И 269

ВЫСШЕЕ ОБРАЗОВАНИЕ

В.И. Игошин

ТЕОРИЯ АЛГОРИТМОВ

УЧЕБНОЕ ПОСОБИЕ

Электронно-
Библиотечная
Система
znanium.com

ВЫСШЕЕ ОБРАЗОВАНИЕ

серия основана в 1996 г.

В.И. Игошин

ТЕОРИЯ АЛГОРИТМОВ

УЧЕБНОЕ ПОСОБИЕ

Рекомендовано
УМО по образованию в области подготовки
педагогических кадров в качестве учебного пособия
для студентов высших учебных заведений,
обучающихся по специальности 44.03.05 (050201.65) — математика

Электронно-
Библиотечная
Система
znanium.com

Москва
ИНФРА-М
2016

УДК 512.8; 161.2(075.8)
ББК 22.12; 87.4я73
И269

ФЗ № 436-ФЗ	Издание не подлежит маркировке в соответствии с п. 1 ч. 4 ст. 11

Игошин В.И.

И269 Теория алгоритмов: Учеб. пособие. — М.: ИНФРА-М, 2016. — 318 с. — (Высшее образование).

ISBN 978-5-16-005205-2

Подробно изложены три формализации понятия алгоритма — машины Тьюринга, рекурсивные функции и нормальные алгоритмы Маркова, доказана их эквивалентность. Рассмотрены основные теоремы общей теории алгоритмов, теория разрешимых и перечислимых множеств, алгоритмически неразрешимые массовые проблемы, теория сложности вычислений и массовых проблем, алгоритмические проблемы математической логики и других разделов математики. Охарактеризованы взаимосвязи теории алгоритмов с компьютерами и информатикой.

Для студентов университетов, технических и педагогических вузов, обучающихся по специальностям «Математика», «Прикладная математика», «Математик-педагог», «Учитель математики» на уровнях бакалавриата, магистратуры, а также специалитета.

УДК 512.8; 161.2(075.8)
ББК 22.12; 87.4я73

ISBN 978-5-16-005205-2

© Игошин В.И., 2012

Предисловие

Конкретные алгоритмы в математике и практике известны давно. В настоящее время, особенно в связи с глобальной компьютеризацией, они прочно вошли в нашу жизнь. Развитие математики и математической логики, а также в немалой степени стремительный прогресс компьютерной техники вызвали к жизни и заставили развиться в 30-60-ые годы прошлого века общую науку об алгоритмах – теорию алгоритмов. В ней объектом исследования стало именно понятие алгоритма как таковое и его самые общие свойства. Предтечами этой теории явились идеи голландского математика Л.Э.Я.Брауэра и немецкого математика Г.Вейля, высказанные ими в 20-ые годы XX в., в которых они обратили внимание на различие между конструктивными и неконструктивными доказательствами в математике. Рассуждая конструктивно при доказательстве существования какого-либо объекта с заданными свойствами, математик указывает способ (алгоритм) его получения. В отличие от этого, неконструктивное доказательство, или, как говорят, доказательство чистого существования не позволяет явно построить такой объект; оно лишь устанавливает логическое противоречие, если предположить, что такого объекта не существует.

Сначала были выработаны несколько формализаций до того лишь интуитивно осознаваемого понятия алгоритма – машины Тьюринга, рекурсивные функции, алгоритмы Маркова, – доказана их эквивалентность и сформулирован знаменитый тезис Чёрча. Это позволило строго доказать алгоритмическую неразрешимость большого количества массовых проблем и начать развивать общую (или абстрактную) теорию алгоритмов, в которой конкретный формализм отошёл на второй план, а главными понятиями, по существу, снова стали интуитивно понимаемые понятия алгоритма и вычислимой функции.

Всем этим вопросам и посвящается данное учебное пособие.

В главе I рассматриваются примеры алгоритмов из реальной жизни и из математики, обсуждается понятие алгоритма на интуитивной основе и отмечается необходимость его уточнения.

Главы II - IV посвящены конкретным формализациям понятия алгоритма – машинам Тьюринга, рекурсивным функциям и нормальным алгоритмам Маркова.

В главе V рассматриваются основные методы общей теории алгоритмов метод нумераций и метод диагонализации (или диагональный метод Кантора). С их помощью доказываются фундаментальные результаты общей теории алгоритмов, установленные С.Клини – теорема о параметризации (s-m-n-теорема) и теорема о неподвижной точке; обсуждаются применения этих теорем в теории и практике программирования.

Разрешимым и перечислимым множествам посвящается глава VI. Здесь два классических способа задания множеств изучаются с точки зрения алгоритмической эффективности, конструктивности. Важность понятий разрешимости и перечислимости множеств для оснований математики связана с тем, что язык теории множеств является в известном смысле универсальным языком математики.

В главе VII рассматриваются алгоритмически неразрешимые массовые проблемы. Сначала в качестве понятия, уточняющего понятие алгоритма, используется понятие машины Тьюринга (§7.1). Затем проблема алгоритмической разрешимости рассматривается в рамках общей теории алгоритмов. Завершается глава рассмотрением частично разрешимых массовых проблем и связанных с ними частично разрешимых предикатов.

Вслед за общей теорией алгоритмов стали возникать и развиваться многочисленные теории различных конкретных алгоритмов, призванных решать те или иные конкретные математические задачи, являвшиеся, в основном, математическими моделями реальных научных и производственных задач. Эти конкретные теории питали своими идеями и проблемами общую теорию алгоритмов. Так, в общей теории возникли проблемы сложности алгоритмов, проблемы сведения одних массовых проблем к другим при их алгоритмическом решении и т.д. В свою очередь, общая теория алгоритмов создавала методологическую основу для многочисленных конкретизаций.

Вопросам сложности вычислений и массовых проблем посвящена глава VIII. В ней рассматриваются способы сравнения и классификации массовых проблем и алгоритмов по их сложности, выделя-

ются различные классы сложности массовых проблем, важнейшими из которых являются классы **P**, **NP** и NP-полных массовых проблем.

Наконец, завершают книгу две главы IX и X, посвящённые алгоритмическим проблемам математической логики и различных разделов математики. Среди них – проблемы полноты формальной арифметики и разрешимости формализованного исчисления предикатов первого порядка, десятая проблема Гильберта о разрешимости диофантовых уравнений, проблема тождества слов в полугруппах и группах, проблема действительных корней многочлена с действительными коэффициентами.

Теория алгоритмов развивалась в тесном взаимодействии с математической логикой. Это, вне всякого сомнения, обусловлено тесной взаимосвязью алгоритмического и логического мышления человека, схожестью алгоритмов с процессами построения логических умозаключений, использованием обеими дисциплинами формальных языков. Теория алгоритмов наряду с неотделимой от неё математической логикой являются фундаментальными теоретическими основаниями, на которых зиждутся теория и практика компьютеров, программирования и информатики. Ведь каждая компьютерная программа представляет собой выражение алгоритма решения задачи на одном из алгоритмических языков, которые тесно связаны с формализованными языками математической логики. Так что овладение основами теории алгоритмов является в настоящее время неотъемлемым компонентом образования всякого высококвалифицированного специалиста, имеющего дело с перечисленными областями.

Курс теории алгоритмов излагается вслед за курсом математической логики. В данном учебном пособии мы будем использовать следующие наши учебные пособия: *Игошин В.И.* Математическая логика. – М.: Издательство ИНФРА-М, 2011 (ссылка на него будет делаться следующим образом: Учебник МЛ) и *Игошин В.И.* Задачи и упражнения по математической логике и теории алгоритмов. – М.: Издательский центр "Академия", 2008 (на него будем ссылаться так: Задачник).

В конце книги приведён обширный список литературы, разделённый по темам.

г. Саратов,
1 июля 2011 г. *В.Игошин*

Глава I

НЕФОРМАЛЬНОЕ (ИНТУИТИВНОЕ) ПРЕДСТАВЛЕНИЕ ОБ АЛГОРИТМАХ

В этой главе рассматриваются примеры алгоритмов из реальной жизни и из математики, более детально обсуждается понятие алгоритма на интуитивной основе и устанавливается необходимость его уточнения.

1.1. Алгоритмы в жизни и в математике

В этом параграфе рассмотрим примеры алгоритмов, известные из практической жизни и из математики.

Алгоритмы в жизни. Понятие алгоритма стихийно формировалось с древнейших времён. Современный человек понимает под *алгоритмом* чёткую систему инструкций о выполнении в определённом порядке некоторых действий для решения всех задач какого-то данного класса.

Многочисленные и разнообразные алгоритмы окружают нас буквально во всех сферах жизни и деятельности. Многие наши действия доведены до бессознательного автоматизма, мы порой и не осознаём, что они регламентированы неким алгоритмом – чёткой системой инструкций. Например, наши действия при входе в магазин "Универсам" (сдать свою сумку, получить корзину с номером, пройти в торговый зал, заполнить корзину продуктами, оплатить покупку в кассе, предъявить чек контролёру, взять свою сумку, переложить в неё продукты, сдать корзину, покинуть магазин). Второй пример – приготовление манной каши (500 мл молока довести до кипения, при тщательном помешивании засыпать 100 г манной крупы, при помешивании довести до кипения и варить 10 минут). Автоматизм выполнения этих и многих других действий не позволяет нам осознавать их алгоритмическую сущность.

Но есть немало таких действий, выполняя которые мы тщательно следуем той или иной инструкции. Это главным образом непривычные действия, профессионально не свойственные нам. Например, если вы фотографируете один-два раза в год, то, купив проявитель для плёнки, будете весьма тщательно следовать инструкции (алгоритму) по его приготовлению: "Содержимое большого пакета растворить в 350 мл воды при температуре 18-20 С. Там же растворить содержимое малого пакета. Объём раствора довести до 500 мл. Раствор профильтровать. Проявлять 3-4 роликовых фотоплёнки". Второй пример. Если вы никогда раньше не пекли торт, то, получив рецепт (алгоритм) его приготовления, постараетесь выполнить в указанной последовательности все его предписания.

Алгоритмы в математике. Большое количество алгоритмов встречается при изучении математики буквально с первых классов школы. Это, прежде всего, алгоритмы выполнения четырёх арифметических действий над различными числами – натуральными, целыми, дробными, комплексными. Вот пример такого алгоритма: "Чтобы из одной десятичной дроби вычесть другую, надо: 1) уравнять число знаков после запятой в уменьшаемом и вычитаемом; 2) записать вычитаемое под уменьшаемым так, чтобы запятая оказалась под запятой; 3) произвести вычитание так, как вычитают натуральные числа; 4) поставить в полученной разности, запятую под запятыми в уменьшаемом и вычитаемом".

Вот пример алгоритма сложения приближенных чисел. Найти сумму чисел p, q и s, где $p \approx 3,1416$, $q \approx 2,718$ и $s \approx 7,45$.

1. Выделим слагаемое с наименьшим числом десятичных знаков. Таким слагаемым является число 7,45 (два десятичных знака).

2. Округлим остальные слагаемые, оставляя в них столько десятичных знаков, сколько их имеется в выделенном слагаемом: $3,1416 \approx 3,14$; $2,718 \approx 2,72$.

3. Выполним сложение приближённых значений чисел: $3,14 + 2,72 + 7,45 = 13,31$.

Итак, $p + q + s \approx 13,31$.

Немало алгоритмов в геометрии: алгоритмы геометрических построений с помощью циркуля и линейки (деление пополам отрезка и угла, опускание и восстановление перпендикуляров, проведение параллельных прямых), алгоритмы вычисления площадей и объёмов различных геометрических фигур и тел.

При изучении математики в вузе были освоены процедуры вычисления наибольшего общего делителя двух натуральных чисел

(алгоритм Евклида), определителей различных порядков, рангов матриц с рациональными элементами, интегралов от рациональных функций, приближённых значений корней уравнений и систем и т.д. Все эти процедуры являются ничем иным, как алгоритмами. Наконец в курсе математической логики были рассмотрены алгоритмы разрешимости формализованного исчисления высказываний (см. Учебник МЛ, §16) и разрешимости в логике предикатов (§23).

Алгоритм Евклида. Это – алгоритм для нахождения наибольшего общего делителя НОД(m, n) двух натуральных чисел m, n. Рассмотрим его поподробнее с точки зрения стоящей перед нами задачи – охарактеризовать общие свойства и черты интуитивно представляемого нами понятия алгоритма.

Даны натуральные числа $m > n > 0$.

1) *Первый шаг*: делим m на n:
$m = nb_1 + r_1.$ $\quad (n > r_1)$
Если $r_1 = 0$, то НОД$(m, n) = n$.
Если $r_1 \neq 0$, то НОД$(m, n) =$ НОД(n, r_1), $\quad$ (1)
и переходим ко второму шагу.

2) *Второй шаг*: делим n на r_1:
$n = r_1 \cdot b_2 + r_2.$ $\quad (r_1 > r_2)$
Если $r_2 = 0$, то НОД$(n, r_1) = r_1$.
Если $r_2 \neq 0$, то НОД$(n, r_1) =$ НОД(r_1, r_2), $\quad$ (2)
и переходим к третьему шагу.

3) *Третий шаг*: делим r_1 на r_2:
$r_1 = r_2 \cdot b_3 + r_3.$ $\quad (r_2 > r_3)$.
Если $r_3 = 0$, то НОД$(r_1, r_2) = r_2$.
Если $r_3 \neq 0$, то НОД$(r_1, r_2) =$ НОД(r_2, r_3), $\quad$ (3)
и переходим к следующему шагу: делим r_2 на r_3 и так далее:
....

$r_{k-2} = r_{k-1} \cdot b_k + r_k.$ $\quad (r_{k-1} > r_k)$,
$r_{k-1} = r_k \cdot b_{k+1} + r_{k+1}.$ $\quad (r_k > r_{k+1})$.

Таким образом, получаем убывающую последовательность натуральных чисел:

$$n > r_1 > r_2 > \ldots > r_{k-1} > r_k > r_{k+1} \geq 0 ,$$

которая обрывается (является конечной), то есть $r_{k+1} = 0$ для некоторого $k \in N$ (так как $n \in N$). Учитывая равенства (1), (2), (3), ... , получаем:

$$\text{НОД}(m,n) = \text{НОД}(n,r_1) = \text{НОД}(r_1,r_2) = \text{НОД}(r_2,r_3) = \ldots$$
$$\ldots = \text{НОД}(r_{k-2},r_{k-1}) = \text{НОД}(r_{k-1},r_k) = r_k \,.$$

Чёткость предписаний данного алгоритма не допускает ни малейшей двусмысленности.

Таким образом, мы видим, что алгоритмы широко распространены как в практике, так и в науке и требуют более внимательного к себе отношения и тщательного изучения методами математической науки.

1.2. Неформальное понятие алгоритма и необходимость его уточнения

Неформальное понятие алгоритма. Прежде чем перейти к математическому изучению понятия алгоритма, постараемся внимательно проанализировать рассмотренные примеры алгоритмов и выявить их общие типичные черты и особенности.

1) Каждый алгоритм предполагает наличие некоторых *начальных* или *исходных данных*, а в результате применения приводит к получению определённого искомого *результата*. Например, в алгоритме с проявителем начальные данные – содержимое большого и малого пакетов, вода. Искомый результат – готовый к употреблению проявитель для плёнки. При вычислении ранга матрицы начальными данными служит прямоугольная таблица, составленная из $m \cdot n$ рациональных чисел, результат – натуральное число, являющееся рангом данной матрицы.

Говоря о начальных данных для алгоритма, имеют в виду так называемые *допустимые начальные данные*, т.е. такие начальные данные, которые сформулированы в терминах данного алгоритма. Так, к числу допустимых начальных данных для алгоритма варки манной каши никак не отнесёшь элементы множества натуральных чисел, а к числу начальных данных алгоритма Евклида – молоко и манную крупу (или даже комплексные числа).

Среди допустимых начальных данных для алгоритма могут быть такие, к которым он применим, т.е. отправляясь от которых можно получить искомый результат, а могут быть такие, к которым данный алгоритм неприменим, т.е. отправляясь от которых искомого результата получить нельзя. Неприменимость алгоритма к допустимым начальным данным может заключаться либо в том, что

алгоритмический процесс никогда не оканчивается (в этом случае говорят, что он бесконечен), либо в том, что его выполнение во время одного из шагов наталкивается на препятствие, заходит в тупик (в этом случае говорят, что он безрезультатно обрывается). Проиллюстрируем на примерах оба случая.

Пример 1.2.1. Приведём пример бесконечного алгоритмического процесса. Всем известен алгоритм деления десятичных дробей. Числа 5,1 и 3 являются для него допустимыми начальными данными, применение к которым алгоритма деления приводит к искомому результату 1,7. Иная картина возникает для чисел 20 и 3, которые также представляют собой допустимые начальные данные для алгоритма деления. Но для них алгоритмический процесс продолжается бесконечно (вечно), порождая бесконечную периодическую дробь: 6,66666... . Таким образом, этот процесс не встречает препятствий и никогда не оканчивается, так что получить искомый результат для начальных данных 20 и 3 оказывается невозможно. Отметим, что обрыв процесса произвольным образом (и взятие приближённого результата) не предусматривается данным алгоритмом.

Пример 1.2.2. Теперь приведём пример алгоритма, заходящего в тупик, безрезультатно обрывающегося. Вот его предписания:

1. Исходное число умножить на 2. Перейти к выполнению п. 2;

2. К полученному числу прибавить 1. Перейти к выполнению п. 3;

3. Определить остаток y от деления полученной в п. 2 суммы на 3. Перейти к выполнению п. 4;

4, Разделить исходное число на y. Частное является искомым результатом. Конец.

Пусть натуральные (целые положительные) числа будут допустимыми начальными данными для этого алгоритма. Для числа 6 алгоритмический процесс будет проходить так:

Первый шаг: $6 \cdot 2 = 12$; переходим к п. 2;
Второй шаг: $12 + 1 = 13$; переходим к п. 3;
Третий шаг: $y = 1$, переходим к п. 4;
Четвёртый шаг: $6 : 1 = 6$. Конец.

Искомый результат равен 6. Иначе будет протекать алгоритмический процесс для исходного данного 7:

Первый шаг: $7 \cdot 2 = 14$; переходим к п. 2;
Второй шаг: $14 + 1 = 15$; переходим к п. 3;

Третий шаг: $y = 0$; переходим к п. 4;

Четвёртый шаг: $7 : 0$ – деление невозможно. Процесс зашёл в тупик, натолкнулся на препятствие и безрезультатно оборвался.

Наконец отметим ещё одно немаловажное качество начальных данных для алгоритма. Они должны быть конструктивными. *Конструктивность* объекта означает, что он весь целиком может быть представлен для его рассмотрения и работы с ним. Примеры: булевы функции; формулы алгебры высказываний; натуральные числа; рациональные числа; матрицы с натуральными и рациональными элементами; многочлены с рациональными коэффициентами. Неконструктивные объекты: действительные числа, представимые бесконечными непериодическими дробями $0,\alpha_1\alpha_1\alpha_1...$, в частности числа π и e.

2) Далее, применение каждого алгоритма осуществляется путём выполнения дискретной цепочки, последовательности неких элементарных действий. Эти действия называют шагами или тактами, а процесс их выполнения называют *алгоритмическим процессом*. Причём, начальными данными для каждого последующего шага являются данные, полученные в результате выполнения предыдущего шага. Шаги должны быть достаточно элементарными и заканчиваться результатом (то есть быть эффективными). Таким образом, отмечается свойство *дискретности алгоритма*, которое означает, что к результату мы приходим, выполняя определенные действия пошаговым, дискретным образом.

3) Непременным условием, которому удовлетворяет алгоритм, является его *детерминированность* или *определённость*. Это означает, что предписания алгоритма с равным успехом могут быть выполнены любым другим человеком и в любое другое время, причём результат получится тот же самый. Другими словами, предписания алгоритма настолько точны, отчётливы, что не допускают никаких двусмысленных толкований и никакого произвола со стороны исполнителя. Они единственным и вполне определённым путём всякий раз приводят к искомому результату. Это наводит на мысль, что выполнение тех или иных алгоритмов может быть поручено машине, что широко и делается на практике.

4) Существенной чертой алгоритма является его *массовый характер*, т.е. возможность применять его к обширному классу начальных данных, возможность достаточно широко эти начальные данные варьировать. Другими словами, каждый алгоритм призван

решить ту или иную *массовую проблему*, т.е. решать класс однотипных задач. Например, задача нахождения наибольшего общего делителя чисел 4 и 6 есть единичная проблема (можно решить её и без применения алгоритма Евклида), но задача нахождения наибольшего общего делителя произвольных натуральных чисел m и n – уже проблема массовая. Суть алгоритма Евклида состоит в том, что он приводит к желаемому результату вне зависимости от выбора конкретной пары натуральных чисел, в то время как при решении указанной единичной проблемы можно предложить такой способ, который окажется неприменимым для другой пары натуральных чисел.

Итак, подводя итоги обсуждению характерных свойств и особенностей алгоритма, можем сформулировать следующее интуитивно описательное определение этого понятия.

Под *алгоритмом* понимается чёткая система инструкций, определяющая дискретный детерминированный процесс, ведущий от варьируемых начальных данных (входов) к искомому результату (выходу), если таковой существует, через конечное число тактов работы алгоритма; если же искомого результата не существует, то вычислительный процесс либо никогда не оканчивается, либо попадает в тупик.

Отметим в заключение, что сам термин "алгоритм" (или "алгорифм") происходит от имени великого средне-азиатского учёного Мухаммеда аль-Хорезми (787 – ок. 850). В своём трактате, написанном по-арабски, латинская версия которого относится к XII веку и начинается словами "Dixit algorizm", то есть "Сказал аль-Хорезми", им среди прочего была описана индийская позиционная система чисел и сформулированы правила выполнения четырёх арифметических действий над числами в десятичной записи.

Необходимость уточнения понятия алгоритма. Понятие алгоритма формировалось с древнейших времён, но до конца первой трети XX века математики довольствовались интуитивным представлением об этом объекте. Термин алгоритм употреблялся в математике лишь в связи с теми или иными конкретными алгоритмами. Утверждение о существовании алгоритма для решения задач данного типа сопровождалось фактическим его описанием.

Парадоксы, обнаруженные в основаниях математики в начале XX века, вызвали к жизни различные концепции и течения, призванные эти парадоксы устранить. В 20-е годы вплотную встали во-

просы о том, что же такое строгая выводимость и эффективное вычисление. Понятие алгоритма само должно было стать объектом математического исследования и поэтому нуждалось в строгом определении. Кроме того, к этому вынуждало развитие физики и техники, быстро приближавшее начало века электронно-вычислительных машин.

Далее, в начале XX века у математиков начали возникать подозрения в том, что некоторые массовые задачи по-видимому не имеют алгоритмического решения. Для точного доказательства несуществования какого-то объекта необходимо иметь его точное математическое определение. Совершенно аналогичная ситуация сложилась в своё время в математике, когда назрела необходимость уточнения таких понятий, как непрерывность, кривая, поверхность, длина, площадь, объём и т.п.

Первые работы по уточнению понятия алгоритма и его изучению, т.е. по теории алгоритмов, были выполнены в 1936–1937 годах математиками А.Тьюрингом, Э.Постом, Ж.Эрбраном, К.Гёделем, А.А.Марковым, А.Чёрчем. Было выработано несколько определений понятия алгоритма, но впоследствии выяснилось, что все они равносильны между собой, т.е. определяют одно и то же понятие.

Глава II

МАШИНЫ ТЬЮРИНГА И ВЫЧИСЛИМЫЕ ПО ТЬЮРИНГУ ФУНКЦИИ

Введение понятия машины Тьюринга явилось одной из первых и весьма удачных попыток дать точный математический эквивалент для общего интуитивного представления об алгоритме. Это понятие названо по имени английского математика (Alan Turing), сформулировавшего его в 1937 году, за 9 лет до появления первой электронно-вычислительной машины.

2.1. Понятие машины Тьюринга и применение машин Тьюринга к словам

В этом параграфе вводится понятие машины Тьюринга и рассматриваются первые примеры, иллюстрирующие это понятие.

Определение машины Тьюринга. Машина Тьюринга есть математическая (воображаемая) машина, а не машина физическая. Она есть такой же математический объект, как функция, производная, интеграл, группа и т.д. И так же, как и другие математические понятия, понятие машины Тьюринга отражает объективную реальность, моделирует некие реальные процессы. Именно, Тьюринг предпринял попытку смоделировать действия математика (или другого человека), осуществляющего некую умственную созидательную деятельность. Такой человек, находясь в определённом "умонастроении" ("состоянии"), просматривает некоторый текст. Затем он вносит в этот текст какие-то изменения, проникается новым "умонастроением" и переходит к просмотру последующих записей.

Машина Тьюринга действует примерно так же. Её удобно представлять в виде автоматически работающего устройства. В каждый дискретный момент времени устройство, находясь в некотором состоянии, обозревает содержимое одной ячейки протягиваемой через устройство ленты и делает шаг, заключающийся в том, что устройство переходит в новое состояние, изменяет содержимое обозреваемой ячейки и переходит к обозрению следующей ячейки, справа или слева. Причём шаг осуществляется на основании предписанной команды. Совокупность всех команд представляет собой программу машины Тьюринга.

Опишем теперь машину Тьюринга Θ более тщательно. Машина Θ располагает конечным числом знаков (символов, букв), образующих так называемый ***внешний алфавит*** $A = \{a_0, a_1, ..., a_n\}$. В каждую ячейку обозреваемой ленты в каждый дискретный момент времени может быть записан только один символ из алфавита A. Ради единообразия удобно считать, что среди букв внешнего алфавита A имеется "пустая буква" и именно она записана в пустую ячейку ленты. Условимся, что "пустой буквой" или символом пустой ячейки является буква a_0. Лента предполагается неограниченной в обе стороны, но в каждый момент времени на ней записано конечное число непустых букв.

Далее, в каждый момент времени машина Θ способна находиться в одном ***состоянии*** из конечного числа внутренних состояний, совокупность которых $Q = \{q_0, q_1, ..., q_m\}$. Среди состояний выделяются два – ***начальное*** q_1 и ***заключительное***, или ***состояние остановки***, q_0. Находясь в состоянии q_1, машина начинает работать. Попав в состояние q_0, машина останавливается.

Работа машины Θ определяется ***программой*** (***функциональной схемой***). Программа состоит из команд. Каждая команда $T(i, j)$ $(i = l, 2, ..., m;\ j = 0, 1, ..., n)$ представляет собой выражение одного из следующих видов:

$$q_i a_j \rightarrow q_k a_l \text{ С}, \qquad q_i a_j \rightarrow q_k a_l \text{ П}, \qquad q_i a_j \rightarrow q_k a_l \text{ Л},$$

где $0 \leq k \leq m$, $0 \leq l \leq n$. В выражениях первого вида символ С будем часто опускать.

Как же работает машина Тьюринга? Находясь в какой-либо момент времени в незаключительном состоянии (т.е. в состоянии отличном от q_0), машина совершает шаг, который полностью определяется ее текущим состоянием q_i и символом a_j, воспринимаемым ею в данный момент на ленте. При этом содержание шага регла-

ментировано соответствующей командой $T(i, j)$: $q_i a_j \to q_k a_l X$, где $X \in$ {С, П, Л }. Шаг заключается в том, что:

1) содержимое a_j обозреваемой на ленте ячейки стирается, и на его место записывается символ a_l, (который может совпадать с a_j);

2) машина переходит в новое состояние q_k (оно также может совпадать с предыдущим состоянием q_i);

3) машина переходит к обозрению следующей правой ячейки от той, которая обозревалась только что, если $X =$ П, или к обозрению следующей левой ячейки, если $X =$ Л, или же продолжает обозревать ту же ячейку ленты, если $X =$ С.

$\boxed{q_i}$ – ***считывающая головка***

В следующий момент времени (если $q_k \neq q_0$) машина делает шаг, регламентированный командой $T(k, l)$: $q_k a_l \to q_r a_s X$, и т.д.

Поскольку работа машины по условию полностью определяется её состоянием q_i в данный момент и содержимым a_j обозреваемой в этот момент ячейки, то для каждых q_i и a_j, $(i = 1, 2, ..., m$; $j = 0, 1, ..., n)$ программа машины должна содержать одну и только одну команду, начинающуюся символами $q_i a_j$. Поэтому программа машины Тьюринга с внешним алфавитом $A = \{a_0, a_1, ..., a_n\}$ и алфавитом внутренних состояний $Q = \{q_0, q_1, ..., q_m\}$ содержит $m(n + 1)$ команд.

Словом в алфавите A или в алфавите Q, или в алфавите $A \cup Q$ называется любая последовательность букв соответствующего алфавита. Под ***k-ой конфигурацией*** будем понимать изображение ленты машины с информацией, сложившейся на ней к началу k-го шага (или слово в алфавите A, записанное на ленту к началу k-го шага), с указанием того, какая ячейка обозревается в этот шаг и в каком состоянии находится машина. Имеют смысл лишь конечные конфигурации, т.е. такие, в которых все ячейки ленты, за исключением, быть может, конечного числа, пусты. Конфигурация называется ***заключительной***, если состояние, в котором при этом находится машина, заключительное.

Если выбрать какую-либо незаключительную конфигурацию машины Тьюринга в качестве исходной, то работа машины будет состоять в том, чтобы последовательно, шаг за шагом, преобразовывать исходную конфигурацию в соответствии с программой машины до тех пор, пока не будет достигнута заключительная конфигурация.

После этого работа машины Тьюринга считается закончившейся, а результатом работы считается достигнутая заключительная конфигурация.

Будем говорить, что непустое слово α в алфавите $A \setminus \{a_0\} = \{a_1, ..., a_n\}$ воспринимается машиной *в стандартном положении*, если оно записано в последовательных ячейках ленты, все другие ячейки пусты, и машина обозревает крайнюю справа ячейку из тех, в которых записано слово α. Стандартное положение называется ***начальным*** (***заключительным***), если машина, воспринимающая слово в стандартном положении, находится в начальном состоянии q_1 (соответственно, в состоянии остановки qq_0). Наконец, будем говорить, что ***слово α перерабатывается машиной в слово β***, если от слова α, воспринимаемого в начальном стандартном положении, машина после выполнения конечного числа команд приходит к слову β, воспринимаемому в положении остановки.

Применение машин Тьюринга к словам. Проиллюстрируем на примерах все введённые понятия, связанные с машинами Тьюринга.

Пример 2.1.1. Дана машина Тьюринга с внешним алфавитом $A = \{0, 1\}$ (здесь 0 – символ пустой ячейки), алфавитом внутренних состояний $Q = \{q_0, q_1, q_2\}$ и со следующей функциональной схемой (программой):

$$q_1 0 \to q_2 0\text{П}, \quad q_2 0 \to q_0 1, \quad q_1 1 \to q_1 1\text{П}, \quad q_2 1 \to q_2 1\text{П}.$$

Посмотрим, в какое слово переработает эта машина слово 101, исходя из стандартного начального положения. Будем последовательно выписывать конфигурации машины при переработке ею этого слова. Имеем стандартное начальное положение:

					q_1				
(1)			1	0	1				

На первом шаге действует команда: $q_1 1 \to q_1 1\text{П}$. В результате на машине создается следующая конфигурация:

						q_1			
(2)			1	0	1	0			

На втором шаге действует команда: $q_1 0 \to q_2 0\text{П}$ и на машине создаётся конфигурация:

							q_2		
(3)			1	0	1	0	0		

Наконец, третий шаг обусловлен командой: $q_2 0 \to q_0 1$. В результате него создаётся конфигурация:

(4)

						q_2			
		1	0	1	0	1			

Эта конфигурация является заключительной, потому что машина оказалась в состоянии остановки q_0.

Таким образом, исходное слово 101 переработано машиной в слово 10101.

Полученную последовательность конфигураций можно записать более коротким способом. Конфигурация (1) записывается в виде следующего слова в алфавите $A \cup Q$: $10q_1 1$ (содержимое обозреваемой ячейки записано справа от состояния, в котором находится в данный момент машина). Далее, конфигурация (2) записывается так: $101q_1 0$, конфигурация (3) – $1010q_2 0$, и наконец, (4) – $1010q_0 1$. Вся последовательность записывается так:

$$10q_1 1 \implies 101q_1 0 \implies 1010q_2 0 \implies 1010q_0 1 .$$

Приведём последовательность конфигураций при переработке этой машиной слова 11011, исходя из начального положения, при котором в состоянии q_1 обозревается крайняя левая ячейка, в которой содержится символ этого слова (самостоятельно проанализируйте каждый шаг работы машины, указывая, какая команда обусловила его):

(1)

		q_1							
		1	1	0	1	1			

(2)

			q_1						
		1	1	0	1	1			

(3)

				q_1					
		1	1	0	1	1			

(4)

					q_2				
		1	1	0	1	1			

(5)

						q_2			
		1	1	0	1	1			

q_2

(6)		1	1	0	1	1	0		

q_0

(7)		1	1	0	1	1	1		

Более короткая запись этой последовательности конфигураций, т.е. процесса работы машины будет

$q_1 11011 \Longrightarrow 1q_1 1011 \Longrightarrow 11q_1 011 \Longrightarrow 110q_2 11 \Longrightarrow 1101q_2 1 \Longrightarrow$
$11011q_2 0 \Longrightarrow 11011q_0 1$.

Таким образом, слово 11011 переработано машиной в слово 110111.

Пример 2.1.2. Машина Тьюринга задаётся внешним алфавитом $A = \{0, 1, *\}$ (как и в предыдущем примере 0 – символ пустой ячейки), алфавитом внутренних состояний $Q = \{q_0, q_1, q_2, q_3\}$ и программой:

$q_1 0 \to q_1 0Л$, $q_2 0 \to q_3 10П$, $q_3 0 \to q_1 0Л$, $q_1 1 \to q_2 0Л$, $q_2 1 \to q_2 1Л$,
$q_3 1 \to q_3 1П$, $q_1 * \to q_0 0$, $q_2 * \to q_2 *Л$, $q_3 * \to q_3 *П$.

Посмотрим, как эта машина перерабатывает некоторые слова, и постараемся обнаружить закономерность в её работе.

Предварительно заметим, что программа машины может быть записана также в виде следующей таблицы:

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

Чтобы определить по таблице, что будет делать машина, находясь, например, в состоянии q_2 и наблюдая в обозреваемой ячейке символ 1, нужно найти в таблице клетку, находящуюся на пересечении столбца q_2 и строки 1. В этой клетке записано $q_2 1Л$. Это означает, что на следующем шаге машина останется в прежнем состоянии q_2, сохранит содержимое обозреваемой ячейки 1 и перейдёт к обозрению следующей левой ячейки на ленте.

Применим эту машину к слову $11 * 11$. Вот последовательность конфигураций, возникающих в процессе работы машины (исходная конфигурация – стандартная начальная):

$$11*1q_11 \Longrightarrow 11*q_210 \Longrightarrow 11q_2*10 \Longrightarrow 1q_21*10 \Longrightarrow q_211*10 \Longrightarrow$$
$$q_2011*10 \Longrightarrow q_311*10 \Longrightarrow 11q_31*10 \Longrightarrow 111q_3*10 \Longrightarrow$$
$$111*q_310 \Longrightarrow 111*1q_30 \Longrightarrow 111*q_110 \Longrightarrow 111q_2*00 \Longrightarrow$$
$$11q_21*0 \Longrightarrow 1q_211*0 \Longrightarrow q_2111*0 \Longrightarrow q_20111*0 \Longrightarrow$$
$$1q_3111*0 \Longrightarrow 11qq_311*0 \Longrightarrow 111q_31*0 \Longrightarrow 1111q_3*0 \Longrightarrow$$
$$1111*q_30 \Longrightarrow 1111q_1*0 \Longrightarrow 1111q_000\,.$$

Таким образом, слово $11*11$ переработано данной машиной в слово 1111. Предлагается проверить самостоятельно, что данная машина Тьюринга осуществляет следующие преобразования конфигураций:

$11*q_11 \Longrightarrow 111q_00$ (т.е. слово $11*1$ перерабатывается в слово 111);

$1*111q_11 \Longrightarrow 11111q_00$, (т.е. слово $1*1111$ перерабатывается в слово 11111);

$11*11q_11 \Longrightarrow 11111q_00$, (т.е. слово $11*111$ перерабатывается в слово 11111);

$11111*1q_11 \Longrightarrow 1111111qq_00$ (т.е. слово $11111*11$ перерабатывается в слово 1111111).

Нетрудно заметить, что данная машина Тьюринга реализует операцию сложения: в результате её работы на ленте записано подряд столько единиц, сколько их было всего записано по обе стороны от звёздочки перед началом работы машины.

Этот маленький опыт работы с машинами Тьюринга позволяет сделать некоторые выводы. Так тщательно описанное устройство этой машины (разбитая на ячейки лента, считывающая головка) по существу не имеет никакого значения. Машина Тьюринга – не что иное, как некоторое правило (алгоритм) для преобразования слов алфавита $A \cup Q$, т.е. конфигураций. Таким образом, для определения машины Тьюринга нужно задать её внешний и внутренний алфавиты, программу и указать, какие из символов обозначают пустую ячейку и заключительное состояние.

Конструирование машин Тьюринга. *Создание* (*синтез*) машин Тьюринга (т.е. написание соответствующих программ) является задачей значительно более сложной, нежели процесс применения данной машины к данным словам.

Пример 2.1.3. Попытаемся построить такую машину Тьюринга, которая из n записанных подряд единиц оставляла бы на ленте

$n-2$ единицы, также записанные подряд, если $n \geq 2$, и работала бы вечно, если $n=0$ или $n=1$.

В качестве внешнего алфавита возьмём двухэлементное множество $A = \{0, 1\}$. Количество необходимых внутренних состояний будет определено в процессе составления программы. Считаем, что машина начинает работать из стандартного начального положения, т.е. когда в состоянии q_1 обозревается крайняя правая единица из n записанных на ленте.

Начнём с того, что сотрём первую единицу, если она имеется, перейдём к обозрению следующей левой ячейки и сотрём там единицу, если она в этой ячейке записана. На каждом таком переходе машина должна переходить в новое внутреннее состояние, ибо в противном случае будут стёрты вообще все единицы, записанные подряд. Вот команды, осуществляющие описанные действия:

$$q_1 1 \to q_2 0\text{Л}, \qquad q_2 1 \to q_3 0\text{Л} .$$

Машина находится в состоянии q_3 и обозревает третью справа ячейку из тех, в которых записано данное слово (n единиц). Не меняя содержимого обозреваемой ячейки, машина должна остановится, т.е. перейти в заключительное состояние q_0, не зависимо от содержимого ячейки. Вот эти команды:

$$q_3 0 \to q_0 0, \qquad q_3 1 \to q_0 1 .$$

Теперь остаётся рассмотреть ситуации, когда на ленте записана всего одна единица или не записано ни одной. Если на ленте записана одна единица, то после первого шага (выполнив команду $q_1 1 \to q_2 0\text{Л}$) машина будет находиться в состоянии q_2 и будет обозревать вторую справа ячейку, в которой записан 0. По условию, в таком случае машина должна работать вечно. Это можно обеспечить, например, такой командой:

$$q_2 0 \to q_2 0\text{П} ,$$

выполняя которую шаг за шагом, машина будет двигаться по ленте неограниченно вправо (или протягивать ленту через считывающую головку справа налево). Наконец, если на ленте не записано ни одной единицы, то машина, по условию, также должна работать вечно. В этом случае в начальном состоянии q_1 обозревается ячейка с содержимым 0, и вечная работа машины обеспечивается следующей командой:

$$q_1 0 \to q_1 0\text{П} .$$

Запишем составленную программу (функциональную схему) построенной машины Тьюринга в виде таблицы:

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

В заключение отметим следующее. Созданная нами машина Тьюринга может применяться не только к словам в алфавите $A = \{0, 1\}$, представляющим собой записанные подряд n единиц ($n \geq 2$). Она применима и ко многим другим словам в этом алфавите, например (проверьте самостоятельно), к словам: 1011, 10011, 111011, 11011, 1100111, 1001111, 10111, 10110111, 10010111 и т. д. (исходя из стандартного начального положения). С другой стороны, построенная машина неприменима (т.е. при подаче этих слов на вход машины она работает вечно) не только к слову "1" или к слову, состоящему из одних нулей. Она неприменима и к следующим словам (проверьте самостоятельно): 101, 1001, 11101, 101101, 1100101101 и т.д.

2.2. Примеры машин Тьюринга

Рассмотрим ещё ряд примеров машин Тьюринга, которые окажутся полезными для нас в дальнейшем.

Пример 2.2.1. **S : Прибавление единицы.** Машина Тьюринга задаётся внешним алфавитом $A = \{0, 1\}$ (как и в предыдущих примерах 0 – символ пустой ячейки), алфавитом внутренних состояний $Q = \{q_0, q_1, q_2, q_3\}$ и программой, записываемой в табличном виде:

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

Пусть на ленте записано подряд n единиц: 011 ... 10 (нули слева и справа – пустые ячейки). Сокращённо данное слово будем записывать так: 01^n0. Посмотрим, в какое слово будет переработано этой машиной данное слово, начав работу с ближайшего левого нуля, то есть в какую конфигурацию будет переработана конфигурация: $q_1 01^n 0$?

$q_1 0 \rightarrow q_2 0$ П : $\quad 0 q_2 1 1 ... 10 \quad$ (n единиц)

$q_2 1 \;\to\; q_2 1\text{ П}:$ $\qquad 011...1q_2 0 \qquad$ (n шагов)

$q_2 0 \;\to\; q_3 1:$ $\qquad 011...1q_3 0 \qquad$ (n единиц)

$q_3 1 \;\to\; q_3 1\text{ Л}:$ $\qquad q_3 011...110 \qquad$ ($n+1$ шагов)

$q_3 0 \;\to\; q_0 0:$ $\qquad q_0 011...110 \qquad$ ($n+1$ единиц)

Таким образом, машина переработала слово $q_1 01^n 0$ в слово $q_0 01^{n+1}$, т.е. $\boxed{q_1 01^n 0 \implies q_0 01^{n+1}}$.

Пример 2.2.2. **А : Перенос нуля.** Эта машина осуществляет переработку: $\boxed{q_1 001^x 0 \implies q_0 01^x 00}$. Она задаётся внешним алфавитом $A = \{0, 1\}$ (как и в предыдущих примерах 0 – символ пустой ячейки), алфавитом внутренних состояний $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$ и программой, записываемой в виде последовательности команд:

$q_1 0 \to q_2 0\text{ П}$	$q_4 1 \to q_5 0$
$q_2 0 \to q_3 1$	$q_5 0 \to q_6 0\text{ Л}$
$q_3 1 \to q_3 1\text{ П}$	$q_6 1 \to q_6 1\text{ Л}$
$q_3 0 \to q_4 0\text{ Л}$	$q_6 0 \to q_0 0$

Выпишите справа от команд последовательность конфигураций, получающихся при выполнении соответствующих команд. (См. Задачник, № 12.24).

Пример 2.2.3. **Б⁻ : Левый сдвиг.** Эта машина осуществляет переработку: $\boxed{01^x q_1 0 \implies q_0 01^x 0}$. Она задаётся внешним алфавитом $A = \{0, 1\}$ (как и в предыдущих примерах 0 – символ пустой ячейки), алфавитом внутренних состояний $Q = \{q_0, q_1, q_2\}$ и программой:

$q_1 0 \;\to\; q_2 0\text{ Л}$

$q_2 1 \;\to\; q_2 1\text{ Л}$

$q_2 0 \;\to\; q_0 0\,.$

Выпишите справа от команд последовательность конфигураций, получающихся при выполнении соответствующих команд.

Пример 2.2.4. **$Б^+$: Правый сдвиг.** Эта машина осуществляет переработку: $\boxed{q_1 0 1^x 0 \implies 0 1^x q_0 0}$. Программа этой машины получается из программы предыдущей машины $Б^-$ заменой символа Л символом П.

Пример 2.2.5. **В : Транспозиция.** Эта машина осуществляет переработку: $\boxed{0 1^x q_1 0 1^y 0 \implies 0 1^y q_0 0 1^x 0}$. Попытаемся объяснить процесс составления (изобретения) программы этой машины Тьюринга.

Сначала слово $01^x q_1 01^y 0$ переводим в слово $01^x q 1^y 00$. При $y > 0$ это достигается следующей программой:

$q_1 0 \to q_2 0П$ $\qquad$ $01^x 0 q_2 1^y 0$

$q_2 1 \to q_2 1П$

$q_2 0 \to q_3 0$ $\qquad$ $01^x 01^y q_3 0$

$q_3 0 \to q_4 0Л$

$q_4 1 \to q_5 0$

$q_5 0 \to q_6 0Л$

$q_6 1 \to q_6 1Л$ $\qquad$ $01^x q_6 01^{y-1} 00$

$q_6 0 \to q_7 1$ $\qquad$ $01^x q_7 1^y 00$

(Обратите внимание на то, что первые три команды представляют собой программу машины $Б^+$). Чтобы получить слово $01^x q_7 1^y 00$ и при $y = 0$, добавляем команду:

$q_4 0 \to q_7 0$ $\qquad$ $01^x q_7 000.$

Теперь начинаем процесс переброски единиц из массива 1^x в промежуток между двумя последними нулями. Начнём с одной единицы. Это достигается следующей программой (считаем пока, что $x > 0$, $y > 0$):

$q_7 1 \to q_8 1Л$ $\qquad$ $01^{x-1} q_8 1 1^y 00$

$q_8 1 \to q_9 0$ $\qquad$ $01^{x-1} q_9 0 1^y 00$

$q_9 0 \to q_{10} 0П$ $\qquad$ $01^{x-1} 0 q_{10} 1^y 00$

$q_{10} 1 \to q_{10} 1П$ $\qquad$ $01^{x-1} 0 1^y q_{10} 00$

$q_{10} 0 \to q_{11} 1$ $\qquad$ $01^{x-1} 0 1^y q_{11} 10$

$q_{11} 1 \to q_{12} 1Л$ $\qquad$ $01^{x-1} 0 1^{y-1} q_{12} 110$

$q_{12}1 \to q_{13}0$ $\quad$ $01^{x-1}01^{y-1}q_{13}010$

$q_{13}0 \to q_{14}0Л$ $\quad$ $01^{x-1}01^{y-2}q_{14}1010$

$q_{14}1 \to q_{14}1Л$ $\quad$ $01^{x-1}q_{14}01^{y-1}010$

$q_{14}0 \to q_{15}1$ $\quad$ $01^{x-1}q_{15}1^{y}010$

Чтобы достичь тот же результат и при $y = 0$, добавляем к записанной программе следующие команды:

$q_{7}0 \to q_{16}1$ $\quad$ $01^{x}q_{16}10$

$q_{16}1 \to q_{17}1Л$ $\quad$ $01^{x-1}q_{17}110$

$q_{17}1 \to q_{15}0$ $\quad$ $01^{x-1}q_{15}010.$

Теперь мы должны переносить следующую единицу из оставшегося массива 1^{x-1} вправо и также вставлять её между двумя нулями. Для этого мы "зацикливаем" программу следующими командами:

$q_{15}1 \to q_{7}1$

$q_{15}0 \to q_{7}0$ $\quad$ $01^{x-1}q_{7}1^{y}010.$

Если $x - 1 > 0$, то на следующем круге слово $01^{x-1}q_{7}1^{y}010$ перерабатывается в слово $01^{x-2}q_{7}1^{y}0110$. Если $x - 2 > 0$, то получим слово $01^{x-3}q_{7}1^{y}01110$ и т.д. Через x циклов получим слово $0q_{7}1^{y}01^{x}0$. Далее, если $y > 0$, то это слово перейдёт в слово $q_{8}01^{y}01^{x}0$; если $y = 0$, то оно перейдёт в слово $q_{17}011^{x}0$. Чтобы получить требуемое, остаётся добавить следующие команды:

$q_{8}0 \to q_{18}0П$

$q_{18}1 \to q_{18}1П$

$q_{18}0 \to q_{0}0$ $\quad$ $01^{y}q_{0}01^{x}0 \ (y > 0)$

$q_{17}0 \to q_{19}0П$

$q_{19}1 \to q_{0}0$ $\quad$ $0q_{0}01^{x}0 \ (y = 0).$

Пример 2.2.6. **Г : Удвоение.** Эта машина осуществляет переработку: $\boxed{q_{1}01^{x}0^{x+3} \Longrightarrow q_{0}01^{x}01^{x}00}$. Попытаемся и здесь объяснить процесс составления программы машины Тьюринга. Начнём с команд:

$q_{1}0 \to q_{2}0П$

$q_{2}1 \to q_{2}1П$ $\quad$ $01^{x}q_{2}0$

Представим полученное слово $01^x q_2 0$ в виде $01^x q_2 00^0 00^0 0^{x+1}$ и далее пишем программу, которая при $x - i > 0$ переводит слово $01^{x-i} q_2 01^i 01^i 0$ в слово $01^{x-(i+1)} q 01^{i+1} 01^{i+1}$. Вот она:

$q_2 0 \to q_3 0Л$ $\qquad 01^{x-(i+1)} q_3 101^i 01^i 0$

$q_3 1 \to q_4 0$

$q_4 0 \to q_5 0П$

$q_5 0 \to q_6 1$

$q_6 1 \to q_6 1П$ $\qquad 01^{x-(i+1)} 01^{i+1} q_6 01^i 0$

$q_6 0 \to q_7 0П$

$q_7 1 \to q_7 1П$

$q_7 0 \to q_8 1$

$q_8 1 \to q_8 1Л$

$q_8 0 \to q_9 0Л$

$q_9 1 \to q_9 1Л$ $\qquad 01^{x-(i+1)} q_9 01^{i+1} 01^{i+1}$.

Зацикливаем данную процедуру с помощью команды:

$q_9 0 \to q_2 0$ $\qquad 01^{x-(i+1)} q_2 01i + 101^{i+1}$.

На следующем цикле мы получим слово $01^{x-(i+2)} q_2 01^{i+2} 01^{i+2}$ и т.д., вплоть до слова $0 q_2 01^x 01^x 0$. Команда $q_2 0 \to q_3 0Л$ создаст конфигурацию $q_3 001^x 01^x 0$. Произведя теперь перенос нуля **А**, затем правый сдвиг **Б**$^+$, ещё раз перенос нуля **А** и, наконец, левый сдвиг **Б**$^-$, получим требуемое слово: $q_0 01^x 01^x 00$.

2.3. Композиция машин Тьюринга

Понятие композиции (суперпозиции, последовательного применения) тех или иных действий широко используется в самых разнообразных областях математики. Хорошо известны понятия суперпозиции функций (сложная функция) в математическом анализе, композиции (произведения) геометрических преобразований в геометрии.

Понятие композиции машин Тьюринга. ОПРЕДЕЛЕНИЕ 2.3.1. Пусть заданы машины Тьюринга Θ_1 и Θ_2, имеющие общий внешний алфавит $A = \{a_0, a_1, ..., a_m\}$ и алфавиты внутренних состояний $Q_1 = \{q_0, q_1, ..., q_n\}$ и $Q_2 = \{q_0, q'_1, ..., q'_t\}$ соответственно. *Композицией* (или *произведением*) машины Θ_1 на машину Θ_2

называется новая машина Θ (которая обозначается $\Theta_1 \cdot \Theta_2$) с тем же внешним алфавитом $A = \{a_0, a_1, ..., a_m\}$, алфавитом внутренних состояний $Q = \{q_0, q_1, ..., q_n, q_{n+1}, ..., q_{n+t}\}$ и программой, получающейся следующим образом. Во всех командах из Θ_1, содержащих символ остановки q_0, заменяем последний на q_{n+1}. Все остальные символы в командах из Θ_1 остаются неизменными. В командах из Θ_2 символ q_0 оставляем неизменным, а все остальные состояния q'_i $(i = 1, ..., t)$ заменяем соответственно на q_{n+i}. Совокупность всех так полученных команд образует программу машины-композиции $\Theta = \Theta_1 \cdot \Theta_2$.

Таким образом, работа машины $\Theta = \Theta_1 \cdot \Theta_2$ состоит в следующем. Сначала работает машина Θ_1; в момент её остановки начинает работу машина Θ_2, которая своей остановкой заканчивает работу машины $\Theta = \Theta_1 \cdot \Theta_2$.

Применение композиции машин Тьюринга для их конструирования. Введённая операция играет важную роль во всех вопросах, связанных с синтезом машин Тьюринга, т.е. является удобным инструментом для конструирования машин Тьюринга. Покажем это на примерах.

ПРИМЕР 2.3.2. **Ц** : **<u>Циклический сдвиг.</u>** Эта машина осуществляет переработку: $\boxed{01^x01^yq_101^z0 \implies 01^zq_001^x01^y0}$.

Эту машину можно составить (синтезировать) в виде композиции ранее построенных машин Тьюринга. Проверим, что такой перевод произойдёт в результате последовательного применения (композиции) ранее построенных машин **В**, **Б**$^-$ и **В**, а именно

$$\boxed{\mathbf{Ц}_3 = \mathbf{ВБ}^- \mathbf{В}} .$$

В самом деле, вычисляем: $\mathbf{ВБ}^-\mathbf{В}(01^x01^yq_101^z0) =$

$= \mathbf{ВБ}^-(\mathbf{В}(01^x01^yq_101^z0)) = \mathbf{ВВ}^-(01^x01^zq01^y0) =$

$= \mathbf{В}(\mathbf{Б}^-(01^x01^zq01^y0)) = \mathbf{В}(01^xq01^z01^y0) = 01^zq_001^x01^y0$.

ПРИМЕР 2.3.3. **<u>К$_2$</u>** : **<u>Копирование.</u>** Эта машина осуществляет переработку: $\boxed{q_101^x01^y \implies 01^x01^yq_001^x01^y}$.

Проверьте, что эта машина представляет собой следующую композицию построенных выше машин:

$$\mathbf{К}_2 = \mathbf{Б}^+ \, \mathbf{ГВБ}^+ \, \mathbf{ВГВБ}^+ \, \mathbf{ВБ}^- \, \mathbf{Б}^- \, \mathbf{ВБ}^+.$$

Обратите только внимание на то, что применять машины нужно в последовательности слева направо: **Б**$^+$, **Г** , **В** , **Б**$^+$ и т.д.

2.4. Вычислимые по Тьюрингу функции

Понятие функции – одно из фундаментальнейших понятий, выработанных математикой на протяжении многовековой своей истории и возникающее практически во всех её областях. Тесно связано оно и с алгоритмами и машинами Тьюринга.

Алгоритмы, функции и машины Тьюринга. Всякая машина Тьюринга (или, что тоже самое, её программа) задает своего рода "инструкцию" для осуществления некоторого алгоритма. Говорят, что данная машина Тьюринга реализует этот алгоритм.

В свою очередь, любой алгоритм можно рассматривать как алгоритм переработки слов, если закодировать участвующие в нём объекты с помощью слов подходящего алфавита.

Наконец, слова во всяком конечном алфавите можно взаимно однозначным образом занумеровать натуральными числами, и тогда алгоритм переработки слов (т.е. процесс реализации данного алгоритма) будет сведён к вычислению некоторой числовой функции, т.е. функции типа $f:\ N^n \to N$.

Нумерация может быть осуществлена, например, следующим образом. Сначала нумеруются все буквы данного алфавита $A = \{a_0, a_1, a_2, ..., a_n\}$. Присвоим каждой букве алфавита *номер* в виде соответствующего по порядку простого числа. Обозначим: Nom a_i – номер буквы a_i. Тогда Nom $a_i = p_i$. В частности:

$$\text{Nom } a_0 = p_0 = 2, \qquad \text{Nom } a_1 = p_1 = 3, \qquad \text{Nom } a_2 = p_2 = 5, \ \ldots$$

После этого можем занумеровать слова; а именно, слову $\alpha = a_{i_1} a_{i_2} a_{i_3} ... a_{i_k}$ припишем номер

$$\text{Nom } \alpha = 2^{Nom\ a_{i_1}} \cdot 3^{Nom\ a_{i_2}} \cdot 5^{Nom\ a_{i_3}} \cdot \ldots \cdot p_{k-1}^{Nom\ a_{i_k}},$$

где p_{k-1} – k-ое по порядку простое число. Ясно, что по данному номеру занумерованное им слово восстанавливается однозначно. Нумерация такого типа называется *гёделевской* (по имени австрийского логика XX века К.Гёделя).

Ясно, что по исходному алгоритму переработки слов можно построить алгоритм для вычисления соответствующей функции; в этом случае такие функции называются *алгоритмически* (или *эффективно*) *вычислимыми*. Возникает вопрос о соотношении между классом алгоритмически вычислимых функций и классом функций, вычислимых на машинах Тьюринга. В частности, совпадают ли эти классы?

Займёмся более подробным изучением понятия функции, вычислимой на машине Тьюринга.

Вычислимость функций на машине Тьюринга. ОПРЕДЕЛЕНИЕ 2.4.1. Функция называется ***вычислимой по Тьюрингу***, или ***вычислимой на машине Тьюринга***, если существует машина Тьюринга, вычисляющая её, т.е. такая машина Тьюринга, которая вычисляет её значения для тех наборов значений аргументов, для которых функция определена, и работающая вечно, если функция для данного набора значений аргументов не определена.

Остаётся договориться о некоторых условностях для того, чтобы это определение стало до конца точным. Во-первых, напомним, что речь идёт о функциях (или возможно о частичных функциях, т.е. не всюду определённых), заданных на множестве натуральных чисел и принимающих также натуральные значения. Во-вторых, нужно условиться, как записывать на ленте машины Тьюринга значения $x_1, x_2, ..., x_n$ аргументов функции $f(x_1, x_2, ..., x_n)$, из какого положения начинать переработку исходного слова и, наконец, в каком положении получать значение функции. Это можно делать, например, следующим образом. Значения $x_1, x_2, ..., x_n$ аргументов будем располагать на ленте в виде следующего слова:

$$0\ \underbrace{1\,1\ldots 1}_{x_1}\ 0\ \underbrace{1\,1\ldots 1}_{x_2}\ 0\ldots 0\ \underbrace{1\,1\ldots 1}_{x_n}\ 0\ ,$$

где в первом массиве число единиц равно x_1, во втором – x_2, и т.д., в последнем – x_n.

Здесь полезно ввести следующие обозначения. Для натурального x обозначаем: $1^x = 1...1$, $0^x = 0...0$. Дополнительно полагаем $0^0 = 1^0 = \Lambda$ – пустое слово. Так что на слова $1^0 = \Lambda$, $1^1 = 1$, $1^2 = 11$, $1^3 = 111, ...$ будем смотреть как на "изображения" натуральных чисел 0,1,2,3,... соответственно. Таким образом, предыдущее слово можно представить следующим образом: $01^{x_1}01^{x_2}...01^{x_n}0$.

Далее, начинать переработку данного слова будем из ***стандартного начального положения***, т.е. из положения, при котором в состоянии q_1 обозревается крайняя правая единица записанного слова. Если функция $f(x_1, x_2, ..., x_n)$ определена на данном наборе значений аргументов, то в результате на ленте должно быть записано подряд $f(x_1, x_2, ..., x_n)$ единиц; в противном случае машина должна работать бесконечно. При выполнении всех перечисленных условий будем говорить, что ***машина Тьюринга вычисляет данную функцию***. Таким образом, сформулированное определение 2.4.1 становится абсолютно строгим.

Обратимся к примерам. Нетрудно понять, что машина Тьюринга из примера 2.1.3 по существу вычисляет функцию $f(x) = x - 2$, а из примера 2.2.1 вычисляет функцию $S(x) = x + 1$ (прибавление единицы).

Пример 2.4.2. Построим машину Тьюринга, вычисляющую функцию $f(x) = x/2$. Эта функция не всюду определена: областью её определения является лишь множество всех чётных чисел. Поэтому, учитывая определение 2.4.1, нужно сконструировать такую машину Тьюринга, которая при подаче на её вход чётного числа давала бы на выходе половину этого числа, а при подаче нечётного – работала бы неограниченно долго.

Сконструировать машину Тьюринга – значит написать (составить) её программу. В этом процессе два этапа: сначала *создать алгоритм вычисления* значений функции, а затем *записать его на языке машины Тьюринга* (*запрограммировать*).

В качестве внешнего алфавита возьмём двухэлементное множество $A = \{0, 1\}$. В этом алфавите натуральное число x изображается словом 11...1, состоящим из x единиц, которое на ленте машины Тьюринга записывается в виде x единиц, стоящих в ячейках подряд. Работа машины начинается из стандартного начального положения: $01...1q_1 10$ (число единиц равно x).

Сделаем начало вычислительного процесса таким: машина обозревает ячейки, двигаясь справа налево, и каждую вторую единицу превращает в 0. Если для последней единицы второй единицы не найдётся, то машина продолжит движение по ленте влево неограниченно, т.е. будет работать бесконечно. Такое начало обеспечивается следующими командами:

(1) $q_1 1 \rightarrow q_2 1$Л,

(2) $q_2 1 \rightarrow q_1 0$Л,

(3) $q_2 0 \rightarrow q_2 0$Л.

Если же число x единиц чётно, то в результате выполнения команд создаётся конфигурация $q_1 0010101...01010$, в которой число единиц равно $x/2$. Остаётся сдвинуть единицы так, чтобы между ними не стояли нули. Для осуществления этой процедуры предлагается следующий алгоритм. Будем двигаться по ленте вправо, ничего на ней не меняя, до первой единицы и перейдём за единицу. Передвижение осуществляется с помощью следующих команд:

(4) $q_1 0 \rightarrow q_3 0$П,

(5) $q_3 0 \rightarrow q_3 0П$,

(6) $q_3 1 \rightarrow q_4 1П$.

В результате их выполнения получим конфигурацию:

$$001q_4 010101...010100 \ . \qquad (*)$$

Заменим 0, перед которым остановились, на 1 и продвинемся вправо до ближайшего 0:

(7) $q_4 0 \rightarrow q_5 1П$,

(8) $q_5 1 \rightarrow q_5 1П$.

Получим конфигурацию $00111q_5 0101...010100$, в которой правее обозреваемой ячейки записаны "пары" 01, ..., 01. Кроме того, на ленте одна единица записана лишняя. Продвинемся по ленте вправо до последней "пары" 01. Это можно сделать с помощью своеобразного цикла:

(9) $q_5 0 \rightarrow q_6 0П$,

(10) $q_6 1 \rightarrow q_5 1П$.

Получим конфигурацию $001110101...01010q_6 00$. Двигаться дальше вправо бессмысленно. Вернёмся на две ячейки назад и заменим единицу из последней "пары" 01 на ноль:

(11) $q_6 0 \rightarrow q_7 0Л$,

(12) $q_7 0 \rightarrow q_7 0Л$,

(13) $q_7 1 \rightarrow q_8 0Л$.

Получим конфигурацию $001110101...01q_8 00$. Число единиц на ленте снова равно $x/2$. Продвинемся влево на одну ячейку с помощью команды

(14) $q_8 0 \rightarrow q_9 0Л$,

в результате чего получим конфигурацию $001110101...010q_9 100$. Теперь уничтожим самую правую единицу и продвинемся по ленте влево до следующей единицы:

(15) $q_9 1 \rightarrow q_{10} 0Л$,

(16) $q_{10} 0 \rightarrow q_{10} 0Л$.

Получим конфигурацию: $001110101 \ ... \ 0q_{10} 100$, $\qquad (**)$

в которой левее обозреваемой ячейки записана серия пар 10, 10, ... , 10 (если читать справа налево). Теперь на ленте не достаёт одной единицы, т.е. число единиц равно $(x/2)-1$. Продвинемся по ленте влево до последней "пары" 10. Это можно сделать с помощью цикла

(17) $q_{10}1 \rightarrow q_{11}1Л$,

(18) $q_{11}0 \rightarrow q_{10}0Л$,

выполнив который придём к следующей конфигурации: $001q_{11}110101...0100$. Вернёмся вправо к ближайшему нулю и превратим его в единицу:

(19) $q_{11}1 \rightarrow q_{12}1П$,

(20) $q_{12}1 \rightarrow q_{12}1П$,

(21) $q_{12}0 \rightarrow q_{13}1П$.

Получим конфигурацию $001111q_{13}101...0100$, в которой число единиц снова равно $x/2$.

Если теперь перешагнём вправо по ленте через обозреваемую единицу и переведём машину в состояние q_4 с помощью команды

(22) $q_{13}1 \rightarrow q_4 1П$,

то придём к следующей конфигурации: $0011111q_4 01...0100$, которая по существу аналогична конфигурации (*). В результате программа зацикливается (т.е. становится циклической): снова ближайший 0 превращается в 1, а самая правая 1 – в 0, затем машина возвращается к самому левому нулю, оказываясь в начале следующего цикла, и т. д.

Как же завершается работа программы? В некоторый момент конфигурация будет иметь вид $00111...1110q_{10}100$. Выполнив команды (17), (18), придём к конфигурации $00111...1q_{11}110100$. Далее выполняются команды (19), (20), (21), что приводит к конфигурации: $00111...111111q_{13}00$. Остаётся остановить машину. Это делается с помощью команды

(23) $q_{13}0 \rightarrow q_0 0Л$.

Заключительная конфигурация имеет вид: $00111...1111q_0 100$.

Запишем программу машины Тьюринга в табличной форме:

Q A	0	1
q_1	$q_3$0П	$q_2$1Л
q_2	$q_2$0Л	$q_1$0Л
q_3	$q_3$0П	$q_4$1П
q_4	$q_5$1П	
q_5	$q_6$0П	$q_5$1П
q_6	$q_7$0Л	$q_5$1П
q_7	$q_7$0Л	$q_8$0Л
q_8	$q_9$0Л	$q_2$1Л
q_9		q_{10}0Л
q_{10}	q_{10}0Л	q_{11}1Л
q_{11}	q_{10}0Л	q_{12}1П
q_{12}	q_{13}1П	q_{13}1П
q_{13}	$q_0$0Л	$q_4$1П

Предлагается самостоятельно проследить за работой этой машины Тьюринга, взяв в качестве исходных конкретные слова: 111, 1111, 111111, 1111111111 .

Правильная вычислимость функций на машине Тьюринга. В предыдущем пункте мы рассмотрели вопрос о том, что значит и каким образом "*данная машина Тьюринга вычисляет функцию* $f(x_1, x_2, ..., x_n)$". Для этого нужно, чтобы каждое из чисел $x_1, x_2, ..., x_n$ было записано на ленту машины непрерывным массивом из соответствующего числа единиц, массивы были разделены символом 0. Если функция $f(x_1, x_2, ..., x_n)$ определена на данном наборе значений аргументов, то в результате на ленте должно быть записано подряд $f(x_1, x_2, ..., x_n)$ единиц. При этом, мы не очень строго относились к тому, в каком начальном положении машина начинает работать (часто это было стандартное начальное положение), в каком завершает работу и как эта работа протекает.

В дальнейшем нам понадобится более сильное понятие вычислимости функции на машине Тьюринга – понятие правильной вычислимости.

ОПРЕДЕЛЕНИЕ 2.4.3. Будем говорить, что машина Тьюринга *правильно вычисляет* функцию $f(x_1, x_2, ...x_n)$, если начальное слово $q_1 01^{x_1} 01^{x_2} ... 01^{x_n} 0$ она переводит в слово $q_0 01^{f(x_1, x_2, ..., x_n)} 0...0$ и при этом в процессе работы не пристраивает к начальному слову новых ячеек на ленте ни слева, ни справа. Если же функция f не определена на данном наборе значений аргументов, то, начав работать из указанного положения, она никогда в процессе работы не будет надстраивать ленту слева.

Проанализируйте работу машины Тьюринга из примера 2.2.1 и убедитесь, что она правильно вычисляет функцию $S(x) = x + 1$.

Пример 2.4.4. **О : Нуль-функция.** Приведём программу машины Тьюринга, правильно вычисляющей функцию $O(x) = 0$. Она осуществляет перевод: $\boxed{q_1 01^x 0 \implies q_0 00^{x+1})}$. Её программа:

$q_1 0 \to q_2 0$ П, $q_2 1 \to q_2 1$ П, $q_2 0 \to q_3 0$ Л, $q_3 1 \to q_4 0$, $q_4 0 \to q_3 0$ Л, $q_3 0 \to q_0 0$.

Пример 2.4.5. I^n_m : **Функции-проекторы:**

$$\boxed{I^n_m(x_1, x_2, ..., x_n) = x_m} \qquad (1 \le m \le n) .$$

Сконструируем машины Тьюринга, правильно вычисляющие функции-проекторы, используя введённую в параграфе 2.3 операцию композиции машин Тьюринга.

Рассмотрим сначала конкретный случай $n = 3$, $m = 2$, т.е. функцию $I^3_2(x_1, x_2, x_3) = x_2$. Мы должны переработать слово $q_1 01^{x_1} 01^{x_2} 01^{x_3} 0$ в слово $q_0 01^{x_2} 0$. Будем применять к начальной конфигурации последовательно сконструированные ранее машины Тьюринга **Б**$^+$, **В**, **Б**$^-$, **О** :

	$q_1 01^{x_1} 01^{x_2} 01^{x_3} 0$
Б$^+$:	$01^{x_1} q 01^{x_2} 01^{x_3} 0$
В :	$01^{x_2} q 01^{x_1} 01^{x_3} 0$
Б$^+$:	$01^{x_2} 01^{x_1} q 01^{x_3} 0$
О :	$01^{x_2} 01^{x_1} q 00^{x_3} 0$
Б$^-$:	$01^{x_2} q 01^{x_1} 00^{x_3} 0$
О :	$01^{x_2} q 00^{x_1} 00^{x_3} 0$
Б$^-$:	$q 01^{x_2} 00^{x_1} 00^{x_3} 0$

Таким образом, функция $I^3_2(x_1, x_2, x_3) = x_2$ вычисляется следующей композицией машин:

$$\mathbf{Б^+ВБ^+ОБ^-ОБ^-} = \mathbf{Б^+ВБ^+(ОБ^-)^2} .$$

Проверьте самостоятельно, что функция $I^2_2(x_1, x_2) = x_2$ вычисляется композицией **Б**$^+$**ВОБ**$^-$.

Теперь мы можем представить себе алгоритм построения композиции машин **Б**$^+$, **В**, **Б**$^-$, **О** для вычисления любой функции вида

$I^n_m(x_1, x_2, ..., x_n) = x_m$. С помощью правого сдвига $\mathbf{Б}^+$, применив его $m-1$ раз, нужно сначала достичь массива 01^{x_m}:

$$(\mathbf{Б}^+)^{m-1}: \qquad 01^{x_1}0...q01^{x_m}0...01^{x_n}0 .$$

Затем, двигаясь влево, транспонировать (с помощью **В**) массив 01^{x_m} с каждым соседним слева массивом, пока массив 01^{x_m} не выйдет на первое место:

$$(\mathbf{В} \cdot \mathbf{Б}^-)^{m-1}: \qquad q01^{x_m}01^{x_1}0...01^{x_{m-1}}01^{x_{m+1}}0...01^{x_n}0.$$

Теперь нужно дойти до крайнего правого массива с помощью $(n-1)$-кратного применения правого сдвига $\mathbf{Б}^+$:

$$(\mathbf{Б}^+)^{n-1}: \qquad 01^{x_m}01^{x_1}0...01^{x_{m-1}}01^{x_{m+1}}0...q01^{x_n}0.$$

Наконец нужно стирать последовательно справа налево все массивы единиц, кроме первого:

$$(\mathbf{ОБ}^-)^{n-1}: \qquad q01^{x_m}00^{x_1}0...00^{x_{m-1}}00^{x_{m+1}}0...00^{x_n}0.$$

Итак, данную функцию (правильно) вычисляет следующая машина Тьюринга:

$$(\mathbf{Б}^+)^{m-1}(\mathbf{В} \cdot \mathbf{Б}^-)^{m-1}(\mathbf{Б}^+)^{n-1} \ (\mathbf{О} \cdot \mathbf{Б}^-)^{n-1}.$$

При $n = 3$, $m = 2$ эта машина имеет вид:

$$\mathbf{Б}^+\mathbf{ВБ}^-(\mathbf{Б}^+)^2(\mathbf{ОБ}^-)^2 = \mathbf{Б}^+\mathbf{ВБ}^+(\mathbf{ОБ}^-)^2,$$

т.е. совпадает с построенной выше машиной. При $n = 2$, $m = 2$ эта машина имеет вид:

$$\mathbf{Б}^+(\mathbf{ВБ}^-)\mathbf{Б}^+(\mathbf{ОБ}^-) = \mathbf{Б}^+\mathbf{ВОБ}^-,$$

т.е. также совпадает с соответствующей рассмотренной выше машиной Тьюринга.

Вычисление сложных функций на машинах Тьюринга. Понятие суперпозиции функций и сложной функции хорошо известно, но нас сейчас эта процедура будет интересовать с точки зрения её взаимоотношений с процессом вычислимости функций с помощью машин Тьюринга. Напомним сначала определение.

ОПРЕДЕЛЕНИЕ 2.4.6. (*Оператор суперпозиции.*) Будем говорить, что n-местная функция φ получена из m-местной функции f и n-местных функций $g_1, ..., g_m$ в результате суперпозиции или с помощью ***оператора суперпозиции***, если для всех $x_1, ..., x_n$ справедливо равенство:

$$\varphi(x_1, ..., x_n) = f(g_1(x_1, ..., x_n), ..., g_m(x_1, ..., x_n)) .$$

(При этом, функция $\varphi(x_1, ..., x_n)$ называется *сложной* функцией.)

Теорема 2.4.7. ***Если функции*** $f(x_1, ..., x_m)$, $g_1(x_1, ..., x_n)$, ... , $g_m(x_1, ..., x_n)$ ***правильно вычислимы по Тьюрингу, то правильно вычислима и сложная функция (суперпозиция функций)***:

$$\varphi(x_1, ..., x_n) = f(g_1(x_1, ..., x_n), ..., g_m(x_1, ..., x_n)).$$

Доказательство. Руководствуясь определением 2.3.1 композиции машин Тьюринга, нетрудно понять, что если машина F правильно вычисляет функцию $f(y)$, а машина G правильно вычисляет функцию $g(x)$, то композиция этих машин $F \cdot G$ правильно вычисляет суперпозицию этих функций $f(g(x))$:

$$q_1 01^x;$$

G: $\quad q01^{g(x)};$

F: $\quad q01^{f(g(x))}.$

Рассмотрим более сложную суперпозицию вычислимых функций:
$\varphi(x, y) = f(g_1(x, y), g_2(x, y))$. Пусть машины F, G_1, G_2 правильно вычисляют функции f, g_1, g_2 соответственно. Сконструируем машину Тьюринга, правильно вычисляющую сложную функцию $\varphi(x, y)$, пользуясь введёнными в §2.2 машинами сдвига, транспозиции, удвоения и нулевой функции:

$$q_1 01^x 01^y$$

$\mathbf{K}_2$: $\quad 01^x 01^y q01^x 01^y$

G_1: $\quad 01^x 01^y q01^{g_1(x,y)}$

$\mathbf{Ц}$: $\quad 01^{g_1(x,y)} q01^x 01^y$

G_2: $\quad 01^{g_1(x,y)} q01^{g_2(x,y)}$

$\mathbf{Б}^-$: $\quad q01^{g_1(x,y)} 01^{g_2(x,y)}$

F: $\quad q01^{f(g_1(x,y), g_2(x,y))}$

$q0 \to q_0 0$: $\quad q_0 01^{f(g_1, g_2)}.$

Сконструируйте самостоятельно композицию машин, правильно вычисляющих функцию $\varphi(x, y) = f(g_1(x, y), g_2(x, y), g_3(x, y))$.

