равенства для $n+1$:

$$T(n+1) = (n+1)T(n) = (n+1) \cdot n! = (n+1)!$$

.

5.3 NP–полные задачи

Большинство экспоненциальных алгоритмов — это просто варианты полного перебора, в то время как полиномиальные алгоритмы обычно можно построить лишь тогда, когда удается найти решение без перебора всех допустимых вариантов данных.

Как уже отмечалось в 5.1, если задача решается за полиномиальное время

$$T(n) = P_k(n) = \sum_{i=0}^{k} a_i n^i,$$

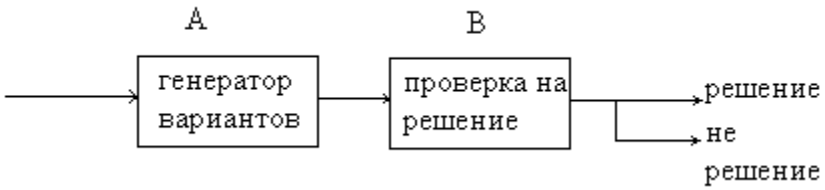


Рис. 5.1: Недетерминированный алгоритм, состоящий из двух фрагментов.

то обычно считается, что эта задача является легко решаемой. Поэтому среди множества всех задач выделен класс P–задач, для которых существует детерминированный алгоритм, решающий эту задачу за полиномиальное время.

Будем называть задачу *труднорешаемой*, если для ее решения не существует полиномиального алгоритма.

Определение 5.2. Класс задач, для решения которых существует недетерминированный алгоритм, решающий эту задачу за полиномиальное время, называется классом NP–задач.

Недетерминированный алгоритм всегда должен выдавать на выходе одно из двух сообщений: "получено решение" или "решение не получено".

Смоделировать такой недетерминированный алгоритм T можно, формируя этот алгоритм из двух частей A и B, которые работают последовательно одна за другой и $T = A \cdot B$. Эти составные части представляют собой недетерминированный алгоритм угадывания и детерминированный алгоритм проверки (см. рис. 5.1). Стадия A — недетерминированное начало алгоритма, стадия B — его детерминированное завершение, как правило, отвечающее на вопрос, построил ли недетерминированный алгоритм угадывания A решение или нет.

Недетерминированность алгоритма T означает, что при запуске многих экземпляров этого алгоритма какой–то из них, возможно, получит решение задачи. Если все возможные варианты запуска экземпляров алгоритма T не получили решения, значит, решение задачи не существует. Тогда можно построить детерминированную модель недетерминированного алгоритма T, основанную на переборе всевозможных вариантов работы алгоритма угадывания A.

Начальный участок алгоритма A формирует какой–то (очередной) вариант данных, которые могут быть (или не быть) решением задачи. Вторая часть — алгоритм B — получает сгенерированный вариант и проверяет, является ли он решением или нет. Если решения не достигнуто, вновь инициируется алгоритм A для получения нового варианта решения (см. рис. 5.2).

Всякая задача, разрешимая за полиномиальное время детерминированным алгоритмом, разрешима также за полиномиальное время недетерминированным алгоритмом, т.е. класс P–задач входит в класс NP–задач. Чтобы убедиться в этом, достаточно заметить, что любой детерминированный алгоритм может быть использован в качестве стадии проверки недетерминированного алгоритма. Стадия угадывания при этом игнорируется, т.е. представляет собой пустой алгоритм с нулевым временем работы. Таким образом, $P \subseteq NP$.

Есть много причин считать включение P–задач в множество NP–задач строгим. Полиномиальные недетерминированные алгоритмы определенно оказываются более мощными, чем полиномиальные детерминированные алгоритмы, и не известны общие методы их превращения в детерминированные полиномиальные алгоритмы. В действительности самый сильный из известных в настоящее время результатов состоит в следующем.

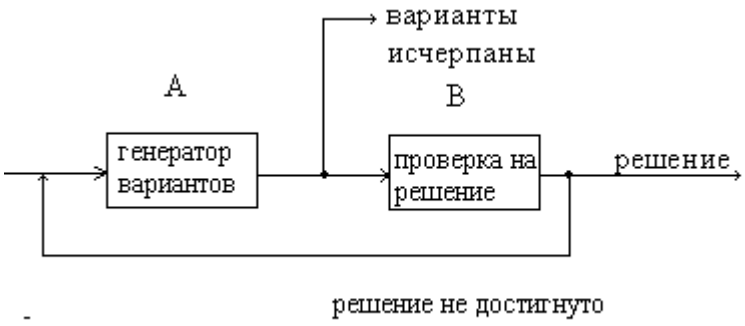


Рис. 5.2: Детерминированная модель недетерминированного алгоритма.

Теорема 5.1. Если задача $Z \in NP$, то существует такой полином $p(n)$, что задача Z может быть решена детерминированным алгоритмом с временной сложностью $O(2^{p(n)})$.

Доказательство. Пусть T — полиномиальный недетерминированный алгоритм решения задачи Z. Тогда существует полином $q(n)$, ограничивающий временную сложность алгоритма T. По определению класса NP, для каждого набора исходных данных длины n найдется некоторая последовательность данных, представляющая собой слово–догадку, длины не более $q(n)$. При обработке этой последовательности–догадки алгоритм T на стадии проверки работает и выдает ответ "да" или "нет" за $q(n)$ шагов. Таким образом, общее число догадок, которые нужно рассмотреть, не превосходит $k^{q(n)}$, где k - число символов, из которых состоит слово–догадка (если слово–догадка короче $q(n)$, его можно дополнить пустыми символами и всегда рассматривать как слово длины $q(n)$).

Теперь построим детерминированый алгоритм решения задачи Z. Для этого достаточно последовательно генерировать все слова–догадки в количестве $k^{q(n)}$ и для каждой из них запустить детерминированную стадию проверки алгоритма T, который работает не более $q(n)$ шагов. Этот алгоритм даст ответ "да" или "нет" и всегда выполняет действия проверки за время, представляющее собой некоторую константу. Теперь достаточно добавить алгоритм анализа ответа, останавливающий процесс генерации очередного слова–догадки и, следовательно, завершающий весь алгоритм. Построенный нами алгоритм, очевидно, будет детерминированным алгоритмом, работающим с временной сложностью $q(n) \cdot k^{q(n)}$. Логарифмируя эту экспоненту, а затем переходя к двоичным логарифмам, получим, что эта сложность не превосходит $O(2^{p(n)})$, где $p(n)$ — полином.□

Безусловно, процесс моделирования, предложенный в доказательстве этой теоремы, можно в некоторой степени ускорить с помощью более тщательного перебора, когда избегаются с очевидностью ненужные слова–догадки. Тем не менее, несмотря на значительную экономию, которая может быть при этом достигнута, не известен метод, осуществляющий такое моделирование быстрее, чем за экспоненциальное время. Таким образом, способность недетерминированного алгоритма проверить за полиномиальное время экспоненциальное число возможностей может навести на мысль, что полиномиальные недетерминированные алгоритмы являются более мощным средством, чем полиномиальные детерминированные алгоритмы. Не удивляет поэтому широко распространенное мнение, что $NP \neq P$, хотя доказательство этой гипотезы отсутствует. На основе накопленного опыта будем представлять себе класс NP так, как от изображен на рис. 5.3, ожидая, что затененная область, обозначающая $NP \setminus P$, не пуста.

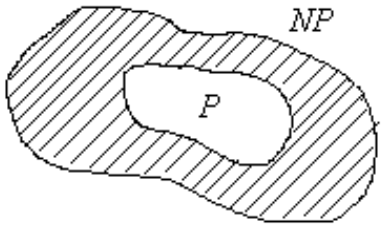


Рис. 5.3: Предполагаемое соотношение между классами P и NP.

Вопрос о соотношении классов P и NP имеет принципиальное значение для теории NP–полных задач. В настоящее время строго математически еще не доказано соотношение $P \subset NP$, однако, при существующем уровне знаний и накопленном опыте программирования естественно считать $P \subset NP$. Если P не совпадает с NP, то различие между P и $NP \backslash P$ очень существенно. Все задачи из P могут быть решены полиномиальными алгоритмами, а все задачи $NP \backslash P$ труднорешаемы.

В классе NP содержатся NP–полные задачи. Это NP–задачи, для решения которых не существует детерминированного алгоритма, работающего за полиномиальное время.

Для доказательства NP–полноты некоторой задачи A можно использовать несколько различных методов:

– провести независимое доказательство для задачи A;

– воспользоваться известным доказательством NP–полноты некоторой задачи B и провести доказательство NP–полноты задачи A по аналогии;

– воспользоваться методом сужения задачи, который заключается в установлении того факта, что поставленная задача A включает в качестве частного случая известную NP–полную задачу;

– воспользоваться полиномиальной сводимостью.

Первые два пути сложны, на практике обычно используется третий или четвертый метод. Если поставленная задача A сводится с помощью алгоритма с полиномиальной временной сложностью к другой задаче B, то любой полиномиальный алгоритм решения второй задачи B может быть превращен в полиномиальный алгоритм решения первой задачи A. Рассмотрим задачи, которые часто используются в качестве базовых NP–полных задач для доказательства NP–полноты других задач.

Список NP–полных задач достаточно велик и содержит сотни задач из различных областей математики: теории графов, математического программирования, теории расписаний, теории языков и автоматов, математической логики и др. Это значит, что при решении необозримого числа практических задач приходится сталкиваться с проблемами NP—полноты. Опытный программист, встречаясь с новой задачей и предполагая ее NP–полноту, должен иметь инструмент для такого доказательства. Используя метод сведения к известным NP–полным задачам, программист должен уметь выбрать из сотен известных задач ту, которая лучше всего подходит в качестве основы искомого доказательства. Хотя теоретически любую из известных NP–полных задач можно наравне с другими выбрать для доказательства NP–полноты новой задачи, на практите оказывается, что некоторые задачи подходят для этой цели гораздо лучше других. Следующие шесть задач входят в список тех, кото-

рые используются наиболее часто и для начинающих они могут служить основным ядром списка известных NP–полных задач.

1. (Выполнимость). Дан набор $C = C_1, \ldots, C_m$ дизъюнкций на конечном множестве переменных U. Существует ли на U набор значений истинности, при котором выполняются все дизъюнкции из C?

2. (Трехмерное сочетание). Дано множество $M \subseteq W \times X \times Y$, причем W, X и Y — непересекающиеся множества, содержащие одинаковое число элементов q, $q = |W| = |X| = |Y|$. Содержится ли в M подмножество $N \subseteq M$, такое, что $|N| = q$ и никакие два разных элемента N не имеют ни одной равной координаты?

3. (Гамильтонов цикл). Имеет ли данный неориентированный граф гамильтонов цикл?

Чтобы получить конкретные представления о содержании данной задачи, вспомним некоторые определения классической теории графов. Пусть $g = (V, E)$ — граф с множеством вершин V и множеством ребер E. *Простым циклом* в графе G называется такая последовательность $\langle v_1, v_2, ..., v_k \rangle$ различных вершин из V, что $\{v_i, v_{i+1}\} \in E$ для $1 \leq i < k$ и $\{v_k, v_1 \in E\}$. *Гамильтоновым циклом* в графе G называется простой цикл, содержащий все вершины графа G.

4. (Раскрашиваемость). Задан граф $G = (V, E)$ и положительное целое число $k \leq |V|$. Является ли данный неориентированный граф k–раскрашиваемым?

Граф называется k–раскрашиваемым, если каждой вершине графа можно поставить в соответствие такое число j (называемое "цветом" вершины), что любые две соседние вершины графа имеют разный цвет. Другими словами, граф $G = (V, E)$ называется k–раскрашиваемым, если существует такая функция $f : V \to \{1, 2, ..., k\}$, что для любой дуги $\{v_i, v_l\} \in E$ имеет место неравенство $f(v_i) \neq f(v_l)$, обозначающее тот факт, что никаким двум смежным вершинам не приписан один и тот же цвет.

5. (Клика). Содержит ли данный граф $G = (V, E)$ некоторую клику мощности не менее заданного целого N.

Кликой мощности не менее N называется такое подмножество вершин $\tilde{V} \subseteq V$, что $|\tilde{V}| \geq N$ и любые две вершины из $\tilde{V}$ соединены ребром в G.

6. (Разбиение). Задано конечное множество A и вес $S(a)$ каждого элемента $a \in A$. Существует ли множество $\tilde{A} \subseteq A$ такое, что

$$\sum_{a \in \tilde{A}} S(a) = \sum_{a \in A \setminus \tilde{A}} S(a)?$$

Используя перечисленные выше задачи в качестве базовых, легко можно доказать NP–полноту достаточно широкого круга задач. В качестве примера такого доказательства NP–полноты рассмотрим *задачу о рюкзаке*. Пусть имеется конечное множество A, для каждого элемента $a \in A$ задана стоимость $S(a)$ и вес $V(a)$. Необходимо выбрать такое подмножество $B \subseteq A$, чтобы, во–первых, рюкзак можно было поднять с учетом его веса и, во–вторых, стоимость предметов в нем была не менее заданной. Математически формулировка задачи описывается следующими ограничениями:

$$\sum_{a \in B} S(a) \geq M \quad \text{и} \quad \sum_{a \in B} V(a) \leq K,$$

где M и K— заданные целые числа.

Для доказательства NP–полноты данной задачи ограничимся рассмотрением только таких индивидуальных задач, в которых для всех $a \in A$ выполняется равенство $S(a) = V(a)$ и

$$M = K = 0.5 \sum_{a \in A} S(a).$$

Тогда задача превращается в задачу о разбиении. Действительно,

$$\sum_{a\in B} S(a) \geq M = K, \quad \sum_{a\in B} V(a) \leq K.$$

Но $\sum_{a\in B} V(a) = \sum_{a\in B} S(a)$, тогда $\sum_{a\in B} S(a) \geq K$ и $\sum_{a\in B} S(a) \leq K$ возможно только при $\sum_{a\in B} S(a) = K = 0.5\sum_{a\in A} S(a)$. Следовательно, стоимость элементов, не вошедших в B, также должна быть равна K. Получили, что для решения данной задачи необходимо найти такое множество $B \subseteq A$, чтобы

$$\sum_{a\in B} S(a) = \sum_{a\in A\setminus B} S(a),$$

что является задачей о разбиении.

В соответствии с представлением алгоритма решения NP–задач с помощью алгоритма угадывания и алгоритма проверки NP–полные задачи требуют полного перебора и решаются рекурсивно, так, что алгоритм поиска решения задачи размера n на каждом шаге рассматривает все возможные варианты решений на глубину 1 и оставшуюся задачу меньшего размера $n-1$.

5.4 NP-полнота задачи о дизъюнкциях

Одной из самых известных NP–полных задач является задача о дизъюнкциях, которую обычно называют задачей "выполнимость". Вспомним формулировку этой задачи.

Дан набор $C = C_1, \ldots, C_m$ дизъюнкций на конечном множестве переменных U. Существует ли на U набор значений истинности, при котором выполняются все дизъюнкции из C?

Теорема 5.2 (Теорема Кука). Задача "выполнимость" есть NP–полная задача.

Доказательство. Как уже отмечалось выше, вопрос о взаимоотношении классов P–задач и NP–задач имеет фундаментальное значение. До тех пор, пока не доказано соотношение $P \neq NP$, нет никакой надежды показать, что некоторая конкретная задача принадлежит классу $NP \setminus P$. По этой причине доказательство NP-полноты задачи Z заключается в доказательстве более слабого утверждения вида: "если $P \neq NP$, то $Z \in NP\setminus P$". Тогда для доказательства NP-полноты необходимо и достаточно доказать два условия, соответствующие определению NP-полной задачи:

1) задача должна лежать в классе NP-задач, т.е. решаться недетерминированным алгоритмом за полиномиальное время;

2) любая NP–полная задача (если такая существует, в соответствии с приведенными выше замечаниями), полиномиально сводится к задаче Z.

Легко видеть, что первое требование выполняется. Недетерминированному алгоритму для решения этой задачи достаточно получить на вход набор значений истинности булевских переменных, а затем выполнить проверку истинности всех заданных дизъюнкций. Эта операция выполняется за время $O(n)$, т.е. за полиномиальное время.

Рассмотрим теперь второе требование полиномиальной сводимости произвольной NP–полной задачи к задаче о дизъюнкциях. Для этого вернемся к уровню языков описания слов–догадок — исходных данных для полиномиальных недетерминированных алгоритмов. Разные языки из NP могут сильно отличаться, число этих языков бесконечно, поэтому невозможно указать отдельное сведение для каждого из них.

Однако, каждый из них является исходной информацией для алгоритма, а каждый алгоритм можно представить машиной Тьюринга. Пусть произвольная NP–полная задача Z решается недетерминированным алгоритмом с полиномиальной временной сложностью $p(n)$, детерминированная фаза проверки которого реализуется машиной Тьюринга $T = (K, \Sigma, \delta, p_0, f, a_0, a_1)$. При решении задачи на вход машины Тьюринга T подается цепочка исходных данных x. Покажем теперь, что из машины Тьюринга T и цепочки x можно построить булевскую формулу, которая принимает значение "истина" тогда и только тогда, когда машина Тьюринга T допускает цепочку x, т.е. выдает сообщение о достижении алгоритмом Z решения на соответствующих данных x. Если x является решением задачи Z, то T переходит в заключительное состояние и записывает на ленту 1. Если x не является решением задачи Z, то T в зависимости от типа поставленной задачи может либо зациклиться, либо перейти в заключительное состояние и записать на ленту 0. В этом последнем случае легко дополнить множество команд T командами чтения результата и зацикливания, если результат равен 0. Таким образом, всегда можем считать, что T останавливается только в том случае, когда x — решение Z.

По определению временной сложности NP–полной задачи, машина Тьюринга T достигает решения для исходной цепочки w длины n не более, чем за $p(n)$ шагов, следовательно, в любой момент работы машины Тьюринга T на ленте занято не болеее $2 \cdot p(n) + 1$ ячеек. Это ограничение на число занятых ячеек следует из того факта, что головка в начальный момент обозревает первую ячейку исходных данных, и на каждом шаге сдвигается на одну ячейку вперед или назад. Таким образом, если первую ячейку исходной цепочки считать имеющией номер 1, то машина Тьюринга T может использовать ячейки от $-p(n)$ до $p(n) + 1$.

Примем соглашение, что если машина Тьюринга T закончит вычисления раньше момента времени $p(n)$, то конфигурация остается неизменной во все моменты времени после остановки, т.е. сохраняется заключительное состояние, положение головки и запись на ленте. Пронумеруем все состояния от 0 до $|K|$, алфавит ленты от 0 до $|\Sigma|$, припишем номера ячейкам ленты используемого участка от $-p(n)$ до $p(n) + 1$. При нумерации состояний условимся сделать заключительное состояние состоянием с номером 1, а начальное — с номером 0.

Введем три типа логических переменных, причем каждый тип будет представлен массивом таких переменных:

1) $Q[i][k]$ — в момент времени i машина Тьюринга T находится в состоянии q_k;

2) $H[i][j]$ — в момент времени i головка обозревает ячейку с номером j;

3) $S[i][j][k]$ — в момент времени i в ячейке j записан символ с номером k.

Построим теперь шесть групп дизъюнкций, каждая из которых будет налагать ограничения определенного типа на любой выполняющий набор значений истинности, а все вместе — соответствовать условию истинности только в том случае, когда x является решением задачи Z и машина Тьюринга T перейдет в заключительное состояние с номером 1. Это следующие группы дизъюнкций:

1) G_1 — в любой момент времени i программа находится ровно в одном состоянии;

2) G_2 — в любой момент времени i головка обозревает ровно одну ячейку;

3) G_3 — в любой момент времени i каждая ячейка содержит ровно один символ алфавита Σ;

4) G_4 — в момент времени 0 головка находится в начале цепочки x, которая записана на ленте;

5) G_5 — не позднее, чем через $p(n)$ шагов T переходит в состояние с номером 1 и, следовательно, допускает x;

6) G_6 — в любой момент времени i ($p(n) \geq i \geq 0$) конфигурация в следующий

момент времени $i+1$ получается с использованием команд из δ.

Легко видеть, что если все шесть групп дизъюнкций действительно осуществляют поставленные цели, то выполняющий набор значений истинности обязан соответствовать значению x на входе, которое является решением задачи Z. Единственное, что остается — это указать способ построения групп дизъюнкций, осуществляющих эти цели.

Рассмотрим группу G_1. В эту группу надо включить дизъюнкции двух типов:

а) $Q[i][0] \vee Q[i][1] \vee ... \vee Q[i][r]$ для всех i от 0 до $p(n)$, r — максимальный номер состояния T; эти дизъюнкции могут быть выполнены, тогда и только тогда, когда в каждый момент времени T находится по крайней мере в одном состоянии;

б) $\neg Q[i][j] \vee \neg Q[i][j']$ для $0 \leq j < j' <| K |$ и для всех i от 0 до $p(n)$; эти дизъюнкции могут быть выполнены тогда и только тогда, когда в каждый момент времени машина Тьюринга T находится не более, чем в одном состоянии;

Дизъюнкции G_2 и G_3 строятся аналогично G_1 для групп переменных $H[i][j]$ и $S[i][j]$ соответственно, а дизъюнкции G_4 и G_5 очень просты. Действительно, поскольку мы потребовали, чтобы машина Тьюринга оставалась в заключительном состоянии после того, как оно достигнуто, то G_5 состоит из одной переменной $Q[p(n)][1]$. Группа дизъюнкций G_4 строится аналогично и состоит из следующих четырех элементов:

а) одной переменной $Q[0][0]$ — в начальный момент времени машина Тьюринга находится в состоянии с номером 0;

б) одной переменной $H[0][1]$ — в начальный момент головка обозревает первый символ исходной цепочки, т.е. ячейку с номером 1;

в) одной переменной $S[0][0][0]$ — в начальный момент времени слева от исходной цепочки в нулевой ячейке пустой символ;

г) совокупности пременных $S[0][1][x_1]$, $S[0][2][x_2]$, ..., $S[0][n][x_n]$ и $S[0][n+1][0]$, $S[0][k+2][0]$, ..., $S[0][p(n)][0]$ — в начальный момент на ленте в ячейках $1, 2, \ldots n$ записана исходная цепочка $x = x_1 x_2 \ldots x_n$, за которой находятся пустые ячейки.

Рассмотрим теперь G_6. В эту группу надо включить дизъюнкции двух типов:

а) если головка не обозревает в момент времени i ячейку j, то содержимое ячейки не изменится, т.е. $\neg S[i][j][k] \vee H[i][j] \vee S[i+1][j][k]$ для всех $0 \leq i \leq p(n)$, $-p(n) \leq j \leq p(n)$, $0 \leq k \leq |\Sigma|$;

б) изменение конфигурации происходит строго в соответствии с командами из δ, т.е. для любой команды надо определить состояние в i-ый и в $(i+1)$-ый момент. Для того, чтобы записать дизъюнкции, определяющие переход в новую конфигурацию в соответствии с командами машины Тьюринга, воспользуемся записью импликации $c_0 \to c_1$, где c_0 и c_1 — описание конфигурации до выполнения команды и после ее выполнения. Вспомним, что импликация $c_0 \to c_1$ эквивалентна $\neg c_0 \vee c_1$. Тогда группа содержит следующие три дизъюнкции для каждой команды $ua \to vbr$ из множества команд δ:

$$\neg H[i][j] \vee \neg Q[i][u] \vee \neg S[i][j][a] \vee H[i+1][j+\tilde{r}] -$$

в следующий момент осуществляется сдвиг головки в направлении r, где $\tilde{r}$ равно 0, 1, -1 для r, равного соответственно E, R, L;

$$\neg H[i][j] \vee \neg Q[i][u] \vee \neg S[i][j][a] \vee Q[i+1][v] -$$

в следующий момент осуществляется переход в новое состояние v;

$$\neg H[i][j] \vee \neg Q[i][u] \vee \neg S[i][j][a] \vee S[i+1][j][b] -$$

в следующий момент в ячейке находится новый символ b.

Каждая полученная нами булевская формула состоит не более чем из $O(p^3(n))$ символов, если считать каждую переменную $H[i][j]$, $Q[i][j]$, $S[i][j][k]$ за один символ. Мы не наложили на исходную машину Тьюринга T никаких ограничений, кроме условия, что соответствующая задача является NP–полной. Поэтому мы показали, что любая NP–полная задача полиномиально сводится к задаче выполнимости булевских формул. □

5.5 Несколько NP–полных задач

Если бы все доказательства NP–полноты были так же сложны, как доказательство NP–полноты задачи о дизъюнкциях, то очень сомнительно, чтобы список NP–полных задач смог бы быть столь обширным, как в настоящее время. Однако, как уже отмечалась ранее в параграфе 5.3, если уже известна одна NP –полная задача, то процедура доказательства NP–полноты других задач существенно упрощается.

Для доказательства NP–полноты заданной задачи Z можно использовать *метод сводимости*: достаточно показать, что какая–нибудь из известных NP–полных задач может быть сведена к Z.

Второй из простых методов доказательства NP–полноты основан на *методе сужения*, т.е. на поиске такой известной NP–полной задачи, которая является частным случаем задачи Z. Суть метода сужения состоит в том, чтобы найти такие дополнительные ограничения на поставленную задачу Z, чтобы получившаяся в результате таких ограничений задача была эквивалентна некоторой известной NP–полной задаче. Например, в качестве анализируемой на NP–полноту задачи Z рассмотрим задачу поиска ориентированного гамильтонова цикла в ориентированной графе. Среди индивидуальных подзадач задачи Z имеется задача поиска указанного цикла в таком ориентированном графе, который вместе с каждой ориентированной дугой (v_i, v_j) содержит ей обратную (v_j, v_i). Очевидно, что поиск гамильтонова цикла в таком ориентированном графе эквивалентен NP–полной задаче поиска гамильтонова цикла в неориентированном графе.

В параграфе 5.3 приведен список из шести известных NP–полных задач, которые часто используются при доказательстве NP–полноты. Рассмотрим еще несколько таких задач.

Задача 1. *Расписание для мультипроцессорной системы.* Задано конечное множество A заданий, которые необходимо выполнить в мультипроцессорной системе, состоящей из m процессоров. Мультипроцессорная система — это система, в которой имеется несколько независимых процессоров, на каждом из которых задача решается независимо от загрузки остальных процессоров. Для каждой задачи $a \in A$ известна длительность $t(a) \in N$ ее решения. Задано время T, в течение которого необходимо решить все задачи множества A.

Вопрос: существует ли такое разбиение множества A на m непересекающихся подмножеств $A = A_1 \cup A_2 \cup ... \cup A_m$, такое, что

$$max\{\Sigma_{a \in A_i} t(a)\} \leq T.$$

Другими словами, можно ли так распределить задачи по процессорам, чтобы при параллельном решении этих задач общее время решения всей совокупности задач не превышало заданное T.

Задача 2. *Минимальный набор тестов.* Задано конечное множество A возможных диагнозов заболевания. Известно, что некоторые заболевания имеют частично

совпадающие симптомы, поэтому задан набор C подмножеств множества A, представляющий двоичные тесты. Имеется некоторое натуральное число m, ограничивающее число тестов, которые должен пройти один больной.

Вопрос: существует ли такое множество тестов $C_1 \subseteq C$, $|C_1| \leq m$, что для любой пары (a_i, a_j) возможных диагнозов из A имеется некоторый тест, различающий a_i и a_j, т.е. такой тест $c \in C$, для которого $|\{a_i, a_j\} \cap c| = 1$.

Задача 3. *Доминирующее множество.* Задан граф $G = (V, E)$ и положительное целое число $K \leq |V|$.

Вопрос: существует ли в графе G доминирующее множество вершин V_1, состоящее не более, чем из K элементов, такое, что $|V_1| \leq K$ и для всех вершин $u \in V \setminus V_1$ существует в G дуга $(u, v) \in E$, связывающая вершину u с какой–либо вершиной v из множества V_1.

Задача 4. *Изоморфизм подграфу.* Заданы два графа $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$.

Вопрос: содержит ли граф G_1 подграф, изоморфный графу G_2? Иначе говоря, существуют ли подмножества V и E, $V \subseteq V_1$, $E \subseteq E_1$, такие, что $|V| = |V_2|$, $|E| = |E_2|$, и такая взаимно–однозначная функция $f : V \to V_2$, что ребро $(u, v) \in E$ тогда и только тогда, когда $(f(u), f(v)) \in E_2$.

Задача 5. *Изоморфный остов.* Задан граф $G = (V, E)$ и дерево $T = (V_t, E_t)$.

Вопрос: верно ли что граф G содержит остовное дерево, изоморфное T?

Задача 6. *Коммивояжер.* Задано множество $C = \{c_1, c_2, ..., c_m\}$ городов. Между каждой парой городов задано расстояние $d(c_i, c_j)$, являющееся натуральным числом. Задано также положительное целое число K.

Вопрос: Существует ли маршрут длины не более K, проходящий через все города из C? Иными словами, существует ли такая перестановка городов $\langle c_{n(1)}, c_{n(2)}, ..., c_{n(m)} \rangle$, что

$$\sum_{i=1}^{m-1} d(c_{n(i)}, c_{n(i+1)}) + d(c_{n(m)}, c_{n(1)}) \leq K.$$

Задача 7. *Сельский почтальон.* Задан граф $G = (V, E)$, длина $l(e)$ каждого ребра $e \in E$ и положительное целое число K.

Вопрос: Существует ли в G цикл, сумма длин ребер которого не меньше K?

Задача 8. *Упаковка множеств.* Задан набор C конечных множеств и положительное целое число $K \leq |C|$.

Вопрос: Верно ли, что множество C содержит по крайней мере K непересекающихся подмножеств?

Замечание: задача остается NP–полной даже при $|c_i| \leq 3$ для всех $c_i \in C$. С помощью метода паросочетаний задача решается за полиномиальное время, когда $|c_i| \leq 2$ для всех $c_i \in C$.

Задача 9. *Расщепление множества.* Задан набор C конечных подмножеств множества S.

Вопрос: Существует ли такое разбиение множества S на два подмножества S_1 и S_2, что ни одно подмножество из C не содержится целиком ни в S_1, ни в S_2?

Замечание: задача остается NP–полной даже при $|c_i| \leq 3$ для всех $c_i \in C$. Задача решается за полиномиальное время, когда $|c_i| \leq 2$ для всех $c_i \in C$ (такая задача называется *раскрашиваемость графа в два цвета*).

Задача 10. *Упаковка в контейнеры.* Задано конечное множество

$$U = \{u_1, u_2, ..., u_m\}$$

предметов, для каждого из которых задано натуральное число $s(u_i)$ — линейный размер предмета u_i. Даны также положительное целое число B — вместимость контейнера и число K — количество контейнеров.

Вопрос: Существует ли такое разбиение множества U на K непересекающихся подмножеств U_1, U_2,...,U_K, что сумма размеров предметов из каждого подмножества U_i не превосходит B?

Задача 11. *Квадратичные сравнения.* Заданы положительные целые числа a, b и c.

Вопрос: Существует ли положительное целое число $x < c$, такое, что $x^2 = a(mod\, b)$?

Замечание: задача остается NP–полной, даже если в условии даны разложение числв b на простые множители и решение данного сравнения по всем простым модулям, входящим в разложение числа b.

Задача 12. *Составление кроссворда.* Заданы конечное множество слов W и матрица A из нулей и единиц размером $n \times n$.

Вопрос: Можно ли составить кроссворд из слов множества W, чтобы слова вписывались в клетки матрицы A, соответствующие нулям в ней?

Задача 13. *Отсутствие тавтологии.* Задано булевское выражение E на множестве переменных $U = \{u_1, u_2, ..., u_k\}$. В выражении используются логические операции $\wedge$ (и), $\vee$ (или), $\neg$ (не), $\rightarrow$ (следует).

Вопрос: Верно ли, что E не тавтология, т.е. существует ли такой набор значений переменных $u_1, u_2, ..., u_k$, что E принимает значение "ложь"?

Задача 14. *Программы с рекурсивными функциями.* Задано конечное множество идентификаторов функций языка Си. Для каждой функции приводится тело функции, содержащее, в частности, вызовы других функций.

Вопрос: Верно ли, что некоторая функция из множества функций A рекурсивна и вызывает не менее k других функций?

Задача 15. *Составление учебного расписания.* Заданы конечное множество H рабочих часов, множество C преподавателей, множество D учебных дисциплин. Для каждого преподавателя $c_i \in C$ дано подмножество $A(c_i) \subseteq H$, называемое допустимыми часами для преподавателя c_i. Для каждой дисциплины $d_j \in D$ дано подмножество $A(d_j) \subseteq H$, называемое допустимыми часами для дисциплины d_j. Для каждой пары $(c_i, d_j) \subseteq C \times D$ задано число $r(c_i, d_j)$, называемое требуемой нагрузкой.

Вопрос: Существует ли учебное расписание, обслуживающее все дисциплины? Другими словами, существует ли функция

$$f : C \times D \times H \rightarrow \{0, 1\},$$

где $f(c_i, d_j, h_k) = 1$ означает, что преподаватель c_i занимается дисциплиной d_j в момент времени h_k.

Задача 16. *Динамическое распределение памяти.* Задано конечное множество $A = \{a_1, a_2, ..., a_k\}$ элементов динамических данных языка Си. Известен требуемый размер памяти в байтах $s(a_i)$, который необходимо выделить для каждого элемента $a_i \in A$. Для каждого данного $a_i \in A$ известно время $n(a_i)$, когда требуется выполнить выделение памяти для этого данного, и время $f(a_i)$ окончания работы с ним. Кроме того, известно положительное целое число M — максимальный размер в байтах области памяти, в которой размещаются все данные.

Вопрос: Существует ли для множества данных A допустимое распределение памяти? Иначе говоря, существует ли такая функция

$$g : A \rightarrow \{1, 2, 3, ..., M\},$$

что для каждого элемента $a_i \in A$ выделен непрерывный участок памяти

$$[g(a_i), g(a_i) + s(a_i) - 1],$$

целиком содержащийся в допустимой области $[1, M]$, причем в одно и то же время разные данные не могут занимать пересекающиеся участки памяти.

Задача 17. *Дерево с заданными вершинами.* Дан граф $G = (V, E)$, подмножество вершин $V_1 \subseteq V$ и целое число $K \leq |V_1| - 1$.

Вопрос: существует ли поддерево в G, содержащее все вершины из V_1 и имеющее не более K ребер?

Задача 18. *Квадратные диофантовы уравнения.* В заключение рассмотрим задачу, иллюстрирующую процесс получения различного спектра задач — от класса P до неразрешимых.

Задача "квадратные диофантовы уравнения"состоит в следующем. Пусть заданы положительные целые числа a, b, c.

Вопрос: существуют ли положительные целые числа x и y, такие , что $ax^2+by = c$? Эта задача является NP–полной. Задача решения диофантова уравнения $ax^k = c$ разрешима в целых числа за полиномиальное время для произвольных k. Общая диофантова задача "найти целый корень для заданого полинома от k переменных с целыми коэффициентами " неразрешима.

5.6 Методы решения NP-полных задач

В соответствии с представлением алгоритма решения NP–полных задач с помощью алгоритма угадывания и алгоритма проверки программы, реализующие NP–полные задачи, требуют полного перебора вариантов и решаются рекурсивно, так, что алгоритм поиска решения на каждом их шаге рассматривает все возможные варианты решений на глубину 1 и оставшуюся задачу меньшего размера.

Рассмотрим в качестве примера решение следующей задачи. Пусть имеется произвольное клеточное поле и плитки размером 1×2. Необходимо покрыть данное поле такими плитками. Очевидно, что произвольное клеточное поле можно представить матрицей, в которой клетки заданного поля имеют следующие значения:

а) 0 — свободны,

б) число $n > 0$ — клетка занята плиткой с номером n,

в) число -1 — не принадлежащие полю клетки.

Сначала все свободные клетки заняты нулями, а все клетки, не подлежащие заполнению, числом -1. Приведем пример подлежащего заполнению поля:

0	0	-1	-1	-1
-1	0	-1	-1	-1
-1	0	0	0	0
-1	-1	0	0	0

В дальнейшем, по мере покрытия, нули в свободных клетках будем заменять на номера положенных туда плиток. Поскольку на каждом шаге придется рассматривать соседние клетки вправо и вниз, для того, чтобы на границе анализ ситуации не отличался от анализа в середине поля, добавим к матрице дополнительную строку и столбец, как показано на рисунке:

1	1	-1	-1	-1	-1
-1	2	-1	-1	-1	-1
-1	2	3	4	4	-1
-1	-1	3	5	5	-1
-1	-1	-1	-1	-1	-1

Тогда для размещения очередной плитки необходимо сначала найти первую свободную клетку. Попытка положить плитку вправо является успешной, если, во–первых, справа есть пустая клетка, а, во–вторых, если дальнейшее решение на следующих уровнях рекурсии позволит выполнить покрытие. Попытка положить плитку вниз предпринимается только при неудачной попытке положить плитку вправо и выполняется аналогично. Задача решена успешно, если все попытки на всех уровнях рекурсии оказались успешными. Если полный перебор укладки плиток не привел к решению, следовательно задача решения не имеет. Следующая программа выполняет указанные действия.

```
// укладка плиток 1*2  на заданное поле
#include <stdio.h>
#include <STDLIB.H>

#define  MAXK  20
int Q[MAXK+1][MAXK+1];    // клеточное поле
int  M, N;      //число строк и столбцов

FILE *in = fopen("plitka.in","r");
FILE *out= fopen("plitka.out","w");

int Set(int n)
// уложить одну очередную плитку
{
int i,j,in,jn,flag=0;
for (i=0; (i<M) && (flag==0); i++)
    for(j=0; (j<N) && (flag==0); j++)
          if (Q[i][j]==0){flag=1; in=i; jn=j;}
if (flag==0) return 1; // все замостили - задача решена
if ( (Q[in][jn+1]!=0) && (Q[in+1][jn]!=0) ) return 0;
     //нет решения, т.к. изолированная плитка
if  (Q[in][jn+1]==0)  // можно положить горизонтально
    {
    Q[in][jn]=n+1; Q[in][jn+1]=n+1;
    if (Set(n+1)) return 1;      // успешное решение
    Q[in][jn]=0; Q[in][jn+1]=0;  // возврат при неудаче
    }
if  (Q[in+1][jn]==0)   // можно положить вертикально
    {
    Q[in][jn]=n+1; Q[in+1][jn]=n+1;
    if (Set(n+1)) return 1;      // успешное решение
    Q[in][jn]=0; Q[in+1][jn]=0;  // возврат при неудаче
    }
return 0;  // нет решения
```

```
}

int main(void)
{
int i,j;
char a[MAXK+1] ; // для ввода строки занятости поля
fscanf(in,"%d %d\n",&M,&N);
for (i=0; i<M; i++)
    {
    fscanf(in,"%s\n",&a);
    for(j=0; j<N; j++)
        if (a[j]=='.')Q[i][j]=0; else Q[i][j]=-1;
    }
for (i=0; i<M; i++) Q[i][N]=-1;
for (j=0; j<N; j++) Q[M][j]=-1;  //выполнили обрамление
if (Set(0)==0) fprintf(out,"Нет решения\n");
   else {
        for (i=0; i<M; i++)
            {
            for(j=0; j<N; j++) fprintf(out,"%3d",Q[i][j]);
            fprintf(out,"\n");
            }
        }
fclose(in); fclose(out);
return 0;
}
```

Исходные данные представлены следующим образом. Первую строку исходного файла *plitka.in* занимают два числа M и N — число строк и столбцов матрицы ($M, N \leq 20$). Далее файл содержит M текстовых строк. Каждый символ строки представляет собой знак ". если поле свободно, и знак "*" при занятом поле.

Результат работы программы записывается в файл *plitka.out* и представляет собой содержимое сформированной матрицы Q.

5.7 Контрольные вопросы к разделу

1. Чем класс P отличается от класса NP?
2. Что представляет собой множество $NP \setminus P$?
3. Чем различаются задачи класса P и NP–полные задачи?
4. Приведите пример NP–полной задачи.
5. Приведите пример задачи класса P.
6. Какие способы доказательства NP–полноты Вы знаете?
7. Сформулируйте задачу о дизъюнкциях.
8. Приведите пример доказательства NP–полноты методом сводимости.
9. Как построить алгоритм решения задачи класса NP–полных задач?
10. Что называется размером задачи?
11. Как определяется временная сложность NP–полных задач?
12. Как вычислить временную сложность для заданной программы на языке Си?

13. Чему равна временная сложность задачи о рюкзаке?

14. Как вычислить рост размера решаемых задач при замене существующего компьютера на новый, скорость которого в 100 раз выше скорости старого компьютера?

15. Что означает запись формулы временной сложности: $T(n) = O(2^n)$?

16. Для решения некоторой задачи имеется две программы. Первая из них имеет временную сложность $T_1(n) = 250n^2$, вторая — сложность $T_2(n) = 3^n$. Какая программа лучше? Всегда ли является предпочтительной только одна из них?

17. Как вычисляется временная сложность фрагмента программы, содержащего цикл?

18. Как вычисляется временная сложность фрагмента программы, содержащего оператор условия?

19. Приведите пример фрагмента программы, имеющего временную сложность полинома второй степени.

20. Приведите пример функции временной сложности, для которой увеличение скорости работы компьютера не приведет к пропорциональному росту размера задачи, решаемой за некоторое фиксированное время.

5.8 Упражнения к разделу

Задание. Доказать NP–полноту поставленной задачи. Написать программу решения задачи и вычислить ее временную сложность.

5.8.1 Задача 3 – выполнимость

Дан набор $C = \{c_1, c_2, ..., c_m\}$ дизъюнкций на некотором конечном множестве логических переменных $U = \{u_1, u_2, ..., u_k\}$, причем каждая дизъюнкция построена точно из трех переменных. Вопрос: существует ли на U такой набор значений истинности, при котором выполняются все дизъюнкции из C?

Решение. Очевидно, что задача "3 – выполнимость" есть просто ограниченный вариант задачи "выполнимость в которой каждая индивидуальная задача имеет ровно три переменных в каждой дизъюнкции. Чтобы показать, что задача "3–выполнимость" является NP–полной, достаточно задачу "выполнимость" полиномиально свести к задаче "3 – выполнимость". Для этого нужно каждую произвольную дизъюнкцию свести к набору трехэлементных дизъюнкций.

Пусть имеется задача "выполнимость". Это значит, что имеется набор дизъюнкций $D = \{d_1, d_2, ..., d_l\}$ на некотором множестве логических переменных $W = \{w_1, w_2, ..., w_m\}$. Новый набор дизъюнкций будет строиться путем замены каждой отдельной дизъюнкции $w_i \in W$ эквивалентным набором дизъюнкций из трех переменных на множестве W исходных переменных и новом множестве дополнительных переменных Z.

Исходное множество D может содержать следующие дизъюнкции:

а) короткие дизъюнкции, содержание одну или две переменных;

б) дизъюнкции, содержащие ровно три переменных; эти дизъюнкции оставим без изменения;

в) длинные дизъюнкции, содержащие более трех переменных.

Рассмотрим сначала короткие дизъюнкции. Введем в множество дополнительных переменных две переменные a_1 и a_2. Вместо короткой исходной дизъюнкции d_i из

одной переменной w_j ($d_i = w_j$) в новое множество дизъюнкций включим четыре дизъюнкции:

$$d_i \vee a_1 \vee a_2,$$

$$d_i \vee \overline{a_1} \vee a_2,$$

$$d_i \vee a_1 \vee \overline{a_2},$$

$$d_i \vee \overline{a_1} \vee \overline{a_2}.$$

Очевидно, что исходная дизъюнкция $d_i = w_j$ будет истинна тогда и только тогда, когда истинны все перечисленные выше дизъюнкции. Аналогично для короткой дизъюнкции d_i, состоящей из двух переменных, в новое множество дизъюнкци включим две дизъюнкции:

$$d_i \vee a_1,$$

$$d_i \vee \overline{a_1},$$

которые истинны тогда и только тогда, когда истинна d_i.

Расмотрим теперь длинные дизъюнкции, состоящие более, чем из трех переменных. Наша задача — разбить их на более короткие дизъюнкции с использованием дополнительных переменных. Это также легко сделать, если каждой дизъюнкции

$$w_1 \vee w_2 \vee w_3 ... \vee w_{k-2} \vee w_{k-1} \vee w_k,$$

поставим в соответствие множество дизъюнкций

$$\begin{gathered} w_1 \vee w_2 \vee y_1, \\ \overline{y_1} \vee w_3 \vee y_2, \\ \overline{y_2} \vee w_4 \vee y_3, \\ \dots \\ \overline{y_{k-3}} \vee w_{k-2} \vee y_{k-2}, \\ \overline{y_{k-2}} \vee w_{k-1} \vee w_k. \end{gathered}$$

Переменные $y_1, y_2, ..., y_{k-2}$ являются уникальными для каждой исходной дизъюнкции.

Для того, чтобы убедиться, что эта сводимость полиномиальная, достаточно заметить, что число новых трехэлементных дизъюнкций ограничено полиномом степени lm, $l = |D|$, $m = |W|$. Таким образом, любую задачу "выполнимость" мы полиномиально свели к задаче "3–выполнимость". Из NP–полноты задачи "выполнимость" следует NP–полнота задачи "3-выполнимость". Таким образом, доказательство NP–полноты поставленной задачи мы выполнили. Перейдем теперь к реализации алгоритма решения этой задачи.

NP–полнота задачи означает необходимость реализации полного перебора значений переменных. Значением каждой переменной является 0 или 1. Для организации такого перебора будем использовать длинное двоичное число, содержвщее столько разрядов, сколько имеется двоичных переменных. Начиная со значения 0 этого длинного числа, будем последовательно увеличивать его на 1, перебирая тем самым все значения соответствующих логических переменных. Увеличение на 1 следует прекратить, когда все варианты значений логических переменных уже исчерпаны. Это произойдет, когда появится единица переноса в следующий ("лишний") разряд.

Каждую трехэлементную дизъюнкцию будем представять номерами трех переменных, составляющих эту дизъюнкцию. Если переменная входит в дизъюнкцию с отрицанием, номер переменной будет иметь знак минус. Исходную информацию

прочитаем из файла. Исходная информация - это число переменных, число дизъюнкций и сами дизъюнкций. Результат работы тоже запишем в файл. Результатом будут значения переменных, если задача имеет решение, или текст сообщения от отсутствии решения. Таким образом, поставленную задачу можно решить с помощью следующей программы.

```
//задача 3-выполнимость
#include <stdio.h>
#include <STDLIB.H>
#define  MAXP  1000 //максимальное число переменных
#define  MAXD  10000 // максимальное число дизъюнкций

FILE * in=fopen("input.txt","r");    // файл с исходными данными
FILE * out=fopen("output.txt","w");  // файл результатов
int  nd, np;       // исходные данные: число дизъюнкций и переменных
unsigned char  p[(MAXP+7)/8];  // значение переменных
int d[MAXD][3];                // дизъюнкции

int GetP(int ip)
// получить значение переменной с номером ip
{
int k,j,i=1; // i - маска выделения переменной из байта
j=ip/8; k=ip%8; // j - номер байта
i=i<<k;
if (p[j] & i) return 1;
return 0;
}

int Next(void)
// получить следующий набор значений переменных
// возвращает 1, если следующий набор существует (не нулевой)
{
int a, j=0; // номер очередного байта
int i=1;    // единица переполнения
do {
   a=p[j]+i;
   if(a&0xff00){ p[j]=0; i=1; }
     else      { p[j]=a; i=0; }
   j++;
   }while(i);
if (GetP(np)) return 0; // все варианты перебрали
return 1;
}

int Diz(void)
// вычислить значения всех дизъюнкций
{
int i,res,a,b,c,pa,pb,pc;
for (i=0; i<nd; i++)
    {
```

```
    a=d[i][0]; pa=GetP(abs(a)-1);  if (a<0) pa=!pa;
    b=d[i][1]; pb=GetP(abs(b)-1);  if (b<0) pb=!pb;
    c=d[i][2]; pc=GetP(abs(c)-1);  if (c<0) pc=!pc;
    res = pa || pb || pc;
    if (res==0) return 0;
    }
return 1;
}

int main(void)
{
int i,j,k,a,b,c,res;
fscanf(in,"%d %d",&np, &nd);
for (i=0; i<nd; i++)
    {
    fscanf(in,"%d %d %d",&a,&b,&c); // ввели одну дизъюнкцию
    d[i][0]=a;d[i][1]=b;d[i][2]=c;
    }
for (i=0; i<(np+7)/8; i++) p[i]=0; // обнулили все переменные
while (!(res=Diz()) && Next());
if (res)  for (i=0; i<np; i++) fprintf(out,"%d ",GetP(i));
else fprintf(out,"Нет решения\n");
fclose(in);
fclose(out);
return 0;
}
```

Перейдем теперь к оценке временной сложности. Введем сначала размер задачи. В задаче имеется две переменных, являющихся претендентами на роль размера задачи — число переменных и число дизъюнкций. Из этих двух переменных число дизъюнкций является вторичным по отношению к числу переменных, т.к. именно от их количества зависит и число дизъюнкций, и время решения задачи. Поэтому в качестве размера выберем число переменных.

Обозначим n — количество переменных, m — количество дизъюнкций. Тогда время работы программы равно

$$T(n) = a + bm + dn + c \cdot f(n),$$

где a — фиксированное время, затраченное на ввод двух переменных; $bm + dn$ — суммарное время ввода m дизъюнкций и вывода n значений переменных; $f(n)$ – количество различных вариантов значений набора логических переменных; c —- фиксированное время проверки значения одной трехэлементной дизъюнкции. Число вариантов $f(n) = 2^n$, тогда

$$T(n) = a + bm + dn + c \cdot f(n) = a + bm + dn + c2^n = O(2^n).$$

В программе значение n ограничено числом разрядов двоичного целого числа:

```
#define MAXP 1000
unsigned char p[(MAXP+7)/8]; .
```

Однако, порядок временной сложности показывает, что при больших размерах n задача не будет решена за разумное время. Поэтому предложенная выше реализация преследует лишь иллюстративные цели, демонстрируя программистский прием перечисления всевозможных значений переменных. Для практических целей решение данной задачи вполне может быть основано на использовании двоичных разрядов целой переменной, т.к. именно граница размерности 15 позволяет решать задачу за приемлемое время. Поэтому вместо массива $p[(MAXP+7)/8]$ достаточно использовать переменную типа int.

```
unsigned int p; // глобальная переменная
          ...
   // по всем 16 - ти разрядам:
for (p=0; (p<=65535) && (!res)) res=Diz(); // вызов в main()
```

Очевидно, что в программе лучше вычислить значение соответствующей верхней границы изменения переменной $p = 2^n - 1$, а не оставлять ее в виде константы 65535, как это сделано в приведеннойм фрагменте. Приведем текст итоговой программы.

```
//задача 3-выполнимость
#include <stdio.h>
#include <STDLIB.H>

#define  MAXP  1000 //максимальное число переменных
#define  MAXD  10000 // максимальное число дизъюнкций

FILE * in=fopen("input.txt","r");    // файл с исходными данными
FILE * out=fopen("output.txt","w");  // файл результатов
int  nd, np;       // исходные данные: число дизъюнкций и переменных
unsigned int p;    // значение переменных
int d[MAXD][3];    // дизъюнкции

int GetP(int ip)
// получить значение переменной с номером ip
{
if (p & (1<<ip) ) return 1;
return 0;
}

int Diz(void)
// вычислить значения всех дизъюнкций
{
int i,res,a,b,c,pa,pb,pc;
for (i=0; i<nd; i++)
    {
    a=d[i][0]; pa=GetP(abs(a)-1);  if (a<0) pa=!pa;
    b=d[i][1]; pb=GetP(abs(b)-1);  if (b<0) pb=!pb;
    c=d[i][2]; pc=GetP(abs(c)-1);  if (c<0) pc=!pc;
    res = pa || pb || pc;
    if (res==0) return 0;
    }
return 1;
```

```
}

int main(void)
{
int i,a,b,c,res=0;
fscanf(in,"%d %d",&np, &nd);
for (i=0; i<nd; i++)
    {
    fscanf(in,"%d %d %d",&a,&b,&c); // ввели одну дизъюнкцию
    d[i][0]=a;d[i][1]=b;d[i][2]=c;
    }
long Max_p=1; for (i=0; i<nd-1; i++) Max_p*=2;
Max_p--;
for (p=0; (!res)&&(p<Max_p); p++) if(res=Diz())
     for (i=0; i<np; i++) fprintf(out,"%d ",GetP(i));
if(!res) fprintf(out,"Нет решения\n");
fclose(in);
fclose(out);
return 0;
}
```

5.8.2 Варианты заданий

Задание: доказать NP–полноту поставленной задачи. Написать программу решения задачи и вычислить временную сложность построенной программы. Определить порядок временной сложности. В качестве вариантов заданий предлагается рассмотреть задачи 1 — 18 из параграфа 5.5.

5.9 Тесты для самоконтроля к разделу

1. Дайте понятие NP–полных задач.

Варианты ответов.