Подстановки указанного вида достаточно специфичны (все функции g имеют одно и то же число аргументов) и не исчерпывают

всевозможных подстановок, которые можно производить над функциями. Но благодаря функциям-проекторам I_m^n этот недостаток легко устраняется: любая суперпозиция функций в функции может быть выражена через суперпозиции указанного вида и функции-проекторы. В самом деле, например, суперпозиция $f(g_1(x_1, x_2),\ g_2(x_1))$ может быть в требуемом виде представлена так: $f(g_1(x_1, x_2), I_1^2(g_2(x_1), g_3(x_1)))$, где $g_3(x_1)$ – любая функция от x_1. В свою очередь, используя подстановку и функции-проекторы, можно переставлять аргументы в функции:

$$f(x_2, x_1, x_3, ..., x_n) = f(I_2^2(x_1, x_2), I_1^2(x_1, x_2), x_3, ..., x_n),$$
$$f(x_1, x_1, x_3, ..., x_n) = f(I_1^2(x_1, x_2), I_1^2(x_1, x_2), x_3, ..., x_n).$$

Поэтому будем считать, что теорема полностью доказана. □

Тезис Тьюринга (основная гипотеза теории алгоритмов). Вернёмся к интуитивному представлению об алгоритмах (см. §1.2). Напомним, одно из свойств алгоритма заключается в том, что он представляет собой единый способ, позволяющий для каждой задачи из некоего бесконечного множества задач за конечное число шагов найти её решение.

На понятие алгоритма можно взглянуть и с несколько иной точки зрения. Каждую задачу из бесконечного множества задач можно выразить (закодировать) некоторым словом некоторого алфавита, а решение задачи – каким-то другим словом того же алфавита. В результате получим функцию, заданную на некотором подмножестве множества всех слов выбранного алфавита и принимающую значения в множестве всех слов того же алфавита. Решить какую-либо задачу – значит найти значение этой функции на слове, кодирующем данную задачу. А иметь алгоритм для решения всех задач данного класса – значит иметь единый способ, позволяющий в конечное число шагов "вычислять" значения построенной функции для любых значений аргумента из её области определения. Таким образом, алгоритмическая проблема – по существу проблема о вычислении значений функции, заданной в некотором алфавите.

Остаётся уточнить, что значит уметь вычислять значения функции. Это значит вычислять значения функции с помощью подходящей машины Тьюринга. Для каких же функций возможно их тьюрингово вычисление? Многочисленные исследования учёных, обширный опыт показали, что такой класс функций чрезвычайно широк. Каждая функция, для вычисления значений которой существует какой-нибудь алгоритм, оказывалась вычислимой посредством некоторой машины Тьюринга. Это дало повод Тьюрингу высказать

следующую гипотезу, называемую основной гипотезой теории алгоритмов или тезисом Тьюринга.

ТЕЗИС ТЬЮРИНГА. ***Для нахождения значений функции, заданной в некотором алфавите, тогда и только тогда существует какой-нибудь алгоритм, когда функция является вычислимой по Тьюрингу, то есть когда она может вычисляться на подходящей машине Тьюринга.***

Это означает, что строго математическое понятие вычислимой (по Тьюрингу) функции является, по существу, идеальной моделью взятого из опыта понятия алгоритма. Данный тезис есть не что иное, как аксиома, постулат, выдвигаемый нами, о взаимосвязях нашего опыта с той математической теорией, которую мы под этот опыт хотим подвести. Конечно же, данный тезис в принципе не может быть доказан методами математики, потому что он не имеет внутриматематического характера (одна сторона в тезисе – понятие алгоритма – не является точным математическим понятием). Он выдвинут, исходя из опыта, и именно опыт подтверждает его состоятельность. Точно так же, например, не могут быть доказаны и математические законы механики; они открыты Ньютоном и многократно подтверждены опытом.

Впрочем, не исключается принципиальная возможность того, что тезис Тьюринга будет опровергнут. Для этого должна быть указана функция, вычислимая с помощью какого-нибудь алгоритма, но не вычислимая ни на какой машине Тьюринга. Но такая возможность представляется маловероятной, – в этом одно из значений гипотезы, – всякий алгоритм, который будет открыт, может быть реализован на машине Тьюринга.

Дополнительные косвенные доводы в подтверждение этой гипотезы будут приведены в двух последующих параграфах, где рассматриваются другие формализации интуитивного понятия алгоритма и доказывается их равносильность с понятием машины Тьюринга.

Машины Тьюринга и современные электронно-вычислительные машины. Изучение машин Тьюринга и практика составления программ для них закладывают фундамент алгоритмического мышления, сущность которого состоит в том, что нужно уметь разделять тот или иной процесс вычисления или какой-либо другой деятельности на простые составляющие шаги. В машине Тьюринга расчленение (анализ) вычислительного процесса на простейшие операции доведено до предельной возможности: распознавание

единичного рассмотренного вхождения символа, перемещение точки наблюдения данного ряда символов в соседнюю точку и изменение имеющейся в памяти информации. Конечно, такое мелкое дробление вычислительного процесса, реализуемого в машине Тьюринга, значительно его удлиняет. Но вместе с тем логическая структура процесса, расчлененного, образно выражаясь, до атомарного состояния, значительно упрощается и предстает в некотором стандартном виде, весьма удобном для теоретических исследований. (Именно такое расчленение на простейшие составляющие вычислительного процесса на машине Тьюринга даёт ещё один косвенный аргумент в пользу тезиса Тьюринга, обсуждавшегося в предыдущем пункте: всякая функция, вычисляемая с помощью какого-либо алгоритма, может быть вычислена на машине Тьюринга, потому что каждый шаг данного алгоритма можно расчленить на ещё более мелкие операции, которые реализуются в машине Тьюринга.) Таким образом, понятие машины Тьюринга есть теоретический инструмент анализа алгоритмического процесса, и значит, анализа существа алгоритмического мышления.

В современных ЭВМ алгоритмический процесс расчленён не на столь мелкие составляющие, как в машинах Тьюринга. Наоборот, создатели ЭВМ стремятся к известному укрупнению выполняемых машиной процедур (на этом пути, конечно, есть свои ограничения). Так, для выполнения операции сложения на машине Тьюринга составляется целая программа, а в современной ЭВМ такая операция является простейшей.

Далее, машина Тьюринга обладает бесконечной внешней памятью (неограниченная в обе стороны лента, разбитая на ячейки). Но ни в одной реально существующей машине бесконечной памяти быть не может. Это говорит о том, что машины Тьюринга отображают потенциальную возможность неограниченного увеличения объёма памяти современных ЭВМ.

Можно провести более подробный сравнительный анализ работы современной ЭВМ и машины Тьюринга. В большинстве ЭВМ принята трехадресная система команд, обусловленная необходимостью выполнения бинарных операций, в которых участвует содержимое сразу трёх ячеек памяти. Например, число из ячейки a умножается на число из ячейки b, и результат отправляется в ячейку c. Существуют ЭВМ двухадресные и одноадресные. Так, одноадресная ЭВМ работает следующим образом: вызывается (в сумматор) число из ячейки a; в сумматоре происходит, например, умножение этого числа на число из ячейки b; результат отправляется из сумматора в

ячейку c. Машину Тьюринга можно считать одноадресной машиной, в которой система одноадресных команд упрощена ещё больше: на каждом шаге работы машины команда предписывает замену лишь единственного знака, хранящегося в обозреваемой ячейке, а адрес обозреваемой ячейки при переходе к следующему такту может меняться лишь на единицу (обозрение соседней справа или слева ячейки ленты) или не меняться вовсе. Это удлиняет процесс, но в то же время резко унифицирует его, делает стандартным.

Наконец, исключительно велико влияние идеологии тьюрингова программирования на программирование в современных компьютерах. Истоки многих парадигм и языков программирования находятся в формальных моделях алгоритмических вычислений. Так, машины Тьюринга привели к императивному программированию, нашедшему своё выражение в языках программирования Фортран, Бейсик, Паскаль. Теория рекурсивных функций, о которой мы будем говорить в следующей главе III, привела к функциональному программированию (язык программирования Лисп), нормальные алгоритмы А.А.Маркова, о которых речь пойдёт в главе IV, привели к продукционному программированию (язык программирования Комит, Снобол, Рефал). К объектно ориентированному программированию и языкам программирования Смолток, C^{++} привела так называемая теория акторов, которой мы здесь не касаемся.

Подводя итоги, можно сказать, что современные ЭВМ есть некие реальные физические модели машин Тьюринга, огрубленные с точки зрения теории, но созданные в целях реализации конкретных вычислительных процессов. В свою очередь, понятие машины Тьюринга и теория таких машин есть теоретический фундамент и обоснование современных ЭВМ.

Глава III
РЕКУРСИВНЫЕ ФУНКЦИИ

Понятие машины Тьюринга – не единственный известный путь уточнения понятия алгоритма. В настоящей и следующей главах будут рассмотрены ещё два способа такого уточнения – рекурсивные функции и нормальные алгоритмы Маркова.

3.1. Начало теории рекурсивных функций

Происхождение рекурсивных функций. Всякий алгоритм однозначно сопоставляет допустимым начальным данным результат. Это означает, что с каждым алгоритмом однозначно связана функция, которую он вычисляет. В предыдущей главе II были рассмотрены примеры функций, которые вычисляются с помощью алгоритма, названного машиной Тьюринга. Возникает естественный вопрос, для всякой ли функции существует вычисляющий её алгоритм. Учитывая сформулированный в §2.4 тезис Тьюринга, утверждающий, что функция вычислима с помощью какого-нибудь алгоритма тогда и только тогда, когда она вычислима с помощью машины Тьюринга, данный вопрос трансформируется в следующий. Для всякой ли функции можно указать вычисляющую её машину Тьюринга? Если нет, то для каких функций существует вычисляющий их алгоритм (машина Тьюринга), как описать такие, как говорят, алгоритмически или эффективно вычислимые функции ?

Исследование этих вопросов привело к созданию в 30-х годах XX века теории рекурсивных функций. При этом класс вычислимых функций (названных здесь рекурсивными) получил такое описание, которое весьма напомнило процесс построения аксиоматической теории на базе некоторой системы аксиом. Сначала были выбраны простейшие функции, эффективная вычисляемость которых была очевидна (своего рода "аксиомы"). Затем сформулированы некоторые правила (названные операторами), на основе которых можно

строить новые функции из уже имеющихся (своего рода "правила вывода"). Тогда требуемым классом функций будет совокупность всех функций, получающихся из простейших с помощью выбранных операторов. Наша цель будет состоять в том, чтобы доказать, что этот класс функций совпадёт с классом функций, вычислимых с помощью машин Тьюринга.

Идея доказательства этого утверждения в одну сторону проста: сначала доказать вычислимость по Тьюрингу простейших функций, а затем – вычислимость по Тьюрингу функций, получающихся из вычислимых по Тьюрингу функций с помощью выбранных операторов.

Простейшие функции. Приступим к построению класса рекурсивных функций в соответствии с изложенными принципами. Напомним, что рассматриваются функции, заданные на множестве натуральных чисел и принимающие натуральные значения. Функции предполагается брать частичные, то есть определённые, вообще говоря, не для всех значений аргументов. В качестве исходных, простейших, функций выберем следующие:

$S(x) = x + 1$ (*функция следования*),

$O(x) = 0$ (*нуль-функция*),

$I_m^n(x_1, x_2, ..., x_n) = x_m$ (*функции-проекторы*, $1 \le m \le n$).

Вычислимость (более того, правильная вычислимость) этих функций с помощью машин Тьюринга установлена в примерах 2.2.1, 2.4.4 и 2.4.5 предыдущей главы II.

Далее, в качестве операторов, с помощью которых будут строиться новые функции, выберем следующие три: оператор суперпозиции, оператор примитивной рекурсии и оператор минимизации.

Оператор суперпозиции. Понятие суперпозиции функций и сложной функции хорошо известно, в частности, из курса математического анализа. Мы напомнили это понятие в §2.4: там мы дали определение суперпозиции функций (определение 2.4.6) и доказали, что функция, получающаяся с помощью оператора суперпозиции из функций, (правильно) вычислимых по Тьюрингу, сама (правильно) вычислима по Тьюрингу (теорема 2.4.7). Если функция $\varphi(x_1, ..., x_n)$ получена из функций $f, g_1, g_2, ..., g_m$ с помощью оператора суперпозиции, то будем писать: $\varphi = S(f; g_1, g_2, ..., g_m)$, где S – оператор суперпозиции.

Свойства оператора суперпозиции:

1) Сохранение всюду определённости функций, т.е. если f, g_1, g_2, ... , g_m – всюду определены, то и $\varphi = S(f; g_1, g_2, ..., g_m)$ – всюду определена. [Если хотя бы одно из значений f, g_1, g_2, ... , g_m не определено, то и значение φ не определено].

2) Сохранение алгоритмической вычислимости функций, т.е. если f, g_1, g_2, ... , g_m – вычислимы, то и $\varphi = S(f; g_1, g_2, ..., g_m)$ – вычислима.

Оператор примитивной рекурсии. Определение 3.1.1. Говорят, что $(n+1)$-местная функция φ получена из n-местной функции f и $(n+2)$-местной функции g с помощью оператора *примитивной рекурсии*, если для любых $x_1, ..., x_n, y$ справедливы равенства:

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

Пара этих равенств называется *схемой примитивной рекурсии*. Обозначение: $\varphi = R(f, g)$, где R – оператор примитивной рекурсии.

Здесь важно отметить то, что независимо от числа аргументов в φ рекурсия ведётся только по одной переменной y; остальные n переменных $x_1, ..., x_n$ на момент применения схемы примитивной рекурсии зафиксированы и играют роль параметров. Кроме того, схема примитивной рекурсии выражает каждое значение определяемой функции φ не только через данные функции f и g, но и через так называемые предыдущие значения определяемой функции φ: прежде чем получить значение $\varphi(x_1, ..., x_n, k)$, придётся проделать $k+1$ вычисление по указанной схеме – для $y = 0, 1, ..., k$.

Свойства оператора примитивной рекурсии:

1) Сохранение всюду определённости функций, т.е. если f, g – всюду определены, то и $\varphi = R(f, g)$ – всюду определена.

2) Сохранение алгоритмической вычислимости, т.е. если f, g – вычислимы, то и $\varphi = R(f, g)$ – вычислима.

Доказательство. Пусть функции f, g вычислимы. Тогда:

$$\begin{aligned} \varphi(a, 0) &= f(a) = b_0, \\ \varphi(a, 1) &= g(a, 0, \varphi(a, 0)) = g(a, 0, b_0) = b_1, \\ \varphi(a, 2) &= g(a, 1, \varphi(a, 1)) = g(a, 1, b_1) = b_2, \\ &\ldots \end{aligned}$$

Таким образом, φ – вычислима. □

Термин "рекурсия" происходит от латинского слова *recursio*, что означает "бегу назад", "возвращаюсь". Примитивная рекурсия – потому что возвращаюсь лишь на один шаг, на единицу.

Рассмотрим два примера схем примитивной рекурсии.

Пример 3.1.2. Определим функцию $\sigma(x,y) = x+y$ с помощью схемы примитивной рекурсии, взяв в качестве функций $f(x)$, $g(x)$ одну и ту же функцию $\nu(x) = x'$ – функция следования. Тогда схема имеет вид (в правом столбце эта же схема примитивной рекурсии записана в более привычных обозначениях для операции сложения):

$$\begin{array}{ll} \sigma(x,0) = x & x+0=x \\ \sigma(x,1) = \nu(x) & x+1=x' \\ \sigma(x,y+1) = \nu(\sigma(x,y)). & x+y' = (x+y)'. \end{array}$$

Пример 3.1.3. Определим функцию $\pi(x,y) = x \cdot y$ с помощью схемы примитивной рекурсии, взяв в качестве функций $f(x)$, $g(x)$ следующие: $f(x) = \Delta(x) = x$ – тождественная функция, $g(x,y) = \sigma(x,y)$ – функция из предыдущего примера. Тогда схема имеет вид (в правом столбце снова эта же схема примитивной рекурсии записана в более привычных обозначениях для операции умножения):

$$\begin{array}{ll} \pi(x,0) \ = \ 0 & x \cdot 0 = 0 \\ \pi(x,1) \ = \ \Delta(x) & x \cdot 1 = x \\ \pi(x,y+1) \ = \ \sigma(x,\pi(x,y)) & x \cdot y' = x \cdot y + x = x \cdot (y+1). \end{array}$$

Примеры других схем рекурсии (не примитивных).

Пример 3.1.4. $u_0 = 0, \quad u_1 = 1, \quad u_{n+1} \ = \ u_n + u_{n-1}$.
Эта схема порождает последовательность (функцию натурального аргумента): 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 ,...
Здесь возврат происходит на два шага. Эта последовательность получила название последовательность Фибоначчи по имени итальянского купца, занимавшегося математикой и называвшегося также Леонардо Пизанским (1180 – 1240).

Пример 3.1.5. $u_0 = 0, \quad u_1 = 1, \quad u_2 = 4, \quad u_{n+3} = 3u_{n+2} - 3u_{n+1} + u_n.$ $(n \geq 0)$ В этой схеме возврат происходит на три шага. Вычислим несколько членов этой последовательности:

$$u_3 = 3u_2 - 3u_1 + u_0 = 3 \cdot 4 - 3 \cdot 1 + 0 = 9 \ = \ 3^2 \ ;$$

$$u_4 = 3u_3 - 3u_2 + u_1 = 3 \cdot 9 - 3 \cdot 4 + 1 = 16 \ = \ 4^2 \ ;$$

$$u_5 = 3u_4 - 3u_3 + u_2 = 3 \cdot 16 - 3 \cdot 9 + 4 = 25 \ = \ 5^2 \ ;$$

. . . .

Возникает гипотеза: $u_n = 3u_{n-1} - 3u_{n-2} + u_{n-3} = n^2$.

Её нетрудно доказать методом полной математической индукции. База индукции уже обоснована. Предположим, что данное утверждение справедливо для всех членов последовательности с номерами $\leq n$. Докажем тогда, что оно справедливо и для $(n+1)$-го члена последовательности. Для этого вычисляем, используя сделанное предположение индукции:

$$u_{n+1} = 3u_n - 3u_{n-1} + u_{n-2} = 3n^2 - 3(n-1)^2 + (n-2)^2 =$$
$$= 3n^2 - 3n^2 + 6n - 3 + n^2 - 4n + 4 = n^2 + 2n + 1 = (n+1)^2.$$

Следовательно, данное утверждение верно для любого члена последовательности (т.е. члена последовательности с любым натуральным номером n).

Понятие примитивно рекурсивной функции. Класс примитивно рекурсивных функций строится генетическим образом. Сначала выбираются первоначальные (или простейшие) функции. В качестве них берутся: $S(x) = x + 1$ – функция следования; $O(x) = 0$ – нуль-функция; $I_m^n(x_1, ..., x_n) = x_m$ – функции-проекторы.

ОПРЕДЕЛЕНИЕ 3.1.6. Функция называется *примитивно рекурсивной* (прф), если она может быть получена из простейших функций S, O, I_m^n с помощью конечного числа применений операторов суперпозиции и примитивной рекурсии.

Таким образом, класс **ПРФ** представляет собой замыкание совокупности первоначальных функций S, O, I_m^n относительно операторов суперпозиции и примитивной рекурсии. Отсюда вытекает простое, но важное свойство класса **ПРФ**.

СВОЙСТВО КЛАССА **ПРФ**: *класс* **ПРФ** *замкнут относительно операторов суперпозиции и примитивной рекурсии.* Другими словами, ***функция, полученная в результате суперпозиции, а также по схеме примитивной рекурсии из примитивно рекурсивных функций сама является примитивно рекурсивной.***

Отметим в заключение, что схема примитивной рекурсии задания функций устроена так, что она позволяет последовательно вычислять значение определяемой функции при каждом натуральном n. Это означает, в частности, что *любая примитивно рекурсивная функция непременно всюду определена.*

Наконец, введём заключительный, третий, оператор.

Оператор минимизации. Определение 3.1.7. Будем говорить, что n-местная функция φ получается из $(n+1)$-местных функций f_1 и f_2 с помощью ***оператора минимизации***, или оператора наименьшего числа, если для любых $x_1, ..., x_n, y$ равенство $\varphi(x_1, ..., x_n) = y$ выполнено тогда и только тогда, когда значения $f_i(x_1, ..., x_n, 0)$, ... , $f_i(x_1, ..., x_n, y-1)$ $(i = 1, 2)$ определены и попарно неравны:

$$f_1(x_1, ..., x_n, 0) \neq f_2(x_1, ..., x_n, 0)$$

$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$

$$f_1(x_1, ..., x_n, y-1) \neq f_2(x_1, ..., x_n, y-1)\ , \qquad \text{а}$$

$$f_1(x_1, ..., x_n, y) = f_2(x_1, ..., x_n, y)\ .$$

Коротко говоря, величина $\varphi(x_1, ..., x_n)$ равна наименьшему значению аргумента y, при котором выполняется последнее равенство. Используется следующее обозначение:

$$\varphi(x_1, ..., x_n) = \mu y[f_1(x_1, ..., x_n, y) = f_2(x_1, ..., x_n, y)]\ .$$

В частном случае может быть $f_2 = 0$. Тогда

$$\varphi(x_1, ..., x_n) = \mu y[f(x_1, ..., x_n, y) = 0]\ .$$

Оператор минимизации называется также ***μ-оператором***.

Этот оператор предназначен для порождения следующего важного класса рекурсивных функций.

Понятие частично рекурсивной и общерекурсивной функции. Класс частично рекурсивных функций также строится генетическим образом.

Определение 3.1.8. Функция называется ***частично рекурсивной*** (чрф), если она может быть получена из простейших функций S, O, I^n_m с помощью конечного числа применений операторов суперпозиции, примитивной рекурсии и μ-оператора. Если функция всюду определена и частично рекурсивна, то она называется *общерекурсивной* (орф).

(Отметим в скобках, что не всегда частично рекурсивную функцию можно эффективно доопределить до общерекурсивной).

Таким образом, класс **ЧРФ** представляет собой замыкание совокупности первоначальных функций S, O, I^n_m относительно операторов суперпозиции, примитивной рекурсии и минимизации. Отсюда вытекает простое, но важное свойство класса **ЧРФ**.

Свойство класса **ЧРФ**: *класс* **ЧРФ** *замкнут относительно операторов суперпозиции, примитивной рекурсии и минимизации.* Другими словами, *функция, полученная в результате суперпозиции, а также по схеме примитивной рекурсии, а также в результате применения μ-оператора из частично рекурсивных функций сама является частично рекурсивной.*

Тезис Чёрча (основная гипотеза теории рекурсивных функций). Ясно, что всякая примитивно рекурсивная функция будет и частично рекурсивной (и даже общерекурсивной, так как каждая примитивно рекурсивная функция всюду определена), поскольку для построения частично рекурсивных функций из простейших используется больше средств, чем для построения примитивно рекурсивных функций. В то же время класс частично рекурсивных функций шире класса примитивно рекурсивных функций, так как все примитивно рекурсивные функции всюду определены, а среди частично рекурсивных функций встречаются и функции не всюду определённые. **ПРФ** $\subset$ **ОРФ** $\subset$ **ЧРФ** .

Понятие частично рекурсивной функции оказалось исчерпывающей формализацией понятия вычислимой функции. При построении аксиоматической теории высказываний (см. Учебник МЛ, глава II) исходные формулы (аксиомы) и правила вывода выбирались так, чтобы полученные в теории формулы исчерпали бы все тавтологии алгебры высказываний. К чему же стремимся мы в теории рекурсивных функций, почему именно так выбрали простейшие функции и операторы для получения новых функций? Рекурсивными функциями мы стремимся исчерпать все мыслимые функции, поддающиеся вычислению с помощью какой-нибудь определённой процедуры механического характера. Подобно тезису Тьюринга и в теории рекурсивных функций выдвигается соответствующая естественнонаучная гипотеза, носящая название тезиса Чёрча в честь американского математика, внесшего значительный вклад в теорию рекурсивных функций.

Тезис Чёрча. *Числовая функция тогда и только тогда алгоритмически (или машинно) вычислима, когда она частично рекурсивна.*

И эта гипотеза не может быть доказана строго математически, она подтверждается практикой, опытом, ибо призвана увязать практику и теорию. Все рассматривавшиеся в математике конкретные функции, признаваемые вычислимыми в интуитивном смысле, оказывались частично рекурсивными.

Теперь мы рассмотрим более подробно классы примитивно рекурсивных функций и частично рекурсивных функций и докажем, что все функции из каждого из этих классов вычислимы на подходящих машинах Тьюринга.

3.2. Примитивно рекурсивные функции

Итак, функция примитивно рекурсивна, если она может быть получена из простейших функций S, O, I^n_m с помощью конечного числа применений операторов суперпозиции и примитивной рекурсии.

Примеры примитивно рекурсивных функций. Рассмотрим ряд примеров примитивно рекурсивных функций. Сначала ещё раз вернёмся к функциям $\sigma(x,y)$ и $\pi(x,y)$ из примеров 3.1.2 и 3.1.3 и покажем, что они примитивно рекурсивны.

ПРИМЕР 3.2.1. Покажем, что функция $\sigma(x,y) = x + y$ может быть получена из простейших с помощью оператора примитивной рекурсии. Для функции верны следующие тождества:

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

которые можно записать в виде

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

или

$$\begin{aligned} \sigma(x,0) &= I^1_1(x), \\ \sigma(x,y+l) &= S(\sigma(x,y)). \end{aligned}$$

А это и есть схема примитивной рекурсии, основывающаяся на простейших функциях I^1_1 и S. Следовательно, функция σ примитивно рекурсивна по определению.

ПРИМЕР 3.2.2. Аналогично операции сложения, очевидные соотношения имеют место для операции умножения:

$$\pi(x,0) = x\cdot 0 = 0 = O(x),$$

$$\pi(x,y{+}1) = x\cdot y{+}x = \pi(x,y){+}x = x{+}\pi(x,y) = \sigma(x,\pi(x,y)).$$

Они говорят о том, что функция $\pi(x,y) = x\cdot y$ получена из простейшей функции $O(x)$ и примитивно рекурсивной функции $\sigma(x,y) =$

$x + y$ с помощью оператора примитивной рекурсии. Следовательно, функция π примитивно рекурсивна ввиду замкнутости класса примитивно рекурсивных функций относительно оператора примитивной рекурсии.

Продолжим построение примеров примитивно рекурсивных функций.

Пример 3.2.3. *Экспоненциальная функция* $\exp(x, y) = x^y$. Схема примитивной рекурсии для этой функции может быть представлена следующим образом:

$$\exp(x, 0) = 1 = S(O(x)),$$
$$\exp(x, y + 1) = x^y \cdot x = \exp(x, y) \cdot x = \pi(x, \exp(x, y)).$$

Таким образом, функция $\exp(x, y) = x^y$ получена из простейших функций $S(x)$ и $O(x)$ и примитивно рекурсивной функции $\pi(x, y) = x \cdot y$ (см. предыдущий пример) с помощью оператора примитивной рекурсии. Следовательно, функция $\exp(x, y)$ примитивно рекурсивна ввиду замкнутости класса примитивно рекурсивных функций относительно оператора примитивной рекурсии.

Пример 3.2.4. ***Функция-факториал*** $\varphi(x) = x!$. Схема примитивной рекурсии для этой функции имеет следующий вид:

$$\varphi(0) = 1 = S(O(x)),$$
$$\varphi(x + 1) = \pi(S(x), \varphi(x)) \quad [= x! \cdot (x + 1)].$$

Таким образом, функция $\varphi(x) = x!$ получена из простейших функций $S(x)$ и $O(x)$ и примитивно рекурсивной функции $\pi(x, y) = x \cdot y$ (см. пример 3.2.2) с помощью оператора примитивной рекурсии. Следовательно, функция $\varphi(x)$ примитивно рекурсивна ввиду замкнутости класса примитивно рекурсивных функций относительно оператора примитивной рекурсии.

Пример 3.2.5. *Предшественник числа* x. Эта функция определяется следующим образом:

$$pd(x) = x \dot{-} 1 = \begin{cases} x - 1, & \text{если } x \geq 1, \\ 0, & \text{если } x < 1. \end{cases}$$

Схема примитивной рекурсии для неё имеет следующий вид:

$$pd(0) = 0 \dot{-} 1 = 0 = O(x),$$
$$pd(x + 1) = (x + 1) \dot{-} 1 = x = I_1^2(x, y).$$

Таким образом, функция $pd(x) = x \dot{-} 1$ получена из простейших функций $O(x)$ и $I_1^2(x,y)$ с помощью оператора примитивной рекурсии. Следовательно, функция $pd(x)$ примитивно рекурсивна по определению.

Пример 3.2.6. *Усечённая разность* $x \dot{-} y$. Эта функция определяется следующим образом:

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

Схема примитивной рекурсии для неё имеет следующий вид:

$$x \dot{-} 0 = x = I_1^2(x,y) ,$$
$$x \dot{-} (y+1) = (x \dot{-} y) \dot{-} 1 = pd(x \dot{-} y) .$$

Эти тождества показывают, что функция $x \dot{-} y$ получена с помощью оператора примитивной рекурсии из простейшей функции $I_1^2(x,y)$ и примитивно рекурсивной функции (см. предыдущий пример) $pd(x)$. Следовательно, функция $x \dot{-} y$ примитивно рекурсивна ввиду замкнутости класса примитивно рекурсивных функций относительно оператора примитивной рекурсии.

Пример 3.2.7. *Знак числа* x. Эта функция определяется следующим образом:

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

Схема примитивной рекурсии для неё имеет следующий вид:

$$sg(0) = 0 = O(x) ,$$
$$sg(x+1) = 1 = S(0) = S(O(x)) .$$

Таким образом, функция $sg(x)$ получена из простейших функций $O(x)$ и $S(x)$ с помощью оператора примитивной рекурсии. Следовательно, функция $sg(x)$ примитивно рекурсивна по определению.

Пример 3.2.8. *Антизнак числа* x. Эта функция определяется следующим образом:

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

Нетрудно проверить, что эту функцию можно представить в виде следующей суперпозиции примитивно рекурсивных функций:

$$\overline{sg}(x) = 1 \dot{-} x = S(O(x)) \dot{-} x .$$

Следовательно, функция $\overline{sg}(x)$ примитивно рекурсивна ввиду замкнутости класса примитивно рекурсивных функций относительно оператора суперпозиции.

Составьте схему примитивной рекурсии для этой функции.

Пример 3.2.9. *Модуль разности* $|x-y|$. Покажем, что для этой функции справедливо следующее выражение через примитивно рекурсивные функции: $\boxed{|x-y| = (x \dot{-} y) + (y \dot{-} x)}$.

В самом деле, если $x > y$, то $x - y > 0$ и $|x-y| = x - y$. С другой стороны $(x \dot{-} y) + (y \dot{-} x) = (x-y) + 0 = x - y$. Если $x = y$, то обе части равенства дают 0. Наконец, если $x < y$, то $|x-y| = -(x-y) = y-x$. С другой стороны $(x \dot{-} y) + (y \dot{-} x) = 0 + (y-x) = y-x$.

Таким образом, из доказанного равенства в силу примитивной рекурсивности функций + (пример 3.2.1) и $x \dot{-} y$ (пример 3.2.6), а также ввиду замкнутости класса примитивно рекурсивных функций относительно оператора суперпозиции заключаем, что функция $|x-y|$ примитивно рекурсивна.

Пример 3.2.10. *Функция минимум* $\min(x,y)$. Покажем, что для этой функции справедливо следующее выражение через примитивно рекурсивную функцию $x \dot{-} y$: $\boxed{\min(x,y) = x \dot{-} (x \dot{-} y)}$.

В самом деле, если $x \le y$, то $\min(x,y) = x$. С другой стороны $x \dot{-} (x \dot{-} y) = x \dot{-} 0 = x - 0 = x$. Если же $x > y$, то $\min(x,y) = y$. С другой стороны $x \dot{-} (x \dot{-} y) = x \dot{-} (x-y) = x-(x-y) = x-x+y = y$.

Таким образом, из доказанного равенства в силу примитивной рекурсивности функции $x \dot{-} y$ (пример 3.2.6), а также ввиду замкнутости класса примитивно рекурсивных функций относительно оператора суперпозиции заключаем, что функция $\min(x,y)$ примитивно рекурсивна.

Пример 3.2.11. *Функция максимум* $\max(x,y)$. Покажите самостоятельно, что для этой функции справедливо следующее выражение через примитивно рекурсивные функции и сделайте вывод о её примитивной рекурсивности: $\boxed{\max(x,y) = y + (x \dot{-} y)}$.

Пример 3.2.12. *Остаток от деления y на x.* При делении с остатком натурального числа y на натуральное число x имеет место равенство $y = x \cdot q + r$, где q называется *частным*, а *остаток* r заключён в пределах $0 \le r < x$. При этом частное q и остаток r представляют собой две функции, зависящие от x, y. Каждая из этих функций является частичной: каждая из ни не определена при $x = 0$. Покажем, что обе эти функции примитивно рекурсивны. Рассмотрим сначала функцию $r(x, y)$. Чтобы сделать её всюду определённой определим её следующим образом (доопределив в 0):

$$r(x,y) = \begin{cases} r \text{ -- остаток от деления } y \text{ на } x, & \text{если } x \neq 0, \\ 0\ , & \text{если } x = 0. \end{cases}$$

Покажем, что схема примитивной рекурсии для неё может быть записана следующим образом:

$$r(x,0) = 0\ ,$$

$$r(x,y+1) = (r(x,y)+1) \cdot sg(x \dot{-} (r(x,y)+1))\ . \qquad (*)$$

При отыскании остатка $r(x, y + 1)$ от деления следующего числа $y + 1$ на x нужно к остатку $r(x, y)$ от деления предыдущего числа y на x прибавить единицу. При этом могут представиться две возможности.

Первая возможность: после прибавления к предыдущему остатку $r(x, y)$ единицы получится число меньшее делителя x: $r(x, y) + 1 < x$. Это будет означать, что новый остаток $r(x, y + 1)$ будет получаться из старого прибавлением к нему единицы: $r(x, y + 1) = r(x, y) + 1$. Этот же результат в этом случае выдаёт и доказываемая формула $(*)$. В самом деле, в ней тогда $x \dot{-} (r(x, y) + 1) = x - (r(x, y) + 1) > 0$ (так как $r(x, y) + 1 < x$) и следовательно, $sg(x \dot{-} (r(x, y) + 1)) = 1$ и правая часть формулы $(*)$ в итоге даёт $r(x, y) + 1$.

Вторая возможность: после прибавления к предыдущему остатку $r(x, y)$ единицы получится число равное делителю x: $r(x, y) + 1 = x$. Это будет означать, что число $y + 1$ делится на x нацело, без остатка, т.е. остаток $r(x, y + 1) = 0$: $y + 1 = x \cdot q + (r + 1) = x \cdot q + x = x \cdot (q + 1)$. Этот же результат в этом случае выдаёт и доказываемая формула $(*)$. В самом деле, в ней тогда $x \dot{-} (r(x, y) + 1) = x \dot{-} x = x - x = 0$ и при умножении этого нуля на $(r(x, y) + 1)$ в правой части формулы $(*)$ получаем 0.

Следовательно, функция $r(x, y)$ примитивно рекурсивна ввиду замкнутости класса примитивно рекурсивных функций относительно оператора примитивной рекурсии.

Пример 3.2.13. *Частное от деления y на x.* Как отмечено в предыдущем примере, при делении с остатком натурального числа y на натуральное число x возникает частичная функция частное q, определяемая равенством $y = x \cdot q + r$, где остаток r заключён в пределах $0 \leq r < x$. Определим эту функцию так, чтобы она стала всюду определённой:

$$q(x,y) = \begin{cases} [y/x] \text{ -- целая часть от деления } y \text{ на } x, & \text{если } x \neq 0, \\ 0\ , & \text{если } x = 0. \end{cases}$$

Покажем, что схема примитивной рекурсии для этой функции может быть записана следующим образом:

$$\begin{aligned} q(x,0) &= 0\ , \\ q(x,y+1) &= q(x,y)+\overline{sg}|x-(r(x,y)+1)|\ . \end{aligned} \qquad (**)$$

Изучите предварительно решение предыдущего примера. Частное $q(x, y+1)$ от деления следующего числа $y+1$ на x либо останется прежним, т.е. равным частному $q(x,y)$ от деления предыдущего числа y на x: $q(x,y+1) = q(x,y)$, либо увеличится на единицу: $q(x,y+1) = q(x,y)+1$. Первое произойдёт в том случае, если $r(x,y)+1 < x$. Но тогда $x-(r(x,y)+1) > 0$, далее, $\overline{sg}|x-(r(x,y)+1)| = 0$ и правая часто доказываемого равенства (**) даст тот же результат $q(x,y)$. Второе произойдёт в том случае, если $r(x,y)+1 = x$. Это будет означать, что число $y+1$ делится на x нацело, без остатка: $y+1 = (x \cdot q + r)+1 = x \cdot q + (r+1) = x \cdot q + x = x \cdot (q+1)$. Но тогда $x - (r(x,y)+1) = 0$ и, значит, $\overline{sg}|x-(r(x,y)+1)| = 0$ и правая часто доказываемого равенства (**) даст тот же результат $q(x,y)+1$.

Следовательно, функция $q(x,y)$ примитивно рекурсивна ввиду замкнутости класса примитивно рекурсивных функций относительно оператора примитивной рекурсии.

Примитивная рекурсивность булевых функций. Напомним (см. Учебник МЛ, §8), что булева функция – это функция (от n аргументов), заданная и принимающая значения в двухэлементном множестве: f: $\{0,1\}^n \to \{0,1\}$.

Теорема 3.2.14. *Все булевы функции примитивно рекурсивны.*

Доказательство. Сначала покажем примитивную рекурсивность отрицания, конъюнкции и дизъюнкции. Это делается посредством их арифметизации – представления через числовые аналоги этих функций, которые на двухэлементном числовом множестве $\{0, 1\}$ ведут себя как указанные булевы функции:

$$x' = 1 \dot{-} x; \quad x \cdot y = \min(x,y); \quad x \vee y = \max(x,y) .$$

Все участвующие здесь функции $\dot{-}$, min, max примитивно рекурсивны (см. примеры 3.2.6, 3.2.10, 3.2.11).

Далее, как известно, всякая булева функция есть суперпозиция этих трёх функций. Поэтому, ввиду замкнутости класса примитивно рекурсивных функций относительно оператора суперпозиции, отсюда заключаем, что всякая булева функция примитивно рекурсивна. □

Операторы суммирования и мультиплицирования (умножения). Пусть дана функция $f(x_1, x_2, ..., x_n, y)$. Сопоставим ей две новые функции:

$$\Sigma f(x_1, x_2, ..., x_n, y) = \Sigma_{i=0}^{y} f(x_1, x_2, ..., x_n, i) \quad \text{и}$$

$$\Pi f(x_1, x_2, ..., x_n, y) = \Pi_{i=0}^{y} f(x_1, x_2, ..., x_n, i) .$$

Договоримся пустую сумму считать равной 0, а пустое произведение равным 1.

Ясно, что если функция f алгоритмически вычислима, то и функции Σf и Πf алгоритмически вычислимы.

Теорема 3.2.15. ***Если функция $f(x_1, ..., x_n, y)$ примитивно рекурсивна, то примитивно рекурсивны и функции Σf и Πf.***

Доказательство. Схема примитивной рекурсии для функции Σf:

$$\Sigma f(x_1, x_2, ..., x_n, 0) = f(x_1, x_2, ..., x_n, 0),$$

$$\Sigma f(x_1, x_2, ..., x_n, y+1) = (\Sigma f(x_1, x_2, ..., x_n, y)) + f(x_1, x_2, ..., x_n, y+1).$$

Схема примитивной рекурсии для функции Πf:

$$\Pi f(x_1, x_2, ..., x_n, 0) = f(x_1, x_2, ..., x_n, 0),$$

$$\Pi f(x_1, x_2, ..., x_n, y+1) = (\Pi f(x_1, x_2, ..., x_n, y)) \cdot f(x_1, x_2, ..., x_n, y+1).$$

Ввиду примитивной рекурсивности функций $+$ и $\cdot$ (примеры 3.2.1 и 3.2.2) и замкнутости класса всех примитивно рекурсивных функций относительно оператора примитивной рекурсии, отсюда заключаем, что примитивно рекурсивны будут и функции Σf и Πf. □

Замечание 3.2.16. Операторы суммирования Σf и мультиплицирования Πf можно рассматривать и в более общем виде, считая, что y, в свою очередь, является некоторой функцией: $y = \varphi(x_1, ..., x_n)$:

$$\Sigma f(x_1, x_2, ..., x_n, \varphi(x_1, ..., x_n)) = \Sigma_{i=0}^{\varphi(x_1,...,x_n)} f(x_1, x_2, ..., x_n, i) \quad \text{и}$$

$$\Pi f(x_1, x_2, ..., x_n, \varphi(x_1, ..., x_n)) = \Pi_{i=0}^{\varphi(x_1,...,x_n)} f(x_1, x_2, ..., x_n, i) \ .$$

Очевидно, что в предположении примитивной рекурсивности функции $\varphi(x_1, ..., x_n)$ доказанная теорема сохраняет свою силу: если f – примитивно рекурсивная функция, то и Σf и Πf – примитивно рекурсивные функции.

Рассмотренные операторы позволяют доказать примитивную рекурсивность ряда функций.

Пример 3.2.17. *Количество различных делителей числа x*: $\mathrm{nd}(x)$. Покажем, что эта функция следующим образом выражается через конечную сумму (оператор суммирования):

$$\boxed{\mathrm{nd}\,(x) = \Sigma_{i=0}^{x} \overline{sg}(r(i, x)) \dot{-} 1} \ .$$

В самом деле, если i делится на x, то остаток от деления $r(i, x) = 0$ и $\overline{sg}(r(i, x)) = 1$. Полученная единица прибавляется в сумму-счётчик. Если же если же i не делится на x, то остаток от деления $r(i, x) \neq 0$ и $\overline{sg}(r(i, x)) = 0$. В сумму-счётчик ничего не прибавляется. В итоге будет сложено столько единиц, сколько делителей имеется у числа x. Из итоговой суммы отнимается единица по той причине, что число 0 также будет сосчитано в качестве делителя числа x.

Итак, поскольку в полученной формуле оператор суммирования применён к примитивно рекурсивной функции $\overline{sg}(r(x, x)) \dot{-} 1$ (её примитивная рекурсивность следует из предыдущих рассмотрений – примеры 3.2.8, 3.2.12, 3.2.6), поэтому по теореме 3.2.15 функция $\mathrm{nd}(x)$ примитивно рекурсивна.

Пример 3.2.18. *Сумма различных делителей числа x*: $\sigma(x)$. Покажем, что эта функция следующим образом выражается через конечную сумму (оператор суммирования):

$$\boxed{\sigma(x) = \Sigma_{i=0}^{x} i \cdot \overline{sg}(r(i, x))} \ .$$

Проанализируйте сначала рассуждение в предыдущем примере. В данном случае, как только i окажется делителем числа x, в сумму будет положена не единица, а $i \cdot 1$, т.е. сам делитель i. В результате в сумматоре накопится сумма всех делителей числа x. В случае, когда $i = 0$, в сумматор прибавится число $0 \cdot \overline{sg}(r(0, x)) = 0 \cdot 1 = 0$.

Итак, поскольку в полученной формуле для функции $\sigma(x)$ оператор суммирования применён к примитивно рекурсивной функции $x \cdot \overline{sg}(r(x,x))$ (её примитивная рекурсивность следует из предыдущих рассмотрений – примеры 3.2.2, 3.2.8, 3.2.12), поэтому по теореме 3.2.15 функция $\sigma(x)$ примитивно рекурсивна.

3.3. Примитивно рекурсивные предикаты

Понятие примитивно рекурсивного предиката. Определив в Учебнике МЛ (начало §14) понятие предиката, мы отметили, что к этому понятию возможен и ещё один подход. Предикат $P(x_1, ..., x_n)$, заданный над множествами $M_1, ..., M_n$, есть функция, заданная на указанных множествах и принимающая значения в двухэлементном множестве $\{0, 1\}$. В теории алгоритмов принято различать предикат P и эту функцию, связанную с ним, а эту функцию называют характеристической функцией предиката P и обозначают χ_P. Таким образом:

$$\chi_P(x_1, ..., x_n) = \begin{cases} 1, & \text{если высказывание } P(x_1, ..., x_n) \text{ – истинно,} \\ 0, & \text{если высказывание } P(x_1, ..., x_n) \text{ – ложно.} \end{cases}$$

Если $M_1 = ... = M_n = N$, то χ_P – функция, заданная на множестве натуральных чисел и принимающая значения в нём же, т.е. – функция, входящая в круг наших рассмотрений, и имеет смысл поставить вопрос о её примитивной рекурсивности.

Определение 3.3.1. Предикат P называется ***примитивно рекурсивным***, если его характеристическая функция χ_P примитивно рекурсивна.

Примеры примитивно рекурсивных предикатов. Рассмотрим ряд примеров примитивно рекурсивных предикатов.

Пример 3.3.2. *Предикат равенства* $P_1(x, y)$: "$x = y$". Его характеристическая функция определяется так:

$$\chi_{P_1}(x, y) = \begin{cases} 1, & \text{если } x = y, \\ 0, & \text{если } x \neq y. \end{cases}$$

Нетрудно понять, что эта функция следующим образом выражается через суперпозицию известных нам примитивно рекурсивных функций: $\chi_{P_1}(x, y) = \overline{sg}|x - y|$.

Поэтому функция $\chi_{P_1}(x, y)$ примитивно рекурсивна, а характеризуемый ей предикат $P_1(x, y)$ примитивно рекурсивен.

Пример 3.3.3. *Предикат нестрогого неравенства* $P_2(x, y)$: "$x \le y$". Его характеристическая функция определяется так:

$$\chi_{P_2}(x, y) = \begin{cases} 1, & \text{если} \quad x \le y, \\ 0, & \text{если} \quad x > y. \end{cases}$$

Нетрудно понять, что эта функция следующим образом выражается через суперпозицию известных нам примитивно рекурсивных функций: $\chi_{P_2}(x, y) = \overline{sg}(x \dot{-} y)$.

Поэтому функция $\chi_{P_2}(x, y)$ примитивно рекурсивна, а характеризуемый ей предикат $P_2(x, y)$ примитивно рекурсивен.

Пример 3.3.4. *Предикат строгого неравенства* $P_3(x, y)$: "$x < y$". Его характеристическая функция определяется так:

$$\chi_{P_3}(x, y) = \begin{cases} 1, & \text{если} \quad x < y, \\ 0, & \text{если} \quad x \ge y. \end{cases}$$

Нетрудно понять, что эта функция следующим образом выражается через суперпозицию известных нам примитивно рекурсивных функций: $\chi_{P_3}(x, y) = \overline{sg}((x + 1) \dot{-} y)$.

Данное выражение получается из выражения для предыдущей функции $\chi_{P_2}(x, y)$, если учесть, что для натуральных чисел: $x < y \iff x + 1 \le y$.

Поэтому функция $\chi_{P_3}(x, y)$ примитивно рекурсивна, а характеризуемый ей предикат $P_3(x, y)$ примитивно рекурсивен.

Пример 3.3.5. *Предикат равенства константе* $P_4(x)$: "$x = c$", где c – константа. Его характеристическая функция выражается так: $\chi_{P_4}(x) = \overline{sg}|x - c|$.

Пример 3.3.6. *Предикат "меньше константы"* $P_5(x)$: "$x < c$", где c – константа. Его характеристическая функция выражается так: $\chi_{P_5}(x) = \overline{sg}((x + 1) \dot{-} c)$.

Пример 3.3.7. *Предикат "больше константы"* $P_6(x)$: "$x > c$", где c – константа. Его характеристическая функция выражается так: $\chi_{P_6}(x) = sg(x \dot{-} c)$.

Пример 3.3.8. *Предикат "больше"* $P_7(x)$: "$x > y$". Его характеристическая функция выражается так:

$$\chi_{P_7}(x) = sg(x \dot{-} y) .$$

Пример 3.3.9. *Предикат чётности* $P_8(x)$: "x – чётно". Его характеристическая функция задаётся следующей схемой примитивной рекурсии:

$$\chi_{P_8}(0) = 1 ,$$

$$\chi_{P_8}(x+1) = \overline{sg}\chi_{P_8}(x) .$$

Пример 3.3.10. *Предикат делимости* $P_9(x,y) \equiv d(x,y)$: "$x|y$" ("x делит y"). Это означает, что найдётся такое натуральное q, что $y = x \cdot q$. Другими словами, в равенстве при делении с остатком $y = x \cdot q + r$ остаток $r = 0$. Поэтому $x|y \iff r = 0$. Следовательно, характеристическая функция данного предиката может быть представлена следующим образом:

$$\chi_{P_9}(x,y) = \begin{cases} 1, & \text{если} \quad r = 0, \\ 0, & \text{если} \quad r \neq 0. \end{cases}$$

Тогда ясно, что её можно следующим образом выразить через известные нам примитивно рекурсивные функции:

$$\chi_{P_9}(x,y) = \overline{sg}(r(x,y)) .$$

Следовательно, функция $\chi_{P_9}(x,y)$ примитивно рекурсивна, а характеризуемый ей предикат делимости $P_9(x,y)$ примитивно рекурсивен.

Пример 3.3.11. *Предикат делимости* $P_{10}(x,y)$: "$x \quad y$" ("x делится на y"). Это означает, что найдётся такое натуральное q, что $x = y \cdot q$. Другими словами, в равенстве при делении с остатком $x = y \cdot q + r$ остаток $r = 0$. Его характеристическую функцию можно следующим образом выразить через известные нам примитивно рекурсивные функции:

$$\chi_{P_{10}}(x,y) = \overline{sg}(r(x,y)) \cdot sg(y) + \overline{sg}(x) \cdot \overline{sg}(y) .$$

Следовательно, предикат делимости $P_{10}(x,y)$ примитивно рекурсивен.

Пример 3.3.12. *Предикат равенства примитивно рекурсивной функции* $P_{11}(x_1, x_2, ..., x_n, y)$: "$y = \varphi(x_1, x_2, ..., x_n)$", где $\varphi(x_1, x_2, ..., x_n)$ – некоторая заданная примитивно рекурсивная функция. Характеристическую функцию этого предиката можно следующим образом выразить через известные нам примитивно рекурсивные функции: $\chi_{P_{11}}(x_1, x_2, ..., x_n, y) = \overline{sg}|y - \varphi(x_1, x_2, ..., x_n)|$. Следовательно, предикат $P_{11}(x_1, x_2, ..., x_n, y)$ примитивно рекурсивен.

Логические операции с примитивно рекурсивными предикатами. Теорема 3.3.13. *Если* $P(x_1, x_2, ..., x_n)$ *и* $Q(x_1, x_2, ..., x_n)$ *– примитивно рекурсивные предикаты, то и следующие предикаты примитивно рекурсивны:* $\neg P(x_1, x_2, ..., x_n)$, $P(x_1, x_2, ..., x_n) \wedge Q(x_1, x_2, ..., x_n)$, $P(x_1, x_2, ..., x_n) \vee Q(x_1, x_2, ..., x_n)$, $P(x_1, x_2, ..., x_n) \to Q(x_1, x_2, ..., x_n)$.

Доказательство. По условию, характеристические функции χ_P и χ_Q примитивно рекурсивны. Характеристические функции новых предикатов следующим образом выражаются через функции χ_P и χ_Q , что снова приводит к примитивно рекурсивным функциям:

$$\chi_{\neg P}(x_1, x_2, ..., x_n) = \overline{sg}\chi_P(x_1, x_2, ..., x_n) ;$$

$$\chi_{P\wedge Q}(x_1, x_2, ..., x_n) = \chi_P(x_1, x_2, ..., x_n) \cdot \chi_Q(x_1, x_2, ..., x_n) ;$$

$$\chi_{P\vee Q}(x_1, x_2, ..., x_n) = \max(\chi_P(x_1, x_2, ..., x_n), \chi_Q(x_1, x_2, ..., x_n)) =$$

$$= sg(\chi_P(x_1, x_2, ..., x_n) + \chi_Q(x_1, x_2, ..., x_n)) ;$$

$$\chi_{P\to Q} = \chi_{\neg P\vee Q} = sg(\chi_{\neg P} + \chi_Q) = sg(\overline{sg}\chi_P + \chi_Q) . \quad \square$$

Ограниченные кванторы общности и существования. Пусть дан $(n+1)$-местный предикат $P(x_1, x_2, ..., x_n, y)$, заданный на множестве N. Сопоставим ему следующие $(n+1)$-местные предикаты от $x_1, x_2, ..., x_n, z$:

$$(\forall y \le z)(P(x_1, x_2, ..., x_n, y)) =$$

$$= P(x_1, x_2, ..., x_n, 0) \wedge P(x_1, x_2, ..., x_n, 1) \wedge \ldots \wedge P(x_1, x_2, ..., x_n, z)$$

и

$$(\exists y \le z)(P(x_1, x_2, ..., x_n, y)) =$$

$$= P(x_1, x_2, ..., x_n, 0) \vee P(x_1, x_2, ..., x_n, 1) \vee \ldots \vee P(x_1, x_2, ..., x_n, z).$$

Первый предикат обозначается $Q(x_1, x_2, ..., x_n, z)$ и про него говорят, что он получен из исходного предиката $P(x_1, x_2, ..., x_n, y)$ в результате применения к нему операции навешивания *ограниченного квантора общности.*

Второй предикат обозначается $R(x_1, x_2, ..., x_n, z)$ и про него говорят, что он получен из исходного предиката $P(x_1, x_2, ..., x_n, y)$ в результате применения к нему операции навешивания *ограниченного квантора существования.*

Пример 3.3.14. Рассмотрим предикат $P(x, y)$: $x + y = 5$, заданный на множестве натуральных чисел N. Применим к нему операции навешивания ограниченного квантора общности и ограниченного квантора существования. Получим соответственно предикаты:

$$Q(x, z) = (\forall y \leq z)(x + y = 5) \quad \text{и} \quad R(x, z) = (\exists y \leq z)(x + y = 5).$$

Легко проверить, что $Q(4, 2) =$ Л (Ложь), $Q(5, 0) =$ И (Истина), $R(1, 3) =$ Л (Ложь), $R(4, 2) =$ И (Истина).

Теорема 3.3.15. *Если предикат $P(x_1, x_2, ..., x_n, y)$ примитивно рекурсивен, то и предикаты $Q(x_1, x_2, ..., x_n, z)$ и $R(x_1, x_2, ..., x_n, z)$, получаемые из $P(x_1, x_2, ..., x_n, y)$ в результате навешивания ограниченных кванторов общности и существования, также примитивно рекурсивны.*

Доказательство. **1)** Рассмотрим сначала *ограниченный квантор общности*. Покажем, что характеристическая функция χ_Q предиката $Q(x_1, x_2, ..., x_n, z)$ имеет следующий вид:

$$\chi_Q(x_1, x_2, ..., x_n, z) = \Pi_{y=0}^{z} \chi_P(x_1, x_2, ..., x_n, y) . \qquad (*)$$

В самом деле, возьмём произвольный набор $(a_1, a_2, ..., a_n, b)$ натуральных чисел. Для предиката $Q(x_1, x_2, ..., x_n, z)$ на нём могут представиться лишь следующие две возможности:

а) $Q(a_1, a_2, ..., a_n, b) =$ И (Истина). Тогда по определению ограниченного квантора общности, это означает, что

$$P(a_1, a_2, ..., a_n, 0) = \text{И}, P(a_1, a_2, ..., a_n, 1) = \text{И}, ... , P(a_1, a_2, ..., a_n, b) = \text{И}.$$

Тогда с одной стороны равенства $(*)$: $\chi_Q(a_1, a_2, ..., a_n, b) = 1, \quad (1)$

а с другой: $\chi_P(a_1, a_2, ..., a_n, 0) = 1$,

$$\chi_P(a_1, a_2, ..., a_n, 1) = 1 , \ ... \ , \chi_P(a_1, a_2, ..., a_n, b) = 1,$$

и значит, $\Pi_{y=0}^{b} \chi_P(a_1, a_2, ..., a_n, y) = 1 . \qquad (2)$

Следовательно, из (1) и (2) видно, что равенство $(*)$ выполняется при $x_1 = a_1, \ x_2 = a_2, \ ... \ , \ x_n = a_n, \ z = b$.

б) $Q(a_1, a_2, ..., a_n, b) =$ Л (Ложь). По определению ограниченного квантора общности это означает, что найдётся такой b_0: $0 \leq b_0 \leq b$, что $P(a_1, a_2, ..., a_n, b_0) =$Л. Тогда с одной стороны:

$$\chi_Q(a_1, a_2, ..., a_n, b) = 0, \qquad (3)$$

а с другой: $\chi_P(a_1, a_2, ..., a_n, b_0) = 0, \quad (0 \leq b_0 \leq b),$ и значит,

$$\Pi_{y=0}^{b} \chi_P(a_1, a_2, ..., a_n, y) = 0 . \qquad (4)$$

Следовательно, из (3) и (4) видно, что равенство $(*)$ снова выполняется при $x_1 = a_1, \ x_2 = a_2, \ ... \ , \ x_n = a_n, \ z = b$.

2) Аналогично доказывается, что характеристическая функция χ_R предиката $R(x_1, x_2, ..., x_n, z)$, получающегося из предиката

$P(x_1, x_2, ..., x_n, y)$ навешиванием *ограниченного квантора существования*, имеет вид:

$$\chi_R(x_1, x_2, ..., x_n, z) = sg(\Sigma_{y=0}^{z} \chi_P(x_1, x_2, ..., x_n, y)) . \qquad (**)$$

Из равенств $(*)$ и $(**)$ видно, что обе функции χ_Q и χ_R являются суперпозициями примитивно рекурсивных функций; кроме того, операторы конечного суммирования и конечного мультиплицирования сохраняют примитивную рекурсивность функций (теорема 3.2.15). Значит, функции χ_Q и χ_R примитивно рекурсивны и вместе с ними примитивно рекурсивны предикаты $Q(x_1, x_2, ..., x_n, z)$ и $R(x_1, x_2, ..., x_n, z)$. □

Замечание 3.3.16. Ограниченные кванторы общности $(\forall y \leq z)$ и существования $(\exists y \leq z)$ можно рассматривать в более общем виде, считая, что z, в свою очередь, является некоторой функцией: $z = \varphi(x_1, x_2, ..., x_n)$.

Очевидно, что в предположении примитивной рекурсивности функции $\varphi(x_1, x_2, ..., x_n)$ доказанная теорема сохраняет свою силу: если предикат $P(x_1, x_2, ..., x_n, y)$ примитивно рекурсивен, то примитивно рекурсивны и предикаты:

$$(\forall y \leq \varphi(x_1, x_2, ..., x_n))(P(x_1, x_2, ..., x_n, y))$$

и

$$(\exists y \leq \varphi(x_1, x_2, ..., x_n))(P(x_1, x_2, ..., x_n, y)).$$

Примеры примитивно рекурсивных предикатов. Доказанная теорема и сделанное замечание позволяют установить примитивную рекурсивность ещё ряда предикатов.

Пример 3.3.17. $P_{12}(x, y, z)$: "$z =$ НОД (x, y)".

Этот предикат можно представить в виде конъюнкции примитивно рекурсивных предикатов:

$$z = \text{НОД } (x, y) \iff$$
$$(z|x) \wedge (z|y) \wedge (\forall t \leq \min(x, y))[((t|x) \wedge (t|y)) \rightarrow (t|z)].$$

Первые два члена этой конъюнкции примитивно рекурсивны ввиду примера 3.3.10. Третий член конъюнкции есть примитивно рекурсивный предикат на основании предыдущей теоремы 3.3.15 ввиду примитивной рекурсивности предикатов "$t|x$", "$t|y$", "$t|z$" (пример 3.3.10) и ввиду сохранения этого свойства предикатов логическими операциями $\wedge$ и $\rightarrow$ над ними (теорема 3.3.13). Наконец, конъюнкция трёх примитивно рекурсивных предикатов снова даёт примитивно рекурсивный предикат (теорема 3.3.13).

Пример 3.3.18. $P_{13}(x, y)$: "*x и y взаимно просты*".

Этот предикат можно представить с помощью ограниченного квантора общности, применённого к примитивно рекурсивному предикату:

$$"x \text{ и } y \text{ взаимно просты}" \iff$$
$$(z|x) \wedge (z|y) \wedge (\forall z \leq \min(x,y))[((z|x) \wedge (z|y)) \rightarrow z = 1].$$

Пример 3.3.19. $P_{14}(x)$: "*x – простое число)*".

Этот предикат также можно представить в виде конъюнкции примитивно рекурсивных предикатов, один из которых, в свою очередь, представлен с помощью ограниченного квантора общности, применённого к примитивно рекурсивному предикату:

$$"x \text{ – простое число}" \iff$$
$$\neg(x = 1) \wedge (\forall y \leq x)[(y|x) \rightarrow (y = 1 \vee y = x].$$

Предикат $P_{14}(x)$ будем обозначать $Pr(x)$.

Оператор условного перехода. (Кусочное задание функций). Вернёмся снова к примитивно рекурсивным *функциям* и отметим ещё одно важное свойство, связанное с примитивной рекурсивностью функций. Мы используем его в последующих рассмотрениях.

Одним из часто встречающихся, особенно в теории алгоритмов, способов задания функций является их задание с помощью так называемого ***оператора условного перехода.*** Этот оператор по функциям $f_1(x_1, x_2, ..., x_n)$, $f_2(x_1, x_2, ..., x_n)$ и предикату $P(x_1, x_2, ..., x_n)$ строит функцию φ:

$$\varphi(x_1, ..., x_n) =$$
$$= \begin{cases} f_1(x_1, ..., x_n), & \text{если высказывание } P(x_1, ..., x_n) \text{ – истинно,} \\ f_2(x_1, ..., x_n), & \text{если высказывание } P(x_1, ..., x_n) \text{ – ложно.} \end{cases}$$

Нетрудно понять, что с помощью характеристических функций предиката P и его отрицания $\neg P$ функцию φ можно выразить следующим образом:

$$\varphi(x_1, x_2, ..., x_n) =$$
$$= f_1(x_1, x_2, ..., x_n) \cdot \chi_P(x_1, x_2, ..., x_n) + f_2(x_1, x_2, ..., x_n) \cdot \chi_{\neg P}(x_1, x_2, ..., x_n).$$

Из этого выражения вытекает следующее утверждение.

Теорема 3.3.20. *Если функции f_1, f_2 и предикат P примитивно рекурсивны, то и функция φ, построенная из f_1, f_2, P с помощью оператора условного перехода, примитивно рекурсивна. Этот факт выражают также, говоря, что оператор условного перехода примитивно рекурсивен.* □

Оператор условного перехода может иметь и более общую форму, когда переход носит не двузначный, и многозначный характер. Пусть дана конечная совокупность функций

$f_1(x_1, x_2, ..., x_n)$, ... , $f_m(x_1, x_2, ..., x_n)$, $f_{m+1}(x_1, x_2, ..., x_n)$,
и конечная совокупность предикатов

$P_1(x_1, x_2, ..., x_n)$, ... , $P_m(x_1, x_2, ..., x_n)$, $(n, m \in N,\ n \geq 1)$,
причём, области истинности предикатов попарно не пересекаются: $P_i^+ \cap P_j^+ = \emptyset$, $i, j = 1, 2, ..., m$. (См. Учебник МЛ, §18, определение 18.3).

Определение 3.3.21. Говорят, что функция $f(x_1, ..., x_n)$ задана *кусочным образом* указанной системой функций и предикатов, если она удовлетворяет следующим условиям:

$$f(x_1, ..., x_n) = \begin{cases} f_1(x_1, ..., x_n), & \text{если } P_1(x_1, ..., x_n) = \text{И}, \\ f_2(x_1, ..., x_n), & \text{если } P_2(x_1, ..., x_n) = \text{И}, \\ \dots\dots\dots & \dots\dots \\ f_m(x_1, ..., x_n), & \text{если } P_m(x_1, ..., x_n) = \text{И}, \\ f_{m+1}(x_1, ..., x_n), & \text{во всех остальных случаях}. \end{cases}$$

Теорема 3.3.22. *Если все функции $f_1, f_2, ..., f_m, f_{m+1}$ и все предикаты $P_1, P_2, ..., P_m$ примитивно рекурсивны, то примитивно рекурсивна и функция f, задаваемая кусочным образом этой системой функций и предикатов.*

Доказательство. Функцию f можно следующим образом представить в виде суперпозиции примитивно рекурсивных функций:

$$f(x_1, ..., x_n) =$$

$$= f_1(x_1, ..., x_n) \cdot \chi_{P_1}(x_1, ..., x_n) + ... + f_m(x_1, ..., x_n) \cdot \chi_{P_m}(x_1, ..., x_n) +$$

$$+ f_1(x_1, ..., x_n) \cdot \overline{sg}(\chi_{P_1}(x_1, ..., x_n) + ... + \chi_{P_m}(x_1, ..., x_n)) .$$

Следовательно, $f(x_1, x_2, ..., x_n)$ – примитивно рекурсивная функция. □

Замечание 3.3.23. В предыдущем определении можно считать, что имеется лишь m функций $f_1, f_2, ..., f_m$ и m предикатов $P_1, P_2, ..., P_m$ таких, что дизъюнкция

$$P_1(x_1, x_2, ..., x_n) \vee P_2(x_1, x_2, ..., x_n) \vee ... \vee P_m(x_1, x_2, ..., x_n)$$
есть тождественно истинный предикат, а все попарные конъюнкции $P_1(x_1, x_2, ..., x_n) \wedge P_2(x_1, x_2, ..., x_n)$ $(i \neq j)$ являются тождественно ложными предикатами. Это означает, что из данных предикатов P_1, ..., P_m, всегда истинен один и только один предикат, или что данной системой предикатов исчерпываются все возможные случаи. В этом случае функция f представляется следующим образом в виде суперпозиции примитивно рекурсивных функций:

$$f(x_1, x_2, ..., x_n) = f_1(x_1, x_2, ..., x_n) \cdot \chi_{P_1}(x_1, x_2, ..., x_n) + \ldots$$
$$\ldots + f_m(x_1, x_2, ..., x_n) \cdot \chi_{P_m}(x_1, x_2, ..., x_n) ,$$

и следовательно, также является примитивно рекурсивной.

Замечание 3.3.24. Последнее замечание позволяет доказать *примитивную рекурсивность функций, заданных на конечных множествах*, ибо все их можно задать с помощью оператора условного перехода. Правда, при этом их придётся доопределить с помощью какой-либо константы на всех остальных натуральных числах, на которых она не определена.

Например, функцию φ, определённую на множестве $\{1,3,9,17\}$ равенствами $\varphi(1) = 8$, $\varphi(3) = 0$, $\varphi(9) = 4$, $\varphi(17) = 6$, с помощью оператора условного перехода можно описать так:

$$\varphi(x) = \begin{cases} 8, & \text{если } x = 1, \\ 0, & \text{если } x = 3, \\ 4, & \text{если } x = 9, \\ 6 & \text{в остальных случаях}. \end{cases}$$

Таким образом, $\varphi(x) = 6$ вне исходной области задания.

Оператор ограниченной минимизации. (Ограниченный μ-оператор). Пусть $P(x_1, ..., x_n, y)$ - всюду определённый $(n+1)$-местный предикат над N.

Определение 3.3.25. Говорят, что функция $f(x_1, ..., x_n)$ получена из предиката $P(x_1, ..., x_n, y)$ и функции $\varphi(x_1, ..., x_n)$ в результате применения *оператора ограниченной минимизации*, если для любых $a_1, ..., a_n \in N$:

$$f(a_1, ..., a_n) = \begin{cases} \text{наименьшему из таких чисел } y, \text{ заключённых} \\ \text{в промежутке } 0 \leq y \leq \varphi(a_1, ..., a_n), \text{что} \\ P(a_1, ..., a_n, y) = \text{И, если такое } y \text{ существует;} \\ 0 \text{ , если такое } y \text{ не существует, т.е. если} \\ P(a_1, ..., a_n, y) = \text{Л для всех } y : \\ 0 \leq y \leq \varphi(a_1, ..., a_n). \end{cases}$$

Обозначение: $f(x_1,...,x_n) = \underset{y \le \varphi(x_1,...,x_n)}{\mu\ y} [P(x_1,...,x_n,y)]$.

Теорема 3.3.26. *Если предикат* $P(x_1,...,x_n,y)$ *и функция* $\varphi(x_1,...,x_n)$ *примитивно рекурсивны, то примитивно рекурсивна и функция*

$$f(x_1,...,x_n) = \underset{y \le \varphi(x_1,...,x_n)}{\mu\ y} [P(x_1,...,x_n,y)] .$$

Доказательство. Покажем, что справедливо выражение:

$$\underset{y \le \varphi(x_1,...,x_n)}{\mu\ y} [P(x_1,...,x_n,y)] = \quad (*)$$

$$= [\Sigma_{j=0}^{\varphi(x_1,...,x_n)} \Pi_{i=0}^{j} \overline{sg}(\chi_P(x_1,...,x_n,i))] \cdot sg(\Sigma_{k=0}^{\varphi(x_1,...,x_n)} \chi_P(x_1,...,x_n,i)).$$

Второй сомножитель равен 1, если $P(x_1,...,x_n,y)$ истинно хотя бы для одного $k \le \varphi(x_1,...,x_n)$ и равно 0 в противном случае. Значит, он обеспечивает выполнимость данного равенства во втором случае определения функции $\mu\ y$, т.е. когда $P(x_1,...,x_n,y) =$ Л для всех y: $0 \le y \le \varphi(x_1,...,x_n)$.

Пусть теперь на промежутке $[0;\ \varphi(x_1,...,x_n)]$ имеются такие y, что $P(x_1,...,x_n,y) =$ И. Тогда произведение $\Pi_{i=0}^{j} \overline{sg}(\chi_P(x_1,...,x_n,i))$ равно 1 для всех j, меньших наименьшего y, для которого $P(x_1,...,x_n,y) =$ И, и оно равно 0 для всех j больших или равных этому y. Затем все эти единицы суммируются и в сумме дают то наименьшее число y, для которого $P(x_1,...,x_n,y) =$ И.

Равенство (*) доказано. Теперь, пользуясь тем, что функции χ_P, sg, $\overline{sg}$ примитивно рекурсивны (примеры 3.2.7 и 3.2.8) и тем, что операторы конечного суммирования и мультиплицирования сохраняют примитивную рекурсивность (т-ма 3.2.15), заключаем, что функция

$$\underset{y \le \varphi(x_1,...,x_n)}{\mu\ y} [P(x_1,...,x_n,y)]$$

примитивно рекурсивна. □

Дальнейшие примеры примитивно рекурсивных функций. Пользуясь ограниченным μ-оператором, можно доказать примитивную рекурсивность ещё ряда функций.

Пример 3.3.27. Функция НОД$(x,y) = \underset{z \le \min(x,y)}{\mu\ z} [P_{12}(x,y,z)]$.

Напомним (см. пример 3.3.17), что предикат $P_{12}(x, y, z)$ есть следующий "$z = \text{НОД}(x, y)$" и в примере 3.3.17 доказана его примитивная рекурсивность. Тогда на основании предыдущей теоремы ввиду того, что ещё и функция $\min(x, y)$ также примитивно рекурсивна (см. пример 3.2.10), заключаем, что данная функция $\text{НОД}(x, y)$ примитивно рекурсивна.

ПРИМЕР 3.3.28. Функция p_x – простое число, имеющее номер x в последовательности простых чисел, расположенных в порядке возрастания (x-ое простое число); причём, $p_0 = 2$. Можно дать следующее примитивно рекурсивное описание этой функции:

$$p_0 = 2,$$
$$p_{x+1} = \mu_{y \le 2^{2^{x+1}}} y \; [y > p_x \;\wedge\; Pr(y)] \,.$$

При выборе ограничивающей функции $y \le 2^{2^{x+1}}$ воспользовались известным из теории чисел свойством: $p_x \le 2^{2^{x+1}}$.

Обоснуйте самостоятельно примитивную рекурсивность данной функции, сославшись на все необходимые примеры и теоремы.

ПРИМЕР 3.3.29. Функция $\exp(x, y)$ – показатель степени, с которым простое число p_y (см. пример 3.3.28) входит в каноническое разложение числа x на простые множители. [Например, поскольку $60 = 2^2 \cdot 3 \cdot 5$, поэтому $\exp(60, 0) = 2$, $\exp(60, 1) = 1$, $\exp(60, 2) = 1$, $\exp(60, 3) = 0$ и т.д.

Докажите, что справедливо следующее выражение этой функции через ограниченный μ-оператор и обоснуйте примитивную рекурсивность данной функции, сославшись на все необходимые примеры и теоремы:

$$\exp(x, y) = \mu_{z \le x} z \; [\, \neg(p_y^{z+1} \mid x) \,] \,.$$

В заключение обсуждения примитивно рекурсивных функций сделаем ещё два важных замечания.

Во-первых, все примитивно рекурсивные функции всюду определены. Это следует из того, что простейшие функции всюду определены и операторы суперпозиции и примитивной рекурсии это свойство сохраняют. (Этот факт мы уже отмечали выше).

Во-вторых, строго говоря, мы имеем дело не с функциями, а с их примитивно рекурсивными описаниями. Различие здесь точно такое

же, как и различие между булевыми функциями и их представлениями в виде формул. Примитивно рекурсивные описания также разбиваются на классы эквивалентности: в один класс входят все описания, задающие одну и ту же функцию. Но задача распознавания эквивалентности примитивно рекурсивных описаний явит ещё один пример алгоритмически неразрешимой задачи (об этом речь пойдёт ниже, в конце §7.2.

3.4. Примитивно рекурсивные функции и функции, вычислимые по Тьюрингу

Теперь мы готовы к тому, чтобы сделать ещё один шаг на пути (в каком-то смысле аксиоматического) описания всех функций, вычислимых с помощью машины Тьюринга. Мы докажем, что всякая примитивно рекурсивная функция вычислима с помощью машины Тьюринга, т.е. что класс примитивно рекурсивных функций включается в класс функций, вычислимых по Тьюрингу. Тем не менее, это включение окажется собственным, т.е. эти классы окажутся не совпадающими.

Вычислимость по Тьюрингу примитивно рекурсивных функций. Докажем сначала следующую теорему.

Теорема 3.4.1. *Функция φ, возникающая примитивной рекурсией из правильно вычислимых на машине Тьюринга функций f и g, сама правильно вычислима на машине Тьюринга.*

Доказательство. Для краткости записей будем считать, что функция φ связана с функциями f и g зависимостью: $\varphi(x,0) = f(x)$, $\varphi(x,i+1) = g(x,\varphi(x,i))$. Обозначим F и G – машины Тьюринга, правильно вычисляющие функции f и g соответственно. Пусть x, y – произвольные натуральные числа. Требуется сконструировать машину Тьюринга, вычисляющую значение $\varphi(x,y)$. Как мы уже отмечали, для вычисления $\varphi(x,y)$ предстоит вычислить $y+1$ значений $\varphi(x,0)$, $\varphi(x,1)$, ... , $\varphi(x,y-1)$, $\varphi(x,y)$.

Начальная конфигурация такова $q_1 01^x 01^y 0$. Применим к ней следующую последовательность машин Тьюринга: **Б$^+$ВГВБ$^+$F**. В результате получим последовательность конфигураций:

$$q_1 01^x 01^y 0 \overset{\text{Б}^+}{\Longrightarrow} 01^x q 01^y 0 \overset{\text{В}}{\Longrightarrow} 01^y q 01^x 0 \overset{\text{Г}}{\Longrightarrow} 01^y q 01^x 01^x 0 \overset{\text{В}}{\Longrightarrow}$$

$$\overset{\text{В}}{\Longrightarrow} 01^x q 01^y 01^x 0 \overset{\text{Б}^+}{\Longrightarrow} 01^x 01^y q 01^x 0 \overset{F}{\Longrightarrow} 01^x 01^y q_\alpha 01^{\varphi(x,0)} 0\,.$$

На последнем шаге, применив машину, вычисляющую функцию $f(x)$, к конфигурации $q01^x$, мы получим значение $f(x)$, которое, согласно схемы примитивной рекурсии для φ, есть $\varphi(x,0)$. (Промежуточным состояниям q мы намеренно не приписывали никаких индексов. Индекс α получило лишь последнее в этой последовательности состояние).

Теперь мы можем приступить к вычислению $\varphi(x,1)$, используя второе соотношение схемы примитивной рекурсии: $\varphi(x,1) = g(x,\varphi(x,0))$. Для этого применим сначала к последней конфигурации команды: $q_\alpha 0 \to q_{\alpha+1}0Л$, $q_{\alpha+1} \to q_{\alpha+2}0$. В результате получим конфигурацию: $01^x01^{y-1}q_{\alpha+2}001^{\varphi(x,0)}0$. Теперь нужно подготовить ленту машины к применению машины G, вычисляющей значение $g(x,\varphi(x,0))$, то есть необходимо получить на ленте конфигурацию $q01^x01^{\varphi(x,0)}$. Для этого применим к последней конфигурации последовательность машин **АБ$^-$ВБ$^+$ВГВБ$^-$ВБ$^+$Б$^+$ВБ$^-$** (слева направо). Получим последовательность конфигураций:

$$01^x01^{y-1}q_{\alpha+2}001^{\varphi(x,0)}0 \overset{\text{А}}{\Longrightarrow} 01^x01^{y-1}q01^{\varphi(x,0)}00 \overset{\text{Б}^-}{\Longrightarrow}$$

$$\overset{\text{Б}^-}{\Longrightarrow} 01^xq01^{y-1}01^{\varphi(x,0)} \overset{\text{В}}{\Longrightarrow} 01^{y-1}q01^x01^{\varphi(x,0)} \overset{\text{Б}^+}{\Longrightarrow} 01^{y-1}01^xq01^{\varphi(x,0)} \overset{\text{В}}{\Longrightarrow}$$

$$\overset{\text{В}}{\Longrightarrow} 01^{y-1}01^{\varphi(x,0)}q01^x \overset{\text{Г}}{\Longrightarrow} 01^{y-1}01^{\varphi(x,0)}q01^x01^x \overset{\text{В}}{\Longrightarrow}$$

$$\overset{\text{В}}{\Longrightarrow} 01^{y-1}01^xq01^{\varphi(x,0)}01^x \overset{\text{Б}^-}{\Longrightarrow} 01^{y-1}q01^x01^{\varphi(x,0)}01^x \overset{\text{В}}{\Longrightarrow}$$

$$\overset{\text{В}}{\Longrightarrow} 01^xq01^{y-1}01^{\varphi(x,0)}01^x \overset{\text{Б}^+}{\Longrightarrow} 01^x01^{y-1}q01\varphi(x,0)01^x \overset{\text{Б}^+}{\Longrightarrow}$$

$$\overset{\text{Б}^+}{\Longrightarrow} 01^x01^{y-1}01\varphi(x,0)q01^x \overset{\text{В}}{\Longrightarrow} 01^x01^{y-1}01^xq01^{\varphi(x,0)} \overset{\text{Б}^-}{\Longrightarrow}$$

$$\overset{\text{Б}^-}{\Longrightarrow} 01^x01^{y-1}q01^x01^{\varphi(x,0)}\,.$$

Теперь мы можем применить машину G и вычислить значение

$$\varphi(x,1) = g(x,\varphi(x,0)): \qquad 01^x01^{y-1}q_\beta01^{\varphi(x,0)}.$$

Применим к этой конфигурации команду: $q_\beta 0 \to q_\alpha 0$, зацикливающую программу. Получим конфигурацию: $01^x01^{y-1}q_\alpha01^{\varphi(x,1)}$.

Начинается следующий цикл, осуществляемый командами, идущими после первого появления состояния q_α. Этот цикл преобразует конфигурацию вида $01^x01^{y-i}q_\alpha01^{\varphi(x,i)}$ в конфигурацию $01^x01^{y-(i+1)}q_\beta01^{\varphi(x,i+1)}$ при условии, что $y > i$. Команда $q_\beta 0 \to q\alpha 0$ зацикливает программу, и в результате работы цикла параметр $y-i$

будет понижаться до тех пор, пока не получится конфигурация: $01^x0q_\alpha01^{\varphi(x,y)}$, которая в силу команды $q_\alpha 0 \to q_{\alpha+1}0Л$ перейдёт в конфигурацию $01^x q_{\alpha+1}001^{\varphi(x,y)}$.

Дополнительные команды $q_{\alpha+1}0 \to q_{\beta+1}0$, **А**, **В**, **О**, **В**, **В**$^-$, **А** создают на ленте требуемую конфигурацию $q_0 01^{\varphi(x,y)}$, доказывающую, что функция $\varphi(x,y)$ правильно вычислена на машине Тьюринга. □

СЛЕДСТВИЕ 3.4.2. *Всякая примитивно рекурсивная функция вычислима по Тьюрингу, т.е. класс примитивно рекурсивных функций включается в класс функций, вычислимых по Тьюрингу.*

ПРФ $\subset$ Выч.Т

Доказательство. Это следует ввиду определения 3.1.6. примитивно рекурсивной функции, из вычислимости по Тьюрингу простейших функций S, O, I^n_m (примеры 2.2.1, 2.4.4, 2.4.5) и свойств сохранения такой вычислимости операторами суперпозиции (теорема 2.4.7) и примитивной рекурсии (теорема 3.4.1). □

Функции Аккермана. Напомним, что мы поставили задачу охарактеризовать вычислимые (с помощью какого-либо алгоритма) функции. Учитывая тезис Тьюринга (конец параграфа 2.4, стр. 27), под алгоритмом достаточно понимать машину Тьюринга. После того, как в предыдущем пункте было доказано, что всякая примитивно рекурсивная функция вычислима (по Тьюрингу), возникает обратный вопрос: исчерпывается ли класс вычислимых (по Тьюрингу) функций примитивно рекурсивными функциями, то есть всякая ли вычислимая (по Тьюрингу) функция будет непременно примитивно рекурсивной?

Применив теоретико-множественные мощностные соображения[1], достаточно легко ответить отрицательно на более общий вопрос – исчерпывается ли класс всех функций примитивно рекурсивными функциями, то есть все ли функции являются примитивно рекурсивными?

ТЕОРЕМА 3.4.3. *Существуют функции, не являющиеся примитивно рекурсивными.*

Доказательство. В самом деле, нетрудно понять, что множество всех примитивно рекурсивных функций счётно. Это объясняется

[1] О теории мощностей множеств см., например, книгу: *Игошин В.И.* Математическая логика как педагогика математики. – Саратов: Издательский центр "Наука", 2009. – 360 с. (§ 8.5.4)

тем, что каждая примитивно рекурсивная функция имеет конечное описание, то есть задаётся конечным словом в некотором (фиксированном для всех функций) алфавите. Множество всех конечных слов счётно. Поэтому и примитивно рекурсивных функций имеется не более, чем счётное множество. В то же время, множество всех функций даже от одного аргумента из N в N имеет мощность континуума. Тем более, континуально множество функций из N в N от любого конечного числа аргументов. Таким образом, непременно существует функция, не являющаяся примитивно рекурсивной. □

Отрицательным будет также ответ и на более узкий вопрос, поставленный в предыдущем пункте: все ли вычислимые (а в силу тезиса Тьюринга – вычислимые по Тьюрингу) функции можно описать как примитивно рекурсивные? Чтобы это установить, необходимо привести пример вычислимой функции, не являющейся примитивно рекурсивной. Идея примера состоит в том, чтобы построить такую вычислимую функцию, которая обладала бы свойством, каким не обладает ни одна примитивно рекурсивная функция. Таким свойством может служить скорость роста функции. Итак, необходимо указать такую функцию, которая растёт быстрее любой примитивно рекурсивной функции и поэтому примитивно рекурсивной не является.

Пример 3.4.4. Искомая функция конструируется с помощью последовательности вычислимых функций, в которой каждая функция растёт существенно быстрее предыдущей.

Начнём с построения такой последовательности. Мы знаем, что произведение растёт быстрее суммы, а степень – быстрее произведения. Начнём построение последовательности именно с этих функций, введя для них следующие единообразные обозначения:

$$P_0(a,y) \;=\; a+y\;, \quad P_1(a,y) \;=\; a\cdot y\;, \quad P_2(a,y) \;=\; a^y\;.$$

Эти функции связаны между собой следующими рекурсивными соотношениями:

$$P_1(a,y+1) \;=\; a+ay \;=\; P_0(a,P_1(a,y)), \quad P_1(a,1) \;=\; a\;;$$
$$P_2(a,y+1) \;=\; a\cdot a^y \;=\; P_1(a,P_2(a,y)), \quad P_2(a,1) \;=\; a\;.$$

Продолжим эту последовательность, положив по определению

для $n = 2, 3, \ldots$

$$\left.\begin{aligned} P_{n+1}(a,0) &= 1, \\ P_{n+1}(a,1) &= a, \\ P_{n+1}(a,y+1) &= P_n(a, P_{n+1}(a,y)). \end{aligned}\right\} \qquad (*)$$

(Первое из этих равенств предназначено для того, чтобы функции $P_n(a,y)$ были всюду определены.) Эта схема имеет вид примитивной рекурсии и, следовательно, все функции $P_n(a,y)$ примитивно рекурсивны. Растут они крайне быстро. Например, $P_3(a,1) = a$, $P_3(a,2) = P_2(a,a) = a^a$, ... , значение $P_3(a,k)$ получается в результате возведения числа a в степень a k раз.

Зафиксируем теперь значение $a = 2$. Получим последовательность функций от одного аргумента: $P_0(2,y)$, $P_1(2,y)$, ... Введём две новые функции: $B(x,y) = P_x(2,y)$ (она перечисляет последнюю последовательность) и $A(x) = B(x,x)$. Первая из них называется ***функцией Аккермана***, а вторая – ***диагональной функцией Аккермана.***

Для функции $B(x,y)$ из соотношений $(*)$ вытекают следующие тождества:

$$\left.\begin{aligned} B(0,y) &= 2+y, \\ B(x+1,0) &= sg\ x, \\ B(x+1,y+1) &= B(x, B(x+1,y)), \end{aligned}\right\} \qquad (**)$$

которые позволяют вычислять значение функции $B(x,y)$, и, следовательно, и значения функции $A(x)$. При этом, при вычислении значения функции $B(x,y)$ в некоторой точке используются вычисленные ранее её значения в неких предыдущих точках. Этим схема $(**)$ похожа на схему примитивной рекурсии. cdeНо примитивная рекурсия ведётся по одному аргументу, а в схеме $(**)$ рекурсия ведётся сразу по двум аргументам (такая рекурсия называется двойной, двукратной, или рекурсией 2-ой ступени). При этом существенно усложняется характер упорядочения точек, и, следовательно, и понятие предшествующей точки. Это упорядочение не предопределено заранее, как в схеме примитивной рекурсии, где число n всегда предшествует числу $n+1$, а выясняется в ходе вычислений и для каждой схемы $(**)$, вообще говоря, различно. Например, $B(3,3) = B(2, B(3,2))$, а так как $B(3,2) = P_3(2,2) = P_2(2, P_3(2,1)) =$

$P_2(2,2) = 2^2 = 4$, то вычислению B в точке (3,3) по схеме (**) должно предшествовать вычисление B в точке (2,4): $B(3,3) = B(2,4) = P_2(2,4) = 2^4 = 16$. Проверьте самостоятельно, что вычислению B в точке (3,4) должно предшествовать вычисление в точке (2,16).

Итак, функция $B(x,y)$, а вместе с ней и функция $A(x)$ вычислимы (по схеме (**)), а значит в силу гипотезы Тьюринга, вычислимы посредством подходящей машины Тьюринга. Возникает вопрос, можно ли их вычисление свести к вычислению по схеме примитивной рекурсии, то есть будут ли эти функции примитивно рекурсивными. Именно такова ситуация с функцией Аккермана $A(x)$. Идея доказательства того, что функция $A(x)$ не является примитивно рекурсивной, состоит в том, что доказывается, что функция $A(x)$ растёт быстрее, чем любая примитивно рекурсивная функция, и поэтому не может быть примитивно рекурсивной. Более точно, для любой примитивно рекурсивной функции $f()$ от одного аргумента найдётся такое n, что для любого $x \geq n$ будет $A(x) > f(x)$. (Это доказал Аккерман в 1928 г.)

Идея доказательства этого утверждения состоит в следующем. Сначала доказываются два свойства функции $B(x,y)$, которые используются в дальнейшем:

$$B(x,y+1) > B(x,y) \qquad (x,y = 1,2,...),$$

$$B(x+1,y) \geq B(x,y+1) \qquad (x \geq 1,\ y \geq 2).$$

Затем вводится понятие B-мажорируемой функции.

Функция $f(x_1,...,x_n)$ называется *B-мажорируемой*, если

$$(\exists m \in N)(\forall x_1,...,x_n)[\max(x_1,...,x_n) > 1 \rightarrow$$

$$\rightarrow f(x_1,...,x_n) < B(m,\max(x_1,...,x_n))].$$

Доказывается, что все примитивно рекурсивные функции B-мажорируемы. (Сначала устанавливается B-мажорируемость простейших функций $O, S(x), I^n_m(x)$. Затем доказывается, что оператор суперпозиции, примененный к B-мажорируемым функциям, даёт B-мажорируемую функцию. Аналогичным свойством обладает и оператор примитивной рекурсии).

Наконец, рассмотрим произвольную примитивно рекурсивную функцию $f(x)$. В силу предыдущего утверждения, для некоторого m и любого $x \geq 2$ имеем $f(x) < B(m,x)$. Но тогда: $A(m+x) =$

$B(m+x, m+x) > B(m, m+x) > f(m+x)$. (Предпоследнее неравенство получено, исходя из двух свойств функции $B(x,y)$, сформулированных вначале). Это и означает, что $A(x) > f(x)$, начиная по меньшей мере с $x = m+2$.

Итак, функция Аккермана $A(x)$ не может быть вычислена по схеме примитивной рекурсии, но может быть вычислена по более сложной схеме (**). Следовательно, примитивно рекурсивные функции не исчерпывают класса всех вычислимых функций, а свойство функции быть примитивно рекурсивной не равносильно её свойству быть вычислимой (в том числе по Тьюрингу). Это говорит о том, что если мы не оставляем затеи породить из простейших функций все вычислимые функции, то мы должны ввести какие-то дополнительные средства (методы) порождения. Таким средством явится оператор минимизации (или оператор наименьшего числа, или μ-оператор).

3.5. Частично рекурсивные функции и функции, вычислимые по Тьюрингу

Напомним (определение 3.1.8), что функция называется *частично рекурсивной* (чрф), если она может быть получена из простейших функций S, O, I^n_m с помощью конечного числа применений операторов суперпозиции, примитивной рекурсии и μ-оператора. Если функция всюду определена и частично рекурсивна, то она называется *общерекурсивной* (орф).

Обратимся ещё раз к μ-оператору, определение которого дано в §3.1 (определение 3.1.7).

Оператор минимизации. В более общем виде его можно определить, как оператор, применяемый к произвольному $(n+1)$-местному предикату $P(x_1, ..., x_n, y)$ и дающий в результате n-местную функцию $\varphi(x_1, ..., x_n)$. Значение $\varphi(a_1, ..., a_n)$ этой функции на наборе аргументов $a_1, ..., a_n$ равно наименьшему из таких чисел y, что высказывание $P(a_1, ..., a_n, y)$ истинно. Оно обозначает $\mu y P(a_1, ..., a_n, y)$. Если же наименьшего среди таких чисел не существует, то значение функции $\varphi(x_1, ..., x_n)$ на этом наборе $a_1, ..., a_n$ не определено. (Это означает, что оператор минимизации может породить частичную, т.е. не всюду определённую функцию). Таким образом, $\varphi(x_1, ..., x_n) =$

$\mu y P(x_1, ..., x_n, y)$. Оператор минимизации называется также *оператором наименьшего числа* или *μ-оператором.*

Предикат $P(x_1, ..., x_n, y)$ (заданный над N), к которому применяется оператор минимизации, может быть сконструирован из двух числовых функций следующим образом (как это и сделано в определении 3.1.7): "$f_1(x_1, ..., x_n, y) = f_2(x_1, ..., x_n, y)$", или из одной: "$f(x_1, ..., x_n, y) = 0$". Так что оператор минимизации можно рассматривать как оператор, заданный на множестве числовых функций и принимающий значения в нём же.

В этом своём качестве оператор минимизации является удобным средством для построения обратных функций. Действительно, функция $f^{-1}(x) = \mu y[f(y) = x] = \mu y[x - f(y) = 0]$ ("наименьший y такой, что $f(y) = x$") является обратной для функции $f(x)$. (Поэтому в применении к одноместным функциям оператор минимизации иногда называют *оператором обращения*).

Рассмотрим примеры действия оператора минимизации для получения обратных функций.

Пример 3.5.1. Рассмотрим следующую функцию, получающуюся с помощью оператора минимизации:

$$d(x, y) = \mu z[y + z = x] = \mu z\ [s(I_2^3(x, y, z), I_3^3(x, y, z)) = I_1^3(x, y, z)].$$

Вычислим, например, $d(7, 2)$. Для этого нужно положить $y = 2$ и, придавая переменной z последовательно значения 0, 1, 2, ..., каждый раз вычислять сумму $y + z$. Как только она станет равной 7, то соответствующее значение z принять за значение $d(7, 2)$. Вычисляем:

$z = 0, \quad 2 + 0 = 2 \neq 7;$
$z = 1, \quad 2 + 1 = 3 \neq 7;$
$z = 2, \quad 2 + 2 = 4 \neq 7;$
$z = 3, \quad 2 + 3 = 5 \neq 7;$
$z = 4, \quad 2 + 4 = 6 \neq 7;$
$z = 5, \quad 2 + 5 = 7.$

Таким образом, $d(7, 2) = 5$.

Попытаемся вычислить по этому правилу $d(3, 4)$:

$z = 0, \quad 4 + 0 = 4 > 3;$
$z = 1, \quad 4 + 1 = 5 > 3;$
$z = 2, \quad 4 + 2 = 6 > 3;$
. . . .

Видим, что данный процесс будет продолжаться бесконечно. Следовательно, $d(3, 4)$ не определено. Таким образом, $d(x, y) = x - y$.

Пример 3.5.2. Аналогично, с помощью оператора минимизации можно получить частичную функцию, выражающую частное от деления двух натуральных чисел:

$$x/y = q(x, y) = \mu z[y \cdot z = x] = \mu z[p(I_2^3(x, y, z), I_3^3(x, y, z)) = I_1^3(x, y, z)].$$

Заметим, что обе функции из двух последних примеров не всюду определены, т.е. являются частичными.

В Задачнике рассматриваются и другие функции, получаемые с помощью оператора минимизации (см. № 13.17).

С помощью μ-оператора можно построить некоторые всюду определённые аналоги рассмотренных функций:

$$[x - y] = \mu z[y + z \geq x] = \mu z[y + z + 1 > x],$$

$$[x/y] = \mu z[yz \geq x] = \mu z[y(z + 1) > x],$$

$$[\sqrt{x}] = \mu y[y^2 \geq x] = \mu y[(y + 1)^2 > x],$$

$$[\log_a x] = \mu y[a^y \geq x] = \mu y[a^{y+1} > x].$$

Первая из этих функций равна 0, если $x < y$, и равна $x - y$, если $x \geq y$. Вторая представляет собой целую часть функций x/y. Функции $\sqrt{x}$, $\log_a x$ соответственно.

Заметим, что механизм проявления неопределённости функций в точке при значении её с помощью μ-оператора такой же, как и при вычислении её на машине Тьюринга: в случае неопределённости процесс вычисления не останавливается, а продолжается неограниченно долго.

Общерекурсивные и частично рекурсивные функции. Итак (определение 3.1.8), функция называется *частично рекурсивной*, если она может быть построена из простейших функций O, S, I_m^n с помощью конечного числа применений операторов суперпозиции, примитивной рекурсии и μ-оператора. Если функция всюду определена и частично рекурсивна, то она называется *общерекурсивной.*

Мы уже отмечали, что класс частично рекурсивных функций шире класса примитивно рекурсивных функций, так как все примитивно рекурсивные функции всюду определены, а среди частично рекурсивных функций встречаются и функции не всюду определённые, например, функции $d(x, y)$ и $q(x, y)$, рассмотренные в примерах 3.5.1 и 3.5.2, а также, например, нигде не определённая функция $f(x) = \mu y[x + 1 + y = 0]$. Примером общерекурсивной, но не примитивно рекурсивной функции может служить и функция Аккермана

$A(x)$. (Доказательство её общерекурсивности можно найти, например, в книге [12], часть III, §1, № 42 а).

Продолжим теперь продвижение к поставленной цели – к описанию всех функций, вычислимых на машинах Тьюринга. Следующий шаг – доказательство того, что все частично рекурсивные функции вычислимы по Тьюрингу.

Вычислимость по Тьюрингу частично рекурсивных функций.

ТЕОРЕМА 3.5.3. *Если функция $f(x,y)$ правильно вычислима на машине Тьюринга, то и функция $\varphi(x) = \mu y[f(x,y) = 0]$, получающаяся с помощью оператора минимизации из функции $f(x,y)$ также правильно вычислима на машине Тьюринга.*

Доказательство. Обозначим F – машину Тьюринга, правильно вычисляющую функцию $f(x,y)$. Используя её, сконструируем такую машину Тьюринга, которая для заданного значения x вычисляет последовательно значения $f(x,0)$, $f(x,1)$, $f(x,2)$,... до тех пор, пока в первый раз получится $f(x,i) = 0$. После этого машина должна выдать на ленту число i, представляющее собой значение функции $\varphi(x) = i$. Если же для всех i будет иметь место $f(x,i) > 0$, то машина должна работать вечно, и это будет означать, что функция φ не определена в точке x. Начальная конфигурация на конструируемой машине такова $q_0 01^x 0$. Будем мыслить её следующим образом: $q_0 01^x 01^0$ и начнём с применения к ней машины "удвоение" $К_2$. Получим конфигурацию: $01^x 01^0 q 01^x 01^0$. Теперь вычислим значение $f(x,0)$, применив машину F: $01^x 01^0 q_\alpha 01^{f(x,0)}$.

Далее, подбираем команды, которые при условии $f(x,i) > 0$ преобразовывают конфигурацию $01^x 01^i q 01^{f(x,i)}$ в конфигурацию $01^x 01^{i+1} q 01^{f(x,i+1)}$:

$q_\alpha 0 \to q_{\alpha+1} 0П$:	$01^x 01^0 0 q_{\alpha+1} 1^{f(x,0)}$
$q_{\alpha+1} 1 \to q_{\alpha+2} 0$:	$01^x 01^0 0 q_{\alpha+2} 01^{f(x,0)-1}$
$q_{\alpha+2} 0 \to q_{\alpha+3} 0Л$:	$01^x 01^0 q_{\alpha+3} 001^{f(x,0)-1}$
$q_{\alpha+3} 0 \to q_{\alpha+4} 1$:	$01^x 01^0 q_{\alpha+4} 101^{f(x,0)-1}$
$q_{\alpha+4} 1 \to q_{\alpha+5} 1П$:	$01^x 01^1 q_{\alpha+5} 01^{f(x,0)-1}$
O :	$01^x 01^1 q 0$
$Б^- Б^-$:	$q 01^x 01^1$
$К_2$:	$01^x 01^1 q 01^x 01^1$

$$F: \qquad 01^x01^1q_\beta 01^{f(x,1)}$$

$$q_\beta 0 \to q_\alpha 0: \qquad 01^x01^1q_\alpha 01^{f(x,1)}$$

Последняя команда зацикливает программу и машина от конфигурации $01^x01^1q_\alpha 01^{f(x,1)}$ переходит к конфигурации $01^x01^2q_\alpha 01^{f(x,2)}$, затем к конфигурации $01^x01^3q_\alpha 01^{f(x,3)}$ и т.д. Допустим, что по истечении некоторого времени машина достигла конфигурации $01^x01^iq_\alpha 01^{f(x,i)}$, при которой $f(x,i) = 0$. Это означает, что $\varphi(x) = i$ и машина должна выдать этот результат. Число i накопленно в "счетчике" 01^i. Поэтому поступаем следующим образом. Уже имеющаяся команда $q_\alpha 0 \to q_{\alpha+1}0П$ приведёт машину к конфигурации $01^x01^i0q_{\alpha+1}0$. Следующие команды выдают на ленту необходимую конфигурацию $q_0 01^i$, т.е. $q_0 01^{\varphi(x)}$:

$$q_{\alpha+1}0 \to q_\gamma 0: \qquad 01^x01^i0q_\gamma 0$$

$$Б^-Б^-: \qquad 01^xq01^i$$

$$ВОБ^-: \qquad q_\delta 01^i$$

$$q_\delta 0 \to q_0 0: \qquad q_0 01^{\varphi(x)}.$$

Теорема доказана. □

СЛЕДСТВИЕ 3.5.4. *Всякая частично рекурсивная функция вычислима по Тьюрингу.* **ЧРФ $\subseteq$ Выч.Т**

Доказательство. Итак, поскольку оператор минимизации сохраняет свойство вычислимости по Тьюрингу (этим же свойством, как было установленно выше обладают и операторы суперпозиции и примитивной рекурсии), простейшие функции O, S, I^n_m вычислимы по Тьюрингу, а всякая частично рекурсивная функция получается из простейших с помощью применения конечного числа раз трёх указанных операторов, то всякая частично рекурсивная функция вычислима по Тьюрингу. □

Частичная рекурсивность функций вычислимых по Тьюрингу. Наконец мы расширили класс вычислимых по Тьюрингу функций до таких размеров, что он исчерпывает класс *всех* вычислимых по Тьюрингу функций. Это нам и предстоит доказать. Другими словами, мы намерены доказать, что вычислимы по Тьюрингу лишь частично рекурсивные функции, т.е. *если функция вычислима по Тьюрингу, то она частично рекурсивна.* Ещё точнее. По функциональной схеме (программе) машины Тьюринга, вычисляющей функцию $f(x_1, ..., x_n)$, можно построить рекурсивное описание этой функции. Эта теорема впервые была доказана Тьюрингом.

ТЕОРЕМА 3.5.5. *Если функция вычислима по Тьюрингу, то она частично рекурсивна*: **Выч.Т $\subseteq$ ЧРФ** .

Доказательство. Прежде чем приступить к доказательству теоремы, обратим внимание на следующее обстоятельство. Машина Тьюринга, преобразовывающая слова в алфавите $A = \{a_0, a_1, ..., a_{m-1}\}$, никак не различает природу этого алфавита. В частности, она никак не отличает числа от "нечисел": работая с числами, машина не считает их в смысле известных арифметических правил, а преобразовывает слова, представляющие собой записи чисел на каком-то языке (в какой-то системе счисления). Правила преобразований задаём мы с помощью соответствующей программы. Мы же даём полученному машиной словесному результату числовую интерпретацию. Именно в этом смысле мы говорили о вычислимости машиной Тьюринга тех или иных числовых функций. Чтобы доказать сформулированную теорему, необходимо идею арифметической (числовой) интерпретации работы машины Тьюринга довести до определенного совершенства и показать, что любое преобразование, осуществляемое машиной Тьюринга, если его интерпретировать как вычисление, является частично рекурсивным.

Разделим доказательство на этапы.

1) Арифметизация машин Тьюринга начинается с того, что мы вводим в машину своего рода арифметическую систему координат. Эта система координат предназначена для конфигураций машины. Как известно, в каждый момент времени конфигурация на ленте машины Тьюринга имеет вид: $\alpha q_i a_j \beta$, где α - слово в алфавите $A = \{a_0, a_1, ..., a_{m-1}\}$, записанное на ленте левее обозреваемой в данный момент ячейки, q_i – состояние, в котором находится в данный момент машина, a_j – содержимое обозреваемой в данный момент ячейки, β – слово в алфавите A, записанное на ленте правее обозреваемой ячейки. Указанной конфигурации однозначно сопоставляется упорядоченная четвёрка $(\tilde{\alpha}, \tilde{q}_i, \tilde{a}_j, \tilde{\beta})$ чисел, определяемых следующим образом: $\tilde{q}_i = i$; $\tilde{a}_j = j$; $\tilde{\alpha}$ – число, изображаемое цифрами, полученными кодированием символов слова α по следующему правилу: эти символы интерпретируются как m-ичные цифры, т.е. цифры m-ичной системы счисления, т.е. a_i кодируется цифрой i; $\tilde{\beta}$ аналогично получается из β, но при этом читается справа налево, т.е. крайняя слева ячейка β содержит самый младший разряд, а крайняя справа – самый старший (это сделано для того, чтобы нули справа от β не влияли на значение $\tilde{\beta}$).

Завершим кодирование элементов машины Тьюринга. Символу

a_0 пустой ячейки сопоставим число 0, сдвигу вправо П сопоставим 1, сдвигу влево Л – 2, отсутствию сдвига С сопоставим О, буквой z закодируем заключительное состояние (состояние остановки) q_0. В итоге алфавит A закодируется числовым множеством $\tilde{A}$, а алфавит Q – числовым множеством $\tilde{Q}$.

Тогда, например, если мы имеем машину Тьюринга с внешним алфавитом $A = \{a_0, a_1\}$ и на её ленте имеется конфигурация $a_1a_0a_1a_1a_0q_3a_1a_1a_0a_1a_1$, то при нашей кодировке (или координатизации) ей соответствует следующая упорядоченная четвёрка натуральных чисел: (22,3,1,13). Поясним, как она получилась. В нашей конфигурации $\alpha = a_1a_0a_1a_1a_0$, $q_i = q_3$, $a_j = a_1$, $\beta = a_1a_0a_1a_1$. Поэтому $\tilde{\alpha} = 10110$ – двоичное число, которое следующим образом переводится в десятичное: $[10110]_2 = 1\cdot 2^4 + 0\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 0\cdot 2^0 = 22$. Далее, $\tilde{q}_3 = 3$, $\tilde{a}_1 = 1$ и наконец, $\tilde{\beta} = [1101]_2 = 13$.

2) При таком кодировании система команд машины Тьюринга с алфавитом A и множеством состояний Q превращается в тройку числовых функций:

$\varphi_a : \tilde{Q} \times \tilde{A} \to \tilde{A}$ (функция печатаемого символа),

$\varphi_q : \tilde{Q} \times \tilde{A} \to \tilde{Q}$ (функция следующего состояния),

$\varphi_d : \tilde{Q} \times \tilde{A} \to 0, 1, 2$ (функция сдвига).

Например, если среди команд машины Тьюринга имеется команда $q_ia_j \to q'_ia'_jX$, то $\varphi_a(\tilde{q}_i, \tilde{a}_j) = \tilde{a}'_j$, $\varphi_q(\tilde{q}_i, \tilde{a}_j) = \tilde{q}'_i$, $\varphi_d(\tilde{q}_i, \tilde{a}_j) = \xi$, где $\xi = 0, 1, 2$, если X =С,П,Л соответственно.

Отметим, что все эти функции φ_a, φ_q, φ_d заданы на конечном множестве $\tilde{Q} \times \tilde{A}$ и потому примитивно рекурсивны (см. Замечание 3.3.24 выше).

3) Выполнив одну команду, машина Тьюринга преобразует имеющуюся на ленте конфигурацию $K = \alpha q_ia_j\beta$ в новую конфигурацию $K' = \alpha' q'_ia'_j\beta'$. При арифметизации это означает, что упорядоченной четвёрке чисел $(\tilde{\alpha}, \tilde{q}_i, \tilde{a}_j, \tilde{\beta})$, соответствующей конфигурации K, однозначно сопоставляется упорядоченная четвёрка чисел $(\tilde{\alpha}', \tilde{q}'_i, \tilde{a}'_j, \tilde{\beta}')$, соответствующая конфигурации K'.

Например, при команде $q_3a_1 \to q_4a_0$П ранее приведённая конфигурация перейдёт в конфигурацию $K' = a_1a_0a_1a_1a_0a_0q_4a_1a_0a_1a_1$, которой соответствует четвёрка чисел (44,4,1,6).

Так будет происходить почти со всеми конфигурациями, связанными с алфавитами A, Q. В итоге на множестве таких конфигураций будет задана (вообще говоря, не всюду определенная, т.е. частичная)

нечисловая функция $\psi_\theta(K) = K'$, обусловленная данной машиной Тьюринга θ. Назовём её ***функцией следующего шага***.

При введённой арифметизации этой функции соответствует четвёрка числовых функций следующего шага. Иначе говоря, $\tilde{\alpha}'$, $\tilde{q}'$, $\tilde{a}'$, $\tilde{\beta}'$ – это числовые функции, каждая из которых зависит от четырёх числовых переменных $\tilde{\alpha}$, $\tilde{q}$, $\tilde{a}$, $\tilde{\beta}$. Попытаемся понять, как эти функции связаны с функциями системы команд φ_a, φ_q, φ_d. Во-первых, ясно, что $\tilde{q}' = \varphi_q$ и функция $\tilde{q}'$ фактически не зависит от $\tilde{\alpha}$, $\tilde{\beta}$, а зависит лишь от $\tilde{q}$, $\tilde{a}$, т.е. $\tilde{q}'(\tilde{\alpha}, \tilde{q}_i, \tilde{a}_j, \tilde{\beta}) = \varphi_q(\tilde{q}_i, \tilde{a}_j)$.

Рассмотрим теперь функцию $\tilde{\alpha}'(\tilde{\alpha}, \tilde{q}_i, \tilde{a}_j, \tilde{\beta})$. Если при рассматриваемом такте работы машины её головка осталась на месте, т.е. обозреваемая ячейка не изменилась, т.е. $\varphi_d(\tilde{q}_i, \tilde{a}_j) = 0$, то ясно, что $\tilde{\alpha}' = \tilde{\alpha}$. Если головка сдвинулась вправо, т.е. $\varphi_d(\tilde{q}_i, \tilde{a}_j) = 1$, то это означает, что степень каждого разряда числа $\tilde{\alpha}$ повысилась на единицу: нулевой разряд стал первым, первый – вторым и т.д., т.е. число $\tilde{\alpha}$ увеличилось в m раз (напомним, что m – число элементов в алфавите A, т.е. основание системы счисления, в которой рассматривается наша арифметизация): $m\tilde{\alpha}$. Кроме того, в образовавшийся младший, нулевой, разряд помещается та m-ичная цифра, которая соответствует при кодировании только что вписанному на ленту элементу из A. Эта цифра равна $\varphi_a(\tilde{q}_i, \tilde{a}_j)$. В итоге получим, что при сдвиге вправо: $\tilde{\alpha}'(\tilde{\alpha}, \tilde{q}_i, \tilde{a}_j, \tilde{\beta}) = m\tilde{\alpha} + \varphi_a(\tilde{q}_i, \tilde{a}_j)$. Наконец, при сдвиге на данном шаге головки влево, т.е. при $\varphi_d(\tilde{q}_i, \tilde{a}_j) = 2$, степень каждого разряда числа $\tilde{\alpha}$ понизилась на единицу: первый разряд стал нулевым, второй – первый и т.д., т.е. число $\tilde{\alpha}$ уменьшилось в m раз: $\tilde{\alpha}/m$. Но поскольку самый младший, нулевой, разряд оказался фактически как бы "отрубленным", то в итоге получилось не частное $\tilde{\alpha}/m$ (оно могло бы оказаться дробным), а его целая часть: $[\tilde{\alpha}/m]$.

Итак, для функции $\tilde{\alpha}'$ мы получаем следующее описание с помощью оператора условного перехода:

$$\tilde{\alpha}'(\tilde{\alpha}, \tilde{q}_i, \tilde{a}_j, \tilde{\beta}) = \begin{cases} \tilde{\alpha}\ , & \text{если} \quad \varphi_d(\tilde{q}_i, \tilde{a}_j) = 0 (\text{нет сдв.}), \\ m\tilde{\alpha} + \varphi_a(\tilde{q}_i, \tilde{a}_j), & \text{если} \quad \varphi_d(\tilde{q}_i, \tilde{a}_j) = 1 (\text{сдвиг впр.}), \\ \tilde{\alpha}/m\ , & \text{если} \quad \varphi_d(\tilde{q}_i, \tilde{a}_j) = 2 (\text{сдвиг вл.}). \end{cases}$$

Совершенно аналогичная, но в определённом смысле двойственная картина наблюдается и для функции $\tilde{\beta}'$: при сдвиге вправо (влево) она ведёт себя также, как функция $\tilde{\alpha}'$ при сдвиге влево (вправо). Так что для неё получаем следующее описание с помощью операто-

ра условного перехода:

$$\tilde{\beta}'(\tilde{\alpha}, \tilde{q}_i, \tilde{a}_j, \tilde{\beta}) = \begin{cases} \tilde{\beta}, & \text{если} \quad \varphi_d(\tilde{q}_i, \tilde{a}_j) = 0(\text{нет сдв.}), \\ \tilde{\beta}/m \ , & \text{если} \quad \varphi_d(\tilde{q}_i, \tilde{a}_j) = 1(\text{сдвиг впр.}), \\ m\tilde{\beta} + \varphi_a(\tilde{q}_i, \tilde{a}_j), & \text{если} \quad \varphi_d(\tilde{q}_i, \tilde{a}_j) = 2(\text{сдвиг вл.}). \end{cases}$$

Наконец рассмотрим функцию $\tilde{a}'(\tilde{\alpha}, \tilde{q}_i, \tilde{a}_j, \tilde{\beta})$. Если сдвига не происходит, то ясно, что $\tilde{a}' = \varphi_a(\tilde{q}_i, \tilde{a}_j)$. Если происходит сдвиг вправо, то машина приходит к обозрению самого левого разряда (напомним, что это есть младший разряд) числа $\tilde{\beta}$: именно он был отброшен при взятии для $\tilde{\beta}'$ целой части частного $\tilde{\beta}/m$. Он как раз и представляет собой остаток от деления числа $\tilde{\beta}$ на m, т.е. в этом случае $\tilde{a}' = r(m, \tilde{\beta})$. Если же происходит сдвиг влево, то двойственно получаем: $\tilde{a}' = r(m, \tilde{\alpha})$. Итак, функция $\tilde{a}'$ получает следующее описание:

$$\tilde{a}'(\tilde{\alpha}, \tilde{q}_i, \tilde{a}_j, \tilde{\beta}) = \begin{cases} \varphi_a(\tilde{q}_i, \tilde{a}_j), & \text{если} \quad \varphi_d(\tilde{q}_i, \tilde{a}_j) = 0 \ (\text{нет сдвига}), \\ r(m, \tilde{\beta}), & \text{если} \quad \varphi_d(\tilde{q}_i, \tilde{a}_j) = 1 \ (\text{сдвиг вправо}), \\ r(m, \tilde{\alpha}), & \text{если} \quad \varphi_d(\tilde{q}_i, \tilde{a}_j) = 2 \ (\text{сдвиг влево}). \end{cases}$$

Теперь сделаем выводы. В полученных выражениях для функций $\tilde{q}'$, $\tilde{\alpha}'$, $\tilde{\beta}'$, $\tilde{a}'$ задействованы только примитивно рекурсивные функции (см. п.2 выше), и указанные функции получены из этих примитивно рекурсивных функций с помощью оператора условного перехода, который, как мы знаем (см. п. 5 выше), сохраняет свойство примитивной рекурсивности, т.е. из примитивно рекурсивных функций создаёт снова примитивно рекурсивную функцию. Следовательно, все функции $\tilde{\alpha}'$, $\tilde{q}'$, $\tilde{a}'$, $\tilde{\beta}'$ примитивно рекурсивны.

Итак, мы доказали, что на каждом шаге любая машина Тьюринга осуществляет примитивно рекурсивное вычисление.

4) Рассмотрим теперь с точки зрения введённой арифметизации работу машины Тьюринга в целом.

Пусть задана начальная конфигурация $K(0) = (\tilde{\alpha}_0, \tilde{q}_0, \tilde{a}_0, \tilde{\beta}_0)$. Тогда конфигурация $K(t)$, возникающая на такте t работы машины, зависит от величины t и компонент $\tilde{\alpha}_0$, $\tilde{q}_0$, $\tilde{a}_0$, $\tilde{\beta}_0$ начальной конфигурации, т.е. оно является векторной функцией

$$K(t) = (K_\alpha(t), K_q(t), K_a(t), K_\beta(t)) \ ,$$

компоненты которой, в свою очередь, являются функциями, зависящими от переменных t, $\tilde{\alpha}_0$, $\tilde{q}_0$, $\tilde{a}_0$, $\tilde{\beta}_0$. Эти функции определяются следующим образом:

$$K_\alpha(0, \tilde{\alpha}_0, \tilde{q}_0, \tilde{a}_0, \tilde{\beta}_0) = \tilde{\alpha}_0,$$

$$K_\alpha(t+1,\tilde{\alpha}_0,\tilde{q}_0,\tilde{a}_0,\tilde{\beta}_0) = \tilde{\alpha}'(K_\alpha(t,\tilde{\alpha}_0,\tilde{q}_0,\tilde{a}_0,\tilde{\beta}_0),K_q(t,\tilde{\alpha}_0,\tilde{q}_0,\tilde{a}_0,\tilde{\beta}_0), K_a(t,\tilde{\alpha}_0,\tilde{q}_0,\tilde{a}_0,\tilde{\beta}_0),K_\beta(t,\tilde{\alpha}_0,\tilde{q}_0,\tilde{a}_0,\tilde{\beta}_0));$$

$$K_q(0) = \tilde{q}_0,$$

$$K_q(t+1) = \tilde{q}'(K_\alpha(t),\ K_q(t),\ K_a(t),\ K_\beta(t));$$

$$K_a(0) = \tilde{a}_0,$$

$$K_a(t+1) = \tilde{a}'(K_\alpha(t),\ K_q(t),\ K_a(t),\ K_\beta(t));$$

$$K_\beta(0) = \tilde{\beta}_0,$$

$$K_\beta(t+1) = \tilde{\beta}'(K_\alpha(t),\ K_q(t),\ K_a(t),\ K_\beta(t)).$$

(В записях для функций K_q, K_a, K_β аргументы $\tilde{\alpha}_0$, $\tilde{q}_0$, $\tilde{a}_0$, $\tilde{\beta}_0$ для краткости опущены).

Эти соотношения представляют собой схемы примитивной рекурсии, определяющие функции K_α, K_q, K_a, K_β с помощью функций $\tilde{\alpha}'$, $\tilde{q}'$, $\tilde{a}'$, $\tilde{\beta}'$. При этом, рекурсия ведётся по переменной t. Примитивная рекурсивность функций $\tilde{\alpha}'$, $\tilde{q}'$, $\tilde{a}'$, $\tilde{\beta}'$ установлена на предыдущем шаге доказательства. Тогда отсюда очевидно следует, что и функции K_α, K_q, K_a, K_β также примитивно рекурсивны.

5) Наконец, на заключительном шаге доказательства покажем, что результат работы машины Тьюринга (т.е. вычисляемая машиной функция) носит рекурсивный характер. (Обратите внимание: не примитивно рекурсивный, а рекурсивный, т.е. здесь придётся использовать оператор минимизации μ).

Здесь мы докажем утверждение, обратное к тому, которое было доказано в предыдущем пункте: ***всякая функция, правильно вычислимая на машине Тьюринга, частично рекурсивна***. Пусть φ – функция, правильно вычисляемая машиной Тьюринга. Такая машина, начав с конфигурации $q_1 a_0 \beta_0$, останавливается в конфигурации вида $q_z a_z \beta_z$, т.е. для такой машины $K_\alpha(0) = 0$, $K_q(0) = 1$, исходное слово на ленте кодируется числом $m\tilde{\beta}_0 + \tilde{a}_0$, заключительное слово – числом $m\tilde{\beta}_z + \tilde{a}_z$, т.е. вычисляется значение функции $\varphi(m\tilde{\beta}_0 + \tilde{a}_0) = m\tilde{\beta}_z + \tilde{a}_z$. Заключительное слово – это слово, написанное на ленте в тот момент t_z, когда машина впервые перешла в заключительное состояние q_z, т.е. в момент $t_z = \mu t[K_q(t) = z]$. Поэтому

$$\tilde{\beta}_z = K_\beta(t_z),$$

$$\tilde{a}_z = K_a(t_z) \text{ и}$$

$$\varphi(m\tilde{\beta}_0+\tilde{a}_0) = m\tilde{\beta}_z+\tilde{a}_z = mK_\beta(t_z,0,1,\tilde{a}_0,\tilde{\beta}_0)+K_a(t_z,0,1,\tilde{a}_0,\tilde{\beta}_0).$$

Учитывая, что $\tilde{\beta}_0 = [x/m]$, $\tilde{a}_0 = r(m,x)$, выражение для результирующего значения $\varphi(x)$ можно со всеми подробностями записать так:

$$\begin{aligned}\varphi(x) &= mK_\beta(\mu\, t[K_q(t,0,1,r(m,x),[x/m]) = z],0,1,r(m,x),[x/m]) + \\ &+ K_a(\mu t[K_q(t,0,1,r(m,x),[x/m]) = z],0,1,r(m,x),[x/m]),\end{aligned}$$

где m, z – константы, зависящие от конкретной машины. Отсюда непосредственно видно, что функция $\varphi(x)$, представляющая собой результат работы машины Тьюринга, построена из примитивно рекурсивных функций с помощью оператора минимизации μ и, следовательно, является частично рекурсивной.

Этим и завершается доказательство того, что всякая вычислимая по Тьюрингу функция частично рекурсивна. □

Соединив вместе следствие 3.5.4 и теорему 3.5.5, приходим к следующей теореме.

Теорема 3.5.6. *Функция вычислима по Тьюрингу тогда и только тогда, когда она частично рекурсивна.* **Выч.Т = ЧРФ** .
□

Итог рассмотрений настоящего параграфа состоит в том, что мы дали некую альтернативную характеристику вычислимым по Тьюрингу функциям: это – те и только те функции, которые частично рекурсивны. Другими словами, класс функций, вычислимых по Тьюрингу, совпадает с классом частично рекурсивных функций. Совпадение этих двух классов вычислимых функций, в основе построения которых лежали совершенно разные подходы к формализации понятия вычислимости функции, говорит о том, что эти подходы оказались весьма состоятельными, и служит косвенным подтверждением того, что как тезис Тьюринга, так и тезис Чёрча не только не безосновательны, но и имеют все права на признание.

Глава IV

НОРМАЛЬНЫЕ АЛГОРИТМЫ МАРКОВА

В этой главе рассматривается ещё одна формализация интуитивного понятия алгоритма – теория нормальных алгоритмов (или алгорифмов, как называл их создатель теории). Эта теория была разработана советским математиком А.А.Марковым (1903 – 1979) в конце 40-х – начале 50-х годов XX века. Эти алгоритмы представляют собой некоторые правила по переработке слов в каком-либо алфавите, так что исходные данные и искомые результаты для алгоритмов являются словами в некотором алфавите. В качестве элементарной операции, на базе которой будут строиться нормальные алгоритмы, А.А.Марков выделил подстановку в слово одного подслова вместо другого.

4.1. Марковские подстановки, нормальные алгоритмы и нормально вычислимые функции

В этом параграфе вводятся основные понятия, связанные с марковским подходом к понятию алгоритма: марковские подстановки, нормальные алгоритмы, нормально вычислимые функции.

Марковские подстановки. *Алфавитом* (как и прежде) называется любое непустое множество. Его элементы называются *буквами*, а любые последовательности букв – *словами* в данном алфавите. Для удобства рассуждений допускаются пустые слова (они не имеют в своем составе ни одной буквы). *Пустое слово* будем обозначать Λ. Если A и B – два алфавита, причём $A \subseteq B$, то алфавит B называется *расширением* алфавита A.

Слова будем обозначать латинскими буквами: P, Q, R, S, T, U, V, W (или этими же буквами с индексами). Слово T называется *составной частью* другого слова V, или *подсловом* второго,

или *вхождением* во второе, или говорят, что слово T *входит* в слово V, если существуют такие (возможно, пустые) слова U, W, что $V = UTW$. Например, если A – алфавит русских букв, то можем рассмотреть такие слова: P_1 = параграф, P_2 = граф, P_3 = ра. Слово P_2 является подсловом слова P_1, а P_3 – подсловом P_1 и P_2, причём, в P_1 оно входит дважды. Особый интерес представляет первое вхождение.

Определение 4.1.1. *Марковской подстановкой* называется операция над словами, задаваемая с помощью упорядоченной пары слов (P, Q), состоящая в следующем. В заданном слове R находят первое вхождение слова P (если таковое имеется) и, не изменяя остальных частей слова R, заменяют в нём это вхождение словом Q. Полученное слово называется *результатом* применения марковской подстановки (P, Q) к слову R. Если же первого вхождения P в слово R нет (и, следовательно, вообще нет ни одного вхождения P в R), то считается, что марковская подстановка (P, Q) не применима к слову R.

Частными случаями марковских подстановок являются подстановки с пустыми словами: (Λ, Q), (P, Λ), (Λ, Λ).

Пример 4.1.2. В таблице рассматриваются примеры марковских подстановок. В каждой её строке сначала даётся преобразуемое слово, затем применяемая к нему марковская подстановка и наконец – получающееся в результате слово.

Преобразуемое слово	Марковская подстановка	Результат
138 578 926	(8 578 9, 00)	130 026
тарарам	(ара, Λ)	трам
шрам	(ра, ар)	шарм
функция	(Λ, ζ)	ζ-функция
логика	(ика, Λ)	лог
книга	(Λ, Λ)	книга
поляна	(пор, т)	[*не применима*]

Для обозначения марковской подстановки (P, Q) используется запись $P \to Q$. Она называется *формулой подстановки* (P, Q). Некоторые подстановки (P, Q) будем называть заключительными (смысл названия станет ясен чуть позже). Для обозначения таких подстановок будем использовать запись $P \to .\ Q$, называя её *формулой заключительной подстановки*. Слово P называется *левой частью*, а Q – *правой частью* в формуле подстановки. Каждое

из слов P и Q может быть пустым словом. При этом предполагается, что стрелка $\to$ и точка $\cdot$ не являются буквами алфавита, в котором построены слова P и Q.

Нормальные алгоритмы и их применение к словам. Упорядоченный конечный список формул подстановок в алфавите A:

$$\begin{cases} P_1 \to (.)\, Q_1 \\ P_2 \to (.)\, Q_2 \\ \dots\dots\dots\dots\dots\dots \\ P_r \to (.)\, Q_r \end{cases}$$

– называется *схемой* (или *записью*) нормального алгоритма в A. (Запись точки в скобках означает, что она может стоять в этом месте, а может отсутствовать.) Данная схема определяет (детерминирует) алгоритм преобразования слов, называемый нормальным алгоритмом Маркова. Дадим его точное определение.

ОПРЕДЕЛЕНИЕ 4.1.3. ***Нормальным алгоритмом*** (*Маркова*) ***в алфавите*** A называется следующее правило (обозначаемое $\mathbf{A}$) построения последовательности V_i слов в алфавите A, исходя из данного слова V в этом алфавите. В качестве начального слова V_0 последовательности берётся слово V. Пусть для некоторого $i \geq 0$ слово V_i построено и процесс построение рассматриваемой последовательности ещё не завершился. Если при этом в схеме нормального алгоритма нет формул левые части которых входили бы в V_i, то V_{i+1} полагают равным V_i и процесс построения последовательности считается завершившимся. Если же в схеме имеются формулы с левыми частями, входящими в V_i, то в качестве V_{i+1} берётся результат марковской подстановки правой части первой из таких формул вместо первого вхождения её левой части в слово V_i; процесс построения последовательности считается ***завершившимся***, если на данном шаге была применена формула заключительной подстановки, и ***продолжающимся*** в противном случае. Если процесс построения упомянутой последовательности обрывается, то говорят, что рассматриваемый ***нормальный алгоритм применим к слову*** V. Последний член W последовательности называется ***результатом применения нормального алгоритма к слову*** V. Говорят, что ***нормальный алгоритм перерабатывает*** V ***в*** W и и пишут $\mathbf{A}(V) = W$.

Последовательность V_i, будем записывать следующим образом:

$$V_0 \Longrightarrow V_1 \Longrightarrow V_2 \Longrightarrow \dots \Longrightarrow V_{m-1} \Longrightarrow V_m\ ,$$

где $V_0 = V$ и $V_m = W$.

Работа нормального алгоритма **A**, задаваемого приведённой выше схемой, может быть описана следующим образом. Пусть дано слово V в алфавите A. Ищем первую в схеме алгоритма **A** формулу подстановки $P_k \to (.)\ Q_k$ такую, что слово P_k входит в слово V. Если ни одно из слов $P_1, ..., P_r$ не входит в V, то алгоритм **A** не применим к слову V. Если находим первую в схеме алгоритма **A** формулу подстановки $P_k \to (.)\ Q_k$ такую, что слово P_k входит в слово V, то совершаем подстановку слова Q_k вместо самого левого (первого) вхождения слова P_k в слово V. Пусть V_1 – результат такой подстановки. Если $P_k \to (.)\ Q_k$ – формула заключительной подстановки, то работа алгоритма заканчивается, и его результатом является слово V_1: $\mathbf{A}(V) = V_1$. Если подстановка $P_k \to (.)\ Q_k$ не является заключительной, то применяем к слову V_1 тот же поиск, который был только что применён к слову V, и так далее. Если мы в конце концов получим такое слово V_m $(m \geq 1)$, что алгоритм **A** не применим к V_m (т.е. ни одно из слов $P_1, ..., P_r$ не входит в V_m), то работа алгоритма **A** заканчивается и V_m будет его результатом: $\mathbf{A}(V) = V_m$. При этом, возможно, что описанный процесс никогда не закончится (будет продолжаться бесконечно). В таком случае также будем считать, что алгоритм **A** не применим к слову V.

Мы определили понятие нормального алгоритма в алфавите A. Если же алгоритм задан в некотором расширении алфавита A, то говорят, что он есть *нормальный алгоритм над* A.

Различие между понятием нормального алгоритма *в* алфавите A и понятием нормального алгоритма *над* алфавитом A является весьма существенным и важным. Всякий нормальный алгоритм *в* алфавите A использует только буквы из A, тогда как нормальный алгоритм *над* алфавитом A может использовать и некоторые дополнительные буквы, не входящие в A, хотя применяться он будет к словам в алфавите A и выдавать будет в результате слово в алфавите A. Ясно, что всякий нормальный алгоритм *в* алфавите A будет также и нормальным алгоритмом *над* алфавитом A. В то же время обратное утверждение не верно: можно показать, что существуют, вообще говоря, алгоритмы *в* A, которые определяются нормальными алгоритмами *над* A, но сами не являются нормальными алгоритмами *в* A.

Рассмотрим примеры нормальных алгоритмов.

Пример 4.1.4. Пусть $A = \{a, b\}$ – алфавит. Рассмотрим следу-

ющую схему нормального алгоритма в A:

$$\begin{cases} a \to .\,\Lambda \\ b \to b \end{cases}$$

Нетрудно понять, как работает определяемый этой схемой нормальный алгоритм. Всякое слово V в алфавите A, содержащее хотя бы одно вхождение буквы a, он перерабатывает в слово, получающееся из V вычеркиванием в нём самого левого (первого) вхождения буквы a. Пустое слово он перерабатывает в пустое. Алгоритм неприменим к таким словам, которые содержат только букву b (будет работать вечно). Например, $aabab \Longrightarrow abab$, $ab \Longrightarrow b$, $aa \Longrightarrow a$, $bbab \Longrightarrow bbb$, $baba \Longrightarrow bba$.

Пример 4.1.5. Пусть $A = \{a_0, a_1, ..., a_n\}$ – алфавит. Рассмотрим схему

$$\begin{cases} a_0 \to \Lambda \\ a_1 \to \Lambda \\ \dots\dots\dots \\ a_n \to \Lambda \\ \Lambda \to .\,\Lambda \end{cases}$$

Она определяет нормальный алгоритм, перерабатывающий всякое слово (в алфавите A) в пустое слово. Например,

$$a_1a_2a_1a_3a_0 \Longrightarrow a_1a_2a_1a_3 \Longrightarrow a_2a_1a_3 \Longrightarrow a_2a_3 \Longrightarrow a_3 \Longrightarrow \Lambda;$$

$$a_0a_2a_2a_1a_3a_1 \Longrightarrow a_2a_2a_1a_3a_1 \Longrightarrow a_2a_2a_3a_1 \Longrightarrow a_2a_2a_3 \Longrightarrow$$

$$\Longrightarrow a_2a_3 \Longrightarrow a_3 \Longrightarrow \Lambda.$$

В примерах 4.1.4 и 4.1.5 рассмотренные нормальные алгоритмы являются алгоритмами *в* данном алфавите A. В следующем примере строится алгоритм *над* алфавитом A (хотя применяться он будет, конечно же, к словам в алфавите A и выдавать будет в результате слово в алфавите A).

Пример 4.1.6. Пусть $A = \{a_0, a_1, ..., a_n\}$ – произвольный алфавит. Для всякого слова $V = a_{i_0}a_{i_1}...a_{i_k}$ в этом алфавите назовём его ***отражением*** слово $V' = a_{i_k}a_{i_{k-1}}...a_{i_1}a_{i_0}$. Построим нормальный алгоритм $\mathbf{A}$ *над* алфавитом A, который каждое слово V в алфавите A перерабатывает в его отражение V': $\mathbf{A}(V) = V'$. Рассмотрим расширение $B = A \cup \{\alpha, \beta\}$ $(\alpha, \beta \notin A)$ алфавита A и следующую

схему нормального алгоритма в алфавите B (над алфавитом A):

$$\begin{cases} \alpha\alpha \to \beta & & (a) \\ \beta\xi \to \xi\beta & (\xi \in A) & (b) \\ \beta\alpha \to \beta & & (c) \\ \beta \to .\,\Lambda & & (d) \\ \alpha\eta\xi \to \xi\alpha\eta & (\xi, \eta \in A) & (e) \\ \Lambda \to \alpha & & (f) \end{cases} .$$

Здесь в подстановках (b) и (e) используются переменные ξ, η, вместо которых могут подставляться любые буквы из алфавита A. Проверьте, что данный алгоритм следующим образом осуществляет переработку, например, слова $a_1a_2a_3$:

$$a_1a_2a_3 \stackrel{f}{\Longrightarrow} \alpha a_1a_2a_3 \stackrel{e}{\Longrightarrow} a_2\alpha a_1a_3 \stackrel{e}{\Longrightarrow} a_2a_3\alpha a_1 \stackrel{f}{\Longrightarrow}$$
$$\alpha a_2a_3\alpha a_1 \stackrel{e}{\Longrightarrow} a_3\alpha a_2\alpha a_1 \stackrel{f}{\Longrightarrow} \alpha a_3\alpha a_2\alpha a_1 \stackrel{f}{\Longrightarrow} \alpha\alpha a_3\alpha a_2\alpha a_1 \stackrel{a}{\Longrightarrow}$$
$$\beta a_3\alpha a_2\alpha a_1 \stackrel{b}{\Longrightarrow} a_3\beta\alpha a_2\alpha a_1 \stackrel{c}{\Longrightarrow} a_3\beta a_2\alpha a_1 \stackrel{b}{\Longrightarrow} a_3a_2\beta\alpha a_1 \stackrel{c}{\Longrightarrow}$$
$$a_3a_2\beta a_1 \stackrel{b}{\Longrightarrow} a_3a_2a_1\beta \stackrel{d}{\Longrightarrow} a_3a_2a_1 .$$

Из работы алгоритма на этом примере ясно, что он любое непустое (любой длины) слово V в алфавите A переработает в его отражение V'.

Пример 4.1.7. Пусть алфавит A не содержит букв α, β, γ, и пусть $B = A \cup \{\alpha\}$ и $C = A \cup \{\alpha, \beta, \gamma\}$.

а) Постройте нормальный алгоритм **A** в алфавите B (над алфавитом A), который в каждом непустом слове в алфавите A стирал бы первую букву, а пустое слово переводил бы в пустое, т.е. $\mathbf{A}(\Lambda) = \Lambda$ и $\mathbf{A}(aV) = V$ для любой буквы $a \in A$ и любого слова V в A.

б) Постройте нормальный алгоритм **A** в алфавите C (над алфавитом A), который удваивал бы любое слово в алфавите A, т.е. любое слово V в алфавите A переводил бы в слово VV: $\mathbf{A}(V) = VV$.

Рассмотрим ещё два примера нормальных алгоритмов в алфавите A.

Пример 4.1.8. В алфавите $A = \{1\}$ схема, состоящая из одной подстановки $\Lambda \to .\,1$, определяет нормальный алгоритм, который к каждому слову в алфавите $A = \{1\}$ (все такие слова суть следующие: Λ, 1, 11, 111, 1111, 11111 и т. д.) приписывает слева 1.

Пример 4.1.9. В алфавите $A = \{1\}$ построить нормальный алгоритм, который для всякого слова V в этом алфавите выдавал

бы слово 1, если число единиц в слове V делится на 3, и выдавал бы пустое слово Λ в противном случае.

Рассмотрим нормальный алгоритм в алфавите $A = \{1\}$ со следующей схемой:

$$\begin{cases} 111 \to \Lambda & (a) \\ 11 \to .\,\Lambda & (b) \\ 1 \to .\,\Lambda & (c) \\ \Lambda \to .\,1 & (d) \end{cases}.$$

Этот алгоритм работает следующим образом: пока число букв 1 в слове не меньше 3, алгоритм последовательно стирает по три буквы (подстановка (a)). Если при этом все буквы оказались стёртыми, то подстановка (d) выдаёт 1 и заканчивает работу алгоритма. Если оказались не стёртыми две единицы 11, то подстановка (b) заменяет их на пустое слово и заканчивает работу алгоритма. Наконец, если не стёртой оказалась одна единица 1, то подстановка (c) заменяет её на пустое слово и заканчивает работу алгоритма. Например:

$$1111111 \overset{a}{\Longrightarrow} 1111 \overset{a}{\Longrightarrow} 1 \overset{c}{\Longrightarrow} \Lambda\,;$$

$$111111111 \overset{a}{\Longrightarrow} 111111 \overset{a}{\Longrightarrow} 111 \overset{a}{\Longrightarrow} \Lambda \overset{d}{\Longrightarrow} 1\,.$$

В двух последних примерах 4.1.8 и 4.1.9 мы фактически рассмотрели нормальные алгоритмы, вычисляющие некоторые функции, а именно, в первом примере функцию следования $S(n) = n + 1$, а во втором – функцию $\varphi_3(n)$, определяемую следующим образом:

$$\varphi_3(11...1) = \begin{cases} 1, & \text{если } n \text{ делится на } 3; \\ \Lambda, & \text{если } n \text{ не делится на } 3; \end{cases}$$

где n – число единиц в слове $11...1$.

Как и машины Тьюринга, нормальные алгоритмы не производят собственно вычислений: они лишь производят преобразования слов, заменяя в них одни буквы другими, по предписанным им правилам. В свою очередь, мы предписываем им такие правила, результаты применения которых мы уже можем интерпретировать как вычисления. Рассмотренные два примера наглядно демонстрируют это. Следующий пример в этом отношении будет ещё более ярким. В нём мы организуем процесс вычисления с помощью нормального алгоритма функции следования $S(n) = n + 1$, не в одноичной системе счисления (как это сделано в примере 4.1.8), а в привычной человеку десятичной системе счисления, хотя никаких привычных нам

арифметических действий по сложению чисел при этом выполняться не будет.

Пример 4.1.10. Построим нормальный алгоритм для вычисления функции $S(n) = n + 1$ в десятичной системе счисления. В качестве алфавита возьмём перечень арабских цифр $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, а нормальный алгоритм будем строить в его расширении $B = A \cup \{a, b\}$. Вот схема этого нормального алгоритма (читается по столбцам):

$$\begin{array}{lll}
0b \to .\,1 & a0 \to 0a & 0a \to 0b \\
1b \to .\,2 & a1 \to 1a & 1a \to 1b \\
2b \to .\,3 & a2 \to 2a & 2a \to 2b \\
3b \to .\,4 & a3 \to 3a & 3a \to 3b \\
4b \to .\,5 & a4 \to 4a & 4a \to 4b \\
5b \to .\,6 & a5 \to 5a & 5a \to 5b \\
6b \to .\,7 & a6 \to 6a & 6a \to 6b \\
7b \to .\,8 & a7 \to 7a & 7a \to 7b \\
8b \to .\,9 & a8 \to 8a & 8a \to 8b \\
9b \to b0 & a9 \to 9a & 9a \to 9b \\
b \to .\,1 & & \Lambda \to a
\end{array}$$

Попытаемся применить алгоритм к пустому слову Λ. Нетрудно понять, что на каждом шаге должна будет применяться самая последняя формула данной схемы. Получается бесконечный процесс:

$$\Lambda \implies a \implies aa \implies aaa \implies aaaa \implies \ldots$$

Это означает, что к пустому слову данный алгоритм неприменим.

Если применить теперь алгоритм к слову 499, получим следующую последовательность слов:

$499 \implies a499$ (применена последняя формула) $\implies 4a99$ (формула из середины второго столбца) $\implies 49a9 \implies 499b$ (дважды применена формула из конца второго столбца) $\implies 499b$ (предпоследняя формула) $\implies 49b0 \implies 4b00$ (дважды применена предпоследняя формула первого столбца) $\implies 500$ (применена формула из середины первого столбца).

Таким образом, слово 499 перерабатывается данным нормальным алгоритмом в слово 500. Проверьте самостоятельно, что $328 \implies 329$, $789 \implies 790$.

В рассмотренном примере нормальный алгоритм построен в алфавите B, являющемся существенным расширением алфавита A то

есть $A \subseteq B$ и $A \neq B$, но данный алгоритм слова в алфавите A перерабатывает снова в слова в алфавите A. Так что, рассмотренный алгоритм задан *над* алфавитом A.

Наша задача теперь будет состоять в том, чтобы охарактеризовать процесс вычисления с помощью нормальных алгоритмов функций от любого числа аргументов, заданных в множестве натуральных чисел и принимающих значения в этом же множестве. Такие функции мы будем называть нормально вычислимыми. Затем нужно будет как-то описать все такие функции, охарактеризовать их.

Нормально вычислимые функции. Итак, займёмся уточнением понятия вычислимости с помощью нормальных алгоритмов натуральных функций от натуральных аргументов.

Прежде всего, нужно условиться о том, в каком виде будут использоваться в нормальных алгоритмах натуральные числа. Для этого зафиксируем алфавит $A_1 = \{1\}$ и его расширение $A_2 = \{1, *\}$. Для всякого натурального числа n определим по индукции следующие слова в алфавите A_1: $\overline{0} = 1$, $\overline{n+1} = \overline{n}1$, т.е. $\overline{1} = 11$, $\overline{2} = 111$ и т.д. Слова $\overline{n}$ будем называть *числами*. Таким образом, натуральное число n мы кодируем словом в алфавите A_1, состоящим из $n+1$ единицы. Будем также обозначать слово $11 \ldots 1$, состоящее из n единиц, символом 1^n. Тогда наше кодирование выглядит следующим образом: $\overline{0} = 1^1$, $\overline{1} = 1^2$, $\overline{2} = 1^3$, ... $\overline{n} = 1^{n+1}$, ...

Тогда для рассмотренного в примере 4.1.8 алгоритма $\mathbf{A}$ имеем: $\mathbf{A}(\overline{n}) = \overline{n+1}$ для любого натурального n.

Теперь договоримся о кодировании упорядоченных n-ок натуральных чисел. (Эта процедура понадобится нам для описания процесса нормальной вычислимости функций n натуральных аргументов). Для этого мы используем расширенный алфавит $A_2 = \{1, *\}$. Поставим в соответствие каждой упорядоченной n-ке $(k_1, k_2, \ldots, k_n)$, где $k_1, k_2, \ldots, k_n$ – натуральные числа, слово $\overline{k_1} * \overline{k_2} * \ldots * \overline{k_n}$ в алфавите A_2, которое обозначим через $\overline{(k_1, k_2, \ldots, k_n)}$. Так, например, $\overline{(1, 4, 0, 2)}$ обозначает слово $11 * 11111 * 1 * 111$. Итак,

$$\overline{(k_1, k_2, \ldots, k_n)} = \overline{k_1} * \overline{k_2} * \ldots * \overline{k_n} .$$

Пусть теперь $f(x_1, x_2, \ldots, x_n)$ – частичная функция n аргументов, заданная на множестве N натуральных чисел и принимающая значения в том же множестве, т.е. $f\colon N^n \to N$.

Определение 4.1.11. Будем называть частичную функцию $f(x_1, x_2, \ldots, x_n)$ *частично вычислимой по Маркову* (или *частично нормально вычислимой*), если существует нормальный алгоритм,

вычисляющий эту функцию, т.е. такой нормальный алгоритм $\mathbf{A}_f$, *над* алфавитом $A_2 = \{1, *\}$, что

$$\mathbf{A}_f(\ \overline{(k_1, k_2, ..., k_n)}\ ) \ = \ \overline{f(k_1, k_2, ..., k_n)}$$

всякий раз, когда хотя бы одна из частей этого равенства определена (тогда определена и другая часть этого равенства и эти части равны). При этом предполагается, что нормальный алгоритм $\mathbf{A}_f$ не применим к словам, отличным от слов вида $\overline{(k_1, k_2, ..., k_n)}$. Если функция $f(x_1, x_2, ..., x_n)$ определена всюду, т.е. для любой упорядоченной n-ки натуральных чисел, и является частично вычислимой по Маркову, то будем называть её ***вычислимой по Маркову*** (или ***нормально вычислимой***).

Рассмотрим примеры нормально вычислимых функций.

Пример 4.1.12. Построим нормальный алгоритм $\mathbf{A}_0$ над алфавитом $A_2 = \{1, *\}$, вычисляющий нуль-функцию $O(x) = 0$ для любого $x \in N$. Он должен быть применим к словам вида $\overline{n} = 1^{n+1}$ и только к таким словам, причём, на каждом таком слове он должен выдавать результат $\overline{0} = 1$, т.е. $\mathbf{A}_0(\overline{n}) = \overline{0}$ для любого слова $\overline{n}$. Покажем, что такой алгоритм может быть задан следующей схемой в расширенном алфавите $B = \{1, *, \alpha\} \subset A_2$:

$$\begin{cases} * \ \rightarrow \ * & (a) \\ \alpha 11 \ \rightarrow \ \alpha 1 & (b) \\ \alpha 1 \ \rightarrow \ .\ 1 & (c) \\ \Lambda \ \rightarrow \ \alpha & (d) \end{cases}$$

В самом деле, проанализируем работу этого нормального алгоритма. Во-первых, в данном слове происходит поиск вхождения буквы $*$. Если такое вхождение находится (а это будет означать, что данное слово не есть число, т.е. не имеет вид $\overline{n} = 1^{n+1}$), тогда выполняется первая подстановка схемы $* \rightarrow *$, затем снова будет выполняться эта подстановка, затем снова, и т.д., то есть алгоритм зацикливается, работает вечно, и это означает, что данный алгоритм не применим к словам, содержащим букву $*$, т.е. к словам, не имеющим вид $\overline{n}$.

Если вхождения буквы $*$ в данное слово не обнаружено, то происходит поиск вхождения слова $\alpha 11$; его нет, так как слово составлено из букв алфавита $A_2 = \{1, *\}$ $(\alpha \notin A_2)$. Затем происходит поиск вхождения слова $\alpha 1$; его также нет по той же причине. Наконец, отрабатывается подстановка $\Lambda \rightarrow \alpha$. Она даёт результат, ибо всякое слово V можно себе мыслить как начинающееся с пустого

слова – ΛV. Эта подстановка приписывает в начало данного слова букву α. Дальше вступает в действие подстановка (b), стирая при каждом её применении одну единицу, пока не останется всего одна единица. Тогда заключительная подстановка (c) завершает работу алгоритма, стирая букву α и оставляя результатом единственную единицу, которая служит кодировкой натурального числа 0. Например:

$$11111 \overset{d}{\Longrightarrow} \alpha 11111 \overset{b}{\Longrightarrow} \alpha 1111 \overset{b}{\Longrightarrow} \alpha 111 \overset{b}{\Longrightarrow}$$
$$\overset{b}{\Longrightarrow} \alpha 11 \overset{b}{\Longrightarrow} \alpha 1 \overset{c}{\Longrightarrow} 1\,.$$

Пример 4.1.13. Построить нормальный алгоритм $\mathbf{A}_1$ над алфавитом $A_2 = \{1, *\}$, вычисляющий функцию следования $S(x) = x+1$ для любого $x \in N$. Он также должен быть применим к словам вида $\overline{n} = 1^{n+1}$ и только к таким словам, причём, на каждом таком слове он должен выдавать результат $\overline{n+1} = 1^{n+2}$, т.е. $\mathbf{A}_1(\overline{n}) = \overline{n+1}$ для любого слова $\overline{n}$. Нетрудно понять, что такой алгоритм может быть задан следующей схемой в расширенном алфавите $B = \{1, *, \alpha\} \subset A_2$:

$$\begin{cases} * \to * & (a) \\ \alpha 1 \to .\ 11 & (b) \\ \Lambda \to \alpha & (c) \end{cases}\,.$$

Пример работы этого алгоритма:

$$11111 \overset{c}{\Longrightarrow} \alpha 11111 \overset{b}{\Longrightarrow} 111111\,.$$

Первая подстановка схемы $* \to *$, как и в предыдущем примере, обеспечивает неприменимость данного алгоритма к словам, содержащим букву $*$, т.е. к словам, отличным от чисел $\overline{n}$.

Пример 4.1.14. Построить нормальный алгоритм $\mathbf{A}^n_m$ над алфавитом $A_2 = \{1, *\}$, вычисляющий функцию-проектор $I^n_m(x_1, x_2, ..., x_n) = x_m$ (где $1 \le m \le n$) для любых $x_1, x_2, ..., x_n \in N$. Он также должен быть применим к тем и только тем словам в алфавите A_2, которые имеют вид $\overline{(k_1, k_2, ..., k_n)}$, причём, $\mathbf{A}^n_m(\ \overline{(k_1, k_2, ..., k_n)}\ ) = \overline{k_m}$ для любой упорядоченной n-ки натуральных чисел $(k_1, k_2, ..., k_n)$.