а) NP–полные задачи — это задачи, для решения которых существует недетерминированный алгоритм, решающий ее за полиномиальное время;

б) NP–полные задачи — это задачи, для решения которых существует недетерминированный алгоритм, решающий ее за экспоненциальное время;

в) NP–полные задачи — это задачи, для решения которых не существует детерминированного алгоритма, решающего ее за полиномиальное время;

г) NP–полные задачи — это задачи, для решения которых не существует недетерминированного алгоритма, решающего ее за полиномиальное время;

д) NP–полные задачи — это задачи, для решения которых существует детерминированный алгоритм, решающий ее за полиномиальное время;

Правильный ответ: в.

2. Даны два алгоритма A_1 и A_2. Временная сложность $T_1(n)$ алгоритма A_1 равна $O(n)$, а временная сложность $T_2(n)$ алгоритма A_2 равна $(O(2^n))$. Экспериментально замерили время работы в секундах соответствующих программ. Результаты измерений свели в таблицу:

сложность	$n = 2$	$n = 4$
$T_1(n)$	128	256
$T_2(n)$	8	32

Какое утверждение про временные характеристики этих алгоритмов справедливо?

Варианты ответов.

а) Алгоритм A_1 всегда лучше алгоритма A_2.

б) Алгоритм A_2 всегда лучше алгоритма A_1.

в) Алгоритм A_1 работает быстрее алгоритма A_2, начиная с $n = 8$.

г) Алгоритм A_1 работает быстрее алгоритма A_2, начиная с $n = 9$.

Правильный ответ: г.

3. Сформулировано несколько утверждений.

а) Задача о дизъюнкциях состоит в том, чтобы для заданного набора дизъюнкций на множестве логических переменных найти такой набор значений этих переменных, на котором все дизъюнкции истинны.

б) Задача о дизъюнкциях имеет полиномиальную временную сложность.

в) Любая NP–полная задача эквивалентна задаче о дизъюнкциях.

г) При решении задачи о дизъюнкциях необходимо реализовать в программе полный перебор вариантов значений истинности логических переменных.

Какое из указанных утверждений ложно?

Правильный ответ: б.

4. Что называется размером задачи? Укажите наиболее правильный ответ из предложенных.

Варианты ответов.

1) Размер задачи — это характеристика объема исходных данных.

2) Размер задачи — величина программы, решающей эту задачу.

3) Размер задачи — время работы алгоритма решения задачи.

4) Размер задачи — это характеристика объема памяти, занимаемой всеми данными в программе, решающей задачу.

5) Размер задачи — эта величина в байтах выполняемого модуля *.exe, предназначенного для решения задачи.

Правильный ответ: 1.

5. Какая из перечисленных ниже задач не является NP–полной:

а) задача о раскрашиваемости графа;

б) задача о трехмерном сочетании;

в) задача поиска Гамильтонова цикла в произвольном графе;

г) задача поиска Эйлерова цикла в произвольном графе;

д) задача о дизъюнкциях.

Правильный ответ: г.

Глава 6

ПОСТРОЕНИЕ И АНАЛИЗ ЭФФЕКТИВНЫХ АЛГОРИТМОВ

6.1 Типы рекурсивных алгоритмов

Как было показано в предыдущем разделе, с теоретической точки зрения рекурсивные определения являются основой всей современной теории вычислимых функций. При разработке алгоритма решения задачи часто именно рекурсивные алгоритмы являются наиболее простыми и естественными в понимании. Рассмотрим случаи, в которых разработка рекурсивных алгоритмов является наиболее эффективной. Обычно рекурсивный алгоритм целесообразно разрабатывать при наличии одного из следующих условий.

Во–первых, при необходимости обработки данных, имеющих рекурсивную структуру. Процедуры анализа рекурсивных структур наиболее эффективны, когда они сами рекурсивны, т.к. эти процедуры отражают особенности построения данных и в результате строение программы соответствует структуре обрабатываемых данных.

Во–вторых, если алгоритм, обрабатывающий набор некоторых данных, можно построить, разбивая эти данные на части и получая из этих частичных решений общее решение на всей совокупности данных. Этот прием, особенно если применять его рекурсивно, часто приводит к эффективному решению задачи, подзадачи которой представляют собой ее меньшие версии. Данный прием получил название "разделяй и властвуй". При этом, как правило, задачу следует разбивать на подзадачи равных размеров. Поддержание равновесия — основной принцип при разработке хорошего алгоритма.

В–третьих, если задача поставлена так, что ее решением является выбор какого–то варианта из некоторого множества возможных решений. Решение задачи определяется после некоторого конечного числа шагов так, что выбирая на каждом шаге вариант решения, мы удаляем часть информации из всей подлежащей обработке информации и пытаемся решить задачу на меньшем объеме данных. Поиск решения завершается в двух случаях: либо когда кончатся данные, либо когда находится решение на текущем наборе данных. В частности, таким методом обычно решаются NP-полные задачи.

В–четвертых, если имеется рекурсивная схема некоторой функции. Существуют некоторые функции, которые легко можно определить рекурсивно, но которые нельзя определить в терминах обычных алгебраических выражений. Примером такой функции является функция Аккермана, при попытке определить которую алгебраически можно получить только последовательность экспонент, записанную через многоточие.

Если алгоритм решения задачи представим какой–либо рекурсивной схемой, то программа полностью эквивалентна этой схеме. Например, функция Аккермана легко вычисляется следующим образом:

```
long int B (long int n, long int x)
{
if (n == 0) return 2 + x;
if (x == 0) return sg(n);
return  B(n-1,B(n,x-1)) ;
}
```

Рассмотрим примеры решения задач различных типов.

Пример 1.

Выражение представлено бинарным деревом, узлами которого являются знаки бинарных операций, а листьями — целые числа. Найти значение заданного выражения. Пример выражения, представленного бинарным деревом, приведен на рис. 6.1.

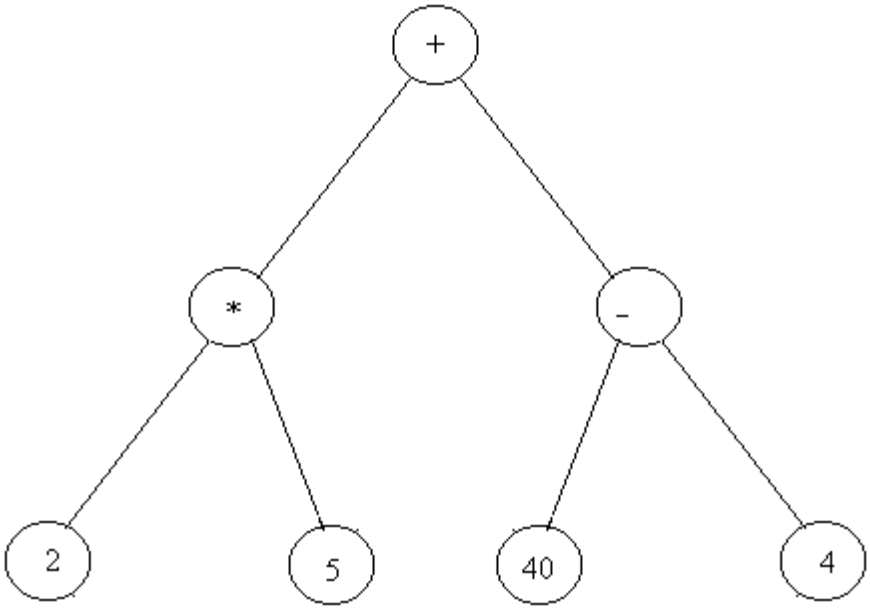


Рис. 6.1: Представление арифметического выражения $2*5+(40-4)$ в виде бинарного дерева.

Рекурсивное определение дерева можно выполнить следующим образом:

```
struct TTree
        {
        TTree* Left; TTree *Right;
        union {int Data; char Oper;};
        };
```

Тогда вычисление выражения соответствует обходу дерева в концевом порядке:

```
int Calc (TTree * T)
{
if (T->Left==NULL)  return T->Data; //лист
  // выполняем операцию над поддеревьями:
switch ( T->Oper )
   {
   case '+': return Calc(T->Left)+Calc(T->Right);
             break;
   case '-': return Calc(T->Left)-Calc(T->Right);
             break;
   case '*': return Calc(T->Left)*Calc(T->Right);
             break;
   case '/': return Calc(T->Left)/Calc(T->Right);
             break;
   }
}
```

Пример 2.

Дано множество S, содержащее n целых чисел ($n > 2$). Найти минимальный элемент в этом множестве.

Для простоты будем считать, что n есть степень числа 2. Применяя метод "разделяй и властвуй" разобьем множество S на два подмножества из $n/2$ элементов в каждом. Тогда достаточно найти минимальный элемент в каждом из полученных подмножеств и выбрать минимальное число из этих двух полученных:

```
int MinEl(Vector S, int i, int n)
//  i - начальный  элемент;  n - число элементов
{
int ndiv2;
ndiv2=n /2;
if (n==2) then return min(S[i],S[i+1])
else return min( MinEl(S,i,ndiv2), MinEl(S,i+ndiv2,ndiv2) );
}
```

Этот метод деления заданного множества на две равные части широко применяется для сокращения числа попарных сравнений. Например, алгоритм сортировки слиянием построен на указанном принципе.

Пример 3.

Рассмотрим задачу поиска выхода из лабиринта. Представим лабиринт графом, тогда задача сводится к поиску пути от заданной вершины A к заданной вершине B. Если $B = A$, то выход найден, в противном случае требуется рассмотреть все дуги AC из вершины A и попытаться найти выход из C. При этом необходимо предотвратить обратный ход в A из C, для чего следует временно убрать дугу AC.

Обычно процедуры такого типа имеют дополнительный параметр — флаг успешного решения задачи, который анализируется при возврате из рекурсии. Будем задавать граф матрицей смежности вершин. Решение можно получить с помощью следующей процедуры поиска маршрута от начальной к целевой вершине:

```
int a[MAXK][MAXK];    // матрица смежности
int n; // число вершин
int first, last;  // начало и конец в лабиринте
```

```
int Labirint(int first)
// first - текущая вершина
{
if (first==last)
   { printf("Последняя вершина пути %d \n",first); return 1;}
int i;
for (i=0; i<n; i++)
    if (a[first][i])
       {
       a[first][i]=0;   a[i][first]=0;
       if (Labirint(i))
          {printf("Маршрут от %d до %d\n",first,i); return 1;}
       }
return 0;
}
```

Следует отметить, что при использовании рекурсивных процедур нужно стараться по возможности уменьшать число формальных параметров, т.к. все они заносятся в стек при каждом новом вызове. В частности, рассмотренная процедура *Labirint* в качестве формального параметра имеет только значение исходной текущей позиции в графе. Значения n, *last* и матрица a являются глобальными данными. Обратите внимание на операторы, стоящие перед рекурсивным вызовом функции

```
a[first][i]=0;   a[i][first]=0;
```

Их назначение — предотвратить бесконечный обход одних и тех же вершин. Чаще всего при реализации полного перебора вариантов решения задачи используется правило полного восстановления измененных данных после возврата из рекурсивного вызова, который не дал искомого решения. Восстановление данных, которые были изменены перед рекурсивным вызовом, дает гарантию, что будет обеспечен полный перебор всех вариантов поиска решения. В данном примере эти данные также могли быть восстановлены. Однако, восстановление измененных значений оператором

```
else  { a[first][i]=1; a[i][first]=1; }
```

непосредственно после выхода из $Labirint(i)$ привело бы к лишним многократным попыткам пройти через вершины, путь через которые уже был проверен на предшествующих рекурсивных обходах. Таким образом, в данной задаче такое восстановление вредно, так как при этом увеличивается время работы программы.

Пример 4.

Дано множество последовательных целых чисел от 0 до $k = 2^N - 1$. Построить эти числа в такую упорядоченную последовательность $a[0]$, $a[1]$, ..., $a[k]$, чтобы в двоичном представлении числа $a[i]$ и $a[i+1]$ различались только в одном разряде.

Для решения задачи предложим схему построения указанной последовательности, основанную на рекурсии по j-ой компоненте двоичного представления некоторого числа. Чтобы рассмотреть все наборы длины j, зафиксируем нулевое значение j-ого разряда числа и перечислим все наборы длины $j-1$ для оставшихся младших разрядов. После этого сменим значение j-го разряда на 1 и снова переберем все наборы длины $j-1$, но уже в обратном порядке. Естественно, эту же схему применим и для наборов меньшей длины. Рекурсивный спуск продолжим до тех пор, пока не

получим нулевое значение переменной j. При j равном 0 строится пустое множество. Указанный алгоритм представим следующей рекурсивной схемой:

$$f(0) = \emptyset,$$
$$f(j+1) = 0 * f(j) \cup 1 * f_{back(j)}.$$

Здесь функция $f(j)$ имеет своим значением множество упорядоченных наборов двоичных последовательностей длины j, отличающихся в одном разряде, $f_{back(j)}$ — те же наборы в обратном порядке. Знак "*" обозначает операцию конкатенации, $\cup$ — объединение множеств. Начальное условие $f(0) = \emptyset$ означает, что при нулевом количестве разрядов в числе получаем пустое множество последовательностей. В указанной зависимости предполагается, что операция конкатенации "*" пустого множества $\emptyset$ и произвольного множества M дает в результате множество M.

Полученная рекурсивная схема легко реализуется с помощью программы, в которой содержится рекурсивная функция вычисления $f(j)$.

```
#include <stdio.h>
#include <STDLIB.H>
#define NMAX 10000
long int   k, a[NMAX]; // кол-во чисел и список чисел в множестве

void f(int j)
{
if (j==0) { k=0; return; }// не добавляем ничего к последовательности
if (j==1) { k=2; a[0]=0; a[1]=1; return; }
long int b; //текущее значение разряда
f(j-1);     // построили  {0}*f(j-1)
b=1<<(j-1); // 1 в текущем разряде
for (int i=0; i<k; i++)
    a[k+i]=b | a[k-i-1];  // добавили {1}*f_инвер(j-1)

k*=2;
}

void main(void)
{
int n;
printf("Введите число разрядов:  ");
scanf("%d",&n);
k=0;
f(n);
for (int i = 0; i<k; i++)    printf("%ld  = 0x%lx\n",a[i],a[i]);
   // вывели числа в 10-ой и 16-ой системах счисления
}
```

Наличие схемы примитивной рекурсии позволяет сделать вывод о возможности построения аналогичного итерационного алгоритма, содержащего простой цикл вместо рекурсивного вызова. Для этой цели будем использовать переменную l, в качестве параметра цикла изменяющуюся от 2 до j (2 - это начальное значение для рекурсивного вызова функции). Получим следующую нерекурсивную программу.

```
#include <stdio.h>
#include <STDLIB.H>
#define NMAX 10000

long int   k, a[NMAX];
    // кол-во чисел и список чисел в результирующем множестве

void f(int j)
{
if (j==0) { k=0; return; } // последовательность остается пустой
k=2; a[0]=0; a[1]=1;
   // выполнили начальные действия вне рекурсивного спуска
if (j==1)  return;
long int b; //текущее значение разряда
int l;
for (l=2; l<=j; l++)
     // 2 - начальное значение для рекурсивного вызова функции
    {
    b=1<<(l-1); // 1 в текущем разряде
    for (int i=0; i<k; i++)
       a[k+i]=b | a[k-i-1];  // добавили {1}*f_инвер(j-1)
    k*=2;
    }
}

void main(void)
{
int n;
printf("Введите число разрядов:  "); scanf("%d",&n);
k=0;
f(n);
for (int i = 0; i<k; i++) printf("%ld  = 0x%lx\n",a[i],a[i]);
   // вывели числа в 10-ой и 16-ой системах счисления
}
```

Пример 5.

В заключение рассмотрим известную задачу о Ханойских башнях – пример простого и эффективного способа разработки алгоритма решения задачи, которую весьма трудно было бы решить без применения рекурсии.

Даны три стержня и n дисков разного размера. Диски можно надевать на стержни, строя таким образом "башни". Пусть вначале диски находятся на стержне A в порядке убывания размера, как показано на рис. 6.2 для $n = 3$. Нужно переместить все диски на стержень C так, чтобы они остались в том же порядке. Этого нужно добиться, соблюдая следующие правила:

а) на каждом шаге ровно один диск перемещается с одного стержня на другой;

б) диск большего размера нельзя перемещать на меньший;

в) стержень B можно использовать в качестве промежуточного.

Задача легко решается, если алгоритм представить в виде рекурсивной процедуры, которая для перемещения n дисков со стрежня A на стержень C выполняет следующие действия:

Рис. 6.2: Ханойские башни при $n = 3$.

1) переместить $n-1$ дисков с A на B, используя C в качестве вспомогательного;
2) переместить последний самый большой n-ый диск на стержень C;
3) переместить $n-1$ дисков с B на C, используя A в качестве вспомогательного.

Осталось определить действия в начальной точке. Такой начальной точкой является $n = 0$. Если число дисков равно нулю, то никакой работы производить не требуется. Все указанные действия выполняет следующая программа.

```
#include <stdio.h>
void hanoi( int n, int a, int b, int c)  //сколько, откуда, куда, через
{
if(n==0)  return;
else
   {
   hanoi(n-1,a,c,b);
   printf("\n переложить кольцо %d с %d на %d", n,a,b);
   hanoi(n-1,c,b,a);
   }
}

void main(void)
{
int n;
printf("Введите n=");  scanf("%d",&n);
hanoi(n,1,2,3);
printf("\nЗадача решена!");
}
```

Подводя итог всему вышесказанному рассмотрим достаточно сложную задачу, иллюстрирующую методику применения рекурсивных данных для уменьшения временной сложности алгоритма.

Пример 6.

Дана матрица из маленьких букв латинского алфавита. Матрица имеет размеры $N \times N$, $N \leq 20$. Слово — это цепочка из букв, принадлежащих клеткам матрицы, причем такая цепочка получается в результате обхода матрицы. Обход может начинаться из произвольной клетки, а каждая последующая клетка является смежной с

текущей клеткой по диагонали, вертикали или горизонтали. Буквы в слове, которе может быть получено в процессе обхода матрицы, должны быть расположены только в порядке возрастания в соответсвии с алфавитным порядком.

Требуется перечислить все слова, состоящие не менее, чем из трех букв, которые можно сформировать при обходе заданной матрицы. Слова необходимо упорядочить по длине, а слова одной длины должны быть упорядочены в лексикографическом порядке. Каждое слово должно быть выдано только один раз.

Исходная информация — значение N и буквенная матрица, заданная по строкам. Результат — перечисление всех формируемых слов. Например, для входной информации

3
one
top
dog

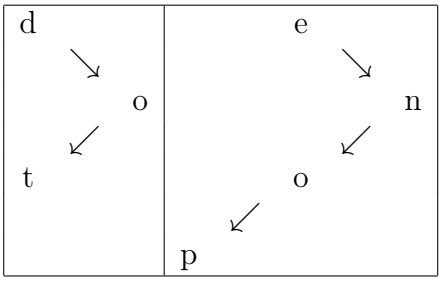
рузультирующий список слов равен dop, dot, eno, ent, eop, eot, gop, got, nop, not, enop, enot.

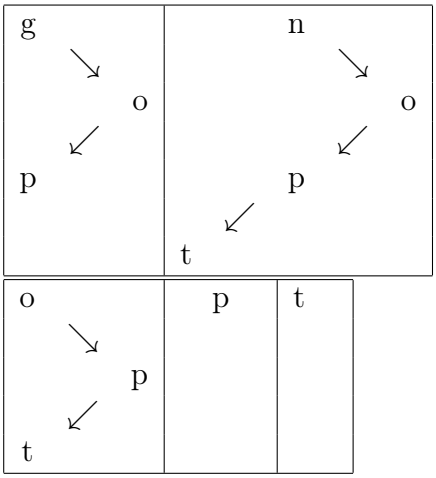
Казалось бы, простейший способ решения задачи заключается в следующем. В качестве исходной точки последовательно будем перебирать клетки матрицы, формируя в результате последовательного рекурсивного обхода матрицы в восьми допустимых направлениях новые слова. Каждое сформированное слово будем заносить в таблицу слов, если в ней сформированное слово отсутствует. Однако оценка порядка временной сложности такого алгоритма показывает его принципиальную непригодность.

Действительно, максимальная длина слова, которое можно сформировать по указанным в задаче правилам, равна 26 — это слово $abcde...xyz$. Такое слово — единственное. Слов длины 25 можно сформировать 26 штук, вычеркивая из максимального слова по одной букве. Слов длины 24 уже 26^2 и т.д. Самых коротких трехбуквенных слов можно сформировать 26^{23}. Это значит, что при создании нового слова его нельзя сравнивать с таблицей уже имеющихся слов — на это просто не хватит ни времени, ни памяти компьютера.

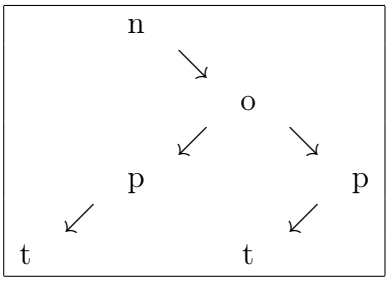
Известно, что проблема лексикографического упорядочивания хорошо решается с помощью бинарных деревьев. В каждой вершине такого дерева будем хранить одну букву. Правое поддерево соответствует "потомкам т.е. буквам, которые могут находиться в цепочке за корневой буквой. Левое поддерево соответствует "соседям т.е. буквам, которые равноправны корневой букве.

Построим сначала по матрице бинарные деревья, которые соответствуют только непосредственным соседям. Например, для указанной выше матрицы существуют два дерева с пустыми потомками для символов p и t и следующие пять деревьев для символов d, e, g, n, o:





Построенные деревья представляют только последовательные пары символов. Чтобы построить деревья, соответствующие цепочкам, надо сформировать для каждого символа новое дерево путей, последовательно присоединяя к вершинам с пустыми правыми связями имеющиеся деревья парных символов. Например, для символа n получится дерево



Постройте самостоятельно дерево для символа e, чтобы убедиться в необходимости рекурсивного присоединения поддеревьев к дереву путей в процессе его построения. Как алгоритм формирования дерева, так и алгоритм вывода всех слов при обходе дерева путей являются рекурсивными.

При таком присоединении теряется информация о наличии последовательного пути в матрице, т.к. одна вершина дерева пар может соответствовать одной и той же букве, расположенной в разных частях матрицы. Чтобы восстановить эту информацию в дереве путей, каждой вершине этого дерева необходимо поставить в соответствие набор индексов матрицы, и либо на этапе формирования дерева путей, либо на этапе его обхода учесть правило преобразования списка индексов текущей вершины в список допустимых индексов ее правого потомка. Ниже приведена программа формирования дерева путей, в которой не выполняется соответствующий контроль. Дополнить программу контролем на допустимые индексы предлагается в качестве упражнения. При формировании такого списка необходимо учесть временную и емкостную сложность контроля.

```
// цепочки из букв на квадрате N*N
#include <stdio.h>
#include <STDLIB.H>

#define  MAXK  20      // размер матрицы
#define  MaxBukva  26  // число букв в алфавите
#define  index(s)   (int)(s)-(int)('a')
```

```
struct TTree {
             char b;           // индекс буквы
             int left,  right; // индексы поддеревьев
             };
int uk[MaxBukva];         // указатель на деревья для букв
TTree a [MaxBukva*(MaxBukva-1)/2];
                  // двухуровневые деревья для букв из матрицы
int first; // указатель на первый свободный элемент в дереве "a"

TTree b [MaxBukva*(MaxBukva-1)*MAXK/2];  // результирующие деревья
int first_b;   // указатель на первый свободный элемент в дереве "b"

char matr[MAXK+3][MAXK+3]; // матрица из слов - по краям нули
int n;                     // фактический размер матрицы


FILE *in = fopen("input.txt","r");
FILE *out= fopen("output.txt","w");

void GetData(void)
{
fscanf(in,"%d\n",&n);
int i,j; char s;
for(i=0; i<MAXK*MAXK; i++) {a[i].left=-1; a[i].right=-1;}
for(i=0; i<MaxBukva; i++) uk[i]=-1;
for(i=0; i<n+2; i++)     for(j=0; j<n+2; j++)    matr[i][j]='\1';
for(i=0; i<MaxBukva; i++)
   {
   a[i].b='\0';   a[i].left =-1;   a[i].right=-1; //нет потомков
   }
first=0;
for(i=0; i<n; i++)
    {
    for(j=0; j<n; j++)
       {
       fscanf(in,"%c",&s);  matr[i+1][j+1]=s;
       }
    while ((s!='\n') && (!feof(in)) ) fscanf(in,"%c",&s);
    }
}

void   SetNewRight(int  t, int next)
{
if (uk[t]==-1)
   {  // пока еще нет потомков справа
   uk[t]=first;
   a[first].b=next; a[first].left =-1; a[first].right=-1;
   first++;
   return;
   }
```

```
if ( a[uk[t]].b == next ) return; // равные не вставляем
if ( a[uk[t]].b > next  )
   {  // надо вставить первым в списке потомков
   a[first].b=next; a[first].right=-1; a[first].left=uk[t];
   uk[t]=first;
   first++;
   return;
   }
// ищем место вставки:
int tek=uk[t];
while ( (a[a[tek].left].b < next)  && (a[tek].left!=-1) )
      tek=a[tek].left;
if (next==a[ a[tek].left].b) return; //равные игнорируем
// вставляем после tek:
a[first].b=next; a[first].right=-1;
a[first].left=a[tek].left; a[tek].left=first;
first++;
return;
}

void Construct(void)
// построить один уровень потомков по соседям matr
{
int i, j, k;
     // приращения индексов для соседа:
int di[]= { 0,  0, 1, -1,  1,  1, -1, -1 },
    dj[]= { 1, -1, 0,  0, -1,  1, -1,  1 };
for(i=1; i<n+1; i++)
    for(j=1; j<n+1; j++)
       for(k=0; k<8; k++)
          if( matr[i][j] < matr[i+di[k]][j+dj[k]] )
             SetNewRight(index(matr[i][j]),
                         index(matr[i+di[k]][j+dj[k]]) );
}

void PrintTree(int t, // вершина дерева
               int i, // текущая длина
               int len, // какой длины слова нужно печатать
               char *res ) // формируемое слово
{
if (t==-1) return;  // нет потомков
int tek=t;
do {
   res[i]=(char)(b[tek].b+'a'); res[i+1]='\0';
   if (i+1==len) fprintf(out,"%s\n",res);
   if((i+1<len) && (b[tek].right!=-1))
                PrintTree(b[tek].right, i+1, len, res);
   tek=b[tek].left;
   } while (tek!=-1);
}
```

```
void CopyTree(int t)
// приписать копию дерева из "a" к листу t дерева "b"
{
if (uk[b[t].b]==-1) return;
b[t].right=first_b; // копируется исходная цепь в конец b
int l=uk[b[t].b], i,j;
for ( j=first_b; l!=-1; j++)
   {
   b[j].b=a[l].b;
   b[j].right=-1;
   if (a[l].left!=-1) b[j].left=first_b+1;
      else b[j].left=-1;
   l = a[l].left;
   first_b++;
   }
}

void Form(int t)
// доформирование дерева в соответствии с его высотой
// t - вершина в массиве b, от которой идет доформирование
{
int  tek=t;
while(tek!=-1)
      {
      if ( (b[tek].right==-1)&& (uk[b[tek].b]!=-1) )
           { CopyTree(tek);  Form(b[tek].right); }
           else if (b[tek].right!=-1) Form(b[tek].right);
      tek=b[tek].left;
      }
}

int main(void)
{
int i,j,len; char res[MaxBukva+2];
GetData();
Construct();
for (i=0; i<first; i++)
    {
    b[i].left=a[i].left;
    b[i].right=a[i].right;
    b[i].b=a[i].b;
    }
first_b=first; // получили копию двухуровневого дерева в "b"

for(j=0; j<MaxBukva; j++)
         Form(uk[j]); // достроили дерево  вглубь
for (len=3; len<=MaxBukva; len++)
    for (i=0; i<MaxBukva; i++)
        {
```

```
        res[0]=(char)('a'+i); res[1]='\0';
        PrintTree(uk[i], 1, len, res);
        }
fclose(in); fclose(out);
return 0;
}
```

6.2 Устранение рекурсии

Как правило, разработка рекурсивного алгоритма в тех случаях, когда это имеет смысл, существенно ускоряет процесс программирования и приводит к простой и ясной структуре программы. К сожалению, большая глубина рекурсивных вызовов может привести к невозможности работы программы. В таких случаях возникает задача преобразования рекурсивной программы в нерекурсивную. Однако, известно, что не существует общего алгоритма преобразования произвольного рекурсивного алгоритма в нерекурсивный без использования моделирования рекурсивных вызовов с помощью стека. Известно лишь, что это можно просто сделать для примитивно–рекурсивных схем и достаточно просто для схем, сводящихся к примитивной рекурсии.

Рассмотрим сначала *схему возвратной рекурсии*. Пусть $g_1(x), \ldots, g_k(x)$ — всюду определенные функции, удовлетворяющие для всех значений x условиям

$$g_i(x) < x \quad (i = 1, \ldots, k).$$

Функция $f(\overline{x}, y)$ получается возвратной рекурсией из функций

$$w(\overline{x}),\ h(\overline{x}, y, z_1, \ldots, z_k)$$

и вспомогательных функций

$$g_1(x), \ldots, g_k(x), \forall_i \forall_x g_i(x) < x,$$

если

$$\begin{aligned} f(\overline{x}, 0) = &\quad w(\overline{x}) \\ f(\overline{x}, y+1) = &\quad h(\overline{x}, y, f(\overline{x}, g_1(y+1)), \ldots, f(\overline{x}, g_k(y+1)). \end{aligned}$$

Иными словами, значение функции f в точке $\langle \overline{x}, y+1 \rangle$ вычисляется рекурсивно через значения этой функции в k предыдущих точках $\langle \overline{x}, g_1(y+1) \rangle$, $\ldots$, $\langle \overline{x}, g_k(y+1) \rangle$, не обязательно отстоящих от этой текущей точки на единицу.

Возвратная рекурсия теоретически сводится к примитивной, однако, в общем случае реализация соответствующей нерекурсивной программы вызывает затруднения. В частных случаях, особенно при $k = 1$, перевод рекурсии в итерацию осуществляется просто. Достаточно определить значение в начальной точке, а затем увеличить аргумент y по обратному закону

$$g^{-1}(y).$$

Рассмотрим, например, алгоритм возведения числа в натуральную степень:

$$\begin{aligned} a^0 = &\quad 1, \\ a^{2n} = &\quad (a^n)^2, \\ a^{2n+1} = &\quad (a^n)^2 * a. \end{aligned}$$

Этой схеме соответствует программа:

```
long int deg(long int a,int n)
{
long int x;
if (n==0) return 1;
x=deg(a,n/2);
if (n%2==0) return x*x;
    else return x*x*a;
}
```

Очевидно, что на каждом уровне этой рекурсивной схемы анализируется соответствующий разряд в двоичном представлении показателя степени. Причем в рекурсивном алгоритме анализ выполняется начиная с младшего разряда, но выполнение умножений возможно только после выполнения вычислений на предыдущем уровне. Спуск продолжается до тех пор, пока не будет достигнут старший разряд числа. Тогда нерекурсивная программа имеет вид:

```
#define FirstSign  0x1000
       // старшая анализаруемая единица степени

long int deg(long int a,int n)
{
long int x;
if (n==0) return 1;
// вычислим положение единицы в FirstSign
int i=FirstSign;
while ( (i & n) ==0x0) {i=i>>1;} // узнали номер старшей позициив степени
i=i>>1; x=a;
while  (i) // не закончилась степень n
      {
      if ( (n & i) == 0x0) x=x*x;
         else x=x*x*a;
      i=i>>1;
      }
return x;
}
```

В общем случае к каждому алгоритму надо подходить индивидуально, моделируя те действия, которые выполняет рекурсивная программа, полученная в результате трансляции. Это означает, что в нерекурсивном эквиваленте рекурсивного алгоритма необходимо описать *магазин*, каждый элемент которого содержит:

1) данные, являющиеся исходной информацией для рекурсивной процедуры;

2) внутренние (локальные) данные рекурсивной процедуры, если они есть.

Каждое обращение к рекурсивной процедуре в нерекурсивной программе соответствует занесению информации в магазин. Каждый выход из рекурсии соответствует стиранию информации из магазина. Общая структура нерекурсивной программы соответствует циклу типа "while который выполняетмя при условии, что магазин не пуст.

Рекурсивный алгоритм всегда реализуется с помощью вызывающей программы, в которой происходит обращение к рекурсивной процедуре с начальными условиями. В нерекурсивном алгоритме этот цикл соответствует рекурсивному обращению к

себе, а подготовка цикла к действиям — занесению в магазин начальных данных с которыми обращалась программа к рекурсивной процедуре.

Пример.

Пусть имеется рекурсивная функция обхода дерева:

```
void  Obhod (TTree * T) ;
//   Корневой обход дерева
{
if (T==NULL) return;
printf ("...",T->Data);
Obhod (T->Left);
Obhod (T->Right);
}
```

Отсутствие дополнительных действий между или после двух последовательных рекурсивных вызовов функции $Obhod(T->Left)$ и $Obhod(T->Right)$ позволяет хранить в магазине только ссылки на обрабатываемые вершины (и не заводить специальные признаки, показывающие, что нужно делать с указанными ссылками).

```
#define MAX_MAG 100
typedef TTree TMag [MAX_MAG];
TMag     Mag;     // магазин
int uk=0;         // указатель  верхушки магазина
TTree  h;         // временная  вершина
Mag[ uk ]= Root;  // соответствует  вызову  Obhod(Root)
while ( uk >= 0 )
    {
    if (Mag[ uk ] == NULL) uk--;
    else   {
           printf ("...",Mag[ uk ]->Data);
           h= Mag[ uk ];
           Mag[ uk++] = h->Right;
           Mag[ uk ]  = h->Left;
           }
    }
```

Если бы алгоритм обхода был другим, например, либо $L \to R \to$ Root, либо $L \to Root \to R$, то в магазин надо было бы дополнительно заносить признак, указывающий действие, которое необходимо выполнить над вершиной из верхушки магазина. Таким действий может быть два: дальнейший обход, начиная с указанной вершины, или печать информации о вершине.

6.3 Методы отсечения

Самый прямолинейный подход при поиске решений методом полного перебора состоит в попытке перепробовать все различные ходы, пока не удастся получить решение. Такого рода попытки по своей сути связаны с поиском при помощи проб и ошибок. Отправляясь от начальной конфигурации задачи, мы можем построить

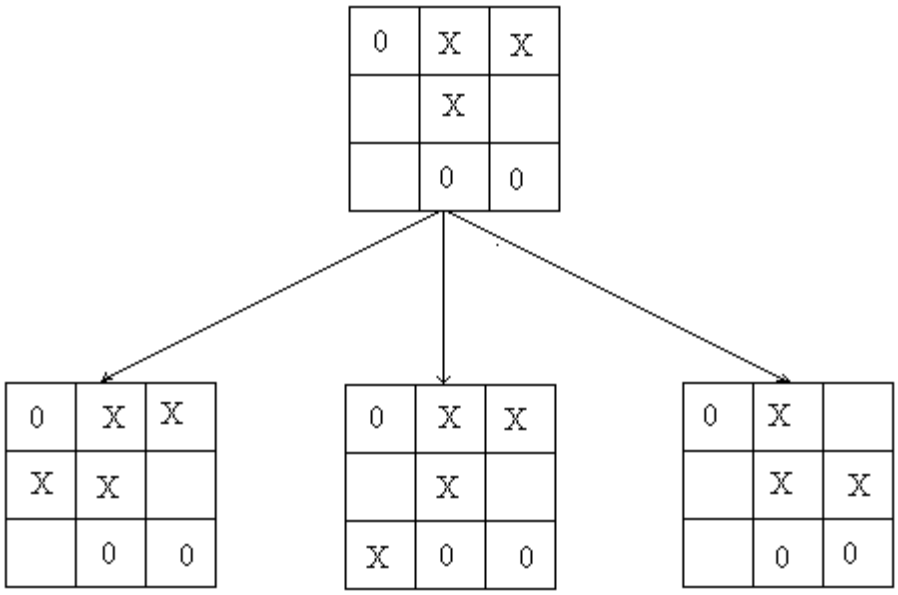


Рис. 6.3: Фрагмент дерева состояний игры КРЕСТИКИ — НОЛИКИ.

все конфигурации, возникающие в результате выполнения каждого из возможных ходов, затем построить следующее множество конфигураций после применения следующего хода и т.д., пока не будет построена целевая конфигурация. Для поиска такого рода методов поиска решения оказывается полезным введение понятия "состояния" задачи. Например, для игры в "крестики-нолики" состояние задачи — это просто конкретное расположение крестиков и ноликов в клетках таблицы. Пространство состояний этой игры, достижимых из начального состояния, состоит из всех тех конфигураций крестиков и ноликов, которые могут быть образованы по правилам записи этих символов в клетки таблицы. Пространство состояний, достижимых из данного начального состояния, полезно представлять в виде графа, вершины которого соответствуют этим состояниям, а дуги определяют один возможный шаг. Например, для игры в "крестики-нолики" один шаг перехода в новое состояние для очередного хода "крестиков" подграф переходов может имееть вид, представленный на рис. 6.3.

Многие из прикладных задач имеют чрезвычайно большие пространства состояний, поэтому методы полного перебора всех вариантов практически неработоспособны из-за временных ограничений. Один из способов ускорения поиска решения состоит в использовании оценочных функций для упорядочивания перебора вариантов. Оценочная функция должна обеспечивать возможность упорядочивания вершин — кандидатов на обработку — с тем, чтобы выделить ту вершину, которая с наибольшей вероятностью находится на лучшем пути к цели. Оценочные функции строятся на основе различных соображений и связаны с конкретной прикладной областью. Один из часто употребляемых вариантов определения функции цели основан на понятии расстояния или другой меры различия между текущей вершиной и множеством целевых вершин. Рассмотрим, например, игру в "восемь" как усеченный вариант известной игры "пятнадцать". Имеется квадратное клеточное поле размером 3 на 3 и восемь квадратных фишек с номерами от 1 до 8. Для произвольной расстановки фишек требуется найти способ их передвижения так, чтобы в результирующей позиции все фишки были упорядочены (см. рис. 6.4).

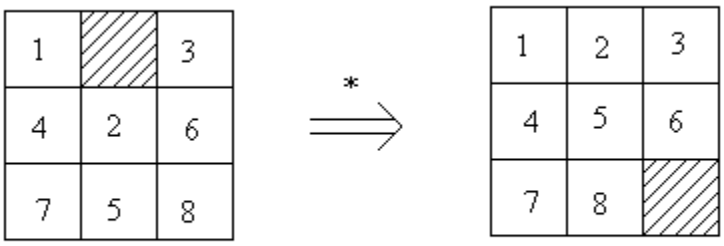


Рис. 6.4: Конечное состояние и вариант начального состояния игры в "восемь".

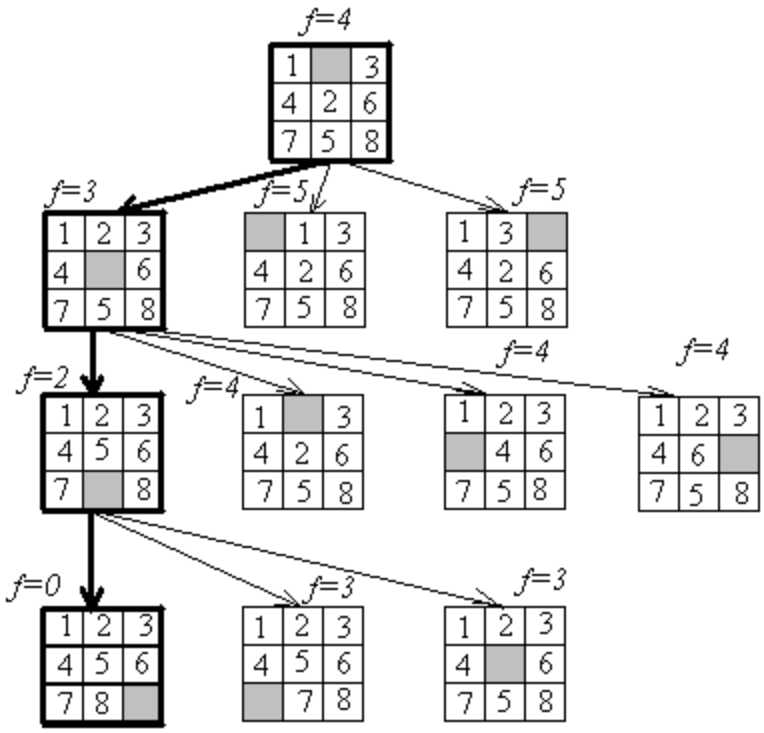


Рис. 6.5: Отсечение вариантов для игры в ВОСЕМЬ.

Выберем в качестве функции цели число позиций, на которых фишки стоят не на своих местах, и будем перебирать вершины в порядке неубывания функции цели. Тогда для начальной позиции рис. 6.4 порядок поиска решения представлен на рис. 6.5.

На каждом шаге существует не более четырех вариантов хода. Выбор лучшего из них определяется минимальным значением функции цели. На рисунке рассмотрен вариант алгоритма выбора очередного хода без контроля возврата в состояние, которое уже встречалась на пути решения. Реализовать такой контроль достаточно просто, если хранить весь путь решения задачи. Следует, однако, отметить существенные затраты времени на полный такой контроль. Значительно легче отсекать только возврат на один шаг назад в предшествующее состояние. Написать соответствующую программу предоставляется учащемуся в качестве упражнения.

6.4 Динамическое программирование

6.4.1 Понятие динамического программирования

Рекурсивная техника полезна, если задачу можно разбить на подзадачи, каждая из которых решается за разумное время, т.е. суммарный размер задач будет небольшим. Из формулы (4.1) оценки временной сложности вытекает, что если сумма размеров подзадач задачи размера n равна an для некоторой постоянной $a > 1$, то рекурсивный алгоритм, вероятно, имеет полиномиальную временную сложность. Но если разбиение задачи размера n сводит ее к n задачам размера $n - 1$, то рекурсивный алгоритм, вероятно, имеет экспоненциальную сложность. В этом случае часто можно получить более эффективные алгоритмы с помощью специальной техники, называемой динамическим программированием. Суть динамического програмирования основана на временном хранении в специальном массиве текущих решений на задачах предшествующих размеров. Динамическое программирование, в сущности, вычисляет решение всех подзадач. Вычисление идет от малых подзадач к большим, и ответы запоминаются в таблице. Преимущество этого метода состоит в том, что если уж подзадача решена, то ее ответ где-то хранится и никогда не вычисляется заново. Рассмотрим применение метода динамического программирования на простых примерах.

Пример 1. [1]

Рассмотрим произвольную последовательность N целых чисел. Между числами необходимо расставить знаки операций "+" или "− так что в результате получится некоторе арифметическое выражение. Это выражение всегда можно вычислить. Пусть, например, имеется последовательность четырех чисел 17, 5, -21, 15. Существует восемь различных выражений:

17 + 5 + -21 + 15 = 16,
17 + 5 + -21 - 15 = -14,
17 + 5 - -21 + 15 = 58,
17 + 5 - -21 - 15 = 28,
17 - 5 + -21 + 15 = 6,
17 - 5 + -21 - 15 = -24,
17 - 5 - -21 + 15 = 48,
17 - 5 - -21 - 15 = 18.

Назовем последовательность целых чисел "делимой" на K, если операции "+" или "−" так можно разместить между этими числами, что результирующее значение будет нацело делиться на K. В приведенном выше примере последовательность не делится на 5, но делится на 7. Делимость на 7, например, следует из выражения

$$17 + 5 + -21 - 15 = -14$$

или из выражения

$$17 + 5 - -21 - 15 = 28.$$

Задача состоит в том, чтобы для заданного числа K проверить делимость на него заданной последовательности. Пусть имеются следующие ограничения на значения исходных данных: $2 \leq K \leq 100$, $1 \leq N \leq 10000$, каждое число не больше 10000 по абсолютной величине. Очевидно, что решение задачи можно организовать с помощью полного перебора всех вариантов простановки знаков операций между числами и вычисления результирующей суммы. Процесс перебора всех возможных

[1] 1999-2000 ACM Notheastern European Regional Programming Contest

вариантов простановки знаков операций имеет экспоненциальную сложность 2^{N-1}, что при заданных ограничениях практически неприемлемо. Однако, динамическое программирование приводит к алгоритму сложности $O(K \cdot N)$.

Рассмотрим рекурсивную схему решения задачи. Пусть $s(j)$ — множество сумм, полученных в результате простановки знаков операций "+" или "−" между j числами $a[0], a[1], ..., a[j-1]$. Тогда рекурсивная схема вычисления этой функции имеет вид:

$$s(0) = \{a[0]\}$$
$$s(j+1) = \{s(j) + a[j]\} \cup \{s(j) - a[j]\}.$$

Заметим, что можно вычислять не сами суммы, а остатки от их деления на число K. В этом случае аналогичная рекурсивная схема для остатков имеет вид:

$$r(0) = \{a[0]/K\}$$
$$r(j+1) = \{(r(j) + a[j])/K\} \cup \{(r(j) - a[j])/K\}.$$

Если множество $r(N)$ содержит среди всех остатков число 0, то заданная последовательность делится на K. Анализ рекурсивной схемы показывает, что для вычисления $r(j+1)$ надо сначала вычислить все остатки $r(j)$, а затем перейти к вычислению нового множества остатков $r(j+1)$. При программировании $r(j)$ должны вычисляться и запоминаться во вспомогательном массиве. Чтобы избежать многократного сравнения вновь полученных остатков с уже имеющимися, введем специальный массив флажков M из K элементов для хранения $r(j)$. Элемент $M[i]$ равен 1, если в множестве остатков имеется остаток, равный i. И если $M[i]$ равен 0, то в множестве остатков отсутствует число i.

Например, для приведенной выше последовательности анализ делимости на 7 приведет к вычислению следующих динамических массивов остатков из 7 элементов (нулевай элемент этого массива означает, что остаток равен 0, первый элемент – остаток равен 1, ..., последний шестой элемент – остаток равен 6).

Ввели первое число 17, построили массив $M = \{0, 0, 0, 1, 0, 0, 0\}$, т.к.

$$17 mod\ 7 = 3.$$

Ввели второе число 5, построили новый массив $M = \{0, 1, 0, 0, 0, 1, 0\}$, т.к.

$$(3+5) mod\ 7 = 1,$$
$$(3-5) mod\ 7 = 5.$$

Ввели третье число -21, построили $M = 0, 1, 0, 0, 0, 1, 0$, т.к.

$$(1+(-21)) mod\ 7 = 1,$$
$$(1-(-21)) mod\ 7 = 1,$$
$$(5+(-21)) mod\ 7 = 5,$$
$$(5-(-21)) mod\ 7 = 5.$$

Наконец, при вводе последнего числа 15 получаем $M = \{1, 0, 1, 0, 1, 0, 1\}$, т.к.

$$(1+15) mod\ 7 = 2,$$
$$(1-15) mod\ 7 = 0,$$
$$(5+15) mod\ 7 = 6,$$
$$(5-15) mod\ 7 = 4.$$

Ненулевое значение $M[0]$ означает делимость без остатка на заданное число 7.

В программе на каждом очередном шаге формируется новый динамический массив остатков. Исходные данные читаются из файла *input.txt*.

```
#include <stdio.h>
#include <STDLIB.H>
#define KMAX   100

FILE * in=fopen("input.txt","r");
int   k,n;        // делитель и кол-во чисел
int r[KMAX],vr[KMAX];
       //  список флагов остатков и временный массив новых флагов

void main(void)
{
fscanf(in,"%d %d",&n,&k);
int i,j,a,b;
for (i=0; i<k; i++) r[i]=0;  // очистили массив
fscanf(in,"%d",&a);  // ввели первое число
a=a%k;
if (a>=0) r[a]=1; else r[k+a]=1;
for (i=1; i<n; i++)
     {
     for (j=0; j<k; j++)  vr[j]=0;
     fscanf(in,"%d",&a);  // ввели очередное число
     for (j=0; j<k; j++)
         if (r[j])
         {
         b=(j+a)%k;
         if (b>=0) vr[b]=1; else vr[k+b]=1;
         b=(j-a)%k;
         if (b>=0) vr[b]=1; else vr[k+b]=1;
         }
     for (j=0; j<k; j++)  r[j]=vr[j];
     }
fclose(in);
if (r[0])    printf("Последовательность делится на k=%d\n",k);
   else      printf("Последовательность не делится на k=%d\n",k);
}
```

Поскольку на каждом очередном шаге нужны остатки только одного предшествующего шага, можно несколько уменьшить время работы программы, используя двумерный массив $r[KMAX][2]$ вместо двух массивов $r[KMAX]$ и $vr[KMAX]$. Ускорение работы программы достигается за счет того, что отсутствует необходимость пересылки данных из одного массива в другой. Переход к новому динамическому массиву осуществляется просто за счет изменения индекса. При этом следует отметить, что порядок функции временной сложности остался неизменным $O(K \cdot N)$, уменьшился лишь коэффициент при выражении $K \cdot N$.

```
#include <stdio.h>
#include <STDLIB.H>
#define KMAX   100
```

```
FILE * in=fopen("input.txt","r");
int   k,n;        // делитель и кол-во чисел
int r[KMAX][2];    //  список флагов остатков
int num;           // индекс текущего остатка в массиве r

void main(void)
{
fscanf(in,"%d %d",&n,&k);
int i,j,a,b;
for (i=0; i<k; i++) r[i][0]=r[i][1]=0;  // очистили массив
num=0;  // индекс текущего вектора остатков
fscanf(in,"%d",&a);  // ввели первое число
a=a%k;
if (a>=0) r[a][num]=1; else r[k+a][num]=1;
for (i=1; i<n; i++)
     {
     for (j=0; j<k; j++)  r[j][1-num]=0;  // вектор остатков
     fscanf(in,"%d",&a);  // ввели очередное число
     for (j=0; j<k; j++)
         if (r[j][num])
         {
         b=(j+a)%k;
         if (b>=0) r[b][1-num]=1; else r[k+b][1-num]=1;
         b=(j-a)%k;
         if (b>=0) r[b][1-num]=1; else r[k+b][1-num]=1;
         }
     num=1-num;
     }
fclose(in);
if (r[0][num]) printf("Последовательность делится на k=%d\n",k);
   else printf("Последовательность не делится на k=%d\n",k);
}
```

6.4.2 Многоуровневые динамические массивы

Как уже отмечалось в предыдущем разделе, динамическое программирование позволяет вычислять и хранить решение для всех подзадач, от задач меньшего размера к задачам большего размера, причем ответы запоминаются в таблице. Рассмотрим более сложный пример применения метода динамического программирования, при котором формируемая динамическая таблица имеет сложную структуру и на каждом шаге в этой таблице используются значения, полученные на всех предшествующих шагах, а не только на последнем, как мы рассмотрели это в задаче о делимости последовательности целых чисел.

Рассмотрим вычисление произведения n матриц

$$M = M_1 \cdot M_2 \cdot ... \cdot M_n,$$

где M_i - матрица с r_{i-1} строками и r_i столбцами. Порядок, в котором эти матрицы перемножаются, может существенно сказаться на общем числе операций, требуемых для вычисления M. Рассмотрим умножение матриц на примере. Пусть даны четыре матрицы $M[10][20], M[20][50], M[50][1], M[1][100]$. Если воспользоваться обычной

формулой вычисления элемента матрицы $C = A \cdot B$:

$$c[i][j] = \sum_{k=1}^{p} a[i][k] \cdot b[k][j],$$

то умножение матрицы $A[q][p]$ на $B[p][h]$ требует $q \cdot p \cdot h$ операций умножения. Поэтому если вычислять M в порядке

$$M = M_1 \cdot (M_2 \cdot (M_3 \cdot M_4))$$

потребуется 125 000 операций умножения, тогда как вычисление в порядке

$$M = (M_1 \cdot (M_2 \cdot M_3)) \cdot M_4$$

потребует всего 2 200 операций.