Требуемый алгоритм будем строить в следующем расширенном алфавите $B = \{1, *, \alpha_1, \alpha_2, ..., \alpha_{2n},\} \subset A_2$. Обозначим через F_i (при $1 \le i < n$) следующий список формул подстановки:

$$\alpha_{2i-1}* \to \alpha_{2i-1}*$$

$$\alpha_{2i-1}1 \to \alpha_{2i}1$$

$$\alpha_{2i}1 \to \alpha_{2i}$$

$$\alpha_{2i}* \to \alpha_{2i+1}$$

В зависимости от того, каково число m по отношению к числу n: $1 < m < n$, или $m = 1$, или $m = n$, схемы алгоритма будут различными. В каждом из этих трёх случаев они приводятся в столбцах следующей таблицы:

$1 < m < n$	$m = 1$	$m = n$
F_1 . . . . . F_{m-1} $\alpha_{2m-1}* \to \alpha_{2m-1}*$ $\alpha_{2m-1}1 \to \alpha_{2m}1$ $\alpha_{2m}1 \to 1\alpha_{2m}$ $\alpha_{2m}* \to \alpha_{2m+1}$ F_{m+1} . . . . . F_{n-1} $\alpha_{2n-1}* \to \alpha_{2n-1}*$ $\alpha_{2n-1}1 \to \alpha_{2n}1$ $\alpha_{2n}1 \to \alpha_{2n}$ $\alpha_{2n}* \to \alpha_{2n}*$ $\alpha_{2n} \to .\,\Lambda$ $\Lambda \to \alpha_1$	$\alpha_1* \to \alpha_1*$ $\alpha_1 1 \to \alpha_2 1$ $\alpha_2 1 \to 1\alpha_2$ $\alpha_2* \to \alpha_3$ F_2 . . . . . F_{n-1} $\alpha_{2n-1}* \to \alpha_{2n-1}*$ $\alpha_{2n-1}1 \to \alpha_{2n}1$ $\alpha_{2n}1 \to \alpha_{2n}$ $\alpha_{2n}* \to \alpha_{2n}*$ $\alpha_{2n} \to .\,\Lambda$ $\Lambda \to \alpha_1$	F_1 . . . . . F_{n-1} $\alpha_{2n-1}* \to \alpha_{2n-1}*$ $\alpha_{2n-1}1 \to \alpha_{2n}1$ $\alpha_{2n}1 \to 1\alpha_{2n}$ $\alpha_{2n}* \to \alpha_{2n}*$ $\alpha_{2n} \to .\,\Lambda$ $\Lambda \to \alpha_1$

При этом, отметим, что в первом случае (левый столбец таблицы) блоки формул $F_{m+1}, \ldots, F_{n-1}$ появляются лишь в том случае, когда $m + 1 \leq n - 1$.

Рассмотрим подробно работу данного алгоритма для конкретных значений n и m, например, для $n = 4$ и $m = 2$, т.е. построим нормальный алгоритм $\mathbf{A}_2^4$. Расширенный алфавит в этом случае будет иметь вид: $B = \{1, *, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha_7, \alpha_8\}$. Используем первую (левую) из схем, приведённых в таблице. Для её составления нам понадобятся списки формул F_1 и F_3. Выписываем схему нормального алгоритма $\mathbf{A}_2^4$:

$$
\begin{array}{rcll}
\alpha_1 * & \to & \alpha_1 * & \\
\alpha_1 1 & \to & \alpha_2 1 & \qquad F_1 \\
\alpha_2 1 & \to & \alpha_2 & \\
\alpha_2 * & \to & \alpha_3 & \\
\\
\alpha_3 * & \to & \alpha_3 * & \\
\alpha_3 1 & \to & \alpha_4 1 & \\
\alpha_4 1 & \to & 1\alpha_4 & \\
\alpha_4 * & \to & \alpha_5 & \\
\\
\alpha_5 * & \to & \alpha_5 * & \\
\alpha_5 1 & \to & \alpha_6 1 & \qquad F_3 \\
\alpha_6 1 & \to & \alpha_6 & \\
\alpha_6 * & \to & \alpha_7 & \\
\\
\alpha_7 * & \to & \alpha_7 * & \\
\alpha_7 1 & \to & \alpha_8 1 & \\
\alpha_8 1 & \to & \alpha_8 & \\
\alpha_8 * & \to & \alpha_8 * & \\
\alpha_8 & \to & .\,\Lambda & \\
\Lambda & \to & \alpha_1 &
\end{array}
$$

Применим данный алгоритм к слову $111 * 1111 * 1 * 11$:

$111{*}1111{*}1{*}11 \Longrightarrow \alpha_1 111{*}1111{*}1{*}11 \Longrightarrow \alpha_2 111{*}1111{*}1{*}11 \Longrightarrow$
$\alpha_2 11 * 1111 * 1 * 11 \Longrightarrow \alpha_2 1 * 1111 * 1 * 11 \Longrightarrow \alpha_2 * 1111 * 1 * 11 \Longrightarrow$
$\alpha_3 1111 * 1 * 11 \Longrightarrow \alpha_4 1111 * 1 * 11 \Longrightarrow 1\alpha_4 111 * 1 * 11 \Longrightarrow$
$11\alpha_4 11 * 1 * 11 \Longrightarrow 111\alpha_4 1 * 1 * 11 \Longrightarrow 1111\alpha_4 * 1 * 11 \Longrightarrow$
$1111\alpha_5 1{*}11 \Longrightarrow 1111\alpha_6 1{*}11 \Longrightarrow 1111\alpha_6{*}11 \Longrightarrow 1111\alpha_7 11 \Longrightarrow$
$1111\alpha_8 11 \Longrightarrow 1111\alpha_8 1 \Longrightarrow 1111\alpha_8 \Longrightarrow 1111$.

Обратим внимание на то, как обеспечивается бесконечная работа алгоритма на словах, не являющихся кодами упорядоченных n-ок натуральных чисел, т.е. не имеющими вида $\overline{(k_1, k_2, ..., k_n)} = \overline{k_1} * \overline{k_2} * ... * \overline{k_n}$. Слова в алфавите $A_2 = \{1, *\}$, не имеющие такого вида, могут быть, в свою очередь, одного из трёх следующих видов:

а) начинаются с буквы $*$;

б) две звёздочки идут подряд;

в) заканчиваются буквой $*$.

В случае а) зацикливание алгоритма обеспечивают команды вида $\alpha_{2i-1}* \to \alpha_{2i-1}*$; в случае б) – команды $\alpha_{2i}* \to \alpha_{2i+1}$, $\alpha_{2i+1}* \to \alpha_{2i+1}*$, и наконец, в случае в) – команда $\alpha_{2n}* \to \alpha_{2n}*$.

Примените самостоятельно данный нормальный алгоритм к словам: 1111*1*111*11, 1*11*111*1111. Составьте схемы нормальных алгоритмов для вычисления функций-проекторов: I_1^4, I_4^4, I_1^3, I_2^3, I_3^3 и проанализируйте их работу.

Принцип нормализации Маркова. Создатель теории нормальных алгоритмов советский математик А.А. Марков выдвинул гипотезу, получившую название принцип нормализации Маркова.

ПРИНЦИП НОРМАЛИЗАЦИИ МАРКОВА. *Для нахождения значений функции, заданной в некотором алфавите, тогда и только тогда существует какой-нибудь алгоритм, когда эта функция нормально вычислима.*

Этот принцип, как и тезисы Тьюринга и Чёрча, носит внематематический характер и не может быть строго доказан. Он выдвинут на основании математического и практического опыта. Все, что в предыдущих параграфах было сказано о тезисах Тьюринга и Чёрча, можно с полным правом отнести к принципу нормализации Маркова. Косвенным подтверждением этого принципа служат теоремы следующего пункта, устанавливающие эквивалентность и этой теории алгоритмов теориям машин Тьюринга и рекурсивных функций.

4.2. Рекурсивные функции и нормально вычислимые функции

Здесь мы покажем, что класс всех частично рекурсивных функций совпадает с классом всех нормально вычислимых функций. Это совпадение будет означать, что понятие частично рекурсивной функции равносильно понятию нормально вычислимой функции, а вместе с ним (с учётом теоремы 3.5.6) – и понятию функции, вычислимой по Тьюрингу.

Совпадение класса всех частично рекурсивных функций с классом всех нормально вычислимых функций. В следствии 3.5.4, используя генетический способ построения класса всех частично рекурсивных функций, мы доказали, что всякая такая функция является функцией, вычислимой по Тьюрингу, т.е.

класс всех частично рекурсивных функций включается в класс всех функций, вычислимых по Тьюрингу.

Аналогичным путём можно доказать, что всякая частично рекурсивная функция является частично нормально вычислимой функцией. В примерах 4.1.12, 4.1.13, 4.1.14 мы уже встали на этот путь: там доказана (частичная) нормальная вычислимость первоначальных рекурсивных функций: нуль-функции $O(x)$, функции следования $S(x)$ и функций-проекторов I^n_m $(1 \leq m \leq n)$. Следуя далее по этому пути, необходимо доказать, что операторы подстановки (суперпозиции), примитивной рекурсии и минимизации (μ-оператор), с помощью которых строится из первоначальных функций класс всех (частично) рекурсивных функций, сохраняют свойство функций быть (частично) нормально вычислимыми, т.е. с помощью этих операторов из (частично) нормально вычислимых функций получаются снова (частично) нормально вычислимые функции. Это и будет доказывать, что всякая частично рекурсивная функция будет частично нормально вычислимой. Весь ход этого достаточно трудоёмкого доказательства подробно описан в книге [13] (предложение 5.8 и используемые в его доказательстве утверждения 5.1. – 5.7).

В теореме 3.5.5 было доказано обратное утверждение по отношению к утверждению 3.5.4. Аналогично имеет место обратное утверждение и для нормально вычислимых функций: если частичная функция является частично нормально вычислимой, то она частично рекурсивна; если же всюду определённая функция нормально вычислима, то она общерекурсивна. (Трудоёмкое доказательство этого утверждения также можно найти в книге [13], следствие 5.11).

Итак, имеет место следующая теорема.

Теорема 4.2.1. *Класс всех частично рекурсивных функций совпадает с классом всех частично нормально вычислимых функций (частично вычислимых по Маркову):* **ЧРФ = Норм. выч.**.

Совпадение класса всех нормально вычислимых функций с классом всех функций, вычислимых по Тьюрингу. Во-первых, это утверждение, конечно же, следует из теорем 3.5.6 и 4.2.1. Тем не менее, оно может быть доказано и непосредственно. Сначала докажем включение в одну сторону.

Теорема 4.2.2. *Всякая функция, вычислимая по Тьюрингу, будет также и нормально вычислимой:* **Выч.Т $\subseteq$ Норм. выч.**.

Доказательство. Пусть машина Тьюринга с внешним алфавитом $A = \{a_0, a_1, ..., a_m\}$ и алфавитом внутренних состояний $Q =$

$\{q_0, q_1, ..., q_n\}$ вычисляет некоторую функцию f, заданную и принимающую значения в множестве слов алфавита A (словарную функцию на A). Попытаемся представить программу этой машины Тьюринга в виде схемы некоторого нормального алгоритма. Для этого нужно каждую команду машины Тьюринга $q_\alpha a_i \to q_\beta a_j X$ представить в виде совокупности марковских подстановок. Конфигурации, возникающие в машине Тьюринга в процессе её работы, представляют собой слова в алфавите $A \cup Q$. Эти слова имеют вид: $a_{i_1} ... a_{i_k} q_\alpha a_{i_{k+1}} ... a_i$. Нам понадобится различать начало слова и его конец (или его левый и правый концы). Для этого к алфавиту $A \cup Q$ добавим ещё два символа (не входящие ни в A, ни в Q): $A \cup Q \cup \{u, v\}$. Эти символы будем ставить соответственно в начало и конец каждого машинного слова w: uwv.

Пусть на данном шаге работы машины Тьюринга к машинному слову w предстоит применить команду $q_\alpha a_i \to q_\beta a_j X$. Это означает, что машинное слово w (а вместе с ним и слово uwv) содержит подслово $q_\alpha a_i$. Посмотрим, какой совокупностью марковских подстановок можно заменить данную команду в каждом из следующих трёх случаев.

а) $X =$ С, т.е. команда имеет вид: $q_\alpha a_i \to q_\beta a_j$. Ясно, что в этом случае следующее слово получается из слова uwv с помощью подстановки $q_\alpha a_i \to q_\beta a_j$, которую мы и будем считать соответствующем команде $q_\alpha a_i \to q_\beta a_j$.

б) $X =$ Л, т.е. команда имеет вид: $q_\alpha a_i \to q_\beta a_j$Л. Нетрудно понять, что в этом случае для получения из слова uwv следующего слова, надо к слову uwv применить ту подстановку из совокупности

$$a_0 q_\alpha a_i \to q_\beta a_0 a_j,$$

$$a_1 q_\alpha a_i \to q_\beta a_1 a_j,$$

$$a_m q_\alpha a_i \to q_\beta a_m a_j,$$

$$u q_\alpha a_i \to u q_\beta a_0 a_j,$$

которая применима к слову uwv. В частности, последняя подстановка применима только тогда, когда q_α – самая левая буква в слове w, т.е. когда надо пристраивать ячейку слева.

в) $X =$ П, т.е. команда имеет вид: $q_\alpha a_i \to q_\beta a_j$П. В этом случае аналогично, чтобы получить из слова uwv следующее слово, нужно к слову uwv применить ту из подстановок совокупности

$$q_\alpha a_i a_0 \to a_j q_\beta a_0,$$

$$q_\alpha a_i a_1 \to a_j q_\beta a_1,$$

$$\cdots\cdots\cdots\cdots$$

$$q_\alpha a_i a_m \to a_j q_\beta a_m,$$

$$q_\alpha a_i v \to a_j q_\beta a_0 v,$$

которая применима к слову uwv.

Поскольку слово $q_\alpha a_i$ входит в слово w только один раз, то к слову uwv применима только одна из подстановок, перечисленных в пунктах б, в. Поэтому порядки следования подстановок в этих пунктах безразличны, важны лишь их совокупности.

Заменим каждую команду из программы машины Тьюринга указанным способом совокупностью марковских подстановок. Мы получим схему некоторого нормального алгоритма. Теперь ясно, что применить к слову w данную машину Тьюринга – это всё равно, что применить к слову uwv построенный нормальный алгоритм. Другими словами, действие машины Тьюринга равнозначно действию подходящего нормального алгоритма. Это и означает, что всякая функция, вычислимая по Тьюрингу, нормально вычислима.

Теорема доказана. □

Верно и обратное утверждение.

ТЕОРЕМА 4.2.3. *Всякая нормально вычислимая функция вычислима по Тьюрингу*: **Норм.выч. ⊆ Выч.Т** .

Доказательство. В книге [77] (теорема 1, с. 246 – 249) фактически доказано, что всякая нормально вычислимая функция частично рекурсивна (см. также следствие 5.11 в книге [13], с. 248 – 249). Добавив сюда доказанное в предыдущей главе III утверждение о том, что всякая частично рекурсивная функция вычислима по Тьюрингу (следствие 3.5.4) , мы и приходим к справедливости сформулированного утверждения о вычислимости по Тьюрингу всякой нормально вычислимой функции. □

Таким образом, объединяя теоремы 4.2.2 и 4.2.3, мы приходим к следующей теореме.

ТЕОРЕМА 4.2.4. *Класс нормально вычислимых функций совпадает с классом функций вычислимых по Тьюрингу:*

Норм. выч. = Выч.Т . □

Эквивалентность различных теорий алгоритмов. Итак, в двух последних главах мы познакомились с тремя теориями, каждая из которых уточняет понятие алгоритма и доказали, что все эти теории равносильны между собой. Другими словами, они описывают один и тот же класс функций, т.е. справедлива следующая теорема.

ТЕОРЕМА 4.2.5. *Следующие классы функций (заданных на множестве натуральных чисел и принимающих значения в этом же множестве) совпадают:*

а) *класс* **Выч.Т** *всех функций, вычислимых по Тьюрингу;*

б) *класс* **ЧРФ** *всех частично рекурсивных функций;*

в) *класс* **Норм.выч.** *всех нормально вычислимых функций.*

$$\boxed{\textbf{Выч.Т} = \textbf{ЧРФ} = \textbf{Норм.выч.}}\ . \quad \square$$

Полезно уяснить смысл и значение этого важного результата. В разное время в разных странах учёные независимо друг от друга, изучая интуитивное понятие алгоритма и алгоритмической вычислимости, создали теории, описывающие данное понятие, которые оказались равносильными. Этот факт служит мощным косвенным подтверждением адекватности этих теорий опыту вычислений, справедливости каждого из тезисов Тьюринга, Чёрча и Маркова. В самом деле, ведь если бы один из этих классов оказался шире какого-либо другого класса, то соответствующий тезис Тьюринга, Чёрча или Маркова был бы опровергнут. Например, если бы класс нормально вычислимых функций оказался шире класса рекурсивных функций, то существовала бы нормально вычислимая, но не рекурсивная функция. В силу её нормальной вычислимости она была бы алгоритмически вычислима в интуитивном понимании алгоритма, и предположение о её нерекурсивности опровергало бы тезис Чёрча. Но классы Тьюринга, Чёрча и Маркова совпадают, и таких функций не существует. Это и служит ещё одним косвенным подтверждением тезисов Тьюринга, Маркова и Чёрча.

Можно отметить, что существуют ещё и другие теории алгоритмов (одним из важных является так называемое λ-исчисление), и для всех них также доказана их равносильность с рассмотренными теориями.

Глава V

ОБЩИЙ ПОДХОД К ТЕОРИИ АЛГОРИТМОВ

Теперь, когда мы достаточно подробно изучили три различных формализации понятия алгоритма, установили их эквивалентность и сформулировали тезисы Тьюринга, Чёрча и Маркова, можно сделать некоторые общие выводы. Из наших рассмотрений следует, что любые утверждения о существовании или несуществовании алгоритмов, сделанные в одной из трёх формализаций, верны и в другой. Это означает, что возможно развитие и изложение теории алгоритмов, инвариантное по отношению к способу формализации понятия "алгоритм". Это – своего рода общая теория алгоритмов. Основные её понятия – алгоритм и вычислимая функция. При интерпретировании этой общей теории, например, в теории рекурсивных функций понятие алгоритм превращается в рекурсивное описание функции, понятие вычислимая функция – в частично рекурсивная (или общерекурсивная) функция. (При этом следует всегда помнить, что алгоритм и вычисляемая им функция – это не одно и то же. Одна и та же функция может вычисляться с помощью разных алгоритмов).

5.1. Нумерации алгоритмов и вычислимых функций

Таким образом, после всех проведённых формализаций понятия алгоритма (машины Тьюринга, рекурсивные функции, нормальные алгоритмы Маркова) в этой главе мы по существу снова возвращаемся к интуитивному пониманию алгоритма.

Предварительные соображения. Итак, мы хотели бы создать математическую теорию, описывающую в самом общем виде свойства объектов, которые мы назвали алгоритмами (в интуитивном смысле) и которые на практике имеют самую разнообразную

природу, т.е. имеют дело с предметами разнообразной природы. Другими словами, мы хотим смоделировать эти реальные объекты практической жизни и реальные отношения между ними какими-нибудь математическими объектами и математическими отношениями между ними с тем, чтобы, изучив свойства этих объектов и отношений математическими методами, применить полученные результаты к практике их реальных прообразов. Этот метод математического моделирования реальных процессов математика использует уже не одну тысячу лет.

Построение модели начинается со следующей процедуры. Для данного класса D объектов, с которыми имеют дело рассматриваемые алгоритмы, выбирается и фиксируется эффективно задаваемое взаимно однозначное отображение $\nu : D \to N$ класса D в множество N натуральных чисел. Говорят, что объект $d \in D$ *кодируется* натуральным числом $\nu(d)$, которое называется (кодовым) *номером*, или *индексом*, или *кодом* объекта d при *кодировании* (нумерации) ν. Кодирование должно быть *обратимым*, т.е. должен существовать неформальный алгоритм для распознавания кодовых номеров и для обратного "декодирующего" отображения кодовых номеров в объекты класса D.

Предположим теперь, что над реальными объектами из класса D имеется некоторая функция $f : D \to D$. Тогда f естественно кодируется функцией $\overline{f} : N \to N$, которая отображает код каждого объекта d из области определения функции f в код объекта $f(d)$. В явном виде функция $\overline{f}$ задаётся как суперпозиция $\overline{f} = \nu \ f \quad \overline{\nu}^1$.

Таким образом, в таком виде теория алгоритмов (или теория вычислимости) предстаёт как формальная наука об отображениях натуральных чисел, т.е. как наука о функциях, заданных на множестве натуральных чисел и принимающих значения также в множестве натуральных чисел.

Для дальнейшей конкретизации процедуры построения нашей модели можно поступить следующим образом. Объекты, с которыми имеет дело алгоритм, можно описать (закодировать) словами некоторого (например, русского) языка с использованием букв некоторого конечного (например, русского) языка. Так что функция $f : D \to D$ над реальными объектами сначала превращается в функцию $f^* : Al^* \to Al^*$ над словами этого алфавита.

Наконец, слова во всяком конечном алфавите можно взаимно однозначным образом занумеровать (закодировать) натуральными числами, так что функция f^* далее превращается (кодируется)

в функцию $\overline{f} : N \to N$, заданную и принимающую значения в множестве натуральных чисел.

Покажем теперь, как может быть осуществлена нумерация слов, составленных из букв конечного алфавита $Al = \{a_0, a_1, a_2, ..., a_n\}$. Сначала нумеруются все буквы данного алфавита следующими по порядку простыми натуральными числами: $Nom\, a_i = p_i$, где $Nom\, a_i$ – номер буквы a_i, а p_i – $(i+1)$-ое по порядку простое число ($p_0 = 2$ – первое простое число). Таким образом, получаем:

$$Nom\ a_0 = p_0 = 2,\ Nom\ a_1 = p_1 = 3,\ Nom\ a_2 = p_2 = 5,\ \ldots$$
$$\ldots,\ Nom\ a_n = p_n\ .$$

После этого можем занумеровать слова, составленные из букв алфавита Al по следующему правилу. Слову $\alpha = a_{i_1} a_{i_2} ... a_{i_k}$ припишем номер:

$$Nom\ \alpha = 2^{Nom\ a_{i_1}} \cdot 3^{Nom\ a_{i_2}} \cdot 5^{Nom\ a_{i_3}} \cdot \ \ldots\ \cdot p_{k-1}^{Nom\ a_{i_k}},$$

где p_{k-1} – k-ое по порядку простое число. Получаем отображение $Nom : Al^* \to N$.

В силу единственности разложения числа на простые множители, данное отображение будет взаимно однозначным, т.е. разные слова из Al^* не могут получить один и тот же номер, т.е. отображение Nom действительно является нумерацией слов. Но отображение Nom не является отображением НА N, т.е. не каждое натуральное число является номером некоторого слова из Al^*. Тем не менее, нумерация Nom является эффективной в том смысле, что можно указать алгоритм, который для любого натурального числа n определяет, является ли n номером в данной нумерации какого-либо слова из Al^*, и если да, то строит (восстанавливает) слово, имеющее номер n. Этот алгоритм также основан на свойстве единственности представления всякого натурального числа в виде произведения степеней простых чисел, и действует он следующим образом. Представить число n в виде произведения степеней простых чисел (разложить на простые множители). Если среди оснований степеней идут подряд без пробелов простые числа, начиная с первого, а среди показателей степеней имеются лишь числа из множества $\{Nom\ a_0,\ Nom\ a_1,\ Nom\ a_2,\ \ldots,\ Nom\ a_n\}$, т.е. из множества $\{2, 3, 5, 7, 11, ..., p_n\}$, то число n является номером в нумерации Nom некоторого слова из Al^*. Это слово содержит столько букв, сколько степеней участвует в произведении. Чтобы восстановить это слово, нужно выписать подряд все те буквы из Al, номера

которых стоят в показателях степеней простых чисел полученного разложения.

Вычислимые функции. Будем рассматривать функции f от одного или нескольких аргументов, заданные на множестве $N = \{0, 1, 2, 3, ..., n, ...\}$ всех натуральных чисел или на некоторых его подмножествах (частичные функции) и принимающие значения в множестве N. Таким образом, рассматриваются функции f, являющиеся отображениями декартовой n-ой степени N^n в множество N. Область определения $Dom\ (f)$ функции f есть подмножество множества $N^n = N \times ... \times N$:

$$Dom\ (f) = \{(x_1, ..., x_n) : f(x_1, ..., x_n) - \text{определено}\}.$$

Область изменения (значений) f есть подмножество множества N:

$$Im\ (f) = \{y \in N : (\exists x_1, ..., x_n)(f(x_1, ..., x_n) = y)\}.$$

В частности, если $Dom\ (f)$ пусто, то f называется *нигде не определённой функцией.*

Для частичных функций используют понятие *условного равенства* таких функций. Пишут $f(x_1, ..., x_n) \cong g(x_1, ..., x_n)$ и говорят, что f *условно равна* g. Это означает, что функции $f(x_1, ..., x_n)$ и $g(x_1, ..., x_n)$ определены на одних и тех же наборах значений аргументов и к тому же на этих наборах значения функций f и g равны. Мы будем придерживаться традиционного обозначения равенства "=", понимая под ним для частичных функций условное равенство.

Функция $f(x_1, ..., x_n)$ называется *вычислимой*, если существует алгоритм, позволяющий вычислять её значения для тех наборов аргументов, для которых она определена, и работающий вечно, если функция для данного набора значений аргументов не определена.

Таким образом, можно сказать, что $f(x_1, ..., x_n)$ условно равна результату применения алгоритма A к набору начальных данных $x_1, ..., x_n$. Другими словами, вычислимая алгоритмом A функция f определена на тех наборах $x_1, ..., x_n$, к которым этот алгоритм применим, причём, результат этого применения как раз и равен значению $f(x_1, ..., x_n)$, и f не определена на тех наборах $x_1, ..., x_n$, к которым алгоритм A не применим.

Например, всюду определённые функции $f_1(x, y) = x + y$ и $f_2(x, y) = x \cdot y$ (обе: $N \times N \to N$) вычислимы. Первую вычисляет алгоритм сложения столбиком, а вторую – алгоритм умножения столбиком двух натуральных чисел. Алгоритм деления углом двух

натуральных чисел, по существу, вычисляет две частичные функции $q: N \times N \to N$ и $r: N \times N \to N$, определённые на тех парах (m, n), для которых $n \neq 0$, и при $n \neq 0$

$$q(m,n) = \text{ (частное от деления } m \text{ на } n) \ ,$$
$$r(m,n) = \text{ (остаток от деления } m \text{ на } n) \ .$$

Нетрудно понять, что композиция (суперпозиция) вычислимых функций есть функция вычислимая, а также, что вычислимой является и нигде не определённая функция.

Далее, если $f: N \to N$ – (частичная) взаимно однозначная функция, отображающая $Dom\ (f)$ на $Im\ (f)$, то определяется обратная функция $f^{-1}: N \to N$ по правилу:

$$f^{-1}(y) = x \iff f(x) = y \ .$$

При этом, $Dom\,(f^{-1}) = Im\,(f)$, $Im\,(f^{-1}) = Dom\,(f)$. Тогда, если функция f вычислима, то вычислима и обратная функция f^{-1}. В самом деле, значение $f^{-1}(y)$ вычисляется следующим алгоритмом: "Перебирать все элементы x из $Dom\ (f)$, вычислять значение $f(x)$ [ведь функция f вычислима], каждый раз сравнивать его с y, пока не произойдёт равенство: $f(x) = y$. Выдать такой элемент x в качестве результата." Если $y \in Dom\ (f^{-1}) = Im\ (f)$, то этот алгоритм выдаст результат $f^{-1}(y)$. Если же $y \notin Dom\ (f^{-1}) = Im\ (f)$, то этот перебор никогда не закончится, и алгоритм будет работать бесконечно долго.

Идеи, применённые при построении рекурсивных функций и машин Тьюринга, могут быть использованы в общей теории вычислимости. Так, во-первых, мы можем естественно считать вычислимыми простейшие функции: следования $S(x) = x + 1$ (пример 2.2.1), нуль-функцию $O(x) = 0$ (пример 2.4.4), функции-проекторы $I^n_m(x_1, ..., x_n) = x_m$, $1 \leq m \leq n$ (пример 2.4.5). Ясно, что композиция (суперпозиция) вычислимых функций есть функция вычислимая (теорема 2.4.7). Далее, функция, возникающая примитивной рекурсией из вычислимых функций, сама вычислима (теорема 3.4.1). Наконец, функции, получающиеся с помощью операторов суммирования и мультиплицирования (теорема 3.2.15), а также с помощью оператора минимизации (теорема 3.5.3) из вычислимых функций, сами будут вычислимы. В силу тезисов Тьюринга и Чёрча, эти выводы справедливы для любых моделей алгоритмической вычислимости.

Наконец, в заключение отметим, что для доказательства вычислимости функции вовсе не обязательно предъявлять вычисляющий её алгоритм; достаточно лишь доказать его существование. Например, функция

$$f(x) = \begin{cases} 1, \text{ если на Марсе есть жизнь}, \\ 0, \text{ если на Марсе нет жизни}, \end{cases}$$

вычислима, так как она равна либо функции тождественно равной 1, либо функции тождественно равной 0, а обе они вычислимы.

Разрешимые предикаты и разрешимые алгоритмические проблемы. В параграфе 3.3 были рассмотрены примитивно рекурсивные предикаты. Осуществляя общий подход к теории алгоритмов, можно следующим образом обобщить это понятие.

Определение 5.1.1. Предикат $P(x_1, x_2, ..., x_n)$ называется ***разрешимым*** (или ***вычислимым***, или ***рекурсивным***, или ***рекурсивно разрешимым***), если вычислима его характеристическая функция

$$\chi_P(x_1, x_2, ..., x_n) =$$

$$= \begin{cases} 1, & \text{если высказывание } P(x_1, x_2, ..., x_n) \text{ -- истинно}, \\ 0, & \text{если высказывание } P(x_1, x_2, ..., x_n) \text{ -- ложно}. \end{cases}$$

Каждый предикат $P(x_1, ..., x_n)$ характеризует некоторое свойство предметов тех множеств, на которых он задан, т.е. ставит некоторую массовую (алгоритмическую) проблему определения того, удовлетворяют ли произвольно выбранные предметы данному свойству. Разрешимость данного предиката представляет собой возможность эффективно (с помощью алгоритма) вычислять характеристическую функцию данного предиката, т.е. для любых предметов определять, выполним ли для них данный предикат, т.е. разрешима ли алгоритмически массовая проблема, описываемая данным предикатом. При этом следует помнить, что говоря о разрешимости (или неразрешимости) предиката или массовой проблемы мы имеем дело с вычислимостью (или невычислимостью) *всюду определённых* функций.

Все многочисленные примеры примитивно рекурсивных предикатов, рассмотренные в параграфе 3.3, служат также примерами разрешимых предикатов. Так. в примере 3.3.10 предикат делимости $d(x, y)$: $x|y$ характеризует массовую проблему, состоящую в

выяснении, является ли число x делителем числа y. Характеристическая функция этого предиката (и этой проблемы) может быть эффективно вычислена алгоритмом, дающим для входов x, y ответ ДА или НЕТ. Поэтому предикат и проблема *алгоритмически* или *эффективно разрешимы*, или просто *разрешимы*.

Как и в случае логических операций с примитивно рекурсивными предикатами (теорема 3.3.13), доказывается, что если разрешимы предикаты $P(x_1, ..., x_n)$ и $Q(x_1, ..., x_n)$, то будут разрешимы и предикаты: $\neg P(x_1, ..., x_n)$, $P(x_1, ..., x_n) \wedge Q(x_1, ..., x_n)$, $P(x_1, ..., x_n) \vee Q(x_1, ..., x_n)$.

Нумерация (перечисление) алгоритмов. Понятие нумерации или перечисления, или кодирования алгоритмов, и особенно эффективность этой процедуры, играет фундаментальную роль в теории вычислимости, значение которой невозможно переоценить. Особенно эффективна роль этой процедуры при доказательствах алгоритмической неразрешимости тех или иных массовых проблем, т.е. при доказательствах несуществования единого алгоритма для решения массовой проблемы.

Мы уже отмечали, что любой алгоритм может быть описан конечным множеством конечных слов, записанных с помощью букв конечного алфавита Al. Как известно, множество Al^* всех таких слов счётно. Каждый алгоритм представляет собой некоторое конечное подмножество множества Al^*, а множество ALG всех алгоритмов равномощно (находится во взаимно однозначном соответствии) с множеством $P_0(Al^*)$ всех конечных подмножеств счётного множества Al^*. Как известно, и это множество тоже счётно. Следовательно, множество ALG всех алгоритмов счётно. А раз так, то существует биекция $\nu: ALG \to N$, ставящая в соответствие каждому алгоритму некоторое натуральное число $\nu(\mathbf{A}) = n$. Это число n называют ***индексом***, или (***гёделевым***) ***номером*** алгоритма $\mathbf{A}$, и пишут $\mathbf{A} = \mathbf{A}_n$, а саму биекцию называют ***нумерацией*** алгоритмов. Таким образом, все алгоритмы выстраиваются в некую фиксированную последовательность (нумерацию или перечисление)

$$\mathbf{A}_0,\ \mathbf{A}_1,\ \mathbf{A}_2,\ \dots\ ,\ \mathbf{A}_n,\ \dots \qquad (1)$$

Это перечисление можно организовать, например, следующим образом. Сначала выбрать все алгоритмы, в описании которых (во всех словах) использована лишь одна буква. Таких слов и тем более, алгоритмов, будет конечное число. Они все нумеруются. Затем выбираются все описания из двух букв; их тоже конечное число, и они

нумеруются следом. И т.д. Тогда ясно, что такая нумерация будет эффективной в том смысле, что по номеру (натуральному числу) n может быть однозначно восстановлен алгоритм $\mathbf{A}$, имеющий в нумерации (1) номер n: $\mathbf{A} = \mathbf{A}_n$, т.е. существует эффективно вычислимая обратная функция $\nu^{-1} : N \to \mathrm{ALG}$ такая, что $\nu^{-1}(n) = \mathbf{A}$.

Ясно, что различных гёделевых нумераций алгоритмов существует бесконечное множество. Частности и детали построения такой нумерации не важны. Существенно лишь то, что нумерующая функция (биекция) и обратная для неё функция эффективно вычислимы. Для излагаемой далее теории подходит всякая другая биекция, удовлетворяющая указанному условию. Поэтому на протяжении всех последующих рассуждений в нашей общей (абстрактной) теории алгоритмов мы будем считать, что выбрана и зафиксирована одна такая нумерация. Эта нумерация будет играть роль своеобразной системы координат в океане алгоритмов, и её фиксация никак не повлияет на инвариантность нашей общей теории подобно тому, как в аналитической геометрии выбор конкретной системы координат никак не влияет на свойства геометрических фигур, изучаемых с помощью этой системы координат, а лишь выявляет эти свойства.

Описанная процедура нумерации алгоритмов является одним из примеров применения общего метода арифметизации конструктивных процедур и объектов, изобретённый Куртом Гёделем и впервые применённый им для доказательства его знаменитой теоремы о неполноте формальной арифметики. Арифметизация состоит в том, что конструктивные процедуры некоторого определённого класса нумеруются натуральными числами так, что всякую процедуру можно однозначно "расшифровать" по её номеру. Свойства процедур превращаются тогда в свойства натуральных чисел, преобразования процедур – в функции на множестве натуральных чисел. Этот метод настолько поразителен по простоте и изяществу идеи и настолько эффективен по силе получаемых с его помощью результатов, что его вполне можно сравнить с другой арифметизацией – декартовой арифметизацией геометрии, т.е. изобретённым Декартом методом координат, являющимся без всякого сомнения одним из величайших достижений математики.

Нумерация машин Тьюринга. Опишем теперь более конкретно процесс нумерации всех машин Тьюринга. Будем считать, что для обозначения внутренних состояний машин Тьюринга используются буквы бесконечной последовательности: q_0, q_1, q_2, ... , q_r, ... , а для обозначения букв внешних алфавитов используются буквы

последовательности: a_0, a_1, a_2, ... , a_s, ...

Выразим (или, как говорят, закодируем) все символы этих бесконечных последовательностей словами конечного стандартного алфавита $A_0 = \{a_0, 1, q, С, П, Л\}$ по следующим правилам:

q_0 обозначим (закодируем) словом q

q_1 обозначим (закодируем) словом qq

q_2 обозначим (закодируем) словом qqq

. . . .

q_i обозначим (закодируем) словом (из i штук букв q) $qq...q$

. . . .

a_1 обозначим (закодируем) словом 1

a_2 обозначим (закодируем) словом 11

a_3 обозначим (закодируем) словом 111

. . . .

a_j обозначим (закодируем) словом (из j единиц) 11...1

. . . .

В стандартном алфавите программу машины Тьюринга можно записать в виде слова, руководствуясь следующим правилом. Сначала все команды программы переводятся на язык стандартного алфавита, для чего в записях этих команд $q_i a_j \to q_l a_m X$, где $X \in \{С, П, Л\}$, опускается символ "$\to$", а буквы q_i, a_j, q_l, a_m заменяются соответствующими словами стандартного алфавита. Затем полученные слова-команды записываются подряд в любом порядке в виде единого слова.

Например, программа машины Тьюринга, рассмотренной в примере 2.1.1, имеет вид:

$$q_1 a_0 \to q_2 a_0 П, \quad q_1 a_1 \to q_1 a_1 П, \quad q_2 a_0 \to q_0 a_1 С, \quad q_2 a_1 \to q_2 a_1 П.$$

Опускаем символ "$\to$", заменяем буквы словами стандартного алфавита и в результате получаем следующие слова в стандартном алфавите, кодирующие соответствующие команды:

$$qqa_0qqqa_0П, \quad qq1qq1П, \quad qqqa_0q1С, \quad qqq1qqq1П \ .$$

Выписываем эти слова подряд и получаем слово, кодирующее программу данной машины Тьюринга:

$$qqa_0qqqa_0\text{П}qq1qq1\text{П}qqqa_0q1\text{С}qqq1qqq1\text{П} .$$

Нетрудно указать алгоритм, позволяющий узнавать, является ли слово в стандартном алфавите программой некоторой машины Тьюринга. Такой алгоритм может, например, состоять в следующем. Нужно анализировать все подслова данного слова, заключенные между всевозможными парами букв из {С, П, Л}. Эти подслова должны иметь следующую структуру: сначала записаны несколько букв q, затем a_0 или несколько букв 1, затем снова – несколько букв q и, наконец, снова a_0 или несколько единиц.

Таким образом, каждая машина Тьюринга вполне определяется некоторым конечным словом в конечном стандартном алфавите A_0. Поскольку множество всех конечных слов в конечном алфавите, как известно, счётно, поэтому и всех мыслимых машин Тьюринга (отличающихся друг от друга по существу своей работы) имеется не более чем счётное количество.

Перенумеруем теперь все машины Тьюринга, для чего все слова стандартного алфавита, представляющие собой программы всевозможных машин Тьюринга, расположим в виде фиксированной счётно бесконечной последовательности, которую составим по такому правилу: сначала выписываются в какой-нибудь фиксированной последовательности все однобуквенные слова: α_0, α_1, ..., α_ξ, представляющие программы машин Тьюринга (множество таких слов конечно, потому что конечен стандартный алфавит, из букв которого строятся слова), затем выписываются все двубуквенные слова $\alpha_{\xi+1}$, ..., α_η, представляющие программы машин Тьюринга (множество таких слов также конечно, потому что конечен стандартный алфавит), затем выписываются трёхбуквенные слова и т.д. Получится последовательность программ

$$\alpha_0, \quad \alpha_1, \quad \alpha_2, \; . \; . \; . \; , \quad \alpha_n, \; . \; . \; .$$

всех мыслимых машин Тьюринга. Число k будем называть ***номером машины Тьюринга***, если программа этой машины записывается словом α_k.

Чтобы сделать более конструктивной процедуру восстановления машины Тьюринга по её номеру, можно слова, кодирующие в стандартном алфавите { a_0, 1, q С, П, Л } программы машин Тьюринга, нумеровать по правилу, описанному в первом пункте настоящего параграфа.

Нумерация вычислимых функций. Напомним, что функция называется вычислимой, если существует вычисляющий её ал-

горитм. Все алгоритмы мы выстраиваем в некоторую фиксированную последовательность (нумерацию, перечисление):

$$\mathbf{A}_0,\ \mathbf{A}_1,\ \mathbf{A}_2,\ \dots\ ,\ \mathbf{A}_n,\ \dots \qquad (1)$$

Вычисляемая алгоритмов A_n функция φ_n также получает номер n, и все вычислимые функции также выстраиваются в последовательность:

$$\varphi_0,\ \varphi_1,\ \varphi_2,\ \cdots\ ,\ \varphi_n,\ \cdots \qquad (2)$$

Но ясно, что соответствие $\mathbf{A}_n \to \varphi_n$ не взаимно однозначно, т.е. одна и та же функция может вычисляться разными алгоритмами. Это означает, что в нумерации (последовательности) (2) имеются повторения. Из неё можно построить нумерацию (перечисление) без повторений. Для этого нужно из всех равных между собой функций последовательности (2) нумеровать следующим новым номером только ту функцию, которая имеет наименьший номер в нумерации (2). Формально новая нумерация (без повторений) может быть описана следующей схемой примитивной рекурсии с использованием оператора минимизации:

$$\begin{cases} f(0) = 0, \\ f(m+1) = \mu z\{\varphi_z \neq \varphi_{f(0)} \wedge \varphi_z \neq \varphi_{f(1)} \wedge\ \dots\ \wedge\ \varphi_z \neq \varphi_{f(m+1)}\}. \end{cases}$$

Тогда
$$\varphi_{f(0)},\ \varphi_{f(1)},\ \varphi_{f(2)},\ \dots \qquad (3)$$

есть перечисление (нумерация) всех вычислимых функций без повторений.

Аналогичным образом из нумерации (3) можно выделить нумерацию всех вычислимых функций от n аргументов:

$$\varphi^{(n)}_{g(0)},\quad \varphi^{(n)}_{g(1)},\quad \varphi^{(n)}_{g(2)},\ \dots \qquad (4)$$

Введём здесь два обозначения, которые будем использовать до конца книги:

$W_x = Dom\ (\varphi_x)$ – область определения вычислимой функции φ_x, которая в нумерации (2) имеет номер x, т.е. вычисляется алгоритмом $\mathbf{A}_x$ с номером x;

$E_x = Im\ (\varphi_x)$ – область значений вычислимой функции φ_x, которая в нумерации (2) имеет номер x, т.е. вычисляется алгоритмом $\mathbf{A}_x$ с номером x.

Если мы хотим подчеркнуть, что речь идёт о функциях n аргументов, то будем писать $W^{(n)}_x$ и $E^{(n)}_x$.

Существование всюду определённой невычислимой функции. Используя нумерацию (4), докажем следующую теорему.

ТЕОРЕМА 5.1.2. *Существует невычислимая всюду определённая функция.*

Доказательство. Рассмотрим перечисление (нумерацию) *всех* вычислимых функций от одного аргумента:

$$\varphi_0^{(1)},\ \varphi_1^{(1)},\ \varphi_2^{(1)},\ \ldots\ \varphi_n^{(1)},\ \ldots \tag{5}$$

Построим всюду определённую функцию f, одновременно отличающуюся от каждой функции из функций последовательности (5). Определим эту функцию так:

$$f(n) = \begin{cases} \varphi_n^{(1)}(n) + 1, & \text{если значение } \varphi_n^{(1)}(n) \text{ определено,} \\ 0\ , & \text{если значение } \varphi_n^{(1)}(n) \text{ не определено.} \end{cases}$$

Увидим, что для каждого n функция f отличается от $\varphi_n^{(1)}$ в точке n. В самом деле, если $\varphi_n^{(1)}(n)$ определено, то f отличается от $\varphi_n^{(1)}$ тем, что $f(n) \neq \varphi_n^{(1)}(n)$ (так как в этом случае $f(n) = \varphi_n^{(1)}(n) + 1$). Если же $\varphi_n^{(1)}(n)$ не определено, то f отличается от $\varphi_n^{(1)}$ тем, что значение $f(n)$ определено.

Таким образом, функция f всюду определена и не входит в последовательность (5), в которой перечислены *все* одноместные вычислимые функции. Следовательно, f – всюду определённая и невычислимая функция. □

Диагональный метод в теории алгоритмов. Доказательство предыдущей теоремы является примером применения так называемого диагонального метода, или метода диагонализации, открытого Георгом Кантором, который впервые применил его для доказательства несчётности множества действительных чисел. Лежащая в его основе идея применима к огромному числу ситуаций, в частности, и в теории алгоритмов. Мы ещё не раз используем её для доказательства теорем, относящихся к вычислимости и разрешимости.

Название метода связано с тем, что если значения функций

$$\varphi_0^{(1)},\ \varphi_1^{(1)},\ \varphi_2^{(1)},\ \ldots\ ,\ \varphi_n^{(1)},\ \ldots$$

расположить в виде следующей бесконечной таблицы

	0	1	2	3	4	...
φ_0	$\varphi_0(0)$	$\varphi_0(1)$	$\varphi_0(2)$	$\varphi_0(3)$	...	
φ_1	$\varphi_1(0)$	$\varphi_1(1)$	$\varphi_1(2)$	$\varphi_1(3)$		
φ_2	$\varphi_2(0)$	$\varphi_2(1)$	$\varphi_2(2)$	$\varphi_2(3)$		
φ_3	$\varphi_3(0)$	$\varphi_3(1)$	$\varphi_3(2)$	$\varphi_3(3)$		
...	...					

то из неё видно, что для построения функции f отбираются диагональные элементы этой таблицы (подчёркнуты) $\varphi_0(0)$, $\varphi_1(1)$, $\varphi_2(2)$, ... Затем они систематически изменяются так, что получаются значения $f(0)$, $f(1)$, $f(2)$, ... , такие, что для каждого n значение $f(n)$ отличается от $\varphi_n(n)$. (Заметим, что если значение $\varphi_n(m)$ не определено, то в таблице записано "*не определено*").

5.2. Теорема о параметризации и универсальные функции и алгоритмы

Теорема о параметризации (называемая также s-m-n-теоремой Клини), а также понятие универсальной функции и теоремы о ней являются двумя фундаментальными основами ("двумя китами") теории вычислимости. Эти результаты опираются на нумерации алгоритмов и вычислимых функций, рассмотренные в предыдущем параграфе 5.1.

Теорема о параметризации (s-m-n-теорема Клини). Пусть $f(x, y)$ – вычислимая функция (не обязательно всюду определённая). Придав аргументу x значение a (т.е. положив $x = a$), мы приходим к вычислимой функции g_a одного аргумента: $g_a(y) = f(a, y)$. Поскольку $g_a(y)$ вычислима, то в рассмотренной в предыдущем параграфе 5.1 нумерации всех вычислимых функций она имеет некоторый номер (индекс) b так что $g_a(y) = f(a, y) = \varphi_b(y)$. Следующая теорема показывает, что индекс b можно эффективно найти по a и тем самым эффективно уменьшить размерность функции при её вычислении.

ТЕОРЕМА 5.2.1 (теорема о параметризации; частный случай). *Для всякой вычислимой функции $f(x, y)$ существует всюду определённая вычислимая функция $k(x)$ такая, что $f(x, y) = \varphi_{k(x)}(y)$*

.

Доказательство. Для доказательства опишем всюду определённый алгоритм, вычисляющий требуемую функцию $k(x)$.

Так как функция $f(x, y)$ вычислима, то, в силу тезиса Тьюринга, существует машина Тьюринга $M(f)$, вычисляющая её. Используя эту машину, для каждого фиксированного a построим машину Тьюринга $M(a)$, вычисляющую функцию $g_a(y) = f(a, y)$. Эта машина начинает работать с конфигурации $q_1 011...10$ (y единиц). На первом этапе нужно подготовить на ленте условия для применения машины $M(f)$ для вычисления значения $f(a, y)$. Для этого нужно начальную конфигурацию переработать в следующую: $q011 \ldots 1011 \ldots 10$ (в первой группе a единиц, во второй – y единиц).

Строим машину Тьюринга, осуществляющую следующую переработку:

$$q_1 011...10 \implies q_0 011...1011...10 \quad \text{или} \quad q_1 01^y 0 \implies q_0 01^a 01^y 0\ .$$

Алгоритм работы этой машины Тьюринга сделаем таким. Сначала продвинемся по ленте вправо до конца данного массива из y единиц (применяем машину "правый сдвиг" $Б^+$ – см. пример 2.2.4). Затем правее этого массива выставим массив из a единиц (для этого нам понадобятся a букв в алфавите Q внутренних состояний машины: q_4, q_5, ... , q_{3+a}). Далее, вернёмся к ячейке с нулём, разделяющим эти два массива, и поменяем эти два массива местами, применив машину Тьюринга В (см. пример 2.2.5). Наконец, сделав левый сдвиг $Б^-$ (см. пример 2.2.3), вернёмся на левый конец массива из a единиц. Останов.

	$q_1 011...10 = q_1 01^y 0$	
$q_1 0 \to q_2 0$ П :	$0 q_2 11...100$	(y единиц)
$q_2 1 \to q_2 1$ П (y раз):	$011...1 q_2 00$	(y шагов)
$q_2 0 \to q_3 0$ П :	$011...10 q_3 0$	(y единиц)
$q_3 0 \to q_4 1$ П :	$011...101_4 0$	
$q_4 0 \to q_5 1$ П :	$011...1011 q_5 0$	
. . . .	. . . .	
$q_{3+a-1} 0 \to q_{3+a} 1$ П :	$011...1011...1 q_{3+a} 0 = 01^y 01^a q_{3+a} 0$	
$Б^-$	$01^y q 01^a 0$	
В	$01^a q 01^y 0$	
$Б^-$	$q_0 01^a 01^y 0$.	

Итак, на ленте машины Тьюринга имеем конфигурацию:

$$q011...1011...10 \ = \ q01^a01^y0.$$

С этого момента начинает работать машина $M(f)$, которая вычисляет значение $f(a,y) = g_a(y)$. Таким образом, композиция этих двух машин Тьюринга даёт требуемую машину $M(a)$.

В нашей единой (гёделевской) нумерации всех машин Тьюринга машина $M(a)$ имеет некоторый номер $k(a)$. Это есть номер $k(a)$ той функции $\varphi_{k(a)}(y)$, которую вычисляет машина $M(a)$. Но по построению машины $M(a)$, она вычисляет функцию $f(a,y)$. Таким образом, $\varphi_{k(a)}(y) = f(a,y)$. Поскольку машина $M(f)$ фиксирована, и нумерация машин Тьюринга эффективна (по номеру восстанавливается машина), функция k оказывается эффективно вычислимой.

Теорема доказана. □

Эта теорема называется теоремой о параметризации потому, что показывает, что индекс $k(x)$ вычислимой функции $f(x,y)$ можно эффективно найти по параметру x, от которого он эффективно зависит.

Обобщение рассмотренной теоремы 5.2.1 состоит в замене единственных переменных x и y на m-наборы и n-наборы $x_1, x_2, \dots, x_m$ и $y_1, y_2, \dots, y_n$ соответственно, а также в рассмотрении вычислимой функции $\varphi_e(x_1, \dots, x_m, y_1, \dots, y_m)$ общего вида из перечисления всех вычислимых функций (e – её номер в этом перечислении). Мы приходим к следующей теореме.

Теорема 5.2.2 (s-m-n-теорема Клини). *Для всяких* $m, n \geq 1$ *существует всюду определённая вычислимая* $(m+1)$*-местная функция* $s_n^m(e, x_1, \dots, x_m)$ *такая, что*

$$\varphi_e(x_1, \dots, x_m, y_1, \dots, y_m) \ = \ \varphi_{s_n^m(e,x_1,\dots,x_m)}(y_1, \dots, y_n) \ .$$

Название этой теоремы связано с обозначением s_n^m для функции, существование которой утверждается в теореме. Она имеет много важных применений в теории алгоритмов.

Универсальные функции и алгоритмы. Пусть $K^{(n)}$ – некоторый класс n-местных функций (вообще говоря, частичных). *Универсальной функцией* для этого класса называется такая $(n+1)$-местная функция $F(y, x_1, \dots, x_n)$, которая удовлетворяет следующим двум условиям:

а) для любого фиксированного a функция n аргументов $F(a, x_1, \dots, x_n)$ принадлежит классу $K^{(n)}$;

б) для любой функции $f(x_1, ..., x_n) \in K^{(n)}$ существует такое значение b, что $F(b, x_1, ..., x_n) = f(x_1, ..., x_n)$.

Например, для семейства степенных функций одного аргумента $K^{(n)} = \{x^n : n = 0, 1, 2, ...\}$ универсальной будет функция двух аргументов $F(y, x) = x^y$.

Можно сказать, что универсальная для данного класса функция – это это функция, в определённом смысле порождающая все функции данного класса, служащая некоей компактной характеристикой целого данного класса функций. В свою очередь, алгоритмы, вычисляющие универсальные функции и называемые *универсальными алгоритмами*, в некотором смысле реализуют (включают в себя) все другие алгоритмы – для вычисления всех функций данного класса. Поэтому универсальные функции и универсальные алгоритмы являются мощным инструментом при доказательстве многих теорем и, в частности, при построении специальных невычислимых функций и неразрешимых предикатов.

Возникает вопрос, будет ли вычислима универсальная функция для класса *всех* n-местных вычислимых функций, т.е. существует ли универсальный алгоритм, вычисляющий эту функцию? Если "да", то всё сказанное в предыдущем абзаце об универсальных функциях и алгоритмах становится справедливым. Положительный ответ на этот, на первый взгляд казалось бы невероятный вопрос, даёт следующая теорема, главным инструментом доказательства которой является всё та же нумерация вычислимых функций (и машин Тьюринга), построенная в предыдущем параграфе 5.1.

Теорема 5.2.3 *Для любого $n \in N$ класс всех n-местных вычислимых функций обладает вычислимой универсальной функцией.*

Доказательство. Напомним: $\varphi_0, \varphi_1, \varphi_2, \dots, \varphi_k, \dots$ – нумерация (перечисление, последовательность) всех (в том числе и частичных) вычислимых функций от n аргументов. Тогда $(n+1)$-местная универсальная функция F для неё определяется следующим образом: для любого k положим: $F(k, x_1, ..., x_n) = \varphi_k(x_1, ..., x_n)$. Покажем, что сама функция F вычислима, указав алгоритм для её вычисления. Он состоит в следующем. Пусть $(y, x_1, ..., x_n)$ – произвольный набор натуральных чисел. По числу y находим алгоритм $\mathbf{A}_y$, вычисляющий функцию φ_y. Применяем этот алгоритм для вычисления функции φ_y на наборе $x_1, ..., x_n$, т.е. вычисляем значение $\varphi_y(x_1, ..., x_n)$. Это значение и есть, по определению F, значение функции F на наборе $(y, x_1, ..., x_n)$: $F(y, x_1, ..., x_n) = \varphi_y(x_1, ..., x_n)$.

Таким образом, алгоритм, вычисляющий функцию F, представляет собой композицию двух алгоритмов: алгоритма, отыскивающего вычислимую функцию φ_y в перечне (нумерации) всех вычислимых функций от n аргументов, и алгоритма $\mathbf{A}_y$, вычисляющего функцию φ_y. Следовательно, функция F вычислима.

Теорема доказана. □

Напомним (определение 5.1.1), что предикат $P(x_1, x_2, ..., x_n)$ называется ***разрешимым*** (или ***вычислимым***, или ***рекурсивным***, или ***рекурсивно разрешимым***), если вычислима его характеристическая функция

$$\chi_P(x_1, x_2, ..., x_n) =$$

$$= \begin{cases} 1, & \text{если высказывание } P(x_1, x_2, ..., x_n) \text{ — истинно,} \\ 0, & \text{если высказывание } P(x_1, x_2, ..., x_n) \text{ — ложно.} \end{cases}$$

Примеры разрешимых предикатов, которые будут использованы в дальнейшем, приводятся в следующем следствии из теоремы 5.2.3 о вычислимости универсальной функции для класса всех n-местных вычислимых функций.

СЛЕДСТВИЕ 5.2.4. ***Для всякого*** $n \geq 1$ ***следующие предикаты разрешимы:***

а) $S_n(e, x_1, ..., x_n, y, t)$: *"Алгоритм A_e, вычисляющий функцию $\varphi_e(x_1, ..., x_n)$, на начальных данных $x_1, ..., x_n$ выдаст результат y за t или меньше шагов"*;

б) $H_n(e, x_1, ..., x_n, t)$: *"Алгоритм A_e, вычисляющий функцию $\varphi_e(x_1, ..., x_n)$, на начальных данных $x_1, ..., x_n$ закончит работу за t или меньше шагов"*.

Возможность эффективно отвечать на эти вопросы обусловлена возможностью вычислять значения функций $\varphi_e(x_1, ..., x_n)$ (для всевозможных e) через посредство вычислений значений единой (универсальной) функции $F(e, x_1, ..., x_n) = \varphi_e(x_1, ..., x_n)$, что гарантируется предыдущей теоремой.

Тщательные доказательства теоремы 5.2.3 и следствия 5.2.4 приведены в [60], стр. 93 – 95, 102 – 105. □

Переформулируем доказанную теорему 5.2.3 для случая одноместных вычислимых функций, учитывая имеющиеся у нас нумерацию таких функций, а также нумерацию двухместных вычислимых функций, из числа которых будет выбираться универсальная функция. Пусть:

$$\varphi_0(x), \quad \varphi_1(x), \quad \varphi_1(x), \ \dots , \quad \varphi_k(x), \ \dots \ -$$

нумерация всех одноместных вычислимых функций. Существует такая двухместная вычислимая функция $F(y,x)$ (и значит, существует её номер z в нумерации всех двухместных вычислимых функций, так что $F(y,x) = \varphi_z(y,x)$), что $F(y,x) = \varphi_y(x)$, или $\boxed{\varphi_z(y,x) = \varphi_y(x)}$. (Напомним, что последнее равенство понимается в условном смысле: если $\varphi_y(x)$ определено, то $\varphi_z(y,x)$ определено и $\varphi_z(y,x) = \varphi_y(x)$; если же $\varphi_y(x)$ не определено, то и $\varphi_z(y,x)$ не определено).

Функция $\varphi_z(y,x)$ называется ***универсальной функцией*** для класса всех вычислимых функций одного аргумента. Алгоритм (программа) $\mathbf{A}_z$, вычисляющий универсальную функцию $\varphi_z(y,x)$, называется ***универсальным алгоритмом***, или ***универсальной программой***. Теорему о существовании вычислимой универсальной функции называют иногда ***теоремой о нумерации***. Эта теорема имеет нетривиальное практическое значение. Она показывает, что при вычислении функций одного аргумента имеется такая критическая степень "технической сложности" (сложность алгоритма $\mathbf{A}_z$), что всякая дальнейшая сложность – сверх этой критической – может быть поглощена возрастающим размером программы и памяти. В этом смысле алгоритм (программу) $\mathbf{A}_z$ называют также ***универсальной машиной***.

Следующая теорема показывает, что если класс всех вычислимых функций сузить до класса всех всюду определённых вычислимых функций, то полученный класс универсальной функцией не обладает.

Теорема 5.2.5 ***Класс всех всюду определённых вычислимых функций одного аргумента не имеет всюду определённой вычислимой универсальной функции.***

Доказательство. Допустим противное, т.е. такая функция $G(y,x)$ существует. Рассмотрим тогда следующую функцию одного аргумента: $g(x) = G(x,x)+1$. Ясно, что она является вычислимой и всюду определённой. Значит, из универсальности функции G следует, что найдётся такое b, что $G(b,x) = g(x)$. Так как $g(x)$ всюду определена, то она определена и на b: $g(b) = G(b,b)$. Но, с другой стороны, в силу определения $g(x)$ при $x = b$ имеем: $g(b) = G(b,b)+1$. Два последних равенства противоречат друг другу. Следовательно, сделанное допущение неверно, и теорема доказана. □

Заметим, что при доказательстве этой теоремы снова был использован диагональный метод Г.Кантора. Ясно, что аналогично может быть доказана соответствующая теорема для n-местных всюду определённых вычислимых функций.

Главные универсальные функции (главные нумерации). Каждая универсальная функция для класса одноместных вычислимых функций характеризует некоторую нумерацию этого класса, а по существу, сама является такой нумерацией: каждому натуральному числу y универсальная функция F ставит в соответствие одноместную вычислимую функцию $\varphi_y(x) = F(y,x)$.

Среди всевозможных универсальных функций для класса всех одноместных вычислимых функций выделяют так называемые главные универсальные функции, или главные нумерации.

Функция $U(y,x)$, универсальная для класса всех одноместных вычислимых функций, называется ***главной универсальной функцией*** для этого класса, если всякая двуместная вычислимая функция $v(y,x)$ может быть представлена в виде: $v(y,x) = U(c(y),x)$, где $c(y)$ – некоторая подходящая всюду определённая вычислимая одноместная функция. Можно сказать, что функция $c(y)$ по номеру y в нумерации v некоторой вычислимой одноместной функции φ даёт её номер $c(y)$ в главной универсальной нумерации U.

Можно доказать теорему о существовании главных нумераций: ***главная универсальная функция для класса всех одноместных вычислимых функций существует.***

После этого возникает вопрос о взаимоотношениях между главными универсальными функциями. Ясно, что если $U_1(y,x)$ и $U_2(y,x)$ – две главные универсальные функции для класса всех одноместных вычислимых функций, то, согласно определения, каждая из них получается из другой с помощью подстановки всюду определённых вычислимых функций:

$$U_1(y,x) = U_2(c_1(y),x) \quad \text{и} \quad U_2(y,x) = U_1(c_2(y),x) \quad .$$

Можно доказать, что в этом случае в качестве функций $c_1(y)$ и $c_2(y)$ могут быть подобраны две взаимно обратные функции. Этот факт носит название ***теоремы об изоморфизме главных нумераций.***

Эффективные операции над вычислимыми функциями. Теорема о параметризации (s-m-n-теорема 5.2.2) и вычислимость универсальных функций позволяют установить в некотором смысле эффективность ряда операций над бесконечными объектами (функциями, множествами). Эти объекты, казалось бы, лежат в стороне даже от нашего неформального понятия вычислимости, которое,

очевидно, применяется к конечным объектам. Тем не менее, как мы убедимся, многие операции являются эффективными, если их рассматривать как операции над индексами (номерами) затрагиваемых объектов.

Эффективность произведения функций. Суть этой эффективности состоит в том, что индекс (номер в нумерации) функции $\varphi_x \cdot \varphi_y$, являющейся произведением двух функций φ_x и φ_y, эффективно находится по индексам x и y перемножаемых функций.

Теорема 5.2.6 *Существует всюду определённая вычислимая функция* $s(x, y)$ *такая, что* $\boxed{\varphi_{s(x,y)} = \varphi_x \cdot \varphi_y}$ *для всех* x, y.

Доказательство. Рассмотрим (двухместную) универсальную функцию F для класса всех одноместных вычислимых функций, так что

$$F(x, z) = \varphi_x(z) \quad \text{и} \quad F(y, z) = \varphi_y(z).$$

Тогда функция, являющаяся произведением функций φ_x и φ_y, имеет вид:

$$\varphi_x(z) \cdot \varphi_y(z) = F(x, z) \cdot F(y, z) = f(x, y, z).$$

По теореме 5.2.3, универсальные функции $F(x, z)$ и $F(y, z)$ вычислимы. Значит, вычислимо и их произведение $f(x, y, z)$. Но тогда по теореме о параметризации (s-m-n-теореме Клини 5.2.2) существует всюду определённая вычислимая функция $s(x, y)$ такая, что $f(x, y, z) = \varphi_{s(x,y)}(z)$. Следовательно, $\varphi_x \cdot \varphi_y = \varphi_{s(x,y)}(z)$, т.е. произведение $\varphi_x \cdot \varphi_y$ есть функция, имеющая в нашей нумерации одноместных вычислимых функций эффективно вычисляемый номер $s(x, y)$. □

В частности, полагая $g(x) = s(x, x)$, где функция s берётся из предыдущей теоремы, имеем для квадрата функции: $(\varphi_x)^2 = \varphi_{g(x)}$.

Теорема 5.2.7 *Существует такая всюду определённая вычислимая функция* $s(x, y)$, *область определения которой совпадает с объединением областей определения функций* φ_x *и* φ_y, *т.е.*

$$\boxed{W_{s(x,y)} = W_x \cup W_y}\,.$$

Доказательство. Рассмотрим следующую функцию:

$$f(x, y, z) = \begin{cases} 1, & \text{если } z \in W_x \text{ или } z \in W_y, \\ \text{не определено}, & \text{в противном случае.} \end{cases}$$

Эта функция вычислима, а значит, по теореме о параметризации (s-m-n-теореме Клини 5.2.2), существует всюду определённая вычислимая функция $s(x,y)$ такая, что $f(x,y,z) = \varphi_{s(x,y)}(z)$. Из определения функции $f(x,y,z)$ теперь видно, что функция $\varphi_{s(x,y)}(z)$ будет определена для таких и только таких z (т.е. $z \in W_{s(x,y)}$), которые принадлежат либо W_x, либо W_y, т.е. $W_{s(x,y)} = W_x \cup W_y$. □

Эффективность операций обращения и суперпозиции функций. Можно показать, что если φ_x – взаимно однозначная (вычислимая) функция, то существует такая всюду определённая вычислимая функция $k(x)$, что функцией, обратной для φ_x будет функция $\varphi_{k(x)}$, т.е. $\varphi_x^{-1} = \varphi_{k(x)}$; причём, область определения функции φ_x будет областью значений функции $\varphi_{k(x)}$, т.е. $W_x = E_{k(x)}$.

Аналогично можно показать, что если φ_x и φ_y – две вычислимые функции одного аргумента, то существует такая всюду определённая вычислимая функция $s(x,y)$, которая по номерам x и y этих функций даёт номер $s(x,y)$ их суперпозиции (композиции) $\varphi_y \circ \varphi_x$. Причём, если $U(n,z)$ – двуместная главная универсальная функция для класса всех вычислимых функций одного аргумента, то функция $s(x,y)$ удовлетворяет соотношению:

$$U(s(x,y),z) = U(x,U(y,z))$$

для всех x, y, z.

Эффективность рекурсивного определения. Рассмотрим $(n+3)$-местную функцию f, определённую посредством следующих рекурсивных соотношений:

$$f(e_1,e_2,\overline{x},0) = \varphi_{e_1}^{(n)}(\overline{x}) ,$$

$$f(e_1,e_2,\overline{x},y+1) = \varphi_{e_2}^{(n+2)}(\overline{x},y,f(e_1,e_2,\overline{x},y)) ,$$

где $\overline{x} = (x_1,x_2,...,x_n)$. Выразим функции в правых частях этих равенств через универсальные функции $F^{(n)}$ и $F^{(n+2)}$ для совокупностей всех n-местных и $(n+2)$-местных вычислимых функций соответственно. Получим:

$$f(e_1,e_2,\overline{x},0) = F^{(n)}(e_1,\overline{x}) ,$$

$$f(e_1,e_2,\overline{x},y+1) = F^{(n+2)}(e_2,\overline{x},y,f(e_1,e_2,\overline{x},y)) .$$

Так как универсальные функции $F^{(n)}$ и $F^{(n+2)}$ вычислимы (теорема 5.2.3), и функция f определена по рекурсии с вычислимыми функциями, поэтому функция f вычислима.

Тогда по теореме о параметризации (s-m-n-теореме Клини 5.2.2) существует всюду определённая вычислимая функция $s(x,y)$ такая, что

$$\varphi^{(n+1)}_{s(e_1,e_2)}(\overline{x},y) \;=\; f(e_1,e_2,\overline{x},y)\ .$$

Это и означает, что $s(e_1,e_2)$ есть индекс (номер) $(n+1)$-местной функции, получаемой из функций $\varphi^{(n)}_{e_1}$ и $\varphi^{(n+2)}_{e_2}$ рекурсией (для произвольных, но фиксированных e_1, e_2).

5.3. Теорема о неподвижной точке и её применения

Теорема о неподвижной точке (называемая также теоремой о рекурсии), доказанная С.Клини, является одной из фундаментальных теорем в теории вычислимых функций, имеющей глубокие приложения в теоретическом программировании. Она даёт простой и действенный общий метод порождения различных патологических структур в теории вычислимых функций. Эта теорема может быть сформулирована в нескольких различных формах, а на интуитивном уровне может интерпретироваться с нескольких точек зрения. В определённом смысле теорема суммирует целый класс диагональных методов, которые мы уже использовали выше и будем использовать в дальнейшем. С другой стороны, эта теорема устанавливает некоторый результат о неподвижной точке и, подобно теоремам анализа о неподвижной точке, может быть использована для доказательства существования многих неявно заданных функций.

Теорема о неподвижной точке. Мы уже отметили, что априори ясно, что в общей нумерации всех одноместных вычислимых функций каждая функция получает не единственный номер ввиду того, что эта нумерация порождается нумерацией алгоритмов, вычисляющих функции, а одна и та же функция, конечно же, может быть вычислена разными алгоритмами. Теорема о неподвижной точке, по существу, выявляет одну закономерность в этой расплывчатой неоднозначности.

ТЕОРЕМА 5.3.1 (С.Клини о неподвижной точке). *Если f – всюду определённая одноместная вычислимая функция, то существует такое число n, что под номерами n и $f(n)$ в нумерации одноместных вычислимых функций стоит одна и та же функция:*

$$\boxed{\varphi_{f(n)} = \varphi_n}\ .$$

Доказательство. Рассмотрим функцию: $\varphi_{f(\varphi_x(x))}(y)$. Её можно рассматривать как функцию двух аргументов x и y (двухместную функцию):

$$\varphi_{f(\varphi_x(x))}(y) = \psi(f(\varphi_x(x),y) = g(x,y) \ .$$

(Если $\varphi_x(x)$ не определено, то мы считаем, что функция g для таких x, y не определена). Тогда по теореме 5.2.1 (s-m-n-теорема Клини о параметризации), существует всюду определённая вычислимая функция $s(x)$ такая, что для всех x будет иметь место условное равенство:

$$\varphi_{f(\varphi_x(x))}(y) = \varphi_{s(x)}(y) \ . \qquad (*)$$

Возьмём теперь какой-нибудь номер m функции s в нумерации всех одноместных вычислимых функций, т.е. такое m, что $s = \varphi_m$. Тогда предыдущее равенство $(*)$ примет вид:

$$\varphi_{f(\varphi_x(x))}(y) = \varphi_{\varphi_m(x)}(y) \ . \qquad (**)$$

Придадим теперь переменной x значение m, т.е. положим $x = m$. Так как функция $s(x) = \varphi_m(x)$ всюду определена, то при $x = m$ для неё имеем $\varphi_m(m) = n$ – какое-то значение. Тогда равенство $(**)$ принимает вид: $\varphi_{f(n)}(y) = \varphi_n(y)$, что и требовалось доказать. □

Отметим, что проделанное доказательство никоим образом не зависит от выбранной нумерации вычислимых функций, поскольку основывается только на s-m-n-теореме и вычислимости универсальной функции. Ни один из этих результатов, в свою очередь, не зависел от особенностей выбранной нумерации вычислимых функций.

Число n в этой теореме называется неподвижной точкой для функции f: но здесь равны не $f(n)$ и n, а нумеруемые этими числами всюду определённые вычислимые функции: $\varphi_{f(n)} = \varphi_n$.

Рассмотрим сначала ряд важных следствий из этой теоремы, а затем обсудим некоторые её интерпретации.

Следствия из теоремы о неподвижной точке. СЛЕДСТВИЕ 5.3.2. *Если f – всюду определённая одноместная вычислимая функция, то существует такое число n, что $W_{f(n)} = W_n$.*

Доказательство. Напомним, что W_x – область определения функции φ_x. Поэтому раз по теореме $\varphi_{f(n)} = \varphi_n$, то и $W_{f(n)} = W_n$. □

СЛЕДСТВИЕ 5.3.3. *Если f – всюду определённая одноместная вычислимая функция, то существуют сколь угодно большие числа n, такие, что* $\boxed{\varphi_{f(n)} = \varphi_n}$.

Доказательство. Зафиксируем произвольное натуральное число k и возьмём такую функцию φ_c, которая отличается от всех функций φ_0, φ_1, φ_2, ... , φ_k. Рассмотрим функцию g, определяемую следующим образом:

$$g(x) = \begin{cases} c\,, & \text{если } x \le k, \\ f(x)\,, & \text{если } x > k. \end{cases}$$

Ясно, что функция g всюду определена и вычислима, так как всюду определены и вычислимы функции c и f. Тогда по теореме 5.3.1, она обладает неподвижной точкой, т.е. существует такое число n, что $\varphi_{g(n)} = \varphi_n$. Не может быть, чтобы $0 \le n \le k$, ибо в этом случае было бы: $g(n) = c$ и $\varphi_{g(n)} = \varphi_c$, но φ_c не равна ни одной функции φ_0, φ_1, φ_2, ... , φ_k, в том числе и функции φ_n, т.е. $\varphi_c \ne \varphi_n$. Но тогда $\varphi_{g(n)} \ne \varphi_n$, что не так. А раз так, то по определению функции g, имеем $g(n) = f(n)$, и значит, $\varphi_{f(n)} = \varphi_n$, т.е. n – неподвижная точка для f. Таким образом, для любого наперёд заданного (как угодно большого) натурального числа k мы нашли натуральное число n, являющееся неподвижной точкой для f. Что и требовалось доказать. □

СЛЕДСТВИЕ 5.3.4. ***Для любой всюду определённой двухместной вычислимой функции $f(x,y)$ существует такой номер (индекс) e в нумерации одноместных вычислимых функций, что***

$$\boxed{\varphi_e(y) = f(e,y)}\,.$$

Доказательство. Применим теорему 5.2.1 (s-m-n-теорему Клини о параметризации) к функции $f(x,y)$. В силу неё существует такая всюду определённая вычислимая функция $s(x)$, что $\varphi_{s(x)}(y) = f(x,y)$. По теореме 5.3.1, функция $s(x)$ обладает неподвижной точкой e: $\varphi_{s(e)} = \varphi_e$. Подставим значение e в предыдущее равенство вместо x: $\varphi_{s(e)}(y) = f(e,y)$. Заменив здесь $\varphi_{s(e)}$ на φ_e, получаем требуемое равенство: $\varphi_e(y) = f(e,y)$. □

Рассмотрим, например, семейство степенных функций $\{x^m : m \in N\}$. Универсальной функцией для него служит функция двух аргументов: $f(m,x) = x^m$. Применим к этой функции следствие 5.3.4. Согласно нему, найдётся такой номер n, что $\varphi_n(x) = f(n,x) = x^n$, т.е найдётся такое число n, что номером функции x^n в нашей нумерации вычислимых функций будет её показатель степени.

Следующее следствие из теоремы о неподвижной точке показы-

вает, что "естественная" нумерация вычислимых функций без повторений, которая была введена в параграфе 5.1, невычислима.

СЛЕДСТВИЕ 5.3.5. *Предположим, что f – всюду определённая возрастающая функция, удовлетворяющая следующим условиям:*

а) *если* $m \neq n$, *то* $\varphi_{f(m)} \neq \varphi_{f(n)}$;

б) $f(n)$ *является наименьшим индексом функции* $\varphi_{f(n)}$.

Тогда функция f невычислима.

Доказательство. Пусть f удовлетворяет условиям теоремы. Из условия а) следует, что функция f не может быть тождественной, а так как, по условию, она возрастающая, то должно найтись такое число k, что $f(n) > n$ при $n \geq k$. Тогда согласно условия б), $\varphi_{f(n)} \neq \varphi_n$ при всех $n \geq k$.

Допустим теперь, что функция f вычислима. Тогда по следствию 5.3.3, существует такое $n \geq k$, что $\varphi_{f(n)} = \varphi_n$. Это противоречит выводу предыдущего абзаца. Значит, f невычислима.

Следствие доказано. □

Неформальные интерпретации теоремы о неподвижной точке. Мы уже отмечали, что теорема 5.3.1 не является в полном смысле слова – о *неподвижной* точке: функция f не индуцирует отображение на классе всех вычислимых функций, для которого функция φ_n, переходя при этом отображении в (равную ей) функцию $\varphi_{f(n)}$, оставалась бы неподвижной точкой при этом отображении. Это можно аргументировать следующими соображениями. С функцией f можно связать отображение f^* на множестве всех алгоритмов, вычисляющих одноместные функции: $f^*(\mathbf{A}_x) = \mathbf{A}_{f(x)}$. Существование неподвижной функции привело бы к существованию неподвижного алгоритма $\mathbf{A}_n$: $f^*(\mathbf{A}_n) = \mathbf{A}_n$, т.е $\mathbf{A}_{f(n)} = \mathbf{A}_n$, т.е алгоритмы $\mathbf{A}_{F(n)}$ и $\mathbf{A}_n$ были бы абсолютно одинаковыми (тождественными, идентичными). Но это не так: алгоритмы все разные, но могут быть два таких, которые по-разному вычисляют одну и ту же функцию. Именно такими являются алгоритмы $\mathbf{A}_{F(n)}$ и $\mathbf{A}_n$ для числа n, являющегося неподвижной точкой для функции f в смысле теоремы 5.3.1: $\varphi_{f(n)} = \varphi_n$. Поэтому теорему 5.3.1 называют также теоремой о *псевдонеподвижной* точке.