Рассмотрим рекурсивную схему вычисления минимального числа умножений. Обозначим через $m(i, j)$ минимальную временную сложность вычисления произведения $M_i \cdot M_{i+1} \cdot \cdots \cdot M_j$, и через $r[i-1]$, $r[i]$ – размерность матрицы M_i. Очевидно, что $m(i, i) = 0$, т.к. матрицу вычислять не нужно. При i строго меньшем j минимальное значение $m(i, j)$ получается при такой расстановке скобок в выражении

$$(M_i \cdot M_{i+1} \cdot ... \cdot M_k) \cdot (M_{k+1} \cdot ... \cdot M_j),$$

когда минимальной является сумма числа выполняемых операций

$$m(i, k) + m(k + 1, j) + r[i - 1] \cdot r[k] \cdot r[j].$$

Таким образом, получаем рекурсивную функцию

$$m(i, j) = 0, \text{ если } i = j,$$

$$m(i, j) = Min\{m(i, k) + m(k + 1, j) + r[i - 1] \cdot r[k] \cdot r[j]\}, \text{ если } i < j.$$

Значения $m(i, j)$ необходимо хранить в динамическом массиве и вычислять в порядке возрастания разностей параметров i и j. Начинают вычисления для $m(i, i)$ для всех i от 1 до n. Затем вычисляют $m(i, i + 1)$ для всех i, затем $m(i, i + 2)$ для всех i, и т.д. При таком порядке вычислений последовательно формируется значение минимальной сложности и в итоге получаем $m(1, n)$. В рассмотренном выше примере умножения четырех матриц $M[10][20]$, $M[20][50]$, $M[50][1]$, $M[1][100]$ получаем следующие значения:

$m(1,1) = 0$	$m(2,2) = 0$	$m(3,3) = 0$	$m(4,4) = 0$
$m(1,2) = 10000$	$m(2,3) = 1000$	$m(3,4) = 5000$	
$m(1,3) = 1200$	$m(2,4) = 3000$		
$m(1,4) = 2200$			

Действительно,

$m(1, 2) = Min\{m(1, 1) + m(2, 2) + 10 * 20 * 50\} = 10000,$

$m(2, 3) = Min\{m(2, 2) + m(3, 3) + 20 * 50 * 1\} = 1000,$

$m(3, 4) = Min\{m(3, 3) + m(4, 4) + 1 * 50 * 100\} = 5000,$

$m(1, 3) = Min\{m(1, 1) + m(2, 3) + 10 * 20 * 1, m(1, 2) + m(3, 3) + 10 * 50 * 1\} = Min\{0 + 1000 + 200, 10000 + 0 + 500\} = 1200$ при k=1,

$m(2, 4) = Min\{m(2, 2) + m(3, 4) + 1 * 50 * 100, m(2, 3) + m(4, 4) + 20 * 1 * 100\} = Min\{0 + 5000 + 5000, 1000 + 0 + 2000\} = 3000$ при $k = 3$,

$m(1,4) = Min\{m(1,1)+m(2,4)+10*20*100, m(1,2)+m(3,4)+10*50*100, m(1,3)+m(4,4)+10*1*100\} = Min\{0+3000+20000, 10000+5000+50000, 1200+0+1000\} = 3000$ при $k = 3$.

Если вместе со значением минимальной сложности хранить и значение переменной k, соответствующей найденному оптимальному значению, то обратный проход по массиву позволяет расставить скобки в выражении.

$m(1,1) = 0$	$m(2,2) = 0$	$m(3,3) = 0$	$m(4,4) = 0$
$k = 1$	$k = 1$	$k = 3$	$k = 4$
$m(1,2) = 10000$	$m(2,3) = 1000$	$m(3,4) = 5000$	
$k = 1$	$k = 2$	$k = 3$	
$m(1,3) = 1200$	$m(2,4) = 3000$		
$k = 1$	$k = 3$		
$m(1,4) = 2200$			
$k = 3$			

Рассмотренный алгоритм реализует следующая программа.

```
#include <stdio.h>
#include <STDLIB.H>
#include <MATH.H>
#define MaxN 100
       // максимальное число умножаемых матриц

int N, x[MaxN+1]; // число матриц и их размерности
unsigned int p[MaxN+1][MaxN+1];       // путь умножения
unsigned long int  m[MaxN+1][MaxN+1]; // динамический массив
FILE *  in  = fopen("input.txt","r");

unsigned long int Dinamik(void)
// число умножений матриц с 1 по N
// если рассматривать число умножений матриц с i по j, то
// m[i][j]=0 при i==j
// m[i][j]=MIN (m[i][k]+m[k+1][j]+x[i-1]*x[k]*x[j]) при i!=j
{
int l, i, j, k; unsigned long int  LokMin, s;
for (i=1;i<=N; i++) {m[i][i]=0;  p[i][i]=i; }
for (l=1; l<N; l++)  //  l - длина умножения - число матриц
    for (i=1; i<N; i++)
        {
        j=i+l;
        m[i][j]=0;
        for (k=i; k<j; k++)
            {
            LokMin=x[i-1]*x[k];
            LokMin=LokMin*x[j];
            LokMin=LokMin+m[i][k]+m[k+1][j];
            if ( (LokMin<m[i][j]) || (k==i) )
                    {m[i][j]=LokMin; p[i][j]=k;}
            }
```

```
        }
return m[1][N];
}

void RecFormula(int i, int j)
{
if (i==j) \{printf("M[%d]",p[i][i]); return;\}
printf("(");
RecFormula( i, p[i][j]);
printf("*");
RecFormula( p[i][j]+1,j);
printf(")");
}

void main(void)
{
int i,j; unsigned long int sum;
fscanf(in,"%d ",&N);
for (i=0; i<N+1; i++)    fscanf(in,"%d",&x[i]);
sum=Dinamik();
printf("Число операций умножения = %lu \n",sum);
printf("Формула вычисления произведения \n");
RecFormula(1,N);
printf(" \n");
fclose(out);
}
```

6.5 Виртуальные графы

Рассмотрим следующую задачу, которая встречается во многих прикладных вопросах: задачу о поиске в заданном графе пути, соединяющем две заданные вершины, на котором достигается минимум или максимум некоторой аддитивной функции, определяемой на дугах и путях. Чаще всего эта функция определяется как длина пути и задача называется задачай о кратчайшем пути. Хорошо известен эффективный алгоритм Дейкстры поиска кратчайшего пути в графе, когда веса всех дуг неотрицательны. Напомним кратко этот алгоритм.

Обозначим исходные данные: V — множество вершин, s — выделенная начальная вершина, от которой находится путь минимальной длины, A — матрица весов дуг, n — число вершин. Результат работы алгоритма — массив D расстояний от источника s до всех вершин графа. Массив T — вспомогательный массив не помеченных вершин; в начальный момент этот массив представляет все вершины, кроме источника s; затем по мере обработки графа из этого массива исключаются обработанные вершины.

1 $for(v \in V)d[v] = A[s][v];$
2 $D[s] = 0;$
3 $T = V \setminus \{s\}$
4 $while(T! = \emptyset)$
5 $\{$
6 найдем u — такую вершину множества непомеченных вершин T, для которой

минимальным является расстояние $D[u] = Min(D[p]|p \in T)$

7 $T = T \setminus \{u\}$

8 $for(v \in T)D[v] = Min(D[v], D[u] + A[u][v])$

9 }

Проанализируем временную оценку алгоритма Дейкстры. Очевидно, что внешний цикл выполняется $(n-1)$ раз, причем каждое его выполнение требует $O(n)$ шагов: $O(n)$ для нахождения вершины u в строке 6 (предположим, что множество T представлено списком) и $O(n)$ шагов для выполнения цикла 8. Таким образом, временная сложность алгоритма Дейкстры составляет $O(n^2)$. При хорошем представлении структур данных в виде бинарного дерева можно получить вариант алгоритма со сложностью $O(m \cdot log_2 n)$, где m — число дуг графа.

Рассмотрим теперь технический прием, представляющий развитие области применения метода Дейкстры. На практике часто встречаются задачи, в которых можно построить граф состояний исследуемого объекта. Известно начальное состояние и требуется кратчайшим путем перевести объект в некоторое конечное состояние. Как правило, число состояний такого объекта достаточно велико и, следовательно, матрицу переходов невозможно разместить в памяти компьютера. На помощь приходят виртуальные графы, т.е. графы, которые не описаны явно и не присутствуют в памяти программы. Анализ алгоритма Дейксты показывает, что для вычисления нового значения минимального расстояния $D[v] = Min(D[v], D[u] + A[u][v])$ требуется длина дуги (u, v) между двумя заданными вершинами. Для вычисления этого расстояния необходимо написать функцию, которая возвращает длину дуги для двух заданных состояний. Эти состояния в программе могут определяться как своими номерами, так и некоторыми параметрами, однозначно определяющими эти состояния.

Пример. [2]

Ребенку подарили набор из F фломастеров разных цветов. Желая проверить новые фломастеры, ребенок нарисовал N пронумерованных цветных кружков, затем соединил некоторые из них цветными ориентированными дугами. Любые два кружка могут быть соединены любым количеством дуг любых цветов. Каждый цвет (либо кружка либо дуги) имеет свой номер от 1 до K. В начале игры ребенок выбирает три целых числа L, K, Q, лежащие в пределах от 1 до N. Затем он ставит одну фишку на кружок с номером Q, а другую — на кружок с номером K. После этого ребенок начинает передвигать фишки, в соответствии со следующими правилами:

- фишка может пройти дугу в том случае, если цвет этой дуги совпадает с цветом кружка, на котором стоит другая фишка;
- фишка может двигаться только в направлении ориентации дуги;
- две фишки нельзя ставить на один и тот же кружок одновременно;
- очередность ходов фишек не установлена, т.е. не обязательно двигать первую и вторую фишки по очереди;
- только одна фишка может быть передвинута за один ход;
- один ход представляет собой движение только по одной дуге.

Игра заканчивается, если одна из фишек достигнет кружка с номером Q. Надо написать программу, которая находит длину кратчайшего возможного пути, если он существует.

Входные данные

Первая строка содержит числа N, L, K, Q, разделенные пробелами. Вторая строка содержит N чисел $c_1, c_2, ..., c_N$, где c_i — цвет i–го кружка. В третьей строке содержится число M, обозначающее количество ориентированных дуг. Затем следуют M

[2] 1997-1998 ACM Northeastern European Regional Programming Contest

строк, содержащих описание дуг. Каждая дуга задается тремя числами a_j, b_j, c_j, где a_j и b_j — номера кружков, соединенных этой дугой, c_j — цвет дуги.

Выходные данные

Первая строка должна содержать слово **ДА**, если этот путь существует, и **НЕТ** в противном случае. Вторая строка должна содержать минимальное число ходов, за которое закончится игра.

Ограничения: число цветов $F <= 100$, число кружков $N <= 100$, число дуг $M <= 10000$.

Решение задачи Очевидно, что состоянием задачи является пара чисел (a, b) — номера вершин, в которых находятся фишки. Поскольку известно N — максимальное число возможных вершин, состояние можно представить целым числом $a * N + b$. Зная номер состояния, можно однозначно определить номера соответствующих вершин, образующих это состояние. Для этой цели в программе реализована функция *int GetNumber (int a, int b)*, возвращающая номер вершины графа состояний. Для вычисления длины дуги между двумя заданными виртуального графа реализована функция *int RebroIgra (int n, int m)*, где n и m — номера состояний. При этом значение -1 означает отсутствие дуги ("бесконечную" длину). Функции *long Sum (long a, long b)* и *long Min (long a, long b)* предназначены соответственно для вычисления суммы расстояний и минимального значения (пункт 8 алгоритма Дейкстры).

```
#include <stdio.h>
#include <STDLIB.H>
#define MaxN 102

FILE *  in  = fopen("input.txt","r");
FILE *  out = fopen("output.txt","w");
int N, //число вершин
    L,  // нач позиция 1-ой пешки
    K,  // нач позиция 2-ой пешки
    Q,  // целевая вершина
    M;  // число дуг
char ColorVer[MaxN]; //цвет вершин
char ColorReb[MaxN][MaxN]; //цвет ребер
long int d[MaxN*MaxN];
char flag[MaxN*MaxN];

void GetData(void)
{
int a,b,i,j,Color;
fscanf(in,"%d %d %d %d",&N,&L,&K,&Q);
for (i=0;i<N;i++) fscanf(in,"%d ",&ColorVer[i]);
   fscanf(in,"%d ",&M);
 for (i=0;i<M;i++)
     {
     fscanf(in,"%d %d %d",&a,&b,&Color);
     ColorReb[a-1][b-1]=Color;
     }
}
```

```
int Result(int n) // является ли вершина n целевой?
{
int i,j;
i=n/N;     j=n%N;
if (i==j) return 0;
if ( (i==Q-1)||(j==Q-1) ) return 1;
return 0;
}

int GetNumber(int a,int b) // номер вершины графа представлений
{  return a*N+b; }

long Sum(long a, long b)
{
if (a==-1) return -1;
if (b==-1) return -1;
return a+b;
}

long Min(long a, long b)
{
   if (a==-1) return b;
   if (b==-1) return a;
   if (a<b) return a; else return b;
}

int RebroIgra(int n, int m)  // есть ли дуга между вершинами
{int i1,i2, // пешка 1
     j1,j2; // пешка 2
i1=n/N;     j1=n%N;
i2=m/N;     j2=m%N;
if ( (i1!=i2)&&(j1!=j2) ) return -1; // одна пешка не должна двигаться
if ( (i1==i2)&&(j1==j2) ) return -1; // обе пешки не двигались
if (  (i1==i2)&& // 1 на месте
       ( ColorReb[j1][j2]==ColorVer[i1] ) ) // правило цветов
        return 1;
if (  (j1==j2)&& // 2 на месте
     ( ColorReb[i1][i2]==ColorVer[j1] ) ) // правило цветов
        return 1;
return -1;
}

void Deikstra(void)
{
int i,a,b,kol,
    s,   //начальная вершина
    i_min, final,d_min;
kol=N*N;
s=GetNumber(L-1,K-1); // начальная вершина
for (i=0;i<kol;i++)
```

```
    { flag[i]=0;     d[i]=RebroIgra(s,i);     }
d[s]=0;
flag[s]=1;
final=0;
while (final==0)
    {
    i_min=0; d_min=-1; final=1;
    for (i=0;i<kol;i++)
        {
        if ( (d[i]!=-1)  && (flag[i]==0) )
           {if(d_min!=Min(d_min,d[i]) )
                   {  final=0; i_min=i;
                   d_min=Min(d_min,d[i]);
                   }
           }
        } // нашли вершину с минимальным d
    if (final==0)
       {
       flag[i_min]=1;
       for (i=0;i<kol;i++)
           {
           if (flag[i]==0)
               d[i]=Min( d[i], Sum(d[i_min],RebroIgra(i_min,i)));
           }
        }
    } // конец расчета Дейкстры
d_min=-1;
for (i=0;i<kol;i++)
   if ( (d[i]!=-1) && (Result(i)==1) )
       d_min=Min(d_min,d[i]);
if (d_min==-1)
    fprintf(out,"NO\n");
else
    fprintf(out,"YES\n%d\n",d_min);
fclose(out);
}

void main(void)
{
  GetData();
  Deikstra();
}
```

6.6 Эффективные алгоритмы на графах

Многие прикладные задачи можно сформулировать в терминах теории графов. Как известно, любой граф однозначно можно задать набором вершин и дуг, связывающих вершины. Будем обозначать произвольный граф $G = (V, E)$, где V – множество

вершин и E – множество дуг.

В разделе, посвященному NP–полным задачам, уже формулировались некоторые NP–полные задачи на графах, например задача "Гамильтонов цикл" и "Клика". К сожалению, достаточно большое число задач на графах являются NP–полными и, следовательно, требуют полного перебора для решения. К таким задачам, кроме уже рассмотренных в предшествующем разделе, можно отнести следующие широко известные задачи.

1. Задача "Вершинное покрытие". Задан граф $G = (V, E)$ и положительное целое число $K <= |V|$. Вершинным покрытием мощности, не превосходящей K, называется такое подмножество вершин $V_1 \subseteq V$, что $|V_1| \leq K$ и для любого ребра $(u, v) \in E$ по крайней мере одна из вершин u или v принадлежит V_1.

Вопрос. Существует ли в графе G вершинное покрытие мощности, не превосходящей K?

2. Задача "Множество вершин, разрезающих циклы". Задан граф $G = (V, E)$ и положительное целое число $K <= |V|$.

Вопрос. Существует ли подмножество вершин $V_1 \subseteq V$, такое, что $|V_1| <= K$ и V_1 содержит по крайней мере одну вершину любого ориентированного цикла в G?

Замечание. Соответствующая задача для неориентированных графов также NP–полна.

3. Задача "Множество дуг, разрезающих циклы". Задан граф $G = (V, E)$ и положительное целое число $K <= |V|$.

Вопрос. Существует ли подмножество дуг $E_1 \subseteq E$, такое, что $|E_1| \leq K$ и E_1 содержит по крайней мере одну дугу из каждого ориентированного цикла в G?

Замечание. Соответствующая задача для неориентированных графов тривиальным образом решается за полиномиальное время.

4. Задача "Независимое множество вершин". Задан граф $G = (V, E)$ и положительное целое число $K <= |V|$.

Вопрос. Верно ли, что в G существует независимое множество вершин мощности не менее K? Иными словами, верно ли, что существует подмножество вершин $V_1 \subseteq V$, такое, что $|V_1| \geq K$ и никакие две вершины из V_1 не соединены ребром?

Замечание. Соответствующая задача для двудольных графов решается за полиномиальное время.

5. Задача "Изоморфизм подграфу". Заданы два графа $G = (V_1, E_1)$ и $H = (V_2, E_2)$. Граф $G = (V_1, E_1)$ называется изоморфным графу $T = (V, E)$, если $|V| = |V_1|$, $|E| = |E_1|$ и такая существует взаимно–однозначная функция $f(u) : V \to V_1$, что $(u, v) \in E$ тогда и только тогда, когда $(f(u), f(v)) \in E_1$.

Вопрос. Содержит ли граф, G подграф, изоморфный графу H?

Замечание. Задача решается за полиномиальное время, если G — лес, а H — дерево.

6. Задача "Стягиваемость графа". Заданы два графа $G = (V_1, E_1)$ и $H = (V_2, E_2)$. Последовательностью стягивания ребер называется такая последовательность шагов, на каждом из которых две соседние вершины u и v заменяются одной вершиной w, соединенной ребрами с теми и только теми вершинами, с которыми были соединены u или v.

Вопрос. Можно ли последовательным стягиванием ребер графа G получить граф, изоморфный H?

Теперь рассмотрим примеры задач на графах, решение которых выполняется за полиномиальное время. Одну из таких задач мы уже рассмотрели в предыдущем разделе 6.5. Это задача поиска наикратчайшего пути, решаемая с помощью алгоритмы Дейкстры за время $O(N^2)$. К полиномиальным алгоритмам относятся, в основном,

алгоритмы решения задач, в которых речь идет о связности в графе. Сюда входят алгоритмы для нахождения остовных деревьев, двусвязных компонент, сильно связанных компонент и путей между узлами. В качестве примера рассмотрим задачу о паросочетании в двудольном графе.

Граф $G = (V, E)$ называется двудольным, если множество V его вершин разбито на два множества M_1 и M_2 и все начала дуг принадлежат множеству M_1, а все концы — множеству M_2. Одна из наиболее характерных задач, решаемых для таких графов, состоит в построении полных или частичных взаимооднозначных соответствий между элементами этих множеств, составленных из допустимых пар элементов (a, b), $a \in M_1$ и $b \in M_2$.

Набор дуг $V_1 \subseteq V$ называется паросочетанием, если для любой пары дуг из V_1 начала и концы этих дуг различны. Рассмотрим задачу построения паросочетания, максимального по числу входящих в него дуг. Существует множество алгоритмов решения этой задачи. Рассмотрим один из простейших алгоритмов, решающий задачу за полиномиальное время. Пусть дан двудольный граф и известно разбиение множества вершин на подмножества M_1 и M_2. Рассмотрим один из простейших методов, основанный на поиске максимального потока в сети. Для этого введем еще две дополнительные вершины графа:

— начальную вершину F_0, из которой постороим дуги во все вершины подмножества M_1;

— конечную вершину F_1, в которой постороим дуги из всех вершин подмножества M_2.

Нетрудно заметить, что максимальное число паросочетаний представляет собой максимальный поток из начальной вершины F_0 в конечную вершину F_1 этого графа.

Задача поиска такого потока решается следующим образом. Сначала находится произвольное паросочетание, затем последовательно выполняется поиск очередной чередующейся цепи и корректируется текущее паросочетание. Эти действия выполняются следующим образом:

— находится произвольная чередующаяся цепь (проходящая строго по чередующимся вершинам типа M_1 и M_2), которая соответствует увеличивающему потоку; всем дугам этой цепи присваивается поток, равный 1; найденный увеличивающий поток с помощью операции XOR с текущим паросочетанием дает поток большей мощности;

— сразу заметим, что дуги, поток в которых равен направлению потока в чередующейся цепи, не участвуют в алгоритме поиска увеличивающего потока;

— повторяем процесс поиска увеличивающего потока, пока он существует.

Пример поиска увеличивающей цепи приведен на рис. 6.6.

Исходное паросочетание $1 \to A$, $2 \to C$, $4 \to E$, $5 \to D$, $6 \to F$ позволяет построить чередующуюся цепь

$$3 \to A \to 1 \to D \to 5 \to E \to 4 \to C \to 2 \to B,$$

в которой используются только либо свободные дуги $3 \to A$, $1 \to D$, $5 \to E$, $4 \to C$, $2 \to B$, либо такие дуги $A \to 1$, $D \to 5$, $E \to 4$, $C \to 2$. главный поток в которых противоположен направлению потока в чередующейся цепи. Следующая теорема доказывает, что данный алгоритм приводит к построению максимального паросочетания.

Теорема Бержа. Паросочетание P в двудольном графе G наибольшее тогда и только тогда, когда в G не существует чередующейся цепи относительно P. (Без доказвтельства.)

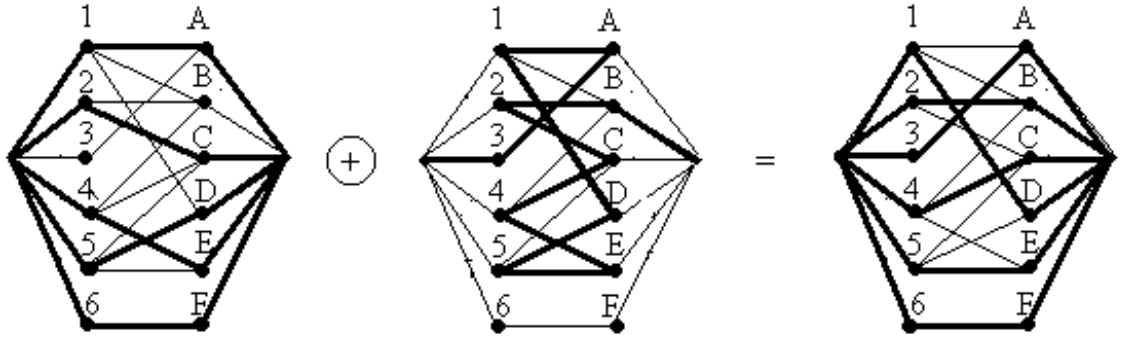


Рис. 6.6: Увеличивающая чередующаяся цепь в двудольном графе.

Можно выполнить оценку временной сложности алгоритма поиска максимального паросочетания. Поскольку на каждом шаге можно прибавить не менее одной дуги, то внешний цикл выполняется N раз, где N — максимальное число вершин в M_1 и M_2. Поиск чередующейся цепи требует $O(N^2)$ операций, так же как и поиск первоначального произвольного паросочетания. Таким образом, временная сложность задачи $O(N^2) + N \cdot O(N^2) = O(N^3)$.

В качестве примера рассмотрим следующую задачу. [3]

Охотник гуляет со своей охотничьей собакой. Охотник идет с постоянной скоростью и его маршрут представляет собой ломаную, проходящую через N точек, координаты (x_i, y_i) которых заданы. Собака гуляет по своему собственному маршруту, но всегда встречается с хозяином в точках излома его маршрута. Собака может бегать со скоростью, в два раза большей скорости хозяина. Пока хозяин путешествует от одной точки до другой, собака может посетить только ей известные специальные точки, число которых M и координаты которых (x_i^1, y_i^1) также известны. Пока хозяин совершает переход между двумя точками его маршрута, собака может посетить не более одной известной ей точки, причем каждая такая точка посещается собакой не более одного раза.

Найти маршрут собаки, при котором она посетит максимальное число интересующих ее точек. Например, пусть $N = 4, M = 5$, координаты пути охотника (1; 4), (5; 7), (5; 2), (-2; 4), а координаты интересующих собаку точек (-4; -2), (3; 9), (1; 2), (-1; 3), (8; -3). Тогда собака может посетить только две интересующие ее точки и двигаться, например, по маршруту (1; 4), (3; 9), (5; 7), (5; 2), (1; 2), (-2; 4). Соответствующее графическое изображение приведено на рис. 6.7.

Рассмотрим решение этой задачи. Поставим в соответствие каждой точке пути охотника $(x[i], y[i])$ вершину $A[i]$ двудольного графа. Каждой из специальных точек собаки $(X[j], Y[j])$ также поставим в соответствие вершину $B[j]$ этого графа. Дуга связывает вершины $A[i]$ и $B[j]$, если собака при движении охотника от $(x[i], y[i])$ к $(x[i+1], y[i+1])$ успевает посетить точку $(X[j], Y[j])$. Получим двудольный граф. Для приведенного в задаче примера он представлен на рис. 6.8, а соответствующий максимальный поток в сети — на рис. 6.9.

Таким образом, решение поставленной задачи эквивалентно поиску максимального паросочетания в двудольном графе. Вариант максимального паросочетания для примера рис. 6.7 представлен на рис. 6.9 и соответствует максимальному потоку в

[3] 1998-1999 ACM Northeaster European Regional Programming Contest

. . . . . .

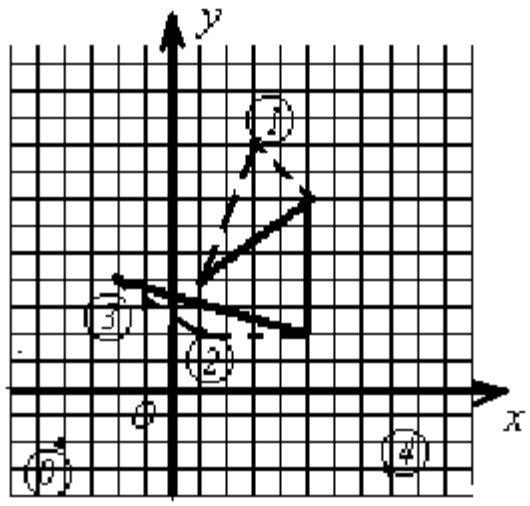


Рис. 6.7: Маршруты охотника и собаки.

сети.

Можно привести пример программы, решающей эту задачу.

```
#include <stdio.h>
#include <STDLIB.H>
#include <STRING.H>
#include <MATH.H>
#define  EPS 2.0e-12           // точность вычислений
#define MaxN 100               // max число точек пути охотника
#define MaxM 100               // max число специальных собачих точек

int N, M;  // фактическое число точек охотника и собаки
int x[MaxN], y[MaxN]; // путь охотника
int xd[MaxM], yd[MaxM]; // специальные собачьи точки
int  Matr[MaxN][MaxM];
          // матрица переходов графа состояний "охотник==>собака"
int C[MaxM+MaxN+2], NumC; // найденная увеличивающая чередующаяся цепь
int Res[MaxN+MaxM+2], NumRes; // результирующий поток и его длина
int FlagHunt[MaxN],FlagDog[MaxM]; // флаги использования в Res
int Stok[MaxM];  // признак стока в собачьей точке
FILE *in =  fopen("dog.in","r");
FILE *out = fopen("dog.out","w");

double Len(int x0, int y0, int x1, int y1)
// расстояние между двумя точками
{
double dx,dy;
dx=x0-x1;  dy=y0-y1;
return sqrt( dx*dx + dy*dy );
}
```

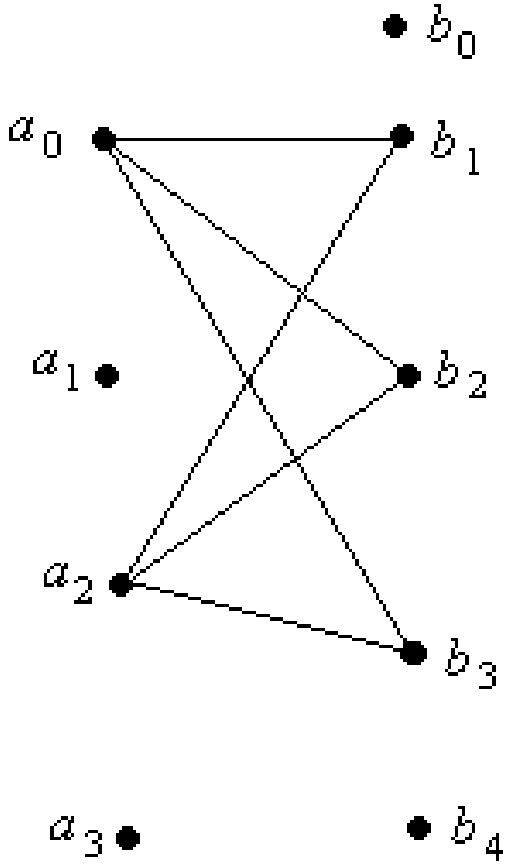


Рис. 6.8: Двудольный граф к задаче об охотнике и собаке.

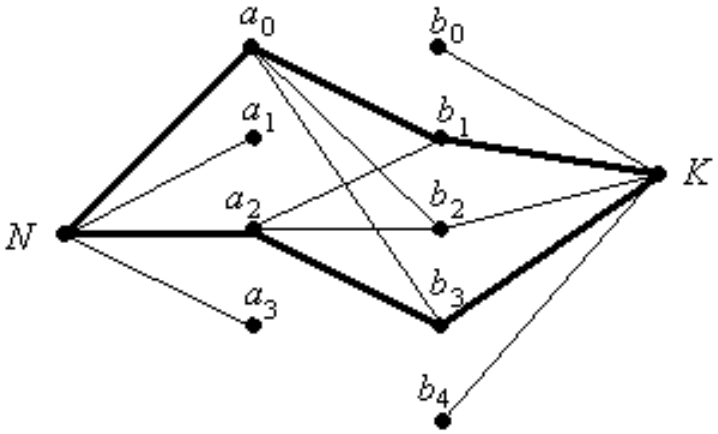


Рис. 6.9: Максимальный поток в сети для задачи об охотнике и собаке.

```
void GetData(void)  // Ввести данные и сформировать граф переходов в Matr
{
int i,j;
fscanf(in,"%d %d",&N,&M);
// вводим путь охотника:
for (i=0; i<N; i++) fscanf(in,"%d %d",&x[i],&y[i]);
// вводим собачьи точки:
for (i=0; i<M; i++) fscanf(in,"%d %d",&xd[i],&yd[i]);
// Строим граф:
// Matr[i][j]=0, если нет пути от i-охотника к j-собаки
// Matr[i][j]=1,  если есть путь
for (i=0; i<N; i++) for (j=0; j<M; j++)  Matr[j][i]=0;
double l,l1,l2;
for (i=0; i<N-1; i++)          // i - охотник
    for (j=0; j<M; j++)     // j - собака
        {
        l =Len(x[i],y[i],x[i+1],y[i+1]);
        l1=Len(x[i],y[i],xd[j],yd[j]);
        l2=Len(x[i+1],y[i+1],xd[j],yd[j]);
        if ( (l1+l2 <= 2*l)  )
             // собака успеет добежать с двойной скоростью
                        Matr[i][j]=1;
        }
}

int Chain(int First)  // найти цепь от охотничьей точки First
{
int i,j, Flag1;
FlagHunt[First]=0;
for (j=0; j<M; j++) // j - собака
     if ( FlagDog[j] && (Matr[First][j]==1) && Stok[j]==0)
        {      // делаем путь First ==> j
        C[NumC++]=j;  FlagDog[j]=0;  Matr[First][j]=-1;
        return 1;
        }
for (j=0; j<M; j++) // j - собака
     if ( FlagDog[j] && (Matr[First][j]==1) )
        {      // делаем путь First ==> j
        C[NumC++]=j;  FlagDog[j]=0;  Matr[First][j]=-1;
        // пытаемся продолжить j ==> i
        for (i=0; i<N ; i++)
               if ( (Matr[i][j]==-1) && FlagHunt[i] )
                  {
                  C[NumC++]=i;
                  if (Chain(i)) return 1;
                            else NumC--; // уничтожили j == > i
                  }   // конец цикла по всем i
        // попали сюда только при условии отсутствия j==>i
        if (Stok[j]==0) return 1;
```

```
              else { NumC--;  FlagDog[j]=1; Matr[First][j]=1; }
        }  // конец цикла по всем j
return 0;
}

int Next(void)    // найти увеличивающую чередующуюся цепь
{
int i, k, Flag=0, Flag1;
for (i=0; (i<N) && (Flag==0); i++)
     {
     Flag1=1;
     for (k=0; (k<NumRes) && Flag1; k+=2)
         if (Res[k]==i) Flag1=0;           // нельзя начинать с i
     if (Flag1) { NumC=1; C[0]=i; Flag=Chain(i);}
     }
return Flag;
}

void XOR(void)  // операция XOR над паросочетанием Res и цепью C
{
int i,j,a,b;
for (j=1; j<NumC-2; j+=2)
    for (i=0; i<NumRes; i+=2)
    if ( (C[j+1]==Res[i]) && (C[j]==Res[i+1]) )
       {  // исключаем Res[i],Res[i+1]
       a=Res[i]; b=Res[i+1];
       Matr[a][b]=1;
       Res[i]=Res[NumRes-2]; Res[i+1]=Res[NumRes-1];
       NumRes-=2;
       }
// исключили все обратные дуги, теперь добавляем прямые
for (i=0; i<NumC; i+=2)
     { Res[NumRes++]=C[i]; Res[NumRes++]=C[i+1]; }
}

void Solve(void)
{
// Находим максимальный поток в сети от источника ко стоку.
// Пропускная способность каждой дуги равна 1.
NumRes=0;
int i,j, N1;
for (i=0; i<N; i++) FlagHunt[i]=1;
for (j=0; j<M; j++) { FlagDog[j]=1; Stok[j]=0; }
for (i=0; i<N; i++)
     for (j=0; (j<M) && FlagHunt[i]; j++)
       if ( (Matr[i][j]==1) && FlagDog[j] )
          {
          Matr[i][j]=-1; FlagDog[j]=0; FlagHunt[i]=0;
          Res[NumRes++]=i;  Res[NumRes++]=j;
          }
```

```
// построили первое попавшееся паросочетание
int Flag;
do
  {
  for (i=0; i<N; i++) FlagHunt[i]=1;
  for (j=0; j<M; j++)
      {
      FlagDog[j]=1;
      for (i=1; i<NumRes; i+=2) if (Res[i]==j) Stok[j]=1;
      }
  Flag=Next();  // следующий увеличивающий поток
  if (Flag) XOR(); // формируем новый результат
  }while (Flag);
int xr[MaxN+MaxM], yr[MaxN+MaxM]; // результирующий путь
int nr; // число точек результирующего пути
int a;
Flag=1;
while (Flag)
      { // сортировка точек паросочетания
      Flag=0;
      for (i=0; i<NumRes-2; i+=2)
           if (Res[i]>Res[i+2])
              {  //переставляем точки паросочетания
              Flag=1;
              a=Res[i];   Res[i]=Res[i+2];   Res[i+2]=a;
              a=Res[i+1]; Res[i+1]=Res[i+3]; Res[i+3]=a;
              }
      }
int j0=0; // начальная точка охотника
nr=0;
for (i=0; i<NumRes; i+=2)
    {
    for (j=j0; j<=Res[i]; j++) {xr[nr]=x[j]; yr[nr++]=y[j];}
        // взяли начальный маршрут
    xr[nr]=xd[Res[i+1]]; yr[nr++]=yd[Res[i+1]];
    j0=Res[i]+1;
    }
for (j=j0; j<N; j++) {xr[nr]=x[j]; yr[nr++]=y[j];}
fprintf(out,"%d\n",nr);
for  (i=0; i<nr; i++) fprintf(out,"%d %d ",xr[i],yr[i]);
fprintf(out,"\n");
}

int main(void)
{
GetData();
Solve();
fclose(in); fclose(out);
return 0;
}
```

6.7 Производящие функции

В комбинаторных задачах на подсчет числа объектов искомым решением часто является последовательность

$$a_0, a_1, a_2, ...,$$

где a_k — число искомых объектов размерности k. Например, если мы ищем число подмножеств множества, состоящего из n элементов, то

$$a_k = C_n^k = \frac{n!}{k!(n-k)!}.$$

В этом случае удобно последовательности

$$a_0, a_1, a_2, ...$$

поставить в соответствие формальный ряд

$$A(x) = \sum_{i=0}^{\infty} a_i x^i, \tag{6.1}$$

называемый *производящей функцией* для данной последовательности. Название "формальный ряд" означает, что формулу (6.1) мы трактуем только как удобную запись нашей последовательности. Совершенно неважно, для каких действительных целей предназначены коэффициенты этого ряда. Несущественно также и для каких значений переменной x этот ряд сходится. Поэтому мы никогда не будем вычислять значение этого ряда для конкретного значения переменной x, мы будем только выполнять некоторые операции на таких рядах, а затем определим коэффициенты при отдельных степенях переменной x. Тем самым мы определим выражение для коэффициента соответствующего члена ряда — элемента последовательности a_i.

Для того, чтобы можно было оперировать такими рядами, определим специальные операции. Пусть даны два произвольных ряда

$$A(x) = \sum_{i=0}^{\infty} a_i x^i,$$

$$B(x) = \sum_{i=0}^{\infty} b_i x^i.$$

Определим операцию *сложения* :

$$A(x) + B(x) = \sum_{i=0}^{\infty} (a_i + b_i) x^i, \tag{6.2}$$

операцию *умножения на число*:

$$pA(x) = \sum_{i=0}^{\infty} p a_i x^i \tag{6.3}$$

и операцию *произведения* :

$$A(x) \cdot B(x) = \sum_{i=0}^{\infty} c_i x^i, \tag{6.4}$$

где

$$c_k = a_0 b_k + a_1 b_{k-1} + ... + a_k b_0 = \sum_{j=0}^{k} a_j b_{k-j}.$$

Из математического анализа известно, что если ряд (6.1) сходится в некоторой окрестности нуля, то его сумма $A(x)$ является аналитической функцией в этой окрестности. Тогда выражение (6.1) — не что иное, как ряд Маклорена. Как известно, ряд Тейлора для функции $f(x)$ в окрестности точки $x = t$ имеет вид

$$f(x) = f(t) + \frac{x-t}{1!} f'(t) + ... + \frac{(x-t)^k}{k!} f^{(k)}(t) + ...$$

При $t = 0$ получаем частный случай ряда Тейлора — ряд Маклорена:

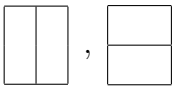
$$f(x) = f(0) + \frac{x}{1!} f'(0) + ... + \frac{x^k}{k!} f^{(k)}(0) + ... = \sum_{k=0}^{\infty} \frac{x^k}{k!} f^{(k)}(0).$$

Аналитический характер функций $A(x)$ и $B(x)$ позволяет считать формулы (6.2) — (6.4) справедливыми, если $A(x)$ и $B(x)$ трактовать как значения функций A и B в точке x. Это сохраняющее операции взаимно однозначное соответствие между рядами, сходящимися в окрестности нуля, и функциями, аналитическими в окрестности нуля, позволяет отождествить формальный ряд (6.1) с определенной через него аналитической функцией.

Рассмотрим теперь простой пример.

Задача. Найти число T_N способов покрытия $2 \times N$ прямоугольника прямоугольными плитками размерами 2×1.

Решение. Поскольку все плитки покрытия идентичны, имеет смысл только ориентация плитки — вертикальная или горизонтальная. Например, существует 3 покрытия 2×3 прямоугольника:  .

Таким образом, $T_3 = 3$. Для $N = 2$ существует два покрытия:  . Следовательно, $T_2 = 2$.

В случае $N = 0$ существует единственный способ выбрать 0 объектов из N, а именно, не выбрать ничего. Говоря иначе, существует ровно один способ покрыть 2×0 прямоугольник — этот способ состоит в том, чтобы не класть ни одной плитки. Следовательно, $T_0 = 1$.

Рассмотрим теперь общий случай. Для любого $N > 2$ алгоритм покрытия выполняется в двух направлениях:

а) положить в начале прямоугольника $2 \times N$ одну вертикальную плитку и решить задачу покрытия меньшего размера $2 \times (N-1)$;

б) положить в начале две горизонтальные плитки и решить задачу размера $2 \times (N-2)$.

Таким образом, получаем зависимость $T_N = T_{N-1} + T_{N-2}$. Такая зависимость соответствует определению чисел Фибоначчи.

Числами Фибоначчи называется последовательность

$$f_0 = 1,\ f_1 = 1,\ f_k = f_{k-1} + f_{k-2}, k > 1. \tag{6.5}$$

Найдем производящую последовательность для чисел Фибоначчи. Построим производящую функцию

$$F(x) = \sum_{i=0}^{\infty} f_i x^i. \tag{6.6}$$

Подставим в (6.6) рекурсивную формулу (6.5) вычисления f_n :

$$\begin{array}{ll} F(x) & = \sum_{i=0}^{\infty} f_i x^i = \\ & = f_0 + f_1 x + \sum_{i=2}^{\infty} f_i x^i = \\ & = f_0 + f_1 x + \sum_{i=2}^{\infty} (f_{i-1} + f_{i-2}) x^i = \\ & = f_0 + f_1 x + \sum_{i=2}^{\infty} f_{i-1} x^i + \sum_{i=2}^{\infty} f_{i-2} x^i = \\ & = f_0 + f_1 x + x \sum_{i=1}^{\infty} f_i x^i + x^2 \sum_{i=0}^{\infty} f_i x^i = \\ & = f_0 + f_1 x + x(F(x) - f_0) + x^2 F(x). \end{array}$$

Из полученного уравнения

$$F(x) = f_0 + f_1 x + x(F(x) - f_0) + x^2 F(x)$$

или

$$F(x) = 1 + x + x(F(x) - 1) + x^2 F(x)$$

определяем $F(x)$:

$$F(x) = \frac{1}{1 - x - x^2}. \tag{6.7}$$

Для того, чтобы найти аналитическое выражение для f_k достаточно разложить (6.7) в степенной ряд и сопоставить коэффициенты при равных степенях переменной x. Сначала преобразуем выражение $1 - x - x^2$ по теореме Виета:

$$1 - x - x^2 = (1 - \alpha x)(1 - \beta x),$$

где

$$\alpha = \frac{1 - \sqrt{5}}{2}, \beta = \frac{1 + \sqrt{5}}{2}.$$

Используя метод неопределенных коэффициентов, получим

$$F(x) = \frac{1}{1 - x - x^2} = \frac{A}{1 - \alpha x} + \frac{B}{1 - \beta x}$$

и тогда

$$A = \frac{\alpha}{\alpha - \beta},\ B = \frac{\beta}{\alpha - \beta}.$$

Разложим в ряд Маклорена функцию $f(x)$, представляющую собой общий вид каждого из двух слагаемых в полученном выражении функции $F(x)$:

$$f(x) = \frac{1}{1 - \alpha x} = \sum_{k=0}^{\infty} \frac{x^k}{k!} f^{(k)}(0),$$

где

$$\begin{array}{l} f'(x) = \alpha(1 - \alpha x)^{-2}, \\ f''(x) = 2\alpha^2(1 - \alpha x)^{-3}, \\ \ldots \\ f^k(x) = k! \cdot \alpha^k (1 - \alpha x)^{-k-1}, \\ \ldots \end{array}$$

Тогда в точке $x = 0$ получаем

$$\begin{aligned} &f'(0) = \alpha, \\ &f''(x) = 2\alpha^2, \\ &... \\ &f^k(x) = k! \cdot \alpha^k, \\ &... \end{aligned}$$

При подстановке в ряд Маклорена полученных значений производных получаем

$$f(x) = \frac{1}{1 - \alpha x} = \sum_{k=0}^{\infty} \alpha^k x^k.$$

Окончательно имеем

$$F(x) = \frac{1}{1 - x - x^2} = \frac{A}{1 - \alpha x} + \frac{B}{1 - \beta x} = \sum_{k=0}^{\infty} \frac{\alpha^k - \beta^k}{\alpha - \beta} x^k.$$

Отсюда получаем общее выражение для f_k:

$$f_k = \frac{\alpha^k - \beta^k}{\alpha - \beta} = \left[\frac{1}{\sqrt{5}} ((\frac{1 + \sqrt{5}}{2})^{k+1} - (\frac{1 - \sqrt{5}}{2})^{k+1}) \right]. \tag{6.8}$$

Значения функций, вычисляемых по формулам (6.8) и (6.5), совпадают. Этот теоретически доказанный факт иллюстрирует следующая программа.

```
// Пример производящей последовательности для чисел Фибоначчи
#include <stdio.h>
#include <math.h>
#include <STDLIB.H>

int main(void)
{
long int a=1, b=1, d, r, i;
double d1, d2, s1, s2, p=5.0, frac;
frac=1.0/sqrt(p);
d1=(1+sqrt(p))/2;    d2=(1-sqrt(p))/2;
s1=d1*d1;            s2=d2*d2;
for(i=3; i<25; i++)
    {
    s1*=d1; s2*=s2;
    d=a+b; r=(long int)(frac*(s1-s2)+0.500001);
    printf("%ld   %ld\n",d,r);
    a=b; b=d;
    }
return 0;
}
```

Приведенная программа преследует только демонстрационные цели, показывая на практике совпадение значений функций (6.8) и (6.5). Реальное использование

этой формулы для вычисления значений чисел Фибоначчи на числах обычной длины неэффективно по следующим причинам. Вычисление степени либо все равно потребует цикла (как это сделано в приведенной выше программе), либо выполняется через функции логарифма и экспоненты (а это фактически приводит к вычислению рядов), либо использует стандартную функцию возведения в степень, которая также вычисляет степенную функциию через экспоненту и логарифм. Все это означает, что ускорения вычислений на числах обычной длины не получится. Только *при реализации длинной арифметики формула (6.8) окажется эффективной*, т.к. в этом случае существенно уменьшится число выполняемых операций сложения. При этом надо аккуратно работать с точностью вычисления рядов, чтобы полученное значение функции было точным на множестве целых чисел.

Приведенный пример показывает сам метод использования производящих функций, которые в более сложных случаях дают хороший инструмент сокращения вычислений. Здесь следует сделать одно очень важное замечание. Далеко не всегда требуется вывести явную формулу вычисления коэффициента, как мы это сделали для чисел Фибоначчи. В большинстве случаев достаточно получить рекуррентное соотношение между коэффициентами. В качестве упражнения предлагается решить следующую задачу.

Задача. Найти число способов заплатить N рублей монетками достоинством по 1, 5, 10, 50 копеек, а также рублевыми монетками — по 1 и 5 рублей.

Указание к решению: запишите бесконечные суммы, представляющие все возможные способы размена. Начать проще всего со случая, когда имеется меньше разновидностей монет, поэтому положите для начала, что у Вас нет никаких монет, кроме однокопеечных. Запишите производящую последовательность. Затем допустите, что кроме монет по одной копейке допускается использовать еще и пятикопеечные. Запишите производящую последовательность. Аналогично продолжайте действия при увеличении достоинства добавляющихся к разменной кассе монет.

В результате решения задачи у Вас должны получиться следующие рекурсивные зависимости между коэффициентами:

$K_n = K_{n-1} + 1$	способ выплаты монетками не старше 1 копейки
$P_n = P_{n-5} + K_n$	способ выплаты монетками не старше 5 копеек
$D_n = D_{n-10} + P_n$	способ выплаты монетками не старше 10 копеек
$Q_n = Q_{n-50} + D_n$	способ выплаты монетками не старше 50 копеек
$R_n = R_{n-100} + Q_n$	способ выплаты монетками не старше 1 рубля
$M_n = M_{n-500} + R_n$	способ выплаты монетками не старше 5 рублей

При решении задач с использованием производящих функций можно использовать таблицу 6.1 аналитических выражений для этих функций.

Таблица 6.1

Таблица простых последовательностей и их производящийх функций

последовательность	производящая функция	формула
$\langle 1,1,1,1,1,1,... \rangle$	$\sum_{i=0}^{\infty} x^i$	$\frac{1}{1-x}$
$\langle 1,-1,1,-1,1,-1,... \rangle$	$\sum_{i=0}^{\infty} (-1)^i x^i$	$\frac{1}{1+x}$
$\langle 1,0,1,0,1,0,... \rangle$	$\sum_{i=0}^{\infty} \left[\frac{2}{i}\right] x^i$	$\frac{1}{1-x^2}$
$\langle 1,0,...,0,1,0,...,0,1,... \rangle$	$\sum_{i=0}^{\infty} \left[\frac{m}{i}\right] x^i$	$\frac{1}{1-x^m}$
$\langle 1,2,3,4,5,... \rangle$	$\sum_{i=0}^{\infty} (i+1) x^i$	$\frac{1}{(1-x)^2}$
$\langle 1,2,4,8,16,... \rangle$	$\sum_{i=0}^{\infty} 2^i x^i$	$\frac{1}{1-2x}$
$\langle 1,k,C_2^k,C_3^k,C_4^k,... \rangle$	$\sum_{i=0}^{\infty} C_i^k x^i$	$(1+x)^k$
$\langle 1,k,k^2,k^3,k^4,... \rangle$	$\sum_{i=0}^{\infty} k^i x^i$	$\frac{1}{1-kx}$
$\langle 0,1,\frac{1}{2},\frac{1}{3},\frac{1}{4},\frac{1}{5},... \rangle$	$\sum_{i=1}^{\infty} \frac{1}{i} x^i$	$ln\frac{1}{1+x}$
$\langle 0,1,-\frac{1}{2},\frac{1}{3},-\frac{1}{4},\frac{1}{5},... \rangle$	$\sum_{i=1}^{\infty} \frac{(-1)^{i+1}}{i} x^i$	$ln(1+x)$

6.8 Контрольные вопросы к разделу

1. В каких случаях целесообразно применение рекурсивных алгоритмов?

2. Перечислите типы данных, имеющих рекурсивную природу. Предложите на языке Си или Паскаль описание соответствующих типов.

3. Что представляет собой метод "разделяй и властвуй"?

4. Что означает принцип балансировки при использовании метода "разделяй и властвуй"?

5. Как устранить рекурсию в случае, когда имеется схема примитивной рекурсии?

6. Как устранить рекурсию в общем случае?

7. Что означает термин "полный перебор"?

8. Какие цели преследует отсечение вариантов?

9. Зачем используется функция цели в методе отсечения?

10. Приведите пример использования функции цели для отсечения вариантов.

11. Напишите программу игры в "крестики-нолики" с отсечением вариантов перебора.

12. Зачем используется динамическое программирование?

13. Поясните принцип динамического программирования.

14. Приведите пример программы, в которой используется принцип динамического программирования.

15. Что такое виртуальный граф и зачем он используется?

16. Приведите пример программы, в которой используется виртуальный граф.

17. Вычислите временную сложность алгоритмы Дейкстры поиска в графе минимального пути от заданной вершины.

18. Какие данные хранятся в стеке при рекурсивном вызове функции?

19. Напишите нерекурсивную программу вычисления функции Аккермана.

20. Приведите пример рекурсивной функции, схема рекурсии которой не является примитивно–рекурсивной.

6.9 Упражнения к разделу

Задание.

Написать программу решения поставленной задачи. При выполнении задания проанализируйте постановку задачи. Не все приведенные в заданиях задачи являются NP–полными и не все — P– задачами. Если в постановке задачи сразу указана размерность, ее величина может дать некоторую подсказку для поиска алгоритма решения. Небольшая величина размерности, как правило, позволяет решить задачу полным перебором, возможно, с некоторым отсечением вариантов. Большие порядки размерности означают, что задачу можно решить только с помощью полиномиальных алгоритмов. В разработке таких алгоритмов хорошую помощь должны оказать рассмотренные в главе методы проектирования эффективных алгоритмов.

Часть заданий сформулирована без указания порядков размеров задачи. В этом случае Вам необходимо рассмотреть максимально эффективные с Вашей точки зрения алгоритмы и сформировать требования к размерности задачи с тем, чтобы задача могла быть решена на реальной памяти компьютера за разумное время.

6.9.1 Задача

Рассмотрим следующую задачу.[4]

Для срочной покупки квартиры мистер Смит получил в банке заем, эквивалентный Q долларам. Он должен выплатить свой долг в течении K лет, выплачивая каждый год еще и проценты P. Это значит, что к концу каждого года долг мистера Смита вырастает на $P * Q_1/100$ (Q_1 – долг на начало года) и его текущие выплаты вычитаются из общей суммы с учетом процентов.

В первый год мистер Смит хочет заплатить такую минимальную сумму, которая позволит ему выплатить весь долг точно за K лет. В каждый последующий год он хочет выплачивать или точно такую же сумму, как в предшествующий год, либо на один цент меньше. Кроме того, он хочет закончить выплачивать долг без переплаты даже одного цента к концу K-го года.

Банк выполняет начисление процентов с точностью в один цент и производит начисление процентов в конце каждого года. При вычислении процентов результат немедленно округляется до ближайшего цента, причем 0.5 центов округлается до 1 цента.

Ограничения на данные. $10 \leq Q \leq 1000000, 0 \leq P \leq 100, 1 \leq K \leq 100$.

Входные данные. Входной файл содержит три числа Q, P и K.

Результаты. Выходной файл содержит выплаты. Если решение невозможно, выходной файл должен содержать единственную строку с текстом "Impossible". В противном случае требуется вывести выплачиваемую сумму и число лет, в течение которых эта сумма выплачивается, в виде строки: "$X for Y year(s)"

Решение.

Рассмотрим рекурсивную схему решения задачи. Пусть функция $Solve(Q, M, i)$ решает задачу распределения выплат и выполняет следующие действия:

а) возвращает значение 1, если при условии выплаты в текущем году M центов на последующих шагах задача успешно решается (здесь i - число лет до окончания срока всех выплат, Q - общая сумма к выплате);

б) возвращает 0, если решение невозможно.

Если известно, что за i-ый год выплатили M центов, то по условию задачи в новом $(i-1)$-ом году можно выплатить M или $M-1$ центов. Таким образом, достаточно вычислить новое значение общей суммы к выплате и проверить два варианта выплат M и $M-1$ центов. Сначала надо вычислить значение нового долга по формуле

$$percent = 1 + P/100;$$

$$NewQ = percent * Q - M + 0.5001.$$

Сразу следует отметить, что эта формула является абсолютно неправильным с точки зрения языка C++ оператором, т.к. в ней не учтено представление разных типов данных в памяти ЭВМ и операции приведения типов (смотрите текст программы, где указаны соответствующий операторы). После того, как начислили проценты и заплатили в данном году, делаем попытку дальнейшего решения:

```
if (Solve(NewQ, M-1, year-1) ||  Solve(NewQ, M, year-1))  return 1;
```

Обратите особое внимание на порядок вызова функции

$$Solve(...M-1...) \text{ и } Solve(...M...)$$

[4] 1997-1998 ACM Northeastern EuropeanRegionalProgramming Contest

для получения минимальной выплаты в каждом году, а не только в первом году.

Решение будет достигнуто при одновременном выполнении условий $(Q == 0)$ и $(i == 0)$, т.е.

```
if ( Q == 0 )  return ( i == 0 );
if ( i == 0 )  return ( Q == 0 );
```

Анализ временной сложности указанного алгоритма приводит к определению порядка сложности $O(2^{100})$, что говорит о необходимости реализации отсечения. Чтобы отсечь на ранних шагах решения неверные варианты выплат, необходимо на каждом шаге контролировать возможность получения решения на последующих шагах. Для этой цели рассмотрим минимально возможное значение выплаты в текущем (первом от начала выплат) году, если все выплаты совершаются за K лет. Минимальное значение выплат в текущем году получается при условии, что на последующих годах нет снижения выплат. Введем обозначения: Q — общая сумма к выплате на начало года, p — коэффициент роста $1 + P/100$. Тогда условию задачи соответствует следующее уравнение для вычисления минимальной суммы выплат a:

$$\underbrace{(...(((Q * p - a) * p - a) * p - a) * ...) * p - a}_{K} = 0$$

или

$$Q * p^K - a * (p^{K-1} + p^{K-2} + ... + p^2 + p + 1) = 0.$$

Отсюда

$$\frac{1}{a} = \frac{p^{K-1} + p^{K-2} + ... + p^2 + p + 1}{Q * p^K}.$$

Или

$$\frac{1}{a} = \frac{1}{Q} \cdot (\frac{1}{p} + \frac{1}{p^2} + \frac{1}{p^3} + ... + \frac{1}{p^K}).$$

Получаем

$$a = Q/(\frac{1}{p} + \frac{1}{p^2} + \frac{1}{p^3} + \cdots + \frac{1}{p^K}).$$

Аналогично надо найти условие максимальной выплаты. В первый год придется выплатить максимальное значение b, если выплаты будут снижаться на всех последующих годах. Тогда получаем следующее уравнение

$$(...(((Q \cdot p - b) \cdot p - b + 1) \cdot p - b + 2) \cdot ...) \cdot p - b + (k - 1) = 0$$

или

$$Qp^K - b \cdot (p^{K-1} + p^{K-2} + ... + p^2 + p + 1) + 1 \cdot p^{k-2} + 2 \cdot p^{k-3} + 3 \cdot p^{k-4} + \cdots + (k-2) \cdot p^1 + (k-1) \cdot p^0 = 0.$$

Получаем максимальное значение первой выплаты

$$b = a + \frac{(1 * p^{k-2} + 2 * p^{k-3} + 3 * p^{k-4} + \cdots + (k-2) * p^1 + (k-1) * p^0)}{(p^{K-1} + p^{K-2} + \cdots + p^2 + p + 1)}.$$

Замечание. Значения сумм приходится вычислять на каждом шаге рекурсии, поэтому можно сформировать вспомогательные массивы числителей и знаменателей перед началом вычислений. При данных по условию ограничениях такие массивы можно не использовать, т.к. задача решается за достаточно разумное время.

Приводим текст программы решения поставленной задачи.

```
#include <STDLIB.H>
#define  MAXK  100

long int Q;
int  P, K; // данные: начальная сумма, проценты и число лет
double percent;  // 1+P/100 - рост долга
FILE *in = fopen("loan.in","r");
FILE *out= fopen("loan.out","w");
long int  payment[MAXK+1];  // выплаты по годам

void GetInterval(long int Q,int year, long int * Min, long int *Max)
// расчет границ возможных выплат [Min,Max] в текущем году
// Q - исходная сумма, year - число лет
{
int i;
double a,b,c,d;
if (year==0) { *Min=*Max=Q; return; }
if (P==0) { (*Max)=(*Min)=Q/year;
          if ((*Min)*year<Q) *Max=(*Min)+1; return;
          }
a=0;  b=1; c=0; d=0;
for (i=0; i<year; i++)
    {
    c+=(year-i-1)*b;   d+=b;
    b=b/percent;
    a+=b;
    }
*Min= Q/a+0.0001;
if ( (*Min)==0 ) *Min=1;
// вычисляем максимальное значение выплаты в первый год
*Max=(*Min) + c/d + 0.5001 +1;// +1 на всякий случай из-за округления 0.5
}

int Solve(long int Q, long int M, int year) // в данном году платим M
{
if ( Q == 0)    return ( year==0 );
if (year == 0)  return (Q == 0); // все время истекло
if ( (Q < 0) || (M <= 0) )    return 0;
long int  MaxNew=M, MNew=M, NewQ;
NewQ=(double)(percent * (double)(Q) + 0.5001) ; //начислили проценты
NewQ=  NewQ - M ; // заплатили в данном году
if (NewQ!=0) GetInterval(NewQ, year-1, &MNew, &MaxNew);
if ((M < MNew) || (M-1 > MaxNew))    return 0;
if (Solve(NewQ, M-1, year-1) ||  Solve(NewQ, M, year-1))
    { payment[year] = M; return 1; }
return 0;
}

int main(void)
{
```

```
double  QFloat;    // float - эквивалент переменной Q
fscanf(in,"%lf %d %d",&QFloat, &P, &K);
Q= QFloat * 100 + 0.0001; // в центах исходная сумма
percent=1+((double)P)/100;
long int Min, Max; // минимальная граница выплат и максимальная на K лет
GetInterval(Q, K, &Min, &Max);
int years, num;
long int i;   // текущее значение минимальной выплаты в первый год
for (i= Min; i<=Max; i++) // попытка выплатить за первый год не более i:
   if (Solve( Q, i, K))
      {
      years= 0;       // число лет одинаковых выплат
      payment[0]= -1; // для организации сравнения на последнем шаге
      for (num=K; num > 0; num--)  //num - общее число лет для выплат
          {
          years++;
          if (payment[num-1] != payment[num])
             {
             fprintf(out,"$%.2lf for %d year(s)\n",
                    (double)(payment[num])/100, years);
             years = 0;
             }
          }
      fclose(in);fclose(out);
      return 0;
      }
fprintf(out,"Impossible\n");
fclose(in);fclose(out);
return 0;
}
```

6.9.2 Варианты заданий

1. "Цепь". Дано множество слов одинаковой длины, первые два слова из них выделены. Построить цепь минимальной длины от первого выделенного слова ко второму так, чтобы все слова этой цепи были только из заданного множества. Соседние слова построенной цепи должны отличаться только одной буквой. Число слов не более 1000, длина каждого слова не превышает 10 символов.

Например, если дано множество
вол, сыр, воз, сор, вор, тор, рот, сон,
то можно построить цепь
вол $\rightarrow$ вор $\rightarrow$ сор $\rightarrow$ сыр.

2. "Кроссворд". Дано множество слов. Построить из них кроссворд заданной конфигурации (число слов может быть больше требуемого количества для заполнения кроссворда). Конфигурация кроссворда задается полем $M \times N$, где M – число строк

и N – число столбцов поля кроссворда. Знак "." на этом поле означает клетку, предназначенную для записи буквы, знак "*" означает неиспользуемую для записи букв клетку. Ограничения: $M \le 5, N \le 7$, число заданных слов не превышает 100.

3. "Муха на кубике". Муха ползет по кубику размерами $N \times N \times N$ сантиметров. Грани кубика разбиты на клетки со стороной в один сантиметр. Каждая такая клетка выкрашена в черный или белый цвет. Переход мухи из клетки в клетку допускается только через общую сторону при условии совпадения цветов этих клеток. Найти кратчайший маршрут мухи из одной заданной клетки в другую заданную клетку этого же цвета. Если существует несколько таких маршрутов, вывести любой из них. Если решение невозможно, вывести соответствующее сообщение.

Заданы начальное и конечное положение мухи, число $N(N \le 10)$ и раскраска граней кубика. Кубик задается разверткой. На развертке черная клетка задается буквой "B белая – буквой "W". Клетка, на которой находится муха, а также целевая клетка также нанесены на развертку. Если муха находится на белой клетке, то она обозначается символом "0 если на черной – символом "1". Целевая клетка всегда того же цвета, что и занятая мухой клетка, и обозначается символом "*".

Развертка содержится в исходном файле, каждая строка и каждая позиция строки которого соответствует 1 сантиметру (или одной клетке на поверхности кубика).

Пример задания развертки в файле исходных данных для $N = 2$:

```
. . . . . . . . . . . . . .
. . . . . B W . . . . . .
. . . . . W B . . . . . .
. . . W B W B W B W B . . . .
. . . 1 B W B B * B B . . . .
. . . . . W B . . . . . .
. . . . . B B . . . . . .
. . . . . . . . . . . . . .
```

4. "Цифровая пирамида". Дана последовательность целых чисел

$$a = (a_1, a_2, \ldots, a_n),\ n < 100,$$

являющаяся основанием целочисленной пирамиды. Пирамида строится снизу вверх. Каждый новый ряд получается как разность или частное соседних элементов предыдущего ряда, т.е. все элементы нового ряда равны

$$b_i = a_{i+1} - a_i$$

или все элементы этого ряда равны

$$b_i = \frac{a_{i+1}}{a_i}.$$

Равенство всех чисел на некотором уровне пирамиды — признак конца построения этой пирамиды. Найти очередной элемент a_{n+1}, продолжающий заданную последовательность $a = (a_1, a_2, \ldots, a_n)$ так, чтобы пирамида кончалась на том же уровне. Одновременно с вычислением a_{n+1} напечатать формулу его вычисления.

Пример.

```
      1     1     1          → 1   операция вычитания
   2     3     4     5       → 6    операция деления
1     2     6     24    120  → 720
```

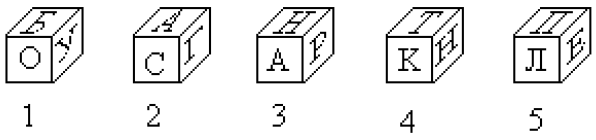


Рис. 6.10: Из кубиков 2, 4, 1 и 5 можно построить слово "стол".

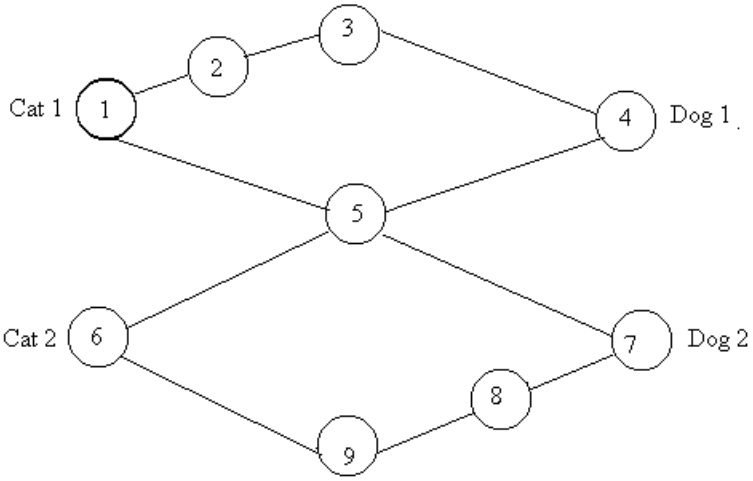


Рис. 6.11: Пример исходных данных к задаче 7.

Пример сформированной формулы вычисления требуемого значения:

$$a_6 = a_5 * (a_5/a_4 + a_4/a_3 - a_3/a_2)).$$

5. "Слова". Имеется K кубиков, $K \leq 10$. На всех гранях каждого кубика находится по одной букве. Можно ли выстроить некоторые из этих кубиков в ряд так, чтобы получилось заданное слово? Кубики можно переворачивать.

В примере, представленном на рис 6.10. слово "СТОЛ" можно выстроить из кубиков 2-4-1-5.

6. "Запасные аэродромы". Из–за нелетной погоды необходимо посадить находящиеся в воздухе над аэропортом самолеты разных классов на несколько запасных аэродромов. Каждый из запасных аэродромов в состоянии принять K_i самолетов класса не выше P_i (i — номер запасного аэродрома). Посадить самолеты при условии, что для каждого самолета известно расстояние, которое он может пролететь на имеющемся горючем.

7. "Кошки и собаки". Парк состоит из площадок, некоторые из которых соединены дорожками. N кошек и N собак оказались на противоположных площадках парка и должны поменяться местами, но так, чтобы не только не встречаться друг с другом, но даже не оказываться на соседних площадках. В любой момент времени только одно животное может передвигаться. Найти порядок движения.

Пример представлен на рис. 6.11. Последовательность передвижений, которая приведет к решению задачи: $Cat_1 : 1 \to 2$, $Cat_2 : 6 \to 9$, $Dog_2 : 7 \to 5$, $Cat_2 : 9 \to 8$, $Dog_2 : 5 \to 6$, $Dog_1 : 4 \to 5$, $Cat_1 : 2 \to 3$, $Dog_2 : 5 \to 1$, $Cat_2 : 8 \to 7$, $Cat_1 : 3 \to 4$.

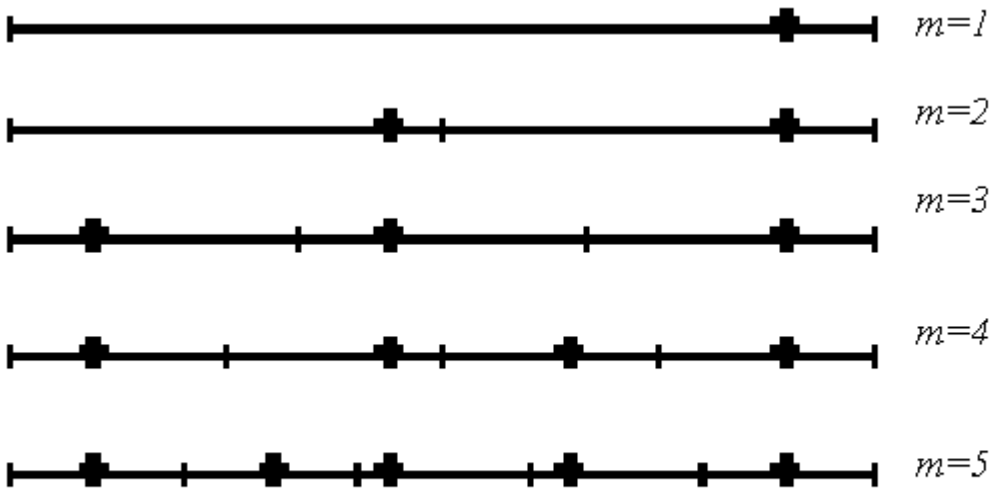


Рис. 6.12: Падающие на отрезок снежинки к задаче 8.

8. "Снежинки". Дан отрезок $[0;1]$. На этот отрезок поочередно падают снежинки. Необходимо последовательно расставить на отрезке N снежинок так, чтобы после нанесения m-ой снежинки ($m = 1, 2, \ldots, N$) в каждом интервале (x_{i+1}, x_i) находилось в точности по одной снежинке, где $i = 1, 2, \ldots, m; x_0 = 0, x_m = 1$, $x_{i+1} - x_i = 1/m$.

Пример для $N = 5$ представлен на рис. 6.12.

9. "Салфетки". Имеется клетчатая квадратная скатерть $N \times N$ клеток. Разрезать ее на M квадратных салфеток так, чтобы не нарушить целостность клеток.

Пример. Для $N = 5, M = 8$ имеется решение, один из вариантов которого приведен на следующем рисунке:

1	1	1	2	2
1	1	1	2	2
1	1	1	3	3
4	5	5	3	3
6	5	5	7	8

10. "Собака и заяц". Заяц, убегая от собаки, подбежал к берегу озера с большим количеством кочек. Перескакивая с кочки на кочку, заяц стремится попасть на противоположную сторону озера, т.к. тогда он успеет убежать. Собака подбегает к озеру и бросается вплавь, направляясь в каждый момент времени на ту кочку, на которой находится заяц в данный момент. Успеет ли заяц убежать от собаки?

Известны координаты точек X[i], Y[i], причем на первой кочке заяц оказался в тот момент, когда собака подбежала к берегу озера. Известно время t, которое заяц сидит на каждой кочке, готовясь к прыжку, а также скорость V, с которой плывет собака. Длина максимально возможного прыжка зайца также известна.

11. "Ремонт". Имеются кафельные плитки, каждая сторона каждой плитки покрашена в один из k цветов. Можно ли M имеющимися плитками выстелить стену заданной конфигурации, занимающей часть поля $N \times N$, так, чтобы грани соседних плиток были одного цвета?

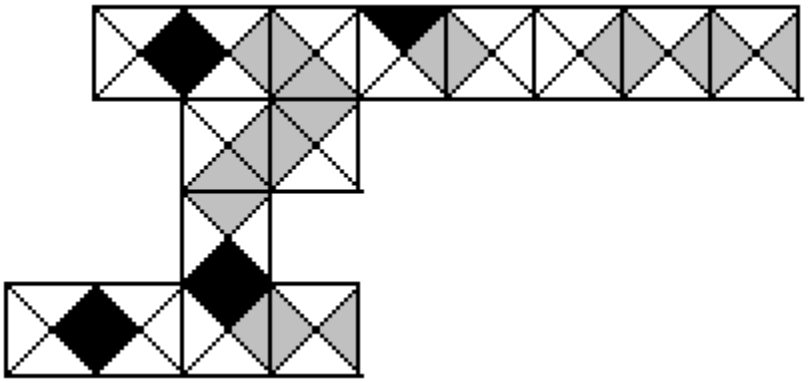


Рис. 6.13: Пример укладки кафельных плиток к задаче 11.

Пример представлен на рис. 6.13.

12. "Головоломка". Есть N кубиков и N красок. Все грани кубиков окрашены, причем каждая из шести граней одного кубика может быть окрашена в любой цвет из заданных. У каждого кубика допускается как наличие нескольких граней одного цвета, так и все грани могут быть окрашены в разные цвета. Можно ли так поставить кубики в столбик, чтобы каждый цвет появлялся ровно один раз на каждой из четырех сторон столбика?

13. "Студенты". Для выполнения лабораторной работы группу из $3 \cdot N$ студентов надо разбить на подгруппы по три человека в каждой. Чтобы сформировать работоспособные коллективы, преподаватель попросил каждого студента назвать фамилии трех студентов, с которыми он хотел бы работать. Сформировать подгруппы таким образом, чтобы в каждой подгруппе студенты были объединены в соответствии с их пожеланиями, или выдать сообщение об отсутствии решения.

14. "Перестановки". Напишите алгоритм нахождения всех перестановок целых чисел от 1 до N.

Указание: Множество перестановок целых чисел от 1 до N можно получить из множества перестановок целых чисел от 1 до $N-1$, вставляя N во все возможные позиции в каждой перестановке длины $N-1$.

15."Завод — автомат". На суперсовременном заводе по производству компьютеров все процессы полностью автоматизированы. Компьютеры автоматически собираются, запаковываются в коробки и поступают на выходную ленту транспортера. На заводе имеется N линий автоматической сборки, $N \leq 100$. Производительность линии l_i — от 1 до 60 компьютеров в час. На изготовление одного компьютера всегда требуется целое число минут. К сожалению, производственные линии иногда ломаются и их приходится ремонтировать. Для контроля работоспособности завода поставили регистрирующий автомат. Регистрирующий автомат стоит на выходе и отмечает время поступления на транспортер каждой очередной коробки с компьютером. Данные измеряются один раз в сутки в течении часа. По этим данным регистрирующий автомат определяет M — минимальное число работающих линий и число минут, которое требуется каждой автоматизированной линии на сборку одного компьютера. Напишите программу для регистрирующего автомата.

Исходные данные. В первой строке входного файла находится целое число K — число измерений за один час. Вторая строка содержит K целых чисел, каждое из

которых находится в пределах от 0 до 59 и представляет собой время появления на транспортере очередной коробки.

Результаты работы записываются в выходной файл и содержат целое число M — минимальное число работающих линий. Для каждой линии в файл вывести целое число — время на изготовление одного компьютера.

15."Полиамино". [5]

Полиамино - это плоская фигура, состоящая из квадратов, прилегающих друг к другу сторонами, так, что за несколько ходов можно дойти до любой его клетки шахматной ладьей (двигаясь по горизонтали и вертикали). На бесконечном, разбитом на клетки листе бумаги отмечены N^2 клеток ($2 \le N \le 5$), образующие полиамино. Вам надо написать программу, делящую заданное полиамино на два других, из которых затем, используя параллельный перенос и поворот, можно получить квадрат $N \times N$. Требуется найти только одно решение.

Так как невозможно задать бесконечный лист, то исходный файл содержит лишь часть его. Не указанные в исходном файле клетки считать непомеченными. В исходном файле знак "." означает непомеченную клетку, а знак "*" — помеченную. В строке не более 100 символов, в файле не более 100 строк.

В выходной файл Ваша программа должна записать аналогичный файл, но вместо символов "*" должны стоять буквы "A" и "B соответствующие разбиению.

Пример входных данных

```
. . . . . . . . .
. . * * * . . . .
. . * . * . . . .
. . * * * * * . .
. . * * * * . . .
. . . . * * . . .
. . . . . . . . .
. . . . . . . . .
```

Пример выходных данных

```
. . . . . . . . .
. . A A A . . . .
. . A . A . . . .
. . A B B B B . .
. . A A A B . . .
. . . . B B . . .
. . . . . . . . .
. . . . . . . . .
```

16. "Новоселье Бабы Яги". Баба Яга решила переехать в новую избушку на курьих ножках. Она упаковала все свои пожитки в N коробков и узелков. Вес i-го багажа равен $a[i]$ килограммов. Личный транспорт Бабы Яги известен – это ступа грузоподъемностью S килограммав. Сможет ли Баба Яга перевезти все свое добро не более, чем за K рейсов? Если сможет, Ваша программа должна дать один из способов такой перевозки. Значения S и $a[i]$ не превышают 3000. Значения N и K не превышают 20.

[5] 1997-1998 ACM Northeastern European Regional Programming Contest

17. "Домино". Дано множество косточек домино. Составить из них кольцо или выдать сообщение об отсутствии решения.

Исходные данные: число косточек домино и для каждой косточки — два числа, соответствующие этой косточке.

18. "Оборудование". В цехе требуется расставить оборудование. План цеха представляет собой прямоугольник заданных размеров. На плане цеха отмечен пожарный проход и ворота. Пожарный проход представляет собой свободный от оборудования коридор на ширину ворот и проходит от ворот до противоположной стены цеха. Ворота расположены в середине узкой стены цеха, ширина ворот задана. Каждая единица оборудования представляет собой прямоугольник заданных размеров. Эти размеры уже содержат дополнительные припуски для прохода рабочих, поэтому прямоугольники оборудования можно ставить вплотную друг к другу. Все размеры заданы в метрах с точностью до одного знака после запятой.

19. "Расписание". Дано множество задач, которые надо решить с помощью компьютера. Решение всех этих задач в совокупности занимает много времени, поэтому возникает проблема составления расписания их решения. Имеется M компьютеров, на которых можно решать эти задачи. Время решения каждой i-ой задачи известно и составляет $t\,[i]$. Для каждой задачи есть контрольный срок, к которому она должна быть решена. Задан частичный порядок на множестве задач, согласно которому каждая задача имеет не более одного предшественника. Составить расписание для M компьютеров, так, чтобы все контрольные сроки были выполнены.

20. "Крайний Север". Как известно, на Крайнем Севере почту доставляют на вертолете в некоторые населенные пункты, а оттуда развозят на собаках по всем маленьким поселкам. Между некоторыми поселками часто ездят на собачьих упряжках, поэтому там проложены хорошо наезженные колеи, по которым легко бежать собакам. Найти минимальное число населенных пунктов, в которые нужно доставить почту на вертолете с тем, чтобы из них можно было развезти почту по оставшимся поселкам. Доставка должна проходить только по наезженным дорогам, причем из каждого пункта, в который доставили почту на вертолете, упряжка должна развезти почту и вернуться обратно, ни разу не проезжая дважны по одной и той же колее.

Исходная информация: число населенных пунктов, число дорог между ними и для каждой дороги указаны номера населенных пунктов, которые она соединяет.

21. "Укрепленный лес". [6] В двлекой стране жил–был король, к которого была маленькая коллекция редких и ценных деревьев. Для защиты деревьев от воров король приказал построить вокруг них высокий забор. Дело было поручено волшебнику. Волшебник очень быстро обнаружил, что единственный материал для постройки забора — это древесина самих деревьев. Другими словами, необходимо срубить некоторые деревья, чтобы построить забор вокруг остальных. Чтобы предотвратить срубание своей собственной головы, волшебник должен минимизировать ценность деревьев, которые должны быть срублены. Вам требуется написать программу для решения задачи, с которой столкнулся волшебник.

Исходные данные: $N, (N \leq 15)$ — число деревьев, для каждого из которых задается четыре целых числа x, y, v, l. Пара чисел (x, y) задает положение дерева на плоскости, число v — стоимость дерева, l — длина забора, который можно построить из этого дерева.

[6] 1998-1999 ACM Final Programming Contest

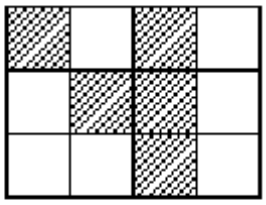


Рис. 6.14: Пример формы компостера к задаче 22.

22."Компостер". [7]

На фабрике есть прямоугольная форма с M рядами и N столбцами для производства компостеров с 1, 2, ... $M \times N$ иглами. Будем считать, что при пробивании прямоугольного билета две его стороны параллельны столбцам игл компостера, а две – параллельны рядам (см. рис. 6.14).

Количество дырок на билете равно числу игл компостера. Следующие действия с билетом возможны при его компостировании:

а) билет может быть пробит с любой стороны;

б) билет может быть вставлен в компостер любым краем (поворот на 90, 180, 270 градусов);

в) возможен параллллельный перенос.

Ваша задача – определить, сколько различных компостеров можно сделать при помощи формы $M \times N$. Два компостера различны, если полученные узоры на билетах нельзя совместить при помощи описанных выше преобразований. Ограничения: $M \leq 4$, $N \leq 8$.

Пример входных данных

3 3

Пример выходных данных

85

6.10 Тесты для самоконтроля к разделу

1. Укажите все, что позволяет метод динамического программирования из нижеперечисленного:

1) уменьшить длину программы;

2) уменьшить время работы алгоритма;

3) уменьшить объем используемых данных.

Правильный ответ: только 2.

2. При использовании метода "разделяй и властвуй" наиболее эффективным является следующее разбиение задачи на части:

1) один элемент и все оставшиеся;

2) на две равные части;

[7] 1997-1998 ACM Northeastern European Regional Programming Contest

3) в зависимости от условия задачи;

4) разбиение задачи на части всегда неэффективно, задачу надо решать всю целиком.

Правильный ответ: 2.

3. Алгоритм Дейкстры поиска кратчайшего пути в графе имеет временную сложность

1) $O(n)$;
2) $O(n^2)$;
3) $O(n^3)$;
4) $O(2^n)$;
5) $O(n!)$.

Правильный ответ: 2.

4. Какие из следующих утверждений истинны?

1) Если программе соответствует схема примитивной рекурсии, то эффективнее использовать рекурсивную программу.

2) Для любой рекурсивной программы с помощью стека можно построить эквивалетную нерекурсивную программу независимо от применяемой в алгоритме рекурсивной схемы.

3) Для любой рекурсивной программы нерекурсивный эвивалент имеет тот же порядок временной сложности.

Правильный ответ: 2 и 3.

5. Какая из следующих задач на графах решается за полиномиальное время:

а) поиск гамильтонова цикла в графе;

б) раскрашиваемость графа;

в) стягиваемость графа;

г) максимальное паросочетание в двудольном графе;

д) множество дуг, разрезающих циклы.

Правильный ответ: г.

Глава 7

ФОРМАЛЬНЫЕ ГРАММАТИКИ И ЯЗЫКИ

7.1 Понятие порождающей грамматики и языка

Прежде, чем рассматривать формальное определение языка и грамматики, рассмотрим такое описание на интуитивном уровне. Язык можно определить как некоторое множество предложений заданной структуры, имеющих, как правило, некоторое значение или смысл. Правила, определяющие допустимые конструкции языка, составляют *синтаксис языка*. Значение (или смысл) фразы определяется *семантикой языка*. Например, по правилам синтаксиса языка Си фраза $(X*2)$ является правильным выражением, в отличие от фразы $(2X*)$. Семантика языка Си определяет, что фраза

$$for(i = 0; i < 10; i++)S[i] += A[i];$$

является оператором цикла, в котором переменная i последовательно принимает значения 0, 1,...,9.

Если бы все языки состояли из конечного числа предложений, то не ставилась бы и проблема описания синтаксиса, т.к. достаточно просто перечислить все допустимые предложения конечного языка — и язык задан. Но почти все языки содержат неограниченное (или очень большое) число правильно построенных фраз, поэтому возникает *проблема описания бесконечных языков с помощью конечных средств*. Различают порождающее и распознающее описание языка. *Порождающее описание языка* означает наличие алгоритма, который последовательно порождает все правильные предложения этого языка. Любая строка принадлежит языку тогда и только тогда, когда она появляется среди генерируемых строк. *Распознающее описание языка* означает наличие алгоритма, который для любой фразы может определить, принадлежит эта фраза языку или нет. Средством порождающего задания языка являются *грамматики.*

Рассмотрим основные понятия, связанные с языками и порождающими грамматиками. Часть из них уже рассматривалась в главе 2.

Алфавит — непустое конечное множество. Элементы алфавита называются символами. Цепочка над алфавитом $\Sigma = \{a_1, a_2, ..., a_n\}$ есть конечная последовательность элементов a_i. Длина цепочки x — число ее элементов, обозначается $|x|$. Цепочка нулевой длины называется пустой цепочкой и обычно обозначается ε. Непустой называется цепочка ненулевой длины.

Цепочки x и y равны, если они одинаковой длины и совпадают с точностью до порядка символов, из которых состоят, т.е. если $x = a_1a_2...a_n$, $y = b_1b_2...b_k$, то $n = k$

и для всех $1 \leq i \leq k$ справедливо равенство $a_i = b_i$.

Пусть $x = a_1a_2...a_n$ и $y = b_1b_2...b_k$ — некоторые цепочки. Конкатенацией (или сцеплением, или произведением) цепочек x и y называется цепочка $xy = a_1a_2...a_nb_1b_2...b_k$, полученная дописыванием символов цепочки y вслед за символами цепочки x. Например, если $x = cca, y = abba$, то $xy = ccaabba$. Поскольку ε — цепочка нулевой длины, то в соответствии с определением конкатенации можно написать $\varepsilon x = x\varepsilon = x$.

Множество всех цепочек (включая пустую цепочку ε) над алфавитом Σ обозначается через Σ^* . Например, $\Sigma = \{a, b\}$, тогда $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, ...\}$. В дальнейшем мы увидим, почему принято обозначение Σ^*.

Язык L над алфавитом Σ есть некоторое множество цепочек над этим алфавитом, т.е. $L \subseteq \Sigma^*$. Необходимо различать пустой язык $L = \emptyset$ и язык, содержащий только пустую цепочку $L = \{\varepsilon\} \neq \emptyset$. Формальный язык L над алфавитом Σ — это язык, выделенный с помощью конечного множества некоторых формальных правил.

Пусть L и M — языки над алфавитом . Произведение языков есть множество $LM = \{xy | x \in L, y \in M\}$. В частности, $\{\varepsilon\}L = L\{\varepsilon\} = L$. Используя понятие произведения, определим итерацию L^* и усеченную итерацию L^+ множества L:

$$L^+ = \bigcup_{i=1}^{\infty} L^i,$$

$$L^* = \bigcup_{i=0}^{\infty} L^i,$$

где степени языка L можно рекурсивно определить следующим образом:

$$L^0 = \{\varepsilon\},\ L^1 = L,\ L^{n+1} = L^nL.$$

Например, пусть $L = \{a\}$, тогда

$$L^* = \{\varepsilon, a, aa, aaa, \dots\},$$

$$L^+ = \{a, aa, aaa, ...\}\ .$$

Определение 7.1. Порождающей грамматикой называется упорядоченная четверка

$$G = (V_T, V_N, P, S),\ \text{где}$$

V_T — конечный алфавит, определяющий множество терминальных символов;

V_N — конечный алфавит, опредедяющий множество нетерминальных символов;

P — конечное множество правил вывода — множество пар вида $u \to v$, где $u, v \in (V_T \cup V_N)^*$;

S — начальный нетерминальный символ — аксиома грамматики, $S \in V_N$.

Определение 7.2. В грамматике $G = (V_T, V_N, P, S)$ цепочка x непосредственно порождает цепочку y, если $x = \alpha u\beta$, $y = \alpha v\beta$ и $u \to v$ является правилом грамматики G, т.е. $u \to v \in P$. Говорят также, что y непосредственно выводится из x. Непосредственная выводимость y из x обозначается $x \Rightarrow y$.

Определение 7.3. В грамматике G цепочка y выводима из цепочки x, если существуют цепочки $x_0, x_1, .., x_k$ такие, что $x = x_0, y = x_k$ и для всех i ($1 \leq i \leq k$) $x_{i-1} \Rightarrow x_i$, т.е.

$$x = x_0 \Rightarrow x_1 \Rightarrow \dots \Rightarrow x_k = y.$$

На каждом шаге вывода применяется одно правило грамматики. Число шагов в выводе цепочки y из цепочки x называется длиной вывода. Выводимость обозначается

$$x \overset{*}{\Rightarrow} y.$$

Определение 7.4. Языком, порождаемым грамматикой $G = (V_T, V_N, P, S)$, называется множество терминальных цепочек, выводимых в грамматике G из аксиомы:

$$L(G) = \{x | x \in V_T^*; S \stackrel{*}{\Rightarrow} x\}.$$

Пример 7.1. Пусть $G = (V_T, V_N, P, S)$, где $V_T = \{a, b\}, V_N = \{S\}, P = \{S \to aSb, S \to ab\}$. Тогда в грамматике существуют выводы

$$S \Rightarrow ab,$$
$$S \Rightarrow aSb \Rightarrow aabb,$$
$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb,$$
$$....$$

Таким образом, $L(G) = \{a^n b^n | n > 0\}$.

Если множество правил приводится без специального указания множества нетерминалов и терминалов, то обычно предполагается, что грамматика содержит в точности те терминалы и нетерминалы, которые встречаются в правилах. Предполагается также, что правые части правил, левые части которых совпадают, можно записывать в одну строку с вертикальной чертой "| " в качестве разделителя. Тогда грамматику, рассмотренную в примере 7.1, можно задать следующим образом:

$$G : S \to aSb|ab.$$

Нетрудно видеть, что терминальные символы грамматики — это такие символы, из которых состоят цепочки языка, порождаемого грамматикой. В языках программирования терминалами являются фактически используемые в них слова и символы, такие, как for, $+$, $float$ и т.д. Нетерминальные символы являются вспомогательными символами, используемыми только в процессе вывода и не входящими в цепочки языка. Обычно нетерминалы предназначены для обозначения некоторых понятий, например, при определении языков программирования нетерминалами служат такие элементы, как ⟨программа⟩, ⟨оператор ⟩, ⟨выражение⟩ и т.д. В силу того, что все цепочки языка выводятся из аксиомы, аксиоме должно соответствовать основное определяемое понятие, в частности, для языков программирования таким нетерминалом может быть ⟨программа⟩.

Чтобы легче было различать нетерминальные и терминальные символы, примем соглашение обозначать терминалы маленькими буквами, а нетерминалы — большими буквами (или заключать в угловые скобки). Будем также считать, что аксиомой является символ, стоящий в левой части самого первого правила грамматики.

7.2 Классификация грамматик

Правила порождающих грамматик позволяют осуществлять самые разные преобразования строк. Определенные ограничения на вид правил позволяют выделить классы грамматик. Общепринятой является предложенная Н. Хомским следующая *система классификации грамматик.*

Определение 7.5. Грамматики типа 0 — это грамматики, на правила вывода которых не наложено никаких ограничений.

Например, правило грамматики типа 0 может иметь вид $aAbS \to SbaaS$.

Определение 7.6. Грамматики типа 1 — грамматики непосредственно составляющих или контекстно–зависимые грамматики — это грамматики, правила вывода которых имеют вид $xAy \to x\phi y$, где $A \in V_N; x, y, \phi \in (V_N \cup V_T)^*$.

Например, правило контекстно–зависимой грамматики может иметь вид $aAbc \to aaabc$ или $Aa \to Ba$.

Определение 7.7. Грамматики типа 2 — бесконтекстные или контекстно–свободные грамматики — это грамматики, правила вывода которых имеют вид $A \to \phi$, где $\phi \in (V_T \cup V_N)^*$, $A \in V_N$.

Например, следующая грамматика является контекстно–свободной:

$$G : A \to aAb|ab.$$

Иногда выделяют специальный подкласс класса контекстно–свободных грамматик (КС–грамматик) — металинейные грамматики, правила вывода которых имеют вид $A \to xBy$ или $A \to \phi$, $x, y, \phi \in V_T^*$, $A, B \in V_N$. Приведенная выше грамматика является линейной.

Определение 7.8. Грамматики типа 3 — автоматные грамматики. Различают два типа автоматных грамматик: леволинейные и праволинейные. Леволинейные грамматики — грамматики, правила вывода которых имеют вид $A \to Ba$ или $A \to a$, где $a \in V_T, A, B \in V_N$. Праволинейные грамматики — это грамматики, правила вывода которых имеют вид $A \to aB$ или $A \to a$.

В дальнейшем мы в основном будем рассматривать контекстно–свободные грамматики (или КС–грамматики) и автоматные грамматики.

Пример 7.2. Язык $\{a^n b^n c^n, n \geq 0\}$ порождается следующей грамматикой типа 0:

$$\begin{aligned} G_0 : \quad & S \to AB \\ & A \to aADb|\varepsilon \\ & Db \to bD \\ & DB \to Bc \\ & B \to \varepsilon. \end{aligned}$$

Например, вывод цепочки $a^2b^2c^2$ имеет вид:

$$\begin{aligned} S \quad & \Rightarrow AB \Rightarrow aADbB \Rightarrow aaADbDbB \Rightarrow aaDbDbB \Rightarrow aabDDbB \Rightarrow \\ & \Rightarrow aabDbDB \Rightarrow aabbDDB \Rightarrow aabbDBc \Rightarrow aabbBcc \Rightarrow aabbcc. \end{aligned}$$

Пример 7.3.

Язык $a^n, n > 0$ порождается леволинейной грамматикой

$$G_2 : S \to Sa|a.$$

Вывод цепочки a^3 имеет вид:

$$S \Rightarrow Sa \Rightarrow Saa \Rightarrow aaa.$$

Определение 7.9. Язык L называется языком типа i, если существует грамматика типа i, порождающая L.

В частности, язык является КС–языком, если существует КС–грамматика, его порождающая. Тогда, язык $a^n b^n$ (см. пример 7.1) является КС–языком, а язык a^n— автоматным языком (см. пример 7.3).

7.3 Основные свойства КС–языков и КС–грамматик

Поскольку любой язык — это некоторое множество цепочек, представляет интерес выполнение специальных языковых и обычных теоретико–множественных операций над языками. Рассмотрим операции пересечения, объединения, итерации, усеченной итерации и произведения, выполняемые над классами КС–языков.

Теорема 7.1. Семейство КС–языков замкнуто относительно операций объединения, произведения, итерации и усеченной итерации.

Доказательство. Язык является контекстно–свободным, если существует КС–грамматика, порождающая его. Пусть L_1 и L_2 — КС–языки, тогда существуют КС–грамматики

$$G_1 = (V_{T1}, V_{N1}, P_1, S_1) \text{ и } G_2 = (V_{T2}, V_{N2}, P_2, S_2) \ ,$$

порождающие соответственно L_1 и L_2. Всегда можно считать, что множества нетерминальных символов этих КС–грамматик не пересекаются и $V_{N1} \cap V_{N2} = \emptyset$. Рассмотрим КС–грамматику

$$G = (V_{T1} \cup V_{T2}, V_{N1} \cup V_{N2} \cup \{S | S \notin V_T \cup V_N\}, P, S),$$

где S — новый нетерминал, а P — множество правил, полученное объединением правил исходных грамматик и двух новых правил $S \to S_1|S_2$, т.е.

$$P = P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}.$$

Любой вывод в G имеет вид $S \Rightarrow S_1 \overset{*}{\Rightarrow} x$ или $S \Rightarrow S_2 \Rightarrow^* y$, причем $S_1 \overset{*}{\Rightarrow} x$ — это всегда вывод в G_1, а $S_2 \overset{*}{\Rightarrow} y$ — это всегда вывод в G_2, т.к. множества нетерминальных символов V_{N1} и V_{N2} не пересекаются. Тогда $L(G) = L(G_1) \cup L(G_2)$.

Аналогично можно показать, что язык L_1L_2 порождается грамматикой G, множество правил которой содержит кроме P_1 и P_2 правило $S \to S_1S_2$.

Можно также показать язык L_1^+ порождается грамматикой, множество правил которой, кроме правил P_1, содержит два дополнительных правила $S \to SS_1|S_1$, а язык L_1^* — грамматикой с дополнительными правилами $S \to SS_1|\varepsilon$. □

Операции произведения, усеченной итерации, итерации и объединения позволяют легко строить КС–грамматики сложных языков через грамматики простых языков. Например, пусть требуется построить КС–грамматику, порождающую идентификаторы языка Паскаль.

Идентификатор — это последовательность цифр и букв, начинающаяся с буквы, следовательно, идентификатор можно определить как произведение понятия <буква> и понятия <знаки>. Знаки — это произвольная последовательность букв и цифр, в том числе и пустая, следовательно <знаки> — итерация элемента <знак>, в качестве которого могут выступать буквы и цифры. Тогда получим грамматику

$$\begin{aligned} G: \ & < \text{идентификатор} > \to < \text{буква} >< \text{знаки} > \\ & < \text{знаки} > \to < \text{знак} >< \text{знаки} > |\varepsilon \\ & < \text{знак} > \to < \text{буква} > | < \text{цифра} > \\ & < \text{буква} > \to A|B|...|Z \\ & < \text{цифра} > \to 0|1|...|9 \end{aligned}$$

Рассмотрим операции дополнения и пересечения, выполняемые над КС–языками. Чтобы доказать незамкнутость семейства КС–языков относительно этих операций, сформулируем пока без доказательства следующую теорему (доказательство рассмотрим далее в параграфе 7.6).

Теорема 7.2. Язык $a^nb^nc^n, n > 0$ в алфавите {a,b,c} не является контекстно–свободным.

Теорема 7.3. Семейство КС–языков не замкнуто относительно операции пересечения.

Доказательство. Для доказательства достаточно привести хотя бы один пример таких КС–языков L_1 и L_2, пересечение которых не является КС–языком. Рассмотрим

языки $L_1 = a^n b^m c^m, n, m > 0$ и $L_2 = a^n b^n c^m, n, m > 0$. Они являются контекстно–свободными, т.к. порождаются КС–грамматиками соответственно

$$\begin{array}{ll} G_1: & S \to AB \\ & A \to aA|a \\ & B \to bBc|bc, \\ G_2: & S \to AB \\ & A \to aAb|ab \\ & B \to cB|c. \end{array}$$

Пересечение L_1 и L_2 есть язык $a^n b^n c^n, n > 0$, по теореме 7.2. не являющийся контекстно–свободным.□

Теорема 7.4. Семейство КС–языков не замкнуто относительно операции дополнения.

Доказательство. Рассмотрим КС–языки L_1 и L_2. Пусть дополнение сохраняет свойства КС–языка оставаться контекстно–свободным. Тогда $\overline{L_1}$ и $\overline{L_2}$ — КС–языки. По теореме 7.1 объединение КС–языков $\overline{L_1}$ и $\overline{L_2}$ — КС–язык, т.е. $\overline{L_1} \cup \overline{L_2}$- КС–язык, но тогда и $\overline{\overline{L_1} \cup \overline{L_2}}$ — тоже КС–язык. Тогда $\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$ — КС–язык для любых КС–языков L_1 и L_2, что противоречит теореме 7.3. □

Следствием доказанных теорем 7.3 и 7.4 является невозможность использования операций пересечения и дополнения при построении КС–грамматик сложных языков из КС–грамматик более простых языков в отличие от операций объединения, произведения, итерации и усеченной итерации.

Кроме правил использования операций объединения, произведения, итерации и усеченной итерации при синтезе КС–грамматик применяется еще одно правило, позволяющее сконструировать цепочки из симметрично стоящих элементов. Для того, чтобы грамматика порождала цепочки вида $x^n z y^n (n \geq 0)$, достаточно в ней иметь правила вида

$$A \to xAy|z.$$

Действительно, любой вывод из нетерминала A имеет вид $A \Rightarrow xAy \Rightarrow x^2Ay^2 \Rightarrow ... \Rightarrow x^nAy^n \Rightarrow x^xzy^n$.

Например, язык $a^n b^{3m} c^m a^{2n} = a^n (bbb)^m c^m (aa)^n$ ($n, m \geq 0$) порождается КС–грамматикой

$$\begin{array}{ll} G: & S \to aSaa|B \\ & B \to bbbBc|\varepsilon. \end{array}$$

7.4 Грамматический разбор

В КС–грамматике может быть несколько выводов, эквивалентных в том смысле, что во всех них *применяются одни и те же правила к одним и тем же нетерминалам* в цепочках, полученных в процессе вывода, различие имеется только в порядке применения этих правил. Например, в грамматике

$$G: S \to ScS|b|a$$

возможны два эквивалентных вывода

$$\begin{array}{l} S \Rightarrow ScS \Rightarrow Scb \Rightarrow acb \\ S \Rightarrow ScS \Rightarrow acS \Rightarrow acb. \end{array}$$

Можно ввести удобное графическое представление вывода, называемое деревом вывода, или деревом грамматического разбора, или синтаксическим деревом.

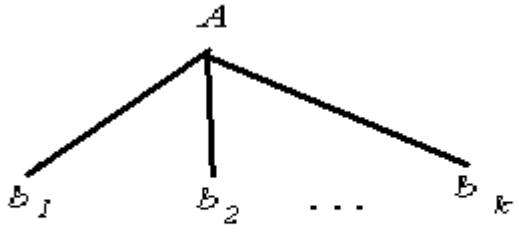
Определение 7.10. Деревом вывода цепочки x в КС–грамматике

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

называется упорядоченное дерево, каждая вершина которого помечена символом из множества $V_T \cup V_N \cup \{\varepsilon\}$ так, что каждому правилу

$$A \to b_1 b_2 \dots b_k \in P,$$

используемому при выводе цепочки x, в дереве вывода соответствует поддерево с корнем A и прямыми потомками b_1, b_2,$\dots$, b_k :

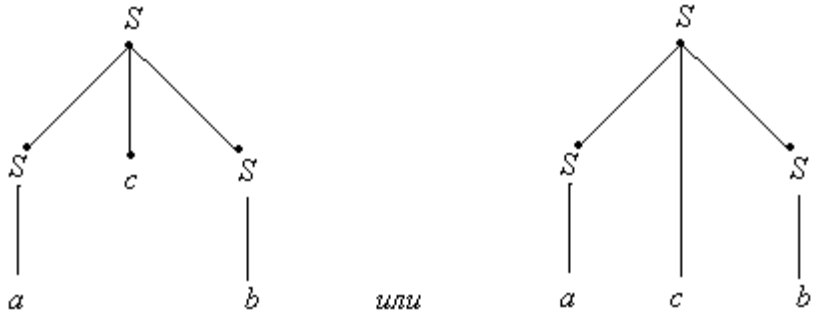


В силу того, что цепочка $x \in L(G)$ выводится из аксиомы S, корнем дерева вывода всегда является аксиома. Внутренние узлы дерева соответствуют нетерминальным символам грамматики. Концевые вершины дерева вывода (листья) — это вершины, не имеющие потомков. Такие вершины соответствуют либо терминалам, либо пустым символам ε при условии, что среди правил грамматики имеются правила с пустой правой частью. При чтении слева направо концевые вершины дерева вывода образуют цепочку, вывод которой представлен деревом. Именно по этой причине деревом вывода должно быть *упорядоченное* дерево.

Например, в рассмотренной выше грамматике с правилами

$$S \to ScS|b|a$$

цепочке acb соответствует дерево вывода

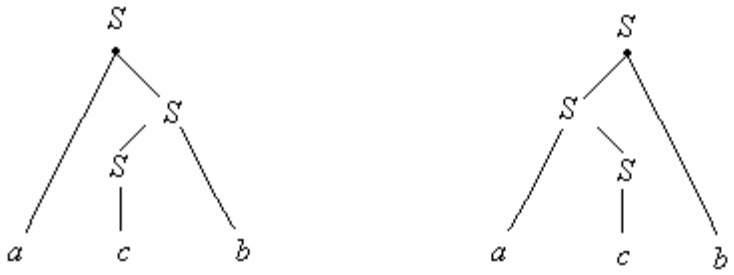


Дерево вывода часто называют также деревом грамматического разбора, деревом разбора, синтаксическим деревом. Процесс построения дерева разбора называется грамматическим разбором или синтаксическим анализом.

Одной цепочке языка может соответствовать более, чем одно дерево, т.к. она может иметь разные выводы, порождающие разные деревья. Например, в грамматике

$$G : S \to aS|Sb|c$$

цепочка $a$cb имеет два разных дерева вывода:



Определение 7.11. КС–грамматика G называется неоднозначной (или неопределенной), если существует цепочка $x \in L(G)$, имеющая два или более дерева вывода.

Если грамматика используется для определения языка программирования, желательно, чтобы она была однозначной. В противном случае программист и компилятор могут по–разному понять смысл некоторых программ. Рассмотрим, например, оператор IF языка Паскаль. Пусть он порождается КС–грамматикой

$$G : S \to \text{ if } V \text{ then } S \text{ else } S| \text{ if } V \text{ then } S|O,$$

где V — соответствует понятию "выражение O — "оператор". Эта грамматика неоднозначна, т.к. существует оператор, имеющий два дерева вывода, представленных на рис. 7.1 и 7.2.

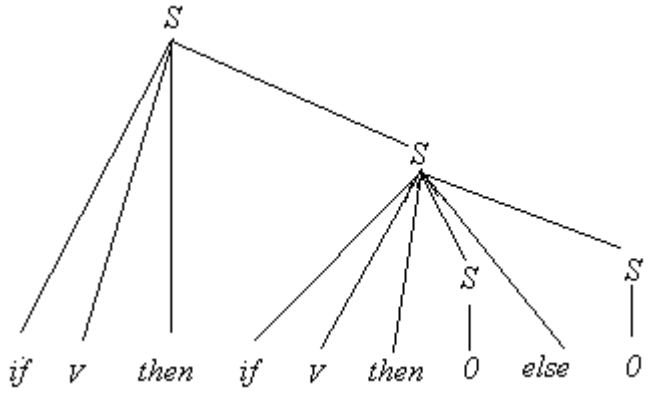


Рис. 7.1: Правильное дерево разбора оператора if.

Первое дерево предполагает интерпретацию if V then (if V then O else O), тогда как второе дерево дает if V then (if V then O) else O. Определенная нами неоднозначность — это свойство грамматики, а не языка. Для некоторых неоднозначных грамматик можно построить эквивалентные им однозначные грамматики. Например,

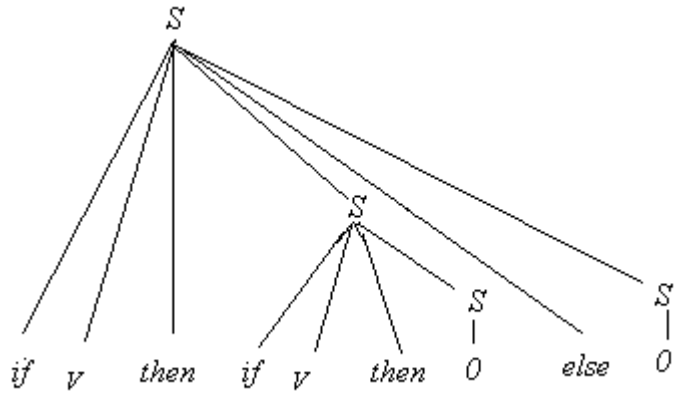


Рис. 7.2: Неправильное дерево разбора оператора if.