Второе своё название – "теорема о рекурсии" – теорема 5.3.1 получила потому, что выражает очень общее определение "по рекурсии". Функция φ_n с помощью равенства $\varphi_n = \varphi_{f(n)}$ определяется эффективно в терминах алгоритма, имеющего номер $f(n)$, для вычисления её через саму себя. Несмотря на то, что это определение выглядит как имеющее порочный круг, согласно теоремы

5.3.1 о рекурсии, мы можем утверждать, что вычислимые функции φ_n, удовлетворяющие такому определению, *существуют*.

Пример применения теоремы о неподвижной точке: существование алгоритма, печатающего свой собственный текст. Это ещё одна неформальная интерпретация обсуждаемой теоремы. Рассмотрим функцию f, которая каждому натуральному числу x ставит в соответствие номер $f(x)$ алгоритма $\mathbf{A}_{f(x)}$, который печатает текст алгоритма $\mathbf{A}_x$ с номером x. Функция f вычислима. Тогда по теореме 5.3.1, существует число n такое, что алгоритмы $\mathbf{A}_{f(n)}$ и $\mathbf{A}_n$ реализуют (вычисляют) одну и ту же функцию. Следовательно, алгоритм (машина) $\mathbf{A}_n$ печатает свой собственный текст (свою собственную программу).

Глава VI

РАЗРЕШИМОСТЬ И ПЕРЕЧИСЛИМОСТЬ МНОЖЕСТВ

Происхождение проблемы. Существуют, как известно, два основных способа задания множества:

а) указанием характеристического свойства его элементов: $M = \{x: \ P(x)\}$;

б) перечислением всех его элементов: $M = \{a_1, a_2, ...a_n\}$.

Зная теперь, что значит уметь "эффективно" вычислять ту или иную функцию, нам хотелось бы под тем же углом зрения, т.е. с точки зрения "эффективности" посмотреть и на способы задания множеств.

С интуитивной точки зрения первый способ задания множества должен быть таким, чтобы характеристическое свойство было "эффективно" распознаваемо, а второй должен быть таким, чтобы перечисление было "в самом деле перечислением", т.е. происходило с помощью некоторого перечисляющего алгоритма.

Эти соображения приводят к понятиям разрешимого (рекурсивного) и перечислимого (рекурсивно перечислимого) множества соответственно.

Важность понятий разрешимости и перечислимости множеств для оснований математики связана с тем, что язык теории множеств является в известном смысле универсальным языком математики. Всякому математическому утверждению можно придать вид утверждения о каких-либо множествах. В связи с этим к способам задания множеств предъявляются повышенные требования. Необходимо точное понимание таких понятий как "конструктивный способ задания множества" и "множество, заданное эффективно". Это и достигается, благодаря понятиям разрешимости и перечислимости множества. Язык разрешимых и перечислимых множеств является универсальным языком для утверждений о существовании (или отсутствии) алгоритмов решения математических проблем.

6.1. Разрешимые множества и их свойства

Понятие разрешимого множества и примеры. Пусть множество $M \subseteq N^n = N \times ... \times N$, т.е. M является подмножеством декартовой n-ой степени множества N всех натуральных чисел.

ОПРЕДЕЛЕНИЕ 6.1.1. Множество   называется ***разрешимым***, если существует алгоритм $\mathbf{A}_M$, который по любому объекту a даёт ответ, принадлежит a множеству M или нет. Алгоритм $\mathbf{A}_M$ называется ***разрешающим алгоритмом*** для M.

Другими словами, разрешимое множество – это множество с алгоритмически разрешимой проблемой вхождения.

Напомним понятие ***характеристической функции*** множества M. Ей называется функция χ_M, заданная на множестве M и принимающая значения в двухэлементном множестве $\{0, 1\}$, определяемая следующим образом:

$$\chi_M(x_1, ..., x_n) = \begin{cases} 0, & \text{если } (x_1, ..., x_n) \notin M, \\ 1, & \text{если } (x_1, ..., x_n) \in M. \end{cases}$$

Отсюда ясно, что ***множество M разрешимо тогда и только тогда, когда его характеристическая функция χ_M вычислима.***

Согласно тезису Чёрча вычислимость функции χ_M означает её частичную рекурсивность (а значит, и её вычислимость по Тьюрингу, и её нормальную вычислимость – см. теорему 4.2.4). Поэтому разрешимые множества называют также ***частично рекурсивными*** (или ***рекурсивными***).

Поскольку имеет место включение **ПРФ** $\subset$ **ЧРФ** (см. § 3.1, пункт "Тезис Чёрча"), поэтому всякое множество M с примитивно рекурсивной характеристической функцией χ_M (такое множество, естественно назвать ***примитивно рекурсивным***) будет рекурсивным, т.е. разрешимым. Таким образом, мы приходим к следующей теореме.

ТЕОРЕМА 6.1.2. ***Всякое примитивно рекурсивное множество разрешимо.***

Отсюда вытекают ПРИМЕРЫ РАЗРЕШИМЫХ МНОЖЕСТВ:

1) Множество N_2 всех чётных натуральных чисел: $N_2 = \{2k : k \in N\}$. Его характеристическая функция: $\chi_{N_2} = \overline{sg}\,(x \dot{-} 2 \cdot [x/2])$.

2) Множество всех простых чисел: $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, \ldots\}$.

3) Множество всех чисел, являющихся квадратами натуральных чисел: $\{k^2 : k \in N, k \neq 0\} = \{1, 4, 9, 16, 25, 36, \ldots\}$. Алгоритм проверки того, принадлежит ли данное число этому множеству, основывается на единственности разложения всякого натурального числа на простые множители.

4) Множество N всех натуральных чисел. Его характеристическая функция $\chi_N = 1$ есть const.

5) Множество всех нечётных чисел: $E = \{2k+1 : k \in N\}$. Его характеристическая функция: $\chi_E = sg\ (x \dot{-} 2 \cdot [x/2])$.

6) Всякое конечное множество $L = \{a_1, a_2, \ldots, a_n\} \subset N$. Его характеристическая функция: $\chi_L = \overline{sg}\ (\Pi_{i=1}^{n} |x - a_i|)$.

7) Множество всех тавтологий логики высказываний. Разрешающий алгоритм состоит в прямом вычислении значений данной формулы на всевозможных наборах значений её пропозициональных переменных (составление таблицы истинности формулы).

Свойства разрешимых множеств.

ТЕОРЕМА 6.1.3. *Объединение и пересечение разрешимых множеств являются множествами разрешимыми; дополнение к разрешимому множеству есть множество разрешимое.*

Доказательство. Пусть множества M и L – разрешимы, т.е. их характеристические функции $\chi_M(x)$, и $\chi_L(x)$ вычислимы. Проверьте тогда, что характеристические функции множеств $M \cup L$, $M \cap L$, $\overline{M}$ могут быть соответственно следующим образом представлены в виде суперпозиции этих функций и известных нам ранее функций:

$$\chi_{M \cup L}(x) = sg\ (\max(\chi_M(x),\ \chi_L(x))) = sg\ (\chi_M(x) + \chi_L(x))\ ;$$

$$\chi_{M \cap L}(x) = \chi_M(x) \cdot \chi_L(x)\ ;$$

$$\chi_{\overline{M}}(x) = \overline{sg}(\chi_M(x))\ .$$

Каждая из функций, участвующих в суперпозициях при построении функций $\chi_{M \cup L}$, $\chi_{M \cap L}$, $\chi_{\overline{M}}$, вычислима. Поскольку, как было отмечено выше, класс вычислимых функций замкнут относительно суперпозиции, поэтому и сами эти функции $\chi_{M \cup L}$, $\chi_{M \cap L}$, $\chi_{\overline{M}}$ вычислимы. А раз так, значит характеризуемые ими множества $M \cup L$, $M \cap L$, $\overline{M}$ разрешимы.

6.2. Перечислимые множества и их свойства

Понятие перечислимого множества и примеры. Пусть M есть подмножество множества N всех натуральных чисел.

Определение 6.2.1. Множество $M \subseteq N$ называется (*рекурсивно*, или *эффективно*, или *алгоритмически*) *перечислимым*, если M либо пусто, либо есть область значений некоторой вычислимой функции, или, другими словами, если существует алгоритм для последовательного порождения (перечисления) всех его элементов.

Другими словами, M перечислимо, если существует такая вычислимая функция $\psi_M(x)$, для которой M является её областью значений: $a \in M \iff (\exists x)\ (a = \psi_M(x))$.

При этом функция ψ_M называется *перечисляющей* множество M (или для множества M); соответственно алгоритм, вычисляющий ψ_M, называется *перечисляющим* или порождающим для M:

$$M = \{\psi_M(0),\ \psi_M(1),\ \psi_M(2),\ \dots\ \}\ .$$

Примеры перечислимых множеств:

1) Множество N_2 всех чётных натуральных чисел: $N_2 = \{2k : k \in N\}$. Его перечисляющая функция: $\psi_{N_2} = 2x$. Эта функция, как мы знаем, примитивно рекурсивна и, значит, вычислима. Поэтому множество N_2 перечислимо.

2) Рассмотрим множество $M = \{1, 4, 9, 16, 25, 36, \dots\}$ квадратов натуральных чисел. Оно перечислимо: для получения его элементов нужно последовательно брать числа $1, 2, 3, 4, \dots$ и возводить их в квадрат. Другими словами, M есть область значений вычислимой функции $f(x) = x^2$: $M = \{1^2, 2^2, 3^2, 4^2, \dots\}$. Более того, это множество является также и разрешимым: для проверки того, принадлежит или нет некоторое число данному множеству, нужно разложить число на простые множители, что позволит выяснить, является ли оно точным квадратом.

Свойства перечислимых множеств.

Теорема 6.2.2. *Объединение и пересечение перечислимых множеств являются множествами перечислимыми.*

Доказательство. Объединение множеств. Пусть множества M и L перечислимы. Это означает, что имеются алгоритмы $\mathbf{A}_M$ и $\mathbf{A}_L$ для порождения элементов множеств M и L:

$\mathbf{A}_M$ последовательно порождает элементы $m_1, m_2, m_3, \ldots$ множества M,

$\mathbf{A}_L$ последовательно порождает элементы $l_1, l_2, l_3, \ldots$ множества L.

Тогда алгоритм $\mathbf{A}_{M \cup L}$, порождающий множество $M \cup L$, получается из алгоритмов $\mathbf{A}_M$ и $\mathbf{A}_L$ в результате их одновременного применения: поочередно с помощью алгоритмов $\mathbf{A}_M$ и $\mathbf{A}_L$ порождаются элементы:

$$m_1, \ l_1, \ m_2, \ l_2, \ m_3, \ l_3, \ \ldots \ \text{и т.д.}$$

Следовательно, множество $M \cup L$ перечислимо.

Пересечение множеств. Пусть теперь алгоритм $\mathbf{A}_M$ последовательно порождает элементы $m_1, m_2, m_3, \ldots$ множества M, а алгоритм $\mathbf{A}_L$ последовательно порождает элементы $l_1, l_2, l_3, \ldots$ множества L. Тогда алгоритм для перечисления элементов множества $M \cap L$ заключается в следующем.

Поочередно с помощью алгоритмов $\mathbf{A}_M$ и $\mathbf{A}_L$ порождаются элементы

$$m_1, \ l_1, \ m_2, \ l_2, \ m_3, \ l_3, \ \ldots \ \text{и т.д.}$$

Каждый вновь порождённый элемент m_i сравнивается со всеми ранее порождёнными элементами $l_1, l_2, \ldots, l_{i-1}$. Если m_i совпадает с одним из них, то он включается во множество $M \cap L$. В противном случае переходим к порождению элемента l_i и сравниваем его со всеми ранее порождёнными элементами $m_1, m_2, \ldots, m_{i-1}, m_i$, и т.д. Описанная процедура позволяет эффективно перечислить все элементы множества $M \cap L$, что и доказывает его перечислимость. □

6.3. Взаимоотношения между разрешимыми и перечислимыми множествами

Сравнив теоремы 6.1.3 и 6.2.2, мы видим, что в последней остался открытым вопрос о перечислимости дополнения перечислимого множества. Оказывается, как мы увидим позже, так будет не всегда. Причина этого кроется в глубоком различии между разрешимыми множествами и перечислимыми множествами, которое будет обнаружено в этом разделе.

Разрешимые и перечислимые множества. Здесь будут доказаны две теоремы.

Теорема 6.3.1. *Всякое непустое разрешимое множество перечислимо.*

Доказательство. Пусть $M \neq \emptyset$ – непустое разрешимое множество. Тогда имеется элемент $a \in M$. Следовательно, характеристическая функция χ_M множества M, как было отмечено выше, вычислима, т.е. имеется алгоритм для её вычисления.

Строим алгоритм для перечисления множества M. Рассмотрим функцию:

$$\psi_M(x) = \begin{cases} x, \text{ если } \chi_M(x) = 1, \text{ т.е. если } x \in M, \\ a, \text{ если } \chi_M(x) = 0 \text{ т.е. если } x \notin M. \end{cases}$$

Функция $\psi_M(x)$ вычислима, т.к. вычислима функция $\chi_M(x)$ и $\psi_M(x)$ задана с помощью $\chi_M(x)$ кусочным образом (см теорему 3.3.20).

Кроме того, $M = \{\psi_M(0), \psi_M(1), \psi_M(2), \psi_M(3), \dots\}$, т.е. M есть множество значений вычислимой функции $\psi_M(x)$. Это и означает, что множество M перечислимо. □

Возникает вопрос, справедливо ли обратное утверждение. Следующая теорема делает важный шаг на пути его решения, но всё же пока даёт на него лишь косвенный отрицательный ответ.

Теорема 6.3.2 (Э.Л.Пост[1]). *Пусть $M \subseteq N$. Множество M разрешимо тогда и только тогда, когда оно само и его дополнение $\overline{M}$ перечислимы.*

Доказательство. Необходимость. Пусть M – непустое разрешимое множество. Тогда по теореме 5.3.1 оно перечислимо. С другой стороны, так как M – разрешимо, то по теореме 5.1.3, его дополнение $\overline{M}$ – также разрешимо. Следовательно, снова по теореме 5.3.1 множество $\overline{M}$ – перечислимо.

Итак, множества M и $\overline{M}$ – перечислимы.

Достаточность. Пусть множества M и $\overline{M}$ перечислимы, т.е.

$M = \{f(0), f(1), f(2), f(3), \dots\}$ и $\overline{M} = \{g(0), g(l), g(2), g(3), \dots\}$,

где f и g – некоторые вычислимые функции. Тогда алгоритм, выясняющий, принадлежит или нет произвольное число n множеству M, действует следующим образом. Последовательно порождаем

[1]ПОСТ Эмиль Леон (1897 - 1954) – американский математик и логик.

элементы $f(0)$, $g(0)$, $f(1)$, $g(1)$, $f(2)$, $g(2)$, ... и на каждом шаге получаемый элемент сравниваем с n. Поскольку n должно принадлежать либо M, либо $\overline{M}$, то на конечном шаге получим $f(k) = n$ или $g(k) = n$. Если $f(k) = n$, то $n \in M$, а если $g(k) = n$, то $n \in \overline{M}$ и, значит, $n \notin M$. Следовательно, множество M разрешимо.

Теорема доказана. □

Возникает вопрос, существуют ли перечислимые, но не разрешимые множества? Прежде, чем дать на него (положительный) ответ, рассмотрим одну интересную и важную математическую процедуру, которой мы воспользуемся при ответе на поставленный вопрос и которая используется не только в теории алгоритмов, но и в других разделах математики.

Канторовская нумерация упорядоченных пар натуральных чисел. Все упорядоченные пары из декартова произведения

$$N^2 = N \times N = \{(x,y): x \in N \wedge y \in N\}$$

множества N на себя можно расположить в последовательность и занумеровать натуральными числами так, что по номеру пары (x,y) однозначно может быть восстановлена сама пара, т.е. обе её компоненты x и y. Вот это расположение:

$$\begin{matrix} (0,0) & (0,1) & (0,2) & (0,3) & (0,4) & \dots \\ (1,0) & (1,1) & (1,2) & (1,3) & (1,4) & \dots \\ (2,0) & (2,1) & (2,2) & (2,3) & (2,4) & \dots \end{matrix}$$

Такой способ нумерации называется *диагональным методом* или *диагональным процессом*. Он изобретён немецким математиком Г.Кантором[2]. Перечисление осуществляется последовательным прохождением по конечным диагоналям бесконечного квадрата, начиная с левого верхнего угла. Пары идут в порядке возрастания суммы их членов, а из пар с одинаковой суммой членов ранее идет пара с меньшим первым членом:

---

[2]КАНТОР Георг (1845 – 1918) – немецкий математик, создатель теории бесконечных множеств и трансфинитных чисел.

(0,0); (0,1), (1,0); (0,2), (1,1), (2,0); (0,3), (1,2), (2,1), (3,0); . . . (*)

Обозначим $c(x, y)$ – номер пары (x, y) в последовательности (*):

$$c(0,0) = 0\ ,\quad c(0,1) = 1\ ,\quad c(1,0) = 2\ ,\quad c(0,2) = 3\ ,$$
$$c(1,1) = 4\ ,\quad c(2,0) = 5\ ,\ \ldots$$

Число $c(x, y)$ называется ***канторовским номером*** пары (x, y).

Обозначим, далее: $l(n)$ – левый член пары с номером n;

$r(n)$ – правый член пары с номером n.

Например: $l(5) = 2$, $r(5) = 0$.

Теорема 6.3.3. *Имеют место следующие выражения:*

$$c(x, y) = \frac{(x+y)(x+y+1)}{2} + x = \frac{(x+y)^2 + 3x + y}{2}\ ;$$

$$l(n) = x = n \dot{-} \frac{1}{2}\left[\frac{[\sqrt{8n+1}]+1}{2}\right]\cdot\left[\frac{[\sqrt{8n+1}] \dot{-} 1}{2}\right]\ ;$$

$$r(n) = y = \left[\frac{[\sqrt{8n+1}] \dot{-} 1}{2}\right]\cdot\left(1 + \frac{1}{2}\left[\frac{[\sqrt{8n+1}]+1}{2}\right]\right) \dot{-} n\ .$$

(где квадратные скобки [ ] обозначают целую часть числа, стоящего в скобках, т.е. наибольшее целое число, не превосходящее числа, стоящего в скобках).

Доказательство. ♠ Для функции $c(x, y)$. Пара (x, y) находится на отрезке: $(0, x+y), (1, x+y-1), \ldots, (x, y), \ldots, (x+y-1, 1), (x+y, 0)$
на x-ом месте после пары $(0, x+y)$. Перед парой $(0, x+y)$ в последовательности (*) имеется $x+y$ отрезков:

Номер (вес) отрезка	Пары, содержащиеся в отрезке	Число пар в отрезке
0	(0,0)	1
1	(0,1), (1,0)	2
2	(0,2), (1,1), (2,0)	3
3	(0,3), (1,2), (2,1), (3,0)	4
4	(0,4), (1,3), (2,2), (3,1), (4,0)	5
. . .	. . . .	. . .
$x+y-1$	$(0, x+y-1), (1, x+y-2), \ldots, (x+y-1, 0)$	$x+y$

Тогда общее число пар в последовательности (*) перед парой $(0, x+y)$ равно (сумма членов арифметической прогрессии):

$$1+2+3+\ldots+(x+y) = \frac{a_1+a_n}{2}\cdot n = \frac{(1+x+y)(x+y)}{2} .$$

Тогда номер пары (x, y) равен:

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

♠ Для функций $l(n)$ и $r(n)$.

Пусть $x = l(n)$, $y = r(n)$; следовательно, $n = c(x, y)$. Тогда из полученной формулы для $c(x, y)$ получаем: $2n = (x+y)^2 + 3x + y$. Обе части этого равенства умножим на 4 и прибавим к обеим частям 1. Получим:

$$8n+1 = 4(x^2+2xy+y^2)+12x+4y+1 , \qquad (1)$$

$$8n+1 = (4x^2+8xy+4y^2+4x+4y+1)+8x ,$$

$$8n+1 = (2x+2y+1)^2+8x .$$

Следовательно, $2x+2y+1 \le [\sqrt{8n+1}]$. (2)

Вернёмся к равенству (1) и преобразуем его правую часть несколько иначе:

$$8n+1 = (4x^2+8xy+4y^2+12x+12y+9)-8y-8 ,$$

$$8n+1 = (2x+2y+3)^2-8y-8 .$$

Отсюда: $[\sqrt{8n+1}] < 2x+2y+3$. (3)

Таким образом, соединяя неравенства (2) и (3), получаем оценку:

$$2x+2y+1 \le [\sqrt{8n+1}] < 2x+2y+3 .$$

Ко всем частям этих неравенств прибавим по 1 и поделим на 2. Получим:

$$x+y+1 \le \frac{[\sqrt{8n+1}]+1}{2} < x+y+2 .$$

Следовательно: $$x+y+1 = \left[\frac{[\sqrt{8n+1}]+1}{2}\right] . \qquad (4)$$

Решаем систему уравнений (0), (4). Из (4) находим:

$$x+y = \left[\frac{[\sqrt{8n+1}]+1}{2}\right]-1 = \left[\frac{[\sqrt{8n+1}]+1}{2}-1\right] =$$

$$= \left[\frac{[\sqrt{8n+1}]-1}{2}\right] = \left[\frac{[\sqrt{8n+1}]\mathbin{\dot{-}}1}{2}\right]. \qquad (5)$$

На последнем переходе знак $-$ заменён на знак $\dot{-}$, ввиду того, что $[\sqrt{8n+1}] \geq 1$.

Выразим x из уже доказанного выражения для функции $c(x,y)$. Это и будет выражение для $l(n)$. Подставим в него значения $x+y+1$ из (4) и $x+y$ из (5), а $c(x,y)$ заменим на n. Получим:

$$l(n) = x = c(x,y)-\frac{(x+y)(x+y+1)}{2} =$$

$$= n - \frac{1}{2}\left[\frac{[\sqrt{8n+1}]+1}{2}\right]\cdot\left[\frac{[\sqrt{8n+1}]\mathbin{\dot{-}}1}{2}\right] =$$

$$= n \mathbin{\dot{-}} \frac{1}{2}\left[\frac{[\sqrt{8n+1}]+1}{2}\right]\cdot\left[\frac{[\sqrt{8n+1}]\mathbin{\dot{-}}1}{2}\right]. \qquad (6)$$

На последнем переходе знак $-$ заменён на знак $\dot{-}$, ввиду того, что $n \geq \frac{(x+y)(x+y+1)}{2}$.

Из равенства (5) находим y и подставляем в полученное выражение только что найденное в (6) значение x. Получаем выражение для $r(n)$:

$$r(n) = y = \left[\frac{[\sqrt{8n+1}]\mathbin{\dot{-}}1}{2}\right] - x =$$

$$= \left[\frac{[\sqrt{8n+1}]\mathbin{\dot{-}}1}{2}\right] + \frac{1}{2}\left[\frac{[\sqrt{8n+1}]+1}{2}\right]\cdot\left[\frac{[\sqrt{8n+1}]\mathbin{\dot{-}}1}{2}\right] - n =$$

$$= \left[\frac{[\sqrt{8n+1}]\mathbin{\dot{-}}1}{2}\right]\cdot\left(1+\frac{1}{2}\left[\frac{[\sqrt{8n+1}]+1}{2}\right]\right) \mathbin{\dot{-}} n.$$

Теорема полностью доказана. □

В дальнейшем нам будет нужен не конкретный вид формул для функций $c(x,y)$, $l(n)$, $r(n)$, а лишь тот факт, что эти функции выражаются с помощью подстановок (суперпозиций) через примитивно рекурсивные функции $+$, $\dot{-}$, $\cdot$, $\sqrt{\ }$, [ ], и, следовательно,

сами являются примитивно рекурсивными, или обобщённо говоря, – (алгоритмически) вычислимыми.

Значение канторовской нумерации заключается *в её эффективности*, понимаемой в данном случае как возможность восстановления самой пары по её номеру.

Существование перечислимого, но не разрешимого множества. Вернёмся теперь к вопросу о существовании перечислимого, но не разрешимого множества, возникшему в конце предыдущего пункта.

Теорема 6.3.4. *Существует перечислимое, но не разрешимое множество натуральных чисел.*

Доказательство. На основании теоремы 5.3.2 Поста достаточно привести пример такого множества U натуральных чисел, которое само было бы перечислимо, а его дополнение $\overline{U}$ перечислимым не было.

Ясно, что перечислимых множеств натуральных чисел, как алгоритмов, имеется лишь счётное количество. Следовательно, *все* перечислимые множества можно расположить в последовательность (перенумеровать):

$$M_0\ ,\ M_1\ ,\ M_2\ ,\ M_3\ ,\ \ldots \qquad (*)$$

Можем считать, что нумерация $(*)$ эффективна, т.е. по любому заданному числу r можно восстановить множество M_r, имеющее r своим номером в этой нумерации. [В начале следующей главы VI мы покажем, как можно эффективно перенумеровать все машины Тьюринга: их можно, затем рассматривать как алгоритмы, вычисляющие функции, порождающие перечислимые множества].

Рассмотрим теперь алгоритм $\mathbf{A}$, который последовательно порождает все элементы следующего множества U. На шаге с номером $c(m,n)$ этот алгоритм вычисляет m-ый элемент множества M_n, и если элемент совпадает с n, то он относит его в множестве U. Таким образом, $$n \in U \iff n \in M_n\ .$$

Итак, U порождается алгоритмом $\mathbf{A}$, т.е. U перечислимо. Поскольку дополнение $\overline{U}$ множества U состоит из всех таких n, что $n \notin M$, то $\overline{U}$ отличается от любого перечислимого множества хотя бы одним элементом. Поэтому $\overline{U}$ не является перечислимым. Значит, на основании теоремы 5.3.2, множество U не разрешимо, что и завершает доказательство. □

Отметим, что доказанная теорема фактически включает в себя в неявном виде знаменитую теорему Гёделя о неполноте формальной арифметики, доказательство которой будет приведено ниже в параграфе 9.1. Эту теорему без преувеличения можно считать самым выдающимся достижением математической науки XX века. Её доказательство основано на методах теории алгоритмов. Так что не только математическая логика оказала влияние на становление и развитие теории алгоритмов, но и теория алгоритмов внесла в математическую логику поистине конструктивные и эффективные идеи, позволившие решить многие стоявшие в ней проблемы.

Разрешимость и перечислимость: итог. Итак, эффективно заданное множество – это множество, обладающее разрешающей или перечисляющей функцией (алгоритмом). Объединение и пересечение перечислимых множеств перечислимы. Непустое множество $M \subseteq N$ разрешимо тогда и только тогда, когда оно само и его дополнение перечислимы. В частности, отсюда следует, что всякое непустое разрешимое множество перечислимо. Тем не менее, существует перечислимое, но неразрешимое множество натуральных чисел. В теореме 5.3.4 такое множество описано. Другим примером такого множества является множество $M = \{x : \mathbf{A}_x(x) \textit{ определён} \}$, т.е. множество номеров самоприменимых алгоритмов.

Таким образом, два основных способа задания множеств при алгоритмическом подходе оказываются неэквивалентными: перечислимость (т.е. эффективное перечисление – более слабый вид эффективности, нежели разрешимость (т.е. эффективное указание характеристического свойства элементов множества). Хотя перечисляющая процедура и задаёт эффективно список элементов множества M, поиск данного элемента a в этом списке может оказаться неэффективным: это неопределённо долгий процесс, который в конечном счёте остановится, если $a \in M$, но не остановится, если $a \notin M$. В случае же перечислимости и дополнения $\overline{M}$ множества M перечисляющая процедура для M становится эффективной и гарантирует разрешимость множества M.

6.4. Связь разрешимых и перечислимых множеств с вычислимыми функциями

Понятия разрешимости и перечислимости множеств, рассмотренные выше, по определению связаны с вычислимыми функциями.

В этом параграфе мы установим ряд теорем, показывающих, что связь между свойствами алгоритмической вычислимости функций и множеств простирается достаточно далеко.

Перечислимые множества и вычислимые функции. Здесь доказываются две теоремы, дающие новые характеристики перечислимых множеств.

ТЕОРЕМА 6.4.1. (Основная теорема о перечислимых множествах). *Множество $M \subset N$ перечислимо тогда и только тогда, когда M является областью определения некоторой (частичной) вычислимой функции.*

$$\boxed{M \text{ – перечислимо} \quad \Longleftrightarrow \quad (\exists x)\,(M = Dom\,\varphi_x)}\ .$$

Доказательство. Необходимость. Пусть M – перечислимо. Если $M = \emptyset$, то M можно считать областью определения нигде не определённой вычислимой функции. Если же $M \neq \emptyset$, то по определению, M есть множество значений некоторой всюду определённой вычислимой функции f. Зададим теперь функцию ψ следующим образом. Чтобы найти значение $\psi(x)$, нужно последовательно порождать значения функции f: $f(0)$, $f(1)$, $f(2)$, ... Как только на некотором конечном шаге среди этих значений появится x, т.е. $f(s) = x$, процесс порождения остановить и считать $\psi(x) = x$; если же среди этих значений x не появляется, и процесс порождения продолжается бесконечно, то считаем, что для этого x функция ψ не определена. Таким образом, $\psi(x) = x$ для $x \in M$, и $\psi(x)$ не определено для $x \notin M$. Причём, функция ψ вычислима (так как вычислима функция f) и её область определения $M = Dom\,\psi$).

Достаточность. Пусть $M = Dom\,\psi$, где ψ – частичная вычислимая функция. Построим всюду определённую функцию f, которая будет порождать M, если $M \neq \emptyset$. (Если $M = \emptyset$, то M перечислимо по определению). Функция f должна быть такой всюду определённой функцией, чтобы её значениями были все те и только те натуральные числа, на которых функция ψ определена. Для задания значений этой функции на каждом натуральном числе сначала проделаем следующую поэтапную процедуру, которая будет напоминать диагональный метод Кантора, применённый им для эффективной нумерации упорядоченных пар натуральных чисел и рассмотренный нами в параграфе 6.3.

Для каждого $x \in M$ значение $\psi(x)$ определено и в силу вычислимости функции ψ оно вычисляется соответствующим алгоритмом за вполне определённое конечное число шагов. Если $x \notin M$,

то значение $\psi(x)$ не определено. Составим бесконечную прямоугольную таблицу: по вертикали: $\psi(0)$, $\psi(1)$, $\psi(2)$, $\psi(3)$, ... – значения функции; по горизонтали: 1, 2, 3, 4, ... – число шагов, за которое вычисляется соответствующее значение. В каждой строке таблицы поставим звёздочку $*$ в том столбце, номер которого равен числу шагов, необходимых для вычисления соответствующего значения. Если соответствующее значение не вычислимо (функция ψ не определена), то в этой строке звёздочки нет.

	1	2	3	4	5	6	7	8	9	...
$\psi(0)$		$*$								
$\psi(1)$					$*$					
$\psi(2)$						$*$				
$\psi(6)$	$*$									
$\psi(4)$										→ не опр.
$\psi(5)$			$*$							
$\psi(6)$							$*$			
$\psi(7)$									$*$	
...										

Теперь проделываем следующую поэтапную процедуру, начинающуюся с левого верхнего угла таблицы. Просматриваем последовательно расширяющиеся из этого угла квадраты таблицы размером: 1×1 (этап 1), 2×2 (этап 2), 3×3 (этап 3), и т.д. На каждом этапе выписываем те значения k, для которых звёздочка $*$ в строке $\psi(k)$ попала в соответствующий квадрат. В нашем примере получаем следующие записи:

Этап 1:
Этап 2: 0
Этап 3: 0
Этап 4: 0, 3
Этап 5: 0, 3, 1 (квадрат выделен в таблице)
Этап 6: 0, 3, 1, 2, 5
Этап 7: 0, 3, 1, 2, 5, 6
Этап 8: 0, 3, 1, 2, 5, 6
. . . .

Все выписанные числа и только они должны стать значениями функции f. Зададим функцию f следующим образом: $f(0) =$ первое выписанное число. (В нашем примере: $f(0) = 0$). Значение $f(n+1)$

определим как наименьшее из тех чисел, находящихся в списке на этапе $n+1$, и которые отсутствуют среди предыдущих значений $f(0)$, $f(1)$, $f(2)$, ... , $f(n)$. Такого может не оказаться: на этапе $n+1$ не добавилось ни одного числа, и все числа из списка уже содержатся среди предыдущих значений. В этом случае полагаем: $f(n+1) = f(0)$. В нашем примере получаем: $f(1) = f(0) = 0$, $f(2) = f(0) = 0$, $f(3) = f(0) = 0$, $f(4) = 3$, $f(5) = 1$, $f(6) = 2$, $f(7) = 5$, $f(8) = 6$, ...

Итак, f определяем следующим образом:

$f(0)$ = первое число, попавшее в список;

$$f(n+1) = \begin{cases} \mu y\,[\,y \text{ находится в списке на этапе } n+1 \text{ и} \\ y \notin \{f(0), f(1), ..., f(n)\}], \text{ если такой } y \text{ существует;} \\ f(0), \quad \text{в противном случае.} \end{cases}$$

Тогда ясно, что каждое из чисел k, для которого значение $\psi(k)$ определено (в строке $\psi(k)$ имеется звёздочка $*$ на конечном месте), окажется среди значений функции f, так как за конечное число шагов будет достигнуто нашим диагональным процессом. Все числа k, для которых $\psi(k)$ не определено, в множество значений f не попадут. Кроме того, f определена для каждого натурального числа.

Таким образом, f – перечисляющая функция для множества $Dom\ \psi(k) = M$. Следовательно, M перечислимо.

Теорема полностью доказана. □

Замечание. Это доказательство взято нами из книги [98], §5.2, теорема V, стр. 84 – 85. Но там значение $f(n+1)$ функции f предлагается определить как наименьшее из тех чисел, которые добавлены к списку на этапе $n+1$, и отсутствуют среди предыдущих значений $f(0), f(1), ..., f(n)$. При таком определении f в нашем примере число 5 не будет значением функции f ни для какого k: этап 6 даёт $f(6) = 2$; этап 7 даёт $f(7) = 6$; этап 8 даёт $f(8) = f(0) = 0$.

Итак, по доказанной теореме, каждое перечислимое множество ассоциируется с некоторой вычислимой функцией φ_x, имеющей это множество в качестве своей области определения. Эта функция в нашей нумерации всех вычислимых функций имеет (гёделевский) номер x. Напомним, что символом W_x мы обозначили область определения функции φ_x: $W_x = Dom\varphi_x$. Число x будем называть также (гёделевским) номером перечислимого множества W_x. Множество W_x может быть областью определения бесконечного

множества вычислимых функций; поэтому множество его гёделевских номеров бесконечно (как, впрочем, и множество гёделевских номеров вычислимой функции).

Следующая теорема, дополняя предыдущую, по существу является её следствием.

Теорема 6.4.2. *Множество $M \subset N$ перечислимо тогда и только тогда, когда M является множеством значений некоторой (частичной) вычислимой функции.*

$$\boxed{M \text{ – перечислимо} \iff (\exists x)\,(M = Im\,\varphi_x)}\,.$$

(Заметим, что от *определения* перечислимого множества эта формулировка отличается тем, что здесь вычислимая функция может быть частичной, а в определении она всюду определена).

Доказательство. Необходимость. Если $M = \emptyset$, то для нигде не определённой (частичной) вычислимой функции ψ имеем: $M = Im\,\psi$. Если же $M \neq \emptyset$, то функция ψ, построенная при доказательстве предыдущей теоремы, пригодна и здесь, так как $M = Dom\,\psi = Im\,\psi$.

Достаточность доказывается, как и в предыдущей теореме, с тем лишь отличием, что теперь к ранее образованному списку для M добавляются по этапам не *входы*, а *выходы*. □

Из этих двух теорем следует, что множество является областью определения частичной вычислимой функции тогда и только тогда, когда оно является множеством значений (вообще говоря, другой) частичной вычислимой функции.

Отметим, что теоремы 6.4.1 и 6.4.2 являются прямыми следствиями двух важнейших свойств алгоритма (вычисляющего функцию ψ) – пошагового выполнения алгоритма и конечности числа таких шагов.

Ещё один пример перечислимого, но не разрешимого множества. Первый пример был построен в параграфе 6.3 (теорема 6.3.4). Теорема 6.4.1 даёт действенный метод, позволяющий устанавливать перечислимость тех или иных множеств. Приведём здесь один такой пример. Напомним, что все (частичные) вычислимые функции мы занумеровали эффективным образом: φ_0, φ_1, φ_1, ... φ_n, ... , т.е. расположили в последовательность так, что по номеру n функции может быть однозначно восстановлена сама функция (при этом, в этой последовательности каждая функция может

встретиться не однажды, т.е. каждой вычислимой функции соответствует, вообще говоря, бесконечное множество номеров).

Рассмотрим множество K номеров x всех таких вычислимых функций, которые определены на своём номере: $K = \{x : \varphi_x(x)$ – определено $\}$. (Другими словами, номер функции принадлежит её области определения: $x \in W_x$). Итак,

$$K = \{x : \varphi_x(x) - \text{определено}\} = \{x : x \in W_x\} .$$

Теорема 6.4.3. *Множество K перечислимо, но не разрешимо.*

Доказательство. Чтобы применить теорему 6.4.1, покажем, что K является областью определения некоторой вычислимой функции. Зададим функцию ψ следующим образом:

$$\psi(x) = \begin{cases} 1 , & \text{если } \varphi_x(x) - \text{ определено}, \\ \text{не определено} , & \text{если } \varphi_x(x) - \text{ не определено}. \end{cases}$$

Ясно, что функция ψ вычислима, так как вычислимы все функции φ_x. Причём, в её область определения $Dom\ \psi$ входят те и только те x, для которых значение $\varphi_x(x)$ определено, т.е. $Dom\ \psi = K$. Следовательно, по теореме 6.4.1, множество K перечислимо.

Покажем теперь, что K не разрешимо. Допустим противное: K разрешимо. Тогда, ввиду доказанной перечислимости K и теоремы 6.3.2 Поста, отсюда следует, что дополнение $\overline{K}$ перечислимо. Значит, по теореме 6.4.1, $\overline{K}$ является областью определения некоторой вычислимой функции: $\overline{K} = W_m$ для некоторого m. Тогда

$$m \in \overline{K} \iff m \in W_m .$$

Но с другой стороны, по определению множества K, имеем:

$$m \in K \iff m \in W_m.$$

Полученное противоречие показывает, что множество K не может быть разрешимым. □

Обратите внимание на то, что при доказательстве второй части теоремы снова применён метод диагонализации.

Примеры множеств, не являющихся перечислимыми. В этом пункте мы применим теорему 6.4.1 и метод диагонализации для доказательства перечислимости множества всюду определённых вычислимых функций. Это будет означать, что все всюду определённые вычислимые функции не могут быть эффективно перенумерованы.

Теорема 6.4.4. *Множество* $L = \{x : \varphi_x$ *всюду определена*$\}$ ***не является перечислимым множеством.***

Доказательство. Допустим противное: L перечислимо. По определению это означает, что существует всюду определённая вычислимая функция f, перечисляющая все элементы этого множества, т.е., иначе говоря, что последовательность

$$\varphi_{f(0)},\ \varphi_{f(1)},\ \varphi_{f(2)},\ \cdots\ ,\ \varphi_{f(n)},\ \cdots$$

есть список *всех* одноместных всюду определённых вычислимых функций. С помощью диагонального метода построим тогда всюду определённую вычислимую функцию g, которая отличается от каждой функции этого списка и, значит, не содержится в этом списке. Это будет противоречить тому, что список исчерпывает *все* такие функции. Основной принцип диагонального метода гласит: добивайся того, чтобы функция g отличалась от функции $\varphi_{f(n)}$ в точке n. Поэтому положим: $g(x) = \varphi_{f(x)}(x) + 1$ для любого x. Эта функция будет всюду определена, так как всюду определены функции φ и f. Но g будет отличаться от каждой функции $\varphi_{f(m)}$ нашего списка, так как в точке m будем иметь: $g(m) = \varphi_{f(m)}(m) + 1 \neq \varphi_{f(m)}(m)$. Полученное противоречие и доказывает теорему. □

Из этой теоремы ввиду теоремы 6.3.1, сразу получаем, что множество L также не является и разрешимым.

Отметим, что можно доказать, что перечислимым не является также и дополнение множества L, т.е. множество $\overline{L} = \{x : \varphi_x$ не всюду определена$\}$. (См. [60]), стр. 139, следствие 2.17). Значит, и множество $\overline{L}$ не разрешимо.

Ещё одна характеристика перечислимых множеств. Эта характеристика ещё больше оттенит различия между разрешимыми и перечислимыми множествами и связана она с вычислимостью некоторой функции. Напомним, что разрешимое множество M можно охарактеризовать как такое, для которого его характеристическая функция

$$\chi_M(x) = \begin{cases} 1, \text{ если } x \in M, \\ 0, \text{ если } x \notin M \end{cases}$$

вычислима. Эта функция всюду определена. Рассмотрим для множества M следующую частичную функцию $h_M(x)$, которую назовём ***квазихарактеристической функцией*** множества M:

$$h_M(x) = \begin{cases} 1, & \text{если } x \in M, \\ \text{не определено}, & \text{если } x \notin M \end{cases}$$

Оказывается, вычислимость этой функции служит признаком перечислимости множества M. Из сопоставления этих функций ясно, что требование вычислимости функции χ_M логически сильнее требования вычислимости функции h_M. Поэтому всякое разрешимое множество перечислимо, но не наоборот.

ТЕОРЕМА 6.4.5. ***Множество $M \subset N$ перечислимо тогда и только тогда, когда его квазихарактеристическая функция h_M вычислима.***

Доказательство. Необходимость. Пусть M перечислимо. Если $M = \emptyset$, то функция h_M нигде не определена и потому вычислима. Если же $M \neq \emptyset$, то по определению, M есть множество значений некоторой всюду определённой вычислимой функции f. Тогда функция h_M будет вычислима. В самом деле, чтобы найти значение $h_M(x)$, нужно последовательно порождать значения f: $f(0)$, $f(1)$, $f(2)$, ... Как только на некотором конечном шаге s среди этих значений появится x, т.е. $f(s) = x$, процесс порождения остановить. Это означает, что $x \in M$ и значит, $h_M(x) = 1$. Если же среди этих значений число x не появляется, и процесс порождения продолжается бесконечно, то это означает, что $x \notin M$, и функция h_M для этого x не определена.

Достаточность. Пусть функция h_M вычислима. Из её определения видно, что её областью определения является множество M. Тогда, в силу теоремы 6.4.1, множество M перечислимо.

Теорема доказана. □

Таким образом, можно сказать, что перечислимое множество – это такое, для которого существует алгоритм, применимый к элементам этого множества (и дающий какой-то положительный ответ, например, "да") и не применимый к элементам, не принадлежащим этому множеству (не даёт для них никакого ответа, работает вечно). И ещё: понятие перечислимого множества есть алгоритмический (динамический) аналог (статического) понятия счётного множества.

Подводя некоторый итог, отметим следующее. В силу теоремы 6.3.1, возможны лишь следующие три типа множеств: (1) перечислимые и разрешимые; (2) перечислимые, но не разрешимые; (3) неперечислимые и неразрешимые. В силу теоремы 6.3.2 Поста, дополнение $\overline{M}$ к множеству M типа (2) есть множество типа (3). В самом деле, оно не может быть перечислимым, ибо тогда по теореме 6.3.2 исходное множество M было бы разрешимым, но это не так. В то

же время $\overline{M}$ не может быть и разрешимым, ибо тогда по теореме 6.3.1 оно было бы перечислимым, что, по доказанному, не так.

Следовательно, множество $\overline{K}$ всех номеров (частичных) вычислимых функций, номера которых не входят в область их определения, неперечислимо и неразрешимо.

Перечислимые множества как проекции разрешимых бинарных отношений. Если теорема Поста (теорема 6.3.2) характеризует разрешимые множества через перечислимые, то приводимая здесь теорема даёт обратную характеристику – перечислимых множеств через разрешимые. Доказывается она с использованием теоремы 6.4.5.

Напомним, что бинарным отношением на множестве натуральных чисел N называется всякое подмножество декартова произведения $N \times N$, т.е. всякое множество упорядоченных пар натуральных чисел: $\rho \subseteq N \times N$. Первой проекцией этого отношения (или просто проекцией) называется множество

$$\text{пр}\, \rho = \{x : (\exists y \in N)\ [(x, y) \in \rho]\} .$$

Теорема 6.4.6. *Множество $M \subset N$ перечислимо тогда и только тогда, когда оно является проекцией некоторого разрешимого бинарного отношения на N.*

Доказательство. Необходимость. Пусть M перечислимо. Тогда по определению, оно генерируется некоторой вычислимой функцией f:

$$M = \{f(0),\ f(1),\ f(2),\ f(3),\ \dots\} .$$

Рассмотрим следующее бинарное отношение на N: $\rho = \{(f(n), n) : n \in N\}$, где на первом месте упорядоченной пары $(f(n), n)$ стоит элемент $x \in M$, а на втором – его номер n при генерации f. Опишем алгоритм, обеспечивающий разрешимость множества ρ. Для произвольной пары $(x, n) \in N \times N$ сделаем n шагов генерации множества M, т.е. рассмотрим элемент $f(n)$. Если на последнем шаге получается элемент x, т.е. $x = f(n)$, то $(x, n) \in \rho$; в противном случае $(x, n) \notin \rho$.

Достаточность. Пусть M есть проекция разрешимого бинарного отношения $\rho \subseteq N \times N$, т.е. $M = \text{пр}\, \rho$. Тогда множество M есть область применения следующего алгоритма (и значит, есть область определения вычисляемой этим алгоритмом функции). Рассмотрим

следующую частичную функцию:

$$f(x) = \begin{cases} 1, & \text{если высказывание } (\exists y)\ [(x, y) \in \rho] \text{ ист.,} \\ \text{не определено}, & \text{если это высказывание ложно.} \end{cases}$$

Докажем её вычислимость. Алгоритм, вычисляющий значение $f(x)$, следующий. Перебираем последовательно все натуральные числа n и каждый раз отвечаем на вопрос "$(x, n) \in \rho$?" Ответ на него находим с помощью разрешающего алгоритма для отношения ρ. Если на некотором конечном шаге получим положительный ответ, то $f(x) = 1$. (Заметим, что это будет означать, что $x \in$ пр ρ). Если положительный ответ никогда не будет получен, и алгоритм будет работать вечно, то f для такого x не определена. (Заметим, что отсутствие такого n, что $(x, n) \in \rho$, означает, что $x \notin$ пр ρ).

Итак, функция f вычислима. Замечания, сделанные в скобках, говорят о том, что f есть квазихарактеристическая функция множества пр $\rho = M$, т.е. имеет вид

$$f(x) = \begin{cases} 1, & \text{если} \quad x \in \text{пр } \rho, \\ \text{не определено}, & \text{если} \quad x \notin \text{пр } \rho, \end{cases}$$

т.е. $f(x) = h_M(x)$. Следовательно, по теореме 6.4.5, множество M перечислимо. □

График вычислимой функции. Закончим этот параграф одним из важнейших фактов теории алгоритмов, устанавливающим ещё одну связь между понятиями вычислимой функции и перечислимого множества.

Графиком функции $f : N \times N$ с непустой областью определения $Dom\ (f) \neq \emptyset$ называется следующее множество упорядоченных пар натуральных чисел: $G_f \rightleftharpoons \{(x, f(x)) : x \in Dom\ (f)\}$.

Теорема 6.4.7. (О графике вычислимой функции). ***Функция*** $f : N \times N$ ***вычислима тогда и только тогда, когда её график*** G_f ***является перечислимым множеством.***

f – вычислима $\iff$ G_f – перечислимо

.

Доказательство. Необходимость. Пусть f вычислима. Тогда по теореме 6.4.1, её область определения $Dom\ (f)$ есть перечислимое множество, порождаемое некоторой перечисляющей функцией ψ:

$$Dom\ (f) = \{\psi(0),\ \psi(1),\ \psi(2),\ \dots,\ \psi(n),\ \dots\}\ .$$

Добавим к алгоритму, вычисляющему ψ, дополнительные инструкции. Вычислив значение $\psi(n)$ (это будет некоторый элемент $x \in Dom\ (f)$), применить к нему алгоритм, вычисляющий функцию f: получим значение $f(\psi(n)) = f(x)$. Таким образом, композиция этих двух алгоритмов порождает множество упорядоченных пар $(x, f(x))$, образующих график G_f функции f. (Число n будет номером этой упорядоченной пары при этом порождении). По определению, это и означает, что множество G_f перечислимо.

Достаточность. Пусть теперь график G_f функции f является перечислимым множеством. Это означает, что имеется алгоритм, последовательно порождающий все упорядоченные пары $(x, f(x)) \in G_f$. Построим на его основе алгоритм, вычисляющий функцию f. Для каждого n значение $f(n)$ он вычисляет следующим образом. Предыдущим алгоритмом нужно генерировать пары (x, y) до тех пор, пока станет $x = n$. На этом шаге генерирование прекратить и выдать результат: $f(n) = y$. Таким образом, функция f вычислима.

Теорема доказана. □

Глава VII

АЛГОРИТМИЧЕСКИ НЕРАЗРЕШИМЫЕ МАССОВЫЕ ПРОБЛЕМЫ

Алгоритмическая проблема – это проблема, в которой требуется найти единый метод (алгоритм) для решения бесконечной серии однотипных единичных задач. Такие проблемы называют также *массовыми проблемами.* Они возникали и решались в различных областях математики на протяжении всей её истории. Примеры таких проблем рассматривались в главе I.

Те или иные алгоритмические задачи с помощью кодирования сводятся к словарным функциям; словарные функции с помощью нумераций сводятся к функциям вида $f: N^n \to N$ (т.е. заданным и принимающим значения в множестве натуральных чисел). Таким образом, алгоритмическая проблема – это задача о нахождении алгоритмов для вычисления значений числовых функций. Если алгоритма для вычисления той или иной функции не существует, то говорят, что соответствующая алгоритмическая проблема неразрешима.

Мы уже отмечали, что в начале XX века у математиков начали появляться подозрения в том, что некоторые массовые проблемы не имеют общего алгоритма для их решения. В связи с этим возникла необходимость дать точное определение самому понятию алгоритма. Мы познакомились с несколькими способами такого уточнения, и в настоящей главе приведём примеры алгоритмически неразрешимых массовых проблем. Алгоритмически неразрешимые массовые проблемы интересны и важны потому, что, среди прочего, они указывают на ограничения, возникающие в теории вычислимости, и тем самым демонстрируют теоретические границы возможностей реальных вычислительных машин.

Важным средством для исследования алгоритмов, в частности, для доказательств несуществования единого алгоритма для решения той или иной массовой проблемы, является понятие нумерации

алгоритмов, введённое нами в параграфе 5.1. Существование нумераций позволяет работать с алгоритмами как с числами. Это особенно удобно при исследовании алгоритмов над алгоритмами. Отсутствие именно таких алгоритмов часто приводит к алгоритмически неразрешимым проблемам.

Сначала в качестве понятия, уточняющего понятие алгоритма, будем использовать понятие машины Тьюринга (§7.1). Затем рассмотрим проблему алгоритмической разрешимости в рамках общей теории алгоритмов (§7.2). Параграф 7.3 посвятим так называемым частично разрешимым массовым проблемам и связанным с ними частично разрешимым предикатам, которые, в свою очередь, оказались тесно связанными со знаменитой 10-ой проблемой Гильберта и способствовали её решению почти через три четверти века после того, как она была поставлена.

7.1. Алгоритмически неразрешимые массовые проблемы, связанные с машинами Тьюринга

Напомним, что в параграфе 5.1 мы осуществили эффективную нумерацию всех мыслимых машин Тьюринга, т.е. такую нумерацию, что по номеру машины мы можем восстановить саму машину. Эта нумерация будет существенно использована при построении примера функции, невычислимой по Тьюрингу.

Существование не вычислимых по Тьюрингу функций. Сначала докажем чистую теорему существования.

Теорема 7.1.1. *Существует функция, не вычислимая по Тьюрингу, т.е. невычислимая ни на одной машине Тьюринга.*

Доказательство. Функции, о которых идёт речь, представляют собой функции, заданные и принимающие значения в множестве слов в алфавите $A_1 = \{1\}$. Ясно, что множество слов в алфавите $A_1 = \{1\}$ счётно, т.е. рассматривается множество всех функций, заданных на счётном множестве и принимающие значения в счётном же множестве. Как известно, это множество имеет мощность континуума. С другой стороны, поскольку множество всевозможных машин Тьюринга, как мы установили в параграфе 5.1, перенумеровав их, счётно, поэтому и множество функций, вычислимых по Тьюрингу, также счётно. Континуальная мощность строго больше

счётной. Следовательно, существуют функции, не вычислимые по Тьюрингу. □

Принимая во внимание тезис Тьюринга, получаем:

Следствие 7.1.2. (С учётом тезиса Тьюринга.) *Существуют невычислимые функции.*

Доказанные теорема и следствие из неё являются так называемыми чистыми теоремами существования. Интересно получить пример конкретной функции, не вычислимой по Тьюрингу (и, следовательно, не вычислимой).

Пример не вычислимой по Тьюрингу функции. Укажем конкретную функцию, которую нельзя вычислить ни на какой машине Тьюринга. На основании тезиса Тьюринга, это будет означать, что не существует вообще никакого алгоритма для вычисления значений такой функции.

Рассмотрим следующую функцию $\psi(\alpha)$, заданную на множестве слов в алфавите $A_1 = \{1\}$, определив её следующим образом. Для произвольного слова α длины n в алфавите $A_1 = \{1\}$ положим:

$$\psi(\alpha) = \begin{cases} \beta_n 1, & \text{если слово } \alpha \text{ перерабатывается машиной} \\ & \text{Тьюринга с номером } n \text{ в слово } \beta_n \\ & \text{алфавита } A_1 = \{1\}\ ; \\ 1\ , & \text{в противном случае}\ . \end{cases}$$

Докажем, что функция $\psi(\alpha)$ не вычислима по Тьюрингу.

Теорема 7.1.3. *Функция $\psi(\alpha)$ не вычислима по Тьюрингу.*

Доказательство. Допустим противное. Это означает, что существует машина Тьюринга T со стандартным алфавитом $\{a_0, 1, q,$ П, Л $\}$, вычисляющая эту функцию. Пусть k – номер этой машины в нумерации, описанной в предыдущем пункте. Посмотрим, чему равно слово $\psi(1^k)$, (напомним, что $1^k = 11...1$ – слово из k единиц), являющееся значением функции $\psi(\alpha)$ при $\alpha = 1^k$. Предположим, что машина T перерабатывает слово 1^k в слово β_k в том же алфавите $A_1 = \{1\}$. Тогда по определению вычислимости функции $\psi(\alpha)$ на машине T это означает, что $\psi(1^k) = \beta_k$. Но с другой стороны, по самому определению функции $\psi(\alpha)$ это означает, что $\psi(1^k) = \beta_k 1$. Полученное противоречие доказывает, что машины Тьюринга, вычисляющей функцию $\psi(\alpha)$, не существует. □

Принимая во внимание тезис Тьюринга, получаем:

Следствие 7.1.4. (С учётом тезиса Тьюринга.) ***Функция $\psi(\alpha)$ алгоритмически не вычислима, т.е. не существует вообще никакого алгоритма для вычисления значений функции $\psi(\alpha)$.***

Это означает, что массовая проблема нахождения значений функции $\psi(\alpha)$ для всевозможных значений аргумента алгоритмически неразрешима.

Проблема распознавания самоприменимости. Рассмотрим ещё два примера неразрешимых алгоритмических проблем – распознавания самоприменимости и распознавания применимости машин Тьюринга. Сначала о первой. Предположим, что на ленте машины Тьюринга записана её собственная функциональная схема в алфавите машины. Если машина применима к такой конфигурации, то будем называть её самоприменяемой, в противном случае - несамоприменяемой. Возникает массовая проблема распознавания самоприменяемых машин Тьюринга, состоящая в следующем. По заданной функциональной схеме (программе) машины Тьюринга установить, к какому классу относится машина: к классу самоприменимых машин или к классу несамоприменимых машин.

Теорема 7.1.5. *Проблема распознавания самоприменимых машин Тьюринга алгоритмически неразрешима.*

Доказательство. Допустим противное, т.е. алгоритм для такого распознавания существует. Значит, на основании тезиса Тьюринга, существует машина Тьюринга, реализующая данный алгоритм. Пусть Θ – такая машина. На её ленту заносится соответствующим образом закодированная программа той или иной машины Тьюринга. При этом, если машина самоприменима, то занесённое слово перерабатывается машиной Θ в какой-то символ σ (имеющий смысл утвердительного ответа на поставленный вопрос о самоприменимости). Если же машина не самоприменима, то занесенное на ленту слово, кодирующее её программу, перерабатывается машиной Θ в какой-то символ τ (имеющий смысл отрицательного ответа на поставленный вопрос).

Рассмотрим теперь такую машину Тьюринга Θ_1, которая попрежнему перерабатывает несамоприменимые коды в τ, а к самоприменимым кодам машина Θ_1 уже не применима. Такая машина получается из машины Θ, если следующим образом слегка изменить ее программу: после появления символа σ вместо остановки машина должна неограниченно его повторять.

Итак, Θ_1 применима ко всякому несамоприменимому коду (вырабатывая символ τ) и не применима к самоприменимым кодам. Это

приводит к противоречию, потому что такая машина не может быть ни самоприменимой, ни несамоприменимой. В самом деле, если машина Θ_1 самоприменима, то она неприменима к своему коду. Значит, машина несамоприменима. Противоречие. С другой стороны, если машина Θ_1 несамоприменима, то её код должен перерабатываться самой машиной Θ_1 в символ τ. Значит, Θ_1 применима к собственному коду, т.е. самоприменима. Снова противоречие. Оно и доказывает теорему. □

Проблема распознавания применимости. На основании доказанной теоремы устанавливается алгоритмическая неразрешимость и некоторых других массовых проблем, возникающих в теории машин Тьюринга, в частности, *проблемы распознавания применимости для машин Тьюринга*, которая состоит в следующем. Заданы функциональная схема (программа) какой-нибудь машины Тьюринга и конфигурация в ней: узнать, применима ли машина к данной конфигурации или нет. Эта проблема носит также название *проблемы остановки.*

ТЕОРЕМА 7.1.6. *Проблема распознавания применимых машин Тьюринга алгоритмически неразрешима.*

Доказательство. Если бы существовал алгоритм для решения этой проблемы, то с его помощью можно было бы узнать, применима ли машина к слову, кодирующему её собственную программу, т.е. самоприменима ли она. Но на основании предыдущей теоремы, известно, что такого алгоритма не существует. □

7.2. Алгоритмически неразрешимые массовые проблемы в общей теории алгоритмов

В главах V и VI мы уже достаточно далеко продвинулись в развитии общей теории вычислимости, так что здесь сможем применить разработанные там методы к установлению алгоритмической неразрешимости ряда массовых проблем в общей теории алгоритмов. Первый шаг в этом направлении нами уже сделан: теорема 5.1.2 утверждает существование всюду определённой невычислимой функции. Отметим, что многие доказательства неразрешимости основываются на рассмотренной в параграфе 5.1 нумерации алгоритмов и на диагональном методе, который мы также обсуждали в параграфе 5.1.

Проблема распознавания самоприменимости алгоритмов. Она формулируется так: существует ли алгоритм, который для любого алгоритма определял бы, применим ли он к своему собственному описанию.

Напомним, что все алгоритмы у нас эффективно занумерованы, и номер x алгоритма в этой нумерации как раз и служит описанием алгоритма, который он нумерует. Этот же номер x служит одновременно и номером функции φ_x, которую этот алгоритм вычисляет. Применимость алгоритма на данных означает, что вычисляемая алгоритмом функция определена на натуральном числе, кодирующем эти данные. Так что исходная проблема может быть сформулирована следующим образом: существует ли алгоритм (эффективная процедура) для установления того, определена ли функция φ_x на числе x, являющемся её собственным номером.

Наконец, алгоритм, которым мы интересуемся, также вычисляет некоторую функцию, которую можно определить так:

$$f(x) = \begin{cases} 1, & \text{если } \varphi_x(x) \text{ определено (или } x \in W_x), \\ 0, & \text{если } \varphi_x(x) \text{ не определено (или } x \notin W_x). \end{cases}$$

Эту функцию можно назвать ***характеристической функцией*** данной массовой проблемы. Так что алгоритмическая разрешимость рассматриваемой проблемы, т.е. существование требуемого алгоритма будет означать (эффективную) вычислимость характеристической функции $f(x)$.

Итак, проблема распознавания самоприменимости алгоритмов в окончательном виде формулируется так: вычислима ли функция $f(x)$? Отрицательный ответ на него даёт следующая теорема.

ТЕОРЕМА 7.2.1. ***Функция $f(x)$ не является вычислимой, и проблема распознавания самоприменимых алгоритмов (т.е. проблема "$x \in W_x$") алгоритмически неразрешима.***

Доказательство. Допустим противное, т.е. функция $f(x)$ вычислима. Применим диагональный метод, чтобы придти к противоречию. С помощью этого метода построим такую вычислимую функцию $g(x)$, область определения которой $Dom\ (g) \neq W_x$ ($= Dom\ (\varphi_x)$) при всех x, чего, очевидно, быть не может.

Как всегда при использовании диагонального метода будем добиваться того, чтобы множество $Dom\ (g)$ отличалось от множества W_x в точке x. Поэтому определим $Dom\ (g)$ следующим образом:

$$x \in Dom\ (g) \iff x \notin W_x\ ,$$

а саму функцию g определим так:

$$g(x) = \begin{cases} 0, & \text{если } x \notin W_x \text{ (т.е. если } f(x) = 0), \\ \text{не определено}, & \text{если } x \in W_x \text{ (т.е. если } f(x) = 1). \end{cases}$$

Поскольку, по допущению, функция $f(x)$ вычислима, то из определения следует, что вычислима и функция $g(x)$. Тогда в нумерации вычислимых функций одного аргумента она имеет некоторый номер m, т.е. $g = \varphi_m$. По определению области $Dom\ (g)$, имеем: $m \in Dom\ (g) \iff x \notin W_x$. Но $Dom\ (g) = Dom\ (\varphi_m) = W_m$ и, значит, $m \in W_m \iff m \notin W_m$, что является противоречием.

Следовательно, функция $f(x)$ невычислима, и рассматриваемая массовая проблема, которую формулируют кратко как "$x \in W_x$", или "$\varphi_x(x)$ определено", алгоритмически неразрешима. □

СЛЕДСТВИЕ 7.2.2. *Существует вычислимая функция h, для которой обе проблемы "$x \in Dom\ (h)$" и "$x \in Im\ (h)$" алгоритмически неразрешимы.*

Доказательство. Этими свойствами обладает, например, следующая функция:

$$h(x) = \begin{cases} x, & \text{если } x \in W_x, \\ \text{не определено}, & \text{если } x \notin W_x. \end{cases}$$

Эта функция вычислима, и ясно, что для неё справедливы утверждения:

$$x \in Dom\ (h) \iff x \in W_x \iff x \in Im\ (h)\ .$$

Поскольку в силу теоремы, проблема "$x \in W_x$" неразрешима, то неразрешимы и равносильные ей проблемы "$x \in Dom\ (h)$" и "$x \in Im\ (h)$" . □

Заметим, что доказанная теорема вовсе *не* утверждает, что мы не можем для любого *конкретного* числа a установить, будет ли определено значение $\varphi_a(a)$. Теорема лишь утверждает, что не существует единого *общего* метода решения вопроса о том, будет ли $\varphi_x(x)$ определено; другими словами, не существует метода, работающего при *всех* x.

Проблема распознавания применимости (или остановки) алгоритмов. Она формулируется так: существует ли алгоритм, который для любого алгоритма и любых наперёд заданных начальных данных определял бы, применим ли этот алгоритм к этим начальным данным, т.е. остановится ли алгоритм, будучи применённым

к этим начальным данным. Рассуждениями, подобными тем, которые были проделаны в начале предыдущего пункта, эта проблема трансформируется к строгой теоретико-алгоритмической проблеме: разрешима ли алгоритмически следующая массовая проблема: "$y \in W_x$", или "*функция* $\varphi_x(y)$ *определена*"; другими словами, вычислима ли характеристическая функция этой массовой проблемы:

$$g(x,y) = \begin{cases} 1, & \text{если } \varphi_x(y) \text{ определено (или } y \in W_x), \\ 0, & \text{если } \varphi_x(y) \text{ не определено (или } y \notin W_x)? \end{cases}$$

Из предыдущей теоремы 7.2.1 немедленно следует отрицательный ответ на этот вопрос. В самом деле, если бы функция g была вычислима, то вычислимой была бы и функция $f(x) = g(x,x)$, но f есть характеристическая функция проблемы "$x \in W_x$" и, согласно теоремы 7.2.1, она не вычислима. Следовательно, функция $g(x,y)$ не вычислима, и мы приходим к следующей теореме.

ТЕОРЕМА 7.2.3. *Массовая проблема распознавания применимости алгоритма к начальным данным (т.е. проблема остановки алгоритма, или проблема "$y \in W_x$") алгоритмически неразрешима.* □

Смысл этого утверждения для теоретического программирования очевиден: не существует общего метода проверки программ на наличие в них бесконечных циклов.

Обратим внимание на метод доказательства последней теоремы: мы, по существу, доказали, что проблема остановки ("$y \in W_x$") не проще проблемы самоприменимости ("$x \in W_x$"), и значит, если неразрешима более простая проблема ("$x \in W_x$"), то уж и подавно неразрешима и более сложная ("$y \in W_x$"). Этот метод называется *методом сведения* одной массовой проблемы к другой: проблема "$x \in W_x$" сводится к проблеме "$y \in W_x$".

Проблема распознавания нулевых функций. Её неразрешимость также доказывается путём сведения к ней неразрешимой проблемы распознавания самоприменимости "$x \in W_x$".

ТЕОРЕМА 7.2.4. *Проблема "$\varphi_x = 0$" алгоритмически неразрешима.*

Доказательство. Рассмотрим функцию f, определяемую формулой:

$$f(x,y) = \begin{cases} 0, & \text{если } x \in W_x, \\ \text{не определено}, & \text{если } x \notin W_x. \end{cases}$$

Эта функция вычислима. Тогда по теореме 5.2.2 (Клини о параметризации) существует всюду определённая вычислимая функция $k(x)$ такая, что $f(x,y) = \varphi_{k(x)}(y)$. Учитывая определение функции f, отсюда получаем:

$$x \in W_x \iff \varphi_{k(x)} = 0 \,. \qquad (*)$$

Обратимся теперь к нашей массовой проблеме "$\varphi_x = 0$" и рассмотрим её характеристическую функцию:

$$g(x) = \begin{cases} 1, & \text{если } \varphi_x = 0, \\ 0, & \text{если } \varphi_x \neq 0. \end{cases}$$

Допустим, что функция g вычислима. Тогда вычислимой будет и функция $h(x) = g(k(x))$ как суперпозиция вычислимых функций. Но эта функция с учётом равносильности $(*)$ может быть представлена в следующем виде

$$h(x) = \begin{cases} 1, & \text{если } \varphi_{k(x)} = 0, \text{ т.е } x \in W_x, \\ 0, & \text{если } \varphi_{k(x)} \neq 0, \text{ т.е } x \notin W_x. \end{cases}$$

Отсюда видно, что $h(x)$ есть характеристическая функция проблемы самоприменимости $x \in W_x$, которая согласно теоремы 7.2.1, алгоритмически неразрешима, а её характеристическая функция $h(x)$ невычислима.

Полученное противоречие доказывает, что функция g невычислима и, значит, характеризуемая ей массовая проблема "$\varphi_x = 0$" алгоритмически неразрешима. □

Эта теорема показывает, что при проверке правильности компьютерных программ мы сталкиваемся с принципиальными ограничениями: не существует общего эффективного метода для проверки того, будет ли программа вычислять нулевую функцию. Этот результат может быть усилен: вместо нулевой функции может быть взята любая фиксированная вычислимая функция. Об этом говорится в следующем пункте.

Проблема распознавания равенства двух вычислимых функций. Следующая теорема вытекает из предыдущей.

ТЕОРЕМА 7.2.5. *Проблема "$\varphi_x = \varphi_y$" алгоритмически неразрешима.*

Доказательство. Ясно, что данная проблема сложнее предыдущей "$\varphi_x = 0$": если взять такой номер c в нумерации вычислимых

функций, что "$\varphi_c = 0$", то видно, что проблема "$\varphi_x = 0$" является частным случаем проблемы "$\varphi_x = \varphi_y$" при "$y = c$". Поэтому если $f(x, y)$ – характеристическая функция проблемы "$\varphi_x = \varphi_y$", то функция $g(x) = f(x, c)$ есть характеристическая функция проблемы "$\varphi_x = 0$". Но по предыдущей теореме 7.2.4 функция g невычислима. Но тогда невычислима и функция f. Следовательно, массовая проблема "$\varphi_x = \varphi_y$" алгоритмически неразрешима. □

Эта теорема добавляет ограничения при проверке правильности компьютерных программ: не существует алгоритма проверки того, вычисляют ли две программы одну и ту же одноместную функцию.

Проблема входа и проблема выхода. Для доказательства неразрешимости этих проблем мы сведём к каждой из них неразрешимую проблему самоприменимости "$x \in W_x$", снова воспользовавшись при этом s-m-n-теоремой 5.2.1 Клини о параметризации.

ТЕОРЕМА 7.2.6. *Для любого числа c следующие проблемы неразрешимы:*

а) (*Проблема входа*) "$c \in W_x$", *или* "$c \in Dom\ (\varphi_x)$", *или* "*определена лн функция φ_x на числе c*", *или* "*остановится ли алгоритм $\mathbf{A}_x$, вычисляющий функцию φ_x, при подаче на вход числа c*";

б) (*Проблема выхода*) "$c \in E_x$", или "$c \in Im\ (\varphi_x)$", или "*является ли число c значением функции φ_x*, или "*появится ли когда-либо число c на выходе алгоритма $\mathbf{A}_x$, вычисляющего функцию φ_x*.

Доказательство. Сведём проблему "$x \in W_x$" к обеим этим проблемам одновременно. Рассмотрим следующую функцию:

$$f(x, y) = \begin{cases} y, & \text{если } x \in W_x, \\ \text{не определено}, & \text{если } x \notin W_x. \end{cases}$$

В силу тезиса Чёрча, функция f вычислима. Тогда по теореме 5.2.1 о параметризации найдётся всюду определённая функция $k(x)$, такая, что $f(x, y) = \varphi_{k(x)}(y)$.

Из определения f видно, что если $x \in W_x$, то $\varphi_{k(x)}(y) = f(x, y) = y$ для любого $y \in N$. Это означает, что $W_{k(x)} = E_{k(x)} = N$. Значит, в этом случае $c \in W_{k(x)}$ и $c \in E_{k(x)}$. Таким образом,

$$x \in W_x \quad \Longrightarrow \quad c \in W_{k(x)} \text{ и } c \in E_{k(x)} \qquad (*)$$

Если же $x \notin W_x$, то из определения f видно, что для любого $y \in N$ значение $\varphi_{k(x)}(y) = f(x,y)$ не определено, и следовательно, в этом случае $W_{k(x)} = E_{k(x)} = \emptyset$, и значит, $c \notin W_{k(x)}$ и $c \notin E_{k(x)}$. Таким образом,

$$x \notin W_x \quad \Longrightarrow \quad c \notin W_{k(x)} \text{ и } c \notin E_{k(x)} \qquad (**)$$

Утверждения (*) и (**) говорят о том, что проблема $x \in W_x$ сведена к проблеме $c \in W_x$; т.е. если мы сможем решать проблему входа, т.е. отвечать на вопрос "$c \in W_x$", то мы сможем решить и проблему самопрменимости: чтобы ответить на вопрос "$c \in W_c$", нужно ответить на вопрос "$c \in W_{k(c)}$". Но проблема "$x \in W_x$" неразрешима (теорема 7.2.1), значит, неразрешима и проблема входа "$c \in W_x$".

Аналогично, неразрешима и проблема выхода "$c \in E_x$".

Рассуждая в терминах характеристических функций проблем, можно следующим образом получить характеристическую функцию $\chi(x)$ проблемы самоприменимости "$x \in W_x$", выразив её через характеристическую функцию $\chi_1(x)$ проблемы входа "$c \in W_x$":

$$\chi_1(k(x)) = \begin{cases} 1, & \text{если } x \in W_x, \\ 0, & \text{если } x \notin W_x \end{cases} = \chi(x),$$

т.е. $\chi(x) = \chi_1(k(x)) = (\chi_1 \circ k)(x))$ т.е $\chi = \chi_1 \circ k$ – суперпозиция функций k и χ_1.

Функция $\chi(x)$ не является вычислимой (теорема 7.2.1). Значит, не вычислима и функция $\chi_1(x)$, ибо если бы мы могли вычислять $\chi_1(x)$, то вместе с вычислимостью функции $k(x)$ это дало бы вычислимость их суперпозиции, т.е. функции $\chi(x)$, и следовательно, разрешимость проблемы самоприменимости "$x \in W_x$", что не так. Следовательно, проблема входа "$c \in W_x$" не разрешима. □

Проблема распознавания общерекурсивных (т.е. всюду определённых вычислимых) функций. Доказательство алгоритмической неразрешимости этой массовой проблемы, впервые полученное Клини в 1936 г., основано на факте вычислимости универсальной функции и методе диагонализации.

ТЕОРЕМА 7.2.7. *Не существует алгоритма для определения по любому x, всюду ли определена функция* "φ_x, *т.е. массовая проблема* "φ_x всюду определена" *алгоритмически неразрешима.*

Доказательство. Характеристическая функция этой массовой

проблемы имеет следующий вид:

$$h(x) = \begin{cases} 1, & \text{если } \varphi_x \text{ всюду определена}, \\ 0, & \text{если } \varphi_x \text{ не всюду определена}. \end{cases}$$

Требуется доказать, что функция h невычислима. Допустим противное, т.е. h вычислима. Используя диагональный метод, построим тогда такую всюду определённую вычислимую функцию f, которая будет отличаться от каждой вычислимой функции φ_x. Зададим f посредством следующего определения:

$$f(x) = \begin{cases} \varphi_x(x) + 1, & \text{если } \varphi_x \text{ всюду определена}, \\ 0, & \text{если } \varphi_x \text{ не всюду определена}. \end{cases}$$

Из определения видно, что f отличается от каждой вычислимой функции φ_x нумерации (перечисления) $\{\varphi_x\}_x$, где пронумерованы *все* одноместные вычислимые функции. В самом деле, если φ_x всюду определена, то в точке x: $f(x) = \varphi_x(x)+1 \neq \varphi_x(x)$ и, значит, $f \neq \varphi_x$. Если же φ_x не всюду определена, то в той точке, в которой φ_x не определена, значение f равно 0 и, значит, снова $f \neq \varphi_x$.