приведенная выше грамматика оператора IF неоднозначна потому, что ELSE можно ассоциировать с двумя разными THEN. Неоднозначность можно устранить, если договориться, что ELSE должно соответствовать последнему из предшествующих ему THEN:

$$G_1: \quad S \to \text{ if } V \text{ then } S| \text{ if } V \text{ then } S1 \text{ else } S|O$$
$$S_1 \to \text{ if } V \text{ then } S_1 \text{ else } S|O$$

Рассмотренный выше оператор имеет в этой грамматике единственное дерево вывода, представленное на рис. 7.3 и являющееся аналогом дерева рис. 7.1. Аналог дерева 7.2 в данной грамматике построить невозможно.

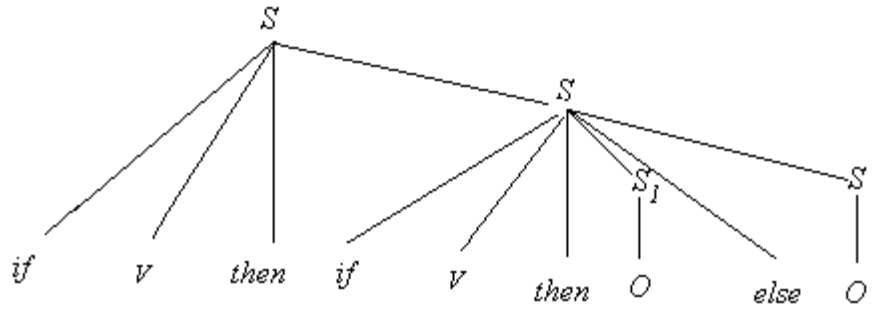


Рис. 7.3: Дерево разбора оператора if в однозначной КС–грамматике.

Это дерево предполагает интерпретацию if V then (if V then O else O). Попытка построить дерево вывода этой цепочки другим способом обречена на неудачу.

Определение 7.12. КС–язык называется существенно–неоднозначным (или существенно–неопределенным), если он не порождается никакой однозначной КС–грамматикой.

Процесс построения дерева вывода называется *грамматическим разбором*. Рассмотрим несколько определений, связанных с процессом построения дерева вывода.

Определение 7.13. Любая цепочка (не обязательно терминальная), выводимая из аксиомы, называется сентенциальной формой.

Например, в грамматике

$$G : S \to aSSb|abS|ab$$

существует вывод

$$S \Rightarrow aSSb \Rightarrow aabSSb \Rightarrow aababSb \Rightarrow aababbabb,$$

тогда $S, aSSb, aabSSb, aababSb, aababbabb$ — сентенциальные формы. Язык $L(G)$ составляют только терминальные сентенциальные формы.

Различают две стратегии разбора: восходящую ("снизу вверх") и нисходящую ("сверху вниз"). Эти термины соответствуют способу построения синтаксических деревьев. При нисходящей стратегии разбора дерево строится от корня (аксиомы) вниз к терминальным вершинам. Например, в грамматике

$$G : S \to aSSb|abS|ab$$

при нисходящем разборе цепочки *aababbabb* получаем последовательность частичных деревьев, представленную на рис. 7.4.

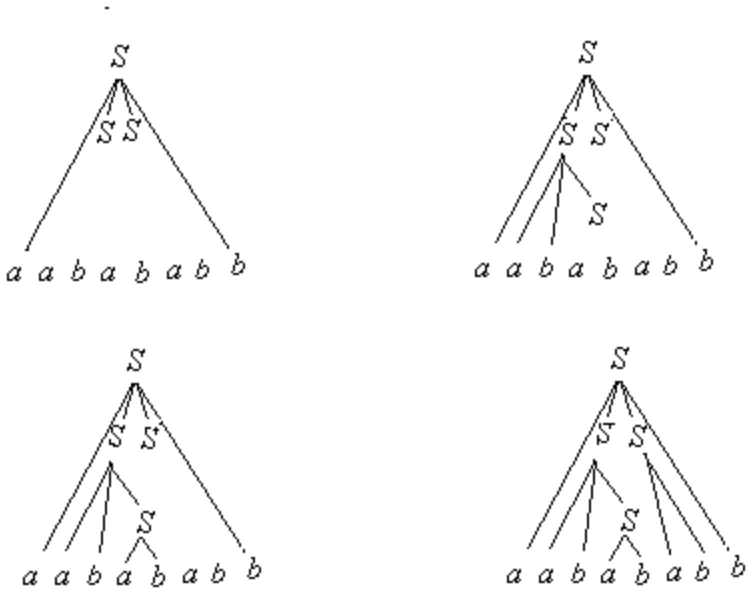


Рис. 7.4: Нисходящий разбор в КС-грамматике $G : S \to aSSb|abS|ab$

Главная задача при нисходящем разборе — выбор того правила $A \to \phi_i$ из совокупности правил $A \to \phi_1|\phi_2|\ldots|\phi_k$, которое следует применить на рассматриваемом шаге грамматического разбора.

При восходящем разборе дерево строится от терминальных вершин вверх к корню дерева — аксиоме. Например, дерево разбора рассмотренной цепочки *aababbabb* по восходящей стратегии представлено на рис. 7.5.

Определение 7.14. Преобразование цепочки, обратное порождению, называется приведением (или редукцией). Цепочка y прямо редуцируема к цепочке x в грамматике G, если x прямо порождает y.

Определение 7.15. Пусть $z = xty$ — сентенциальная форма, тогда t называется фразой сентенциальной формы z для нетерминального символа A, если $S \stackrel{*}{\Rightarrow} xAy$ и

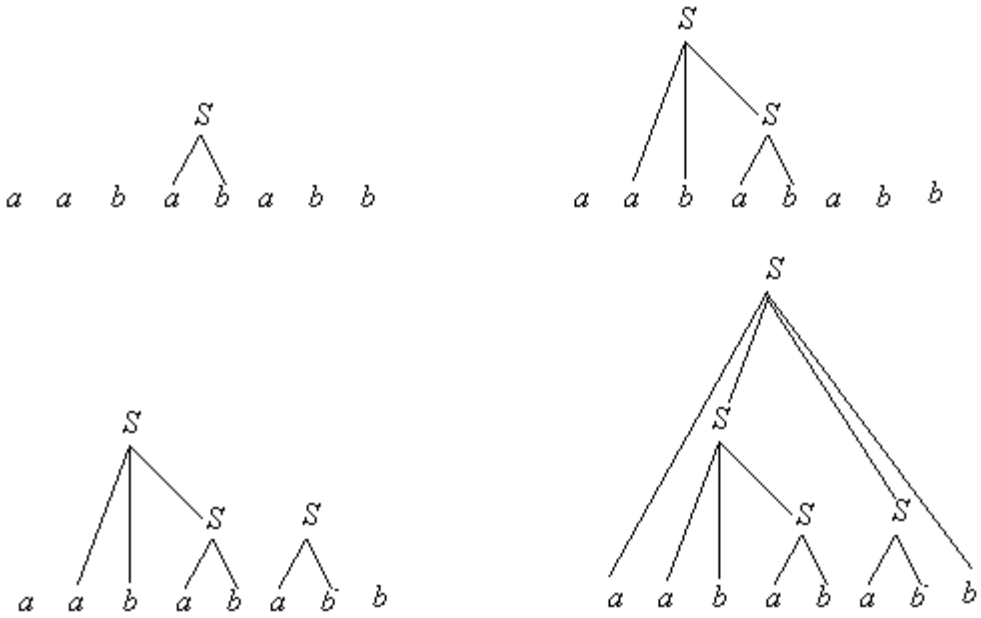


Рис. 7.5: Восходящий разбор в КС-грамматике $G : S \to aSSb|abS|ab$

$A \stackrel{*}{\Rightarrow} t$. Цепочка t называется простой фразой, если t является фразой и $A \to t$ — правило грамматики.

Обычно грамматический разбор выполняют слева направо, т.е. сначала обрабатывают самые левые символы рассматриваемой цепочки и продвигаются по цепочке вправо только тогда, когда это необходимо. При выполнении разбора по восходящей стратегии слева направо на каждом шаге редуцируется самая левая простая фраза. Самая левая простая фраза сентенциальной формы называется *основой.* Цепочка вправо от основы всегда содержит только терминальные символы.

Главная задача при восходящем разборе — поиск основы и, если грамматика содержит несколько правил с одинаковыми правыми частями, выбор нетерминального символа, к которому должна редуцироваться основа.

Если $x \notin L(G)$, то не существует вывода $S \stackrel{*}{\Rightarrow} x$ в грамматике G, а это значит, что для x нельзя построить дерево вывода. Любой компилятор выполняет синтаксический анализ исходного модуля и выдает сообщение об ошибке, если дерево разбора исходного модуля не существует.

Пример 7.4. Даны две КС–грамматики, порождающие выражение, которое может содержать знаки операций, круглые скобки и символы "a" в качестве операндов:

$$G_1 : S \to S + S|S - S|S * S|S/S|(S)|a,$$

$$\begin{aligned} G_2 : \quad & S \to S + A|S - A|A \\ & A \to A * B|A/B|B \\ & B \to a|(S). \end{aligned}$$

Требуется определить, являются ли эти грамматики однозначными. Если какая–либо из этих грамматик неоднозначна, привести пример цепочки, для которой существуют два различных дерева разбора.

Очевидно, что в грамматике G_1 не учитываются приоритеты операций, поэтому любое выражение, содержащее знаки различных приоритетов, может быть выведено

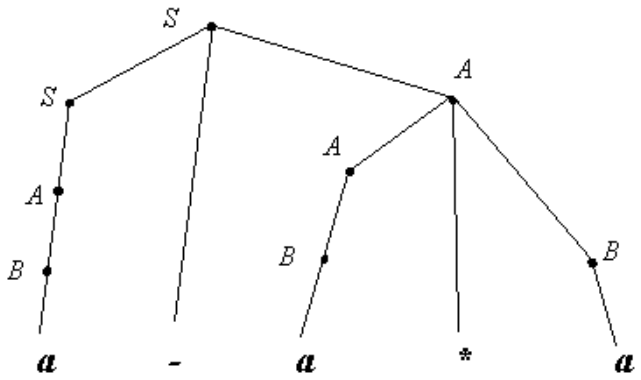
из аксиомы разными способами. Например, выражение $a - a * a$ можно вывести по–разному:

$$S \Rightarrow S - S \Rightarrow a - S \Rightarrow a - S * S \Rightarrow a - a * S \Rightarrow a - a * a,$$

$$S \Rightarrow S * S \Rightarrow S * a \Rightarrow S - S * a \Rightarrow a - S * a \Rightarrow a - a * a.$$

Этим выводам соответствуют разные деревья грамматического разбора.

Если первое из них отражает реальный порядок выполнения операций в языках программирования типа Си, то второе дерево противоречит таким правилам. Грамматика G_1 неоднозначна. В то же время в грамматике G_2 выражению $a - a * a$ соответствует единственное дерево разбора:



7.5 Преобразования КС–грамматик

Для любого КС–языка существует бесконечное число КС–грамматик, порождающих этот язык. Часто требуется изменить грамматику так, чтобы она удовлетворяла определеным требованиям, не изменяя при этом порождаемый грамматикой язык.

Определение 7.16. Грамматики G_1 и G_2 называются эквивалентными, если совпадают порождаемые ими языки.

7.5.1 Преобразование 1 — правила с одним нетерминалом

Рассмотрим первое эквивалентное преобразование — удаление правил вида "< нетерминал >→< нетерминал >". Очевидно, что присутствие таких правил в грамматике существенно увеличивает высоту дерева разбора, а, следовательно, процесс разбора занимает больше времени.

Теорема 7.5. Для любой КС–грамматики можно построить эквивалентную грамматику, не содержащую правил вида $A \to B$, где A и B — нетерминальные символы.

Доказательство. Пусть имеется КС–грамматика

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

где $V_N = \{A_1, A_2, ..., A_n\}$, а в P входят правила указанного вида. Разобъем P на два непересекающихся подмножества:$P = P_1 \cup P_2$, $P_1 \cap P_2 = \emptyset$, где в P_1 включены все правила вида $A_i \to A_k$ $(A_i, A_k \in V_N)$, а в P_2 — все остальные правила, т.е. $P_2 = P \setminus P_1$. Определим для каждого $A_i \in V_N$ множество правил $P(A_i)$, включив в него все такие

правила $A_i \to \phi$, что $A_i \stackrel{+}{\Rightarrow} A_j$ и $A_j \to \phi$ — правило из множества P_2. Построим новую КС–грамматику $G_1 = (V_T, V_N, P_1, S)$, в которой множество терминальных и нетерминальных символов и аксиома совпадают с соответствующими объектами грамматики G, а множество правил получено объединением правил множества P_2 и правил $P(A_i)$ для всех $1 \le i \le n$:

$$P = \bigcup_{i=1}^{n} P(A_i) \cup P_2.$$

Эта грамматика не содержит правил вида $A_i \to A_k$. Покажем, что она эквивалентна исходной, т.е. покажем справедливость $L(G) \subseteq L(G_1)$ и $L(G_1) \subseteq L(G)$. С этой целью рассмотрим вывод цепочки x в грамматике G. Рассмотрим самый левый шаг вывода, на котором применялось правило из P_1 (заметим, что если правила из P_1 не применялись, то вывод в G является одновременно и выводом в G_1). Цепочка x является терминальной в силу своей принадлежности языку $L(G)$, тогда существует шаг вывода, на котором применено правило из P_2. Но тогда в G_1 существует правило $A_i \to \phi \notin P_1$, т.е. часть вывода в G можно заменить на один шаг вывода в G_1. Продолжая эту процедуру через конечное число шагов (не более длины вывода) из вывода в G получим вывод x в G_1.Таким образом, показали $L(G) \subseteq L(G_1)$. Обратная процедура построения вывода в G по выводу в G_1 очевидна.□

Доказательство теоремы 7.5. дает алгоритм эквивалентного преобразования грамматик с целью удаления правил вида $A \to B$.

Пример 7.5. Пусть задана КС–грамматика

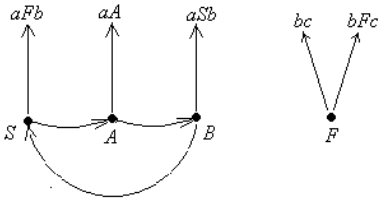
$$\begin{array}{ll} G: & S \to aFb|A \\ & A \to aA|B \\ & B \to aSb|S \\ & F \to bc|bFc. \end{array}$$

Построим множества P_2, $P(S)$, $P(A)$, $P(B)$, $P(F)$:
1) $P_2 = \{S \to aFb, A \to aA, B \to aSb, F \to bc|bFc\}$;
2) $S \Rightarrow A \Rightarrow B$, т.е. $S \Rightarrow^* A, S \Rightarrow^* B$, тогда $P(A) = \{S \to aA, S \to aSb\}$;
3) $A \Rightarrow B \Rightarrow S$, т.е. $A \Rightarrow^* B, A \Rightarrow^* S$, тогда $P(A) = \{A \to aSb, A \to aFb\}$;
4) $B \Rightarrow S \Rightarrow A$, т.е. $B \Rightarrow^* S, B \Rightarrow^* A$, тогда $P(B) = \{B \to aFb, B \to aA\}$;
5) из F нельзя вывести нетерминальный символ, тогда $P(F) = \emptyset$.
Объединяя все эти правила, получаем грамматику, эквивалентную исходной:

$$\begin{array}{ll} G: & S \to aFb|aA|aSb \\ & A \to aA|aSb|aFb \\ & B \to aSb|aFb|aA \\ & F \to bs|bFc. \end{array}$$

Этот алгоритм удобно использовать при автоматизированном преобразовании грамматик с помощью ЭВМ. При неавтоматизированном преобразовании он оказывается довольно сложным. В таком случае проще применить графическую модификацию метода. С этой целью каждому нетерминальному символу и каждой правой части правил множества P_2 поставим в соответствие вершину графа. Из вершины с меткой U в вершину с меткой V направлено ребро, если существует в грамматике правило $U \to V$. Правило $A \to w$ принадлежит новой грамматике, если из вершины с меткой A существует путь в вершину с меткой w.

В нашем случае получим граф



7.5.2 Преобразование 2 — правила с одинаковыми правыми частями

При удалении правил вида $A \to B$ в множестве правил появляется много правил с одинаковой правой частью. Рассмотрим построение для произвольной КС–грамматики эквивалентной КС–грамматики, все правила которой имеют различные правые части.

Пусть $G_n = (V_T, V_N, P_n, S)$ — некоторая КС–грамматика, все нетерминальные символы которой имеют непустые несовпадающие множества нижних индексов:

$$V = \{A_{i_1 i_2 \ldots i_k}\}, i_n < i_m \text{ при } n < m.$$

Пусть в множестве P_n существуют правила с совпадающими правыми частями

$$\begin{array}{l} A_{i_1 i_2 \ldots i_k} \to \phi \ , \\ A_{j_1 j_2 \ldots, j_m} \to \phi \ , \\ A_{l_1 l_2 \ldots l_r} \to \phi \ . \end{array} \tag{7.1}$$

Построим грамматику G_{n+1} следующим образом:

1) заменим правила (7.1) на $A_{n_1 n_2 \ldots n_t} \to \phi$, где $n_i < n_j$ при $i < j$ и множество $\{n_1, n_2, \ldots n_t\}$ есть объединение индексов нетерминалов, стоящих в левых частях правил (7.1);

2) если некоторое A_i для $i \in \{n_1, n_2, \ldots, n_t\}$ есть аксиома S грамматики G_n, то новому множеству правил P_{n+1} принадлежит правило $S \to A_{n_1, n_2 \ldots n_t}$;

3) если множеству P_n принадлежат правила вида

$$B \to \xi A_{p_1 p_2 \ldots p_u} \eta, \tag{7.2}$$

такие, что $\{p_1, p_2, \ldots p_u\} \subset \{n_1, n_2, \ldots n_t\}$, то к новому множеству правил добавляются правила вида $B \to \xi A_{n_1 n_2 \ldots n_t} \eta$, полученные из (7.2) с помощью подстановки $A_{n_1 n_2 \ldots n_t}$ вместо некоторых (быть может, никаких) $A_{p_1 p_2 \ldots p_u}$ в правой части правила (7.2). Назовем такое построение G_{n+1} по G_n операцией частичного склеивания по индексам.

Пример 7.6. Пусть задана грамматика

$$\begin{array}{ll} G_1: & S \to bA_1 | baA_2 \\ & A_1 \to aaA_1 | ab | d \\ & A_2 \to aaA_2 | ab | c \end{array}$$

Выполним частичное склеивание, удалив правила $A_1 \to ab$ и $A_2 \to ab$:

$$\begin{array}{ll} G_2: & S \to bA_1 | baA_2 | bA_{12} | baA_{12} \\ & A_1 \to aaA_1 | d | aaA_{12} \\ & A_2 \to aaA_2 | c | aaA_{12} \\ & A_{12} \to ab \end{array}$$

Выполним опять частичное склеивание, удалив правила $A_1 \to aaA_{12}$ и $A_2 \to aaA_{12}$:

$$\begin{array}{ll} G_1: & S \to bA_1|baA_2|bA_{12}|baA_{12} \\ & A_{12} \to ab|aaA_{12} \\ & A_1 \to aaA_1|d|aaA_{12} \\ & A_2 \to aaA_2|c|aaA_{12}. \end{array}$$

При выполнении частичного склеивания может оказаться, что грамматике G_{n+1} одновременно принадлежат такие правила с одинаковыми правыми частями

$$A_{i_1 \ldots i_t} \to \xi \ , \tag{7.3}$$

$$A_{n_1 \ldots n_k} \to \xi \ , \tag{7.4}$$

что $\{i_1, i_2, \ldots i_t\} \subset \{n_1, n_2, \ldots n_k\}$. Назовем обобщенным склеиванием по индексам такое построение грамматики G_{n+1} по грамматике G_n, когда

1) выполнено частичное склеивание G_n к G_{n+1};

2) если среди множества полученных правил грамматики G_{n+1} окажутся правила вида (7.3), (7.4), то из множества правил грамматики удаляется правило (7.3) как более слабое.

В нашем примере грамматике одновременно принадлежат правила $A_1 \to aaA_{12}$, $A_2 \to aaA_{12}$, $A_{12} \to aaA_{12}$, первые два из которых можно удалить. Правые части правил грамматики, полученной в результате такого удаления, различны:

$$\begin{array}{ll} G_2: & S \to bA_1|baA_2|bA_{12}|baA_{12} \\ & A_1 \to aaA_1|d \\ & A_2 \to aaA_2|c \\ & A_{12} \to ab|aaA_{12} \end{array}$$

Можно доказать следующую теорему.

Теорема 7.6. В результате последовательного применения к любой КС–грамматике операции обобщенного склеивания строится эквивалентная КС–грамматика, все правила которой имеют различные правые части.

Доказательство. Очевидно, что последовательное применение алгоритма обобщенного склеивания преобразует любую КС-грамматику к такой форме, когда все правила имеют различные правые части. Осталось показать эквивалентность исходной грамматики и грамматики, полученной в результате операции обобщенного склеивания. Сначала заметим, что частичное склеивание является эквивалентным преобразованием грамматики G_n, т.к. замена (7.1) и подстановка (7.2) каждому выводу в G_n взаимно однозначно ставят вывод в G^1_{n+1}, полученной из G_n операцией частичного склеивания. Пусть грамматика G_{n+1} получена из грамматики G_n операцией обобщенного склеивания. Покажем, что удаление более слабых правил (7.3) при наличии правил (7.4) не нарушает эквивалентности грамматик. Действительно, пусть в выводе некоторой цепочки x в грамматике G_n применялось правило вида (7.3):

$$S \stackrel{*}{\Rightarrow} z_1 A_{i_1,\ldots,i_t} z_2 \to z_1 \xi z_2 \stackrel{*}{\Rightarrow} x.$$

Нетерминал $A_{i_1,\ldots,i_t}$ появился в этом выводе на одном из предшествующих шагов:

$$S \stackrel{*}{\Rightarrow} y_1 B y_2 \Rightarrow y_1 \omega A_{i_1 \ldots i_t} \eta y_2 = z_1 A_{i_1 \ldots i_t} z_2 \to z_1 \xi z_2 \stackrel{*}{\Rightarrow} x.$$

Но тогда по правилу подстановки при выполнении операции частичного склеивания в множество правил новой грамматики G_{n+1} включается новое првило

$$B \to \omega A_{n_1 n_2 \ldots n_t} \eta$$

и, следовательно, существует эквивалентный вывод в G_{n+1}:

$$S \overset{*}{\Rightarrow} y_1 B y_2 \Rightarrow y_1 \omega A_{n_1 n_2 \ldots n_t} \eta y_2 = z_1 A_{n_1 n_2 \ldots n_t} z_2 \to z_1 \xi z_2 \overset{*}{\Rightarrow} x.$$

Таким образом, $L(G_n) = L(G_{n+1})$. $\square$

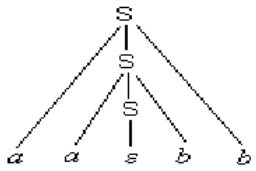
7.5.3 Преобразование 3 — неукорачивающие грамматики

Определение 7.17. Грамматика, не содержащая правил с пустой правой частью, называется неукорачивающей грамматикой.

При выводе в неукорачивающей грамматике длина выводимой цепочки не уменьшается при переходе от k–го шага вывода к $(k+1)$–му. В грамматике с правилами вида $A \to \varepsilon$ длина выводимой цепочки может уменьшаться:

$$\begin{aligned} G: \quad & S \to aSb | \varepsilon \\ & S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb. \end{aligned}$$

В соответствии с определением (7.17) грамматика называется укорачивающей, если на некотором шаге может уменьшиться длина разбора выводимой цепочки. Восходящий разбор укорачивающих грамматик, как правило, сложнее по сравнению с разбором в неукорачивающих грамматиках, так как необходимо найти такой фрагмент входной цепочки, где можно вставить пустую цепочку ε:



Основным требованием, предъявляемым к языку программирования, является возможность определить для каждой данной последовательности символов, является ли она программой, написанной на этом языке. Это необходимо в первую очередь для того, чтобы сделать возможной автоматическую трансляцию программ. На формальном уровне это означает, что должна быть разрешима проблема принадлежности любой цепочки любому заданному КС–языку. Другими словами, должен существовать алгоритм, позволяющий определить, принадлежит ли произвольная заданная цепочка x произвольному заданному КС–языку L. С этой точки зрения особенный интерес представляют неукорачивающие КС–грамматики в силу следующих теорем.

Теорема 7.7. Пусть $G = (V_T, V_N, P, S)$ есть неукорачивающая КС–грамматика и n — число нетерминальных символов в множестве V_N. Цепочка $x \in V_T^*$ длины k тогда и только тогда принадлежит языку $L(G)$, когда существует ее вывод

$$S \Rightarrow U_1 \Rightarrow \ldots \Rightarrow U_p = x, \tag{7.5}$$

такой, что $p \leq n^k$.

Доказательство. Пусть $G = (V_T, V_N, P, S)$ и $x \in V_T^*$. Согласно определению языка $L(G)$, если существует вывод (7.5), то $x \in L(G)$. Допустим теперь, что цепочка x выводима из S в G. Пусть r — наименьшее число, для которого существует вывод $S \Rightarrow Z_1 \Rightarrow \ldots \Rightarrow Z_r = x$. Поскольку G — неукорачивающая КС–грамматика, то $|Z_i| \leq |Z_{i+1}|$ для каждого i. Предположим, что $r > n^k$. Тогда в выводе x найдутся цепочки $Z_l = Z_m$ при $l < m$. Отсюда следует, что имеется более короткий вывод $S \Rightarrow Z_1 \Rightarrow \ldots \Rightarrow Z_l \Rightarrow Z_{m+1} \Rightarrow \ldots \Rightarrow Z_r = x$, что противоречит свойству минимальности рассмотренного нами вывода.□

Теорема 7.8. Существует алгоритм, позволяющий узнать, принадлежит ли произвольная цепочка языку, порождаемому данной неукорачивающей КС–грамматикой.

Доказательство. Пусть x — произвольная цепочка из Σ^* и G — некоторая неукорачивающая КС–грамматика с тем же множеством терминальных символов. Пусть длина цепочки x равна k. Из теоремы 7.7 следует, что $x \in L(G)$ тогда и только тогда, когда x порождается с помощью некоторого вывода, длина которого не превышает $m = n^{|x|}$, где n — число нетерминальных символов в G. Тогда достаточно просмотреть все выводы, длины которых не превышают m. Цепочка x принадлежит $L(G)$ тогда и только тогда, когда ее можно получить с помощью хотя бы одного из рассмотренных выводов.□

Представляет интерес преобразование любой КС–грамматики в эквивалентную неукорачивающую грамматику. Если G — КС–грамматика и $\varepsilon \in L(G)$, то хотя бы одно правило вида $A \to \varepsilon$ в множестве правил G имеется. Если $\varepsilon \notin L(G)$, то по грамматике G всегда можно построить КС–грамматику, тоже порождающую язык $L(G)$, но не содержащую ни одного правила с пустой правой частью. Алгоритм такого построения мы рассмотрим при доказательстве следующей теоремы.

Теорема 7.9. Для произвольной КС–грамматики, порождающей язык без пустой цепочки, можно построить эквивалентную неукорачивающую КС–грамматику.

Доказательство. Построим множество всех нетерминальных символов грамматики $G = (V_T, V_N, P, S)$, из которых выводится пустая цепочка. С этой целью выделим следующие множества нетерминалов:

$$W_1 = \{A | A \to \varepsilon \in P\},$$
$$W_{m+1} = W_m \cup \{B | B \to \phi \in P, \phi \in W_m^*\}.$$

Ясно, что $A \stackrel{*}{\Rightarrow} \varepsilon$ тогда и только тогда, когда $A \in W_n$ для некоторого n ($n < |V_N|$).

Рассмотрим множество P_1 , состоящее из всех правил $A \to \tilde{\alpha}$, полученных из правил $A \to \alpha \in P$ при удалении из α некоторых (возможно, никаких) вхождений символов из множества W_n:

$$P_1^0 = P \setminus \{A \to \varepsilon | A \to \varepsilon \in P\}$$
$$P_1^{i+1} = \{A \to \tilde{\alpha} | A \to \alpha \in P_1^i;\ \alpha = \alpha_1 B \alpha_2;\ B \in W_n \& \tilde{\alpha} = \alpha_1\alpha_2\}\ .$$

Грамматика $G_1 = (V_T, V_N, P_1, S)$ является неукорачивающей. Покажем, что она эквивалентна G. Пусть $x \in L(G_1)$, тогда существует вывод x в G_1 . Рассмотрим первый левый шаг вывода, на котором применялось первое правило, не принадлежащее P. По построению множеству P принадлежит такое правило $A \to \alpha$, что $\alpha = x_1 A_1 x_2 A_2 \ldots x_r A_r x_{r+1}$ и $A_1, A_2, \ldots A_r \in Wn$. Это означает, что в G выводимо $A_1 \stackrel{*}{\Rightarrow} \varepsilon, \ldots, A_r \stackrel{*}{\Rightarrow} \varepsilon$, но тогда рассмотренный шаг вывода в G_1 можно заменить на вывод в G :

$$x_1 A_1 x_2 A_2 ... x_r A_r x_{r+1} \Rightarrow x_1 x_2 A_2 ... x_r A_r x_{r+1} \Rightarrow \ldots \Rightarrow x_1 x_2 ... x_{r+1}\ .$$

Выполняя такие преобразования на каждом шаге вывода в G_1 , получим вывод x в G, следовательно, $L(G_1) \subseteq L(G)$. Аналогично можно показать, что $L(G) \subseteq L(G_1)$. С этой целью вывод x в G преобразуется в вывод x в G_1 удалением справа налево всех правил вида $A \to \varepsilon$.$\square$

Пример 7.7. Пусть задана КС–грамматика

$$\begin{array}{rl} G: & S \to AbA|cA|Bbb \\ & A \to aAb|\varepsilon \\ & B \to AA|ca. \end{array}$$

Построим множества нетерминалов, из которых выводится ε :

1) $W_1 = \{A\}$, т.к. имеется правило $A \to \varepsilon$;

2) $W_2 = \{A, B\}$, т.к. имеется правило $B \to AA$ и $A \in W_1$;

3) $W_3 = W_2$.

Построим неукорачивающую грамматику, эквивалентную исходной:

$$\begin{array}{rl} G: & S \to AbA|cA|Bbb|Ab|bA|b|c|bb \\ & A \to aAb|ab \\ & B \to AA|ca|A. \end{array}$$

Определение 7.18. КС–грамматики G_1 и G_2 эквивалентны с точностью до пустой цепочки, если

$$L(G_1) \setminus \{\varepsilon\} = L(G_2) \setminus \{\varepsilon\}.$$

Следовательно, для любой КС–грамматики можно построить неукорачивающую, эквивалентную исходной с точностью до ε.

7.5.4 Преобразование 4 — непродуктивные нетерминалы

Рассмотрим грамматику

$$\begin{array}{rl} G: & S \to bS|a|aA \\ & A \to aAb. \end{array}$$

Нетерминальный символ A не участвует в выводе терминальных цепочек и поэтому может быть исключен из грамматики вместе со всеми правилами, в которые он входит.

Определение 7.19. Нетерминальный символ A называется непродуктивным (непроизводящим), если он не порождает ни одной терминальной цепочки, т.е. не существует вывода $A \overset{*}{\Rightarrow} x$, где $x \in V_T^*$.

Представляет интерес удаление из грамматики всех непродуктивных нетерминальных символов.

Теорема 7.10. Для произвольной КС–грамматики можно построить эквивалентную КС–грамматику, все нетерминальные символы которой продуктивны.

Доказательство. Пусть $G = (V_T, V_N, P, S)$ — произвольная КС–грамматика. Построим множество всех продуктивных нетерминалов этой грамматики. С этой целью выделим множество W_1 нетерминалов, стоящих в левой части терминальных правил:

$$W_1 = \{A | A \to \phi \in P; \phi \in V_T^*\}.$$

Затем построим множества $W_2, W_3, \ldots, W_n$ (n — число нетерминальных символов в G):

$$W_{k+1} = W_k \cup \{B | B \to x \in P;\ x \in (V_T \cup W_k)^*\}.$$

Все $B \in W_1$ продуктивны по построению W_1. Пусть продуктивны все $B \in W_k (k \geq 1)$. Покажем, что продуктивными являются и все $B \in W_{k+1}$. Действительно, если $B \in W_{k+1}$, то либо $B \in W_k$ и является продуктивным, либо $B \notin W_k$, но тогда $B \to x \in P$ и $x \in (V_T \cup W_k)^*$. Пусть $x = x_1 A_1 x_2 \dots x_m A_m x_{m+1}$, тогда все A_i являются продуктивными, следовательно, B — продуктивный нетерминал. Пусть теперь B — продуктивный нетерминал, минимальный терминальный вывод из B имеет длину $k+1$ и $B \Rightarrow x_1 A_1 x_2 \dots x_m A_m x_{m+1} \stackrel{*}{\Rightarrow} y \in V_T^*$, все $x \in V_T^*$, все $A_i \in V_N^*$. Для каждого A_i существует вывод $A_i \stackrel{*}{\Rightarrow} y_i V_T t^*$ длины не более k, тогда все $A_i \in W_i (i \leq k)$ и по построению множества W_{k+1} ему принадлежит B. Таким образом, показали, что $B \in W_n$ тогда и только тогда, когда он продуктивен. Все символы множества $V_N \setminus W_n$ являются непродуктивными, не используются в выводе никакой терминальной цепочки и их можно удалить из грамматики вместе со всеми правилами, в которые они входят.□

Пример 7.8. Пусть задана грамматика:

$$\begin{aligned} G: \quad & S \to SA|BSb|bAb \\ & A \to aSa|bb \\ & B \to bBb|BaA. \end{aligned}$$

Построим множество продуктивных нетерминалов:

1) $W_1 = \{A\}$, т.к. имеется правило $A \to bb$;

2) $W_2 = \{A, S\}$, т.к. имеется правило $S \to bAb$ и $A \in W_1$;

3) $W_3 = W_2$.

Эквивалентная исходной грамматика, все символы которой продуктивны, имеет вид:

$$\begin{aligned} G_1: \quad & S \to SA|bAb \\ & A \to aAa|bb. \end{aligned}$$

7.5.5 Преобразование 5 — независимые нетерминалы

Существует еще один тип нетерминальных символов, которые можно удалять из грамматики вместе с правилами, в которые они входят. Рассмотрим, например, грамматику

$$\begin{aligned} G: \quad & S \to aS|b \\ & A \to aAb|ab. \end{aligned}$$

Нетерминальный символ A не участвует ни в каком выводе в этой грамматике, т.к. из аксиомы нельзя вывести цепочку, содержащую A.

Определение 7.20. В КС–грамматике G нетерминальный символ A зависит от нетерминального символа B, если в G существует вывод $A \stackrel{*}{\Rightarrow} xBy$.

В рассмотренном выше примере аксиома S не зависит от символа A, поэтому A можно удалить из грамматики.

Теорема 7.11. Для произвольной КС–грамматики можно построить эквивалентную КС–грамматику, аксиома которой зависит от всех нетерминальных символов.

Доказательство. Пусть $G = (V_T, V_N, P, S)$ — произвольная КС–грамматика. Построим множество нетерминалов, от которых зависит аксиома. С этой целью выделим множества $W_1, W_2, ...W_n$:

$$\begin{aligned} & W_1 = \{S\}, \\ & W_{k+1} = W_k \cup \{B | A \to xBy \in P, A \in W_k\}. \end{aligned}$$

Аналогично доказательству теоремы 7.10 можно показать, что $B \in W_n$ тогда и только тогда, когда S зависит от B (доказательство оставляется в качестве упражнения).

Все нетерминалы, не содержащиеся в W_n, можно удалить из грамматики вместе с правилами, в которые они входят. □

Пример 7.9. Пусть задана КС–грамматика

$$\begin{array}{rl} G: & S \to AS|bb \\ & A \to aAb|ab \\ & B \to SB|aAb. \end{array}$$

Найдем нетерминалы, от которых зависит аксиома:

1) $W_1 = \{S\}$;

2) $W_2 = \{S, A\}$, т.к. имеется правило $S \to AS$ и $S \in W_1$;

3) $W_3 = W_2$.

Эквивалентная грамматика, аксиома которой зависит от всех нетерминальных символов:

$$\begin{array}{rl} G_1: & S \to AS|Sb \\ & A \to aAb|ab \end{array}$$

Определение 7.21. КС–грамматика $G = (V_T, V_N, P, S)$ называется приведенной, если S зависит от всех нетерминалов из V_N и в V_N нет непродуктивных символов.

Из теорем 7.10 и 7.11 следует, что для любой КС–грамматики можно построить приведенную эквивалентную КС–грамматику.

7.5.6 Преобразование 6 — терминальные правила

Рассмотрим в КС–грамматике $G = (V_T, V_N, P, S)$ удаление правила с терминальной правой частью $A \to \beta$, где $\beta \in V_T^*$. Любой вывод с использованием этого правила имеет вид

$$S \Rightarrow^* x_1 A x_2 \Rightarrow x_1 \beta x_2.$$

Но нетерминал A в сентенциальной форме $x_1 A x_2$ появился на некотором предыдущем шаге вывода $B \to uAv$, тогда

$$S \stackrel{*}{\Rightarrow} z_1 B z_2 \stackrel{*}{\Rightarrow} z_1 u A v z_2 = x_1 A x_2 \Rightarrow x_1 \beta x_2 \ .$$

Если в правило грамматики $B \to uAv$ вместо A подставить β, получим правило $B \to u\beta v$ и длина вывода сократится на один шаг при неизменности выводимой цепочки. Следовательно, для того, чтобы удалить терминальное правило грамматики $A \to \beta$, необходимо рассмотреть следующие варианты :

а) для A больше нет правил, тогда во всех правых частях A заменяется на β;

б) для A есть другие правила, тогда добавляются новые правила, в которых A заменяется на β.

Пример 7.10. Рассмотрим грамматику, порождающую идентификаторы (см. 7.2), обозначив для наглядности символами b — букву, c — цифру:

$$\begin{array}{rl} G: & S \to bA \\ & A \to AZ|\varepsilon \\ & Z \to b|c. \end{array}$$

Удалим правила для Z, получим эквивалентную грамматику

$$\begin{array}{rl} G_1: & S \to bA \\ & A \to Ab|Ac|\varepsilon. \end{array}$$

7.5.7 Преобразования 7 и 8 — леворекурсивные и праворекурсивные правила

Некоторые специальные методы грамматического разбора неприменимы к леворекурсивным или праворекурсивным грамматикам, поэтому рассмотрим устранение левой или правой рекурсии. Следует отметить, что бесконечный язык порождается грамматикой с конечным числом правил только благодаря рекурсии, следовательно, вообще избавиться от рекурсии в правилах невозможно. Можно лишь преобразовать один вид рекурсии в другой.

Теорема 7.12. Для любой леворекурсивной КС–грамматики существует эквивалентная праворекурсивная.

Доказательство. Пусть нетерминал A леворекурсивен, т.е. правила для него имеют вид

$$A \to Ax_1|Ax_2|\ldots|Ax_p|w_1|w_2|\ldots|w_r, \tag{7.6}$$

где x_i и w_j — цепочки над $V_T \cup V_N$. Введем дополнительные нетерминалы B и D и заменим (7.6) на эквивалентные правила

$$\begin{array}{ll} A \to & AB|D \\ B \to & x_1|x_2|\ldots|x_p \\ D \to & w_1|w_2|\ldots|w_r. \end{array}$$

Вывод из A имеет вид $A \Rightarrow AB \Rightarrow ABB \Rightarrow ... \Rightarrow AB^* \Rightarrow DB^*$ и, следовательно, для A можно определить эквивалентные правила

$$\begin{array}{ll} A \to & DK \\ K \to & BK|\varepsilon. \end{array}$$

Выполняя подстановку B и D в эти правила, получим праворекурсивные правила

$$\begin{array}{l} A \to w_1K|w_2K|...|w_rK \\ K \to x_1K|x_2K|...|x_pK|\varepsilon\ . \end{array}$$

□

Пример 7.11. Для грамматики

$$\begin{array}{ll} G: & S \to SaA|ab \\ & A \to Aab|aAb|c \end{array}$$

можно построить эквивалентную нелеворекурсивную

$$\begin{array}{ll} G_1: & S \to abT \\ & T \to aAT|\varepsilon \\ & A \to aAbR|cR \\ & R \to abR|\varepsilon. \end{array}$$

Аналогично можно построить алгоритм устранения правой рекурсии и доказать эквивалентность соответствующего преобразования.

Теорема 7.13. Для любой праворекурсивной КС–грамматики существует эквивалентная леворекурсивная.

Доказательство. Пусть нетерминал A праворекурсивен, т.е. правила для него имеют вид

$$A \to x_1A|x_2A|\ldots|x_pA|w_1|w_2|\ldots|w_r, \tag{7.7}$$

где x_i и w_j — цепочки над $V_T \cup V_N$. Введем дополнительные нетерминалы B и D и заменим (7.7) на эквивалентные правила

$$A \to BA|D$$
$$B \to x_1|x_2|\ldots|x_p$$
$$D \to w_1|w_2|\ldots|w_r.$$

Вывод из A имеет вид $A \Rightarrow BA \Rightarrow BBA \Rightarrow ... \Rightarrow B^*A \Rightarrow B^*D$ и, следовательно, для A можно определить эквивалентные правила

$$A \to KD$$
$$K \to KB|\varepsilon.$$

Выполняя подстановку B и D в эти правила, получим леворекурсивные правила

$$A \to Kw_1|Kw_2|...|Kw_r$$
$$K \to Kx_1|Kx_2|...|Kx_p|\varepsilon.$$

□

7.6 Теорема о языке $a^nb^nc^n$

В 7.2 мы сформулировали, но не доказали теорему о языке $a^nb^nc^n$. Рассмотрим теперь ее доказательство. Сразу отметим, что при доказательстве этой теоремы мы будем использовать только понятие выводимости и определение дерева грамматического разбора, не пытаясь применить полученные из теоремы 7.2 выводы о незамкнутости семейства КС–языков относительно операций пересечения, дополнения и разности.

Лемма 7.1. Для любой КС – грамматики, порождающей бесконечный язык, существуют такие натуральные числа p и q, что каждая цепочка $w \in L(G), |w| > p$, может быть представлена в виде $w = xuyvz$, где $|uv| > q$ и для любого $n > 0$ цепочка $xu^nyv^nz \in L(G)$.

Доказательство. Рассмотрим всевозможные выводы $S \overset{*}{\Rightarrow} t$ терминальных цепочек $t \in V_T^*$ из аксиомы S, удовлетворяющие тем условиям, что в дереве разбора цепочки t на пути из корня в любой лист один и тот же нетерминал не встречается два раза. Возьмем в качестве числа p максимальную длину таких цепочек t. Возьмем произвольную цепочку w такую, что $|w| > p$ и $S \overset{*}{\Rightarrow} w$. По построению числа p в дереве вывода цепочки w на каком–то пути из S в лист существует повторяющийся нетерминал A_i (см. рис. 7.6).

Мы знаем, что любая грамматика может быть преобразована так, что она является неукорачивающей и не содержит правил вида нетерминал–нетерминал, следовательно, $|uv| > 0$. Возьмем в качестве числа $q = |uv|$. Тогда существует вывод $A_i \overset{*}{\Rightarrow} u^nA_iv^n$ для любого n. Следовательно, в G существует вывод

$$S \overset{*}{\Rightarrow} xA_iz \overset{*}{\Rightarrow} xu^nA_iv^nz \Rightarrow xu^nyv^nz.$$

□

Теорема 7.2. Язык $a^nb^nc^n$, $n \geq 0$ — не КС–язык.

Доказательство. Пусть $a^nb^nc^n$ — КС–язык, т.е. существует КС–грамматика G, его порождающая. По доказанной лемме существует такая цепочка $xvyuz \in L(G)$, что $xv^kyu^kz \in L(G)$. Рассмотрим все варианты определения подцепочки v в слове

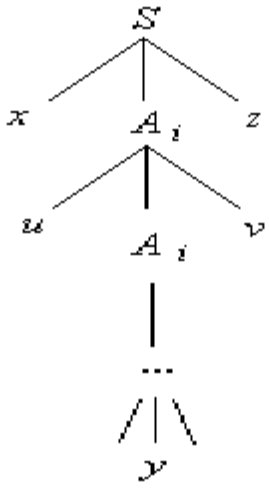


Рис. 7.6: Повторяющийся нетерминал A_i в дереве вывода.

$a^n b^n c^n$: a^l, $a^l b^m$, b^l, $b^l c^m$, c^m, $a^l b^m c^p$. Покажем, что ни один из этих вариантов определения v не может иметь места.

Пусть $v = a^l$, тогда $a^t (a^l)^k y u^k z \in L(G)$, что невозможно из–за нарушения соотношения между равным числом символов a и b или a и c.

Пусть $v = a^l b^m$, тогда $a^t (a^l b^m)^k y u^k z L(G)$, чего быть не может, т.к. при $k > 1$ в полученной цепочке после символа b встречается символ a.

Аналогично можно показать невозможность оставшихся вариантов определения v. Следовательно, язык $a^n b^n c^n$ не может порождаться КС–грамматикой. Пример грамматики типа 0, порождающей этот язык, приведен в примере 7.2. □

7.7 Контрольные вопросы к разделу

1. Поясните понятие терминальных символов грамматики.

2. Чем терминальные символы отличаются от нетерминальных?

3. Чем отличаются КС–грамматики от грамматик непосредственно составляющих?

4. Какой класс грамматик является более широким: грамматики типа 0 или грамматики типа 2?

5. Как построить грамматику для итерации языка, заданного КС–грамматикой?

6. Как построить грамматику для объединения языков, заданных КС–грамматиками?

7. Замкнуто ли множество КС–языков относительно операции пересечения?

8. Можно ли для произвольной КС–грамматики построить эквивалентную неукорачивающую? Если можно, то как это сделать?

9. Как в заданной КС–грамматике избавиться от правил с одинаковыми правыми частями?

10. Чем характеризуется восходящая стратегия разбора?

11. Приведите пример КС–грамматики, для которой возникают затруднения при нисходящем грамматическом разборе.

12. Какая КС-грамматика называется приведенной?
13. К какому классу языков относится язык $a^n b^n c^n$?
14. Что называется деревом грамматического разбора?
15. Как устранить левую рекурсию в правилах КС–грамматики?
16. Можно ли устранить центральную рекурсию в правилах КС–грамматики?
17. Устраните правую рекурсию в грамматике

$$G : S \to SaaSbS|SbS|bSbb|aaSb.$$

18. Устраните левую рекурсию в грамматике задания 17.
19. Какую цель преследуют при устранении правил вида "нетерминал — нетерминал"?
20. Приведите пример грамматики, в которой имеет смысл устранить некоторые правила с терминальной правой частью.

7.8 Упражнения к разделу

Задание. Построить КС-грамматику, порождающую заданный язык и привести пример дерева разбора в построенной грамматике.

7.8.1 Задача

Дан язык

$$(ab)^{n+1}c^{3n} \cup (a^*bc^+ \cup (ad)^+)^*, n \geq 0.$$

Построить КС–грамматику и привести пример дерева разбора в построенной грамматике.

Решение. Обозначим символом S аксиому грамматики. Язык $(ab)^{n+1}c^{3n} \cup (a^*bc^+ \cup (ad)^+)^*$ построен с помощью операции объединения языков $(ab)^{n+1}c^{3n}$ и $(a^*bc^+ \cup (ad)^+)^*$, поэтому для аксиомы S необходимо определить два правила

$$S \longrightarrow A|B,$$

где символы A и B соответствуют указанным языкам. Рассмотрим язык $(ab)^{n+1}c^{3n}$, который должен выводиться из нетерминального символа A. Для того, чтобы получить правила для нетерминала A, запишем равенство $(ab)^{n+1}c^{3n} = (ab)^n ab(ccc)^n$ и воспользуемся правилом самовставленя нетерминального символа. Получаем правила грамматики:

$$A \longrightarrow abAccc|ab.$$

Язык $(a^*bc^+ \cup (ad)^+)^*$, соответствующий нетерминалу B, получен с помощью операции итерации языка $a^*bc^+ \cup (ad)^+$. Поэтому записываем правила граммматики:

$$B \longrightarrow BT|\varepsilon$$

$$T \longrightarrow R|F$$

Языки, соответствующие нетерминалам R и F, получаются с помощью операций конкатенации, итерации и усеченной итерации над элементарными цепочками. Таким

образом, получаем КС-грамматику

$$\begin{aligned} G: \quad & S \longrightarrow A|B \\ & A \longrightarrow abAccc|ab \\ & B \longrightarrow BT|\varepsilon \\ & T \longrightarrow R|F \\ & R \longrightarrow MbK \\ & M \longrightarrow Ma|\varepsilon \\ & K \longrightarrow Kc|c \\ & F \longrightarrow Fad|\varepsilon. \end{aligned}$$

Можно несколько упростить грамматику, удалив нетерминал R:

$$\begin{aligned} G: \quad & S \longrightarrow A|B \\ & A \longrightarrow abAccc|ab \\ & B \longrightarrow BT|\varepsilon \\ & T \longrightarrow MbK|F \\ & M \longrightarrow Ma|\varepsilon \\ & K \longrightarrow Kc|c \\ & F \longrightarrow adF|\varepsilon. \end{aligned}$$

Построим дерево разбора цепочки *ababccc*. Эта цепочка получена с помощью вывода

$$S \Rightarrow A \Rightarrow abAccc \Rightarrow ababccc,$$

поэтому дерево грамматического разбора этой цепочки имеет вид, представленный на рис. 7.7.

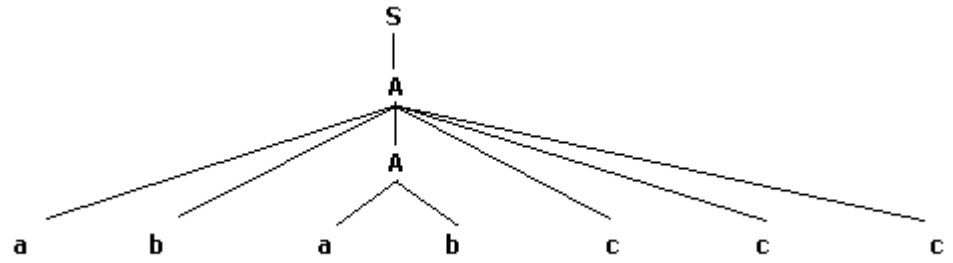


Рис. 7.7: Дерево разбора, соответствующее выводу $S \Rightarrow A \Rightarrow abAccc \Rightarrow ababccc$

Построим теперь дерево разбора цепочки *aabcadadad*. Эта цепочка может быть получена с помощью различных выводов:

$$\begin{gathered} S \Rightarrow B \Rightarrow BT \Rightarrow BTT \Rightarrow BTTT \Rightarrow BTTTT \Rightarrow TTTT \Rightarrow \\ \Rightarrow MbKTTT \Rightarrow MabKTTT \Rightarrow MaabKTTT \Rightarrow aabKTTT \Rightarrow \\ \Rightarrow aabcTTT \Rightarrow aabcFTT \Rightarrow aabcadFTT \Rightarrow aabcadTT \Rightarrow aabcadFT \Rightarrow \\ \Rightarrow aabcadadFT \Rightarrow aabcadadT \Rightarrow aabcadadF \Rightarrow \\ \Rightarrow aabcadadadF \Rightarrow aabcadadad. \end{gathered} \tag{7.7}$$

или

$$\begin{gathered} S \Rightarrow B \Rightarrow BT \Rightarrow BTT \Rightarrow TT \Rightarrow MbKT \Rightarrow \\ \Rightarrow MabKTT \Rightarrow MaabKT \Rightarrow aabKT \Rightarrow aabcT \Rightarrow aabcF \Rightarrow \\ \Rightarrow aabcadF \Rightarrow aabcadadF \Rightarrow aabcadadadF \Rightarrow aabcadadad. \end{gathered} \tag{7.8}$$

Очевидно, что грамматика, которую мы построили, является неоднозначной. Соответствующие этим выводам деревья разбора представлены на рис. 7.8 и 7.9.

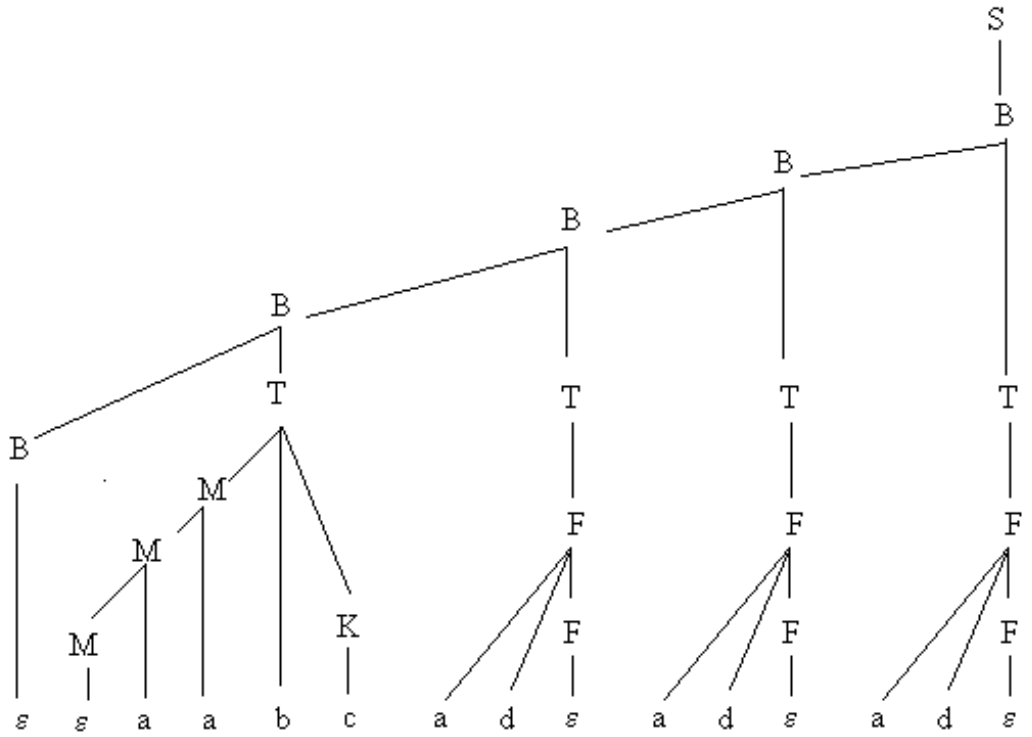


Рис. 7.8: Дерево разбора для вывода (7.7).

7.8.2 Варианты заданий

1. $(abc^+ \cup a^n cccb^{2n+1} \cup (aba)^+)^*, n \geq 0$
2. $((acb)^* ad^+ (a^+ ba)^{n+1} cb^{3n+1} \cup (a^* bc^+)^*)^* \cup a^* b^+, n \geq 0$
3. $ac^* ad^+ (a^+ ba)^{2n+1} cb^n \cup (ba^* c^* b)^* \cup (ab)^{2k} b^k, n, k \geq 0$
4. $c^+ d^+ (a^* ba)^{n+1} cb^{4n} \cup ((ba^* c^* b)^+ \cup (ca^+ b)^{2k} b^k)^*, n \geq 0, k \geq 1$
5. $(cca^{2n+2} ab^+ (bc)^n)^* \cup (a^* b^+ \cup (bc)^k dda^{2k})^+, n, k \geq 1$
6. $ac^* ad^+ (a^+ ba)^{2n+1} cb^n \cup (ba^* c^* b)^* \cup (ab)^{2k} b^k, n, k \geq 1$
7. $(bc^* \cup (ad)^n cb^{2n+1} \cup (ba^*)^+)^*, n \geq 0$
8. $(a^* ba)^{n+1} cb^{2n} \cup ((ba^* c^* b)^+ \cup (ca^* b)^{3k} ccb^k)^*, n \geq 1, k \geq 0$
9. $(ac^* ad^+ (a^+ ba)^{2n+1} cb^{3n} \cup (a^* bc^+)^*)^* \cup a^k b^k, n, k \geq 0$
10. $ac^+ ad^+ (a^+ ba)^{2n+1} cb^n \cup (ba^* c^* b)^+ \cup (ca^+ b)^{2k} b^k, n \geq 0, k \geq 1$
11. $(a^{2n+2} b^+ (bc)^n \cup (a^* b^+ \cup a + c^* \cup (bc)^k dda^{2k})^+)^*, n, k \geq 0$
12. $(bc^* ad \cup (a^+ ba)^n cb^{3n+2} \cup (ba^*)^*)^*, n \geq 0$
13. $(a^{2n+2} b^+ (bc)^n \cup (a^m b^m \cup c + b^* \cup (aac)^{k+2} dda^{2k})^+)^*, n, k, m \geq 0$
14. $(ac^* ad^+ (a^+ ba)^{2n+1} cb^n \cup (a^* bc^+)^*)^+ \cup a^{2k} b^k, n, k \geq 0$
15. $(a^{2n+2} b^+ (bc)^n (a^* b^* \cup c + b^* \cup (aac)^{k+2} dda^{2k})^+)^*, n, k \geq 0$
16. $(ab^{n+2} c(bc)^n)^* \cup (a^+ b^+ \cup abc^k dda^{2k})^+, n, k \geq 1$
17. $ac^+ ad^+ (a^+ ba)^{2n+1} cb^n \cup ((ba^* c^* b)^+ \cup (ca^+ b)^{2k} b^k)^*, n \geq 0, k \geq 1$
18. $(a^{2n+2} b^+ (bc)^n)^+ \cup (a^* b^+ \cup a + c^* \cup (bc)^k dda^{2k})^+, n, k \geq 1$
19. $((acb)^* ad^+ \cup (a^+ ba)^{n+1} cb^{3n+1} \cup (a^* bc^+)^*)^*, n \geq 0$
20. $(ac^* ad^+ (a^+ ba)^{2n+1} cb^n)^+ \cup (a^* bc^+)^* \cup a^{2k} b^k, n, k \geq 0$

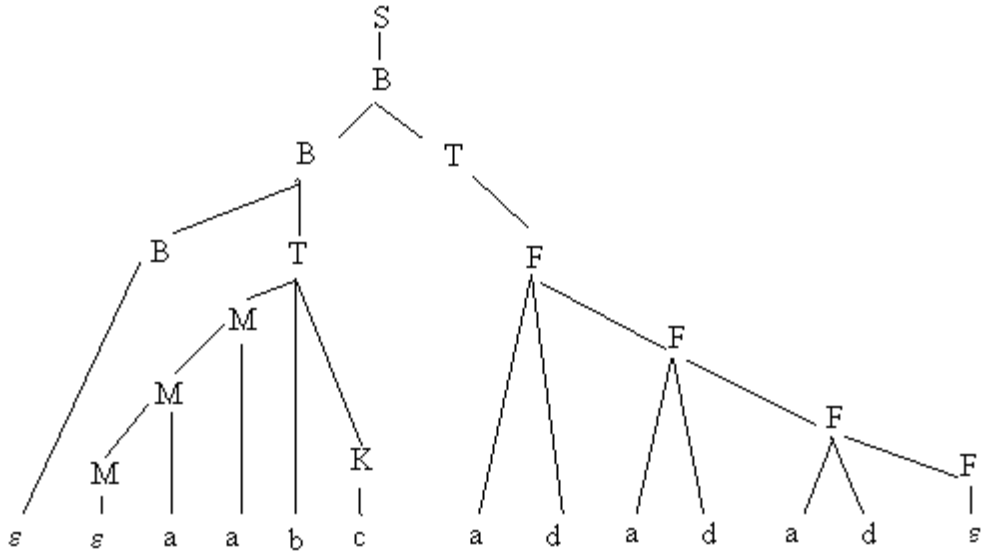


Рис. 7.9: Дерево разбора для вывода (7.8).

7.9 Тесты для самоконтроля к разделу

1. Какой язык порождает грамматика

$$G: \quad S \longrightarrow ASa|b$$
$$A \longrightarrow aA|c$$

Варианты ответов:

а) $(a^*c)^n ba^n, n \geq 0$;
б) $(a^*c)^n ba^n, n > 0$;
в) $(a^*c)^* ba^*$;
г) $a^*c^*b^*a^*$;
д) $a^n c^n b^n a^n, n \geq 0$;
е) $a^n c^n b^n a^n, n > 0$;
ж) $a^n cbca^n, n > 0$;
ж) $a^n cbca^n, n \geq 0$;

Правильный ответ: а.

2. Какой из перечисленных ниже языков не является регулярным:

а) $(ac)^n ba^m,\ n, m > 0$;
б) $(a^*c)^n ba^n,\ n > 0$;
в) $(a^*c)^* ba^*$;
г) $a^*c^*b^*a^*$;
д) $a^n cb^m a^k,\ n, m, k > 0$;
ж) $a^n cbca^+,\ n > 0$;
ж) $a^* cbca^n,\ n \geq 0$;

Правильный ответ: б.

3. Какие символы являются продуктивными в грамматике

$$\begin{array}{rl} G: & S \longrightarrow ASa|bD|F \\ & A \longrightarrow aA|cAS|DaF \\ & B \longrightarrow aA|\varepsilon \\ & D \longrightarrow aA|cB \\ & F \longrightarrow aA|cF|Fa \end{array}$$

Варианты ответов:

а) все нетерминалы S, A, B, D, F продуктивны;
б) S;
в) S, B, D;
г) S, A, D, F;
д) A, B, D, F;
е) B, D;
ж) все нетерминалы непродуктивны; .

Правильный ответ: в.

4. Дана КС–грамматика:

$$\begin{array}{rl} G_3: & S \longrightarrow aSA|Ab|Bc \\ & A \longrightarrow aAb|\varepsilon \\ & B \longrightarrow AA|aBc. \end{array}$$

Какая неукорачивающая КС–грамматика ей эквивалентна?

Варианты ответов:

$$\begin{array}{rl} \text{а) } G_1: & S \longrightarrow aSA|Ab|Bc \\ & A \longrightarrow aAb|ab \\ & B \longrightarrow AA|aBc \\ \text{б) } G_2: & S \longrightarrow aSA|Ab|aS|b|Bc|c \\ & A \longrightarrow aAb|ab \\ & B \longrightarrow AA|aBc \\ \text{в) } G_3: & S \longrightarrow aSA|aS|b|c \\ & A \longrightarrow aAb|ab \\ & B \longrightarrow aBc|ac \\ \text{г) } G_4: & S \longrightarrow aSA|Ab|aS|b|Bc|c \\ & A \longrightarrow aAb|ab \\ & B \longrightarrow AA|aBc|ac|A \end{array}$$

д) Для заданной КС–грамматики нельзя построить эквивалентную неукорачивающую.

Правильный ответ: г.

5. Какой язык из перечисленных ниже не является контекстно–свободным:

а) $(a^*c)^n ba^n, n \geq 0$;
б) $(a^*c)^n b^{n+3} \cup a^{2n}, n > 0$;
в) $a^*c^*b^*$;
г) $a^n b^n \cup c^n, n > 0$;
д) $a^n c^n b^n a^n, n \geq 0$.

Правильный ответ: д.

Глава 8

ЯЗЫКИ И АВТОМАТЫ

8.1 Понятие автомата и типы автоматов

Автомат — это алгоритм, определяющий некоторое множество и, возможно, преобразующий это множество в другое множество. В главе 2 мы уже рассмотрели один из видов автоматов — машины Тьюринга. Машина Тьюринга имеет управляющее устройство, которое находится в некотором состоянии из конечного множества состояний, и бесконечную ленту, предназначенную для хранения информации. Сразу отметим, что лента машины Тьюринга используется для нескольких целей:

а) перед началом работы на ней записаны исходные данные;

б) в процессе работы лента используется как рабочая память, где хранятся необходимые для работы данные;

в) после завершения работы на ленте находится результат вычислений.

Учитывая различный характер использования ленты машины Тьюринга, будем считать, что в общем случае автомат может иметь три ленты: для исходных данных, для результатов и рабочую ленту - см. рис. 8.1. Рассмотрим сначала неформальное понятие автомата в общем виде.

Автомат имеет входную ленту, управляющее устройство с конечной памятью для хранения номера состояния из некоторого конечного множества состояний, может иметь вспомогательную (или рабочую) ленту, а также может иметь выходную ленту. Автомат без выхода часто называется распознавателем, автомат с выходом — преобразователем.

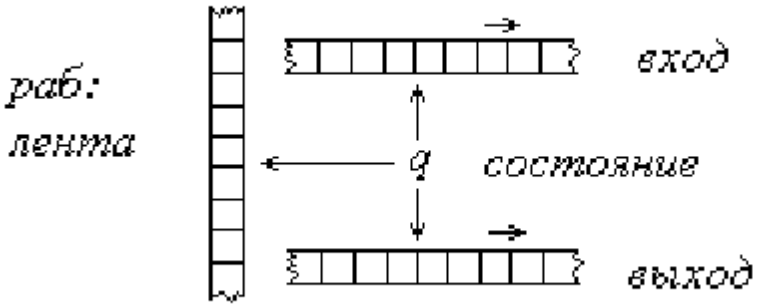


Рис. 8.1: Схематическое изображение автомата.

Входную ленту можно рассматривать как линейную последовательность ячеек, причем каждая ячейка содержит точно один символ из некоторого конечного входного алфавита. Лента бесконечна, но занято на ней в каждый момент только конечное число ячеек. Самую левую и самую правую ячейки из занятой области могут занимать специальные концевые маркеры; маркер может стоять только на правом конце ленты; маркеров может не быть совсем.

Входная головка в каждый момент времени читает (или, как иногда говорят, обозревает) одну ячейку входной ленты. За один шаг работы автомата входная головка может сдвинуться на одну ячейку вправо или остаться неподвижной. Входная головка только читает символы с входной лента, т.е. в процессе работы автомата символы на входной ленте не меняются. Более того, входная головка не может возвращаться к уже прочитанному символу. Чтение символа с входной ленты всегда означает сдвиг головки на один символ вправо.

Так же как и входная лента, рабочая и выходная ленты представляют собой последовательность ячеек, в каждой из которых может находиться только один символ некоторого алфавита. Рабочая лента — вспомогательное хранилище информации. Данные с рабочей ленты могут читаться автоматом, могут и записываться на нее. Именно сложность рабочей ленты определяет сложность автомата. Чем проще рабочая лента, тем проще автомат. Очевидно, что самый алгоритмически простой автомат не иеет рабочей ленты совсем.

Управляющее устройство — это программа, управляющая поведением автомата. Управляющее устройство представляет собой конечное множество состояний вместе с отображением, которое описывает, как меняются состояния в соответствии с текущим входным символом, читаемым входной головкой, и текущей информацией, извлеченной с рабочей ленты. Управляющее устройство определяет также, в каком направлении сдвинуть рабочую и входную головки и какую информацию записать на рабочую ленту.

Автомат работает, выполняя некоторую последовательность тактов. В начале такта читается текущий входной символ и исследуется информация на рабочей ленте. В зависимости от прочитанной информации и текущего состояния определяются действия автомата:

1) входная головка сдвигается на одну позицию вправо или остается в исходном положении;

2) на рабочую ленту может записываться некоторая информация;

3) изменяется состояние управляющего устройства;

4) на выходную ленту (если она имеется) может записываться символ.

Поведение автомата удобно описывать в терминах конфигураций автомата. Конфигурация — это как бы мгновенный снимок автомата, на котором изображены:

1) состояние управляющего устройства;

2) содержимое входной ленты с положением входной головки;

3) содержимое рабочей ленты вместе с положением рабочей головки;

4) содержимое выходной ленты, если она есть.

Управляющее устройство может быть недетерминированным и детерминированным. Если управляющее устройство недетерминированное, то для каждой конфигурации существует конечное множество возможных следующих тактов, любой из которых автомат может сделать, исходя из этой конфигурации. Управляющее устройство называется детерминированным, если для каждой конфигурации существует не более одного возможного следующего такта.

Как уже отмечалось, сложность рабочей ленты определяет сложность автомата. Обычно рассматривают следующую иерархию автоматов по уровням сложности:

— *машина Тьюринга*, имеющая бесконечную рабочую ленту, по которой головка может перемещаться в произвольном направлении, считывая содержимое ячеек ленты и записывая туда новые символы; при этом нет никаких ограничений ни на длину рабочей ленты, ни на способ доступа к ее ячейкам;

— *линейно–ограниченный автомат*, который по способу доступа к ячейкам рабочей ленты не отличается от машины Тьюринга, но имеет ограничение на длину рабочей ленты: если на вход автомата поступает цепочка x длины $|x|$, то длина рабочей ленты ограничена линейной функций от $|x|$;

— *автомат с магазинной памятью* (МП–автомат), у которого в качестве рабочей ленты используется магазин — память, работающая по принципу LIFO (Last In — First Out, или последний записанный — первый считанный);

— *конечный автомат* (КА), не имеющий рабочей ленты.

Автоматом, обладающим наибольшей алгоритмической сложностью (как эквивалент алгоритма в соответствии с тезисом Тьюринга), является машина Тьюринга. В главе 2 мы рассматривали машину Тьюринга с одной лентой, которая в начальный момент является входной, в заключительном состоянии является выходной, в процессе работы используется как рабочая. Такая машина Тьюринга эквивалентна машине Тьюринга с расщепленной лентой на три ленты в соответствии с их назначением : входная, выходная, рабочая.

Автомат без выходной ленты называется *распознавателем*, т.к. он только проверяет правильность исходных данных, т.е. распознает, принадлежит ли входная цепочка заданному множеству L или нет.

Автомат с выходной лентой является *преобразователем*, т.к. входную цепочку x при условии $x \in L$ этот автомат преобразует в новую цепочку y некоторого другого языка L_1.

Сложность автомата уменьшается с уменьшением сложности рабочей ленты: у машины Тьюринга лента неограничена в обе стороны, у линейно–ограниченного автомата длина рабочей ленты является линейной функцией длины входной цепочки, у МП–автомата рабочая лента работает по принципу магазина, ограничивая тем самым направление чтение и записи на ленту, у конечного автомата рабочая лента отсутствует. В пределах данной главы мы будем иметь дело только с распознавателями и будем называть их для краткости просто автоматами.

8.2 Формальное определение автомата

Если мы рассмотрим автоматы без рабочей ленты (конечные автоматы), то поведение такого автомата определяется только его состояниями и входными символами. Тогда для того, чтобы задать конечный автомат, необходимо определить множество его состояний K, входной алфавит X и функцию переходов как отображение $K \times X$ в множество K. В зависимости от того, зафиксировано или нет начальное состояние автомата и множество его конечных состояний, имеет автомат выходную ленту или нет, можно рассматривать различные типы автоматов.

Если введем в рассмотрение автомат с рабочей лентой, то поведение такого автомата принципиально отличается от поведения автомата без рабочей ленты. Во–первых, дополнительно необходимо определить алфавит рабочей ленты Z, во–вторых, функция переходов существенно сложнее, т.к. поведение автомата в этом случае определяется не только текущим состояним и входным символом, но и содержимым рабочей ленты.

Определение 8.1. Неинициальный конечный автомат с выходом — это пятерка

$(K, X, Y, \delta, \gamma)$, где K, X, Y — алфавиты (называемые соответственно множеством состояний, входным и выходным алфавитом), δ — функция переходов — отображение $K \times X \to K$, γ — функция выходов — отображение $K \times X \to Y$.

Функционирование автомата можно задать множеством команд вида $qx \to py$, где $q, p \in K, x \in X$ — входной символ, $y \in Y$ — выходной символ.

Пусть на некотором такте t_i блок управления находится в состоянии q и с входной ленты читается символ x. Если в множестве команд имеется команда $qx \to py$, то на том же такте t_i на выходную ленту записывается символ y, а к следующему такту t_{i+1} блок управления перейдет в состояние p. Если команды $qx \to py$ в множестве команд нет, то автомат оказывается блокированным, т.е. он никак не реагирует на символ, принятый в момент t_i, а также перестает воспринимать символы, подаваемые на вход в последующие моменты.

В соответствии с определением 8.1 в начальный момент состояние автомата может быть произвольным. Если зафиксировано некоторое начальное состояние, то такой автомат называется инициальным автоматом:

$$\begin{array}{ll} q(0) = & q_0, \\ q(t+1) = & \delta(q(t), x(t)), \\ y(t) = & \gamma(q(t), x(t)), \end{array}$$

где $q(i) \in K$ — состояние на такте $i, x(i) \in X$ и $y(i) \in Y$ — соответственно входные и выходные символы на такте i.

Мы будем рассматривать, как правило, инициальные автоматы, поэтому основным будем считать следующее определение автомата.

Определение 8.2.

Инициальный конечный автомат с выходом — это шестерка

$$A = (K, X, Y, \delta, \gamma, q_0, F),$$

где q_0 — начальное состояние, F — множество заключительных состояний, а остальные элементы имеют тот же смысл, что и в определении 8.1.

Формальное определение автомата с рабочей лентой, кроме алфавита рабочей ленты Z, должно фиксировать вид функции переходов как частный случай отображения $K \times X \times Z$ в множество $K \times Z \times R$, где R — множество, определяющее направление движения головки по рабочей ленте.

Указанные определения соответствуют детерминированной функции переходов. Можно как для конечных автоматов, так и для автоматов более общего вида перейти к недетерминированным автоматам, рассматривая для конечных автоматов отображение $K \times X \to 2^K$ и для автоматов более обшего вида отображение $K \times X \times Z \to 2^{K \times Z \times R}$.

Таким образом, можно дать следующие два определения машины Тьюринга с расщепленной лентой, первое из которых является эквивалентом определения из главы 2, а второе определяет недетерминированную машину Тьюринга. Оба варианта определения рассмотрим для инициального автомата.

Определение 8.3. Инициальной детерминированной машиной Тьюринга называется

$$T = (K, X, Y, Z, R, \delta, \gamma, q_0, F), \text{ где}$$

K, X, Y, Z, R — алфавиты (называемые соответственно множеством состояний, входным и выходным алфавитом, алфавитом рабочей ленты и множеством направлений движения головки по рабочей ленте), δ — функция переходов — отображение $K \times X \times Z \to K \times Z \times R$, γ — функция выходов — отображение $K \times X \times Z \to Y$.

Определение 8.4. Инициальной недетерминированной машиной Тьюринга называется

$$T = (K, X, Y, Z, R, \delta, \gamma, q_0, F), \text{ где}$$

K, X, Y, Z, R имеют тот же смысл, что в определении 8.3, а δ — функция переходов — отображение $K \times X \times Z \to 2^{K \times Z \times R}$, γ — функция выходов — отображение $K \times X \times Z \to 2^Y$.

Задача грамматического разбора заключается в нахождении вывода заданной цепочки в заданной грамматике и в построении дерева вывода этой цепочки. Для решения этой задачи будем использовать автоматы.

Языки могут быть заданы двумя способами: грамматиками и автоматами. Грамматики являются порождающим способом определения языка, т.к. с помощью грамматик формализуется способ порождения (или вывода) всех цепочек языка. Автоматы являются распознающим способом формализации языка, так с помощью автомата можно определить множество цепочек, которое распознает этот автомат при переходе из начального состояния в заключительное. Поэтому, если автоматы используются для определения языков, то необходимо рассматривать инициальные автоматы.

В данной главе мы покажем, что различным по сложности автоматам соответствуют разные типы языков. Простейшим типом автоматов являются конечные автоматы. Докажем, что этим автоматам соответствуют линейные грамматики — леволинейные и праволинейные. Как уже отмечалось, для определения синтаксиса языков программирования обычно используются КС–грамматики. Докажем, что КС–грамматикам соответствуют МП–автоматы.

8.3 Конечные автоматы

Конечный автомат имеет входную ленту, с которой за один такт может быть прочитан один входной символ. Возврат по входной ленте не допускается, как, впрочем, он не допускается обычно для любого типа автоматов с разделенной по функциональному назначению лентой. В отличие от предшествующего параграфа, где было введено определение автомата в общем виде, сейчас мы перейдем к определению конечного инициального автомата–распознавателя.

Определение 8.5. Конечным автоматом называется шестерка вида

$$A = (K, \Sigma, \delta, p_0, F), \text{ где}$$

K — конечное множество состояний,

Σ — алфавит,

δ — функция переходов, в общем случае — недетерминированное отображение $\delta : K \times \Sigma \to 2^K$,

p_0 — начальное состояние, $p_0 \in K$,

F — множество заключительных состояний, $F \subseteq K$.

Частным случаем конечных автоматов являются детерминированные конечные автоматы с функцией переходов $\delta : K \times \Sigma \to K$.

Любой автомат, в том числе и конечный, можно определить как формальную систему через состояния, символы, которые пишутся или читаются с ленты или нескольких лент, и набора команд. В частности, конечный автомат можно представить командами, графом, таблицей переходов и матрицей преходов.

Пример 8.1. Автомат, распознающий язык a^*bc^*, может быть представлен следующим списком команд:

$$p_0, a \to p_0,$$
$$p_0, b \to p_1,$$
$$p_1, c \to p_1,$$

где p_0 — начальное состояние, p_1 — заключительное состояние.

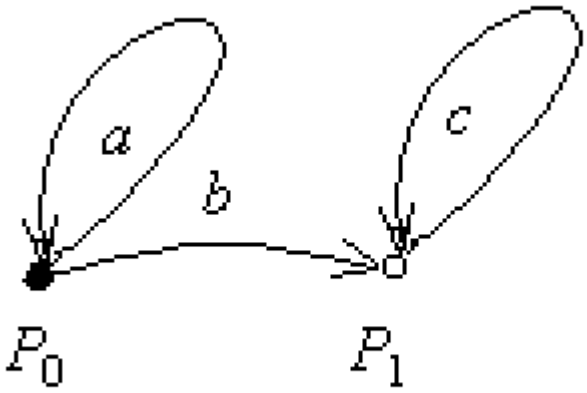
Этот же автомат можно задать матрицей переходов:

	p_0	p_1
p_0	a	b
p_1		c

Эквивалентное задание автомата таблицей переходов имеет вид:

	a	b	c
p_0	p_0	p_1	
p_1			p_1

Этот же автомат может быть представлен графом:



8.4 Регулярные множества

Определение 8.6. Язык, распознаваемый конечным автоматом

$$A = (K, \Sigma, \delta, p_0, F)$$

— это множество цепочек, читаемых автоматом A при переходе из начального состояния в одно из заключительных состояний:

$$L(A) = \{a_1 a_2 ... a_n | p_0 a_1 \to p_1, ..., p_{n-1} a_n \to p_n; p_n \in F\}$$

Определение 8.7. Автоматы называются эквивалентными, если совпадают распознаваемые ими языки.

Определение 8.8. Множество называется регулярным, если существует конечный детерминированный автомат, распознающий это множество.

Теорема 8.1. Для любого конечного автомата распознаваемое множество регулярно.

Доказательство. Формулировка теоремы означает, что для любого недетерминированного конечного автомата можно построить эквивалентный детерминированный автомат. Выполним такое построение. Пусть $A = (K, \Sigma, \delta, p_0, F)$ — произвольный недетерминированный конечный автомат. Построим новый автомат

$$A_1 = (K_1, \Sigma, \delta_1, [p_0], F_1) \ .$$

Множество состояний автомата A_1 состоит из всевозможных подмножеств состояний исходного автомата, причем, если хотя бы одно состояние исходного автомата A, принадлежащее составному состоянию автомата A_1, является заключительным, то новое составное состояние является заключительнвм для автомата A_1:

$$\begin{array}{ll} K_1 = & \{[p_{i_1}, ..., p_{i_j}] | p_{i_1}, ..., p_{i_j} \in K\}, \\ F_1 = & \{[p_{i_1}, ..., p_{i_k}] | p_{i_j} \in F\}. \end{array}$$

Функция переходов δ_1 формируется следующим образом : для каждого состояния $[p_{i_1}, ..., p_{i_k}]$ в левой части команды переход по некоторому символу a в новое состояние в правой части этой команды формируется как объединение всех состояний p_{j_m}, в которые возможен переход в A из всех $p_{i_l} (1 \leq l \leq k)$:

$$\begin{array}{ll} \delta_1 & = \{[p_{i_1}, ..., p_{i_t}]a \to [p_{j_1}, .., p_{j_r}] | \\ & \forall p_{i_l} \exists \ p_{j_t} (p_{i_l} a \to p_{j_t} \in \delta), \ \forall p_{j_t} \exists \ p_{i_l} (p_{i_l} a \to p_{j_t} \in \delta)\} \end{array}$$

В соответствии с определением языка, распознаваемого автоматом, для любой цепочки $x \in L(A)$ существует последовательность команд автомата A, переводящая его из начального состояния в заключительное при чтении цепочки x. Для каждой такой команды в δ найдется одна и только одна команда δ_1, читающая тот же символ, следовательно A_1 читает все цепочки, которые читает исходный автомат A. Обратно: для любой цепочки, читаемой автоматом A_1 по построению δ_1 существует одна или несколько последовательностей команд автомата A, выполняющая те же действия. Таким образом, $L(A) = L(A_1)$. $\square$

Для реализации рассмотренного алгоритма удобно использовать таблицу переходов автомата. Для простоты новые состояния лучше обозначать не множествами, а символами с индексами.

Очевидно, что алгоритм, рассмотренный в доказательстве теоремы 8.1, генерирует полное множество состояний, так что $|K_1| = 2^{|K|}$. В множество K_1 входят все возможные состояния, в том числе и не достижимые из начального состояния. Чтобы такие состояния не рассматривать, лучше последовательно строить переходы для всех вновь появляющихся состояний, начиная от состояния $[P_0]$.

Пример 8.2. Пусть исходная таблица переходов конечного автомата имеет вид:

	a	b	c	
P_0	P_0		P_0, P_1	нач.
P_1	P_1, P_3	P_0		
P_2			P_1	закл.
P_3		P_2	P_0	закл.

Выполняя операции объединения состояний, получим

	a	b	c	
P_0	P_0		P_{01}	нач.
P_{01}	P_{013}	P_0	P_{01}	
P_{013}	P_{013}	P_{02}	P_{01}	закл.
P_{02}	P_0		P_{01}	закл.

8.5 Минимизация конечных автоматов

Определение 8.9. Два состояния p_1 и p_2 конечных автоматов соответственно A_1 и A_2 называются n–эквивалентными, если, начиная действие из этих состояний, автомат распознает совпадающие множества цепочек длины не более n.

Сразу следует отметить, что в этом определении для общности рассматриваются два автомата. Однако, ничто не препятствует рассматривать и два состояния одного автомата. В этом случае появляется возможность сравнивать состояния одного автомата и, возможно, заменять их на одно состояние.

Пример 8.3. Рассмотрим два автомата, представленных на рис. 8.2.

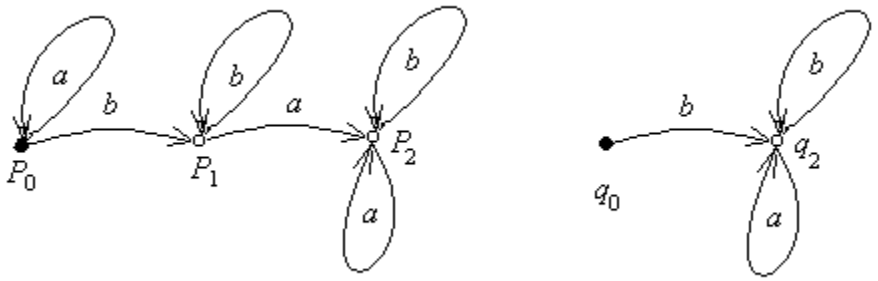


Рис. 8.2: Автоматы с 2–эквивалентными состояниями.

Пусть $n = 2$, тогда, например, из состояний p_0, q_0, p_1, q_2 распознаются следующие множества цепочек длины не более n:

$$p_0 \Rightarrow b, ab, bb, ba, \qquad q_0 \Rightarrow b, ba, bb,$$
$$p_1 \Rightarrow b, a, bb, ba, ab, aa, \quad q_2 \Rightarrow b, a, bb, ba, ab, aa.$$

Состояния p_1 и q_2 являются 2–эквивалентными, а, например, состояния p_0 и q_0 не являются 2–эквивалентными, но являются 1–эквивалентными.

Определение 8.10. Два состояния p_1 и p_2 конечных автоматов соответственно A_1 и A_2 называются эквивалентными, если они n–эквивалентны для любого n.

Определение 8.11. Конечный автомат называется минимальным, если никакие его два состояния не эквивалентны друг другу.

Теорема 8.2. Для любого конечного автомата $A = (K, \Sigma, \delta, p_0, F)$ можно построить эквивалентный минимальный автомат, выполняя не более $|K|$ разбиений матрицы переходов.

Доказательство. Доказательство основано на выделении групп n–эквивалентных состояний. Пусть имеется матрица переходов автомата A. Сразу можно заметить, что все заключительное и все незаключительное состояния не эквивалентны друг другу, т.к. в заключительном состоянии может закончиться процесс распознавания, а в незаключительном он обязан продолжиться. Таким образом, в заключительном состоянии распознается пустая цепочка ε, а в незаключительном состоянии

— пустое множество цепочек. Это означает 0–неэквивалентность заключительных и незаключительных состояний. Тогда уже на первом шаге алгоритма определения эквивалентных состояний мы можем разбить состояния на две группы: заключительные и незаключительные состояния как не эквивалентные для $n = 0$.

Пусть теперь построены группы n–эквивалентных состояний $H_1, H_2, ..., H_t$. В соответствии с определением, из каждого состояния $p_k \in H_i$ распознается одно и то же множество цепочек M_i длины не более n. Перейдем к анализу $(n+1)$–эквивалентности.

Если все группы эквивалентности содержат точно по одному состоянию, то любая пара сотояний не эквивалентна друг другу, исходный автомат является минимальным. Пусть хотя бы одна группа n–эквивалентных состояний H_i содержит более одного состояния. Рассмотрим все переходы из $p_k \in H_i$ и из $p_m \in H_i$ в соответствии с матрицей переходов как внутри группы эквивалентности H_i, так и в некоторую другую группу H_j. Для того, чтобы состояния p_k и p_m оставались $(n+1)$–эквивалентными, необходимо и достаточно, чтобы переход в каждую группу n–эквивалентности H_j осуществлялся по одним и тем же символам $\{a_{j1}, a_{j2}, ...a_{jr}\}$: тогда для каждого из этих состояний допускается одно и то же множество цепочек $\{a_{j1}, a_{j2}, ...a_{jr}\} \cdot M_j$. Другими словами, в каждой подматрице, соответствующей группе эквивалентности, каждая строка должна содержать одни и те же символы. Если строки хотя бы одной подматрицы содержат разные символы, то соответствующие состояния не являются $(n+1)$–эквивалентными. Разбиение групп n–эквивалентных состояний на подгруппы $(n+1)$–эквивалентных состояний закончится не более чем за $|K|$ шагов, т.к. на каждом шаге отделяется по меньшей мере одно неэквивалентное состояние от какой–либо группы. □

В соответствии с доказательством теоремы можно предложить следующий *алгоритм построения минимальноого автомата*, эквивалентного исходному.

1. Строится матрица переходов конечного автомата.

2. В матрице группируются отдельно заключительные и незаключительные состояния. Между ними проводится граница как по строкам, так и по столбцам.

3. Рассматриваются поочередно все подматрицы полученной матрицы. В каждой строке такой подматрицы должны быть одинаковые символы, что означает переход либо между подгруппами, либо внутри группы по одному и тому же символу. Если строки содержат разные символы, их нужно сгруппировать так, чтобы в каждой подгруппе содержались одни и те же символы и выполнить разбиение по границам между группами.

4. Повторяется п.4 до тех пор, пока возможно разбиение.

5. Когда разбиение закончено, каждой группе эквивалентности сопоставляется одно состояние.

Пример 8.4. Дана следующая матрица переходов:

	p_1	p_2	p_3	p_4	p_5	p_6	p_7	
p_1		a				a d		нач
p_2	b				b	a	b	
p_3				b		a b		
p_4		a	a			a	d	
p_5			a			d	a	
p_6		c	c					закл
p_7			c					закл

Применим алгоритм минимизации и получим разбиение этой матрицы

	p_1	p_4	p_5	p_2	p_3	p_6	p_7	
p_1				a		a d		нач
p_4				a	a	a	d	
p_5					a	d	a	
p_2	b		b			a	b	
p_3		b				a b		
p_6				c	c			закл
p_7					c			закл

В каждой подматрице получили одинаковые строки, поэтому каждой группе эквивалентных состояний поставим в соответствие одно состояние нового автомата:

	p_1	p_2	p_3	
p_1		a	a d	нач
p_2	b		a b	
p_3		c		закл

8.6 Операции над регулярными языками

Так как произвольному конечному автомату однозначно соответствует детерминированный конечный автомат, операции над конечными автоматами эквивалентны операциям над регулярными множествами.

Известно, что для произвольного конечного автомата можно построить эквивалентный автомат без циклов в начальном и (или) конечных состояниях. Для построения алгоритмов таких преобразований рассмотрим следующие теоремы.

Теорема 8.3. Для произвольного конечного автомата существует эквивалентный автомат без циклов в начальном состоянии.

Доказательство. Пусть $A = (K, \Sigma, \delta, p_0, F)$ — произвольный конечный автомат. Построим новый конечный автомат

$$A_1 = (K \cup \{q_0 | q_0 \notin K\}, \Sigma, \delta \cup \{q_0 a \to p_i | p_0 a \to p_i \in \delta\}, q_0, F \cup \{q_0 | p_o \in F\}).$$

Рассмотрим сначала цепочку ε. Цепочка $\varepsilon \in L(A)$ тогда и только тогда, когда $p_0 \in F$, но в этом случае по построению автомата A_1 его начальное состояние q_0 является заключительным и, следовательно, $\varepsilon \in L(A_1)$.

Перейдем теперь а анализу цепочек x общего вида: $x \neq \varepsilon$. По построению автомат A не имеет циклов в начальном состоянии q_0. Любая цепочка $x = a_1, a_2, ..., a_k \neq \varepsilon$ принадлежит $L(A)$ тогда и только тогда, когда существует последовательность команд автомата A

$$p_0 a_1 \to p_1 \ , \ p_1 a_2 \to p_2 \ , \ldots , \ p_{k-1} a_k \to p_k \ , \ p_k \in F$$

и соответствующая ей последовательность команд автомата A_1

$$q_0 a_1 \to p_1 \ , \ p_1 a_2 \to p_2 \ , \ldots , \ p_{k-1} a_k \to p_k \ .$$

Следовательно, $L(A) = L(A_1)$.□

Теорема 8.4. Для произвольного конечного автомата существует эквивалентный автомат без циклов в заключительном состоянии.

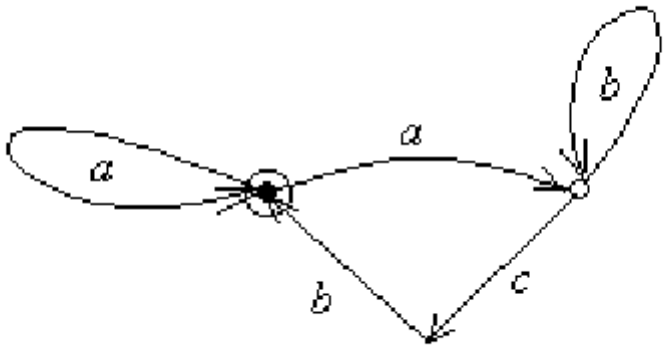


Рис. 8.3: Конечный автомат с циклами в начальном и заключительном состояниях.

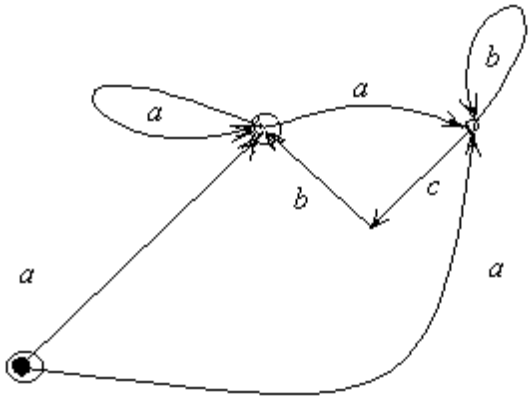


Рис. 8.4: Конечный автомат без циклов в начальном состоянии.

Доказательство. По теореме 8.3 можем считать, что исходный автомат $A = (K, \Sigma, \delta, p_0, F)$ не имеет циклов в начальном состоянии. Сопоставим заданному произвольному конечному автомату A новый автомат

$$A_1 = (K \cup \{f | f \notin K\}, \Sigma, \delta \cup \{p_j a \to f | p_j a \to p_i \in \delta \& p_i \in F\}, p_0, \{f\} \cup \{p_0 | p_0 \in F\}).$$

По построению A не имеет циклов в заключительном состоянии f. Если заключительным состоянием является также и состояние p_0, то циклы в этом состоянии отсутствуют по условию предварительного преобразования исходного автомата по теореме 8.4. Эквивалентность $L(A) = L(A_1)$ очевидна. □

Пример 8.5. Построим автомат без циклов в начальном и заключительном состояниях эквивалентный автомату, представленному на рис 8.3.

Предварительно удалим циклы, проходящие через начальное состояние. Получим автомат, представленный на рис. 8.4.

Затем удалим циклы, проходящие через заключительные состояния. Таких состояний три, однако, начальное состояние, одновременно являющееся и заключительным, от циклов свободно. В результате получим автомат, представленный на рис 8.5.

Рассмотрим теперь операции над регулярными множествами или, что то же самое, над языками, допускаемыми конечными автоматами.

Теорема 8.5. Множество регулярных языков замкнуто относительно операций

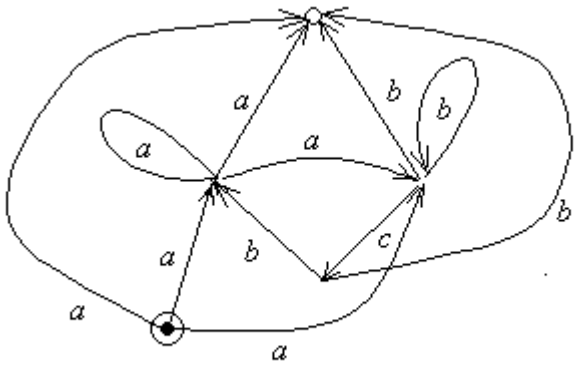


Рис. 8.5: Конечный автомат без циклов в заключительном состоянии.

итерации, усеченной итерации, произведения, объединения, пересечения, дополнения, разности.

Доказательство. Для доказательства необходимо выполнить операции над соответствующими конечными автоматами и показать, что в результате таких преобразований построенный автомат допускает требуемый язык.

Поскольку можно удалить циклы из начальных и заключительных состояний любого автомата, всегда будем предполагать, что такие преобразования сделаны, если это необходимо. Тогда для выполнения указанных в теореме операций необходимо выполнить соответствующие преобразования над заданными автоматами.

Операция итерации реализуется удалением циклов из начальных и конечных состояний и объединением этих полученных состояний. Действительно, объединение начального и заключительного состояний означает, что построенный автомат допускает цепочку ε. Однократный переход из начального в заключительное состояние исходного автомата соответствует допуску цепочек языка L. Поскольку эти состояния объединены, построенный автомат допускает цепочки языков LL, LLL и т.д., т.е. он распознает язык $\{\varepsilon\} \cup L \cup L^2 \cup \cdots = L^*$.

Операция произведения над $L(A_1)$ и $L(A_2)$ выполняется с помощью двух преобразований:

a) удалим циклы из заключительного состояния A_1 или из начального состояния A_2;

b) каждому заключительному состоянию p_i автомата A_1 поставим в соответствие свой экземпляр автомата A_2 и объединяем каждое заключительное состояние автомата A_1 с начальным состоянием соответствующего экземпляра автомата A_2.

Объединение $L(A_1)$ и $L(A_2)$ строится с помощью удаления циклов в начальных состояниях A_1 и A_2 и объединения полученных начальных состояний.

Усеченная итерация может быть построена как произведение вида

$$L(A_1)^+ = L(A_1)^* L(A_1)$$

или вида

$$L(A_1)^+ = L(A_1) L(A_1)^*.$$

Соответствующие операции над автоматами мы уже рассмотрели.

Рассмотрим дополнение $L(A_1)$ до Σ^*. Допустим, автомат A_1 является детерминированным, т.к. по теореме 8.1 любой автомат можно преобразовать к эквивалентному детерминированному. Известно, что автомат можно задать графом переходов, тогда

любая цепочка $x = a_1a_2..a_n$ распознается детерминированным автоматом A_1 по единственному маршруту:

$$\begin{array}{ll} P_0a_1 \to & P_{i_1} \\ P_{i_1}a_2 \to & P_{i_2} \\ \dots & \\ P_{i_{n-1}}a_n \to & P_z, \text{ где } P_z \in F\ . \end{array}$$

Автомат A_1 не распознает те и только те цепочки, которые

a) либо представляют собой начальную часть цепочки $a_1a_2...a_j$, при чтении которой автомат переходит в состояние, не являющееся заключительным;

b) либо имеют вид $y = a_1a_2...a_lbc_1c_2...c_r, l \le n$, где начало $a_1a_2...a_l$ совпадает с началом цепочки $x \in L(A_1)$, но за символом a_l стоит такой символ b, что автомат A_1 его прочитать не может.

Следовательно, для того, чтобы построить автомат, распознающий дополнение языка $L(A)$, надо для детерминированного конечного автомата A выполнить следующие действия:

a) все заключительные состояния сделать незаключительными, а все незаключительные — заключительными;

b) ввести дополнительное состояние q, сделать его заключительным и из каждого состояния p_i провести в новое состояние q такие дуги $\overrightarrow{p_iq}$, каждая из которых соответствует символам алфавита, не читаемым в состоянии p_i;

c) в построенном дополнительном состоянии q построить петли для всех символов алфавита, чтобы обеспечить чтение произвольного окончания цепочки $c_1c_2...c_r$.

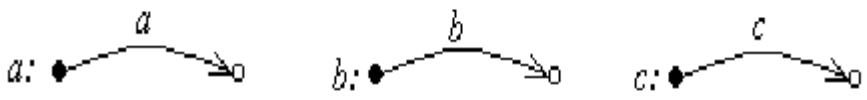
Оставшиеся операции разности и пересечения регулярных языков можно определить через уже рассмотренные операции с помощью следующих тождеств:

$$\begin{array}{ll} L(A_1)\backslash L(A_2) = & \overline{L(A_1) \cap \overline{L(A_2)}} \\ L(A_1) \cap L(A_2) = & \overline{\overline{L(A_1)} \cup \overline{L(A_2)}} \end{array}$$

□

Рассмотренная теорема предлагает алгоритмы построения конечных автоматов, позволяя последовательно синтезировать автоматы на базе уже построенных.

Пример 8.6. Построить конечный автомат, распознающий язык $a^*b \cup b^+c$. Для решения этой задачи последовательно построим автоматы, начиная от простейших и заканчивая автоматом, распознающим заданный язык в целом. Сначала построим автоматы, которые распознают элементарные языки, состоящие из единственной цепочки.



Затем выполним операцию итерации и операцию усеченной итерации соответственно над языками $\{a\}$, $\{b\}$. Получим автоматы, представленные на рис. 8.6.

Построим a^*b как произведение a^* на b. Удалим циклы из начального состояния полученного автомата, получим автомат, представленный на рис. 8.7.

Рис. 8.6: Конечные автоматы, распознающие a^* и b^+.

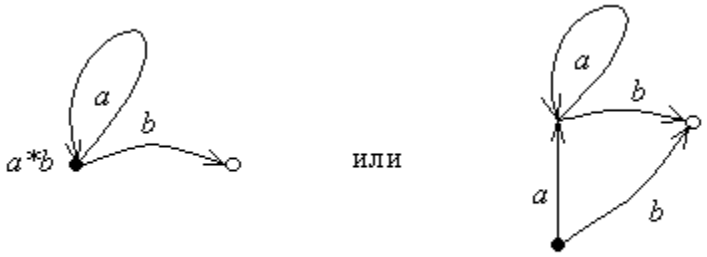


Рис. 8.7: Конечный автомат, распознающий a^*b.

Построим произведение b^+ на c, получим автомат, представленный на рис. 8.8.

Терерь осталось построить объединение языков a^*b и b^+c.. Начальные состояния соответствующих автоматов циклов не содержат, поэтому просто объединим эти состояния. В результате получим автомат, распознающий язык $a^*b \cup b^+c$ (см. рис. 8.9).

8.7 Автоматные грамматики и конечные автоматы

Мы рассмотрели два вида автоматных грамматик: леволинейные и праволинейные с правилами вида $A \to Ba, A \to a$ или $A \to aB, A \to a$ соответственно. Покажем, что языки, порождаемые линейными грамматиками, совпадают с языками, распознаваемыми конечными автоматами.

Теорема 8.6. Для каждой праволинейной грамматики существует эквивалентный конечный автомат.

Доказательство. Каждому нетерминальному символу $A_i \in V_N$ произвольной праволинейной грамматики $G = (V_T, V_N, P, A_0)$ поставим в соответствие состояние A_i конечного автомата A. Добавим еще одно состояние F и сделаем его единственным конечным состоянием. Состояние, соответствующее аксиоме, сделаем начальным.

Каждому правилу $A_i \to aA_j$ поставим в соответствие команду $A_i, a \to A_j$ автомата A, а каждому терминальному правилу $A_i \to a$ — команду $A_i, a \to F$. Тогда каждому выводу в грамматике

$$A_0 \Rightarrow a_1 A_{i_1} \Rightarrow a_1 a_2 A_{i_2} \Rightarrow \ldots \Rightarrow a_1 a_2 ... a_{k-1} A_{i_{k-1}} \Rightarrow a_1 a_2 ... a_{k-1} a_k$$

взаимно–однозначно соответствует последовательность команд построенного авто-

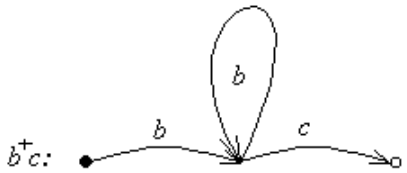


Рис. 8.8: Конечный автомат, распознающий b^+c.

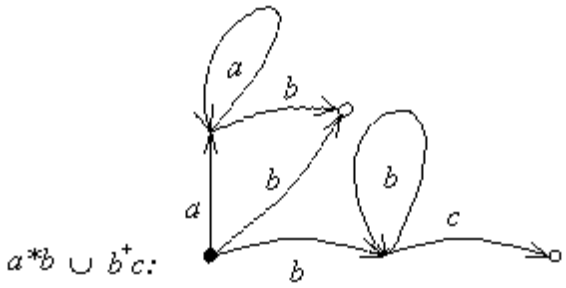


Рис. 8.9: Конечный автомат для примера 8.6.

мата

$$A_0, a_1 \to A_{i_1},$$
$$A_{i_1}, a_2 \to A_{i_2},$$
$$\ldots$$
$$A_{i_{k-2}}, a_{k-1} \to A_{i_{k-1}},$$
$$A_{i_{k-1}}, a_k \to F.$$

Следовательно, $L(G) = L(A)$. □

Пример 8.7. Для заданной грамматики

$$\begin{array}{ll} G: & S \to aS|bB \\ & A \to aA|bS \\ & B \to bB|c|cA \end{array}$$

эквивалентный конечный автомат представлен на рис. 8.10.

Теорема 8.7. Для произвольного конечного автомата существует эквивалентная праволинейная грамматика.

Доказательство. Каждому состоянию p_i произвольного конечного автомата $A = (K, \Sigma, \delta, p_0, F)$ поставим в соответствие нетерминальный символ P_i грамматики, причем начальному состоянию p_0 поставим в соответствие аксиому. Тогда для каждой команды $p_i, c \to p_j$ в множество правил грамматики включим правило $P_i \to cP_j$, причем если P_j — заключительное состояние, то добавим правило $P_i \to c$. Эквивалентность исходного автомата и построенной грамматики очевидна. □

Теорема 8.8. Для каждой леволинейной грамматики существует эквивалентный конечный автомат.

Доказательство. Каждому нетерминальному символу A_i произвольной леволинейной грамматики $G = (V_T, V_N, P, A_0)$ поставим в соответствие состояние p_i конечного автомата A, причем состояние p_0, соответствующее аксиоме A_0, сделаем заключительным. Добавим еще одно состояние N и сделаем его начальным состоянием.

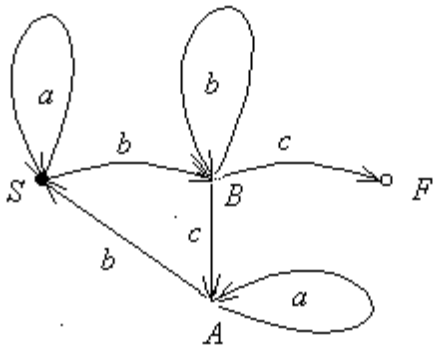


Рис. 8.10: Конечный автомат для грамматики примера 8.7.

Каждому правилу $A_i \to A_j a$ поставим в соответствие команду $p_j, a \to p_i$, а каждому терминальному правилу $A_i \to a$ — команду $N, a \to A_i$. Тогда каждому выводу в грамматике

$$A_0 \Rightarrow A_{i_1} a_k \Rightarrow A_{i_2} a_{k-1} a_k \Rightarrow \ldots \Rightarrow A_{i_{k-1}} a_2 ... a_k \Rightarrow \ldots \Rightarrow a_1 a_2 ... a_k$$

взаимо–однозначно соответствует последовательность команд построенного автомата

$$N, a_1 \to p_{i_{k-1}}, \ldots, p_{i_2}, a_{k-1} \to p_{i_1}, p_{i_1} a_k \to p_0.$$

Следовательно, $L(G) = L(A)$. □

Теорема 8.9. Для произвольного конечного автомата существует эквивалентная леволинейная грамматика.