Если $F(x,x)$ – универсальная функция для класса всех одноместных вычислимых функций, то определение для функции $f(x)$ с учётом определения для функции $h(x)$ можно представить в виде:

$$f(x) = \begin{cases} F(x,x) + 1, & \text{если } h(x) = 1, \\ 0, & \text{если } h(x) = 0, \end{cases}$$

где $F(x,x) = \varphi_x(x)$. В силу вычислимости функций F (теорема 5.2.3) h (допущение) из этого определения следует, что и функция f вычислима. Это противоречит выводу, полученному в конце предыдущего абзаца: f не равна ни одной одноместной вычислимой функции.

Следовательно, h невычислима, и массовая проблема "*φ_x всюду определена*" алгоритмически неразрешима. □

Не всякая частичная вычислимая функция может быть доопределена до всюду определённой вычислимой функции. Раз нельзя указать единого алгоритма, отличающего всюду определённые вычислимые (т.е. общерекурсивные) функции от частичных (частично рекурсивных), попытаемся подойти к проблеме частичных вычислимых функций с другой стороны. Может быть, возможно ***каждую*** частичную вычислимую функцию доопределить на

неопределимых точках так, чтобы получилась всюду определённая вычислимая функция. Оказывается, и эта задача неразрешима, что следует из приводимой ниже теоремы.

ТЕОРЕМА 7.2.8. *Существует такая частичная вычислимая функция, что никакая всюду определённая вычислимая функция не является её доопределением.*

Доказательство. Снова применяем диагональный метод. Рассмотрим следующую частичную функцию:

$$f(x) = \begin{cases} \varphi_x(x)+1, & \text{если } \varphi_x(x) \text{ определёно,} \\ \text{не определена}, & \text{если } \varphi_x(x) \text{ не определёно.} \end{cases}$$

Выражая $\varphi_x(x)$ через универсальную функцию $\varphi_x(x) = F(x,x)$ (которая вычислима на основании теоремы 5.2.3), приходим к тому, что функция f вычислима.

Покажем, что f будет отличаться от любой всюду определённой вычислимой функции хотя бы в одной точке. Пусть $g(x)$ – произвольная всюду определённая вычислимая функция. Тогда она входит в нумерацию всех вычислимых функций под некоторым номером c: $g = \varphi_c$. Так как g всюду определена, то $g(c) = \varphi_c(c)$. Тогда, по определению f, имеем: $f(c) = \varphi_c(c)+1 = g(c)+1 \neq g(c)$. Это и означает, что $f \neq g$. Теорема доказана. □

Эта теорема означает, что следовательно, существуют частичные алгоритмы, которые нельзя доопределить до всюду определённых алгоритмов.

Существование перечислимого, но не разрешимого множества. Пример такого множества уже был приведён нами в главе VI (теорема 6.3.2 Поста). Здесь, используя полученные результаты, мы укажем один класс множеств, обладающих указанными свойствами.

ТЕОРЕМА 7.2.9. *Область определения всякой частичной вычислимой функции, не имеющей всюду определённого вычислимого продолжения, является перечислимым, но неразрешимым множеством.*

Доказательство. Пусть $f(x)$ – такая частичная вычислимая функция, что никакая всюду определённая вычислимая функция не является её продолжением (доопределением). На основании предыдущей теоремы 7.2.8, такие функции существуют. Так как $f(x)$

вычислима, то по теореме 6.4.1, область определения этой функции $Dom\ (f)$ есть перечислимой множество.

Покажем, что множество $Dom\ (f)$ не разрешимо. Допустим противное, т.е. $Dom\ (f)$ – разрешимо. Рассмотрим тогда следующую функцию:

$$g(x) = \begin{cases} f(x), & \text{если } x \in Dom\ (f), \\ 0, & \text{если } x \notin Dom\ (f). \end{cases}$$

Ясно, что $g(x)$ является всюду определённым продолжением функции $f(x)$. Покажем, что функция $g(x)$ вычислима. Её вычислимость обеспечивается следующим алгоритмом. Сначала устанавливаем принадлежность (или непринадлежность) $x \in Dom\ (f)$. В силу разрешимости, по допущению, множества $Dom\ (f)$, имеется такая эффективная процедура (алгоритм), которая для каждого натурального числа даёт ответ на вопрос, принадлежит ли это число множеству $Dom\ (f)$. Затем, если $x \in Dom\ (f)$, то к x применяется алгоритм вычисления значения $f(x)$. Полученный результат будет значением $g(x)$. Если же $x \notin Dom\ (f)$, то алгоритм останавливается с выводом результата $g(x) = 0$.

Таким образом, функция $f(x)$ получает всюду определённое вычислимое продолжение, что противоречит условию. Следовательно, сделанное допущение неверно, и множество $Dom\ (f)$ неразрешимо. □

Интуитивный смысл этой теоремы состоит в том, что существуют множества, для которых имеется перечисляющий алгоритм, но нет разрешающего алгоритма, который для любого $x \in N$ давал бы ответ на вопрос, принадлежит ли x данному множеству. Другими словами, перечисляющие алгоритмы обладают большей эффективностью нежели разрешающие алгоритмы, поскольку с помощью перечисляющих алгоритмов может быть охарактеризован больший класс множеств нежели с помощью разрешающих алгоритмов.

Из этой теоремы вытекают два простых, но важных следствия.

Следствие 7.2.10. *Существуют множества, не являющиеся перечислимыми.*

Доказательство. Такими множествами будут, например, дополнения множеств, являющихся областями определения всякой частичной вычислимой функции, не имеющей всюду определённого вычислимого продолжения. В самом деле, если бы такое дополнение оказалось перечислимым, то сама область определения, будучи перечислимой по теореме 7.2.9, оказалась бы по теореме 6.3.2 Поста

разрешимым множеством, что также противоречило бы по теореме 7.2.9. □

Следствие 7.2.11. *Существуют функции, не являющиеся (алгоритмически) вычислимыми.*

Доказательство. Таковыми будут, например, характеристические функции всех множеств, не являющихся перечислимыми. Ведь все такие множества не являются и разрешимыми (в силу теоремы 6.3.1), что и означает невычислимость их характеристических функций. Кроме того, таковыми будут также характеристические функции всех множеств, являющихся перечислимыми, но не являющихся разрешимыми (такие существуют в силу теоремы 7.2.9). □

Проблема распознавания нетривиальных свойств вычислимых функций. Рассмотренные ранее неразрешимые проблемы носили довольно экзотический характер: все они были так или иначе связаны с самоприменимостью алгоритма, когда алгоритм работает с собственным описанием (находится значение вычислимой функции φ_x в точке, являющейся её собственным номером в выбранной нумерации вычислимых функций: $\varphi_x(x)$). Сейчас будет доказана теорема высокой степени общности, которая раскроет перед нами целый спектр алгоритмически неразрешимых массовых проблем. Из неё последуют некоторые результаты, доказанные ранее, а при доказательстве мы снова воспользуемся $s-m-n-$теоремой Клини (о параметризации) и методом сведения рассматриваемой проблемы к проблеме самоприменимости "$x \in W_x$".

Теорема 7.2.12. (Райс). *Пусть C – любой непустой собственный класс вычислимых функций от одного аргумента (существуют как функции, принадлежащие C, так и вычислимые функции, не принадлежащие C). Тогда не существует алгоритма, который бы по номеру x функции φ_x определял бы, принадлежит φ_x классу C или нет. Иначе говоря, проблема "$\varphi_x \in C$" алгоритмически неразрешима. Другими словами, множество $\{x : \varphi_x \in C\}$ неразрешимо.*

Доказательство. Характеристическая функция этой массовой проблемы есть характеристическая функция множества $M = \{x : \varphi_x \in C\}$:

$$\chi_M(x) = \begin{cases} 1, & \text{если } x \in M, \quad \text{т.е. } \varphi_x \in C, \\ 0, & \text{если } x \notin M, \quad \text{т.е. } \varphi_x \notin C. \end{cases}$$

Пусть g_0 – нигде не определённая функция. Рассмотрим сначала случай, когда $g_0 \notin C$. Выберем тогда какую-нибудь конкретную вычислимую функцию $g \in C$ и рассмотрим функцию $f(x, y)$, задаваемую следующим образом:

$$f(x,y) = \begin{cases} g(y), & \text{если } \varphi_x(x) \text{ опр., т.е. } x \in W_x, \\ g_0(y) \text{ (т.е. не опр.)}, & \text{если } \varphi_x(x) \text{ не опр., т.е. } x \notin W_x. \end{cases}$$

Функция $f(x, y)$ вычислима. Для её вычисления надо вычислять $\varphi_x(x)$: если $\varphi_x(x)$ определено, то этот процесс когда-нибудь остановится и нужно будет перейти к вычислению $g(y)$; если же $\varphi_x(x)$ не определено, то процесс не остановится, что равносильно вычислению нигде не определённой функции $g_0(y)$.

Из вычислимости функции $f(x, y)$ по теореме о параметризации ($s - m - n-$теорема Клини 5.2.1) заключаем, что существует такая всюду определённая вычислимая функция $k(x)$, что $f(x, y) = \varphi_{k(x)}(y)$. Отсюда с учётом определения функции $f(x, y)$ приходим к следующим выводам:

$$x \in W_x \implies \varphi_{k(x)} = g \implies \varphi_{k(x)} \in C\ ;$$

$$x \notin W_x \implies \varphi_{k(x)} = g_0 \implies \varphi_{k(x)} \notin C\ ,$$

или
$$x \in W_x \iff \varphi_{k(x)} \in C\ . \qquad (*)$$

Это означает, что с помощью вычислимой функции $k(x)$ мы свели проблему самоприменимости "$x \in W_x$" к исследуемой проблеме "$\varphi_x \in C$".

Предположим теперь на минуту, что наша проблема "$\varphi_x \in C$" разрешима, т.е. её характеристическая функция χ_M вычислима. Тогда вычислимой будет и функция $h(x) = \chi_M(k(x))$ (как суперпозиция вычислимых функций). Эта функция имеет вид:

$$h(x) = \begin{cases} 1, & \text{если } k(x) \in M, \text{ т.е. } \varphi_k(x) \in C, \\ 0, & \text{если } k(x) \notin M, \text{ т.е. } \varphi_k(x) \notin C. \end{cases}$$

Учитывая равносильность $(*)$, функция h приводится к виду:

$$h(x) = \begin{cases} 1, & \text{если } x \in W_x, \\ 0, & \text{если } x \notin W_x, \end{cases}$$

т.е. это есть характеристическая функция проблемы самоприменимости "$x \in W_x$", неразрешимость которой, а также невычислимость функции h доказаны в теореме 7.2.1.

Полученное противоречие доказывает теорему. □

Отметим, что из теоремы Райса немедленно вытекает неразрешимость проблемы "$\varphi_x = 0$" (теорема 7.2.4), если в качестве C взять множество, состоящее из единственной нуль-функции 0.

Дальнейшие примеры алгоритмически неразрешимых массовых проблем. Перечислим ряд массовых проблем, алгоритмическая неразрешимость которых может быть доказана с использованием $s-m-n$-теоремы Клини о параметризации и метода диагонализации. Доказательства проводятся методом сведения каждой из этих проблем к проблеме самоприменимости "$x \in W_x$" (теорема 7.2.1), т.е. доказывается, что если бы мы смогли эффективно решить исследуемую проблему, то мы могли бы воспользоваться этим для получения эффективного метода решения проблемы самоприменимости алгоритмов.

Вот некоторые из этих проблем:

1) Проблема "$\varphi_x = c$" определения по любому x, является ли функция φ_x постоянной величиной.

2) Проблема "$x \in E_x$" определения по любому x, входит ли x в область значений функции φ_x.

3) Проблема "E_x *бесконечно*" определения по любому x, бесконечно ли множество значений функции φ_x.

4) Проблема "$W_x = W_y$" определения по любым x, y, совпадают ли области определения функции φ_x и φ_y.

5) Проблема "$W_x = \emptyset$" определения по любому x, является ли функция φ_x нигде не определённой.

6) Проблема "$\varphi_x(x) = 0$" определения по любому x, принимает ли функция φ_x в точке x значение 0.

7) Проблема "$\varphi_x(y) = 0$" определения по любым x, y, принимает ли функция φ_x в точке y значение 0.

8) Проблема "$\varphi_x = g$" определения по любому x, является ли функция φ_x любой фиксированной наперёд заданной вычислимой функцией g.

9) Проблема определения по любому x, является ли функция φ_x всюду определённой и постоянной.

О значении алгоритмически неразрешимых массовых проблем для теории алгоритмов и практики программирования. Начнём с комментариев по поводу теоремы 7.2.10 Райса. Она, по существу означает, что не существует единого алгоритма, который для каждой вычислимой функции (по её номеру) определял бы, обладает ли эта функция тем или иным свойством или нет, например, является ли эта функция постоянной, монотонной, периодической, ограниченной и т.п. Но это лишь первое приближение к пониманию смысла этой теоремы. Дело в том, что мы пытаемся создать единый алгоритм, который имеет дело с функциями. Но что значит иметь дело с функцией? Функция должна быть как-то задана. В данном случае функция f_x задаётся вычисляющим её алгоритмом $\mathbf{A}_x$ (мы помним, что каждая функция может вычисляться множеством алгоритмов). Разыскиваемый нами единый алгоритм как раз и имеет дело с алгоритмами, вычисляющими функции. Так вот, смысл теоремы Райса состоит в том, что *по описанию алгоритма, вычисляющего функцию, ничего нельзя узнать о свойствах функции, которую он вычисляет.* Ещё раз подчеркнём – не существует *единого* алгоритма, применимого к описаниям *всех* вычисляющих алгоритмов.

В частности, оказывается неразрешимой проблема *эквивалентности алгоритмов* (упоминавшаяся нами в конце параграфа 3.2): *по двум заданным алгоритмам нельзя узнать, вычисляют они одну и ту же функцию или нет.*

Каждый, кто имел дело с программированием (написанием компьютерных программ), знает, что по тексту сколько-нибудь сложной программы, не запуская её в работу, трудно понять, что она делает (какую функцию вычисляет). Если это понимание и приходит, то каждый раз по-своему; единого метода здесь не существует. Это своего рода практическое проявление теоремы Райса.

В курсе математической логики мы обсуждали проблемы синтаксиса и семантики языка при рассмотрении формальных аксиоматических теорий (см. Учебник МЛ, §28, п.п. "Интерпретации и модели формальной теории", "Семантическая выводимость"). В теории алгоритмов появляется ещё один аспект этой проблемы. Синтаксические свойства алгоритма – это свойства описывающих его текстов, т.е. свойства конечных слов в фиксированном алфавите. Семантические (или смысловые) свойства алгоритма связаны с тем, что делает алгоритм, что вычисляет; эти свойства естественно описывать в терминах функций, вычисляемых алгоритмом. Хорошо известно, что в процессе отладки программ синтаксические ошибки

отыскиваются довольно легко (этому, в частности, способствуют и дополнительные программы-алгоритмы). Главные неприятности связаны именно с анализом семантики неотлаженной программы, т.е. с попытками установить, что же она делает вместо того, чтобы делать то, что мы хотим (и здесь нам уже никакие дополнительные программы помочь не могут). Образно выражаясь, можно сказать, что теорема Райса звучит так: *по синтаксису алгоритма ничего нельзя узнать о его семантике*, точнее, *не существует единого алгоритма, позволяющего узнавать это единым способом для всех алгоритмов*. Таким образом, теорема Райса отделяет синтаксис алгоритма от его семантики.

Обратим теперь внимание на роль понятия частичной определённости в теории алгоритмов, выявляемую в связи с обнаруженными алгоритмически неразрешимыми массовыми проблемами. Выявляя ещё на интуитивном уровне типические черты алгоритмов, мы отмечали одной из важнейших – результативность, т.е. достижение реального результата. Тем не менее, мы не исключили возможность того, что на некоторых допустимых начальных данных результат может быть не достигнут, т.е. мы допустили к рассмотрению частичные алгоритмы и вычисляемые ими частичные (не всюду определённые) функции. Это привело нас в конечном итоге к выявлению массовых проблем, которые не могут быть решены алгоритмическим путём. Важнейшая из них – проблема остановки: мы не можем эффективно определять для каждого алгоритма и набора начальных данных, даст ли алгоритм результат на этих данных, или нет. Чтобы ликвидировать возникшее неудобство, на ум приходят два пути: либо вообще исключить частичные алгоритмы из рассмотрения в общей теории алгоритмов, либо ввести стандартный метод доопределения частичных алгоритмов до полных (всюду определённых) алгоритмов. Доказанные теоремы об алгоритмической неразрешимости проблем показывают, что ни первое, ни второе эффективными методами сделать нельзя. Первый путь не возможен из-за теоремы 7.2.7 – нет эффективного способа распознавать частичные алгоритмы среди множества всех алгоритмов, и, следовательно, предлагаемый отбор невозможен. Второй путь не годится из-за теоремы 7.2.8, показывающей, что существуют частичные алгоритмы, которые нельзя доопределить до всюду определённых алгоритмов.

В итоге общая теория алгоритмов пошла по пути признания частичных алгоритмов и смирения с фактом существования алгоритмически неразрешимых массовых проблем. С теоретической точки

зрения алгоритмическая неразрешимость массовой проблемы – не неудача, а научный факт. Он означает лишь отсутствие единого способа для решения всех единичных задач данной бесконечной серии, в то время как каждая индивидуальная задача серии вполне может быть решена своим индивидуальным способом. Более того, может оказаться разрешимой (своим индивидуальным методом) не только каждая отдельная задача этого класса, но и целые подклассы задач этого класса. Поэтому, если проблема неразрешима в общем случае, нужно искать её разрешимые частные случаи. Задача в более общей постановке имеет больше шансов оказаться неразрешимой.

Так, несмотря на отсутствие единого алгоритма, позволяющего для каждой формулы логики предикатов определить, является ли она выполнимой или общезначимой, мы в курсе математической логики (см. Учебник МЛ, §21) ответили на этот вопрос применительно к конкретным индивидуальным формулам. Более того, мы даже сумели отыскать алгоритмы решения данной задачи для целых классов формул некоторых специальных видов (§23). Аналогична ситуация с диофантовыми уравнениями. Например, для частного случая диофантова уравнения

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \ = \ 0$$

хорошо известно, что все его целые корни следует искать среди делителей свободного члена a_0. Подробнее об этих алгоритмических проблемах логики и математики речь пойдёт в последующих главах IX и X.

В то же время понимание сути проблемы алгоритмической неразрешимости и знание основных алгоритмических неразрешимостей является одним из важных элементов современной математической и логической культуры, особенно, для тех, кто имеет профессиональные дела, связанные с компьютерами, программированием, и информатикой.

7.3. Частично разрешимые предикаты и частично разрешимые алгоритмические проблемы

В определении 5.1.1, обобщая понятие примитивно рекурсивного предиката, рассмотренное в параграфе 3.3, было введено понятие разрешимого предиката. Это такой предикат, характеристическая функция которого вычислима. Такие предикаты тесно связаны с разрешимыми множествами: множество M разрешимо тогда и только тогда, когда его характеристическая функция χ_M вычислима (см. параграф 6.1), что равносильно тому, что предикат принадлежности "$x \in M$" разрешим. В теореме 6.4.5 мы установили, что множество M перечислимо тогда и только тогда, когда его квазихарактеристическая функция h_M вычислима, что равносильно тому, что предикат принадлежности "$x \in M$", как говорят, частично разрешим. (Функция h_M отличается от функции χ_M тем, что она является частичной и не определена для $x \notin M$.)

Вполне естественно обобщить понятие частичной разрешимости предиката с предиката принадлежности "$x \in M$" на произвольные предикаты, рассмотреть свойства этого понятия и его связь с понятием разрешимости предиката. Кроме того, разрешимые предикаты тесно связаны с алгоритмически разрешимыми массовыми проблемами, а алгоритмически неразрешимые массовые проблемы, которые рассматривались в предыдущем параграфе 7.2, – с неразрешимыми предикатами. Так, фундаментальная проблема распознавания самоприменимости алгоритмов есть, по существу, проблема разрешимости предиката "$x \in W_x$", которая получила в теореме 7.2.1 отрицательное решение: предикат "$x \in W_x$" неразрешим. Этим вопросам и посвящается настоящий параграф.

Частично разрешимые предикаты. Начнём с определения.

Определение 7.3.1. Предикат $P(x_1, x_2, ..., x_n)$ называется *частично разрешимым* (или *квазиразрешимым*), если вычислима его частичная характеристическая (или квазихарактеристическая) функция:

$$h_P(x_1, x_2, ..., x_n) =$$

$$= \begin{cases} 1, & \text{если высказывание } P(x_1, x_2, ..., x_n) \text{ — истинно,} \\ \text{не опр.,} & \text{если высказывание } P(x_1, x_2, ..., x_n) \text{ — ложно.} \end{cases}$$

Это означает, что алгоритм, вычисляющий функцию h_P, выдаёт значение 1 на тех начальных данных $x_1, x_2, \ldots, x_n$, которые превращают предикат P в истинное высказывание, и работает бесконечно на тех начальных данных $x_1, x_2, \ldots, x_n$, которые превращают предикат P в ложное высказывание. Говорят, что такой алгоритм (вычислительная процедура) является *частично разрешающим* для предиката P.

Вместе с этим возникает понятие частичной (алгоритмической) разрешимости массовой проблемы, обобщающее понятие алгоритмической разрешимости массовой проблемы. При этом, некоторые неразрешимые массовые проблемы оказываются частично разрешимыми.

Из этого определения легко следует, что *любой разрешимый предикат частично разрешим*: для доказательства этого достаточно разрешающий алгоритм продолжить так, чтобы при получении результата 0 он зацикливался бы и работал бесконечно.

Приведём примеры частично разрешимых предикатов.

Предикат $P_7(x)$: "*В десятичном разложении числа* π *хотя бы раз встречается ровно* x *последовательно идущих друг за другом цифр* 7" является частично разрешимым. В самом деле, если описанная ситуация для данного x действительно имеет место, то соответствующий алгоритм рано или поздно (за конечное число шагов) её отыщет и выдаст результат 1; если же для данного x описанная ситуация места не имеет, то этот алгоритм будет работать вечно.

Аналогично устанавливается частичная разрешимость следующих предикатов:

$P_1(n)$: "*n является числом Ферма.*" (Число n называется числом Ферма, если существуют такие натуральные числа $x, y, z > 0$, что $x^n + y^n = z^n$).

$P_2(x)$: "$E_x^{(n)} \neq \emptyset$" (где число n фиксировано), т.е. "*Функция* $\varphi_x^{(n)}$ *не является нигде не определённой*".

$P_3(x, y)$: "*Значение* $\varphi_x(y)$ *есть полный квадрат*".

$P_4(x)$: "$x \in W_x$", т.е. "*Вычислимая функция* φ_x *определена на своём номере* x". (Проблема самоприменимости). Всякий алгоритм, вычисляющий функцию φ_x, будет частичной разрешающей процедурой для данного предиката: он даёт ответ ДА, когда $x \in W_x$, но он продолжает работать бесконечно, если утверждение $x \in W_x$ не выполняется.

$P_5(x)$: "$y \in W_x$", т.е. "*Вычислимая функция* φ_x *определена на* y". (Проблема остановки). Аналогично предыдущему примеру.

Приведём теперь пример предиката, который не является частично разрешимым.

$P_6(x)$: "$x \notin W_x$", т.е. "*Вычислимая функция* φ_x *не определена на своём номере* x". (Отрицание проблемы остановки). Напомним, что W_x – это область определения унарной вычислимой функции φ_x. Тогда рассуждаем. По определению, квазихарактеристическая функция h_{P_6} определена, т.е. $x \in Dom\ (h_{P_6})$, тогда и только тогда, когда $x \notin W_x$. Это означает, что области определения $Dom\ (h_{P_6})$ функции h_{P_6} и W_x функции φ_x не просто не совпадают, а даже не имеют ни одного общего элемента. Значит, $h_{P_6} \neq \varphi_x$, и так будет происходить для любого x. Значит, функция h_{P_6} не совпадает ни с одной вычислимой функцией от одного аргумента. Следовательно, квазихарактеристическая функция $h_{P_6}(x)$ не вычислима, а предикат P_6 не является частично разрешимым.

Установим теперь ряд свойств частично разрешимых предикатов.

Теорема 7.3.2. ***Если предикаты*** $P(x_1, ..., x_n)$ ***и*** $Q(x_1, ..., x_n)$ ***частично разрешимы, то их конъюнкция*** $P(x_1, ..., x_n) \wedge Q(x_1, ..., x_n)$ ***и дизъюнкция*** $P(x_1, ..., x_n) \vee Q(x_1, ..., x_n)$ ***также являются частично разрешимыми предикатами. В то же время отрицание*** $\neg P(x_1, ..., x_n)$ ***не обязательно является частично разрешимым предикатом.***

Доказательство. По условию, квазихарактеристические функции h_P и h_Q предикатов P и Q соответственно вычислимы. Тогда ясно, что квазихарактеристическая функция $h_{P \wedge Q}$ предиката $P \wedge Q$ равна 1 тогда и только тогда, когда обе функции h_P и h_Q равны 1. Следовательно, функция $h_{P \wedge Q}$ вычислима, так как вычислимы функции h_P и h_Q. Аналогично, вычислима функция $h_{P \vee Q}$, чем доказывается частичная разрешимость предиката $P \vee Q$.

Что касается операции отрицания, то примером, подтверждающим утверждение теоремы, может служить предикат $P(x)$: "$x \in W_x$". Он сам является частично разрешимым, а его отрицание $\neg P(x)$: "$x \notin W_x$" таковым не является. (См. пример $P_4(x)$ перед этой теоремой). □

Теорема 7.3.3. ***Предикат*** $P(x_1, ..., x_n)$ ***частично разрешим тогда и только тогда, когда его множество истинности*** $P^+(x_1, ..., x_n)$ ***является областью определения некоторой вычислимой функции*** $g(x_1, ..., x_n)$: $P^+ = Dom\ (g)$.

Доказательство. Если предикат $P(x_1, ..., x_n)$ частично разрешим, т.е. его квазихарактеристическая функция h_P вычислима, то по определению функции h_P, это и означает, что $P(x_1, ..., x_n)$ истинно (т.е. $(x_1, ..., x_n) \in P^+(x_1, ..., x_n))$ тогда и только тогда, когда $h_P(x_1, ..., x_n)$ определена (т.е. $(x_1, ..., x_n) \in Dom\ (h_P))$. Таким образом, $P^+ = Dom\ (h_P)$.

Обратно, пусть $P^+ = Dom\ (g)$ для некоторой вычислимой функции $g(x_1, ..., x_n)$, т.е.

$$(x_1, ..., x_n) \in P^+(x_1, ..., x_n) \iff (x_1, ..., x_n) \in Dom\ (g).$$

Это означает, что

$$h_P(x_1, ..., x_n) = 1 \iff (x_1, ..., x_n) \in Dom\ (g(x_1, ..., x_n)),$$

т.е. функция g определена для этого набора $(x_1, ..., x_n)$.

Но тогда квазихарактеристическую функцию h_P предиката P можно представить как суперпозицию $1 \circ g(x_1, ..., x_n)$ двух вычислимых функций: 1 (тождественная единица) и функции $g(x_1, ..., x_n)$, которая равна 1 для тех $(x_1, ..., x_n)$, для которых определена функция g, и не определена для тех $(x_1, ..., x_n)$, для которых не определена функция g: $h_P = 1 \circ g$. Следовательно, функция h_P вычислима (как суперпозиция вычислимых функций) и предикат $P(x_1, ..., x_n)$ частично разрешим. □

Связь частично разрешимых предикатов с разрешимыми предикатами. Здесь будут установлены разнородные связи между частично разрешимыми предикатами и разрешимыми предикатами.

ТЕОРЕМА 7.3.4. *Предикат $P(x_1, ..., x_n)$ частично разрешим тогда и только тогда, когда существует такой разрешимый предикат $R(x_1, ..., x_n, y)$, что предикаты $P(x_1, ..., x_n)$ и $Q(x_1, ..., x_n) \equiv (\exists y)(R(x_1, ..., x_n, y))$ равносильны (т.е. обладают одинаковыми множествами истинности):*

$$\boxed{P \text{ – ч.р.п.} \iff (\exists R)[R \text{ – разрешим, } P^+ = (\exists y)(R)^+]} \ .$$

Доказательство. Достаточность. Предположим, что $R(x_1, ..., x_n)$ есть разрешимый предикат, причём, такой, что $P(x_1, ..., x_n)$ истинно тогда и только тогда, когда истинно $(\exists y)(R(x_1, ..., x_n, y))$:

$$P(x_1, ..., x_n) \text{ – истинно} \iff (\exists y)(R(x_1, ..., x_n, y)) \text{ – истинно.} \quad (*)$$

Раз предикат $R(x_1, ..., x_n, y)$ разрешим, значит, его характеристическая функция $\chi_R(x_1, ..., x_n, y)$ вычислима. Рассмотрим тогда час-

тичную функцию, получающуюся с помощью оператора минимизации:

$$g(x_1, ..., x_n) \quad = \quad \mu y \,[R(x_1, ..., x_n, y)] \quad = \quad \mu y \,[\chi_R(x_1, ..., x_n, y) = 1].$$

В силу теоремы 3.5.3 и рассуждений пункта "*Вычислимые функции*" параграфа 5.1, функция g вычислима, так как получается с помощью оператора минимизации μ, исходя из вычислимой функции. Причём, функция $g(x_1, ..., x_n)$ определена для таких $(x_1, ..., x_n)$, для которых хотя бы для одного y истинным будет $R(x_1, ..., x_n, y)$:

$$(x_1, ..., x_n) \in Dom\ (g) \quad \Longleftrightarrow \quad (\exists y)(R(x_1, ..., x_n, y)) - \text{истинно.} \quad (**)$$

Из утверждений $(*)$ и $(**)$ следует:

$$P(x_1, ..., x_n) \ - \ \text{истинно} \qquad \Longleftrightarrow \qquad (x_1, ..., x_n) \in Dom\ (g),$$

т.е. $P^+ = Dom\,(g)$. Тогда в силу теоремы 7.3.3, предикат $P(x_1, ..., x_n)$ частично разрешим.

Необходимость. Предположим, что предикат $P(x_1, ..., x_n)$ частично разрешим и что частичная разрешающая процедура даётся алгоритмом A. Определим тогда предикат $R(x_1, ..., x_n, y)$ следующим образом:

$R(x_1, ..., x_n, y)$: "*Алгоритм A на начальных данных $x_1, ..., x_n$ заканчивает работу за y шагов.*"

В силу следствия 5.2.5 б, этот предикат разрешим. Более того, в силу частичности разрешающей процедуры. даваемой алгоритмом A, значение $P(x_1, ..., x_n)$ будет истинным тогда и только тогда, когда $P(x_1, ..., x_n)$ закончит работу, т.е. тогда и только тогда, когда $(\exists y)(R(x_1, ..., x_n, y))$ – истинно. Таким образом, предикаты $P(x_1, ..., x_n)$ и $(\exists y)(R(x_1, ..., x_n, y))$ равносильны, т.е. их множества истинности равны. □

Теорема 7.3.4 подсказывает важный способ интерпретации частично разрешимых предикатов. Она говорит о том, что частично разрешимые процедуры всегда могут быть представлены в виде процедуры неограниченного поиска числа y, обладающего некоторым разрешимым свойством $R(x_1, ..., x_n, y)$. Такой поиск проще всего проводить, полагая последовательно $y = 0, 1, 2, ...$ Если и когда найдено такое y, для которого свойство $R(x_1, ..., x_n, y)$ выполнено, то происходит остановка. В противном случае поиск продолжается бесконечно долго.

О следующей теореме, доказываемой на основании предыдущей, говорят, что она устанавливает замкнутость класса частично разрешимых предикатов относительно операции взятия квантора существования.

Теорема 7.3.5. *Если предикат* $P(x_1, ..., x_n, y)$ *частично разрешим, то частично разрешим и предикат* $(\exists y)\ (P(x_1, ..., x_n, y))$.

Доказательство. В силу предыдущей теоремы 7.3.4, для частично разрешимого предиката найдётся такой разрешимый предикат $R(x_1, ..., x_n, y, z)$, что $P(x_1, ..., x_n, y)$ истинно тогда и только тогда, когда истинно $(\exists z)(R(x_1, ..., x_n, y, z))$. А раз равносильны предикаты $P(x_1, ..., x_n, y)$ и $(\exists z)(R(x_1, ..., x_n, y, z))$, то будут равносильны и предикаты, получающиеся из них связыванием одной переменной квантором существования:

$$(\exists y)\ (P(x_1, ..., x_n, y)) \iff (\exists y)(\exists z)\ (R(x_1, ..., x_n, y, z)). \quad (*)$$

В теореме 5.3.3 было подробно описано, как можно взаимно однозначно и эффективно занумеровать упорядоченные пары натуральных чисел (канторовская нумерация). Применим здесь эту нумерацию для пар y, z. Тогда поиск пары чисел y, z, для которых истинно $R(x_1, ..., x_n, y, z)$, сведётся к поиску одного числа $u = c(y, z)$, такого, что истинно $R(x_1, ..., x_n, l(u), r(u))$. Равносильность $(*)$ тогда превращается в следующую:

$$(\exists y)\ (P(x_1, ..., x_n, y)) \iff (\exists u)\ (R(x_1, ..., x_n, l(u), r(u))).$$

Поскольку функции $l(u)$ и $r(u)$ вычислимы, а предикат $R(x_1, ..., x_n, y, z)$ разрешим, то разрешим и предикат $S(x_1, ..., x_n, u) \equiv R(x_1, ..., x_n, l(u), r(u))$, получающийся в результате подстановки в разрешимый предикат вычислимых функций. Следовательно, по предыдущей теореме 7.3.4, предикат $(\exists y)\ (P(x_1, ..., x_n, y))$ является частично разрешимым. □

Обобщением в результате m-кратного применения доказанной теоремы является следующее утверждение.

Следствие 7.3.6. *Если частично разрешим предикат* $P(x_1, ..., x_n,\ y_1, ..., y_m)$, *то частично разрешим и предикат* $(\exists y_1)...(\exists y_m)\ (P(x_1, ..., x_n,\ y_1, ..., y_m))$.

Отметим, что если предикат $P(x_1, ..., x_n, y)$ частично разрешим, то предикат $(\forall y)\ (P(x_1, ..., x_n, y))$ не обязательно является частично разрешимым.

Полученные результаты позволяют установить частичную разрешимость ряда предикатов.

Примеры 7.3.7.

а) Предикат "$x \in E_y^{(n)}$" (где n фиксировано) частично разрешим. Это так называемая проблема выхода: является ли число x значением вычислимой функции $\varphi_y^{(n)}$? В теореме 7.2.6 мы установили, что эта проблема (и соответствующий предикат) не разрешима. Частичная разрешимость этого предиката следует из того, что его можно следующим образом представить равносильным предикатом:

$$x \in E_y^{(n)} \iff (\exists z_1)...(\exists z_n)(\exists t)\ [\mathbf{A}_y(z_1, ..., z_n)\ \downarrow\ x \text{ за } t \text{ шагов}],$$

где предикат, стоящий в квадратных скобках *"Алгоритм $\mathbf{A}_y$, вычисляющий функцию $\varphi_y^{(n)}$, на начальных значениях $z_1, ..., z_n$ выдаст ответ x за t шагов"*, разрешим (на основании следствия 5.2.5 а). Значит, этот предикат частично разрешим, а следовательно, по следствию 7.3.6, частично разрешим предикат в правой части рассматриваемой равносильности, а вместе с ним частично разрешим и предикат "$x \in E_y^{(n)}$".

б) Предикат "$W_x \neq \emptyset$" частично разрешим. Он характеризует проблему определения по любому номеру x, не является ли функция φ_x нигде не определённой. Отметим, что отрицание этого предиката "$W_x = \emptyset$" характеризует неразрешимую проблему. Данный предикат можно представить равносильным образом:

$$W_x \neq \emptyset \iff (\exists y)(\exists t)\ [\mathbf{A}_x(y)\ \downarrow \text{ за } t \text{ шагов}]\ ,$$

где предикат, стоящий в квадратных скобках *"Алгоритм $\mathbf{A}_x$, вычисляющий функцию φ_x, на начальном значении y закончит работу за t шагов"*, разрешим, поэтому вновь достаточно воспользоваться следствием 7.3.6.

в) В формализованном исчислении предикатов (или высказываний) свойство доказуемости формул можно рассматривать как предикат. В самом деле, доказательство формулы F можно рассматривать как некоторый конечный объект, представляющий собой конечное множество d формул, построенных по определённым правилам. Таким образом, мы получаем предикат:

$P(d, F)$: *"Последовательность d" формул является доказательством формулы F"*.

Этот предикат разрешим: руководствуясь определением доказательства, для любых d и F можно узнать, является ли последовательность d формул доказательством формулы F. Тогда свойство до-

казуемости формулы F можно выразить следующим одноместным предикатом:

$$\text{"}F\text{ - доказуема"} \iff (\exists d)\ (P(d, F)) .$$

По теореме 7.3.5, предикат в правой части частично разрешим, а вместе с ним частично разрешимо и свойство доказуемости формул "*F – доказуема*".

Следующая теорема показывает, что относительно операций взятия ограниченных кванторов как существования, так и общности класс частично разрешимых предикатов также замкнут.

ТЕОРЕМА 7.3.8. *Если предикат $P(x_1, ..., x_n, y)$ частично разрешим, то частично разрешимы и предикаты:*

$$Q(x_1, ..., x_n, z) \equiv (\forall y < z)\ (P(x_1, ..., x_n, y)) \quad и$$

$$R(x_1, ..., x_n, z) \equiv (\exists y < z)\ (P(x_1, ..., x_n, y)) ,$$

получающиеся из данного предиката $P(x_1, ..., x_n, y)$ навешиванием ограниченного квантора общности или существования.

Доказательство. Подобно аналогичной ситуации с примитивно рекурсивными предикатами (теорема 3.3.15), квазихарактеристические функции этих предикатов имеют следующий вид:

$$h_Q(x_1, ..., x_n, z) = \Pi_{y=0}^{z} h_P(x_1, ..., x_n, y) ,$$

$$h_R(x_1, ..., x_n, z) = sg[\Sigma_{y=0}^{z} h_P(x_1, ..., x_n, y)].$$

Все функции, участвующие в построении функций h_Q и h_R, вычислимы. Следовательно вычислимы и сами функции h_Q и h_R, а значит, соответствующие предикаты частично разрешимы. □

Следующую теорему, характеризующую разрешимые предикаты, полезно сравнить с теоремой 5.3.2 Поста, характеризующей разрешимые множества: разрешимость достигается тогда и только тогда, когда одновременно объект и его дополнение обладают более слабым свойством – перечислимости или частичной разрешимости.

ТЕОРЕМА 7.3.9. *Предикат $P(x_1, ..., x_n)$ разрешим тогда и только тогда, когда он сам $P(x_1, ..., x_n)$ и его отрицание $\neg P(x_1, ..., x_n)$ частично разрешимы.*

Доказательство. Необходимость. Если предикат $P(x_1, ..., x_n)$ разрешим то очевидно разрешим и предикат $\neg P(x_1, ..., x_n)$: его характеристическая функция вычисляется через вычислимую характеристическую функцию предиката $P(x_1, ..., x_n)$: $\chi_{\neg P} = 1 \dot{-} \chi_P$.

Достаточность. Предположим, что алгоритмы $\mathbf{A}_1$ и $\mathbf{A}_2$ дают частичные разрешающие процедуры для предикатов $P(x_1, ..., x_n)$ и $\neg P(x_1, ..., x_n)$ соответственно. Это означает, что

$$P(x_1, ..., x_n) \text{ – истинно} \iff \mathbf{A}_1(x_1, ..., x_n) \text{ остановится } ;$$

$$\neg P(x_1, ..., x_n) \text{ – истинно} \iff \mathbf{A}_2(x_1, ..., x_n) \text{ остановится } .$$

При этом, в силу частичной разрешимости, для любого набора $x_1, ..., x_n$ остановится точно один из алгоритмов – *либо* $\mathbf{A}_1(x_1, ..., x_n)$, *либо* $\mathbf{A}_2(x_1, ..., x_n)$, но никак *не оба* алгоритма вместе. Тогда можем организовать следующий разрешающий алгоритм для предиката $P(x_1, ..., x_n)$: взяв набор значений $x_1, ..., x_n$, будем одновременно производить вычисления $\mathbf{A}_1(x_1, ..., x_n)$, и $\mathbf{A}_2(x_1, ..., x_n)$; точнее, будем выполнять последовательно шаг одного алгоритма, затем следующий шаг другого алгоритма, и т.д., до тех пор, пока один из алгоритмов не завершит работу. Если остановится алгоритм $\mathbf{A}_1(x_1, ..., x_n)$, то мы заключаем, что утверждение $P(x_1, ..., x_n)$ *истинно*; если же остановится алгоритм $\mathbf{A}_2(x_1, ..., x_n)$, то заключаем, что утверждение $P(x_1, ..., x_n)$ *ложно.* □

Из этой теоремы вытекает, в частности, ещё одно доказательство того, что предикат "$x \notin W_x$" не является частично разрешимым. В самом деле, поскольку предикат "$x \in W_x$" не разрешим (теорема 7.2.1), но частично разрешим (пример $P_4(x)$ после определения 7.3.1), поэтому его отрицание "$x \notin W_x$" не может быть частично разрешимым, поскольку в противном случае предикат "$x \in W_x$" был бы разрешим (по теореме 7.3.9), что не так.

Аналогично получаем следующее утверждение.

СЛЕДСТВИЕ 7.3.10. *Предикат "$y \notin W_x$" ("Вычислимая функция φ_x" не определена на y"* (проблема расхождения)) *не является частично разрешимым.*

Доказательство. Поскольку предикат "$y \in W_x$" (проблема остановки) не разрешим (теорема 7.2.3), но частично разрешим (пример $P_5(x)$ после определения 7.3.1), поэтому его отрицание "$y \notin W_x$" не может быть частично разрешимым, ибо в противном случае предикат "$y \in W_x$" был бы разрешим (по теореме 7.3.9), что не так. □

Признак вычислимости частичной функции. Следующая теорема на языке частичной разрешимости предикатов даёт полезный способ доказательства вычислимости различных функций.

ТЕОРЕМА 7.3.11. *Целочисленная функция* $f(x_1, ..., x_n)$ *вычислима тогда и только тогда, когда предикат* "$f(x_1, ..., x_n) = y$" *частично разрешим.*

Доказательство. Необходимость. Если функция $f(x_1, ..., x_n)$ вычислима с помощью алгоритма **A**, то предикат "$f(x_1, ..., x_n) = y$" можно представить следующим равносильным образом:

$$f(x_1, ..., x_n) = y \iff (\exists t)\ [\mathbf{A}(x_1, ..., x_n) \downarrow y \text{ за } t \text{ шагов}] ,$$

где предикат, стоящий в квадратных скобках "*Алгоритм* **A**, *вычисляющий функцию* f, *на начальном наборе значений* $x_1, ..., x_n$ *выдаст ответ* y *за* t *шагов*", разрешим (на основании следствия 5.2.5 а). А раз так, то по теореме 7.3.4 из приведённой равносильности следует частичная разрешимость предиката "$f(x_1, ..., x_n) = y$".

Достаточность. Предположим, что предикат "$f(x_1, ..., x_n) = y$" частично разрешим. Тогда по теореме 7.3.4, существует такой разрешимый предикат $R(x_1, ..., x_n, y, t)$, для которого выполняется равносильность:

$$f(x_1, ..., x_n) = y \iff (\exists t)\ (R(x_1, ..., x_n, y, t)) .$$

Тогда следующий алгоритм будет вычислять функцию $f(x_1, ..., x_n)$: зафиксировав набор значений $x_1, ..., x_n$, будем последовательно перебирать все пары чисел y, t и для каждой пары y, t с помощью алгоритма, разрешающего предикат $R(x_1, ..., x_n, y, t)$, будем определять, истинно или ложно значение $R(x_1, ..., x_n, y, t)$; если оно *истинно*, то это значит, что $f(x_1, ..., x_n) = y$; если алгоритм перебора, работая бесконечно, всё время выдаёт значение *ложно*, то это значит, что функция f для этих $x_1, ..., x_n$ не определена.

Отметим, что более строгое описание алгоритма, вычисляющего функцию f, состоит в том, чтобы, как и в доказательстве теоремы 7.3.5, с помощью канторовской нумерации занумеровать пары натуральных чисел, введя функции $u = c(y, t)$, $y = l(u)$, $t = r(u)$. Тогда поиск пары чисел y, t, для которых истинно $R(x_1, ..., x_n, y, t)$, сведётся к поиску одного числа $u = c(y, t)$, для которого истинно $S(x_1, ..., x_n, u) \equiv R(x_1, ..., x_n, l(u), r(u))$.

Таким образом, функция f вычислима. □

В параграфе 10.1 мы увидим, что частично разрешимые предикаты оказались тесно связанными со знаменитой 10-й проблемой Гильберта и помогли решить её почти через три четверти века после того, как она была поставлена.

Глава VIII

СЛОЖНОСТЬ ВЫЧИСЛЕНИЙ И МАССОВЫХ ПРОБЛЕМ

Когда от теоретических рассмотрений мы переходим к реальной практике вычислений, то на первый план выдвигается вопрос не о принципиальной возможности вычисления той или иной функции, а о практической возможности её вычисления. Другими словами, существует ли алгоритм (программа) и реальная машина, вычисляющая эту функцию за время и в том объёме памяти, которыми мы располагаем. Таким образом, при условии, что вычислительная задача разрешима, результат её реального решения будет зависеть от того, насколько хорош алгоритм (программа) её решения, насколько хороша машина (компьютер), реализующая алгоритм (быстродействие, объём памяти), и насколько сложна решаемая задача. Вопросами, относящимися к первому кругу и к третьему кругу, занимается теория сложности вычислений, массовых проблем и алгоритмов. Основам этой теории посвящается настоящая глава. Эти основы мы постараемся рассмотреть, исходя из нашего общего подхода к теории алгоритмов (см. главу V), не привязываясь к какой-либо определённой формализации понятия алгоритма, хотя, конечно же, выходы теории сложности в эти формализации (или вычислительные модели – машины Тьюринга, рекурсивные функции, нормальные алгоритмы Маркова и т.п.) имеют свои нюансы и тонкости.

Итак, если математические формализации понятия алгоритма (подобные машинам Тьюринга) привели математиков 30-ых годов XX века к изучению алгоритмически неразрешимых массовых проблем, то современные компьютеры поставили перед математиками вопросы иного рода, а именно, вопросы, связанные со сложностью практической реализации вычислений при компьютерном решении алгоритмически разрешимых массовых проблем.

8.1. Как измерять сложность вычислительных задач и массовых проблем

Мы хотим проанализировать на сложность вычислительные задачи (вычисления), массовые проблемы и алгоритмы их решения.

С интуитивной точки зрения *сложность вычисления* (решения вычислительной задачи, т.е. работы данного алгоритма на заданных входных данных) определяется ресурсами, требующимися для этого вычисления. Важнейшими в этом смысле ресурсами являются время (количество шагов или тактов работы алгоритма) и объём памяти (например, наибольший номер ячейки, над которой побывала считывающая головка машины Тьюринга в процессе вычисления).

Чтобы охарактеризовать сложность конкретного алгоритма (т.е. конкретного способа решения массовой задачи), нужно принять во внимание работу этого алгоритма на всех допустимых начальных данных.

Наконец, измерить сложность массовой проблемы – задача наиболее трудная, и решается она разными способами. В конце настоящего параграфа мы обсудим один, на первый взгляд, естественный способ характеризации сложности массовой проблемы и установим его несостоятельность.

Единичные вычислительные задачи и массовые проблемы. Мы уже отмечали, что массовая проблема представляет собой бесконечное множество однотипных вычислительных задач. Меняя (варьируя) начальные данные (параметры), мы переходим от одной конкретной задачи к другой. При этом, с каждой единичной задачей связывается некоторое натуральное число r, называемое ***размером*** этой задачи и характеризующее количество и величину данных, задействованных в этой задаче. Так, размером задачи нахождения наибольшего общего делителя чисел m и n может служить число $r = \max(m, n)$, задачи о вычислении определителя – порядок этого определителя, задачи о поиске пути на графе – число вершин графа. Т.о., размер задачи – это, по существу, то количество данных, которое будет подано на вход алгоритма, который будет решать эту задачу.

Пусть единичная задача $[Z, r]$ размера r является единичной задачей массовой проблемы Π, то есть $[Z, r] \in \Pi$. При этом в массовой проблеме Π может оказаться бесконечное множество единичных задач одного и того же размера r, а также имеются задачи сколь

угодно большого размера r.

К понятию размера единичной задачи можно подойти более строго, тем более, что эта величина будет в будущем главным аргументом функции, характеризующей сравнительную сложность различных единичных задач. Мы предполагаем, что с каждой массовой проблемой связана некоторая фиксированная схема кодирования, которая каждую единичную задачу этой проблемы (точнее, начальные данные этой задачи) записывает в виде конечной цепочки (слова) символов (букв) конечного алфавита в терминах входа алгоритма (или компьютера, исполняющего алгоритм). Тогда размер (или входная длина) единичной задачи Z из массовой проблемы Π определяется как число символов в цепочке, полученной применением к задаче Z схемы кодирования для массовой проблемы Π.

Пример 8.1.1. КОММИВОЯЖЕР. Имеются n городов $C = \{c_1, c_2, ..., c_n\}$, для которых известны расстояния между двумя любыми двумя c_i и c_j из них. Требуется найти кратчайший по длине путь обхода всех этих городов, начинающийся и заканчивающийся в одном и том же городе, такой, чтобы в каждом городе побывать лишь однажды. Таким образом, решение этой задачи – это такой упорядоченный набор (перестановка) $< c_{\pi(1)}, c_{\pi(2)}, ..., c_{\pi(n)} >$ заданных городов, который минимизирует величину:

$$\Sigma_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)}) .$$

Это выражение даёт величину маршрута, начинающегося в городе $c_{\pi(1)}$, проходящего последовательно через все города, заходя в каждый город только один раз, и возвращающегося в город $c_{\pi(1)}$ непосредственно из последнего города $c_{\pi(n)}$.

Рассмотрим, например, следующую единичную задачу этой массовой проблемы при $n = 4$: $C = \{c_1, c_2, c_3, c_4\}$, $d(c_1, c_2) = 10$, $d(c_1, c_3) = 5$, $d(c_1, c_4) = 9$, $d(c_2, c_3) = 6$, $d(c_2, c_4) = 9$, $d(c_3, c_4) = 3$.

Схему кодирования этой массовой проблемы можно создать на основании следующего алфавита:

$$A = \{ c, [,], /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}.$$

Тогда рассмотренная единичная задача о коммивояжере будет закодирована следующей цепочкой символов (входным словом):

$$c[1]\ c[2]\ c[3]\ c[4]\ //\ 10\ /\ 5\ /\ 9\ //\ 6\ /\ 9\ //\ 3\ .$$

Число символов в этой цепочке (букв в этом слове) равно **32**. Это и есть размер (или входная длина) данной единичной задачи: $r = 32$.

Более сложные единичные задачи кодируются аналогичным образом и определяется их размер.

Решением рассмотренной единичной задачи коммивояжера является следующий упорядоченный набор (перестановка) городов $\{c_1, c_2, c_4, c_3\}$, поскольку соответствующий маршрут имеет минимально возможную длину 27. Ясно, что тривиальным алгоритмом решения этой задачи является алгоритм перебора всех возможных маршрутов, т.е. перебора всех перестановок из четырёх элементов (их всего 4!=16), нахождения суммарной длины соответствующего маршрута и выбора из них наименьшего. Так, например, длина маршрута, соответствующего перестановке $\{c_4, c_3, c_2, c_1\}$, равна $3+6+9+10 = 28$. Решение может быть не единственным: суммарной длиной 27 обладает также маршрут $\{c_2, c_4, c_3, c_1\}$.

Как известно, с ростом n число перестановок из n элементов, т.е. число $n!$ быстро возрастает, и задача становится "труднорешаемой" и даже практически неразрешимой.

Вычислительная сложность единичной задачи. Предположим теперь, что массовая проблема П алгоритмически разрешима, т.е. имеется алгоритм **А**, решающий все единичные задачи $[Z, r]$ массовой проблемы П.

Вычислительной сложностью работы алгоритма **А** при решении единичной задачи $[Z, r] \in \Pi$ называется число шагов (тактов работы) алгоритма, сделанных для решения этой задачи, до получения ответа. Обозначение: $vs_A([Z, r])$. Таким образом, это есть функция размера решаемой задачи или размера входа алгоритма, а её значением является, по существу, время работы алгоритма для решения задачи. Очевидно, что с ростом размера r задач Z вычислительная сложность $vs_A([Z, r])$ одного и того же алгоритма A при решении таких задач возрастает. С другой стороны, чем меньше вычислительная сложность $vs_A([Z, r])$ алгоритма **А**, тем большее количество единичных задач фиксированного размера r (или размера, не превосходящего r) можно решить в отведённое время. Наконец, чем медленнее растёт функция $vs_A([Z, r])$ от r, тем больше размер r единичных задач, которые можно решить с помощью алгоритма A в отведённое время.

Как мы уже говорили, с более абстрактной точки зрения массовая проблема есть проблема вычисления (частичной) функции $\varphi^{(n)}(x_1, ..., x_n)$ от n аргументов, заданной и принимающей значения в множестве N натуральных чисел. Каждая единичная задача этой массовой проблемы есть задача вычисления значения этой функции на конкретном наборе значений её аргументов. Все вычис-

лимые функции эффективно занумерованы (см. параграф 5.1), т.е. выстроены в последовательность

$$\varphi_0^{(n)}, \quad \varphi_1^{(n)}, \quad \varphi_2^{(n)}, \quad \dots \quad \varphi_k^{(n)}, \quad \dots$$

так, что по номеру (натуральному числу) k функция может быть однозначно восстановлена. Этот же номер (или индекс) является также номером алгоритма $\mathbf{A}_k$, вычисляющего функцию $\varphi_k^{(n)}$.

ОПРЕДЕЛЕНИЕ 8.1.2. Для каждого алгоритма $\mathbf{A}_e$ введём функцию:

$$t_e^{(n)}(x_1, ..., x_n) = \begin{cases} \text{число элементарных шагов, сделанных} \\ \text{при вычислении функции } \varphi^{(n)}(x_1, ..., x_n) \\ \text{алгоритмом } A_e\ ; \\ \text{не определена, если } \varphi^{(n)}(x_1, ..., x_n) \\ \qquad\qquad\qquad \text{не определена.} \end{cases}$$

Эта функция называется *вычислительной сложностью* алгоритма $\mathbf{A}_e$, или функции $\varphi^{(n)}(x_1, ..., x_n)$. (Она называется также *сигнализирующей функцией*).

Если считать, что каждый элементарный шаг, сделанный алгоритмом при вычислении функции, выполняется за определенную единицу времени, то вычислительная сложность алгоритма – это по существу время его работы, и чем быстрее работает алгоритм, тем он лучше, эффективнее.

В следующей лемме устанавливаются некоторые простые, но важные свойства сигнализирующих функций.

ЛЕММА 8.1.3.

а) $Dom\ (t_e^{(n)}(x_1, x_2, ..., x_n)) = Dom\ (\varphi_e^{(n)}(x_1, x_2, ..., x_n))$

для всех n, e.

б) Для каждого $n \in N$ предикат

$M(e, x_1, x_2, ..., x_n, y)$: "$t_e^{(n)}(x_1, x_2, ..., x_n)) = y$" разрешим.

Доказательство. Первое свойство вытекает непосредственно из только что введённого определения функции $t_e^{(n)}$. Предикат из второго свойства равносилен предикату $H_n(e, x_1, x_2, ..., x_n, t)$ из следствия 5.2.5б (переменную t нужно переобозначить через y) и, в силу этого следствия, также разрешим. □

Оказывается, в рамках так определяемого понятия меры вычислительной сложности можно доказать, что существуют функции со сколь угодно высокой сложностью вычислений, т.е. произвольно сложные вычислимые функции. Точнее, какую бы вычислимую функцию мы ни взяли, всегда найдётся вычислимая функция, вычислимая сложность которой будет выше (больше), вычислительной сложности этой функции. При этом, данное свойство имеет место, во-первых, даже для функций одного аргумента, а, во-вторых, как говорят, почти для всех натуральных n, или для достаточно больших n. Определим последнее понятие более точно.

Говорят, что предикат (свойство) $S(n)$ выполняется почти для всех n или почти всюду (п.в.), если $S(n)$ выполняется для всех, кроме конечного (включая пустое) множества натуральных чисел (или эквивалентно, если существует такое натуральное число n_0, что $S(n)$ истинно для всех $n \geq n_0$).

Теорема 8.1.4. *Пусть $b(n)$ – всюду определённая вычислимая функция. Существует всюду определённая вычислимая функция $f(n)$ (принимающая только значения 0 и 1), вычислительная сложность которой больше функции $b(n)$ почти всюду.*

Доказательство. Функция f в последовательности вычислимых функций имеет индекс e, т.е. $f = \varphi_e$, и утверждение теоремы означает, что $t_e(x) > b(n)$ почти для всех n. Построение функции f будет осуществлено с помощью диагонального метода. Функция $f = \varphi_e$ должна быть построена такой, что если для некоторого индекса i имеет место почти для всех m неравенство $t_i(m) \leq b(m)$, то $f \neq \varphi_i$, т.е. f отличается от φ_i хотя бы при одном таком значении m.

Функцию f определим рекурсией следующим образом. На каждом шаге n при построении значения $f(n)$ функции f мы сначала либо определяем величину i_n, либо за конечный промежуток времени (конечное число шагов) решаем, что i_n не определено. Если i_n определено, то определяем $f(n)$ так, чтобы $f(n) \neq \varphi_{i_n}(n)$. Итак, сначала, считая, что значения $f(0)$, $f(1)$, ... , $f(n-1)$ уже определены, полагаем:

$$i_n = \begin{cases} \mu i\ [i \leq n \text{ и } i \text{ отличается от всех, ранее определённых} \\ \text{значений } i_k \text{ и } t_i(n) \leq b(n)], \quad \text{если такое } i \text{ существует;} \\ \text{не определено,} \quad \text{в противном случае .} \end{cases}$$

После этого определяем значение:

$$f(n) = \begin{cases} 1, & \text{если } i_n \text{ определено и } \varphi_{i_n}(n) = 0; \\ 0, & \text{в противном случае (т.е если } i_n \text{ не определено,} \\ & \text{или } i_n \text{ определено, но } \varphi_{i_n}(n) \neq 0). \end{cases}$$

Ясно, что функция $f(n)$ всюду определена и, кроме того, вычислима. Последнее объясняется тем, что существует конечная процедура, которая говорит нам для данного i, будет ли $t_i(n) \leq b(n)$, так как

$$t_i(n) \leq b(n) \iff (\exists y \leq b(n))[t_i(n) = y]$$

и предикат в правой части разрешим. (последнее утверждение не слишком очевидно, но вытекает из теоремы 5.2.3 о вычислимости универсальной функции). Кроме того, в случае, когда i_n определено, имеем $f(n) \neq \varphi_{i_n}(n)$ и, значит, $f = \varphi_{i_n}$.

Поскольку функция $f(n)$ вычислима, поэтому в последовательности всех вычислимых функций она имеет некоторый номер (индекс): $f = \varphi_e$. Из конца предыдущего абзаца следует, что если i_n определено, то $e \neq i_n$. Покажем теперь, что если i является таким индексом, что $t_i(m) \leq b(m)$ почти для всех m, то $i = i_n$ для некоторого n, и следовательно, $i \neq e$. Это будет означать, что функции $f = \varphi_e$ нет среди тех функций, для которых сложность вычислений $t(m) \leq b(m)$ для почти всех m, т.е. функция $f = \varphi_e$ – такова, что для неё $t_e(m) > b(m)$ почти для всех m.

В самом деле, пусть i таков, что $t_i(m) \leq b(m)$ для бесконечно многих m. Положим тогда: $p = 1 + \max\{k : i_k$ определено и $i_k < i\}$; если же нет ни одного определённого $i_k < i$, то положим $= 0$. Возьмём теперь такое n, что $n \geq i$, $n \geq p$ и $t_i(m) \leq b(m)$. Если для некоторого $k < n$ имеет место $i = i_k$, то требуемое доказано. Предположим теперь, что $i \neq i_k$ для всех $k < n$. Тогда на n-ом шаге определения значения $f(n)$ имеем: $i \leq n$, и i отличается от всех ранее определённых i_k, и $t_i(m) \leq b(m)$. Значит, по определению i_n, величина i_n определена и $i_n \leq i$. Но поскольку $n \geq p$, т.е. n больше максимального k, для которого i_k определено, поэтому $i_n \geq i$. Следовательно, $i = i_n$.

Этим доказательство теоремы завершается. □

Отметим, что существуют и другие подходы к характеризации сложности единичной вычислительной задачи. Один из таких основан, в частности, на оценке объёма машинной памяти, необходимого для её решения.

Сложность массовых проблем. Попытаемся теперь понять, как можно было бы измерить сложность массовой проблемы.

Введя в предыдущем пункте понятие вычислительной сложности (работы) алгоритма A при решении единичной задачи $[Z, r] \in \Pi$, мы получили возможность сравнивать единичные задачи по сложности, а алгоритмы их решения по качеству. Если при решении двух единичных задач $[Z_1, r] \in \Pi$ и $[Z_2, r] \in \Pi$ одного и того же размера r массовой проблемы Π одним и тем же алгоритмом A имеет место неравенство:

$$vs_A[Z_1, r] < vs_A[Z_2, r] ,$$

то будем говорить, что сложность относительно алгоритма A у второй задачи выше, чем у первой. Если это неравенство имеет место для любого алгоритма, решающего массовую проблему Π, то говорят, что задача Z_2 (алгоритмически) *сложнее* задачи Z_1. Если при решении одной единичной задачи $[Z, r] \in \Pi$ двумя различными алгоритмами A_1 и A_2 имеет место неравенство

$$vs_{A_1}[Z, r] < vs_{A_2}[Z, r] ,$$

то будем говорить, что алгоритм A_1 *лучше (быстрее) алгоритма* A_2 *для задачи* $[Z, r]$. Если это неравенство имеет место для любой единичной задачи $[Z, r] \in \Pi$, то говорят, что алгоритм A_1 *лучше (быстрее)* алгоритма A_2.

Чтобы определить теперь сложность массовой проблемы Π, выберем в ней самую сложную задачу $[Z_0, r] \in \Pi$ и применим для её решения самый лучший алгоритм A_0. Вычислительную сложность этого алгоритма при решении этой задачи и назовём *сложностью массовой проблемы* Π и обозначим $T_\Pi(r)$. Таким образом, $T_\Pi(r) = vs_{A_0}[Z_0, r]$, или в соответствии с определением:

$$T_\Pi(r) = \min_A \{\max_{[Z,r]\in\Pi} (vs_A[Z, r])\} =$$
$$= \min_A \{vs_A[Z_0, r]\} = vs_{A_0}[Z_0, r] .$$

Сложность массовой проблемы также выступает как функция размера r решаемых задач и получается, как говорят, в результате анализа "худшего случая" или является сложностью "в худшем случае". Последнее означает, что функция строится в предположении существования наиболее неблагоприятных для алгоритма (с точки зрения вычислений) единичных задач в данной массовой проблеме.

Но в этом, казалось бы, вполне естественном определении сложности массовой проблемы имеется ещё одно допущение – существование самого лучшего (наилучшего) алгоритма, решающего эту

массовую проблему. И это допущение оказывается неосуществимым на практике. Нам предстоит доказать результат, принадлежащий М.Блюму, называемый *теоремой об ускорении* и показывающий, что всегда найдётся такая функция f, для вычисления которой не будет существовать наилучшего алгоритма, вычисляющего её. Доказательство этого утверждения проделаем в два этапа. Сначала будет доказана ослабленная форма этого утверждения – теорема о псевдоускорении. Затем из неё будет выведена сама теорема об ускорении.

ТЕОРЕМА 8.1.5. (М.Блюм; теорема о псевдоускорении). *Пусть w – любая всюду определённая вычислимая функция. Существует всюду определённая вычислимая функция $f(x)$ такая, что для любого алгоритма A_i, вычисляющего $f(x)$, найдётся другой алгоритм A_j, обладающий следующими свойствами:*

а) *функция φ_j, вычисляемая алгоритмом A_j, всюду определена и $\varphi_j(x) = f(x)$ почти для всех $x \in N$;*

б) $w(t_j(x)) < t_i(x)$ *почти для всех $x \in N$* .

(Последние слова "*почти для всех $x \in N$*" означают, что соответствующее равенство или неравенство выполняется для всех натуральных x, за исключением, быть может, конечного их числа, что равносильно тому, что эти равенство или неравенство выполняются для всех натуральных x, больших некоторого n_0: $x > n_0$. Таким образом, в теореме о псевдоускорении отыскивается ускоренный алгоритм, вычисляющий данную функцию $f(x)$ не во всех точках, а *почти во всех*. В теореме об ускорении будет найден ускоренный алгоритм, вычисляющий данную функцию $f(x)$, т.е. вычисляющий её значения *во всех* точках.)

Доказательство. Сначала надо зафиксировать конкретную всюду определённую вычислимую функцию s, получаемую по s-m-n-теореме Клини (теорема 5.2.2), такую, что $\varphi_e^{(2)}(u,x) = \varphi_{s(e,u)}(x)$. Найдём конкретный индекс e, такой, что функция $\varphi_e^{(2)}$ всюду определена и обладает следующими свойствами (здесь через g_u обозначается следующая функция: $g_u(x) = \varphi_e^{(2)}(u,x)$):

а) $g_0 = f$ (функция, которую требуется найти в теореме);

б) для каждого u имеет место $g_u(x) = g_0(x)$ почти всюду;

в) если $f = \varphi_i$, то для g_{i+1} найдётся индекс j, такой, что $w(t_j(x)) < t_i(x)$ почти для всех $x \in N$; в действительности можно взять $j = s(e, i+1)$.

Очевидно, этого достаточно для доказательства теоремы.

На минуту представим себе e как произвольный, но фиксированный индекс. Рассмотрим u как параметр, определим вычислимую функцию $g(u,x)$, которая будет также явно и эффективно зависеть от e. Для конкретного числа e, которое мы выберем позднее, g явится определённой выше функцией $\varphi_e^{(2)}(u,x)$.

Функцию $g(u,x)$ определим рекурсией по x с фиксированным u следующим образом. При этом для её определения будут строиться также рекурсивным образом некоторые вспомогательные конечные множества $C_{u,x}$, называемые множествами вычеркнутых индексов. Будем определять значение $g(u,x)$, считая, что множества $C_{u,0}, C_{u,1}, \dots, C_{u,x-1}$ уже построены, а значения $g(u,0)$, $g(u,1)$, ..., $g(u,x-1)$ уже определены. Тогда полагаем:

$$C_{u,x} = \begin{cases} \{i :\ u \le i < x,\ i \notin \cup_{y<x} C_{u,y} \text{ и } t_i(x) \le w(t_{s(e,i+1)}(x))\}, \\ \qquad\qquad \text{если } t_{s(e,i+1)}(x) \text{ определено для } u \le i < x; \\ \text{не определено,} \qquad \text{в противном случае}. \end{cases}$$

(Конечно, если $x \le u$, то $C_{u,x} = \emptyset$ и определено.) Заметим, что для каждого i, если $t_{s(e,i+1)}(x)$ определено, то можно решить, будет ли $t_i(x) \le w(t_{s(e,i+1)}(x))$ (для этого нужно применить лемму 8.1.3 б) и будет ли $t_i(x)$ определено или нет.

Теперь $g(u,x)$ определяем следующим образом:

$$g(u,x) = \begin{cases} 1 + \max\{\varphi_i(x) :\ i \in C_{u,x}\}, & \text{если } C_{u,x} \text{ определено;} \\ \text{не определена,} & \text{в противном случае}. \end{cases}$$

(Если $C_{u,x}$ определено, то для каждого $i \in C_{u,x}$ должно быть определено $\varphi_i(x)$, так что в этом случае $g(u,x)$, конечно, определено.)

По тезису Чёрча таким образом определённая функция $g(u,x)$ является частичной вычислимой функцией, которая явно и эффективно зависит от e. Следовательно, согласно следствию 5.3.4 из теоремы о неподвижной точке (слегка обобщённому), найдётся индекс e, такой, что

$$g(u,x) = \varphi_e^{(2)}(u,x) \quad . \qquad (*)$$

С этого момента положим e равным такому индексу, что выполняется равенство (*); тогда e окажется индексом, обсуждавшимся в начале доказательства. Остаётся убедиться в том, что он обладает требуемыми свойствами.

Сначала покажем, что из $(*)$ следует всюду определённость функции $g(u,x)$. Фиксируем x; для $u \geq x$ множество $C_{u,x} = \emptyset$, так что сразу из определения получаем $g(u,x) = 1$. Для $u < x$ покажем обратной индукцией по u, что $g(u,x)$ определено. Предположим, что все значения $g(x,x)$, $g(x-1,x)$, ... , $g(u+2,x)$, $g(u+1,x)$ определены. Тогда из $(*)$ и определения функции s получаем, что все значения $\varphi_{s(e,x)}(x), \varphi_{s(e,x-1)}(x), \dots, \varphi_{s(e,u+1)}(x)$ определены; следовательно, определены и значения $t_{s(e,i+1)}(x)$ для всех i: $u \leq i < x$. Это в свою очередь означает, что множество $C_{u,x}$ определено, а следовательно, определено и значение $g(u,x)$. Таким образом, $g(u,x)$ – всюду определённая функция.

Теперь, обозначая функцию $g(u,x)$ через g_u, имеем

$$g_u = g(u,x) = \varphi_e^{(2)}(u,x) = \varphi_{s(e,u)}(x) .$$

(Второе равенство есть равенство $(*)$, третье – вытекает из определения функции s).

Мы должны проверить вышеуказанные свойства а) – в).

а) Если положим $f = g_0$, то функция f, конечно же, всюду определена, как это и требуется в теореме.

б) Фиксируем число u; нам надо показать, что $g(0,x)$ и $g(u,x)$ отличаются только в конечном множестве точек x. Из построения множества $C_{u,x}$ ясно, что для всякого x

$$C_{u,x} = C_{0,x} \cap \{u, u+1, ..., x-1\} .$$

Так как все множеств $C_{u,x}$ попарно не пересекаются (по построению), поэтому можно найти следующее число $v = \max\{x :$ $C_{0,x}$ содержит некоторый индекс $i < u\}$. Тогда для $x > v$ мы имеем $C_{0,x} \subseteq \{u, u+1, ..., x-1\}$, а следовательно, $C_{0,x} = C_{u,x}$. Это означает, что $g(0,x) = g(u,x)$ для $x > v$. Таким образом, $g(0,x) = g(u,x)$ почти всюду.

в) Пусть i является индексом для f; беря $j = s(e, i+1)$, получаем $\varphi_j = \varphi_{s(e,i+1)} = g_{i+1}$ (из приведённого выше), так что j является индексом для g_{i+1}. Покажем теперь, что $w(t_j(x)) = w(t_{s(e,i+1)}) < t_i(x)$ для всех $x > e$. Если бы это не имело место, то i должно было бы быть вычеркнуто в определении $g(0,x)$ для некоторого $x > i$, т.е. должно было бы быть $x > i$ с $i \in C_{0,x}$. Но тогда по построению g было бы $g(0,x) \neq \varphi_i(x)$, что является противоречием.

Это и завершает доказательство теоремы о псевдоускорении. □

Заметим, что теорема о псевдоускорении *эффективна*: по данному алгоритму $\mathbf{A}$, вычисляющему функцию f, можно эффективно найти другой алгоритм, который вычисляет функцию f почти всюду и почти всюду быстрее чем алгоритм $\mathbf{A}$.

Покажем теперь, как видоизменить приведённое доказательство теоремы о псевдоускорении, чтобы получить следующую теорему.

ТЕОРЕМА 8.1.6. (М.Блюм; теорема об ускорении). *Пусть w – любая всюду определённая вычислимая функция. Существует всюду определённая вычислимая функция $f(x)$ такая, что для любого алгоритма $\mathbf{A}_i$, вычисляющего $f(x)$, найдётся другой алгоритм $\mathbf{A}_k$, вычисляющий функцию $f(x)$ и такой, что $w(t_k(x)) < t_i(x)$ почти для всех $x \in N$.*

Доказательство. Без нарушения общности можно предположить, что функция w возрастает (или заменить w на большую, чем w, возрастающую функцию). Сначала посредством небольшой модификации доказательства теоремы 8.1.5 мы получим всюду определённую вычислимую функцию $f(x)$ такую, что для любого алгоритма $\mathbf{A}_i$, вычисляющего $f(x)$, найдётся другой алгоритм $\mathbf{A}_j$, такой, что

а) функция φ_j, вычисляемая алгоритмом $\mathbf{A}_j$, всюду определена и $\varphi_j(x) = f(x)$ почти для всех $x \in N$;

б) $w(t_j(x) + x) < t_i(x)$ почти для всех $x \in N$.

Для этого просто перепишем определение множества $C_{u,x}$, заменив "... и $t_i(x) \le w(t_{s(e,i+1)}(x))$" на "... и $t_i(x) \le w(t_{s(e,i+1)}(x) + x)$". Мы покажем, что получаемая таким образом функция $f(x)$ удовлетворяет требованиям теоремы.

Теперь предположим, что $f = \varphi_j$ и j выбрано с учётом указанных выше свойств а), б). Теперь наша задача состоит в изменении алгоритма $\mathbf{A}_j$ так, чтобы получить алгоритм A_{j^*}, который вычисляет $f(x)$ для всех x. Пусть $\varphi_j(x) = f(x)$ для всех $x > v$, и $f(m) = b_m$ для $m \le v$.

Изменим алгоритм $\mathbf{A}_j$, поместив в его начало некоторые дополнительные команды, которые должны задать значения функции $f(x)$ для $0 \le x \le v$.

Новый алгоритм $\mathbf{A}_{j^*}$ будет начинаться с команд, вычисляющих значения функции f на аргументах от 0 до v. Он последовательно будет проверять, равен ли аргумент 0, 1, 2, ... $v - 1$, v. Если при этом $x = m$ $(0 \le m \le v)$, то присваивается $f(x) := b_m$, и алгоритм прекращает свою работу (останов); если же $x \ne m$, то переходят

к проверке $x = m + 1$, и т.д., пока $x \leq v$. Если окажется, что $x \neq 0$, и $x \neq 1$, и $x \neq 2$, и т.д., $x \neq v$, то в действие вступает алгоритм $\mathbf{A}_j$, вычисляющий значения функции $\varphi_j(x)$ на всех остальных значениях аргумента (натуральных числах) $x > v$ (на этих значениях аргумента имеет место совпадение $\varphi_j(x) = f(x)$); вычислив соответствующее значение, алгоритм $\mathbf{A}_j$ останавливается, завершая работу всего алгоритма $\mathbf{A}_{j^*}$.

Очевидно, алгоритм $\mathbf{A}_{j^*}$ вычисляет функцию f; кроме того, найдётся число c, такое, что дополнительные команды добавляют не более чем c шагов ко всякому вычислению; т.е. для всех x имеет место $t_{j^*}(x) \leq t_j(x) + c$.