Доказательство. Каждому состоянию p_i произвольного конечного автомата $A = (K, \Sigma, \delta, p_0, F)$ поставим в соответствие нетерминальный символ A_i грамматики. Добавим еще один нетерминал S и сделаем его аксиомой. Для каждой команды $p_i, a \to p_j$ автомата A в множество правил грамматики включим правило $A_j \to A_i a$, причем если p_j — заключительное состояние, то дополнительно сформируем правило $S \to A_i a$, а если A_i — начальное состояние, то дополнительное правило $A_j \to a$. Тогда последовательности команд

$$\begin{aligned} &p_0, a_1 \to p_{i_1}, \\ &p_{i_1}, a_2 \to p_{i_2}, \\ &... \\ &p_{i_{k-1}}, a_k \to F \end{aligned}$$

взаимнооднозначно соответствует вывод

$$S \Rightarrow A_{i_{k-1}} a_k \Rightarrow ... \Rightarrow A_{i_2} a_3 ... a_k \Rightarrow A_{i_1} a_2 a_3 ... a_k \Rightarrow a_1 a_2 ... a_k.$$

Языки $L(A)$ и $L(G)$ совпадают. □

Ранее (см. теорему 8.1) мы показали регулярность множеств, распознаваемых конечными автоматами. Тогда эквивалентность языков, порождаемых линейными грамматиками, и языков, распознаваемых конечными автоматами, можно сформулировать в терминах регулярных множеств.

Следствие 8.1. Языки, порождаемые линейными грамматиками, регулярны.

Следствие 8.2. Классы леволинейных и праволинейных грамматик эквивалентны.

Пример 8.8. Для заданной леволинейной грамматики построить эквивалентную праволинейную грамматику.

$$\begin{aligned} G_1: \quad & S \to Aa \\ & A \to Bc|d \\ & B \to Ba|b \end{aligned}$$

Преобразование леволинейной грамматики в праволинейную можно выполнить с помощью построения конечного автомата в качестве промежуточного шага такого преобразования. Сначала строим конечный автомат для грамматики G_1 (см. рис. 8.11.).

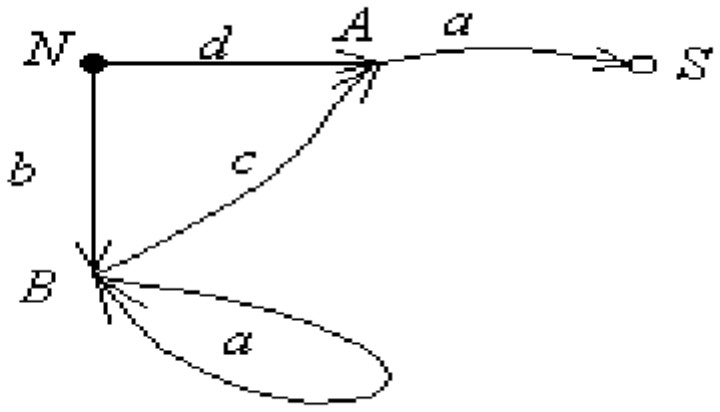


Рис. 8.11: Конечный автомат для леволинейной грамматики примера 8.8.

Этому автомату соответствует праволинейная грамматика:

$$\begin{aligned} G_2: \quad & N \to dA|bB \\ & A \to a \\ & B \to aB|cA \end{aligned}$$

Цепочка, например, $baaca$ выводится как в грамматике G_1:

$$S \Rightarrow Aa \Rightarrow Bca \Rightarrow Baca \Rightarrow Baaca \Rightarrow baaca,$$

так и в грамматике G_2:

$$N \Rightarrow bB \Rightarrow baB \Rightarrow baaB \Rightarrow baacA \Rightarrow baaca.$$

Ранее мы показали, что класс КС–языков не замкнут относительно операций пересечения. Сейчас мы докажем важный факт, позволяющий, когда это необходимо, путем пересечения с подходящим регулярным множеством отбрасывать те цепочки данного КС–языка, которые не имеют интересующей нас формы, и причем так, что остающиеся цепочки также образуют КС–язык. Это свойство подтверждает фундаментальную роль регулярных множеств в теории КС–языков.

Теорема 8.10. Пересечение КС–языка и регулярного множества является КС–языком.

Доказательство. Пусть $L(G)$ — КС–язык, порождаемый заданной грамматикой $G = (V_T, V_N, P, S)$. В силу существования эквивалентого преобразования к неукорачивающей форме можно считать G неукорачивающей грамматикой. Пусть M — регулярное множество, распознаваемое конечным автоматом $A = (K, \Sigma, \delta, p_O, F)$, причем

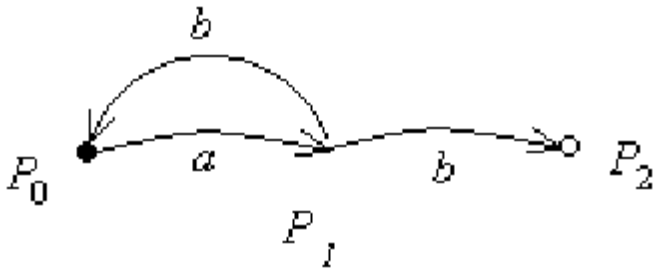


Рис. 8.12: Конечный автомат, распознающий язык $(ab)^+$.

$\varepsilon \notin M$. По теореме 8.4 можно считать, что A имеет единственное заключительное состояние и $F = \{p_z\}$.

Будем считать, $V_T = \Sigma$, т.к. любое из них всегда можно дополнить до $V_T \cup \Sigma$. Построим новую КС–грамматику

$$G^1 = (V_T, V_N^1, P^1, S^1) \ ,$$

где V_N^1 состоит из трехэлементных символов $V_N^1 = K \times V_N \times K$, и $S^1 = (p_O, S, p_z)$.

Построим множество правил грамматики G^1 так, чтобы они одновременно отслеживали правила вывода в G и элементарные такты работы автомата A в процессе распознавания входной цепочки:

1) если $B \to a_1 a_2 ... a_k \in P, B \in V_N, a_i \in V_T \cup V_N$, то в множество P^1 включим всевозможные правила вида

$$(p_i B p_j) \to (p_i a_1 p_{i_1})(p_{i_1} a_2 p_{i_2})...(p_{i_{k-2}} a_{k-1} p_{i_{k-1}})(p_{i_{k-1}} a_k p_j)$$

для всех $p_i, p_j, p_{i_t} \in K$;

2) в множество P^1 включим правило $(p, a, q) \to a$, если $a \in V_T$ и $\delta(p, a) = q$, т.е. имеется команда автомата $pa \to q$.

Для каждого вывода $S \stackrel{*}{\Rightarrow} a_1 a_2 ... a_t$ в грамматике G очевидно существование соответствующего вывода в G^1

$$(p_0 S p_z) \stackrel{*}{\Rightarrow} (p_0 a_1 p_{i_1})(p_{i_1} a_2 p_{i_2})...(p_{i_{t-1}} a_t p_z)$$

и обратно, каждому такому выводу в G^1 соответствует вывод $S \stackrel{*}{\Rightarrow} a_1 a_2 ... a_t$ в G. Далее, по построению G^1 правило $(paq) \to a$ принадлежит P^1 тогда и только тогда, когда $a \in V_T$ и $p, a \to q \in \delta$, следовательно, $p_0, p_{i_1}, p_{i_2}, ..., p_z$ должны быть теми состояниями автомата A, через которые проходит процесс распознавания цепочки $a_1 a_2 ... a_t$. Таким образом, $a_1 a_2 ... a_t \in L(A) \cap L(G)$ тогда и только тогда, когда $(p_0 S p_z) \stackrel{*}{\Rightarrow} a_1 a_2 ... a_t$ в построенной КС–грамматике G^1, т.е. $L \cap M$ — КС–язык. $\square$

Пример 8.9. $M = (ab)^+$, $L = a^n b^n$, $n > 0$. Ясно, что $M \cap L = ab$. Покажем этот факт средствами, используемыми при доказательстве теоремы 8.10. Действительно, язык L порождает КС–грамматика

$$G : S \to aSb|ab$$

Автомат, распознающий M, представлен на рис. 8.12.

Строим новую грамматику по правилам, применяемым при доказательстве теоремы:

$$\begin{array}{ll} G_1: & (p_iSp_j) \to (p_iap_t)(p_tSp_k)(p_kbp_j) \\ & (p_iSp_j) \to (p_iap_t)(p_tbp_j) \\ & (p_0ap_1) \to a \\ & (p_1bp_0) \to b \\ & (p_1bp_2) \to b, \end{array}$$

где (p_0Sp_2) — аксиома. Так как для аксиомы имеется только два правила грамматики, то можно рассмотреть два варианта вывода терминальных цепочек языка, порождаемого грамматикой G_1. Рассмотрим самый короткий вывод из аксиомы с использованием аналога правила $S \to ab$:

$$(p_0Sp_2) \Rightarrow (p_0ap_k)(p_kbp_2) = (p_0ap_1)(p_1bp_2) \Rightarrow a(p_1bp_2) \Rightarrow ab.$$

Вывод из аксиомы с использованием аналога правила $S \to aSb$ имеет вид:

$$(p_0Sp_2) \Rightarrow (p_0ap_1)(p_1Sp_1)(p_1bp_2) \Rightarrow a(p_1ap_t)(p_tSp_k)(p_kbp_1)b \Rightarrow ...$$

Этот вывод никогда не приведет к терминальной цепочке, т.к. для любых t и k для нетерминалов (p_1ap_t) и (p_kbp_1) нет правил. Следовательно, цепочка ab —единственная цепочка, принадлежащая языку, и $L(G_1) = ab$.

8.8 Автоматы с магазинной памятью и КС–языки

В отличие от конечного автомата МП–автомат имеет рабочую ленту — магазин.

Определение 8.12. МП–автомат — это семерка вида

$$M = (K, \Sigma, \Gamma, \delta, p_0, F, B_0), \text{ где}$$

K — конечное множество состояний,
Σ — алфавит,
Γ — алфавит магазина,
δ — функция переходов,
p_0 — начальное состояние,
F — множество заключительных состояний,
B_0 — символ из Γ для обозначения маркера дна магазина.

В общем случае это определение соответствует недетерминированному автомату. В отличие от конечного автомата для произвольного недетерминированного МП–автомата нельзя построить эквивалентный детерминированный МП–автомат. Примем этот факт без доказательства.

Основное использование распознавательных средств задания языков — это построение алгоритмов грамматического разбора, тогда необходимо для произвольной КС–грамматики уметь строить эквивалентный МП–автомат. МП–автоматы представляют интерес как средство разбора в КС–грамматиках произвольного вида. Этот факт сформулирован в следующей теореме.

Теорема 8.11. Языки, порождаемые КС–грамматиками, совпадают с языками, распознаваемыми МП–автоматами.

Доказательство. Существуют две стратегии разбора: восходящая и нисходящая. При восходящей стратегии разбора необходимо найти основу и редуцировать ее в соответствии с правилами грамматики к какому–нибудь нетерминалу. Это можно сделать, если реализовать следующий алгоритм функционирования МП–автомата:

a) любой входной символ записывается в магазин;

b) если в верхушке магазина сформирована основа, совпадающая с правой частью правила, то она заменяется на нетерминал в левой части этого правила;

c) разбор заканчивается, если в магазине аксиома, а входная цепочка просмотрена полностью.

В соответствии с указанным алгоритмом для КС–грамматики $G = (V_t, V_n, P, S)$ построим МП–автомат

$$M = (K, V_t, \Gamma, \delta, p_0, F, B_0) \ ,$$

где $\Gamma = V_t \cup V_n \cup \{B_0\}, K = \{p_0, f\}, F = \{f\}$ и функция переходов δ содержит команды следующего вида:

a) $p_0, a, \varepsilon \to p_0, a$ для любого терминала $a \in V_t$;

b) $p_0, \varepsilon, \widetilde{\varphi} \to p_0, A$ для всех правил $A \to \varphi \in P$ ($\widetilde{\varphi}$ — зеркальное отображение φ) ;

c) $p_0, \varepsilon, SB_0 \to f, B_0$.

Очевидно, что любому выводу в G взаимно однозначно соответствует последовательность команд построенного автомата M.

Обратное построение КС–грамматики по произвольному МП–автомату также возможно, но не представляет практического интереса. □

Пример 8.10. Построим МП–автомат для грамматики

$$\begin{array}{ll} G: & S \to S + A | S - A | A \\ & A \to a | (S) \end{array}$$

Эквивалентный МП–автомат содержит следующие команды:

a) команды переноса терминалов в магазин

$$\begin{array}{ll} P_0, a, \varepsilon & \to P_0, a \\ P_0, +, \varepsilon & \to P_0, + \\ P_0, -, \varepsilon & \to P_0, - \\ P_0, (, \varepsilon & \to P_0, (\\ P_0,), \varepsilon & \to P_0,) \end{array}$$

b) команды редукции по правилам грамматики

$$\begin{array}{ll} P_0, \varepsilon, A + S & \to P_0, S \\ P_0, \varepsilon, A - S & \to P_0, S \\ P_0, \varepsilon,)S(& \to P_0, A \\ P_0, \varepsilon, A & \to P_0, S \\ P_0, \varepsilon, a & \to f, A \end{array}$$

c) команду проверки на завершение

$$P_0, \varepsilon, SB_0 \to f, B_0$$

Подадим на вход МП–автомата цепочку $(a+a)-a$. Действия, которые выполняет этот автомат, представлены на рис. 8.13.

На третьем шаге можно применить одну из двух команд: $P_0, \varepsilon, a \to P_0, A$ или $P_0, +, \varepsilon \to P_0, +$. Применение второй из них не приведет к завершению разбора.

Рассмотренное доказательство теоремы 8.11 основано на восходящей стратегии разбора. Рассмотрим нисходящую стратегию разбора. На каждом шаге нисходящей стратегии разбора должно применяться какое–либо правило. В начальный момент таким нетерминалом является аксиома. Тогда соответствующий МП–автомат должен выполнять следующие действия:

состояние	магазин					вход
P_0	B_0					(
P_0	B_0	(				a
P_0	B_0	(	a			
P_0	B_0	(	A			
P_0	B_0	(	S			$+$
P_0	B_0	(	S	$+$		a
P_0	B_0	(	S	$+$	a	
P_0	B_0	(	S	$+$	A	
P_0	B_0	(	S			)
P_0	B_0	(	S	)		
P_0	B_0	A				
P_0	B_0	S				-
P_0	B_0	S	-			a
P_0	B_0	S	-	a		
P_0	B_0	S	-	A		
P_0	B_0	S				
F	B_0					

Рис. 8.13: Грамматический разбор цепочки $(a + a) - a$ по восходящей стратегии для грамматики примера 8.10.

a) в начальный момент в магазин заносится аксиома:

$$p_0, \varepsilon, \varepsilon \to p_1, S;$$

b) для любого правила $A \to \varphi \in P$ нетерминал заменяется на правую часть правила с помощью команды

$$p_1, \varepsilon, A \to p_1, \widetilde{\varphi};$$

c) для любого терминала $a \in V_t$ выполняется сравнение символа на входе с символом в верхушке магазина по команде

$$p_1, a, a \to p_1, \varepsilon$$

(символ a был записан в магазин на некотором предшествующем шаге разбора, когда применялась правило грамматики);

d) разбор заканчивается по команде

$$p_1, \varepsilon, B_0 \to f, B_0,$$

когда цепочка прочитана полностью и магазин пуст.

Пример 8.11. Для грамматики из примера 8.10 работающий по нисходящей стратегии МП–автомат имеет множество команд:

a) команда записи аксиомы в магазин в начале работы

$$P_0, \varepsilon, \varepsilon \to P_1, S;$$

б) команды применения правил грамматики к нетерминалу в верхушке магазина

$$P_1, \varepsilon, S \to P_1, A + S$$
$$P_1, \varepsilon, S \to P_1, A - S$$
$$P_1, \varepsilon, S \to P_1, A$$
$$P_1, \varepsilon, S \to P_1, A$$
$$P_1, \varepsilon, A \to P_1,)S($$
$$P_1, \varepsilon, A \to P_1, a;$$

в) команды сравнения терминала в верхушке магазина с терминалом на входной ленте

$$P_1, a, a \to P_1, a$$
$$P_1, +, + \to P_1, \varepsilon$$
$$P_1, -, - \to P_1, \varepsilon$$
$$P_1, (, (\to P_1, \varepsilon$$
$$P_1,),) \to P_1, \varepsilon;$$

г) команда завершения работы

$$P_1, \varepsilon, B_0 \to F, B_0.$$

Подадим на вход цепочку $(a + a) - a$, тогда процесс разбора может быть представлен на рис. 8.14. Сразу отметим, что при нисходящей стратегии разбора недетерминированный МП–автомат может выполнить грамматический разбор в леворекурсивной КС–грамматике именно в силу своей недетерминированности. Детерминированная модель недетерминированного алгоритма, построенная на основе перебора всех вариантов применения подходящего правила, не может работать в леворекурсивной КС–грамматике. Дело в том, что если при нисходящем грамматическом разборе анализатор по некоторым причинам решил применить леворекурсивное правило $A \to A\beta$, то правая часть $A\beta$ этого правила заменит в магазине находящийся там нетерминал A, причем в верхушке магазина опять окажется тот же самый нетерминал A, а, значит, ситуация повторяется. В результате анализатор бесконечно будет применять одно и то же леворекурсивное правило.

8.9 Разбор с возвратом

Рассмотренные МП–автоматы работают недетерминированно. Если цепочка принадлежит языку, порождаемому грамматикой, то какой–то из вариантов функционирования автомата выполнит правильный разбор. Если цепочка не принадлежит языку, то никакой вариант не приведет к цели. Отсутствие эквивалентного детерминированного автомата для произвольной КС–грамматики означает невозможность построения универсальной простой однопроходной программы синтаксического анализа. Поэтому для эффективного разбора необходимо выделять специальные классы КС–грамматик, удовлетворяющие требованиям конкретных типов анализаторов. Если требуется выполнить разбор для произвольной КС–грамматики, то придется использовать детерминированную программную модель недетерминированного МП–автомата.

Для того, чтобы запрограммировать недетерминированный МП–автомат, необходимо обеспечить перебор всех возможных вариантов на каждом шаге разбора. В общем виде для любой стратегии перебора алгоритм реализуется рекурсивной процедурой и должен обеспечивать следующие действия.

состояние	магазин							вход
P_0	B_0							
P_1	B_0	S						
P_1	B_0	A	-	S				
P_1	B_0	A	-	A				
P_1	B_0	A	-	)	S	(		(
P_1	B_0	A	-	)	S			
P_1	B_0	A	-	)	S	+	A	
P_1	B_0	A	-	)	S	+	a	a
P_1	B_0	A	-	)	S	+		+
P_1	B_0	A	-	)	S			
P_1	B_0	A	-	)	A			
P_1	B_0	A	-	)	a			a
P_1	B_0	A	-	)				)
P_1	B_0	A	-					-
P_1	B_0	A						
P_1	B_0	a						a
P_1	B_0							
F	B_0							

Рис. 8.14: Грамматический разбор цепочки $(a + a) - a$ по нисходящей стратегии.

1) Проверяется условие завершения разбора. Для нисходящей стратегии условием успешного завершения разбора является пустой магазин и полностью прочитанная цепочка, для восходящей — в магазине находится аксиома и цепочка прочитана полностью. Если разбор закончен, то рекурсивная функция грамматического разбора возвращает значение 1 в качестве признака успешного разбора и завершает работу.

2) Проверяется условие невозможности дальнейшего разбора. Для нисходящей стратегии разбора таким условием является несовпадение терминального символа в верхушке магазина с терминалом на входе. Момент возврата для разбора по восходящей стратегии определяется по условию невозможности продвижения вперед по исходной цепочке: цепочка прочитана полностью, а дерево построено лишь частично или не построено вообще, т.е. в магазине находится не аксиома грамматики.

3) Если оба предшествующих условия не выполнены, это означает, что разбор можно продолжить, применяя правила грамматики. Поэтому организуется цикл по всем правилам грамматики, предусматривающий выполнение следующей последовательности действий.

a) Применяется очередное правило грамматики.

b) Выполняется рекурсивный вызов функции грамматического разбора.

c) Проверяется признак возврата из рекурсивной функции. Если функция вернула 1, то действия были выполнены правильно и разбор успешно завершен. Если функция вернула 0, то текущее правило было неверным. Тогда восстанавливается состояние до применения правила — это верхушка магазина и указатель входной цепочки. После восстановления переходим к следующему правилу грамматики, если оно существует. При завершении перебора, когда ни одно правило не привело к успешному завершению разбора, функция возвращает 0.

Приведем в качестве примера универсальную программу нисходящего разбора с

возвратом. Разбор выполняет рекурсивная функция $Rec(int\ k)$, параметр которой определяет положение указателя исходной цепочки. Пусть правила грамматики задаются в файле по строкам. Каждая строка определяет одно правило грамматики, причем нетерминалами являются большие латинские буквы, а терминалами — маленькие латинские буквы. Нетерминал в левой части правила отделяется от правой части последовательностью символов $->$, окруженной пробелами.

Сразу отметим, что рекурсивный алгоритм нисходящего разбора не может работать на леворекурсивных правилах, т.к. выбор первого леворекурсивного правила грамматики блокирует все остальные. Единственный результат, который будет достигнут в этом случае — сообщение *stack overflow!* . Это ограничение распространяется на любые нисходящие методы разбора, а не только на рекурсивные.

Левая рекурсия может быть не только явной, но и скрытой, например, при наличии правил $S \to Ac$, $A \to BaB$, $B \to SaA$ получаем леворекурсивную зависимость $S \Rightarrow Ac \Rightarrow BaBc \Rightarrow SaAaBc$. Скрытая рекурсия при нисходящем разборе также приведет к сообщению о переполнении стека.

```
// грамматический разбор на нисходящий стратегии с возвратами
#include <stdio.h>
#include <string.h>
#include <STDLIB.H>

#define  MAX_STACK 2000
#define  MAXK  100    // максимальное число правил грамматики
#define  MAX_LEX 500  // максимальная длина исходной цепочки

typedef char LEX[MAX_LEX];
int len[MAXK];
char a[MAXK]; //левые части
LEX fi[MAXK]; // правые части правил грамматики
int  m;       // фактическое число правил грамматики
LEX x;        // исходная цепочка
FILE *gr = fopen("gramm.txt","r"); //файл, содержащий правила грамматики
FILE *in = fopen("input.txt","r"); // файл с исходной цепочкой
char stack[MAX_STACK];    //стек
int uk;                   // указатель первого свободного элемента стека

void GetData(void)
{
LEX ss; // вспомогательная строка для ввода "->"
int i;
fscanf(gr,"%d\n",&m);// число правил
for (i=0; i<m; i++)
     {
      fscanf(gr,"%c %s %s\n",&a[i],&ss,&fi[i]);
       // правило задается в форме a -> fi
     if (fi[i][0]=='e') {len[i]=0; fi[i][0]='\0';}
         else len[i]=strlen(fi[i]);
     }
fscanf(in,"%s",&x);
```

```
}

int Rec(int k)
// грамматический разбор от текущей позиции
{
int i,j,old_uk;
if ( (x[k]=='\0') && (uk==0)) return 1;
if ( (uk>0) && (stack[uk-1]<='z') && (stack[uk-1]>='a'))
    { // в верхушке магазина терминал
    if (x[k]==stack[uk-1]) { uk--; return Rec(k+1);}
       else return 0;
    }
// в верхушке магазина нетерминал
for (i=0; i<m; i++)
    if (stack[uk-1]==a[i])
       { // применяем правило грамматики
       old_uk=uk; uk--;
       printf("Правило %c -> %s   указатель цепочки k=%d\n",
               a[i], fi[i],k);
       for (j=0; j<len[i]; j++)
            stack[uk-j+len[i]-1]=fi[i][j]; // зеркальное отображение
       uk+=len[i];
       if (Rec(k)==1)
         { printf ("Правильно!\n");
         return 1;
         }
       // возврат:
       uk=old_uk; stack[uk-1] = a[i]; printf("Возврат!\n");
       }
return 0;
}


int main(void)
{
GetData();
uk=1; stack[0]=a[0]; // первый нетерминал - аксиома
if (Rec(0)==1) printf("Разбор закончен.\n");
    else printf("Ошибки во входной цепочке.\n");
fclose(in); fclose(gr);
return 0;
}
```

Пример файла gram.txt:
2
$S->aSb$
$S->c$
Пример файла input.txt для правильной цепочки:
aaacbbb
Заметим, что практического значения при построении трансляторов рассмотрен-

ный алгоритм не имеет, т.к. он основан на переборе всех применимых на текущем шаге действий. Используемые на практике КС–грамматики языков программирования удовлетворяют дополнительным ограничениям, которые позволяют строить более эффективные алгоритмы синтаксического анализа. Как правило, КС–грамматики языков программирования обладают такими свойствами, что существуют алгоритмы грамматического разбора с временной сложностью порядка $O(|x|)$, где $|x|$ — длина анализируемой цепочки.

Тем не менее алгоритм разбора с возвратом представляет не только теоретический, но и практический интерес, т.к. *общая схема нисходящего синтаксического анализа произвольной КС–грамматики, основанная на замене нетерминала в верхушке магазина правой частью правила, остается неизменной и для алгоритмов эффективного разбора.*

В качестве упражнения оставляется задача реализации универсального рекурсивного восходящего анализатора — программы грамматического разбора с возвратом по восходящей стратегии разбора. Замечание относительно эффективности грамматического разбора по нисходящей стратегии с возвратом полностью справедливо и для восходящей стратегии разбора.

8.10 Контрольные вопросы к разделу

1. Что представляет собой память автомата?

2. Приведите иерархию автоматов по сложности .

3. Чем отличаются автоматы — преобразователи от автоматов — распознавателей?

4. Перечислением каких объектов задается конечный автомат?

5. Какое множество называется регулярным?

6. Какие операции не выводят из класса регулярных множеств?

7. Как построить автомат, распознающий объединение регулярных множеств?

8. Почему при доказательстве теоремы об объединении двух регулярных языков необходимым условием является отсутствие циклов в начальных состояниях конечных автоматов, распознающих эти языки?

9. Почему при доказательстве теоремы о дополнении регулярного языка необходимым условием является детерминированность конечного автомата?

10. Как построить автомат, распознающий дополнение регулярного множества?

11. Какие два состояния конечного автомата называются k–эквивалентными?

12. Почему при доказательстве теоремы о минимизации конечного автомата на первом шаге все заключительные состояния отделяются от незаключительных?

13. Дайте определение автомата с магазинной памятью.

14. Как для заданной грамматики построить МП–автомат, выполняющий восходящий разбор?

15. Как для заданной грамматики построить МП–автомат, выполняющий нисходящий разбор?

16. Почему МП–автомат, построенный для произвольной КС–грамматики, является в общем случае недетерминированным?

17. Как написать программу, выполняющую восходящий грамматический разбор в соответствии с командами недетерминированного МП–автомата?

18. Как написать программу, выполняющую нисходящий грамматический разбор в соответствии с командами недетерминированного МП–автомата?

19. Чему равна верхняя граница числа шагов при минимизации произвольного конечного автомата?

20. Чем МП–автомат отличается от конечного автомата?

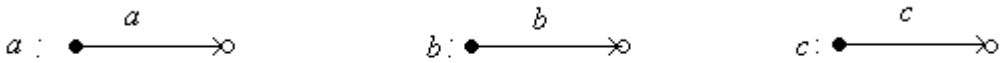
8.11 Упражнения к разделу

Задание. Построить детерминированный конечный автомат, распознающий заданный язык L. Для полученного автомата построить эквивалентную леволинейную и эквивалентную праволинейную грамматику. Привести пример цепочки $x \in L$, показать процесс распознавания автоматом этой цепочки, а также построить вывод этой цепочки в обеих грамматиках.

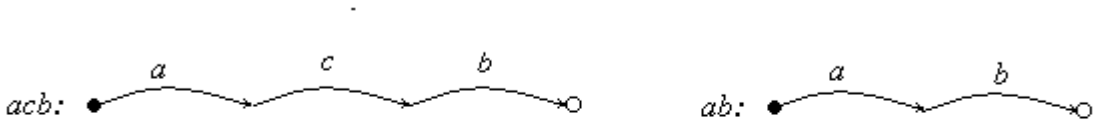
8.11.1 Задача

$$L = (ab)^*c^* \cup (acb)^* \cup a^+.$$

Решение. Очевидно, что элементарные языки, содержащие по одной цепочке a, b, c соответственно, распознаются автоматами, представленными на рисунке:



С помощью операции произведения получим языки ab, acb, а для построения соответствующих автоматов просто соединим начальное состояние автомата b с заключительным состоянием автомата a — для языка ab, и последовательно заключительное состояние автомата a с начальным состоянием автомата c и заключительное состояние этого автомата с начальным состоянием автомата b — для языка acb. Построенные автоматы имеют вид:



Итерация языка распознается автоматом, у которого объединены начальное и заключительное состояние исходного автомата. Соответствующие автоматы представлены на рис. 8.15.

Рассмотрим теперь построение автомата, распознающего язык $(ab)^*c^*$ — произведение языков $(ab)^*$ и c^*. В процессе доказательства теоремы об операциях над регулярными языками мы выяснили, что при построении автоматов, распознающих произведение двух заданных языков необходимо избавиться от циклов в одном из объединяемых состояний — заключительном состоянии первого автомата или в начальном состоянии второго автомата. Удалим циклы из начального состояния автомата, распознающего c^*, а затем объединим указанные состояния. В результате получим автоматы, представленные на рис. 8.16.

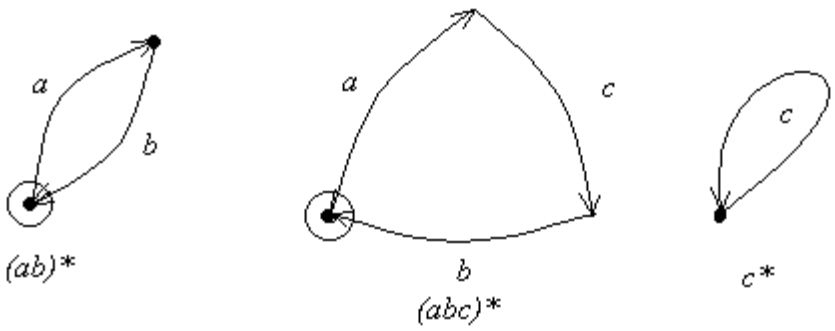


Рис. 8.15: Конечные автоматы, распознающие итерации языков ab, acb, c.

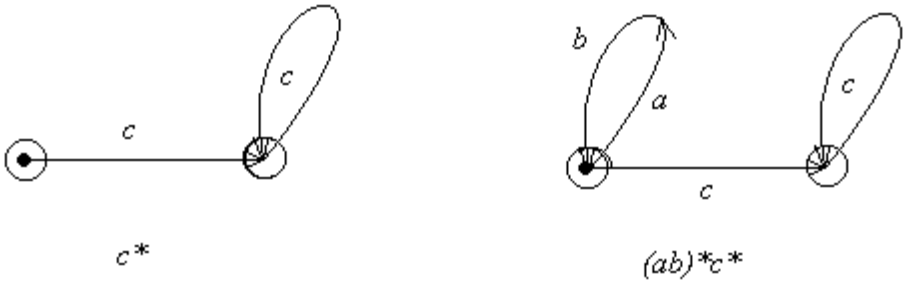


Рис. 8.16: Конечные автоматы, распознающие языки c^*, $(ab)^*c^*$.

Теперь для построения автомата, распознающего заданный язык

$$(ab)^*c^* \cup (acb)^* \cup a^+.$$

достаточно воспользоваться той же теоремой и объединить начальные состояния автоматов $(ab)^*c^*$, $(acb)^*$ и a^+, предварительно устранив циклы из начальных состояний, если они там были. Соответствующие автоматы и результирующий автомат представлены на рис. 8.17.

Построенный автомат не является детерминированным, поэтому необходимо выполнить алгоритм детерминизации полученного автомата. Сначала построим таблицу переходов полученного автомата:

	a	b	c	
p_0	$p_1p_4p_5$		p_3	закл
p_1		p_2		
p_2	p_1		p_3	закл
p_3			p_3	закл
p_4	p_4			закл
p_5			p_7	
p_6	p_5			закл
p_7		p_6		

Начиная из состояния p_0 будем строить объединенные состояния нового автомата:

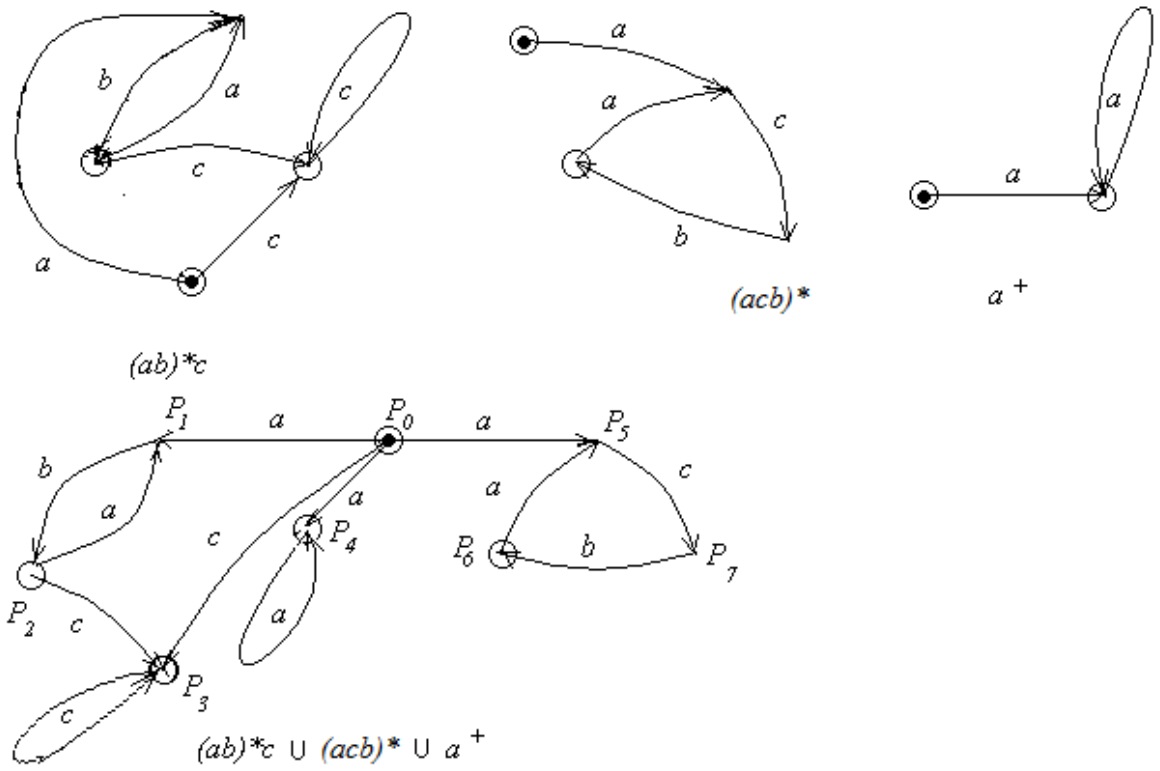


Рис. 8.17: Конечные автоматы без циклов в начальных состояниях, распознающие языки $(ab)^*c^*$, $(acb)^*$ и a^+; автомат, распознающий язык $(ab)^*c^* \cup (acb)^* \cup a^+$.

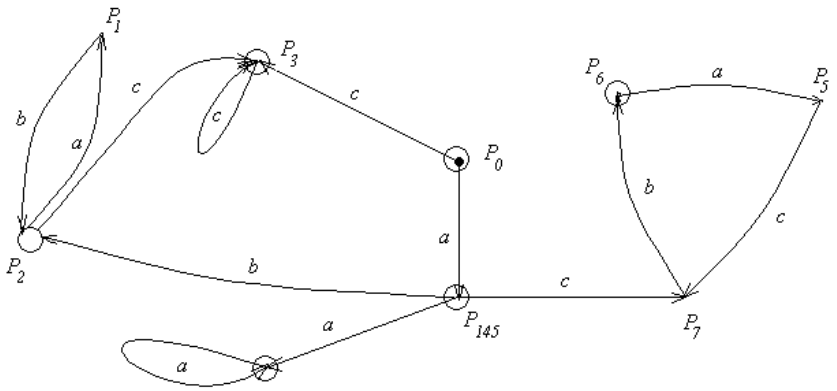


Рис. 8.18: Детерминированный конечный автомат, распознающий язык $(ab)^c \cup (acb)^* \cup a^+$.

	a	b	c	
p_0	p_{145}		p_3	закл
p_{145}	p_4	p_2	p_7	закл
p_4	p_4			закл
p_2	p_1		p_3	закл
p_7		p_6		
p_1		p_2		
p_3			p_3	закл
p_6	p_5			закл
p_5			p_7	

Следует отметить, что у исходного автомата недетерминированным являлось только одно состояние p_0, а второй символ цепочки однозначно определяет тип цепочки, поэтому только одно новое состояние p_{145} появляется у нового детерминированного автомата. Построенный автомат представлен на рис. 8.17.

Построим теперь праволинейную и леволинейную грамматики, эквивалентные полученному детерминированному автомату. Для этого сначала каждому состоянию автомата поставим в соответствие нетерминальный символ:

состояние	p_0	p_{145}	p_4	p_2	p_7	p_1	p_3	p_6	p_5
нетерминал	N	B	C	D	E	F	K	T	M

Для праволинейной грамматики аксиомой является символ N, соответствующий начальному состоянию автомата. Для каждой команды автомата запишем правило грамматики. Например, команде $p_2, a \to p_1$ ставится в соответствие правило $D \to aF$. Добавим терминальные правила, определяющие завершение процесса порождения цепочки в грамматике. Например, правило $F \to b$ появляется из-за команды перехода $p_1, b \to p_2$ в заключительное состояние p_2. Остальные правила строятся аналогично. Получим праволинейную грамматику:

$$
\begin{array}{ll}
G_1: & N \to aB|a|cK|c \\
& B \to aC|bD|cE|a|b \\
& C \to aC|a \\
& D \to aF|cK|c \\
& E \to bT|b \\
& F \to bD|b \\
& K \to cK|c \\
& T \to aM \\
& M \to cE
\end{array}
$$

Перейдем теперь к построению леволинейной грамматики. В качестве аксиомы возьмем дополнительный нетерминальный символ S, для которого в множество правил грамматики необходимо включить правила, соответствующие командам автомата перехода в заключительное состояние. Например, для команды $p_{154}, b \to p_2$ перехода в заключительное состояние p_2 записываем правила грамматики $S \to Bb$. Терминальные правила грамматики строятся для команд перехода из начального исходного состояния автомата. Например, правило $B \to a$ записывается для команды $p_0, a \to p_{145}$. В результате таких построений получим грамматику:

$$
\begin{array}{ll}
G_2: & S \to Bb|Ca|Ba|Na|Eb|Db|Kc \\
& N \to \\
& B \to Na|a \\
& C \to Ba|Ca \\
& D \to Bb|Fb \\
& E \to Bc|Mc \\
& F \to Da \\
& K \to Kc|Nc|Dc|c \\
& T \to Eb \\
& M \to Ta
\end{array}
$$

Построенная грамматика содержит непродуктивный нетерминал N. Приведем грамматику:

$$
\begin{array}{ll}
G_3: & S \to Bb|Ca|Ba|Eb|Db|Kc \\
& B \to a \\
& C \to Ba|Ca \\
& D \to Bb|Fb \\
& E \to Bc|Mc \\
& F \to Da \\
& K \to Kc|Nc|Dc|c \\
& T \to Eb \\
& M \to Ta
\end{array}
$$

Следует заметить, что подстановка терминального правила $B \to a$ в правые части правил грамматики G_3 уменьшит число нетерминалов, но в результате нарушится структура правил и грамматика перестанет быть леволинейной.

Наконец, рассмотрим примеры вывода в грамматиках и процессы распознавания цепочек исходным автоматом. Заданный язык

$$(ab)^*c^* \cup (acb)^* \cup a^+.$$

содержит цепочки трех типов. Для контроля правильности наших построений рассмотрим примеры цепочек каждого из указанных типов.

а) Цепочка aaa распознается автоматом с помощью команд:

$$p_0, a \to p_{154},\ p_{154}, a \to p_4,\ p_4, a \to p_4.$$

Эта же цепочка порождается в грамматике G_1:

$$N \Rightarrow aB \Rightarrow aaC \Rightarrow aaa.$$

Вывод aaa в грамматике G_3:

$$S \Rightarrow Ca \Rightarrow Baa \Rightarrow aaa.$$

б) Цепочка $acbacb$ распознается автоматом:

$$p_0, a \to p_{154},\ p_{154}, c \to p_7,\ p_7, b \to p_6,\ p_6, a \to p_5,\ p_5, c \to p_7,\ p_7, b \to p_6.$$

Эта же цепочка порождается в грамматике G_1:

$$N \Rightarrow aB \Rightarrow acE \Rightarrow acbT \Rightarrow acbaM \Rightarrow acbacE \Rightarrow acbacb.$$

Вывод $acbacb$ в грамматике G_3:

$$S \Rightarrow Eb \Rightarrow Mcb \Rightarrow Tacb \Rightarrow Ebacb \Rightarrow Bcacb \Rightarrow acbacb.$$

в) Цепочка $ababcc$ распознается автоматом:
$p_0, a \to p_{154}$, $p_{154}, b \to p_2$, $p_2, a \to p_1$, $p_1, b \to p_2$, $p_2, c \to p_3$, $p_3, c \to p_3$.
Эта же цепочка порождается в грамматике G_1:

$$N \Rightarrow aB \Rightarrow abD \Rightarrow abaF \Rightarrow ababD \Rightarrow ababcK \Rightarrow ababcc.$$

Вывод $ababcc$ в грамматике G_3:

$$S \Rightarrow Kc \Rightarrow Dcc \Rightarrow Fbcc \Rightarrow Dabcc \Rightarrow Bbabcc \Rightarrow ababcc.$$

8.11.2 Варианты заданий

1. $(cab)^+ \cup (b)^* \cup bc^*$.
2. $(aca)^* \cup (ca)^* cb^* \cup ac^*$.
3. Дополнение $bac^* \cup (bc)^+$.
4. $ac^* \cup (bca)^*(ba)^*$.
5. Дополнение $bac^* \cup (ac)^+$.
6. $(baa)^*(bc)^* \cup ca^*ac^*$.
7. $(ba)^*c^* \cap (a^*c^*b^*)^+$.
8. $(ca)^+ \cup (b)^* \cup bc^* \cup cc \cup ac$.
9. Дополнение $b^*a \cup bab^+$.
10. $ac^*a \cup (bca)^* \cup (ba)^*$.
11. $cc(ba)^+ \cup (ca^*ac)^* \cup a^*$.
12. $(ab)^*(a^*ba)^* \cup (bac)^*$.
13. $ac^*(ba)^* \cup (ca)^* cb^*$.
14. $(ab)^*(cc)^* \cap (b^*c^*a^*)^*$.
15. Дополнение $ac^* \cup (bca)^*$.
16. $(ba)^+ \cup ca^*ac^* \cup a^*$.
17. $(bc)^* \cup bc^*a \cup cc^+ \cup ac^*$.
18. $ac^*(bca)^* \cup (ba)^* ca^*$.
19. Дополнение $c^*a \cup (cac)^+$.
20. $(baa)^* \cup (bc)^* \cup ca^* \cup ac^*$.

8.12 Тесты для самоконтроля к разделу

1. Автоматом какого типа распознается язык $a^{n+1}b * c^n \cup (ab)^*$? Если существуют несколько типов автоматов, распознающих данный язык, укажите наиболее простой из них.

Варианты ответов:

а) недетерминированным конечным автоматом;

б) детерминированным конечным автоматом;

в) недетерминированным МП–автоматом;

г) детерминированным МП–автоматом;

д) недетермирированной машиной Тьюринга;

е) детерминированной машиной Тьюринга.

Правильный ответ: в.

2. Укажите ложные утверждения из следующего перечня утверждений.

1) Для любого недетерминированного конечного автомата можно построить эквивалентный детерминированный конечный автомат.

2) Для любого недетерминированного МП–автомата можно построить эквивалентный детерминированный МП–автомат.

3) Для любого недетерминированного конечного автомата можно построить эквивалентный детерминированный МП-автомат.

Варианты ответов:

а) ложно 1;

б) ложно 2;

в) ложно 3;

г) ложно 1 и 2;

д) ложно 1 и 3;

е) ложно 2 и 3;

ж) ложно 1, 2 и 3.

Правильный ответ: б.

3. Сколько состояний содержит минимальный конечный автомат, распознающий язык $a^*b^*a^* \cup a^*b^+c^*$?

Варианты ответов:

а) 1;

б) 2;

в) 3;

г) 4;

д) 5;

е) 6.

Правильный ответ: г.

4. Какие из следующих утверждений истинны?

1) Пересечение произвольных регулярных языков является регулярным.

2) Пересечение произвольных КС–языков является КС–языком.

3) Пересечение произвольного регулярного языка и произвольного КС–языка является КС–языклм.

Варианты ответов:

а) все утверждения ложны;

б) истинно только 1;

в) истинно только 2;
г) истинно только 3;
д) истинны 1 и 2;
е) истинны 1 и 3;
ж) истинны 2 и 3;
з) все утверждения истинны.

Правильный ответ: е.

5. Необходимо построить МП–автомат, выполняющий восходящий грамматический разбор в грамматике

$$G : S \longrightarrow aSbb|a$$

Какое множество команд должен содержать автомат, выполняющий указанные действия?

Варианты ответов:

а) $p_0, \varepsilon, S \longrightarrow p_0, a$
$p_0, \varepsilon, S \longrightarrow p_0, bbSa$
$p_0, a, a \longrightarrow p_0, \varepsilon$
$p_0, b, b \longrightarrow p_0, \varepsilon$
$p_0, \varepsilon, B_0 \longrightarrow p_1, B_0$

б) $p_0, a \longrightarrow p_0$
$p_0, b \longrightarrow p_1$
$p_1, b \longrightarrow p_1$
$p_1, \varepsilon \longrightarrow p_2$

в) $p_0, a, a \longrightarrow p_0, \varepsilon$
$p_0, b, b \longrightarrow p_0, \varepsilon$
$p_0, \varepsilon, bbSa \longrightarrow p_0, S$
$p_0, \varepsilon, a \longrightarrow p_0, S$
$p_0, \varepsilon, S \longrightarrow p_1, \varepsilon$

г) $p_0, a, \varepsilon \longrightarrow p_0, a$
$p_0, b, \varepsilon \longrightarrow p_0, b$
$p_0, \varepsilon, S \longrightarrow p_0, bbSa$
$p_0, \varepsilon, S \longrightarrow p_0, a$
$p_0, \varepsilon, S \longrightarrow p_1, \varepsilon$

д) $p_0, a, \varepsilon \longrightarrow p_0, a$
$p_0, b, \varepsilon \longrightarrow p_0, b$
$p_0, \varepsilon, bbSa \longrightarrow p_0, S$
$p_0, \varepsilon, a \longrightarrow p_0, S$
$p_0, \varepsilon, S \longrightarrow p_1, \varepsilon$

Правильный ответ: д.

Глава 9

АВТОМАТЫ — ПРЕОБРАЗОВАТЕЛИ

9.1 Поведение автоматов с выходом

Как уже отмечалось в разделе 7, автоматы–преобразователи имеют выходную ленту (или некоторое другое средство для фиксации выходных символов). В зависимости от вида функции, отображающей множество состояний и входных символов в множество выходных символов и новых состояний, а также от типа рабочей ленты определяют разные типы преобразователей. Мы рассмотрим только конечные автоматы–преобразователи, т.е. автоматы без рабочей памяти. Кроме того, в зависимости от того, выделяется или нет начальное состояние и множество заключительных состояний, говорят об инициальных и неинициальных автоматах соответственно. Рассмотрим пока инициальные автоматы.

Определение 9.1. Конечным преобразователем называется шестерка

$$P = (K, X, Y, f, g, q_0),$$

где K — конечное множество состояний, X, Y — входной и выходной алфавит, f, g — соответственно функции переходов и выходов, q_0 — начальное состояние. Типы отображений f и g определяют различные типы автоматов.

Следует заметить, что в отличие от автоматов–распознавателей, которые мы рассмотрели в предшествующей главе, в определении 9.1 не фиксируется множество заключительных состояний. Для простоты будем считать, если специально не оговорено противное, что любое состояние является заключительным. Это означает, что процесс преобразования определяется для любой входной цепочки, реакция на которую существует. Именно по той причине, что преобразователь предназначен не для контроля входной цепочки, а для перевода ее в некоторую другую цепочку, не вводится обязательное требование завершения работы автомата только в некоторых специально выделенных заключительных состояниях.

Если g — отображение $K \times X$ во множество Y, то конечный преобразователь называется синхронным. В общем случае это отображение $K \times X$ во множество Y^* и соответствующий преобразователь называется асинхронным.

Определение 9.2. Пусть $P = (K, X, Y, f, g, q_0)$ — конечный преобразователь. Отображение, $S(x) = g(q_o, x)$, определенное для любой цепочки $x \in X^*$, называется конечным преобразованием.

Теорема 9.1. Каждое конечное преобразование сохраняет свойства множества быть регулярным и быть КС–языком.

Доказательство. Пусть $A = (K, X, Y, f, g, q_0)$ — произвольный конечный преобразователь. Рассмотрим сначала преобразование заданного КС–языка L преобразователем A. Так же, как при доказательстве теоремы 7.10, построим новую КС–грамматику $G_1 = (V_T, V_N^1, P_1, S_1)$ по грамматике

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

порождающей язык L. В качестве нетерминалов новой КС–грамматики возьмем составные элементы

$$V_N^1 = \{(pBq) \mid p, q \in K; B \in V_N \cup V_{NT}\},$$

где множество дополнительных символов V_{NT} — это множество символов–двойников терминалов из V_T:

$$V_{NT} = \{\widetilde{A} \mid a \in V_T, \widetilde{A} \notin V_N \cup V_T\}.$$

Алгоритм построения новой грамматики G_1 аналогичен алгоритму построения из доказательства теоремы 7.10 за тем исключением, что в правых частях получаемых правил соответствующие терминалам символы заменяем новыми символами–двойниками из V_{NT}. Эти символы–двойники заменятся терминальными элементами в соответствии с функцией выходов конечного преобразователя A. Таким образом, правила грамматики G_1 строятся по следующему алгоритму:

a) если $B \to a_1 a_2 ... a_m \in P$, то в P_1 включаются всевозможные правила вида

$$(p_i B p_j) \to (p_i \widetilde{a}_1 p_{k_1})(p_{k_1} \widetilde{a}_2 p_{k_2})...(p_{k_{m-1}} \widetilde{a}_m p_j)$$

для всех $p_i, p_j, p_{k_l} \in K$ (здесь $\widetilde{a}_i = \widetilde{A}_i$ при $a_i \in V_T$ и $\widetilde{a}_i = A_i$ при $a_i \in V_N,\ a_i = A_i$);

b) правило $(p\widetilde{A}q) \to \varphi$, включается в P_1 для всех $p, q \in K$ и всех $\widetilde{A} \in V_{NT}$, если при этом $f(p, a) = q, g(p, a) = \varphi$.

Очевидно, (см. доказательство теоремы 7.10), что

$$f(p_0, L) = L(G_1),$$

т.е. КС–язык отображается в КС–язык.

Пусть теперь L — регулярное множество. Можно считать грамматику, порождающую L, праволинейной. Тогда все правила вывода грамматики G_1 имеют одну из следующих форм

1)$A \to \widetilde{B}D$ и $A \to \widetilde{B}$ в соответствии с правилами грамматики языка L и правилом (a) построения грамматики G_1;

2) $\widetilde{B} \to \varphi(\varphi \in V_T)$ в соответствии с правилом (b) построения грамматики G_1.

Удаляя правила вида нетерминал–нетерминал, получим эквивалентную грамматику с правилами вида $A \to \widetilde{B}D|\varphi$ и $\widetilde{B} \to \varphi$, где $\varphi = a_1 a_2 ... a_m, a_i \in V_T^*$. Грамматику такого вида можно преобразовать к эквивалентной праволинейной форме:

$$\{A \to BD, B \to a_1 a_2 ... a_m\} = \{A \to a_1 T_1, T_1 \to a_2 T_2, ..., T_{k-1} \to a_k D\}\ ,$$

$$\{A \to a_1 a_2 ... a_m\} = \{A \to a_1 R_1, R_1 \to a_2 R_2, ..., R_{m-2} \to a_{m-1} R_{m-1}, R_{m-1} \to a_m\}.$$

Преобразованная грамматика является праволинейной и, следовательно, множество $g(p_0, L)$ регулярно при условии регулярности L. $\square$

9.2 Автоматы Мили

В определении 9.2 зафиксировано начальное состояние, в котором находится преобразователь в начальный момент. Это состояние существенно влияет на процесс конечного преобразования, т.к. определяет не только результирующую цепочку, но и множество допустимых входных цепочек. Рассмотрим теперь поведение неинициальных автоматов — автоматов, которые могут начинать работу из любого указанного состояния. Естественно считать, что такой автомат получает на вход одну цепочку бесконечной длины и перерабатывает ее. Реакция такого преобразователя на определенные воздействия не предсказуема, если неизвестно его начальное состояние. Поэтому одна из задач анализа, имеющая важное практическое значение, — это задача определения того состояния автомата, в котором он находится в момент, начиная с которого исследуется поведение этого автомата. Другой важной задачей является задача распознавания конечного состояния, т.е. того состояния, в которое перешел автомат после завершения испытательной операции, проведенной исследователем. Очевидно, что это состояние будет начальным для следующей серии испытаний. Соответствующие задачи получили название экспериментов по распознаванию состояний.