Таким образом, мы получаем

$$w(t_{j^*}(x)) \leq w(t_j(x) + c) \leq w(t_j(x) + x) < t_i(x)$$

почти всюду. (Второе неравенство выполняется для всех $x \geq c$ – в силу что, что функция w – возрастающая; третье неравенство – в силу условия б). Следовательно, выбрав $k = j^*$, мы доказываем теорему. □

Можно показать, что теорема об ускорении, в отличие от теоремы о псевдоускорении, ***не является эффективной.***

Если в доказанной теореме Блюма об ускорении взять, в частности, например, $w(x) = 2x$, то получим $2t_k(x) < t_i(x)$, то есть алгоритм $\mathbf{A}_k$ будет вычислять функцию f в два раза быстрее (лучше), чем алгоритм $\mathbf{A}_i$. Это означает, что для всякого алгоритма, вычисляющего функцию f, можно найти лучший него (более быстрый) алгоритм, также вычисляющий f, т.е. найти алгоритм, ускоряющий предыдущий алгоритм. Значит, наилучшего алгоритма, вычисляющего f, не существует.

Теорема М.Блюма об ускорении, доказанная в 1971 г., открывает серию результатов, полученных в 1970-ые годы, лежащих в основе современной теории сложности вычислений, не зависящей от того применительно к каким машинам (компьютерам) она рассматривается.

Теорема М.Блюма, с одной стороны, устанавливает тот факт, что нет предела совершенствования в составлении алгоритмов для решения массовых задач, но с другой стороны показывает несостоятельность наивного подхода к определению сложности массовой проблемы по сложности наилучшего алгоритма, её решающего. Наконец, эта теорема продолжает линию "отрицательных" теорем математической логики и теории алгоритмов, начатой теоремами

Гёделя о неполноте формальной арифметики и Райса об алгоритмической нераспознаваемости нетривиальных свойств вычислимых функций.

Забавным следствием теоремы об ускорении является следующий факт. Представим, что у нас имеется вычислительная машина, которая выполняет один шаг работы алгоритма за 1 секунду, и мы заменяем её новой машиной, работающей в 100 раз быстрее. Тогда вычисление, требующее $t(x)$ секунд на старой машине, будет выполняться за $t(x)/100$ секунд на новой. Рассмотрим теперь функцию f, определяемую теоремой об ускорении с ускоряющим множителем 100. Предположим, что функция f вычисляется алгоритмом $\mathbf{A}_i$ на новой быстрой машине за время $t_i(x)$. По теореме об ускорении существует новый алгоритм $\mathbf{A}_k$ для вычисления f такой, что $100t_k(x) < t_i(x)$ почти для всех x, т.е. $t_k(x) < t(x)/100 < t_i(x)$. Последнее неравенство говорит о том, что если мы теперь запускаем новый алгоритм $\mathbf{A}_k$ на старой машине, то вычислим функцию f быстрее, чем это сделает алгоритм $\mathbf{A}_i$ на новой (быстродействующей) машине. Отсюда мы делаем вывод, что по крайней мере для некоторых функций новая машина не имеет преимущества перед старой (для большинства входов)!

Отметим, что теорема об ускорении доказывается методом диагонализации и потому не является эффективной, т.е. нет пути практического построения той функции f, существование которой в теореме устанавливается. Эта функция остаётся экзотикой и не имеет никакого отношения к реальной практике вычислений. С точки же зрения теории вычислимости она решает важнейшую задачу: показывает, что определение сложности массовой проблемы по сложности наилучшего алгоритма, решающего её, невозможно.

Поэтому принят другой способ характеризации (измерения) сложности массовой проблемы (или соответствующей функции). Он состоит в том, что выделяются классы таких массовых проблем (функций), вычисление которых возможно при тех или иных задаваемых ограничениях на потребляемые ресурсы (машинное время и объём памяти). Этому подходу к характеризации сложности массовой проблемы посвящается следующий параграф.

8.2. Сравнение и классификация массовых проблем и алгоритмов по их сложности

Хотя наилучшего (наибыстрейшего) алгоритма, решающего ту или иную массовую проблему, не существует, тем не менее, в реальной практике составления алгоритмов и компьютерных программ для решения конкретной массовой проблемы мы должны стремиться именно к составлению, по возможности, максимально быстрой программы (алгоритма), занимающей при своей работе как можно меньше ячеек памяти, с тем, чтобы мы смогли практически реализовать выполнение этой программы и решение поставленной проблемы на имеющемся в нашем распоряжении на данный момент компьютере в имеющееся в нашем распоряжении время.

В этом параграфе мы хотим оценить, какими функциями могут характеризоваться вычислительные сложности алгоритмов с тем, чтобы они реально могли быть реализованы на вычислительных машинах.

Концепция сравнения массовых проблем и алгоритмов по их сложности. В предыдущем параграфе 8.1 мы отметили, что вычислительная сложность алгоритма для решения той или иной массовой проблемы представляет собой функцию, зависящую от размера единичной задачи этой массовой проблемы, т.е. от объема её входных данных, и равную числу шагов (времени) работы алгоритма до получения ответа. Обозначим такую функцию через $S(r)$. Её называют также *временной функцией сложности.* Тогда при сопоставлении двух алгоритмов, решающих одну и ту же массовую проблему, нас в первую очередь будет интересовать поведение функций сложности этих алгоритмов при неограниченном росте размера задач, т.е. объёма входных данных, т.е. сравнительное асимптотическое поведение этих функций при $r \to \infty$.

Методы асимптотического сравнения бесконечно больших функций (величин) давно разработаны математическим анализом. Практически нет необходимости в точном построении выражения для функции сложности $S(r)$, что для конкретных задач выполнить зачастую бывает довольно сложно. Функцию сложности достаточно представить в виде так называемой асимптотической оценки, т.е. сравнить поведение этой функции при $r \to \infty$ с некоторой идеализированной (более простой и наглядной) функцией $g(r)$. Возникающей в результате такого представления ошибкой (погрешностью) на

практике можно пренебречь. Достаточно определить лишь порядок роста функции сложности с увеличением размерности решаемой задачи. Такой подход позволяет применить найденную оценку при решении задачи на различных типах компьютеров, абстрагируясь от того, сколько времени занимает выполнение одной операции, какую систему команд имеет этот компьютер и т.п.

Таким образом, нас интересует скорость роста сложности алгоритма. Здесь будут полезны следующие понятия, известные из математического анализа.

Если $\lim_{r\to\infty}\frac{S(r)}{g(r)}=\infty$, то пишут $S(r)=\Omega(g(r))$ и говорят, что функция $S(r)$ имеет *большую*, чем $g(r)$, а $g(r)$ имеет *меньшую*, чем $S(r)$, сложность. Иначе говоря, $S(r)=\Omega(g(r))$, если существует такая константа $C>0$, что $S(r)\geq C{\cdot}g(r)$ для достаточно больших r.

Если $\lim_{r\to\infty}\frac{S(r)}{g(r)}=C\neq 0$, то пишут $S(r)=O(g(r))$ и говорят, что функции $S(r)$ и $g(r)$ *одной сложности*. Иначе говоря, $S(r)=O(g(r))$, если существует такая константа $C>0$, что $S(r)\leq C{\cdot}g(r)$ для достаточно больших r.

Если $\lim_{r\to\infty}\frac{S(r)}{g(r)}=1$, то функции $S(r)$ и $g(r)$ называют *асимптотически эквивалентными* и пишут $S(r)\equiv g(r)$.

Так например, функции $S(r)=5n^2+15n-14$ и $g(r)=n^2$ одной сложности, а функция $g_2(r)=\exp(r)$ имеет сложность большую, чем функция $S(r)$. Приведём примеры функций, расположенных а порядке возрастания их асимптотической сложности при $r\to\infty$:

$$\ldots,\ \ln\ln r,\ \ldots,\ \ln r,\ \ldots,\ r^k,\ \ldots,\ \tfrac{1}{r},\ \ldots,\ \exp(r),\ \ldots,\ r!,\ldots,\ r^r,\ \ldots$$

Итак, при изучении сложности алгоритма будем интересоваться только его поведением при применении к задачам достаточно большого размера, поскольку именно такие задачи определяют границы применимости алгоритма.

Сложностные классы массовых проблем. Теперь мы можем группировать массовые проблемы, собирая в один класс такие проблемы, функции сложности которых асимптотически не сложнее некоторой выбираемой наперёд функции. Если под массовой проблемой понимать проблему вычисления значений функции $\varphi(x)$, то взяв произвольную всюду определённую вычислимую функцию $b(x)$, естественно образовать класс K_b всех вычислимых функций, у которых имеются алгоритмы с функцией сложности (временем работы), не превосходящей функцию $b(x)$. Точнее говоря, по определению,

$$K_b = \{\varphi_e : \ \varphi_e \text{ всюду определена и } t_e(x) \leq b(x) \text{ почти для всех } x\}.$$

Класс K_b называется ***классом сложности***, функции $b(x)$ (относительно временной меры сложности $t_e(x)$) или ***сложностным классом.*** Напомним, что e – это индекс вычислимой функции φ_e в нумерации всех вычислимых функций.

Если $b'(x)$ – другая всюду определённая вычислимая функция такая, что $b'(x) \geq b(x)$ для всех x, то $K_b \subseteq K_b$. В этом случае естественно ожидать, что класс $K_{b'}$ содержит некоторые функции, не входящие в K_b, особенно если $b'(x)$ намного больше, чем $b(x)$. Тем не менее, это не так. Можно показать, что существуют такие функции $b(x)$ и $b'(x)$, что $b'(x)$ значительно больше $b(x)$ (в любое заданное число раз), но $K_{b'} = Kb$. Точнее говоря, теорема Бородина, называемая теоремой о пробелах (или о скачке) показывает, что функции $b(x)$ и $b'(x)$ можно выбрать так, что не существует временной функции сложности вычисления (или просто времени вычисления) $t_e(x)$, которая лежит между $b(x)$ и $b'(x)$ для бесконечного множества разных x. Дадим точную формулировку этой теоремы и докажем её.

ТЕОРЕМА 8.2.1. (Бородин; теорема о пробелах). ***Пусть $r(x)$ – всюду определённая вычислимая функция, такая, что $r(x) \geq x$ для всех x. Тогда существует всюду определённая вычислимая функция $b(x)$ такая, что:***

а) ***для каждого e и $x > e$, если $t_e(x)$ определена и $t_e(x) \geq b(x)$, то $t_e(x) \geq r(b(x))$;***

б) ***$K_b = K_{r \circ b}$ (где $r \circ b$ – суперпозиция функций r и b).***

Доказательство. Определим неформально функцию $b(x)$ следующим образом. Определим сначала последовательность натуральных чисел $k_0 < k_1 < ... < k_x$ с помощью следующих равенств:

$$k_0 = 0;$$

$$k_{i+1} = r(k_i) + 1 \quad (i < x) \ .$$

Рассмотрим попарно непересекающиеся полуинтервалы $[k_i, \ r(k_i))$, $0 \leq i \leq x$. (Никакие два из них не пересекаются ввиду определяющих равенств для чисел k_i и свойства функции r: $r(x) \geq x$; так что каждый последующий полуинтервал расположен на числовой прямой правее предыдущего и не "зацепляется" с ним.) Таких полуинтервалов имеется $x + 1$ штук. А поскольку чисел $t_e(x)$ для $0 \leq e < x$ имеется x штук, то по меньшей мере один из этих полуинтервалов не содержит ни одного числа $t_e(x)$ для $0 \leq e < x$. Пусть

i_x – наименьшее из чисел i таких, что полуинтервал $[k_i,\ r(k_i))$ не содержит ни одного числа $t_e(x)$, $0 \le e < x$. Положим: $b(x) = k_{i_x}$.

Нетрудно понять, что существует эффективная процедура для нахождения числа i_x: для всевозможных e и i нужно проверять $t_e(x) \in [k_i,\ r(k_i))$. А раз так, то в силу вычислимости функции r и определения с её помощью чисел k, становящимися значениями вновь определяемой функции $b(x)$, заключаем, что и функция $b(x)$ будет вычислимой.

Для завершения доказательства утверждения а) предположим, что $x > e$ и $t_e(x) \ge b(x)$. Тогда по построению функции $b(x)$, мы имеем $t_e(x) \notin [b(x),\ r(b(x)))$. А раз так, то ввиду того, что $t_e(x) \ge b(x)$, отсюда следует, что $t_e(x) > r(b(x))$, и мы доказали утверждение а).

Перейдём теперь к утверждению б). Ввиду того, что функция r такова, что $r(b(x)) \ge b(x)$, поэтому $K_b \subseteq K_{rob}$ (см. определение этих множеств). Покажем, что верно и обратное включение. Допустим противное. Это означает, что существует функция f такая, что $f \in K_{rob}$, но $f \notin K_b$. Это, в свою очередь, означает, что f вычислима с помощью такого алгоритма A_e, что $t_e(x) \le r(b(x))$ почти для всех x, но $t_e(x) > b(x)$ для бесконечно многих x (в противном случае мы имели бы $f \in K_b$). Это, очевидно, противоречит доказанному выше утверждению а). Следовательно, имеет место обратное включение: $K_{rob} \subseteq K_b$. И окончательно, $K_b = K_{rob}$.

Теорема доказана. □

Массовые проблемы принято следующим образом классифицировать по их сложности:

1) Проблемы, для решения которых имеются алгоритмы, сложность которых пропорциональна r или меньше (сложность не более линейной): $S(r) \le C \cdot r$.

2) Проблемы, для решения которых имеются алгоритмы полиномиальной сложности: $S(r) \le p_k(r)$, где $p_k(r) = r^k + a_1 r^{k-1} + \ldots + a_{k-1} r + a_k$.

3) Проблемы, сложность которых не больше чем экспоненциальная: $S(r) \le \exp(r)$, для которых не известны алгоритмы полиномиальной сложности, но и не доказано, что таких алгоритмов нет.

4) Проблемы, сложность которых не меньше, чем экспоненциальная: $\exp(r) \le S(r)$.

Проблемы первого класса – самые легко решаемые проблемы. К этому классу относятся, например, проблемы нахождения мини-

мального элемента массива чисел, вычисления значения линейной функции с единичным старшим коэффициентом, в теории графов – нахождение остова графа, проверка планарности графа и т.д.

Проблемы второго класса практически нужных размеров могут быть решены на компьютере. В него включаются также и все проблемы первого класса. Этот класс принято обозначать буквой **P**. К нему относятся, например, проблемы вычисления значений многочленов, определителей, решение систем линейных уравнений, проверка простоты данного натурального числа, проверка графа на связность, решение задач линейного программирования.

Граница, разделяющая хорошо и трудно решаемые проблемы, проходит между вторым и третьим классами. Среди специалистов по вычислительным наукам широко распространено мнение, что алгоритм оказывается полезным для решения массовой проблемы, если его сложность растёт полиномиально относительно размеров входных данных. Таким образом, эффективность вычислений фактически отождествляется с полиномиальностью. Для обоснования этого мнения есть две группы аргументов – теоретические и практические. С точки зрения теории, во-первых, полиномы удобны тем, что замкнуты относительно композиции. Во-вторых, класс **P** можно весьма точно определить с помощью любого математического формального определения понятия алгоритма (машины Тьюринга, рекурсивные функции, нормальные алгоритмы Маркова и т.п.), причём, все эти разумные модели вычисления обладают замечательным свойством: если задачу можно решить за полиномиальное время в одной из них, то её можно решить за полиномиальное время и во всех остальных. С практической точки зрения опыт программирования показал, что если для полиномиальных алгоритмов практически возникающие задачи решаются за разумное время, то для большинства алгоритмов, не имеющих полиномиальных верхних оценок, решение практических задач занимает неприемлемо большое время. Таким образом, **P** – это класс проблем, для решения которых имеются полиномиальные (т.е. эффективные) детерминированные алгоритмы, и мы можем найти точное решение каждой единичной задачи любой проблемы из этого класса за разумный промежуток времени. Такие проблемы называются также ***практически решаемыми***.

В третий класс входят проблемы: разложение натуральных чисел на простые множители, выполнимость формул логики высказываний, нахождение гамильтонова цикла графа, задача коммивояжера и т.д.

Наконец, к четвёртому классу относятся, например, проблемы генерации объектов, число которых возрастает экспоненциально или ещё быстрее. В частности, проблемы порождения всех перестановок данного множества объектов, нахождения всех циклов данного графа и т.д.

Итак, проблемы из третьего и четвёртого классов будем называть *труднорешаемыми*. Это – такие проблемы, для решения которых не существует алгоритма полиномиальной сложности. К ним, прежде всего, следует отнести массовые проблемы, вообще не имеющие алгоритмов для своего решения – алгоритмически неразрешимые массовые проблемы, рассматривавшиеся в главе VII. Поскольку они не могут быть решены никаким алгоритмом, то тем более неразрешимы и полиномиальным алгоритмом и, значит, действительно трудноразрешаемы в самом сильном смысле. Вторую группу труднорешаемых проблем образуют проблемы, для решения которых существуют экспоненциальные алгоритмы, в большинстве случаев представляющие собой алгоритмы полного перебора. Для многих из этих проблем доказано, что они не могут быть решены за полиномиальное время даже с помощью "недетерминированного" вычислительного устройства, обладающего способностью параллельно выполнять неограниченное количество независимых вычислений. В следующем параграфе мы увидим, что эта "нереалистичная" модель вычислительного устройства будет играть важную роль в теории так называемых NP-полных массовых проблем.

Первые примеры труднорешаемых "разрешимых" задач были построены в 60-ые годы XX века. Но они носили искусственный характер и были специально построены, чтобы обладать соответствующими свойствами. Только в начале 70-ых годов удалось показать, что некоторые "естественные" разрешимые проблемы труднорешаемы.

Отметим, что отнесение алгоритмически разрешимой массовой проблемы к тому или иному классу сложности состоит из двух этапов. Сначала нужно получить верхнюю оценку сложности, т.е. указать решающий алгоритм, принадлежащий одному из перечисленных классов, а затем доказать отсутствие более эффективных алгоритмов, чем известные на сегодняшний день. Вторая задача, неизмеримо более сложная, чем первая, поэтому для большинства массовых проблем их точная сложность остаётся неизвестной.

Рассмотрим некоторые простые примеры получения оценок сложности алгоритмов.

Пример 8.2.2. Пусть имеется некоторое множество целых чисел $A = \{a_1, a_2, ..., a_n\}$. Рассмотрим алгоритм нахождения минимального элемента массива A.

Шаг 1. Ввести n элементов массива A.

Шаг 2. Положить $i = 1$, $min = MAXINT$, где $MAXINT$ есть наибольшее целое число, представимое в компьютере.

Шаг 3. Если $min > a_i$, то положить $min = a_i$.

Шаг 4. Если $i < n$, то положить $i := i + 1$ и перейти к Шагу 3.

Шаг 5. Печатать минимальное число массива min.

Оценим сложность получения минимального элемента массива. Шаг 1 выполняется один раз и требует n единиц времени. Шаг 2 также выполняется один раз и на каждое присваивание значений величинам i и min необходима одна единица времени, то есть этот шаг требует 2 единицы времени.

Шаги 3 и 4 выполняются n раз каждый. Шаг 3 требует не более двух единиц времени, когда условие выполняется (единица времени на проверку условия и единица времени на присваивание нового значения величине min). Шаг 4 также требует 2 единицы времени – единица на проверку условия и единица на увеличение значения величины i.

Наконец, шаг 5 требует одну единицу времени.

Таким образом, функция сложности $T(n) = n + n \cdot (2 + 2) + 1 = 5n + 1$ показывает, что поиск минимального элемента массива A требует $5n + 1$ единиц времени. Асимптотическая оценка сложности равна $O(n)$, так как $\lim_{n \to \infty} \frac{5n+1}{n} = 5$ – константа.

Эту же оценку можно получить проще: шаг 1 требует для своего выполнения $O(n)$ единиц времени, шаги 3-4 также требуют $O(n)$ единиц времени, то есть в целом для нахождения минимального элемента массива нужно затратить $O(n) + O(n) = O(n)$ единиц времени.

Данный пример также показывает, что нахождение сложности алгоритма по его описанию или по реализующей его программе (сложность программы) приводит, как правило, к одному и тому же результату.

Пример 8.2.3. Рассмотрим алгоритм сортировки элементов массива $A = \{a_1, a_2, ..., a_n\}$ целых чисел "пузырьковым" методом в порядке возрастания их значений.

Шаг 1. Ввести n элементов массива A.

Шаг 2. Положить $i = 1$.

Шаг 3. Положить $j := i + 1$.

Шаг 4. Если $a_i > a_j$, то положить $x := a_i$, $a_i := a_j$, $a_j := x$.
Шаг 5. Если $j < n$, то положить $j := j + 1$ и перейти к шагу 4.
Шаг 6. Если $i < n - 1$, то положить $i := i + 1$ и перейти к шагу 3.
Шаг 7. Печатать элементы упорядоченного массива.

Найдём асимптотическую оценку временной сложности алгоритма. Шаг 1 выполняется один раз и требует для выполнения $O(n)$ единиц времени. Шаги 3-5 выполняются $O(n)$ раз, а шаги 2–6 выполняются $O(n) \cdot O(n) = O(n^2)$ раз. Шаг 7 также выполняется один раз и требует для выполнения $O(n)$ единиц времени. Следовательно, общее время работы алгоритма равно $O(n^2)$.

8.3. Основы теории NP-полных массовых проблем

В конце предыдущего параграфа мы уже отметили, что доказать, что задача труднорешаема зачастую бывает не менее трудно, чем найти эффективный алгоритм. Теория NP-полных массовых проблем предлагает ряд сравнительно простых методов доказательства того, что та или иная конкретная проблема столь же трудна, как и большое число других проблем, признанных очень трудными и уже много лет не поддающихся усилиям специалистов. Основное предназначение этой теории состоит в том, чтобы помочь разработчикам алгоритмов и направить их усилия на выбор таких подходов к решению массовых проблем, которые, вероятнее всего, приведут к практически полезным алгоритмам.

В основе теории NP-полных массовых проблем лежит несколько фундаментальных идей, высказанных С.Куком, Р.Карпом и рядом другим математиков в начале 70-х годов прошлого века. Первая идея связана с понятием алгоритмической сводимости одной массовой проблемой к другой, причём сводимость за полиномиальное время, то есть сводимость, которая выполняется с помощью алгоритма с полиномиальной временной сложностью. Этому понятию посвящается первый пункт настоящего параграфа. Вторая идея, и ей посвящается второй пункт, – выделение класса массовых проблем распознавания (проблем предусматривающих ответ ДА или НЕТ). Третья идея – связь этих проблем распознавания с формальными языками. Формальным языкам и грамматикам посвящается третий пункт, а их связи с проблемами распознавания – четвёртый. Затем выделяются классы **P** и **NP** формальных языков и соответствующих им массовых проблем распознавания, причём второй – с исполь-

зованием новых понятий – недетерминированных алгоритмов и недетерминированных машин Тьюринга соответственно. Обсуждаются взаимоотношения между классами **P** и **NP** языков и проблем распознавания. Далее, понятие полиномиальной сводимости формализуется на уровне формальных языков и с его помощью вводится понятие NP-полных языков и NP-полных проблем распознавания, приводятся примеры таких проблем. Наконец, обсуждаются взаимоотношения между классами **P**, **NP**, NP-полных проблем и трудно решаемых проблем.

Алгоритмическая сводимость массовых проблем. История развития науки показывает, что новое научное знание зачастую возникает с опорой на ранее полученные знания. Так, в математике новые теоремы доказываются с использованием теорем, доказанных ранее. В теории алгоритмов этот принцип находит своё выражение в сведении вопроса об алгоритмической разрешимости или неразрешимости одной массовой проблемы к аналогичному вопросу для другой массовой проблемы. Так, неразрешимость проблем остановки алгоритмов, распознавания нулевых функций, равенства двух вычислимых функций и т.д. была доказана сведением этих проблем к проблеме самоприменимости алгоритмов, неразрешимость которой была установлена раньше. Аналогично и для разрешимых массовых проблем. Например, массовая проблема решения биквадратных уравнений $x^4 + ax^2 + b = 0$ подстановкой $t = x^2$ сводится к алгоритмически разрешимой массовой проблеме решения квадратных уравнений $t^2 + at + b = 0$ и, следовательно, сама оказывается разрешимой.

Дадим определение алгоритмической сводимости одной массовой проблемы к другой. Рассмотрим две массовые проблемы $\Pi^1 = \{Z_i^1\}$ и $\Pi^2 = \{Z_j^2\}$, состоящие из единичных задач Z_i^1, $i = 1, 2, ...$, Z_j^2, $j = 1, 2, ...$ Каждая из задач Z_i^1 и Z_j^2 имеет свои исходные данные и свой ответ, находимый в процессе решения задачи. Пусть $\mathbf{A}_1$ и $\mathbf{A}_2$ – алгоритмы, решающие массовые проблемы Π^1 и Π^2 соответственно.

Предположим теперь, что между массовой проблемой Π^1, т.е. между множеством задач $\{Z_i^1\}$, и некоторым подмножеством $\hat{\Pi}^2$ множества задач массовой проблемы Π^2 установлено определённое соответствие

$$\Pi^1 = \{Z_i^1\} \longrightarrow \{\hat{Z}_j^2\} = \hat{\Pi}^2 \subseteq \Pi^2 ,$$

задаваемое алгоритмами $\mathbf{A}_1'$ и $\mathbf{A}_2'$: алгоритм $\mathbf{A}_1'$ по исходным данным задачи $Z_i^1 \in \Pi^1$ находит исходные данные соответствую-

щей ей задачи $\hat{Z}_j^2 \in \hat{\Pi}^2 \subseteq \Pi^2$, а алгоритм $\mathbf{A}_2'$ находит по ответу к задаче $\hat{Z}_j^2 \in \hat{\Pi}^2$ ответ к задаче $Z_i^1 \in \Pi^1$. При этом предполагается, что решение задачи Z_i^1 существует тогда и только тогда, когда существует решение соответствующей ей задачи $\hat{Z}_j^2$.

Определение 8.3.1. Говорят, что массовая проблема Π^1 *алгоритмически сведена* к массовой проблеме Π^2, если существуют алгоритмы $\mathbf{A}_1'$ и $\mathbf{A}_2'$, устанавливающие указанное выше соответствие между множеством задач массовой проблемы Π^1 и некоторым подмножеством $\hat{\Pi}^2$ задач массовой проблемы Π^2 . Будем этот факт записывать следующим образом: $\Pi^1 \Longrightarrow \Pi^2$.

Образно выражаясь, можно сказать, что массовая проблема Π^1 (алгоритмически) сводится к массовой проблеме Π^2, если имеется процедура, которая преобразует все составные части проблемы Π^1 в эквивалентные составные части проблемы Π^2. Причём, это преобразование сохраняет информацию: всякий раз, когда решение какой-либо индивидуальной задачи из Π^1 даёт положительный ответ, такой же ответ должен быть и в соответствующей индивидуальной задаче из проблемы Π^2, и наоборот. Или же, можно сказать, если метод решения проблемы Π^2 можно преобразовать в метод решения проблемы Π^1.

Тогда очевидно, что если массовая проблема Π^1 сведена к массовой проблеме Π^2 и массовая проблема Π^2 алгоритмически разрешима, т.е. существую алгоритмы $\mathbf{A}_1'$, $\mathbf{A}_2'$ и $\mathbf{A}_2$, то алгоритмически разрешима и массовая проблема Π^1: алгоритм $\mathbf{A}_1$ её решения – это последовательное исполнение алгоритмов $\mathbf{A}_1'$, $\mathbf{A}_2$ и $\mathbf{A}_2'$. Соответственно, если массовая проблема Π^1 сведена к массовой проблеме Π^2 и массовая проблема Π^1 алгоритмически неразрешима, то алгоритмически неразрешима и массовая проблема Π^2.

С точки зрения теории сложности алгоритмов важное значение имеет алгоритмическая сводимость специального вида – полиномиальная сводимость.

Определение 8.3.2. Говорят, что массовая проблема Π^1 *полиноминально сводится* к массовой проблеме Π^2, если существуют алгоритмы $\mathbf{A}_1'$ и $\mathbf{A}_2'$ с полиномиальной вычислительной сложностью, устанавливающие указанное выше соответствие. В этом случае будем писать $\Pi^1 \stackrel{P}{\Longrightarrow} \Pi^2$.

Тогда очевидно, что если массовая проблема Π^1 полиноминально сведена к массовой проблеме Π^2 и массовая проблема Π^2

имеет полиномиальную сложность, то и массовая проблема Π^1 также имеет полиномиальную сложность. Это объясняется тем, что последовательное исполнение трёх полиномиальных алгоритмов $\mathbf{A}_1'$, $\mathbf{A}_2$ и $\mathbf{A}_2'$ является алгоритмом полиномиальной сложности. Соответственно, если массовая проблема Π^1 полиноминально сведена к массовой проблеме Π^2 и не существует полиномиального алгоритма решения массовой проблемы Π^1, то не существует полиномиального алгоритма решения и массовой проблемы Π^2.

Нетрудно понять, что отношения полиномиальной сводимости обладает свойством транзитивности: если проблема Π^1 полиноминально сводится к Π^2, а проблема Π^2 полиноминально сводится к Π^3, то Π^1 полиномиально сводится к Π^3, так как последовательное выполнение двух полиномиальных алгоритмов является очевидно полиномиальным алгоритмом, т.е. если $\Pi^1 \stackrel{P}{\Longrightarrow} \Pi^2$, $\Pi^2 \stackrel{P}{\Longrightarrow} \Pi^3$, то $\Pi^1 \stackrel{P}{\Longrightarrow} \Pi^3$.

Приведём простой пример сведения одной массовой проблемы к другой. Пусть первая проблема состоит в том, чтобы выяснить, принимает ли одна из пропозициональных переменных X_1, X_2, ... X_n значение "истина", и при положительном решении выдать ответ ДА, а при отрицательном – НЕТ. Вторая задача заключается в том, чтобы найти максимальное значение в списке целых чисел. Каждая из них допускает простое и ясное решение, но предположим на минуту, что мы знаем решение задачи о поиске максимума, а задачу про пропозициональные переменные решать не умеем. Мы хотим свести задачу о пропозициональных переменных к задаче о максимуме целых чисел. Напишем алгоритм преобразования набора значений пропозициональных переменных в список целых чисел, который значению "ложь" сопоставляет число 0, а значению "истина" – число 1. Затем воспользуемся алгоритмом поиска максимального элемента в списке. По тому, как составлялся список, заключаем, что этот максимальный элемент может быть либо нулём, либо единицей. Такой ответ можно преобразовать в ответ в задаче о пропозициональных переменных, выдавая ответ ДА, если максимальное значение равно 1, и НЕТ, если оно равно 0.

Мы видели в примере 8.2.2, что поиск максимального элемента массива выполняется за линейное время (правда, там отыскивался минимальный элемент, но ясно, что этот алгоритм легко переделывается в алгоритм поиска максимального элемента массива). Кроме того, сведение первой задачи ко второй тоже требует линейного времени, поэтому задачу о пропозициональных переменных тоже

можно решить за линейное время.

Рассмотрим ещё один пример сведения одной массовой проблемы к другой. Первая проблема, называемая ВЫПОЛНИМОСТЬ (сокращённо, ВЫП), состоит в том, чтобы определить, выполнима ли формула

$$F(X_1, X_2, ..., X_n) \equiv D_1 \wedge D_2 \wedge ... \wedge D_m$$

алгебры высказываний, представляющая собой конъюнктивную нормальную форму (конъюнкцию дизъюнктивных одночленов D_1, D_2, ... , D_m) от пропозициональных переменных X_1, X_2, ... , X_n, т.е определить, принимает ли данная формула значение 1 ("истина") хотя бы при одном наборе значений её пропозициональных переменных X_1, X_2, ... , X_n.

ВЫПОЛНИМОСТЬ является одной из центральных проблем в математической логике, и построение эффективных алгоритмов для её решения имеет большое значение. Естественно, одно из решений состоит в том, чтобы испытать всевозможные наборы значений истинности для пропозициональных переменных X_1, X_2, ... , X_n и посмотреть, есть ли среди них набор, выполняющий данную формулу. К сожалению, этот очевидный и наглядный алгоритм неэффективен, т.е. для достаточно больших n не может быть осуществлён за разумное время, так как потребуется испытать 2^n наборов значений истинности (для каждой переменной возможны два значения – 0 или 1). Эффективного алгоритма, который решал бы проблему ВЫПОЛНИМОСТЬ, к настоящему времени не известно.

Поэтому интересно, что проблему ВЫПОЛНИМОСТЬ можно свести к другой известной проблеме из другой области математики – из области линейной алгебры – к задаче ЦЕЛОЧИСЛЕННОГО ЛИНЕЙНОГО ПРОГРАММИРОВАНИЯ (ЦЛП), т.е. проблему ВЫП можно сформулировать как проблему ЦЛП. Соответствующая формулировка получается моментально, если отождествить (интерпретировать) значение "истина" с 1 и значение "ложь" – с нулём 0. Тогда логическая операция дизъюнкция ("или") превращается в арифметическую операцию сложения, логическое отрицание $\neg X$ выражается как $1 - X$. Формула $F(X_1, X_2, ..., X_n) \equiv D_1 \wedge D_2 \wedge ... \wedge D_m$ выполнима тогда и только тогда, когда выполним каждый её дизъюнкт $D_1, D_2, ... , D_m$, что, в свою очередь, будет тогда и только тогда, когда в каждом её дизъюнкте D_i, $1 \leq i \leq m$ содержится некоторая переменная или её отрицание, принимающая или принимающее значение "истина", что на язык арифметики переводится следующим образом:

$$\Sigma_{X \in D_i} X \ + \ \Sigma_{\neg X \in D_i} (1 - X) \ \geq \ 1$$

для всех $1 \leq i \leq m$. Кроме того, накладываются ограничения на переменные

$$0 \ \leq \ X_1, \ X_2, \ ..., \ X_n \ \leq 1 \ \text{ – целые.}$$

Например, если формула F имеет вид

$$F(X_1, X_2, ..., X_n) \ \equiv \ (X_1 \vee X_2 \vee X_3) \ \wedge \ (X_1 \vee \neg X_2) \ \wedge \ (X_2 \vee \neg X_3) \ \wedge \\ \wedge \ (X_3 \vee \neg X_1) \wedge (\neg X_1 \vee \neg X_2 \vee \neg X_3), \qquad (*)$$

то соответствующая задача ЦЛП для неё будет иметь вид:

$$\begin{gathered} X_1 \ + \ X_2 \ + \ X_3 \ \geq \ 1 \ , \\ X_1 \ + \ (1 - X_2) \ \geq \ 1 \ , \\ X_2 + (1 - X_3) \ \geq \ 1 \ , \qquad (**) \\ X_3 \ + \ (1 - X_1) \ \geq \ 1 \ , \\ (1 - X_1) \ + \ (1 - X_2) \ + \ (1 - X_3) \ \geq \ 1 \ , \\ 0 \ \leq \ X_1, \ X_2, \ X_3 \ \leq 1 \ \text{ – целые.} \end{gathered}$$

Легко теперь понять, как выполнимость формулы (*) выражается ограничениями (*): формула (*) выполнима тогда и только тогда, когда в соответствующей задаче (**) имеется допустимая точка. Таким образом, формулировка (**) не задаёт задачу ЦЛП, поскольку мы не ищем минимума некоторого линейного функционала. Тем не менее, её легко преобразовать в эквивалентную обычную задачу ЦЛП. Например, в (**) первое неравенство можно заменить на $X_1 \ + \ X_2 \ + \ X_3 \ \geq \ y$ и максимизировать y. Тогда формула (*) выполнима в том и только в том случае, если оптимальное значение y' существует и удовлетворяет неравенству $y' \geq 1$.

Задача ЦЛП (**) принадлежит важному классу задач ЦЛП, в которых переменные могут принимать только два значения: 0 или 1. Такие задачи ЦЛП называются задачами двоичного линейного программирования, или задачами 0-1-линейного программирования (НОЛП). Последняя строка задачи ЦЛП (*) обычно записывают в виде: $X_j \in \{0, 1\}, \ j = 1, ..., n$.

Отметим, что и для решения задачи ЦЛП, к которой свелась задача ВЫП, несмотря на многолетние интенсивные исследования, также не найдено практического алгоритма для случаев большого числа переменных. Отметим также, что задача ЦЛП имеет очень широкие рамки, внутри которых могут быть сформулированы многие задачи разного характера. По-видимому, именно из-за этой своей

общности задача ЦЛП, а вместе с ней и задача ВЫП и многие другие задачи, сводящиеся в к ЦЛП трудны для практического решения.

Ещё один пример сведения одной массовой проблемы к другой, а именно проблемы ГАМИЛЬТОНОВ ЦИКЛ (ГЦ) к проблеме КОММИВОЯЖЕР (КВ) рассмотрен в примере 8.3.14 ниже.

Проблемы распознавания. Это такие массовые проблемы, каждая задача которых требует только два возможных ответа ДА или НЕТ. Таким образом, массовую проблему распознавания П можно мыслить себе как упорядоченную пару множеств D_Π, Y_Π, где D_Π – множество всех единичных задач массовой проблемы П, Y_Π – множество всех единичных задач этой проблемы с ответом ДА; $Y_\Pi \subseteq D_\Pi$.

Массовую проблему распознавания П, задачи которой требуют ответа ДА или НЕТ, можно рассматривать как предикат $\Pi(x)$, заданный на множестве слов α, являющихся записями исходных данных задач этой проблемы: он превращается в истинное высказывание "*Единичная задача* $\Pi(\alpha)$ *массовой проблемы* П *имеет ответ* ДА", если это действительно так. Если же единичная задача $\Pi(\alpha)$ имеет ответ НЕТ, то сформулированное высказывание будет ложным.

Если ввести определённую на множестве слов α функцию, принимающую только два значения 1 и 0:

$$\chi_\Pi(x) = \begin{cases} 1, & \text{если единичная задача массовой проблемы П} \\ & \text{с исходным словом } x \text{ имеет решение,} \\ 0, & \text{в противном случае,} \end{cases}$$

то эту функцию можно рассматривать как характеристическую функцию и массовой проблемы П и соответствующего предиката $\Pi(x)$.

Таким образом, рассмотрение вопроса о существовании решения у задачи массовой проблемы П с исходными данными α и вопроса об истинности высказывания $\Pi(\alpha)$, получающегося из предиката $\Pi(x)$ при $x = \alpha$, сводится к выполнению одного и того же действия – нахождению значения характеристической функции $\chi_\Pi(x)$ при $x = \alpha$ и оценке вычислительной сложности алгоритма, вычисляющего эти значения.

Этот подход позволяет отождествлять массовые проблемы распознавания и соответствующие им предикаты и в каждом конкретном случае использовать ту терминологию, которая более удобна.

В качестве примера рассмотрим одну известную массовую проблему распознавания из теории графов.

Пример 8.3.3. ИЗОМОРФИЗМ ПОДГРАФУ. *Условие*. Заданы два графа $G_1 = (V_1, E_1)$ и $G_2 = (V_2, E_2)$. *Вопрос*. Верно ли, что G_1 содержит подграф изоморфный G_2? Другими словами, существуют ли:
(1) подмножества $V' \subseteq V_1$ и $E' \subseteq E_1$ такие, что $|V'| = |V_2|$, $|E'| = |E_2|$;
(2) взаимно однозначное отображение $f : V_2 \to V'$ такое, что $(u, v) \in E_2$ тогда и только тогда, когда $(f(u), f(v)) \in E'$?

Наконец, отметим, что решение многих массовых проблем оптимизации, т.е. проблем вида "*найти наибольшее (наименьшее) значение k, при котором утверждение $P(k)$ истинно*", которые исключительно часто встречаются на практике, может быть сведено к решению соответствующих проблем распознавания, введя для этого некоторые дополнительные параметры.

Пример 8.3.4. Сформируем, например, задачу распознавания, соответствующую известной задаче о коммивояжере. (см. пример 8.1.1). *Условие*. Задано конечное множество $C = \{c_1, c_2, ..., c_n\}$ "городов", для каждой пары "городов" c_i, $c_j \in C$ "расстояние между ними" $d(_{i,j}) \in N^+$, граница $B \in N^+$. *Вопрос*. Существует ли "маршрут", проходящий по одному разу через все города из C, длина которого не превосходит B? Другими словами, существует ли последовательность $< c_{\pi(1)}, c_{\pi(2)}, ..., c_{\pi(n)} >$ элементов C такая, что

$$\Sigma_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) \; + \; d(c_{\pi(n)}, c_{\pi(1)}) \;\leq\; B \; ?$$

Относительно рассматриваемого соответствия между проблемами оптимизации и проблемами распознавания отметим одно важное обстоятельство. Поскольку значение целевой функции в проблеме оптимизации легко оценивается, поэтому проблема распознавания не может быть сложнее соответствующей проблемы оптимизации. Так, если для проблемы о коммивояжере можно за полиномиальное время найти маршрут минимальной длины, то совершенно ясно, как за полиномиальное время решить соответствующую задачу распознавания. Для этого нужно, найдя маршрут минимальной длины и вычислив его длину, сравнить её с заданной границей B. Следовательно, выводы, которые будут получены для проблем распознавания, вполне могут быть применены и к проблемам оптимизации. Таким образом, мы сводим изучение проблем оптимизации к изучению проблем распознавания. В свою очередь, проблемы распознавания удобны для изучения тем, что они допускают весьма естественное формальное описание с помощью так называемых языков

и их грамматик. Следующий пункт будет посвящён рассмотрению этих понятий.

Формальные языки и грамматики. Как только мы приняли тезис Тьюринга о том, что каждый алгоритм может быть реализован машиной Тьюринга (см. § 2.4), мы тут же приходим к тому, что всякий алгоритмический процесс можно представить в виде конечной последовательности элементарных шагов, записываемых строчками символов. Считая эти символы буквами некоторого алфавита, а строчки – словами, мы получаем, что алгоритм, по существу, является набором правил, по которым в некотором искусственном языке одни слова преобразуются в другие. Изучение таких искусственных или, как их называют, формальных языков имеет исключительно важное значение. Теория формальных языков имеет самое прямое отношение к проектированию алгоритмических языков программирования, а также проектированию трансляторов с этих языков, особенно в части грамматического и семантического анализа текста программы. Без знания основ теории формальных языков невозможно продуктивно работать в области автоматизации проектирования информационно-справочных систем, локальных вычислительных сетей и других инструментов современных информационных технологий.

Перейдём теперь к точным определениям. *Алфавитом* называется всякое конечное непустое множество X, а его элементы называются *буквами.* Конечная последовательность $x_1, x_2, \dots, x_n$, составленная из букв алфавита X, называется *словом* (или *строкой*). *Длина слова* – это количество букв в нём. Например, если $= \{0, 1\}$, то слово 0110111 имеет длину 7. В число слов включают и пустое слово, обозначаемое Λ, считая, что оно не содержит ни одной буквы и его длина равна нулю.

Далее, совокупность всех слов в алфавите X обозначим через X^*. Множество X^* является счётным, и его элементы, т.е. слова в алфавите X, можно расположить в виде бесконечной последовательности в порядке возрастания длины слов. В частности, для двоичного алфавита $X = \{0, 1\}$ это расположение может быть, например, таким: $X^* = \{\Lambda, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$.

Формальным языком (или просто *языком*) над алфавитом X называется всякое подмножество L множества X^*, т.е. $L \subseteq X^*$. Другими словами, язык – это произвольный набор слов в данном алфавите. Тем не менее, абсолютно произвольные наборы слов представляют весьма незначительный интерес. Дело в том, что формальные (искусственные) языки призваны служить своего рода моделями

языка естественного. В свою очередь, естественный язык наделён двумя важнейшими качествами – синтаксисом и семантикой. *Синтаксис* или грамматика естественного языка представляет собой свод правил, регулирующих образование и изменение слов, соединение их в предложения. *Семантика* касается внутреннего смысла и значения слов и предложений, по существу, их связи и связи всего языка с окружающим миром. В языках искусственных (формальных) происходит отказ от осмысленности слов и предложений в её интуитивном понимании, а всё внимание сосредотачивается на синтаксисе – формах слов, способах их порождения и преобразования. Для этого заключается соглашение, которое позволяет классифицировать слова на имеющие смысл и не имеющие смысла с точки зрения их формы, не вникая в их значение.

Ярким примером может служить понятие формулы алгебры высказываний, рассматриваемое в математической логике. Совокупность всех таких формул (или правильно построенных выражений) образует язык алгебры высказываний. Аналогично, в математической логике рассматривается язык логики предикатов.

Для описания бесконечных формальных языков используются так называемые грамматики. Формальные грамматики бывают двух типов: распознающие и порождающие. *Распознающая грамматика* даёт возможность для любого предъявленного набора алфавитных символов установить, является ли он словом данного языка. *Порождающая грамматика* позволяет построить любое слово данного языка, и все построенные ею слова входят в этот язык. Основной интерес для теории программирования представляют порождающие грамматики. Их мы и будем рассматривать далее, называя просто *грамматиками.*

Под *грамматикой* понимается упорядоченная четвёрка $\Gamma =< N, T, P, S >$ следующих объектов: N и T – непустые конечные алфавиты вспомогательных (*нетерминальных*) и основных (*терминальных*) символов соответственно, причём $N \cap T = \emptyset$ (N и T не имеют общих элементов); P – конечное множество выражений вида $\alpha \to \beta$ (*правила* или *продукции*), где α и β – некоторые слова над объединённым алфавитом $N \cap T$; $S \in T$ – выделенный вспомогательный символ, называемый *начальным*, или *источником.* Вспомогательные символы будем обозначать большими латинскими буквами, основные символы – малыми латинскими, слова над алфавитом $N \cap T$ – малыми греческими буквами.

Говорят, что слово δ *непосредственно выводится* в грамматике

Γ из слова γ, если в слове γ есть такое подслово α, а среди правил есть такое правило $\alpha \to \beta$, что, заменив в γ подслово α на подслово β, получим слово δ. При этом, слово α называется *подсловом* слова γ, если $\gamma = \sigma\alpha\rho$, где σ, ρ – некоторые, возможно и пустые, слова в алфавите $N \cap T$. В этом случае будем писать: $\gamma \Longrightarrow \delta$. Если имеется последовательность слов $\gamma = \varepsilon_1, \varepsilon_2, ..., \varepsilon_n = \delta$ такая, что каждое входящее в неё слово, начиная со второго, непосредственно выводится из предыдущего, то эта последовательность называется *выводом* последнего слова δ из первого слова γ, а само слово δ *выводится* из слова (или *порождается* словом) γ. В этом случае будем писать $\gamma \stackrel{*}{\Longrightarrow} \delta$.

Множество всех слов в основном алфавите T, которые можно вывести из начального символа S, называется *языком, порождаемый грамматикой* Γ, и обозначается $L(\Gamma)$.

Рассмотрим примеры порождающих грамматик и соответствующих языков.

Пример 8.3.5. Пусть грамматика $\Gamma_1 = < N_1, T_1, P_1, S_1 >$, где $N_1 = \{S\}$, $T_1 = \{1\}$, $P_1 = \{S \to 1, S \to 1S\}$, $S_1 = S$. Тогда нетрудно понять, что язык $L(\Gamma_1)$ состоит из слов 1, 11, 111, ... Вспоминая введённое при изучении машин Тьюринга обозначение $1^n = 11...1$ (n единиц, записанных подряд), можем записать: $L(\Gamma_1) = \{1^n : n \in N\}$. В самом деле, всякое слово вида 1^n принадлежит $L(\Gamma_1)$, т.е. выводимо из S в грамматике Γ_1:

$$S \Longrightarrow 1S \Longrightarrow 11S \Longrightarrow 111S \Longrightarrow ... \Longrightarrow$$

$$\Longrightarrow 11...1S \; (n-1 \text{ единица}) \Longrightarrow 11...11 \; (n \text{ единиц}) \; .$$

В этом выводе сначала было применено $n - 1$ раз второе правило: $S \to 1S$, а затем на последнем шаге – первое правило $S \to 1$ (согласно нему подслово S заменено на слово 1). Обратно, всякое слово в алфавите T_1, выводимое в грамматике Γ_1 , имеет вид 1^n.

Если последовательность, состоящую из n единиц, считать записью натурального числа n, как это мы делали при изучении машин Тьюринга, то можно сказать, что грамматика Γ_1 порождает натуральный ряд чисел. Заменив второе правило этой грамматики на $S \to 11S$, получим грамматику Γ_2, порождающую нечётные числа.

Пример 8.3.6. Грамматика $\Gamma_3 = < N_3, T_3, P_3, S >$, где $N_3 = \{S\}$, $T_3 = \{0, 1\}$, $P_3 = \{S \to 1S, S \to S0S \to 10\}$, порождает язык $L(\Gamma_3) = \{1^m 0^n : m, n \in N\}$. Напишем, например, вывод слова 1111000 :

$$S \Longrightarrow 1S \Longrightarrow 11S \Longrightarrow 111S \Longrightarrow 111S0 \Longrightarrow 111S00 \Longrightarrow 1111000\ .$$

Как следует изменить правила грамматики Γ_3, чтобы в порождаемом новой грамматикой языке допускались и слова, состоящие только из единиц (только из нулей) ?

Грамматики, порождающие один и тот же язык, называются *эквивалентными*. Приведём пример грамматики, которая порождает такой же язык, как и грамматика Γ_3.

Пример 8.3.7. Грамматика Γ_4 с вспомогательным алфавитом $N_4 = \{S, A\}$, основным алфавитом $T_4 = \{0, 1\}$ и набором правил $P_4 = \{S \to 1S, S \to S0, S \to A, 1A0 \to 10\}$, так же, как и грамматика Γ_3, порождает двоичные слова вида 1^m0^n, т.е. $L(\Gamma_4) = L(\Gamma_3)$. Напишем, например, вывод в грамматике Γ_4 слова 1111000:

$$S \Longrightarrow 1S \Longrightarrow 11S \Longrightarrow 111S \Longrightarrow 1111S \Longrightarrow 1111S0 \Longrightarrow$$

$$\Longrightarrow 1111S00 \Longrightarrow 1111S000 \Longrightarrow 1111A000 \Longrightarrow 1111000\ .$$

В зависимости от того, какой вид имеют грамматические правила (продукции), осуществляется классификация грамматик.

Если все правила грамматики $\Gamma = < N, T, P, S >$ имеют вид $A \to \alpha$, где A – символ из вспомогательного алфавита N, α – слово в алфавите $N \cup T$, то грамматика Γ называется *контекстно свободной*. В противном случае грамматика называется *контекстно зависимой*. Таким образом, все правила контекстно зависимой грамматики имеют вид $\alpha \to \beta$, где $\alpha = \gamma_1 A \gamma_2$, а $\gamma_1, \gamma_2 \in N \cup T$, $A \in N$, $\delta \in (N \cup T) \setminus \{\Lambda\}$. В этом определении строки γ_1 и γ_2 могут рассматриваться как контекст, в котором x может заменяться на δ. Если "контексты" γ_1 и γ_2 в правилах отсутствуют (т.е. являются пустыми строками Λ), то мы приходим к правилам контекстно свободной грамматики. Таким образом, контекстно свободная грамматика есть частный случай контекстно зависимой грамматики.

В рассмотренных примерах грамматики Γ_1,Γ_2, Γ_3 являются контекстно свободными, а Γ_4 – контекстно зависимой. В правилах грамматик Γ_1 – Γ_3 символ S вспомогательного алфавита заменяется некоторым символом независимо от своего окружения (контекста). В грамматике Γ_4 правило $1A0 \to 10$ разрешает заменять символ A (пустым словом) только в контексте: когда слева от A стоит символ 1, а справа 0.

Частным случаем контекстно свободной грамматики является регулярная грамматика. Контекстно свободная грамматика называется *регулярной*, если все её правила имеют вид: $A \to aB$, где $A, B \in N$, $a \in T$, т.е. правая часть каждого правила представляет

собой либо один символ из основного алфавита T (если $B = \Lambda$), либо один символ из T, за которым следует один символ из вспомогательного алфавита N. Регулярной является, например, грамматика Γ_1.

Таким образом, мы приходим к следующей классификации грамматик, называемой *иерархией грамматик по Хомскому.* Самым обширным является класс Г грамматик самого общего вида, на правила которых не накладывается никаких ограничений. Эти грамматики называют также *грамматиками Хомского типа 0.* Его подклассом является класс КЗГ конкретно зависимых грамматик, называемых *грамматиками Хомского типа 1.* Он содержит в себе подкласс КСГ конкретно свободных грамматик, или *грамматик Хомского типа 2.* Наконец, подклассом класса КСГ является класс РГ регулярных грамматик, называемых также *грамматиками Хомского типа 3.* Итак, РГ $\subset$ КСГ $\subset$ КЗГ $\subset$ Г .

Языки, порождаемые каким-либо из этих типов грамматик, имеют соответствующие названия. Так, язык L называется *контекстно свободным*, если существует контекстно свободная грамматика Г, порождающая его: $L = L(\Gamma)$. Если язык может быть порождён регулярной грамматикой, он называется *регулярным.* В примере 8.3.6. было показано, что язык $L_3 = L(\Gamma_3) = \{1^m0^n : m,n \in N\}$ контекстно свободен. Нетрудно показать, что контекстно свободным будет и язык $L'_3 = \{1^n0^n : n \in N\}$, являющийся частью языка L_3. В самом деле, он порождается следующей контекстно свободной грамматикой $\Gamma'_3 = < N_3, T_3, P'_3, S >$, множество правил которой $P'_3 = \{S \to 1S0,\ S \to 10\}$. Можно показать (и это уже не столь очевидно!), что язык L'_3 не может быть порождён никакой регулярной грамматикой, т.е. не является регулярным (см. [27], стр. 273 – 274).

В то же время язык $L_3 = \{1^m0^n : m,n \in N\}$ является регулярным. Его порождает также и следующая регулярная грамматика $\Gamma_5 = < N_5, T_5, P_5, S >$, у которой $N_5 = \{S, A\}$, $T_5 = \{0,1\}$, $P_5 = \{S \to 1S,\ S \to 1A,\ A \to 0A,\ A \to 0\}$. Покажем, например, как в этой грамматике получается вывод слова 1111000:

$$S \Longrightarrow 1S \Longrightarrow 11S \Longrightarrow 111S \Longrightarrow 1111A \Longrightarrow 11110A \Longrightarrow$$

$$\Longrightarrow 111100A \Longrightarrow 1111000\,.$$

В заключение приведём ещё один пример.

Пример 8.3.8. Построить грамматику для языка $L_6 = \{a^nb^nc^n : n \in N\}$ и определить тип этого языка.

Покажем, что порождающей грамматикой рассматриваемого языка является $\Gamma_6 = < N_6, T_6, P_6, S_6 >$, где $N_6 = \{S, A, B\}$,

$T_6 = \{a, b, c\}$, $P_6 = \{(1) : S \to aSAB,\ (2) : S \to aAB,\ (3) : aA \to ab,\ (4) : bA \to bb,\ (5) : bB \to bc,\ (6) : BA \to AB,\ (7) : cB \to cc\}$.

Вначале покажем, как с помощью правил данной грамматики можно из начального символа S получить слово $a^n b^n c^n$ для любого $n \in N$:

$$S \overset{(1)}{\Longrightarrow} a^{n-1}S(AB)^{n-1} \overset{(2)}{\Longrightarrow} a^n(AB)^n = a^n ABAB...AB \overset{(6)}{\Longrightarrow}$$

$$a^n A^n B^n = a^{n-1}aAA...AB^n \overset{(3)}{\Longrightarrow} a^{n-1}abA...AB^n \overset{(4)}{\Longrightarrow}$$

$$a^n b^n B^n = a^n b^{n-1}bBB^{n-1} \overset{(5)}{\Longrightarrow} a^n b^{n-1}bcBB^{n-2} \overset{(7)}{\Longrightarrow}$$

$$a^n b^n ccBB^{n-3} \overset{(7)}{\Longrightarrow} a^n b^n c^n\ .$$

Следовательно, $L_6 = L(\Gamma_6)$. Чтобы доказать обратное включение, нужно показать, что никакие другие строки не могут быть порождены грамматикой L_6. В самом деле, хотя возможны изменения в порядке применения правил (1), (2) и (6), любое предложение должно выводиться посредством формы $a^n AB\alpha$, где α состоит из $n-1$ символов A и $n-1$ символов B. Для того, чтобы получить слово в алфавите T_6, мы должны использовать правила (4), (5) и (7). При этом правило (7) может преобразовывать B в c только в контексте cB, а правило (5) может делать то же преобразование только в контексте bB. Таким образом, терминальный символ c не может появиться в строке раньше терминального символа b. Аналогично, для замены A на b при помощи правил (4) и (3) требуются контексты bA и aA соответственно. Следовательно, терминальный символ b не может появиться в строке раньше терминального символа a.

Таким образом, рассматриваемую грамматику Γ_6 и порождаемый ею язык $L(\Gamma_6)$ можно рассматривать как контекстно - зависимый.

Проблемы распознавания и формальные языки. Мы уже отмечали, что массовые проблемы описываются (кодируются) с помощью схем кодирования, которые каждую единичную задачу массовой проблемы описывают подходящим словом в некотором фиксированном алфавите. В частности, если рассматривается массовая проблема распознавания П и для её кодирования используется схема кодирования e с алфавитом E, то каждой единичной задаче из П соответствует некоторое слово из E^*. При этом, некоторые слова из E^* будут кодировать единичные задачи, имеющие положительный ответ (ДА) на вопрос, некоторые слова будут кодировать единичные задачи, имеющие отрицательный ответ (НЕТ) на вопрос, а

оставшиеся слова из E^* вообще не будут служить кодами никаких единичных задач из П.

Первый класс слов (т.е. подмножество множества E^*) как раз и есть тот формальный язык, который ставится в соответствие массовой проблеме распознавания П при кодировании e и обозначается L[П, e]. Таким образом,

$$L[\Pi, e] = \{x \in E^* : \begin{array}{ll} E - & \text{алфавит схемы кодирования } e; \\ x - & \text{код единичной задачи при} \\ & \text{схеме кодирования } e \text{ с ответом ДА}\}. \end{array}$$

В силу этого соответствия, будем говорить, что некоторый результат имеет место для проблемы П распознавания, если при некоторой схеме кодирования e этот результат имеет место для формального языка L[П, e]. При этом, говорят о так называемых "разумных схемах кодирования" т.е. таких схемах, которые, во-первых, достаточно "сжаты" т.е. сохраняют при кодировании естественную краткость формулировки единичной задачи, и, во-вторых, допускают декодирование, т.е. по данной компоненте условия задачи можно было бы указать алгоритм с полиномиальным временем работы, который из любого заданного кода единичной задачи позволял бы извлечь описание этой компоненты.

Детерминированные алгоритмы и класс P. Термином "*детерминированные алгоритмы*" мы будем называть обычные алгоритмы, с которыми мы имели дело до сих пор, с тем, чтобы подчеркнуть их разницу с алгоритмами недетерминированными, которые будут определены и рассмотрены в следующем пункте. В силу тезиса Тьюринга, любой (детерминированный) алгоритм можно проинтерпретировать в виде машины Тьюринга, которую также в этом случае будем называть *детерминированной.*

Применительно к задачам распознавания слегка модифицируем понятие детерминированной машины Тьюринга (ДМТ). Будем считать, что в алфавите Q её внутренних состояний вместо одного заключительного состояния q_0 имеется два заключительных состояния q_Y и q_N (от слов Yes – ДА, No – НЕТ).

Будем говорить, что ДМТ M, имеющая входной алфавит A, *принимает* входное слово $x \in A^*$, если будучи применённой ко входу x, она останавливается в состоянии q_Y. Совокупность всех таких слов в алфавите A, которые предусматривают ответ ДА (остановка в состоянии q_Y), образует язык, который обозначим L_M и будем называть языком, *распознаваемым* машиной M: $L_M = \{x \in A^* : M \text{ принимает } x\}$.

Соответствие между "распознаванием" языков и "решением" проблем следующем образом. Будем говорить, что ДМТ M ***решает проблему распознавания*** П со схемой кодирования e, если M останавливается на всех словах, составленных из букв входного алфавита A, и $L_M = L[\Pi, e]$.

Пример 8.3.9. Рассмотрим ДМТ с внешним алфавитом $A = \{a, b\}$, алфавитом внутренних состояний $Q = \{q_1, q_2, q_3, q_4, q_Y, q_N\}$ и программой:

	a	b	Λ
q_1	$q_1 a$П	$q_1 b$П	$q_2 \Lambda$Л
q_2	$q_3 \Lambda$Л	$q_4 \Lambda$Л	$q_N \Lambda$Л
q_3	$q_Y \Lambda$Л	$q_N \Lambda$Л	$q_N \Lambda$Л
q_4	$q_N \Lambda$Л	$q_N \Lambda$Л	$q_N \Lambda$Л

(Λ – символ пустой ячейки).

Посмотрим, какое слово выдаст и в каком состоянии остановится машина, начав работать со слова $q_1 babaa$:

$$q_1 babaa \Longrightarrow bq_1 abaa \Longrightarrow baq_1 baa \Longrightarrow babq_1 aa \Longrightarrow babaq_1 a \Longrightarrow$$

$$\Longrightarrow babaaq_1 \Lambda \Longrightarrow babaq_2 a\Lambda \Longrightarrow babq_3 a\Lambda\Lambda \Longrightarrow baq_Y b\Lambda\Lambda\Lambda \ .$$

Проверьте самостоятельно, что $q_1 baba \Longrightarrow bq_N a\Lambda\Lambda\Lambda$; $q_1 bab \Longrightarrow q_N b\Lambda\Lambda\Lambda$; $q_1 babab \Longrightarrow b_1 aq_N \Lambda\Lambda\Lambda$.

Докажите, что данная машина Тьюринга распознаёт язык, состоящий из тех и только тех слов алфавита $A = \{a, b\}$, которые оканчиваются двумя буквами a. □

Здесь мы можем детализировать понятие вычислительной сложности алгоритма, представляемого детерминированной машиной Тьюринга (ДМТ) M. Мерой этой сложности выступает функция $T_M : N^+ \to N^+$ такая, что $T_M(n)$ есть максимальное число шагов, требуемое для вычисления на машине M результата для всевозможных входных слов $x \in A^*$ длины n. Если эта функция ограничена некоторым полиномом $p(n)$, т.е. $T_M(n) \leq p(n)$ для всех $n \in N^+$, то детерминированная машина Тьюринга M называется ***полиномиальной.***

Говорят, что формальный язык L принадлежит ***классу*** $\mathbf{P}$, если он распознаётся некоторой полиномиальной детерминированной машиной Тьюринга, т.е.

$$\mathbf{P} = \{L : (\exists M)(M - \text{полиномиальная ДМТ и } L_M = L)\} \ .$$

Говорят, что проблема распознавания П принадлежит ***классу* P** при схеме кодирования e, если язык этой проблемы при этом кодировании принадлежит классу **P**, т.е. $L[\Pi, e] \in \mathbf{P}$; или существует полиномиальная ДМТ, которая решает проблему П при кодировании e.

Таким образом, снова применив тезис Тьюринга, можно сказать, что класс **P** – это класс таких массовых проблем распознавания, каждая из которых может быть решена полиномиальным алгоритмом. (Название класса происходит от Polynomial - полиномиальный).

Отметим, что сложностные классы **P** и **NP** алгоритмических проблем были независимо введены в работах Кобхема (Alan Kobhem, 1964) и Эдмондса (Jack Edmondson, 1965), которые стремились к тому, чтобы дать теоретический ответ на вопрос, какие алгоритмические проблемы могут быть эффективно решены на компьютере, т.е. решены за реальное (разумное) время.

Недетерминированные алгоритмы и класс NP. В этом пункте вводится второй важный класс языков (а вместе с ними – и соответственных проблем распознавания свойств) – класс **NP**, который будет включать класс **P**.

Предварительно поясним смысл одного важного понятия, лежащего в основе определения класса **NP** – понятия *недетерминированного алгоритма*. Такой алгоритм состоит из двух различных стадий – стадии угадывания и стадии проверки. На первой стадии для заданной единичной задачи $Z \in \Pi$ происходит просто "угадывание" некоторой структуры s (Угадывающее устройство называют ***оракулом***). Затем Z и s (разумеется, в закодированном виде) вместе подаются на вход обычного детерминированного алгоритма, который осуществляет стадию проверки. Работа этого алгоритма либо заканчивается ответом ДА, либо заканчивается ответом НЕТ, либо продолжается вечно без остановки (последние две возможности можно не различать). Таким образом, отличие недетерминированного алгоритма от детерминированного состоит в наличии оракула.

Будем считать, что недетерминированный алгоритм "***решает***" проблему распознавания $\Pi = (D_\Pi, Y_\Pi)$, если для любой единичной задачи $Z \in D_\Pi$ выполнены следующие два свойства:

1) если $Z \in Y_\Pi$, то существует такая структура s, угадывание которой для входа Z приведёт к тому, что стадия проверки, начиная работу на входе (Z, s), заканчивается ответом ДА;

2) если $Z \notin Y_\Pi$, то *не* существует такой структуры s, угадывание которой для входа Z обеспечило бы окончание стадии проверки

на входе (Z, s) ответом ДА.

Далее, говорят, что недетерминированный алгоритм, решающий проблему распознавания П, работает в течение "*полиномиального времени*", если найдётся такой полином p, что для любой единичной задачи $Z \notin Y_\Pi$ найдётся некоторая догадка s, приводящая на стадии детерминированной проверки при входе (Z, s) к ответу ДА за время (число шагов) $p(r)$, где r – размер единичной задачи Z. Отсюда, в частности, следует, что и размер угадываемой структуры s будет обязательно ограничен полиномом от r, так как на проверку догадки s может быть затрачено не более чем полиномиальное время.

Образно выражаясь, можно сказать, что если детерминированный алгоритм в каждый момент времени может выполнять какое-либо одно действие, то недетерминированный алгоритм из любого данного состояния может переходить в более одного допустимого следующего состояния, и в каждый данный момент времени может выполнять более одного действия. Можно также сказать, что недетерминированный алгоритм производит вычисления до тех пор, пока не доходит до места, в котором должен быть сделан выбор из нескольких альтернатив. Детерминированный алгоритм исследовал бы в этом случае одну альтернативу, а потом бы возвращался для исследования другой альтернативы. Недетерминированный алгоритм может исследовать все возможности одновременно, как бы "копируя" самого себя для каждой альтернативы. Все копии работают независимо, не сообщаясь друг с другом каким-либо образом. Эти копии, конечно, могут создавать дальнейшие копии и т.д. Если копия обнаруживает, что она сделала неправильный (или безрезультатный) выбор, она прекращает выполняться. Если копия находит решение, она объявляет о своем успехе, и все копии прекращают работать.

Итак, недетерминированный алгоритм характеризуется двухшаговым подходом к решению задачи. На первом шаге имеется недетерминированный алгоритм, генерирующий возможное решение такой задачи – что-то вроде попытки угадать решение. Иногда такая попытка оказывается успешной, и мы получаем оптимальный или близкий к оптимальному ответ; иногда – безуспешной: ответ далёк от оптимального. На втором шаге проверяется, действительно ли ответ, полученный на первом шаге, является решением исходной задачи. Если каждый из этих шагов по отдельности требует полиномиального времени, то алгоритм называется недетерминированным полиномиальным. Но вопрос в том, что мы не знаем, сколько раз

нам придётся повторить оба эти шага, чтобы получить искомое решение. Хотя оба шага и полиномиальны, число обращений к ним может оказаться экспоненциальным или даже факториальным.

Таким образом, понятие недетерминированного алгоритма есть попытка формализовать и сделать объектом строгого математического изучения, так называемые эвристические процедуры вычисления, т.е. такие процедуры, когда вычисления осуществляются не "тупо", не в "в лоб", не путём перебора всех возможных случаев, а с использованием каких-то дополнительных соображений, позволяющих ограничиться вычислениями лишь в таких случаях, в которых ожидаемый результат будет наиболее вероятен. Это достигается в результате отказа от одного из основополагающих свойств обычного алгоритма – его детерминированности, когда на каждом шаге алгоритм выполняет однозначно определённые действия, не допускающие никаких разночтений. Таким образом, недетерминированные алгоритмы моделируют вычислительные процедуры с некоторой "свободой выбора" шагов, обладая теоретической способностью параллельно выполнять неограниченное количество независимых вычислений.

Теперь мы можем определить *класс* **NP**. Это – класс всех таких проблем распознавания, которые при разумном кодировании могут быть решены недетерминированными алгоритмами за полиномиальное время. (Название **NP** происходит от N - nondeterministic и P - polynomial).

Пример 8.3.10. Вспомним известную проблему КОММИВОЯЖЕР, описание которой дано в начале параграфа 8.1, а в начале настоящего параграфа она была переформулирована как задача распознавания (см. пример 8.3.4.). В условии даны: множество городов, расстояния между любыми двумя из них и граница B; спрашивается, существует ли проходящий через все города маршрут длины, не превосходящий B.

Эта задача имеет многочисленные модификации и применения в самых разнообразных жизненных сферах. Её можно применять, например, для определения порядка эффективного сбора мусора из баков на улицах города или выбора кратчайшего пути распространения информации по всем узлам компьютерной сети. Если имеется 8 городов, то их можно упорядочить 40320 ($= 8!$) всевозможными способами, а для десяти городов это число возрастает уже до 3628800. Поиск кратчайшего пути (или пути, длина которого не превосходит B) требует перебора всех этих возможностей.

Предположим, что у нас есть алгоритм, способный подсчитать длину маршрута через 15 городов в указанном порядке. Если за секунду такой алгоритм способен пропустить через себя 100 вариантов, то ему потребуется больше четырёх веков, чтобы исследовать все возможности и найти кратчайший путь. Даже если в нашем распоряжении имеется 400 компьютеров, все равно у них уйдёт на это год, а ведь мы имеем дело лишь с 15 городами. Для 20 городов миллиард компьютеров должен будет работать параллельно в течение девяти месяцев, чтобы найти кратчайший путь. Ясно, что быстрее и дешевле путешествовать хоть как-нибудь, чем ждать, пока компьютеры выдадут оптимальное решение.

Можно ли найти кратчайший маршрут, не просматривая все мыслимые пути? До сих пор никому не удалось придумать алгоритм, требующий полиномиального времени, в частности, алгоритм, который не занимается, по существу, перебором (просмотром) всех возможных маршрутов. Итак, детерминированный алгоритм, сравнивающий все возможные способы упорядочивания городов, работает чересчур долго. Чтобы показать, что эта задача относится к классу **NP**, нам необходимо понять, как её можно решить посредством описанной выше двухшаговой процедуры.

Предположим, что для некоторой единичной задачи кем-то получен ответ ДА. Тогда, рассмотрев предъявляемый маршрут нетрудно проверить, действительно ли он удовлетворяет условию задачи: для этого надо вычислить его длину и сравнить её с границей B. Более того, ясно, что эту "процедуру проверки" можно представить в виде алгоритма, временная сложность которого ограничена полиномом от размера r единичной задачи. В качестве стадии угадывания можно взять процедуру простого выбора произвольной последовательности городов. Очевидно, что для любой единичной задачи Z найдётся такая догадка s (последовательность городов), что результатом работы стадии проверки при входе (Z, s) будет ДА в том и только в том случае, когда для единичной задачи Z существует маршрут искомой длины.

Проще говоря, для решения задачи о коммивояжере на первом шаге случайным образом генерируется некоторое упорядочивание городов. Поскольку это недетерминированный процесс, каждый раз будет получаться новый порядок. Очевидно, что процесс генерации можно реализовать за полиномиальное время: мы можем хранить список городов, генерировать случайный номер, выбирать из списка город с этим именем и удалять его из списка, чтобы он не появился второй раз. Такая процедура выполняется за $O(N)$ операций,

где N – число городов. На втором шаге происходит подсчёт длины маршрута по городам в указанном порядке. Для этого нам нужно просто просуммировать расстояния между последовательными парами городов в списке, что также требует $O(N)$ операций. Оба шага полиномиальны, поэтому задача о коммивояжере лежит в классе **NP**. Времяёмкой делает её именно необходимое число итераций этой процедуры.

Пример 8.3.11. Аналогичными рассуждениями показывается, что проблема ИЗОМОРФИЗМ ПОДГРАФУ также принадлежит классу **NP** (см. пример 8.3.3).

Таким образом, класс **NP** проблем недетерминированной полиномиальной сложности состоит из проблем, которые *практически* неразрешимы, т.е. для них не известны алгоритмы, способные решить их за разумное время. Отметим только, что сложность всех известных детерминированных алгоритмов, решающих проблемы из класса **NP**, либо экспоненциальна, либо факториальна. Сложность некоторых из них равна 2^N, где N – количество входных данных. В этом случае при добавлении к списку входных данных одного элемента время работы алгоритма удваивается. Если для решения такой задачи на входе из десяти элементов алгоритму требовалось 1024 операции, то на входе из 11 элементов число операций составит уже 2048. Это значительное возрастание времени при небольшом удлинении входа.

Введённое определение класса **NP** как класса специфических проблем распознавания нельзя считать формально строгим. Оно становится таким, если класс **NP** (подобно тому, как это делалось в предыдущем пункте для класса **P**) определять как класс формальных языков, распознаваемых недетерминированными машинами Тьюринга за полиномиальное время.

Сначала, как обычно, выбирается одна из формализаций понятия алгоритма – машина Тьюринга. Согласно тезису Тьюринга любой недетерминированный алгоритм можно проинтерпретировать в виде недетерминированной одноленточной машины Тьюринга (НДМТ).

НДМТ также обладает алфавитом символов на ленте $A = \{a_1, a_2, \dots, a_m\}$, алфавитом внутренних состояний $Q = \{q_1, q_2, \dots, q_n,$

$q_Y, q_N\}$. Но в отличие от ДМТ, её команды имеют вид списков:

$$q_i a_j \longrightarrow \begin{cases} q_{i_1} a_{j_1} X_{k_1} \\ q_{i_2} a_{j_2} X_{k_2} \\ \dots\dots\dots\dots\dots \\ q_{i_l} a_{j_l} X_{k_1} \end{cases},$$

где l – максимальное число вариантов выполнения действия. Перед выполнением очередного шага по некоторой команде из одной машины возникает l машин. Записи на их лентах одинаковы – такие, какая была у них "предка"; но с этого момента их пути расходятся: каждая машина выполняет свой вариант действия из списка. И так происходит на каждом шаге, так что число вариантов продвижения к результату резко множится; действия машины происходят не по линейной цели, как у ДМТ, а одновременно по многим направлениям, образуя дерево; НДМТ решает задачу одновременно многими способами. В результате по одной из цепочек действия заканчиваются, и устройство управления оказывается в одном из состояний остановки: q_Y (задача решена положительно; ДА), или q_N (решение пока не найдено). В первом случае все остальные машины, размножившиеся к этому моменту, также заканчивают работу, и в этом случае говорят, что вычисление было ***принимающим*** с данным входом x. Отметим, что НДМТ M может иметь бесконечное число возможных вычислений при данном входе x. Во втором случае все остальные машины продолжают работу. Если же по всем цепочкам происходит остановка в состоянии q_N или остановка не происходит, то это означает, что задача не имеет решения и в этом случае говорят, что вычисление ***непринимающее*** с данным входом x.

Будем говорить, что НДМТ M принимает x, если, по меньшей мере, одно из её вычислений его входом x является принимающим. Совокупность всех слов x, которые данная НДМТ M принимает, образуют язык, называемый языком, ***распознаваемым*** этой машиной:

$$L_M = \{x \in A^* : M \text{ принимает } x\}.$$

Время, требующееся НДМТ M для того, чтобы принять слово $x \in L_M$ – это минимальное число шагов выполненных машиной M до достижения заключительного состояния q_Y, где минимум берётся по всем принимающим вычислениям машины M со входом x.

Временная сложность НДМТ M – это функция $T_M : N^+ \to N^+$, определяемая следующим образом:

$$T_M(n) = \max(\{1\} \cup \{m : (\exists x \in L_M)(|x| = n \text{ и время принятия } x \text{ машиной } M \text{ равно } m\}).$$

Заметим, что эта функция зависит только от числа шагов, выполняемых машиной в принимающих вычислениях. Если же нет ни одного входа длины n, принимаемого машиной M, то полагаем $T(n) = 1$.