Обычно рассматривают два типа неинициальных автоматов с выходом — автоматы Мили и автоматы Мура.

Определение 9.3. Автомат Мили — это пятерка

$$M = (K, X, Y, f, g) \ ,$$

где f - отображение $K \times X \to K$ и g — отображение $K \times X \to Y$, такие, что функции f и g связаны соотношениями

$$\begin{array}{ll} q(0) = & q_0, \\ q(t+1) = & f(q(t), x(t)), \\ y(t) = & g(q(t), x(t)), \end{array}$$

где q_0 — некоторое состояние в начальный момент работы автомата, $q(i) \in K, x(i) \in X, y(i) \in Y$.

Представить автомат Мили можно графом, в котором каждой дуге соответствует пара "вход/выход т.к. в соответствии с определением выход в текущий момент определяется состоянием и входом в этот же момент.

Пример 9.1. Рассмотрим автомат Мили, который читает текст, состоящий из разделенных пробелами русских слов. Автомат должен считать все слова, начинающиеся с "к" и кончающиеся на "р" (такие, как "компьютер "компрессор "курьер" и т.п.). Для простоты и наглядности все буквы, кроме "к" и "р обозначим буквой "v знак пробела — символом "+". Тогда автомат имеет должен выдавать на выход последовательность нулей, заканчивающуюся знаком единицы при распознавании цепочек вида $k\{k, v, p\}^* p+$ и цепочку из одних нулей при распознавании любого другого слова. Такой автомат представлен на рис. 9.1.

Очевидно, что этот автомат будет выполнять указанные действия, если начнет работу из состояния q_0, читая первое слово с его начала.

Таблица переходов автомата Мили состоит из двух таблиц: для состояний и для выходов. Для рассмотренного в примере 9.1 автомата она имеет вид, представленный на рис 9.2.

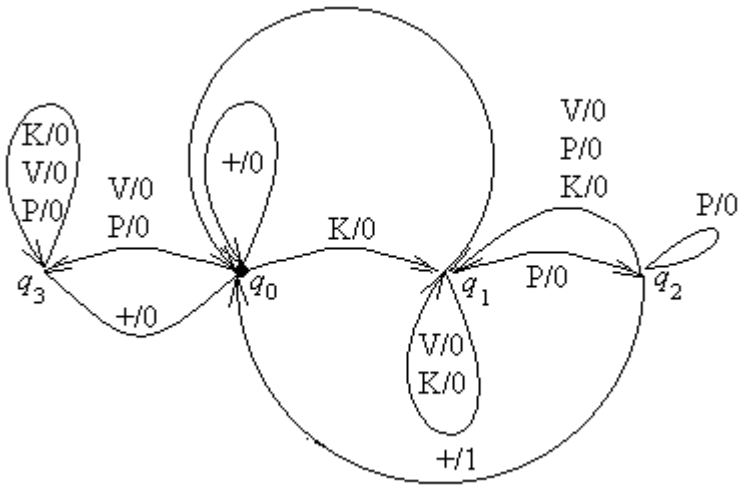


Рис. 9.1: Пример автомата Мили.

	выходы				состояния			
входы:	k	p	v	+	k	p	v	+
q_0	0	0	0	0	q_1	q_3	q_3	q_0
q_1	0	0	0	0	q_1	q_2	q_1	q_0
q_2	0	0	0	1	q_1	q_1	q_1	q_0
q_3	0	0	0	0	q_3	q_3	q_3	q_0

Рис. 9.2: Таблица переходов и выходов автомата Мили.

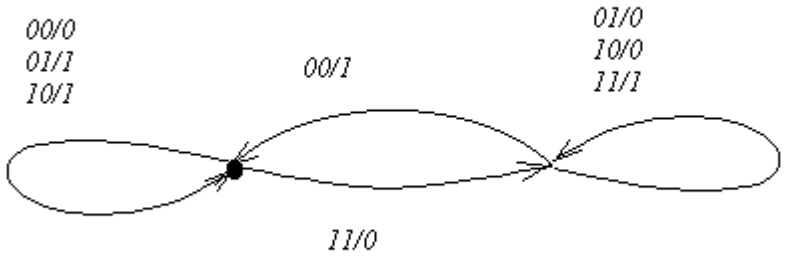


Рис. 9.3: Автомат Мили, выполняющий сложение двоичных чисел.

Пример 9.2. Построим автомат Мили, вычисляющий сумму двух двоичных чисел. Известно, что при выполнении сложения возникает единица переноса в следующий разряд:

$0+0=0;$	1 переноса $+\ 0+0=1;$
$0+1=1;$	1 переноса $+\ 0+1=0,$ 1 переноса;
$1+0=1;$	1 переноса $+\ 1+0=0,$ 1 переноса;
$1+1=0,$ 1 переноса;	1 переноса $+\ 1+1=1,$ 1 переноса.

Входной символ такого автомата представляет собой пару двоичных разрядов первого и второго слагаемого: 00, 01, 10, 11. Будем рассматривать такую пару как один символ входного алфавита. Автомат имеет два состояния. Одно из них должно соответствовать переносу в следующий разряд, второе состояние означает отсутствие переноса. Пусть это будут состояния q_1 и q_0 соответственно. Тогда автомат Мили, выполняющий сложение, можно представить графом на рис. 9.3. Автомат должен начинать работу из состояния q_0, т.к. перед сложением единица переноса отсутствует.

Итак, операцию сложения автомат Мили выполнить может, причем, никакого ограничения на длину складываемых чисел не вводится. Покажем, что не существует конечного преобразователя, способного перемножать сколь угодно длинные двоичные числа. Для такого доказательства достаточно привести пример двух чисел, перемножить которые с помощью конечного автомата невозможно. Воспользуемся методом доказательства от противного и предположим, что автомат A, умножающий сколь угодно большие числа, существует. Пусть автомат A имеет N состояний. Возьмем некоторое целое число $p > N$ и вычислим с помощью автомата A произведение $2^p \cdot 2^p$. Автомат A начинает работу в некотором состоянии. В течение первых $2p$ тактов A должен выдавать в качестве выхода 0 независимо от того, в каком состоянии он находится и получает ли он в качестве входа 00 или 11. На $(2p+1)$ – ом такте должен быть определен старший разряд результата, равный единице. На этом такте автомат A будет находиться в некотором состоянии q_i, в котором он уже был по меньшей мере один раз после того, как на такте p получил на вход 11. Это следует, во–первых, из того, что вследствие неравенства $p > N$ не все состояния, в которых A находился после получения на вход 11, различны, а, во–вторых, из того, что после входа 11 автомат A еще раз получил на вход 00 на такте с номером $p+1$, а затем получает этот же вход и на всех последующих тактах. Поэтому вход 00 на $(2p+1)$ – ом шаге должен приводить к выходу 0, чего быть не должно.

Этот пример демонстрирует ограниченность возможностей автоматов Мили — как мы уже отмечали ранее, отсутствие рабочей ленты у автомата существенно сни-

жает его возможности. При длинных входных последовательностях последние выходы конечного преобразователя зависят только от последних входов.

9.3 Автоматы Мура

Определение 9.4. Автомат Мура — это пятерка

$$U = (K, X, Y, f, h) \ ,$$

где K, X, Y, f означают то же, что и в определении 9.3 для автомата Мили (множество состояний, входной и выходной алфавит, функция переходов), а h — функция выходов, являющаяся отображением $Z \to Y$. Функции f и h связаны для автомата Мура соотношениями

$$\begin{array}{ll} q(0) = & q_0, \\ q(t+1) = & f(q(t), x(t)), \\ y(t) = & h(q(t)), \end{array}$$

Представленный графом автомат Мура имеет помеченные входными символами дуги, а каждое состояние помечено не только символом состояния, но и выходным символом.

При формальном сравнении определений 9.3 и 9.4 может показаться, что автоматы Мура могут быть заданы как входо–независимые автоматы Мили, т.е. такие автоматы Мили, выходная функция которых удовлетворяет условию

$$\forall a \in X \forall b \in X \forall z \in Z \, (g(z, a) = g(z, b)) \ .$$

Представление об автоматах Мура как о входо–независимых автоматах Мили, однако, не соответствует представлению о способе функционирования автоматов Мура в соответствии с определением 9.4. В автоматах Мура реализуется иная временная связь между переходами из одного состояния в другое и выходом по сравнению с автоматами Мили. У последних выход, соответствующий некоторому входу и определенному состоянию, порождается во время перехода автомата в следующее состояние. У автоматов Мура сначала порождается выход, а потом происходит переход в следующее состояние, причем выход определяется только состоянием автомата. В частности, автомат Мура порождает некоторый выход еще перед тем, как получит первый вход — это выход, соответствующий начальному состоянию автомата. Конечно, этот первый выход не представляет особого интереса.

Пример 9.2. Рассмотрим автомат Мура, обрабатывающий цепочки из 0 и 1 и представленный на рис. 9.4.

Этот автомат обладает таким интересным свойством, что для любой цепочки $w \in X^*$ существуют такие два состояния, выбранные в качестве начальных, что совпадают результаты переработки цепочки w. Действительно, пусть первый символ цепочки "0 тогда начиная работу как из Z_1, так и из Z_3 автомат порождает выход "0" и переходит в Z_4. Аналогично любая начинающаяся символом "1" цепочка перерабатывается одинаково из Z_1 и Z_2. Наконец, пустая цепочка перерабатывается одинаково из состояний Z_1, Z_2 и Z_3.

9.4 Равносильность автоматов Мили и Мура

Определение 9.5. Автомат Мура $U = (K_1, X, Y, f_1, h)$ и автомат Мили $M = (K, X, Y, f, g)$ называются равносильными, если множества их реакций совпадают:

$$L(M) = \{g_z | z \in K\} = \{h_t | t \in K_1\} = L(U).$$

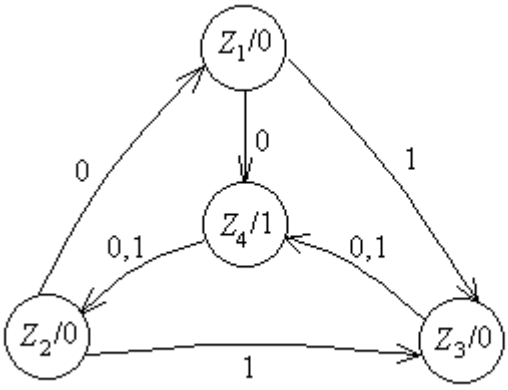


Рис. 9.4: Автомат Мура.

Теорема 9.2. Для каждого автомата Мура может быть построен равносильный автомат Мили.

Доказательство. Пусть $U = (K_1, X, Y, f_1, h)$ — произвольный автомат Мура. Граф равносильного автомата Мили M получаем, если каждому ребру графа переходов и выходов автомата U

$$z/y \xrightarrow{x} z_i/y_i$$

сопоставим ребро графа переходов и выходов автомата M

$$z \xrightarrow{x/y_i} z_i \quad .$$

Пусть $w = x_1x_2...x_k$ — входная цепочка. Тогда реакция автомата Мура M из состояния z_0 имеет вид

$$z_0/y_1, x_1 \to z_1/y_2, x_2 \to ... \to z_{k-1}/y_k, x_k \tag{9.1}$$

Последний символ цепочки x_k в этой реакции не рассматривается, т.к. он не приводит к генерации выхода на этом такте. Реакции (9.1) взаимно однозначно соответствует реакция построенного автомата Мили:

$$z_0, x_1/y_1 \to z_1, x_2/y_2 \to ... \to z_{k-1}x_k/y_k.$$

Реакции автоматов эквивалентны. □

В соответствии с доказательством теоремы 9.2 можно представить автомат Мура как частный случай автомата Мили. Действительно, пусть $U = (K_1, X, Y, f_1, h)$ — некоторый автомат Мура. Построим автомат Мили $M = (K, X, Y, f, g)$, для чего положим $g(z, x) = h(f(z, x))$ для каждого $z \in K$ и каждого $x \in X$. Тогда автомат Мили M с функцией переходов g для всех непустых входных последовательностей порождает такие же выходные последовательности, как и автомат U, если, конечно, не учитывать самый первый выход автомата U.

Рассмотрим теперь обратное соответствие.

Теорема 9.3. Для любого автомата Мили может быть построен эквивалентный автомат Мура.

Доказательство. Пусть задан $M = (K, X, Y, f, g)$ — произвольный автомат Мили.

Построим автомат Мура $U = (K_1, X, Y, f_1, h)$ следующим образом. В качестве множества его состояний возьмем множество пар $K_1 = K \times Y$. Для обеспечения

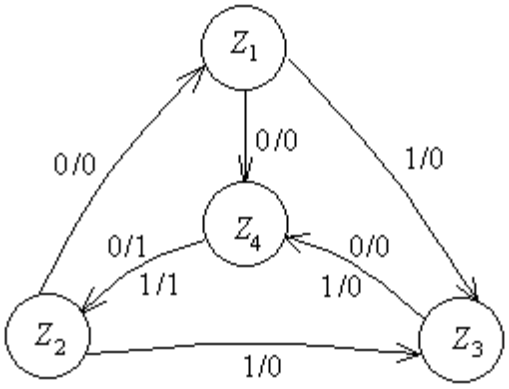


Рис. 9.5: Автомат Мили, эквивалентный автомату Мура рис. 9.4.

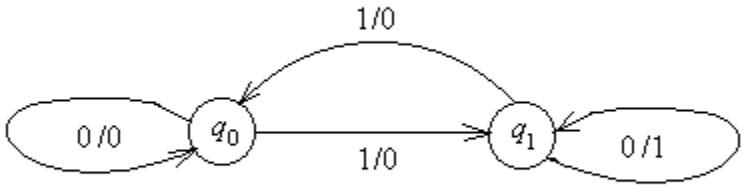


Рис. 9.6: Автомат Мили, выполняющий преобразование $((0/0)^*1/0(0/1)^*1/0)^*$.

равносильности $M = U$ определим функцию переходов и выходов для U следующим образом:

$$f_1(p \cdot y, a) = \{q \cdot b | f(p, a) = q, b \in X\},$$
$$h(p \cdot y) = y.$$

Если реакция автомата M на цепочку $x_1x_2...x_k$ из состояния z_0 имеет вид

$$z_0, x_1/y_1 \to z_1, x_2/y_2 \to ... \to z_{k-1}, x_k/y_k \ , \tag{9.2}$$

то существует такое состояние $z_0 \cdot x_1$ недетерминированного автомата U, что начиная работу из этого состояния автомат U выполняет следующие команды

$$z_0 \cdot x_1/y_1, x_1 \to z_1 \cdot x_2/y_2, x_2 \to ... \to z_{k-1} \cdot x_k/y_k, x_k \tag{9.3}$$

И обратно, из существования реакции (9.3) следует существование реакции (9.2), следовательно, M и U равносильны.

Пример 9.3. Автомату Мура из примера 9.2 (см. рис. 9.4) соответствует автомат Мили, представленный на рис. 9.5.

Пример 9.4.

Автомату Мили, представленному на рис. 9.6, соответствует автомат Мура, представленный на рис. 9.7.

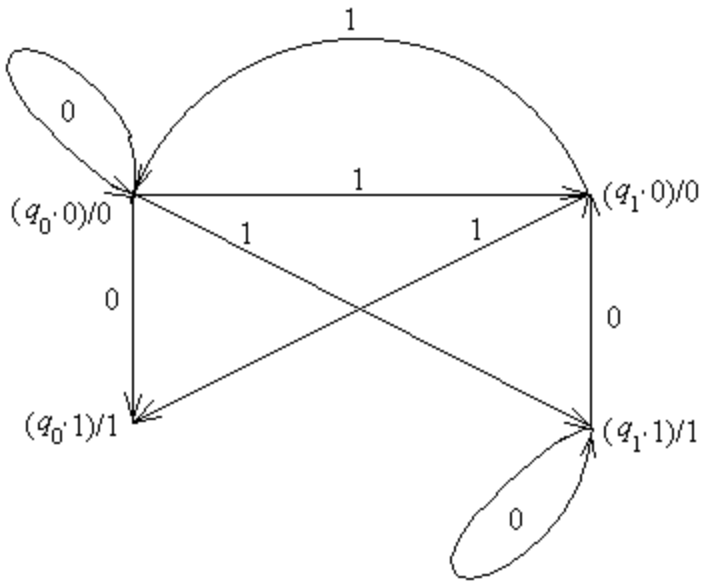


Рис. 9.7: Автомат Мура, эквивалентный автомату Мили на рис. 9.6.

9.5 Синтез конечных преобразователей

Простейший синтез конечных преобразователей в форме автоматов Мили можно выполнить по тем же алгоритмам, что и синтез преобразователей, если рассматривать конечный автомат–преобразователь как распознаватель языка над алфавитом пар символов

$$\Sigma = \{a/b \mid a \in X; b \in Y\} \ ,$$

где X и Y — соответственно входной и выходной алфавит.

Пример 9.5. Построить конечный преобразователь, моделирующий работу светофора. Пусть светофор переключается при поступлении единичного сигнала ($X = \{1\}$) и выдает на выход сигналы "красный "желтый "зеленый". Обозначим выходной алфавит $Y = \{$к,ж,з$\}$, тогда работа преобразователя может быть описана следующим языком

$$L = (1/\text{к}\ 1/\text{ж}\ 1/\text{з}\ 1/\text{ж})^*.$$

Соответствующий преобразователь имеет вид, представленный на рис. 9.8. Этому автомату Мили соответствует автомат Мура, представленный на рис 9.9.

Допустим теперь, что входных сигнала два ($X = \{0, 1\}$) и автомат должен реагировать на правильную входную последовательность $(01)^*$. Тогда

$$L = \{0/\text{к}, 1/\text{ж}, 0/\text{з}, 1/\text{ж}\}^*$$

и матрица переходов автомата имеет вид

	о/к		
		1/ж	
			0/з
1/ж			

Если считать, что не требуется выполнять контроль входной последовательности, можно доопределить пустые клетки так, чтобы можно было применить алгоритм минимизации и сократить число состояний автомата. Заметим только, что при таком

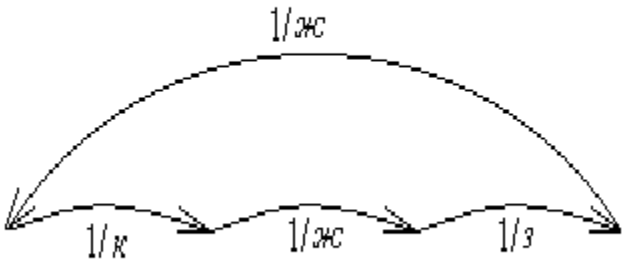


Рис. 9.8: Автомат Мили, моделирующий работу светофора.

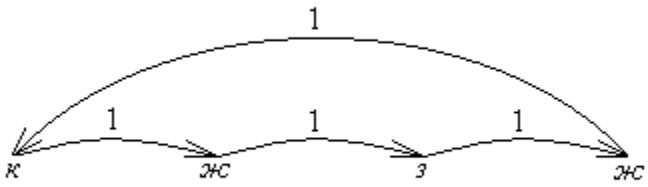


Рис. 9.9: Автомат Мура, моделирующий работу светофора.

доопределении автомат необходимо оставить детерминированным. Можно, например, доопределить автомат следующим образом:

	о/к		1/ж
0/к		1/ж	
	1/ж		0/з
1/ж		0/з	

Получим эквивалентный минимальный автомат с двумя состояниями:

о/к	1/ж
1/ж	о/з

Граф переходов и выходов этого автомата представлен на рис. 9.10.

9.6 Эксперименты по распознаванию состояний.

Реакция нетривиального автомата ($|\Sigma| > 1$) на определенную входную цепочку непредсказуема, если неизвестно начальное его состояние. С другой стороны, эта реакция всегда может быть определена, если начальное состояние известно. Таким образом, одна из основных задач анализа конечных автоматов состоит в том, чтобы распознать некоторое состояние исследуемого автомата. Можно сформулировать следующие задачи распознавания состояния:

– задача определения начального состояния, т.е. состояния, в котором находился автомат перед началом испытаний;

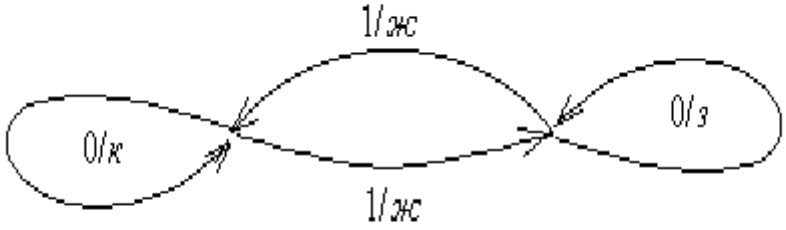


Рис. 9.10: Автомат Мили с минимальным числом состояний, моделирующий работу светофора.

– задача определения конечного состояния, т. е. состояния, в которое перейдет автомат после завершения испытательных операций.

Кроме этих диагностических задач существует установочная задача по приведению автомата в заданное состояние, если начальное состояние его неизвестно.

Процесс приложения входных последовательностей к автоматам и наблюдения получаемых выходных цепочек называется экспериментом. Будем различать два типа экспериментов:

– безусловные эксперименты, когда входная цепочка полностью определена заранее;

– условные эксперименты, когда входная цепочка x состоит из двух или более подцепочек $x = x_1x_2...x_n$ и каждая последующая подцепочка x_{i+1} выбирается на основе реакции на x_i.

Рассмотрим сначала диагностические эксперименты для двух состояний. Пусть автомат Мили $A = (K, X, Y, f, g)$ имеет $n = |K|$ состояний. По аналогии с автоматами–распознавателями можно рассмотреть алгоритм минимизации автоматов с выходом, поэтому можем считать, что A — минимальный автомат и, следовательно, он не имеет эквивалентных состояний. Тогда любые $p_i \in K$ и $p_j \in K$ являются $(n-1)$–различимыми.

Определение 9.6. Входная цепочка $x, |x| \leq n-1$, которая, будучи приложена к M/p_i и M/p_j, вызывает различные выходные цепочки, называется диагностической цепочкой для пары состояний (p_i, p_j).

Теорема 9.4. Для любых двух состояний p_i, p_j минимального автомата $A = (K, X, Y, f, g)$ существует диагностическая цепочка длины не более $n-1$, где $n = |K|$.

Доказательство. Рассмотрим способ минимизации преобразователя как модификацию метода минимизации распознавателя, выполняя разбиение множества состояний на k–эквивалентные для $k = 1, 2, ..., n-1$. Только, в отличие от распознавателей, такое разбиение должно проводиться с учетом выходных цепочек. Тогда в матрице переходов перестановка строк и столбцов и выделение группы эквивалентности выполняется на основе анализа пары символов "вход/выход". Если диагностируемые состояния p_i и p_j оказались смежными в матрице M_k и разобщенными в матрице M_{k+1}, то они являются $(n-k)$– эквивалентными и $(n-k+1)$–различимыми. Тогда должен существовать хотя бы один входной символ b, такой, что цепочка $b\phi$ из p_i и p_j вызывает разную реакцию. Этот символ определяется по строкам p_i и p_j матрицы M_k с элементами b/c_1 и b/c_2 в некоторых столбцах p_m и p_l. Выполняя ана-

логичную операцию поиска диагностирующего входного символа для найденных p_m и p_l, построим всю диагностирующую последовательность.□

Следствие 9.1. Диагностическая задача с двумя допустимыми состояниями для автомата, содержащего n состояний, разрешима простым безусловным экспериментом длины не более $n-1$.

Пример 9.6. Найти диагностическую последовательность для состояний p_1 и p_2 следующего автомата.

	1	2	3	4	5
1	a/0			b/0	
2	a/0				b/0
3	b/1				a/0
4			a/1	b/1	
5		a/1			b/1

Решение задачи основано на доказанной теореме. Объединим состояния 1—2, 4—5, 3, содержащие в строках матрицы одинаковый набор пар вход/выход (пары a/0, b/0 для 1–2; a/1,b/1 для 3; a/1,b/1 для 4—5). Получим матрицу переходов, в которооой выделены группы 1–эквивалентных состояний:

	1	2	3	4	5
1	a/0			b/0	
2	a/0				b/0
3	b/1				a/0
4			a/1	b/1	
5		a/1			b/1

Состояния 4 и 5 не эквивалентны, т.к. переходы из этих состояний в неэквивалентные группы 1–2 и 3 выполняется по–разному. Выполним разделение на группы:

	1	2	3	4	5
1	a/0			b/0	
2	a/0				b/0
3	b/1				a/0
4			a/1	b/1	
5		a/1			b/1

Теперь очевидно, что не эквивалентны и состояния p_1 и p_2. Разбиваем их на группы:

	1	2	3	4	5
1	a/0			b/0	
2	a/0				b/0
3	b/1				a/0
4			a/1	b/1	
5		a/1			b/1

Возвращаясь назад — от последнего разбиения к первому — построим диагностическую последовательность. Первый шаг обратного хода соотвествует последнему разбиению состояний по группам эквивалентности. Последнее разбиение было выполнено для состояний p_1 и p_2: эквивалентность этих состояний нарушает переход в неэквивалентные состояния p_4 и p_5 по символу b. Этот символ является первым символом диагностической последовательности. Второй шаг обратного хода — переход

из p_4 и p_5 в неэквивалентные состояния p_2 и p_3 по символу a. В результате диагностическая последовательность начинается символами ba. Третий шаг обратного хода — переход из p_2 и p_3 в p_1 и p_5 по символу a или b. Теперь получили две цепочки, начинающих диагностическую последовательность: baa и bab. Наконец, последний шаг обратного хода соответствует первому разбиению p_1 и p_5 по причине различного выхода при $a/0$ и $a/1$. Таким образом, получили две итоговых диагностических последовательности: $baaa$ и $baba$. Результирующий выход, например, на последовательность $baaa$ из состояния p_1 — цепочка 1101, а из состояния p_2 — цепочка 1100, т.е. последний символ 1 или 0 диагностической последовательности однозначно определяет начальное состояние p_1 или p_2.

Рассмотрим эксперименты по распознаванию произвольного числа m ($m > 2$) состояний. Для этого сначала построим диагностическое дерево. Каждая вершина этого дерева соответствует списку групп состояний, а дуги — входным символам, по которым осуществляется переход в одну или разные группы потомков. Если в результате перехода по дуге дерева выходные символы различны, то группа результирующих состояний делится на подгруппы в соответствии с выходными символами. Состояния в одной группе могут повторяться, в списке могут быть одинаковые группы. Группа состояний, содержащая единственный элемент, называется простой. Группа, содержащая два или более одинаковых элементов, называется кратной. Список групп называется простым, если все его группы простые. Вершина k–го уровня диагностического дерева является листом, если она удовлетворяет одному из следующих условий:

– вершина помечена меткой, совпадающей с меткой некоторого предка этой вершины (очевидно, что в этом случае диагностическая последовательность, соответствующая пути от корня к данной вершине, не позволит различить ни одно из состояний);

– вершина помечена меткой, содержащей некоторую кратную группу (неразличимы состояния кратной группы);

– метка вершины соответствует простому списку (соответствующая диагностическая последовательность позволяет различить все состояния корня);

– имеется в дереве другая вершина k–го уровня, соответствующая простому списку.

Таким образом, появление хотя бы одного простого списка на некотором уровне завершает построение дерева.

Пример 9.7. Диагностическое дерево для автомата из примера 9.6 и группы допустимых состояний $\{2, 3, 4, 5\}$ имеет вид, представленный на рис. 9.11. Последовательность входных символов aaa позволяет различить все четыре состояния.

Определение 9.7. Диагностическим путем называется путь в диагностическом дереве, конечная вершина которого помечена простым списком. Очевидно, что входная цепочка, соответствующая диагностическому пути, есть диагностическая цепочка для состояний, соответствующих корню. В примере 9.7 диагностическая последовательность aaa, примененная к $\{2, 3, 4, 5\}$ решает задачу определения состояния:

1) выходная цепочка 000 — состояние 2;

2) выходная цепочка 010 — состояние 3;

3) выходная цепочка 101 — состояние 4;

4) выходная цепочка 100 — состояние 5.

Не для любого исходного множества состояний диагностическое дерево завершается вершиной с простым списком. Например, для данного автомата нельзя выполнить диагностику всех пяти состояний, т.к. соответствующее дерево имеет вид, представленный на рис. 9.12.

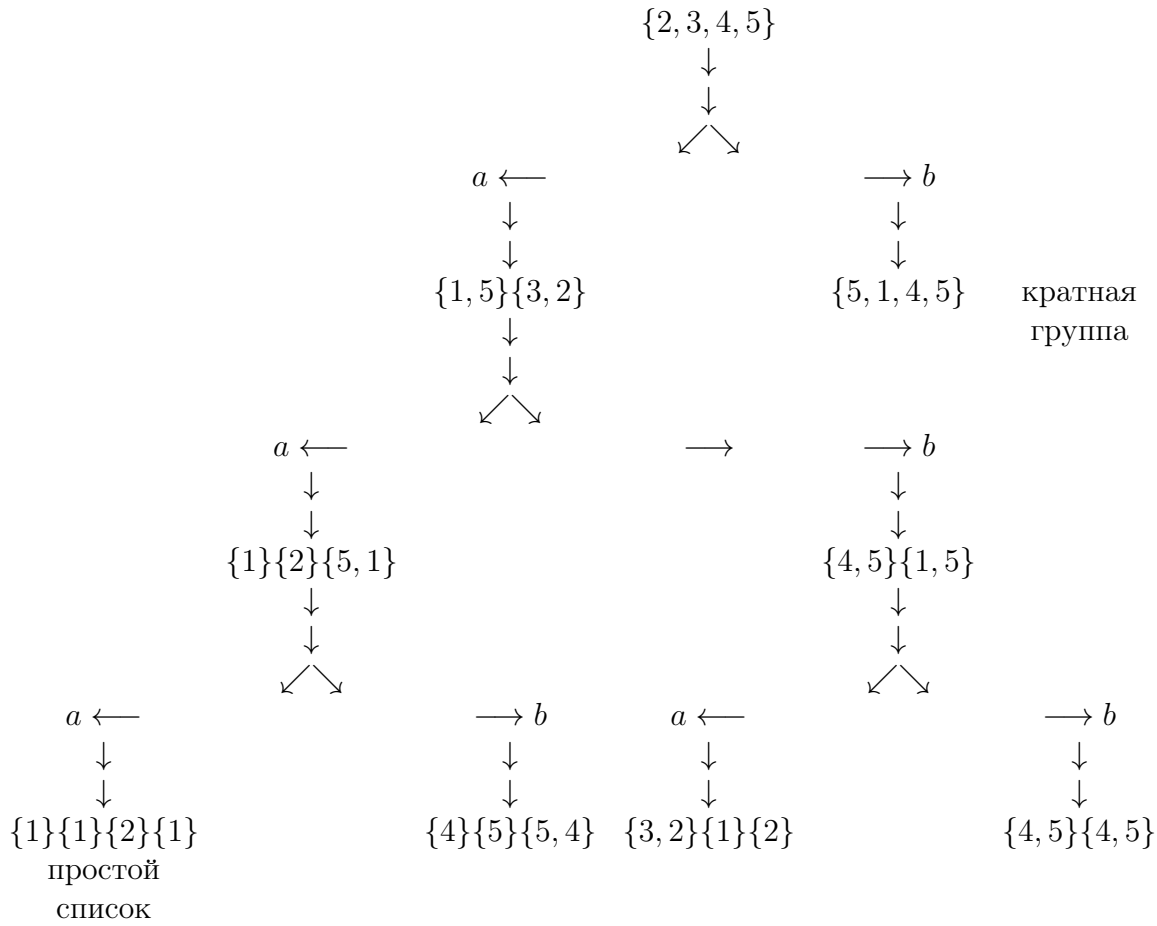


Рис. 9.11: Диагностическое дерево для четырех состояний.

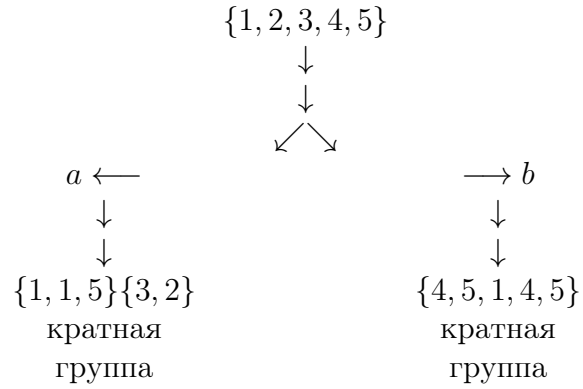


Рис. 9.12: Диагностическое дерево для пяти состояний.

Рассмотрим теперь условные эксперименты. Пусть в диагностическом дереве для группы состояний A имеется путь, который ведет к вершине, содержащей простую группу T (сама вершина может не быть простой, т.к. она может содержать другие группы). Следовательно, если окажется, что состояние, соответствующее T является истинным состоянием, то его можно распознать последовательностью, соответствующей пути, завершающемуся в T. Если состояние не является T, то можно применить новую диагностическую последовательность для оставшихся состояний. Эта схема является решением диагностической задачи с помощью простого условного эксперимента. Очевидный недостаток простого эксперимента заключается в том, что он может оказаться разрушительным. Если имеется только один экземпляр автомата, то неудачный эксперимент может перевести его в такую группу состояний, диагностика которых невозможна. В этом случае используют так называемые кратные эксперименты на нескольких экземплярах исходного автомата. Используя теорему 9.4, можно с помощью кратких экспериментов однозначно определить исходное состояние.

Упражнение. Написать программу для ЭВМ кратных экспериментов для автомата с m состояниями.

9.7 Алгоритмически разрешимые и неразрешимые проблемы теории формальных грамматик и языков

Напомним, что свойство объектов определенного класса называется алгоритмически распознаваемым, если существует общий алгоритм, позволяющий для любого конкретного объекта этого класса ответить на вопрос, обладает ли он данным свойством. Свойство называется алгоритмически неразрешимым, если такого алгоритма не существует. Чтобы установить неразрешимость некоторого свойства для данного класса C, достаточно установить его неразрешимость для какого–либо подкласса класса C.

Одним из основных условий, которое должно выполняться для любого языка программирования, является возможность определить по каждой заданной последовательности символов является ли она программой на этом языке. Это необходимо для того, чтобы сделать возможным автоматическое распознавание программ и их трансляцию. Фактически, это требование означает возможность обнаружения ошибок при компиляции программы. Для описания синтаксиса языков программирования обычно используются КС–грамматики. Тогда на формальном уровне это требование означает, что проблема вхождения любой цепочки в КС–язык алгоритмически разрешима. Другими словами, существует алгоритм, позволяющий определить, принадлежит ли произвольная цепочка заданному КС–языку. Этот факт мы доказали в теореме 6.7. К сожалению, большинство интересных алгоритмических проблем решается отрицательно для КС–языков, т.е. соответствующих алгоритмов не существует. Среди таких вопросов можно назвать следующие:

– по двум произвольным КС–грамматикам узнать, порождают ли они один и тот же язык;

– пересекаются ли языки, порождаемые двумя произвольными заданными КС–грамматиками;

– существует ли конечный преобразователь, отображающий заданный КС–язык на другой заданный КС–язык.

Если программисту все же необходимо получать ответы на такие вопросы, то либо языки программирования должны принадлежать к некоторому подходящему подклассу класса всех КС–языков, либо должны быть формализованы дополнительные аспекты языков программирования, например, семантические.

Упражнения. Доказать разрешимость поставленных выше проблем для автоматных грамматик. Это значит, что для произвольных регулярных языков L_1 и L_2 необходимо доказать разрешимость следующих соотношений:

a) $L(G_1) = L(G_2)$?;

b) $L(G_1) \cap L(G_2) = \emptyset$?;

c) $P(L(G_1)) = L(G_2)$?.

Здесь G_1 и G_2 — автоматные грамматики, порождающие языки L_1 и L_2 соответственно, P — конечное преобразование.

Определение 9.8. Пусть имеются два произвольных набора из n непустых слов в афавите Σ :

$$\begin{array}{c}(x_1, x_2, ..., x_n),\\ (y_1, y_2, ..., y_n).\end{array} \tag{9.4}$$

Спрашивается, существует ли непустая конечная последовательность индексов

$$i_1, i_2, \ldots, i_k,$$

такая, что совпадают слова

$$x_{i_1} x_{i_2} \ldots x_{i_k} = y_{i_1} y_{i_2} \ldots y_{i_k}.$$

Проблема поиска последовательности индексов, удовлетворяющей указанному условию, называется проблемой соответствий слов или *проблемой Поста.*

Например, если даны два набора

(ОРИТ,ЛГ,А,ОР,М) и (РИ,О,АЛГ,ТИ,ТМ),

то последовательность индексов 3, 2, 1, 5 сформирует слово

А—ЛГ—ОРИТ—М = АЛГ–О—РИ—ТМ.

Теорема 9.5. Проблема соответствий слов неразрешима.

Доказательство. Пусть заданы наборы слов типа (9.4):

$$x_i, y_i \in \Sigma, x_i = a_{i_0} a_{i_1} \cdots a_{i_p} \text{ и } y_i = b_{i_0} b_{i_1} \cdots b_{i_q} \ .$$

Будем рассматривать числа в P–ичной системе счисления, где в качестве P выберем число символов в алфавите Σ. Тогда цепочкам x_i и y_i соответствуют числа $\hat{x_i}$, $\hat{y_i}$ и их длины $|x_i|, |y_i|$. Решение проблемы соответствия эквивалентно поиску такого числа, которое строится из $\widehat{x_i}$ и $\widehat{y_i}$ двумя способами:

$$\sum_{j=0}^{t} \hat{x}_{i_j} P^{\sum_{k=0}^{j-1} |x_{i_k}|} = \sum_{j=0}^{t} \hat{y}_{i_j} P^{\sum_{k=0}^{j-1} |y_{i_k}|}$$

Таким образом, решение проблемы соответствия эквивалентно поиску целочисленного решения диофантового уравнения, что, как известно, является неразрешимой проблемой. (Диофантовым уравнением называется уравнение с целочисленными коэффициентами. Задача поиска целочисленного решения диофантова уравнения называется десятой проблемой Гильберта и является неразрешимой проблемой, как это уже отмечалось в главе 4.)□

Неразрешимость проблемы Поста часто используется при доказательстве неразрешимости многих других проблем. В качестве примера таких проблем рассмотрим проблему пустоты пересечения КС–языков и проблему неоднозначности КС–грамматик.

Теорема 9.6. Проблема пустоты пересечения КС–языков неразрешима.

Доказательство. Покажем, что эта проблема уже неразрешима для случая металинейных грамматик. Грамматика называется металинейной, если ее правила имеют вид $A \to \alpha B \beta$ или $A \to \alpha$, где A, B — нетерминальные символы, а α, β — терминальные цепочки.

Рассмотрим n пар произвольных цепочек над алфавитом $\Sigma = \{a, b\}$:

$$(g_1, h_1),\ (g_2, h_2),\ \ldots,\ (g_n, h_n).$$

Рассмотрим также n различных символов — маркеров m_1, m_2, ..., m_n и две металинейные грамматики:

$$\begin{array}{rl} G_1: & S_1 \to m_1 S_1 g_1 | m_2 S_1 g_2 | ... | m_n S_1 g_n \\ & S_1 \to c \end{array}$$

$$\begin{array}{rl} G_2: & S_2 \to m_1 S_2 h_1 | m_2 S_2 h_2 | ... | m_n S_2 h_n \\ & S_2 \to c \end{array}$$

Проблема пустоты пересечения $L(G_1) \cap L(G_2)$ по существу оказывается проблемой Поста, т.к. она равносильна вопросу о возможности равенства

$$m_{i_1} m_{i_2} ... m_{i_k} c g_{i_k} ... g_{i_2} g_{i_1} = m_{i_1} m_{i_2} ... m_{i_k} c h_{i_k} ... h_{i_2} h_{i_1}$$

для какой–нибудь последовательности индексов. Таким образом, проблема неразрешима уже для металинейных грамматик, следовательно, она неразрешима и в общем виде. □

Теорема 9.7. Проблема неоднозначности КС–грамматики неразрешима.

Доказательство. Так же, как и для теоремы 9.6, покажем, что эта проблема уже неразрешима для случая металинейных грамматик. Построим грамматику, порождающую объединение языков $L(G_1)$ и $L(G_2)$ теоремы 9.6:

$$\begin{array}{rl} G_3: & S \to S_1 | S_2 \\ & S_1 \to m_1 S_1 g_1 | m_2 S_1 g_2 | ... | m_n S_1 g_n \\ & S_1 \to c \\ & S_2 \to m_1 S_2 h_1 | m_2 S_2 h_2 | ... | m_n S_2 h_n \\ & S_2 \to c \end{array}$$

Грамматика G_3 является неоднозначной тогда и только тогда, когда существует хотя бы одна цепочка x, которую можно вывести двумя способами:

$$S \Rightarrow S_1 \stackrel{*}{\Rightarrow} m_{i_1} m_{i_2} ... m_{i_k} c g_{i_k} ... g_{i_2} g_{i_1},$$

$$S \Rightarrow S_2 \stackrel{*}{\Rightarrow} m_{i_1} m_{i_2} ... m_{i_k} c h_{i_k} ... h_{i_2} h_{i_1}.$$

Узнать, существует ли такая цепочка x — это опять проблема Поста. □

9.8 Контрольные вопросы к разделу

1. Чем формальное определение конечного автомата – распознавателя отличается от формального определения конечного автомата – преобразователя?

2. Чем автомат Мили отличается от автомата Мура?

3. Как определяются действия синхронного преобразователя и чем они отличаются от действий асинхронного преобразователя?

4. Чем инициальный автомат отличается от неинициального автомата?

5. Как для произвольного автомата Мили построить эквивалентный автомат Мура?

6. Как для произвольного автомата Мура построить эквивалентный автомат Мили?

7. Как построить автомат Мили, преобразующий один регулярный язык в другой?

8. Как минимизировать автомат Мили?

9. Зачем используются диагностические эксперименты по распознаванию состояний?

10. Приведите пример диагностического дерева.

11. Сформулируете теорему о длине диагностической последовательности для минимального конечного преобразователя.

12. В какой язык конечный преобразователь преобразует произвольный КС-язык?

13. Каким языком является пересечение регулярного языка и КС-языка?

14. Сформулируете проблему Поста.

15. Проблема Поста разрешима?

16. Приведите пример разрешимой проблемы для КС-языков.

17. Приведите пример неразрешимой проблемы для КС–языков.

18. Сколько состояний имеет автомат Мура, эквивалентный заданному автомату Мили с n состояниями?

19. Сколько состояний имеет автомат Мили, эквивалентный заданному автомату Мура с n состояниями?

20. Как Вы считаете, какой из автоматов — Мили или Мура — проще построить и почему?

9.9 Упражнения к разделу

Задание. Построить автомат Мили, выполняющий заданное синхронное преобразование. Если в соответсвии со структурой исходной и результирующей цепочки может оказаться, что синхронное преобразование в требуемый язык не всегда возможно, выполнить уточнение задания и построить точную формулу синхронного преобразования.

9.9.1 Задача

Задано следующее синхронное преобразование:

$$a^*b^+ \overset{*}{\Rightarrow} (01)^*. \tag{9.5}$$

Решение. В соответствии с заданием, исходная цепочка может иметь произвольную длину, начиная от 2, а длина результирующей цепочки всегда кратна 2. Поэтому будем выполнять синхронный перевод в последовательность символов 01 до тех пор, пока не закончится исходная цепочка. Таким образом, перевод будет выполняться в язык $(01)^*(\varepsilon \cup 0)$. Это можно сделать, если исходную цепочку также рассмотреть состоящий из пар символов:

$$(aa)^*(\varepsilon \cup a)b^+ = (aa)^*b^+ \cup (aa)^*ab^+ = (aa)^*b(bb)^*(\varepsilon \cup b) \cup (aa)^*ab(bb)^*(\varepsilon \cup b).$$

Тогда требуемое преобразование можно описать формулой:

$$(a/0a/1)^*b/0(b/1b/0)^*(\varepsilon \cup b/1) \cup (a/0a/1)^*a/0b/1(b/0b/1)^*(\varepsilon \cup b/0)$$

или в более простой форме, полученной после вынесения за скобки:

$$(a/0a/1)^*(b/0(b/1b/0)^*(\varepsilon \cup b/1) \cup a/0b/1(b/0b/1)^*(\varepsilon \cup b/0)).$$

Автомат должен быть инициальным, т.к. каждая отдельно взятая цепочка исходного языка должна распознаваться из одного и того же состояния. Следовательно, над алфавитом $\{a/0, a/1, b/0, b/1\}$ можно построить автомат, представленный на рис. 9.13.

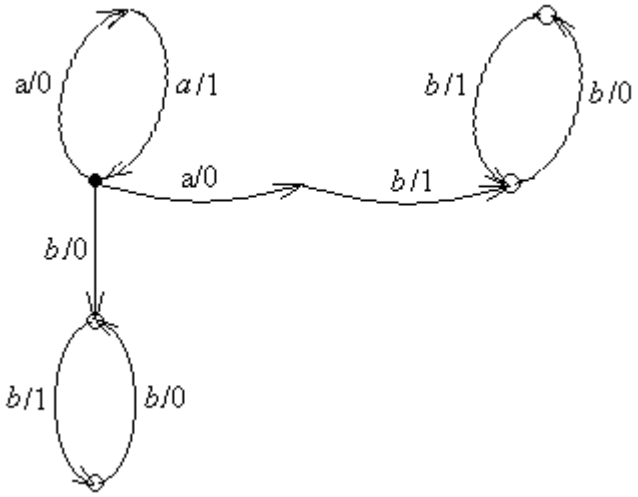


Рис. 9.13: Автомат Мили, выполняющий синхронное преобразование (9.5).

Можно заметить, что автомат не является минимальным. Для минимизации автомата построим его матрицу переходов и выделим группы эквивалентных состояний:

	a/1						
a/0		a/0				b/0	нач.
					b/0		закл.
						b/0	закл.
			b/1				закл.
				b/1			закл.

Заменим каждую пару эквивалентных состояний на одно состояние, тогда минимальному автомату соответствует матрица:

	a/1				
a/0		a/0		b/0	нач.
				b/0	закл.
			b/1		закл.

Этот автомат можно представить в виде графа на рис. 9.14.

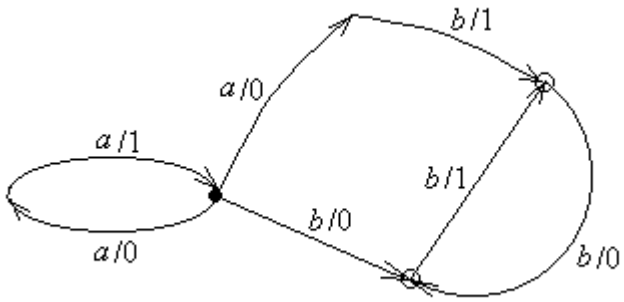


Рис. 9.14: Минимальный автомат Мили, выполняющий синхронное преобразование (9.5).

9.9.2 Варианты заданий

1. $(abc)^* \Rightarrow 1(01)^*$.
2. $(b \cup aa)^* \Rightarrow (1)^*1$.
3. $(a \cup bb)^+ \Rightarrow 0(1)^*1$.
4. $(ab \cup ba)^* \Rightarrow (100)^*$.
5. $(acc)^+ \Rightarrow 00(1)^*0$.
6. $(bc)^* \Rightarrow 1(001)^*$.
7. $(b \cup bab)^+ \Rightarrow (1)^*0$.
8. $(a \cup ba)^* \Rightarrow (10)^*$.
9. $(aa \cup bb)^+ \Rightarrow 0(1)^*0$.
10. $(bac)^* \Rightarrow (01)^*1$.
11. $(bca \cup b)^+ \Rightarrow 0(1)^*$.
12. $(a \cup b)^+ \Rightarrow (10)^*1$.
13. $(abb)^+ \Rightarrow 0(1)^*0$.
14. $(ac)^* \Rightarrow (101)^*1$.
15. $(c \cup ab)^+ \Rightarrow (1110)^*$.
16. $(a \cup bb)^+ \Rightarrow 0(10)^*1$.
17. $(a)^* \Rightarrow 0(01)^*1$.
18. $(abc)^+ \Rightarrow 0(1)^*0$.
19. $(ca \cup b)^+ \Rightarrow (110)^*$.
20. $(ba)^* \Rightarrow 0(1)^*1$.

9.10 Тесты для самоконтроля к разделу

1. Сколько состояний имеет минимальный конечный автомат Мили, который преобразует язык a^*b^* в язык c^*?

Варианты ответов:

а) 1;

б) 2;

в) 3;

г) 4;

д) 5.

Правильный ответ: в.

2. Сформулировано несколько утверждений.

1) Для любого автомата Мили существует равносильный автомат Мура.

2) Каждое конечное преобразование сохраняет свойство множества быть КС–языком.

3) Каждое конечное преобразование сохраняет свойство множества быть регулярным.

Какое из указанных утверждений ложно?

Варианты ответов:

а) ложно 1;

б) ложно 2;

в) ложно 3;

г) ложны 1 и 2;

д) ложны 1 и 3;

е) ложны 2 и 3;

ж) все утверждения ложны;

з) все утверждения истинны.

Правильный ответ: з.

3. Дан конечный преобразователь Мили:

	p_0	p_1	p_2	p_3
p_0		$a/1$	b_1	
p_1		$a/0$	$b/1$	
p_2			$a/0$	$b/1$
p_3	$a/0$			$b/0$

Какая цепочка из указанных ниже является диагностической для распознавания состояний p_0 и p_2?

1) a; 2) b; 3) aa; 4) ab; 5) abb;

Варианты ответов:

а) все, кроме 1;

б) все, кроме 2;

в) только 3 и 4;

г) только 3 и 5;

д) все, кроме 1 и 2;

е) только 5.

Правильный ответ: г.

4. Какие из следующих определений сформулированы правильно?

1) Условным диагностическим экспериментом называется процесс приложения заранее определенных входных последовательностей к автоматам и наблюдение выходных цепочек.

2) Диагностической цепочкой для пары состояний p_i и p_j автомата M называется входная цепочка, которая, будучи приложена к M/p_i и M/p_j, вызывает различные выходные цепочки.

3) Диагностическим деревом называется дерево, каждая вершина которого соответствует состояниям, а дуги – входным символам, по которым осуществляется переход в новые состояния.

Варианты ответов:

а) все определения сформулированы ошибочно;

б) все определения сформулированы правильно;

в) правильно только 1;
г) правильно только 2;
д) правильно только 3;
е) правильно только 1 и 2;
ж) правильно только 1 и 3;
з) правильно только 2 и 3.

Правильный ответ: г.

5. Проблема Поста:
а) является примером алгоритмически неразрешимой проблемы;
б) является примером проблемы, разрешимой за полиномиальное время;
в) является примером NP–полной проблемы;
г) это только сформулированная проблема поиска соответствий слов, но на современном этапе развития науки еще не доказано, разрешима она или нет.

Правильный ответ: а.

ЗАКЛЮЧЕНИЕ

Теория формальных исчислений является одним из важнейших компонентов подготовки специалистов по направлению "Информатика и вычислительная техника". Курс служит формированию знаний и умений, которые образуют теоретический фундамент, необходимый для корректной постановки и решения проблем в области программирования.

Формализация понятия алгоритма позволяет ставить вопрос о строгом доказательстве разрешимости или неразрешимости поставленной проблемы. Рассмотренные элементы теории алгоритмов дают понимание того, что можно и чего нельзя сделать с помощью вычислительных машин. Многие алгоритмически неразрешимые проблемы имеют вполне реальную интерпретацию в практике программирования. Неразрешимости возникают при решении достаточно широкого круга задач. Для программиста важно знать, что отсутствие общего алгоритма, решающего поставленную проблему, вовсе не означает, что в каждом частном случае нельзя разработать алгоритм решения. Появление неразрешимостей — это, как правило, следствие черезмерной общности задачи.

Теория формальных языков возникла на стыке математической логики, теории алгоритмов и алгебры. Математическая логика и теория алгоритмов дали математике новый объект исследования — текст, обладающий синтаксической структурой и смыслом (семантикой языка). Мощным стимулом для изучения формальных языков явились приложения математики к естественным языкам и языкам программирования.

Рассмотренные нами понятия, алгоритмы и методы теории формальных языков, грамматик и автоматов являются одним из фундаментальных осеов современной теоретической и практической информатики. Наряду с классическими приложениями, такими как проектирование программ обработки текстов и построение трансляторов языков программирования, в последнее время появились новые области применения этой теории.

Дальнейшие детали и обсуждение затронутых вопросов можно найти в работах, предлагаемых в списке литературы. **Основная литература**

1. Хопкрофт Д., Мотвани Р., Ульман Д. Введение в теорию автоматов, языков и вычислений. — М. "Вильямс", 2008, 378с.

Дополнительная литература

1. Ахо А., Ульман Дж.. Теория синтаксического анализа, перевода, компиляции. В 2 т. Т. 1,2. — М.: Мир, 1980.

2. Бек Л. Введение в системное программирование. - М.: Мир, 1988, 448 с.

3. Вирт Н. Алгоритмы и структуры данных. — СПб: "Невский диалект" , 2001,

351 с.

4. Гордеев А.В., Молчанова А.Ю. Системное программное обеспечение. — СПб, "Питер" , 2002, 736 с.

Оглавление