НДМТ называется *НДМТ с полиномиальным временем работы*, если найдётся полином p такой, что $T_M(n) < p(n)$ при всех $n > 1$.

Теперь можем формально определить *класс* **NP** как класс языков, распознаваемых недетерминированными машинами Тьюринга с полиномиальным временем работы:

$$\mathbf{NP} = \{L: \text{ существует НДМТ } M \text{ с полиномиальным временем работы и } L_M = L\}.$$

В силу соответствия между формальными языками и проблемами распознавания будем говорить, что проблема распознавания П принадлежит классу **NP** при схеме кодирования e, если язык этой проблемы при этом кодировании принадлежит классу **NP**, т.е. $L[\Pi, e] \in \mathbf{NP}$.

Взаимоотношения между классами P и NP. Поскольку понятие ДМТ представляет собой частный случай понятия НДМТ с числом разветвлений $l = 1$, поэтому все проблемы класса **P** являются также и проблемами класса **NP**, т.е. $\boxed{\mathbf{P} \subseteq \mathbf{NP}}$.

Этот факт также вытекает и из общих соображений, поскольку всякий детерминированный алгоритм можно рассматривать как недетерминированный алгоритм, у которого отсутствует стадия угадывания, а стадия проверки представлена данным детерминированным алгоритмом.

После этого естественно возникает вопрос, совпадают ли классы **P** и **NP**, т.е. $\mathbf{P} = \mathbf{NP}$, или класс **P** является собственным подклассом в **NP**, т.е. $\mathbf{P} \neq \mathbf{NP}$. Ответ на этот вопрос к настоящему времени не известен. Другими словами, не известна ни одна проблема из класса **NP**, которая не входила бы в класс **P**, т.е. которая решалась бы недетерминированным полиномиальным алгоритмом, но не может быть решена ни одним детерминированным полиномиальным алгоритмом. Эта проблема считается важнейшей в науке о вычислениях и выделена в ряд важнейших общематематических проблем XXI века. Её значимость обусловлена тем, что огромное количество "естественных" массовых проблем входят в класс **NP** в то время, как эффективно решаемые проблемы образуют класс **P**.

В пользу утверждения $\mathbf{P} \neq \mathbf{NP}$ говорят многие доводы. В частности, полиномиальные недетерминированные алгоритмы опре-

делённо оказываются более мощными, чем полиномиальные детерминированные алгоритмы, и не известны общие методы их превращения в детерминированные полиномиальные алгоритмы. К настоящему времени известно, что всякая проблема из класса **NP** может быть решена детерминированным алгоритмом, но с экспоненциальной сложностью $O(2^{p(n)})$, где p – некоторый полином. Это говорит о способности недетерминированного алгоритма проверять за полиномиальное время экспоненциальное число возможностей, что наталкивает на мысль о значительно большей мощности таких алгоритмов по сравнению с детерминированными полиномиальными алгоритмами. Возможно, именно поэтому для многих проблем класса **NP**, таких как КОММИВОЯЖЕР, ИЗОМОРФИЗМ ПОДГРАФУ и большого числа других задач до сих пор не найдено детерминированного полиномиального алгоритма, несмотря на упорные усилия многих известных математиков. Так что вопрос, выполняется ли равенство $\mathbf{P} = \mathbf{NP}$, до сих пор остаётся предметом исследований по всему миру.

С учётом накопленного опыта и при существующем в настоящее время уровне знаний, по-видимому, более разумно работать при ощущении, что $\mathbf{P} \neq \mathbf{NP}$. Хотя ещё раз подчеркнём, что прямого доказательства этой гипотезы пока не существует.

Полиномиальная сводимость формальных языков массовых проблем распознавания. В начале настоящего параграфа 8.3 уже были высказаны некоторые неформальные соображения, касающиеся алгоритмической сводимости одних массовых проблем к другим. С учётом разработанной теории формальных языков в их связи с проблемами распознавания мы можем теперь придать этим соображениям более формальный характер, после чего они станут основой для определения ещё одного важного класса проблем распознавания – класса NP-полных проблем.

Определение 8.3.12. Будем говорить, что язык $L_1 \in X_1^*$ *полиномиально сводится* к языку $L_2 \in X_2^*$, если существует функция (отображение) $f: X_1^* \to X_2^*$, вычисляемая за полиномиальное время некоторой детерминированной машиной Тьюринга (ДМТ), и такая, что

$$(\forall x \in X_1^*)\ (x \in L_1 \longleftrightarrow f(x) \in L_2)\ .$$

Будем писать в этом случае $L_1 \Longrightarrow L_2$, и говорить "L_1 *сводится* к L_2" (опуская слово "полиномиально").

Лемма 8.3.13. *Если* $L_1 \Longrightarrow L_2$, *то из* $L_2 \in \mathbf{P}$ *следует, что* $L_1 \in \mathbf{P}$ (*или, что эквивалентно: из* $L_1 \notin \mathbf{P}$ *следует, что* $L_2 \notin \mathbf{P}$).

Доказательство. Пусть X_1 и X_2 – алфавиты языков L_1 и L_2 соответственно, функция $f : X_1^* \to X_2^*$ осуществляет полиномиальную сводимость L_1 к L_2, M_f – ДМТ, полиномиально вычисляющая f, и M_2 – полиномиальная ДМТ, распознающая язык L_2. Тогда полиномиальная ДМТ, распознающая язык L_1, может быть композицией данных машин Тьюринга: $M_1 = M_2 \circ M_f$. Ко входу $x \in X_1^*$ вначале применяется машина M_f, на выходе которой выдаётся $f(x) \in X_2^*$. Затем к $f(x)$ применяется машина M_2, выясняющая, верно ли, что $f(x) \in L_2$. Поскольку f обладает тем свойством, что $x \in L_1$ тогда и только тогда, когда $f(x) \in L_2$, то это описание действительно даёт машину M_1, распознающую язык L_1. Поскольку время работы каждой машины M_f и M_2 ограничено полиномом, то ясно, что и время работы их композиции M_1 также ограничено некоторым полиномом от длины $|x|$ исходного входного слова x. □

Теперь понятие полиномиальной сводимости может быть перенесено на проблемы распознавания.

Пусть Π_1 и Π_2 – проблемы распознавания, e_1 и e_2 – их схемы кодирования с алфавитами E_1 и E_2 соответственно, $L(\Pi_1, e_1)$ и $L(\Pi_2, e_2)$ – языки, соответствующие этим проблемам. Будем говорить, что Π_1 ***(полиномиально) сводится*** к Π_2 и писать $\Pi_1 \Longrightarrow \Pi_2$, если имеет место (полиномиальная) сводимость языка $L(\Pi_1, e_1)$ к языку $L(\Pi_2, e_2)$. Другими словами, полиномиальная сводимость проблемы $\Pi_1 = (D_1, Y_1)$ к проблеме $\Pi_2 = (D_2, Y_2)$ означает наличие функции (отображения) $f : D_1 \longrightarrow D_2$, вычисляемой алгоритмом полиномиальной сложности и удовлетворяющей условию:

$$(\forall z \in D_1)(z \in Y_1 \quad \longleftrightarrow \quad f(z) \in Y_2) \ .$$

Тогда доказанная лемма 8.3.13 позволяет интерпретировать сводимость проблем $\Pi_1 \Longrightarrow \Pi_2$ как утверждение, что Π_2 "не проще" проблемы Π_1, а проблема Π_1 "не сложнее" проблемы Π_2; в частности, если Π_2 имеет полиномиальную сложность, то такую же сложность имеет и Π_1.

Пример 8.3.14. Рассмотрим одну проблему из теории графов и покажем её полиномиальную сводимость к проблеме КОММИВОЯЖЕР (КВ).

Пусть $G = < V, E >$ – граф с множеством вершин V и множеством рёбер E. ***Простым циклом*** в G называется такая последовательность $< v_1, v_2, ..., v_k >$ различных вершин из V, что $(v_i, v_{i+1}) \in E$, $1 \leq i < k$, и $(v_k, v_1) \in E$. ***Гамильтоновым циклом*** в G называется такой простой цикл, который содержит все вершины графа G, т.е.

при $k = |V|$. Проблема **ГАМИЛЬТОНОВ ЦИКЛ (ГЦ)** состоит в том, чтобы для произвольного графа $G =< V, E >$ выяснить, верно ли, что G содержит гамильтонов цикл.

Покажем, что имеет место сводимость ГЦ $\Longrightarrow$ КВ. Для этого нужно указать функцию f, которая отображает каждую единичную задачу из ГЦ в соответствующую единичную задачу из КВ и удовлетворяет двум условиям. Функция f определяется следующим образом. Пусть $G =< V, E >$, $|V| = n$ означает фиксированную единичную задачу из массовой проблемы ГЦ. Соответствующая задача из КВ строится так: множество C городов совпадает с V; для любых двух городов v_i, $v_j \in C$ расстояние $d(v_i, v_j)$ между ними полагаем равным 1, если $(v_i, v_j) \in E$, и равным 2 в противном случае. Граница B для длины искомого маршрута берётся равной n.

Ясно, что f может быть вычислена за полиномиальное время, так как для вычисления каждого из $n(n-1)/2$ расстояний $d(v_i, v_j)$ необходимо лишь выяснить, принадлежит ли пара (v_i, v_j) множеству E или нет (т.е. есть ли в графе ребро с вершинами v_i и v_j). Второе требование также выполняется, поскольку ясно, что G содержит гамильтонов цикл тогда и только тогда, когда в $f(G) = V$ имеется проходящий через все города маршрут длины, не превосходящей $B = n$.

Таким образом, мы доказали, что ГЦ $\Longrightarrow$ КВ. Это, в частности, означает, что проблема ГЦ "не сложнее" проблемы КВ, а КВ "не проще" ГЦ. Образно говоря, если КВ простая, то и ГЦ простая, а если ГЦ сложная, то и КВ сложная.

Вернёмся теперь снова к терминологии формальных языков и установим ещё одно свойство их полиномиальной сводимости, называемое транзитивностью.

Лемма 8.3.15. *Если* $L_1 \Longrightarrow L_2$ *и* $L_2 \Longrightarrow L_3$, *то* $L_1 \Longrightarrow L_3$.

Доказательство. Пусть X_1, X_2, X_3 – алфавиты языков L_1, L_2, L_3 соответственно, функция $f_1 : X_1^* \to X_2^*$ полиномиально сводит L_1 к L_2, а $f_2 : X_2^* \to X_3^*$ – полиномиально сводит L2 к L3. Тогда функция $f : X_1^* \to X_3^*$, являющаяся суперпозицией функций f_1 и f_2, т.е. которая для всех $x \in X_1^*$ определяется соотношением $f(x) = f_2(f_1(x))$, реализует искомую полиномиальную сводимость языка L_1 к языку L_2. □

Из этой леммы вытекает также аналогичное свойство транзитивности для отношения сводимости массовых проблем (на неформальном уровне мы говорили о нём в конце первого пункта настоящего параграфа): если $\Pi_1 \Longrightarrow \Pi_2$ и $\Pi_2 \Longrightarrow \Pi_3$, то $\Pi_1 \Longrightarrow \Pi_3$.

Свойство транзитивности отношения сводимости говорит о том, что это отношение является отношением квазипорядка (свойство рефлексивности $L \Longrightarrow L$, или соответственно $\Pi \Longrightarrow \Pi$, очевидно). Последнее позволяет ввести на множестве всех языков (или проблем распознавания) отношение эквивалентности посредством следующего определения.

Два языка L_1 и L_2 (соответственно, проблемы распознавания Π_1 и Π_2) называются ***полиномиально эквивалентными***, если каждый из них сводится к другому, т.е. $L_1 \Longrightarrow L_2$ и $L_2 \Longrightarrow L_1$ (соответственно, $\Pi_1 \Longrightarrow \Pi_2$ и $\Pi_2 \Longrightarrow \Pi_1$). Возникающие при этом классы эквивалентности упорядочены отношением сводимости $\Longrightarrow$. Тогда первая из двух доказанных лемм означает, с одной стороны, что класс **P** языков (проблем) является одним из классов данного отношения эквивалентности, а, с другой стороны, что этот класс является "наименьшим" (или, по крайней мере, "минимальным") относительно данного частичного порядка. С вычислительной точки зрения его можно рассматривать как класс "самых лёгких" языков (задач распознавания).

Класс NP-полных языков и NP-полных массовых проблем распознавания. Здесь мы определим ещё один класс введённого в конце предыдущего пункта отношения эквивалентности, связанного с отношением сводимости. Этот класс, в отличие от класса **P** будет содержать, напротив, "самые трудные" языки (соответственно, задачи распознавания) из класса **NP**.

ОПРЕДЕЛЕНИЕ 8.3.16. Язык L называется *NP-полным*, если $L \in$ **NP** и любой другой язык из класса **NP** сводится к L, т.е. если $L' \in \mathbf{NP}$, то $L' \Longrightarrow L$. Соответственно, проблема распознавания Π называется *NP-полной*, если она принадлежит классу **NP** и всякая проблема из **NP** к ней сводится.

Понятие NP-полноты было введено независимо Куком (Stephen Cook, 1971) и Левиным (1973).

Таким образом, лемма 8.3.13 из предыдущего пункта позволяет рассматривать NP-полные проблемы из **NP** как "самые трудные" проблемы из этого класса: если хотя бы одна NP-полная проблема будет решена детерминированным алгоритмом за полиномиальное время, то и все проблемы из **NP** смогут быть решены детерминированными полиномиальными алгоритмами. Это означало бы, что $\mathbf{NP} \subseteq \mathbf{P}$ и значит, $\mathbf{P} = \mathbf{NP}$. Напротив, если хотя бы одна проблема из **NP** не может быть решена детерминированным алгоритмом за полиномиальное время (труднорешаема), то и все NP-полные проб-

лемы не могут быть решены детерминированными алгоритмами за полиномиальное время (т.е. являются труднорешаемыми). Следовательно, любая NP-полная проблема П не может оказаться в классе **P**, если только $\mathbf{P} \neq \mathbf{NP}$, т.е. если $\mathbf{P} \neq \mathbf{NP}$, то $\Pi \in \mathbf{NP} \setminus \mathbf{P}$. Более тщательные рассмотрения позволяют установить, что в предположении $\mathbf{P} \neq \mathbf{NP}$ можно показать, что в классе $\mathbf{NP} \setminus \mathbf{P}$ должны существовать не только NP-полные проблемы, но и проблемы, не являющимися NP-полными, но неразрешимые за полиномиальное время.

Многие верят, что полиномиальные детерминированные алгоритмы решения NP-полных проблем не существуют. Но как можно было бы доказать, что не существует полиномиального алгоритма решения той или иной задачи. Для этого нужно попробовать оценить снизу минимальный объём работы, необходимый для её решения. При этом, получающаяся функция должна превышать по скорости роста любой многочлен.

Прежде, чем привести хотя бы один пример NP-полной проблемы, докажем лемму, которая послужит мощным инструментом для доказательства NP-полноты многочисленных массовых проблем, но после того как будет найдена хотя бы одна NP-полная проблема.

Лемма 8.3.17. *Если один принадлежащий классу* **NP** *язык, являющийся NP-полным, сводится к другому языку, принадлежащему классу* **NP**, *то и этот другой язык также является NP-полным.*

Доказательство. Пусть языки $L_1, L_2 \in \mathbf{NP}$, язык L_1 – NP-полный и $L_1 \Longrightarrow L_2$. Покажем, что тогда и язык L_2 также является NP-полным. Рассмотрим любой язык $L' \in \mathbf{NP}$. Так как L_1 – это NP-полный язык, то $L' \Longrightarrow L_1$. В силу транзитивности отношения сводимости из сводимостей $L' \Longrightarrow L_1$ и $L_1 \Longrightarrow L_2$ следует, что $L' \Longrightarrow L_2$. Так как L' был взят произвольно, то это и означает, что к L_2 сводится любой язык из **NP**, т.е. язык L_2 является NP-полным. □

На уровне проблем распознавания эта лемма указывает простой путь доказательства NP-полноты некоторой новой проблемы П, если известна хотя бы одна NP-полная проблема Π_0. Для этого достаточно доказать, что $\Pi \in \mathbf{NP}$ и $\Pi_0 \Longrightarrow \Pi$.

Но прежде чем можно будет воспользоваться этим методом доказательства, необходимо найти некоторую исходную NP-полную проблему. В свою очередь, для доказательства NP-полноты некоторой первой проблемы необходимо будет явно доказать, что все

другие проблемы из класса **NP** полиномиально сводятся к рассматриваемой проблеме. А для этого в доказательстве можно будет использовать лишь самые общие характеристики всевозможных проблем из **NP**. Такими характеристиками, согласно определения, естественно являются существование для каждой индивидуальной задачи $z \in$ П, допускающей положительное решение, такой догадки s, и такого алгоритма A проверки догадки, который получив на входе (z, s), закончит работу с ответом ДА.

NP-полнота проблемы выполнимости для формул логики высказываний. Честь быть "первой" NP-полной проблемой выпала на долю хорошо известной из математической логики проблемы выполнимости формул алгебры высказываний. Она состоит в следующем. Дана формула алгебры высказываний $F(X_1, X_2, ..., X_n)$. Спрашивается, выполнима ли она, т.е. существует ли такой набор α_1, α_2, ... , α_n значений её пропозициональных (булевых) переменных $X_1, X_2, ..., X_n$, который превращает эту формулу в истинное высказывание $F(\alpha_1, \alpha_2, ..., \alpha_n) = 1$. (Здесь $\alpha_1, \alpha_2, ..., \alpha_n \in \{0, 1\}$). Такой набор значений называется *выполняющим набором.* Для определенности считают, что формула $F(X_1, X_2, ..., X_n)$ представлена в конъюнктивной нормальной форме, т.е. в виде конъюнкции дизъюнктивных одночленов (дизъюнктов), т.е. формул вида $X_1 \vee \neg X_2$, $\neg X_2 \vee X_3 \vee \neg X_6$, $X_1 \vee \neg X_3 \vee X_5$ и т.п. Назовём эту проблему ВЫПОЛНИМОСТЬ (или сокращенно, ВЫП). Замети, что если данная формула F не представлена в конъюнктивной нормальной форме, то равносильными преобразованиями, используя известные законы алгебры логики, её к такой форме легко привести. Ввиду равносильности преобразований, полученная конъюнктивная нормальная форма будет выполнима в том и только в том случае, если выполнима исходная формула F.

Следующая фундаментальная теорема была доказана американским математиком Стивеном Куком в 1971 г.

Теорема 8.3.18. *Проблема* ВЫПОЛНИМОСТЬ *есть NP-полная проблема.*

Идея доказательства. Нетрудно понять, что проблема ВЫП лежит в классе **NP**. Если $F(X_1, X_2, ..., X_n) \cong D_1 \wedge D_2 \wedge ... \wedge D_m$, где $D_1, D_2, ..., D_m$ – дизъюнктивные одночлены от переменных X_1, X_2, ... , X_n, то недетерминированному алгоритму для решения достаточно указать набор значений переменных $\alpha_1, \alpha_2, ..., \alpha_n$ и осуществить проверку того, что этот набор превращает все дизъюнкты D_1, D_2, ... , D_m в 1. Всё это легко сделать (недетерминированным образом)

за полиномиальное время. Таким образом, ВЫП $\in$ **NP**.

Доказательство второй части теоремы может быть проведено на одном из двух языков – на языке массовых проблем или на языке формальных языков, представляющих массовые проблемы.

В первом случае нужно показать, что если проблема $\Pi \in \mathbf{NP}$, то Π полиноминально сводится к проблеме ВЫП. Для этого нужно, используя лишь тот факт, что $\Pi \in \mathbf{NP}$, для каждой единичной задачи $x \in \Pi$ уметь построить такую формулу $F(x)$ логики высказываний, что x будет единичной задачей массовой проблемы распознавания Π с ответом ДА тогда и только тогда, когда формула $F(x)$ выполнима. Причём, это построение должно потребовать только полиномиального – относительного размера $|x|$ единичной задачи x – количества времени. Доказательство в этом ключе приведено в книге [92], стр. 366 – 369.

Во втором случае нужно представить проблему ВЫП языком $L_{\text{ВЫП}} = L[\text{ВЫП}, e]$ для некоторой разумной схемы кодирования e и затем доказать, что для всех языков $L \in \mathbf{NP}$ имеет место полиномиальная сводимость $L \Longrightarrow L_{\text{ВЫП}}$. Для этого должна быть построена функция f, отображающая множество слов X^* в алфавите X языка L в множество слов (в некотором алфавите), которые кодируют единичные задачи из проблемы ВЫП. Однако, здесь удобно и наглядно взять не множество кодирующих слов, а множество самих единичных задач из ВЫП, т.е. фактически множество формул логики высказываний. Причём эта функция f должна обладать тем свойством, что:

$$(\forall x \in X^*)(x \in L \longleftrightarrow f(x) \text{ – выполнима}) ,$$

и единичная задача $f(x)$ может быть построена за время, ограниченное полиномом от длины $|x|$ слова x. Доказательство в этом ключе приведено в книге [49], стр. 57 – 63.

Доказательство теоремы Кука. Приведём доказательство второй части теоремы Кука на языке массовых проблем (т.е., как описано в предыдущем пункте – в первом случае). Предварительно проанализируем ещё раз основные понятия, связанные с формулировкой этой теоремы.

В начале пункта "*Недетерминированные алгоритмы и класс NP*" настоящего параграфа было описано понятие недетерминированного алгоритма и решения (в частности, за полиномиальное время) таким алгоритмом проблем распознавания. Далее, класс NP определён как класс таких проблем распознавания, которые могут быть ре-

шены недетерминированными алгоритмами за полиномиальное время.

Формализуем теперь следующим образом понятие алгоритма, который осуществляет стадию проверки догадки, т.е., начиная работу со входом (z, s), где z – код задачи, s – код догадки, заканчивает её ответом ДА для некоторой догадки s, если данная задача $z \in$ П действительно имеет положительное решение, и работает бесконечно для любой догадки s, если данная задача $z \in$ П не имеет положительного решения. В качестве такой формализации выберем следующую модификацию понятия машины Тьюринга.

Этот алгоритм можно представлять как устройство, которое преобразует символы из алфавита Σ, записанные по одному в ячейки ленты (строки). Чтение и запись происходят по одному символу за единицу времени с помощью читающе-пишущей головки, работа которой управляется программой, состоящей из команд. В исходном состоянии головка обозревает самый левый символ строки, и программа начинает выполнять свою первую команду.

Команды программы имеют вид: $\boxed{l\text{: } \mathbf{if}\ \sigma\ \mathbf{then}\ (\sigma';\ o;\ l')}$, где l и l' – номера команд (метки), σ и σ' – символы алфавита Σ, o – одно из чисел 1, 0 или -1. Выписанная выше команда имеет следующий смысл:

Если в данный момент времени обозревается символ σ, то стереть его и написать на его месте σ', сдвинуться на o позиций вправо и далее выполнять команду с меткой l'; в противном случае выполнять следующую команду.

Сдвиг на 0 или -1 разрядов вправо означает соответственно, что головка остаётся в той же позиции или сдвигается на 1 разряд влево. Последняя команда программы имеет вид: $\boxed{|A|\text{: } \mathbf{accept}}$, где $|A|$ – длина программы, т.е. общее число команд в ней.

Будем говорить, что строка начальных данных (z, s) *принимается* алгоритмом A, если этот алгоритм, начиная в правильном положении обрабатывать данную строку, после не более $p(|z|)$ шагов приходит к последней команде $|A|$: **accept** , где $p(n)$ – полиномиальная граница для A. Если алгоритм делает $p(|z|)$ шагов и не достигает команды **accept** ("принять") или если головка сходит со строки, то будем говорить, что строка начальных данных (z, s) *отвергается*.

Тогда *класс* **NP** можно более формально определить как множество проблем распознавания П, для которых существуют алгоритм A проверки догадок и полином $p(n)$, такие, что z является индивидуальной задачей проблемы П с ответом ДА тогда и только

тогда, когда существует такая догадка (строка) s, что $|s| \leq p(|z|)$ и алгоритм A принимает строку начальных данных (z, s).

Описанным формальным определением мы воспользуемся для доказательства второй части теоремы Кука, а именно утверждения о том, что если П – некоторая проблема из NP, то П полиномиально преобразуется в проблему ВЫПОЛНИМОСТЬ. Другими словами, для данной строки z (кодирующей задачу $z \in$ П) мы должны построить формулу $F(z)$ алгебры высказываний – используя только тот факт, что $\Pi \in NP$, – такую, что z является индивидуальной задачей проблемы П с ответом ДА тогда и только тогда, когда формула $F(z)$ выполнима. Для этого рассмотрим алгоритм A проверки догадок для проблемы П, описанный выше, время и память которого по нашему предположению о том, что $\Pi \in NP$, ограничены некоторым полиномом $p(n)$.

Формула $F(z)$ алгебры высказываний будет содержать следующие булевы (пропозициональные) переменные.

а) Булевы переменные $X_{ij\sigma}$ для всех $0 \leq i,\ j \leq p(|z|)$ и $\sigma \in \Sigma$. Переменная $X_{ij\sigma}$ будет соответствовать утверждению: ***j-я позиция строки в момент времени i содержит символ σ.***

б) Булевы переменные Y_{ijl} для всех $0 \leq i \leq p(|z|)$, $0 \leq j \leq p(|x|) + 1$ и $1 \leq l \leq |A|$, где $|A|$ – число команд в алгоритме A. Переменная Y_{ijl} будет соответствовать утверждению: ***в момент времени i обозревается j-я позиция и выполняется l-я команда.*** Если $j = 0$ или $j = (||) + 1$, то это означает, что головка сошла со строки и, следовательно, вычисление безуспешно.

Построим теперь из этих булевых переменных такую формулу $F(z)$ алгебры высказываний, что $F(z)$ выполнима тогда и только тогда, когда z – это индивидуальная задача проблемы П с ответом ДА. Если булевы переменные имеют указанный смысл, то $F(z)$ будет, по существу, утверждать, что алгоритм A, начиная работу со строкой $(z, \ldots)$, в левой части которой стоит z, может в результате принять эту строку; по определению это будет означать, что существует подходящая догадка s (т.е. такая догадка, для которой строка начальных данных (z, s) ***принимается*** алгоритмом A); и, следовательно, что z – индивидуальная задача проблемы П с ответом ДА.

Формула $F(z)$ является конъюнкцией четырёх частей (подформул): $F(z) = U(z) \wedge S(z) \wedge W(z) \wedge (z)$.

1. Подформула $U(z)$ выражает тот факт, что в каждый момент времени i, $0 \leq i \leq p(|z|)$, в каждой позиции строки содержится

единственный символ, головка обозревает *единственную* позицию в пределах строки и программа выполняет *единственный* оператор:

$$U(z) = (\wedge_{0\le i,\ j\le p(|z|),\ \sigma\ne\sigma'} (\neg X_{ij\sigma} \vee \neg X_{ij\sigma'})) \wedge$$
$$\wedge\ (\wedge_{0\le i\le p(|z|),\ j\ne j' \text{ или } l\ne l'} (\neg Y_{ijl} \vee \neg Y_{ij'l'})) \wedge$$
$$\wedge\ (\wedge_{0\le i\le p(|z|),\ 1\le l\le |A|} (\neg Y_{i0l} \vee \neg Y_{i,p(|z|)+1,l})) \wedge$$
$$\wedge\ (\wedge_{0\le i\le p(|z|)} ((\wedge_{0\le j\le p(|z|)} \vee_{\sigma\in\Sigma} X_{ij\sigma}) \vee_{0\le j\le p(|z|),\ 1\le l\le |A|} Y_{ijl})).$$

2. Подформула $S(z)$ утверждает, что A корректно *начинает* работу; другими словами, в нулевой момент времени самые левые $|z|+1$ символов в строке образуют слово za_0, читающе-пишущая головка обозревает самый левый символ строки и программа переходит к выполнению первой команды алгоритма:

$$S(z) = (\wedge_{0\le j\le |z|} X_{0jz(j)}) \wedge X_{0,\ |z|+1,\ a_0} \wedge Y_{011}\ .$$

(Здесь $z(j)$ обозначает j-й символ строки z).

3. Подформула $W(z)$ утверждает, что A *работает* правильно в соответствии с командами программы; $W(z)$ является конъюнкцией формул $W_{ij\sigma l}$, по одной для каждых $0\le i\le p(|z|)$, $1\le j\le p(|z|)$, $\sigma\in\Sigma$ и $1\le l<|A|$, таких, что l-я команда алгоритма A имеет вид

$$l\text{: } \mathbf{if}\ \sigma\ \mathbf{then}\ (\sigma';\ o;\ l')\ .$$

Формула $W_{ij\sigma l}$ определяется следующим образом:

$$W_{ij\sigma l} = (\neg X_{ij\sigma} \vee \neg Y_{ijl} \vee X_{i+1,\ j,\ \sigma'}) \wedge$$
$$\wedge\ (\neg X_{ij\sigma} \vee \neg Y_{ijl} \vee Y_{i+1,\ j+0,\ l'}) \wedge$$
$$\wedge_{\tau\ne\sigma}((\neg X_{ij\tau} \vee \neg Y_{ijl} \vee X_{i+1,\ j,\ \tau}) \wedge (\neg X_{ij\tau} \vee \neg Y_{ijl} \vee Y_{i+1,\ j,\ l+1})).$$

Она означает, что если $X_{ij\sigma}$ и Y_{ijl} оба истинны, то в следующий момент времени переменные X и Y, утверждающие, что алгоритм A выполнил правильные действия, также должны быть истинными. Для последней команды алгоритма A и для каждых i, j и σ добавляется формула

$$W_{ij\sigma|A|} = (\neg X_{ij\sigma} \vee \neg Y_{ij|A|} \vee Y_{i+1,\ j,\ |A|})\ ,$$

утверждающая, что как только алгоритм достигает команды **accept**, он остаётся в этом положении. Кроме того, $W(z)$ содержит дизъюнкты

$$\bigwedge_{\substack{0 \le i \le p(|z|),\\ \sigma \in \Sigma\\ 1 \le l \le |A|\\ j \ne j'}} (\neg X_{ij\sigma} \vee \neg Y_{ij'l} \vee X_{i+1,\, j,\, \sigma}) ,$$

означающие, что если A обозревает позицию, отличную от j-й, то символ в j-й позиции остаётся неизменным.

4. Последняя часть формулы $F(z)$ утверждает просто, что A корректно *заканчивает* свою работу, т.е. программа при этом выполняет команду **accept**. Она состоит всего из одного дизъюнкта:

$$E(z) = \vee_{0 \le i \le p(|z|)} Y_{p(|z|),\, j,\, |A|} \ .$$

Этим завершается построение $F(z)$. Прежде всего заметим, что это построение требует только полиномиального – относительно $|z|$ – количества времени. Это следует из того факта, что общая длина формулы F (число вхождений литералов, умноженное на длину индексов этих литералов в формуле F) не превосходит $O(p^3(|z|)) \cdot \log p(|z|)$. Поэтому, чтобы показать, что это построение является полиномиальным сведением (преобразованием) проблемы A в проблему ВЫПОЛНИМОСТЬ, остаётся доказать следующее утверждение.

Утверждение. ***Формула $F(z)$ выполнима в том и только в том случае, если z является индивидуальной задачей проблемы*** П ***с ответом*** ДА.

Для доказательства *необходимости* предположим, что $F(z)$ выполнима. Тогда все формулы $U(z)$, $S(z)$, $W(z)$ и $E(z)$ выполнимы при одном и том же наборе t значений пропозициональных (булевых) переменных. Так как на наборе t выполняется $U(z)$, то для всех i и j ровно одна переменная $X_{ij\sigma}$ должна быть истинной; пусть это означает, что j-я ячейка в момент времени i содержит символ σ. Кроме того, для всех i ровно одна из переменных Y_{ijl} истинна; пусть это означает, что в момент времени i обозревается j-я позиция строки и выполняется оператор l. Наконец, никакая переменная вида Y_{i0l} или $Y_{i,p(|z|)+1,l}$ не может быть истинной, что означает, что головка никогда не сходит со строки. Таким образом, набор значений истинности t описывает некоторую последовательность строк, положений головки и команд. Покажем, что эта последовательность образует правильное принимающее вычисление для алгоритма A при входе (z, s) для некоторой догадки s.

Так как $S(z)$ также должно выполняться на наборе t, то указанная последовательность начинается корректно: вначале первые

$|z| + 1$ мест заняты правильной строкой za_0 и при выполнении, первой команды обозревается первый символ строки z.

Тот факт, что $W(z)$ также выполняется на наборе t, означает, что последовательность изменяется согласно правилам выполнения алгоритма A. Наконец, $E(z)$ выполняется на наборе t только в том случае, если алгоритм оканчивается последней принимающей командой. Следовательно, если формула $F(z)$ выполнима, то существует догадка s подходящей длины, такая, что A принимает строку начальных данных (z, s); поэтому z является индивидуальной задачей проблемы П с ответом ДА.

Для доказательства *достаточности* допустим, что z – индивидуальная задача с ответом ДА. Тогда существует такая догадка (строка) s длины $p(|z|) - |z| - 1$, что алгоритм A принимает строку начальных данных (z, s). Это означает, что существует последовательность $p(|z|)$ (с первой строкой (z, s)) номеров команд и обозреваемых позиций, допустимая для алгоритма A и оканчивающаяся принятием строки (z, s). Эта последовательность непосредственно определяет набор значений истинности t переменных X, Y, который с необходимостью выполняет формулу $F(z)$. Этим завершается доказательство утверждения.

Рассматриваемое утверждение показывает, что формула алгебры высказываний $F(z)$ является индивидуальной задачей проблемы ВЫПОЛНИМОСТЬ с ответом ДА тогда и только тогда, когда z является индивидуальной задачей проблемы П с ответом ДА. Поэтому описанное преобразование (сведение) является полиномиальным преобразованием (сведением) проблемы П к проблеме ВЫПОЛНИМОСТЬ. Так как в качестве П была выбрана произвольная проблема из класса **NP**, то теорема доказана. □

Итак, проблема ВЫПОЛНИМОСТЬ NP-полна.

Примеры NP-полных массовых проблем. Теперь мы можем начать пользоваться леммой 8.3.17 для доказательства NP-полноты тех или иных проблем: для этого нужно доказывать полиномиальную сводимость проблемы ВЫП к исследуемой массовой проблеме. Приведём примеры шести NP-полных проблем, NP-полнота которых доказывается сведением к ним проблемы ВЫП. Эти шесть проблем входят в число тех, которые наиболее часто используются при доказательствах результатов об NP-полноте в качестве "известных NP-полных проблем".

3-ВЫПОЛНИМОСТЬ (3-ВЫП)

Условие. Дана конъюнктивная нормальная форма $F(X_1, X_2, ..., X_n) \cong D_1 \wedge D_2 \wedge ... \wedge D_m$ от переменных $X_1, X_2, ..., X_n$, каждый дизъюнкт которой $D_1, D_2, ..., D_m$ содержит точно три из этих переменных.

Вопрос. Выполнима ли формула $F(X_1, X_2, ..., X_n)$, т.е. существует ли двоичный набор $\alpha_1, \alpha_2, ..., \alpha_n$ значений переменных, превращающий F в 1: $F(\alpha_1, \alpha_2, ..., \alpha_n) = 1$?

ТРЁХМЕРНОЕ СОЧЕТАНИЕ (3-С)

Условие. Дано множество $M \subseteq X \times Y \times Z$, где X, Y, Z – непересекающиеся множества, содержащие одинаковое число q элементов.

Вопрос. Верно ли, что M содержит трёхмерное сочетание, т.е. подмножество $M' \subseteq M$ такое, что $|M'| = q$ и никакие два различных элемента из M' не имеют ни одной равной координаты?

ВЕРШИННОЕ ПОКРЫТИЕ (ВП)

Условие. Дан граф $G = < V, E >$ и положительное целое число k, $k \leq |V|$.

Вопрос. Имеются ли в графе G вершинное покрытие не более чем из k элементов, т.е. такое подмножество $V' \subseteq V$, что $|V'| \leq k$ и для каждого ребра $\{u, v\} \in E$ хотя бы одна из вершин u или v принадлежит V' ?

КЛИКА

Условие. Дан граф $G = < V, E >$ и положительное целое число k, $k \leq |V|$.

Вопрос. Верно ли, что G содержит некоторую клику мощности не менее k, т.е. такое подмножество $V' \subseteq V$, что $|V'| \geq k$ и любые две вершины из V' соединены ребром из E ?

ГАМИЛЬТОНОВ ЦИКЛ (ГЦ)

Условие. Дан граф $G = < V, E >$.

Вопрос. Верно ли, что G содержит гамильтонов цикл, т.е. такую последовательность $< v_1, v_2, ..., v_n >$ вершин графа G, что $n = |V|$, $\{v_n, v_1\} \in E$ и $\{v_i, v_{i+1}\} \in E$ для всех i таких, что $1 \leq i \leq n - 1$?

РАЗБИЕНИЕ.

Условие. Заданы конечное множество A и вес $s(a) \in N^+$ для каждого $a \in A$.

Вопрос. Существует ли подмножество $A' \in A$ такое, что $\Sigma_{a \in A'} s(a) = \Sigma_{a \in A \setminus A'} s(a)$?

NP-полнота этих проблем впервые была доказана американским математиком Р.М.Карпом (Karp) в 1992 г. В книге [49] приводятся

подробные доказательства NP-полноты этих проблем методом сведения к ним проблемы ВЫПОЛНИМОСТЬ.

Таким образом, перечисленные шесть проблем, а также и все другие NP-полные проблемы столь же "трудны", как и проблема о выполнимости формул логики высказываний. В настоящее время класс NP-полных проблем насчитывает сотни представителей. Среди них проблемы коммивояжера, целочисленного линейного программирования, о пересечении трёх матроидов и т.д. Они образуют класс эквивалентности, состоящий из "самых трудных" проблем из класса **NP**. По мере того как все больше и больше проблем, представляющих интерес с разных точек зрения, попадают в этот класс эквивалентности, его значимость постоянно возрастает.

NP-полнота проблемы 3-ВЫПОЛНИМОСТЬ. Покажем, например, как осуществляется сведение проблемы ВЫПОЛНИМОСТЬ к проблеме 3-ВЫПОЛНИМОСТЬ и тем самым устанавливается NP-полнота последней проблемы. Проблема 3-ВЫПОЛНИМОСТЬ есть просто ограниченный вариант проблемы ВЫПОЛНИМОСТЬ, в которой каждая индивидуальная задача, т.е. каждая конъюнктивная нормальная форма в каждом своём дизъюнкте содержит ровно три буквы (пропозициональных переменных). Тем не менее, решение проблемы ВЫП может быть сведено к решению проблемы 3-ВЫП и тем самым доказана NP-полнота последней. Из-за своей более простой структуры эта проблема часто используется для доказательства результатов об NP-полноте.

ТЕОРЕМА 8.3.19. *Проблема* 3-ВЫПОЛНИМОСТЬ *есть NP-полная проблема.*

Доказательство. Ясно, что 3-ВЫП $\in$ **NP**. Это следует из того, что недетерминированному алгоритму необходимо угадать лишь набор значений истинности переменных данной конъюнктивной нормальной формы и проверить за полиномиальное время, будут ли при таком наборе значений переменных превращаться в 1 все дизъюнкты этой формы.

Докажем теперь, что проблема ВЫП сводится к проблеме 3-ВЫП. Тогда отсюда последует NP-полнота проблемы 3-ВЫП, ибо ввиду NP-полноты проблемы ВЫП (теорема 8.3.18) к ней полиномиально сводится всякая проблема из класса **NP**, а значит, в силу транзитивности отношения сводимости, всякая проблема из **NP** полиномиально сводится к проблеме 3-ВЫП.

Пусть $F(X_1, X_2, ..., X_n) \cong D_1 \wedge D_2 \wedge \ ... \ \wedge D_m$ – произвольная конъюнктивная нормальная форма от переменных $X_1, X_2, ..., X_n$,

определяющая произвольную индивидуальную задачу из проблемы ВЫП. Обозначим $\overline{X} = \{X_1, X_2, ..., X_n\}$ – множество пропозициональных переменных и $\overline{D} = \{D_1, D_2, ..., D_m\}$ – множество дизъюнктов этой формы. Построим набор $\overline{D'}$ трёхбуквенных дизъюнкций на некотором множестве $\overline{X'}$ пропозициональных переменных, такой, что все дизъюнкции из $\overline{D'}$ превращаются в 1 тогда и только тогда, когда в 1 превращаются все дизъюнкции из $\overline{D}$. Другими словами, набор $\overline{D'}$ выполним тогда и только тогда, когда набор $\overline{D}$ выполним.

Набор $\overline{D'}$ будет строиться путём замены каждого отдельного дизъюнкта $D_j \in \overline{D}$ "эквивалентным" набором $\overline{D'_j}$ трёхбуквенных дизъюнктов от переменных из $\overline{X}$ (из исходного множества переменных) и из множества $\overline{X'_j}$ некоторых дополнительных переменных; причём, переменные из $\overline{X'_j}$ будут использоваться только в дизъюнктах из $\overline{D'_j}$. Другими словами,

$$\overline{X'} = \overline{X'} \cup (\cup_{j=1}^{m} \overline{X'_j}) \quad \text{и} \quad \overline{D'} = \cup_{j=1}^{m} \overline{D'_j} \ .$$

Таким образом, нужно только показать, как, исходя из дизъюнкта $D_j \in \overline{D}$, построить набор $\overline{X'_j}$ дополнительных переменных и набор $\overline{D'_j}$ трёхбуквенных дизъюнктов от переменных из $\overline{X} \cup \overline{X'_j}$.

Пусть D_j – дизъюнктивный одночлен (дизъюнкт) от переменных X_1, X_2, ... , $X_n \in \overline{X}$. Способ образования множеств $\overline{X'_j}$ и $\overline{D'_j}$ зависит от значения k.

1) $\underline{k=1}$. В этом случае D_j есть X_1 или $\neg X_1$. Тогда $\overline{X'_j} = \{Y_j^1, Y_j^2\}$ и $\overline{D'_j} = \{D_j \vee Y_j^1 \vee Y_j^2,\ D_j \vee Y_j^1 \vee \neg Y_j^2,\ D_j \vee \neg Y_j^1 \vee Y_j^2,\ D_j \vee \neg Y_j^1 \vee \neg Y_j^2, \}$.
Таким образом, в этом случае дизъюнкт D_j в данной формуле $F(X_1, X_2, ... , X_n)$ заменяется равносильной формулой, представляющей собой конъюнкцию всех дизъюнктов из $\overline{D_j}$:

$$D_j \cong (D_j \vee Y_j^1 \vee Y_j^2) \wedge (D_j \vee Y_j^1 \vee \neg Y_j^2) \wedge$$
$$\wedge (D_j \vee \neg Y_j^1 \vee Y_j^2) \wedge (D_j \vee \neg Y_j^1 \vee \neg Y_j^2) \ .$$

2) $\underline{k=2}$. В этом случае дизъюнкт D_j содержит две переменных X_1 и X_2 и может иметь один из следующих видов: $X_1 \vee X_2$, $X_1 \vee \neg X_2$, $\neg X_1 \vee X_2$, $\neg X_1 \vee \neg X_2$. Добавим одну новую переменную Y_j^1, т.е. $\overline{X'_j} = \{Y_j^1\}$ и рассмотрим множество из трёхбуквенных дизъюнктов: $\overline{D'_j} = \{D_j \vee Y_j^1,\ D_j \vee \neg Y_j^1\}$.
Таким образом, в этом случае дизъюнкт D_j в данной формуле $F(X_1, X_2, ... , X_n)$ заменяется равносильной формулой, представляющей собой конъюнкцию двух трёхбуквенных дизъюнктов из $\overline{D_j}$:

$$D_j \cong (D_j \vee Y_j^1) \wedge (D_j \vee \neg Y_j^1) \ .$$

3) $\underline{k=3}$. В этом случае дизъюнкт D_j содержит требуемые три переменные X_1, X_2, X_3, и его ни на что заменять не надо, т.е. в этом случае: $\overline{X_j'} = \emptyset$ и $\overline{D_j'} = \{D_j\}$. Таким образом, в этом случае

$$D_j \cong D_j \ .$$

4) $\underline{k>3}$. В этом случае дизъюнкт D_j содержит переменные X_1, X_2, ... , X_k. Не нарушая общности, можно считать, что в нём отсутствуют сочетания вида $X_i \vee X_i$, а также вида $X_i \vee \neg X_i$, ибо в последнем случае дизъюнкт D_j становится тождественно истинным и не влияет на выполнимость всей формулы $F(X_1, X_2, ..., X_n)$. Таким образом, в дизъюнкте D_j каждая из переменных X_1, X_2, ... , X_k встречается точно один раз – либо сама, либо со знаком отрицания, и, кроме того, эти переменные записаны в порядке возрастания их индексов; так что дизъюнкт D_j можно представить в виде: $D_j \equiv X_1^* \vee X_2^* \vee ... \vee X_k^*$, где X_i^* $(1 \le i \le k)$ – обозначает либо X_i, либо $\neg X_i$ в зависимости от того, без знака отрицания или со знаком отрицания входит переменная X_i в исходный дизъюнкт D_j.

Нужно заменить дизъюнкт D_j конъюнкцией трёхбуквенных дизъюнктов. При этом, заменить на равносильную конъюнкцию (как это было в предыдущих случаях 1 - 3) нам не удастся. Но этого и не требуется. Достаточно заменить на такую конъюнкцию, которая была бы "эквивалентна" дизъюнкту D_j в смысле их выполнимости: формула D_j выполнима тогда и только тогда, когда выполнима конъюнкция трёхбуквенных дизъюнктов.

Введём в этом случае следующие дополнительные переменные $\overline{X_j'} = \{Y_j^1, Y_j^2, ..., Y_j^{k-3}\}$ и рассмотрим следующее множество трёхбуквенных дизъюнктов от "старых" переменных: $X_1, X_2, ..., X_k$ и "новых" переменных Y_j^1, Y_j^2, ... , Y_j^{k-3}, конъюнкция которых даст требуемую формулу:

$$\overline{D_j'} = \{X_1^* \vee X_2^* \vee Y_j^1\} \cup$$
$$\{\neg Y_j^i \vee X_{i+2}^* \vee Y_j^{i+1} : 1 \le i \le k-4\} \cup \{\neg Y_j^{k-3} \vee X_{k-1}^* \vee X_k^*\}.$$

Рассмотрим ПРИМЕР. Пусть

$$F(X_1, X_2, X_3, X_4, X_5, X_6) \equiv \neg X_1 \wedge (X_1 \vee \neg X_2) \wedge (X_1 \vee X_2 \vee X_3) \wedge$$
$$\wedge (\neg X_1 \vee X_2 \vee X_3 \vee \neg X_4) \wedge (\neg X_1 \vee X_2 \vee \neg X_3 \vee \neg X_4 \vee X_5 \vee X_6) \equiv$$
$$\equiv D_1 \wedge D_2 \wedge D_3 \wedge D_4 \wedge D_5 \ .$$

Применяем правила 1 - 4 к дизъюнктам D_1, D_2, D_3, D_4, D_5:

1) $D_1 \equiv \neg X_1 \cong (\neg X_1 \vee Y_1^1 \vee Y_1^2) \wedge (\neg X_1 \vee Y_1^1 \vee \neg Y_1^2) \wedge$
$\wedge (\neg X_1 \vee \neg Y_1^1 \vee Y_1^2) \wedge (\neg X_1 \vee \neg Y_1^1 \vee \neg Y_1^2)$.

2) $D_2 \equiv X_1 \vee \neg X_2 \cong (X_1 \vee \neg X_2 \vee Y_1^2) \wedge (X_1 \vee \neg X_2 \vee \neg Y_1^2)$.

3) $D_3 \equiv X_1 \vee X_2 \vee X_3 \cong X_1 \vee X_2 \vee X_3$.

4) $D_4 \equiv \neg X_1 \vee X_2 \vee X_3 \vee \neg X_4$. Этот дизъюнкт заменяется на следующую конъюнкцию трёхбуквенных дизъюнктов:

$$(\neg X_1 \vee X_2 \vee Y_4^1) \wedge (\neg Y_4^1 \vee X_3 \vee \neg X_4) .$$

$D_5 \equiv \neg X_1 \vee X_2 \vee \neg X_3 \vee \neg X_4 \vee X_5 \vee X_6$. Этот дизъюнкт заменяется на следующую конъюнкцию трёхбуквенных дизъюнктов:

$$(\neg X_1 \vee X_2 \vee Y_5^1) \wedge (\neg Y_5^1 \vee \neg X_3 \vee Y_5^2) \wedge$$
$$(\neg Y_5^2 \vee \neg X_4 \vee Y_5^3) \wedge (\neg Y_5^3 \vee X_5 \vee X_6).$$

Конъюнкция всех полученных формул представляет собой формулу, зависящую от переменных $X_1, X_2, X_3, X_4, X_5, X_6, Y_1^1, Y_1^2, Y_2^1$, Y_4^1, Y_5^1, Y_5^2, Y_5^3, являющуюся конъюнкцией трёхбуквенных дизъюнкций и выполняющуюся тогда и только тогда, когда выполняется исходная формула $F(X_1, X_2, X_3, X_4, X_5, X_6)$. Собственно это утверждение ещё требует уточнения и доказательства.

Вернёмся к доказательству теоремы.

Для доказательства того, что здесь в самом деле имеет место сводимость проблемы **ВЫП** к проблеме **3-ВЫП**, необходимо показать, что набор дизъюнкций $\overline{D'}$ выполним тогда и только тогда, когда выполним исходный набор $\overline{D}$ дизъюнкций.

Предположим вначале, что $t = (\alpha_1, \alpha_2, ..., \alpha_n)$ есть набор значений истинности (т.е. нулей и единиц), выполняющий $\overline{D}$, т.е. превращающий в 1 все дизъюнкции $D_1, D_2, ..., D_m$ при подстановке $X_1 = \alpha_1$, $X_2 = \alpha_2$, ... , $X_n = \alpha_n$. Покажем, что t может быть продолжен до набора значений истинности $t' = (\alpha_1, \alpha_2, ..., \alpha_n; \beta_1, \beta_2, ...)$, который выполняет $\overline{D'}$, т.е. превращает в 1 все дизъюнкты из этого набора.

Поскольку переменные в множестве $\overline{X'} \setminus \overline{X}$ делятся на группы $\overline{X'_j}$ и так как переменные в каждой группе $\overline{X'_j}$ входят только в дизъюнкты, принадлежащие набору $\overline{D'_j}$, то достаточно показать, как t может быть продолжен на каждое множество $\overline{X'_j}$ отдельно, и в каждом случае нужно проверить, что выполняются все дизъюнкты в соответствующем множестве $\overline{D'_j}$.

Доказательства возможности такого продолжения проделаем в каждом из четырёх случаев 1) – 4) построения множеств $\overline{X'_j}$ и $\overline{D'_j}$.

В случаях 1) – 3) утверждение очевидно ввиду того, что дизъюнкты заменяются на равносильные им формулы, и если дизъюнкт D_j при подстановке t: $X_1 = \alpha_1$, $X_2 = \alpha_2$, ... , $X_n = \alpha_n$ принял значение 1, то и равносильная ему формула при этих же значениях для переменных $X_1, X_2, \dots, X_n$ также примет значение 1, вне зависимости от того, какие значения принимают дополнительные ("новые") переменные из $\overline{X'_j}$ (этим переменным можно придавать при этом любые значения).

Рассмотрим случай 4). Дизъюнкт D_j имеет вид: $D_j \equiv X_1^* \vee X_2^* \vee \dots \vee X_k^*$. Поскольку подстановка t: $X_1 = \alpha_1$, $X_2 = \alpha_2$, ... , $X_n = \alpha_n$ превращает его в 1, то найдётся такое целое $l: 1 \le l \le k$, что один из членов этой дизъюнкции превратится в 1: $X_l^*|_{X_l=\alpha_l} = \alpha_l^* = 1$.

Если $l = 1$ или 2, то полагаем $\beta_1 = \beta_2 = \dots = \beta_{k-3} = 0$. В этом случае первый дизъюнкт из $\overline{D'_j}$ примет значение 1, так как $X_1^*|_{X_1=\alpha_1} = \alpha_1^* = 1$ или $X_2^*|_{X_2=\alpha_2} = \alpha_2^* = 1$. Остальные дизъюнкты из $\overline{D'_j}$ содержат отрицания переменных $\neg Y_j^i$, $1 \le i \le k-3$, которые при подстановке $Y_j^1 = \beta_1 = 0$, $Y_j^2 = \beta_2 = 0$, ... , $Y_j^{k-3} = \beta_{k-3} = 0$ примут значение 1, обеспечив выполнимость всех этих дизъюнктов.

Если $l = k-1$ или k (другой крайний случай), то аналогично полагаем: $\beta_1 = \beta_2 = \dots = \beta_{k-3} = 1$. В этом случае все дизъюнкты из $\overline{D'_j}$, кроме последнего, имеют своими слагаемыми переменные Y_j^1, Y_j^2, ... , Y_j^{k-3}, которые при подстановке $Y_j^1 = \beta_1 = 1$, $Y_j^2 = \beta_2 = 1$, ... , $Y_j^{k-3} = \beta_{k-3} = 1$ примут значение 1, обеспечив выполнимость всех этих дизъюнктов. Последний дизъюнкт примет значение 1 в силу того, что значение 1 примет член X_{k-1}^* (если $\alpha_{k-1} = 1$) или член X_k^* (если $\alpha_k = 1$).

Наконец, если $l \ne 1$, $l \ne 2$, $l \ne k-1$, $l \ne k$, то полагаем $\beta_1 = \dots = \beta_{l-2} = 1$ и $\beta_{l-1} = \dots = \beta_{k-3} = 0$. В этом случае первые $l-2$ дизъюнктов из $\overline{D'_j}$ примут значение 1, благодаря входящим в них слагаемым Y_j^1, ... , Y_j^{l-2}, а остальные $(k-2)-(l-2) = k-l$ дизъюнктов примут значение 1, благодаря входящим в них слагаемым $\neg Y_j^{l-1}$, ... , $\neg Y_j^{k-3}$, для которых будет выполняться подстановка $Y_j^{l-1} = \beta_{l-1} = 0$, ... , $Y_j^{k-3} = \beta_{k-3} = 0$.

Обратно, если t' – некоторый выполняющий набор значений истинности для всех дизъюнктов из $\overline{D'_j}$, то его ограничение t на исход-

ные переменные $X_1, X_2, \dots, X_k$ обеспечит выполнимость дизъюнкта D_j. Это утверждение требует доказательства только в случае 4. В самом деле, если допустить, что подстановка $t' = (\alpha_1, ..., \alpha_k, \beta_1, ..., \beta_{k-3})$ такова, что её ограничение $t = (\alpha_1, ..., \alpha_k)$ не выполняет формулу D_j, то это означает, что $\alpha_1^* = \dots = \alpha_k^* = 0$. Делаем соответствующие подстановки в дизъюнкты из $\overline{D'_j}$:

$$\alpha_1^* \vee \alpha_2^* \vee \beta_1 = 1, \quad \alpha_1^* = \alpha_2^* = 0 \implies \beta_1 = 1\,;$$

$$\neg\beta_1 \vee \alpha_3^* \vee \beta_2 = 1, \quad \alpha_3^* = 0, \ \beta_1 = 1 \implies \beta_2 = 1\,;$$

$$\neg\beta_2 \vee \alpha_4^* \vee \beta_3 = 1, \quad \alpha_4^* = 0, \ \beta_2 = 1 \implies \beta_3 = 1\,;$$

. . . .

$$\neg\beta_{k-4} \vee \alpha_{k-2}^* \vee \beta_{k-3} = 1, \quad \alpha_{k-2}^* = 0, \ \beta_{k-4} = 1 \implies \beta_{k-3} = 1.$$

Тогда вычисляем значение последнего дизъюнкта из $\overline{D'_j}$:

$$\neg\beta_{k-3} \vee \alpha_{k-1}^* \vee \alpha_k^* = \neg 1 \vee 0 \vee 0 = 0 \vee 0 = 0\ .$$

Таким образом, мы находим в $\overline{D'_j}$ невыполнимый дизъюнкт.

Итак, если подстановка t' выполняет множество дизъюнктов $\overline{D'_j}$, то её ограничение t выполняет формулу (дизъюнкт) D_j.

Окончательно приходим к тому, что набор трёхбуквенных дизъюнкций $\overline{D'}$ выполним тогда и только тогда, когда выполним набор $\overline{D}$ исходных произвольных дизъюнкций. Это и означает, что проблема ВЫП сведена к проблеме 3-ВЫП.

Для того чтобы убедиться, что эта сводимость полиномиальная, достаточно заметить, что число трёхбуквенных дизъюнкций в $\overline{D'}$ ограничено полиномом от mn. Следовательно, размер индивидуальной задачи из 3-ВЫП ограничен сверху некоторым полиномом от размера соответствующей задачи из ВЫП, а так как все детали построения сводимости очевидны, то нетрудно проверить, что эта сводимость является полиномиальной.

Теорема полностью доказана. □

Ограниченная структура проблемы 3-ВЫП делает её гораздо более полезной при доказательстве результатов об NP-полноте по сравнению с проблемой ВЫП. Малая длина дизъюнкций позволяет использовать сводимости, которые вообще не могли бы быть построены для индивидуальных задач с дизъюнкциями большей длины. Это наводит на мысль, что было бы ещё лучше, если бы удалось показать, что аналогичная проблема 2-ВЫПОЛНИМОСТЬ (каждая дизъюнкция содержит ровно две буквы) является NP-полной. Однако проблема 2-ВЫП может быть решена методом "резолюций" за

время, ограниченное полиномом от произведения числа дизъюнкций и числа переменных заданной индивидуальной задачи[1] (см. также[2]), и, следовательно, принадлежит классу **P**.

NP-полнота проблемы КЛИКА. Напомним, что проблема КЛИКА состоит в том, чтобы узнать, имеется ли в графе клика данной мощности. (Под кликой в графе понимается вполне связанное подмножество его вершин, т.е. такой набор его вершин, что любые две из них соединены ребром).

Нетрудно понять, что проблема КЛИКА входит в класс **NP**. В самом деле, предположим, что нам дана индивидуальная задача из проблемы КЛИКА с ответом ДА; другими словами, даны граф $G = (V; E)$ и целое число k такие, что в G имеется клика C мощности k. Недетерминированная машина Тьюринга сначала может "догадаться", какие k вершин графа составляют вполне связанное подмножество, а затем проверить, что любая пара этих вершин соединена ребром из E. При этом, для проверки достаточно $O(n^3)$ шагов, где n – длина кода индивидуальной задачи о клике. Это демонстрирует "силу" недетерминизма: все подмножества из k вершин проверяются "параллельно" независимыми экземплярами распознающего устройства.

Не известно, содержится ли проблема КЛИКА в классе **P**. Очевидный путь решения этой задачи мог бы состоять в проверке для всех подмножеств множества V, имеющих мощность k (а их число равно числу сочетаний из $|V|$ элементов по k элементов), удовлетворяют ли они требованиям задачи. Число таких подмножеств с ростом $|V|$ и k возрастает экспоненциально, и полиномиального алгоритма для решения этой задачи не известно.

Докажем теперь NP-полноту проблемы КЛИКА, полиномиально сведя к ней проблему ВЫПОЛНИМОСТЬ.

Теорема 8.3.20. *Проблема* КЛИКА *есть* NP-*полная проблема.*

Доказательство. Пусть $F(X_1, X_2, ..., X_n) \cong D_1 \wedge D_2 \wedge \ ... \ \wedge D_m$ – произвольная конъюнктивная нормальная форма от пропозициональных переменных $X_1, X_2, ..., X_n$, определяющая произволь-

---

[1] *Cook S.A.* The complexity of theorem-proving procedures. – Proc. 3rd Ann. ACM Symp. On Theory of Computing, Association for Computing Mashinary / New York, 1971, 151 – 158.

[2] *Even S., Itai A., Shamir A.* On the complexity of timetable and multycommodity flow problems. – SIAM J.: Comput., 1976, 5, 691 – 703.

ную индивидуальную задачу из проблемы ВЫП. Здесь D_1, D_2, ... , D_m – дизъюнкты, т.е. дизъюнкции пропозициональных переменных и/или их отрицаний; так что каждый дизъюнкт D_i имеет вид $X_{i1} \vee X_{i2} \vee \ldots \vee X_{ik_i}$, где символ X_{ij}, называемый ***литералом***, обозначает переменную или отрицание переменной, стоящую в i-ом дизъюнкте на j-ом месте.

Построим неориентированный граф $G = (V; E)$, вершинами которого служат пары целых чисел $[i, j]$ для всех $1 \le i \le m$ и $1 \le j \le k_i$. Первая компонента такой пары представляет дизъюнкт, а вторая – литерал, входящий в него. Таким образом каждая вершина графа естественным образом соответствует конкретному вхождению в конкретный дизъюнкт.

Вершинами графа G будут служить такие пары $([i, j], [k, l])$, для которых $i \ne j$ и $X_{ij} \ne X_{kl}$. Неформально, это означает, что вершины $[i, j]$ и $[k, l]$ смежны в G, если они соответствуют различным дизъюнктам и можно так присвоить значения переменным из литералов X_{ij} и X_{kl}, что оба литерала примут значение 1 (тем самым давая значение 1 своим дизъюнктам D_i и D_k). Иными словами, либо $X_{ij} = X_{kl}$, либо переменные, входящие в литералы X_{ij} и X_{kl}, различны.

Число вершин в G, очевидно, меньше длины формулы F, а число рёбер не превосходит квадрата числа вершин. Поэтому граф G можно закодировать в виде цепочки, длина которой ограничена полиномом от длины формулы F, и, что ещё важнее, такой код можно найти за время, ограниченное полиномом от длины формулы F.

Докажем, что G содержит клику мощности m тогда и только тогда, когда формула F выполнима. Отсюда будет следовать, что по данному алгоритму, решающему задачу о клике за полиномиально ограниченное время, можно построить алгоритм с полиномиально ограниченным временем работы для задачи из проблемы ВЫПОЛНИМОСТЬ, построив по формуле F с m дизъюнктами граф G, содержащий m-клику тогда и только тогда, когда формула F выполнима.

Итак, покажем, что *для того, чтобы построенный по формуле F граф G содержал m-клику, необходимо и достаточно, чтобы формула F была выполнима.*

Необходимость. Пусть граф G содержит клику мощности m. Первые компоненты вершин, составляющих такую клику, должны быть различны, поскольку вершины с одинаковыми первыми компонентами не соединяются рёбрами. Так как в этой клике точно m вершин, то вершины клики взаимно однозначно соответствуют

дизъюнктам формулы F. Пусть вершины клики имеют вид $[i, m_i]$: $1 \leq i \leq m$. Пусть

$$S_1 = \{X : X_{im_i} = X, \ 1 \leq i \leq m, \ X \in \{X_1, X_2, ..., X_n\}\} \text{ и}$$

$$S_2 = \{X : X_{im_i} = \neg X, \ 1 \leq i \leq m, \ X \in \{X_1, X_2, ..., X_n\}\}.$$

Иными словами, S_1 и S_2 – множества переменных и отрицаний переменных соответственно, представленных вершинами клики. Тогда $S_1 \cap S_2 = \emptyset$, ибо в противном случае какие-то две вершины $[s, m_s]$ и $[t, m_t]$, для которых $X_{sm_s} = \neg X_{tm_t}$, соединялись бы ребром. Если положить переменные из S_1 равными 1, а переменные из S_2 равными 0, то каждый дизъюнкт D_i примет значение 1, а значит и вся формула F, являющаяся их конъюнкцией, также примет значение 1. Поэтому формула F выполнима.

Достаточность. Пусть формула F выполнима. Тогда существует двоичный набор значений пропозициональных переменных, состоящий из нулей и единиц, при котором $F = 1$. При этом наборе каждый дизъюнкт формулы F также принимает значение 1. Следовательно, каждый дизъюнкт D_i содержит по меньшей мере один литерал, принимающий значение 1. Пусть в D_i таким литералом будет X_{im_i}: $X_{im_i} = 1$.

Мы утверждаем, что множество вершин $\{[i, m_i] : 1 \leq i \leq m\}$ образует в графе G клику мощности m. Если бы это было не так, то нашлись бы такие i и j, что $i \neq j$ и вершины $[i, m_i]$ и $[j, m_j]$ не соединены ребром. Отсюда следовало бы, что $X_{im_i} = X_{jm_j}$ (по определению множества рёбер графа G). Но это невозможно, поскольку $X_{im_i} = X_{jm_j} = 1$ в силу выбора литералов X_{im_i} и X_{jm_j} в дизъюнктах D_i и D_j соответственно. □

Пример 8.3.21. Рассмотрим формулу

$$F(X_1, X_2, X_3) \equiv (X_1 \vee \neg X_2) \wedge (X_2 \vee \neg X_3) \wedge (X_3 \vee \neg X_1).$$

Её литералы по дизъюнктам таковы:

$$X_{11} = X_1, X_{12} = \neg X_2; \ X_{21} = X_2, X_{22} = \neg X_3; \ X_{31} = X_3, X_{32} = \neg X_1.$$

Конструкция из теоремы 8.3.20 даёт следующий граф (изобразите этот граф графически):

{ ([1, 1], [2, 1]), ([1, 1], [3, 1]), ([1, 1], [2, 2]),
([1, 2], [2, 2]), ([1, 2], [3, 1]), ([1, 2], [3, 2]),
([2, 1], [3, 1]), ([2, 1], [3, 2]), ([2, 2], [3, 2]) } .

Например, узел [1, 1] не соединён с узлом [1, 2], потому что первые компоненты одинаковы, и не соединён с [3, 2], потому что $X_{11} = X_1$ и $X_{32} = \neg X_1$, а с остальными тремя узлами соединён.

В формуле F три дизъюнкта, и, оказывается, что в построенном графе две клики мощности 3, а именно $\{[1,1], [2,1], [3,1]\}$ и $\{[1,2], [2,2], [3,2]\}$. В первой клике представлены три литерала X_1, X_2, X_3. Это переменные без отрицаний, и первая клика соответствует присвоению значений $X_1 = X_2 = X_3 = 1$, выполняющему формулу F. Вторая клика соответствует другому набору значений, обращающему формулу F в 1, а именно $X_1 = X_2 = X_3 = 0$.

В книге [33], стр. 429, приводится ещё ряд NP-полных массовых проблем, доказательство NP-полноты которых осуществляется методом сведения, начиная с основополагающей проблемы ВЫПОЛНИМОСТЬ.

Приведём формулировки этих проблем.

УЗЕЛЬНОЕ ПОКРЫТИЕ (УП). Имеет ли данный граф узельное покрытие мощности k? (***Узельным покрытием*** неориентированного графа $G = (V; E)$ называется такое подмножество $S \subseteq V$, что каждое ребро графа G инцидентно некоторой вершине (узлу) из S).

РАСКРАШИВАЕМОСТЬ (Р). Является ли данный граф k-раскрашиваемым? (Неориентированный граф $G = (V; E)$ называется k-***раскрашиваемым***, если существует такое приписывание целых чисел 1, 2, ... , k, называемых ***цветами***, вершинам графа G, что никаким двум смежным вершинам не приписан один и тот же цвет. ***Хроматическим числом*** графа G называется такое наименьшее число k, что граф G k-раскрашиваем.)

МНОЖЕСТВО УЗЛОВ, РАЗРЕЗАЮЩИХ ЦИКЛЫ (МУРЦ). Имеет ли данный ориентированный граф k-элементное множество узлов, разрезающих циклы? (В ориентированном графе $G = (V; E)$ ***множеством узлов, разрезающих циклы***, называется такое подмножество $S \subseteq V$, что каждый цикл в G содержит узел из S.)

МНОЖЕСТВО РЁБЕР, РАЗРЕЗАЮЩИХ ЦИКЛЫ (МРРЦ). Имеет ли данный ориентированный граф k-элементное множество рёбер, разрезающих циклы? (В ориентированном графе $G = (V; E)$ ***множеством рёбер, разрезающих циклы***, называется такое подмножество $F \subseteq E$, что каждый цикл в G содержит ребро из F.)

Проблема ГАМИЛЬТОНОВ ЦИКЛ (ГЦ) для неориентированных графов была сформулирована выше. Проблема ОГЦ (ОРИЕН-

ТИРОВАННЫЙ ГАМИЛЬТОНОВ ЦИКЛ) формулируется аналогично, но для ориентированных графов.

ПОКРЫТИЕ МНОЖЕСТВАМИ (ПМ). Существует ли для данного семейства множеств $S_1, S_2, \ldots, S_n$ такое подсемейство из k множеств $S_{i_1}, S_{i_2}, \ldots, S_{i_k}$, что

$$\cup_{j=1}^{k} S_{i_j} = \cup_{j=1}^{n} S_j \ ?$$

ТОЧНОЕ ПОКРЫТИЕ (ТП). Существует ли для данного семейства множеств $S_1, S_2, \ldots, S_n$ подсемейство попарно непересекающихся множеств, являющееся покрытием ?

О частных случаях NP-полных массовых проблем. Вполне естественным и очевидным априорным соображением является то, что чем в более общей формулировке рассматривается та или иная задача, тем труднее её решить. И наоборот, частные случаи не поддающейся решению общей задачи могут оказаться достаточно легко решаемыми. В этом пункте проиллюстрируем эти соображения применительно к теории NP-полных задач массовых проблем.

Основная мысль состоит в том, что некоторые частные случаи NP-полных проблем *не обязаны быть трудными*, в частности, не обязаны быть NP-полными. Это соображение может быть очень важным на практике. Предположим, что в практической ситуации нас интересует получение точных оптимальных решений для данной комбинаторной задачи оптимизации. К сожалению, мы вскоре осознаём, что эта задача NP-полна и, следовательно, нет надежды решить общую задачу эффективно. Должны ли мы сдаться ? Не сразу. Возможно (на самом деле вероятно), что мы оказались жертвами ненужной общности – подобно многим исследователям, которые формулируют каждую дискретную задачу оптимизации в виде задачи целочисленного программирования, а каждую задачу упорядочения в виде проблемы КОММИВОЯЖЕР лишь для того, чтобы отказаться от решения, как только они поймут, что для этих общих задач трудно найти точное решение. Гораздо лучше формулировать задачи в наименее общем виде и пытаться использовать любые специальные свойства интересующих нас индивидуальных задач.

Например, если мы рассматриваем задачу о маршрутах, в которой участвуют графы, то может оказаться, что интересующие нас графы обладают некоторыми приятными свойствами, такими, как планарность, ограниченные степени вершин и т.д. С другой стороны, при доказательстве NP-полноты проблемы, возможно, использовались – что часто бывает – сведения (ударение на второй слог), в

которых строятся графы, сильно непланарные и имеющие большие степени. Поэтому остаётся надежда, что существует эффективный алгоритм для решения данной задачи в частном случае, когда граф планарен и степени малы.

Конечно, нельзя сказать, что NP-полнота общей задачи в этом случае не имеет никакого значения. Если такой результат доказан, то дело оптимистов – объяснить, как они надеются решить частный случай, используя его свойства.

ПРИМЕР 8.3.22. Вспомним проблему КЛИКА, которая, как мы только что доказали (теорема 8.3.20), NP-полна. Предположим, что мы рассматриваем проблему ПЛАНАРНАЯ КЛИКА, т.е. её ограничение на планарные графы. Напомним, что неориентированный граф называется ***планарным***, если его так можно нарисовать на плоскости, что никакие два его ребра не пересекаются. Теорема Понтрягина-Куратовского характеризует планарные графы как такие, которые не содержат в качестве подграфов двух вполне связанных графов – пятиэлементного графа K_5 и шестиэлементного графа $K_{3,3}$ (см., например[3], с. 231, или[4], с. 133). Это означает, что в планарном графе не может быть клик с пятью или более вершинами. Следовательно, максимальная клика в планарном графе $G = (V; E)$ может иметь не более четырёх вершин, и поэтому её можно найти полным перебором за время $O(|V|^4)$. На самом деле возможен даже алгоритм с оценкой $O(|V|)$ (см. задачу 5). Таким образом, проблему ПЛАНАРНАЯ КЛИКА действительно является полиномиальным частным случаем NP-полной проблемы КЛИКА и принадлежит классу **P** .

Тем не менее, не следует думать, что всякий частный случай трудной задачи приводит к задаче легко решаемой. Применительно к теории NP-полноты можно увидеть, что некоторые NP-полные проблемы продолжают оставаться NP-полными даже тогда, когда соответствующие им единичные задачи существенно ограничены.

Очень часто наиболее интересные вопросы, касающиеся NP-полноты, включают в себя точное понимание того, в каких частных случаях рассматриваемой NP-полной задачи содержится сложность этой задачи. Доказательство того, что некоторый частный случай NP-полной задачи сам является NP-полным, обычно включает в себя преобразование специального вида. Цель такого преобразования – модифицировать произвольную данную индивидуальную задачу

---

[3] *Берж К.* Теория графов и её применения. – М.: ИЛ, 1962.

[4] *Харари Ф.* Теория графов. – М.: Мир, 1973.

для общей задачи так, чтобы избавиться от особенностей, недопустимых в рассматриваемом частном случае, и не изменить ответа ДА или НЕТ в этой индивидуальной задаче. Мы уже встречались с таким преобразованием при доказательстве NP-полноты задачи 3-ВЫПОЛНИМОСТЬ (теорема 8.3.19), когда наша цель состояла в том, чтобы заменить дизъюнкты с числом литералов, отличным от трёх, эквивалентными множествами дизъюнктов с тремя литералами. Приведём другой пример такого доказательства.

Пример 8.3.23. *Проблема* ВЫПОЛНИМОСТЬ *остаётся* NP-*полной даже для формул, в которых каждая переменная появляется один или два раза без отрицания и один раз с отрицанием.*

Покажем это. Рассмотрим произвольную формулу F и произвольную переменную X, встречающуюся в F. Пусть в целом в F переменная X входит $k > 3$ раз. Вместо первого вхождения переменной X можно подставить новую переменную X_1 вместо второго вхождения – переменную X_2 и т.д. вплоть до k-го вхождения. Теперь надо обеспечить, чтобы в любом наборе значений истинности, выполняющем формулу F, значения истинности всех k переменных $X_1, X_2, \dots, X_k$ были одинаковы. Этого можно добиться, добавляя к формуле дизъюнкты

$$(X_1 \vee \neg X_2) \ \wedge \ (X_2 \vee \neg X_3) \ \wedge \ \dots \ \wedge (X_k \vee \neg X_1) \ .$$

Нетрудно проверить, что эти новые дизъюнкты выполняются (принимают значение 1) только тогда, когда все переменные $X_1, X_2, \dots$, X_k принимают одно и то же значение истинности.

Если мы проделаем это преобразование для всех переменных, появляющихся в формуле более трёх раз, то получим новую формулу F', в которой все переменные встречаются не более трёх раз, и эта формула будет выполнима тогда и только тогда, когда выполнима формула F. Выделим теперь все переменные, которые входят в формулу либо только без отрицания, либо только с отрицанием. Очевидно, при любых попытках выполнить F' мы можем выполнить все дизъюнкты, в которых эти переменные встречаются, придав им соответственно значения ИСТИНА или ЛОЖЬ, при этом не возникнет никаких противоречий. Поэтому все эти дизъюнкты можно удалить из F', и полученная формула опять будет выполнимой тогда и только тогда, когда выполнима формула F. Если в новой формуле некоторая переменная X встречается дважды с отрицанием и один раз без отрицания – это единственный оставшийся случай, когда нарушаются ограничения теоремы, – мы подставляем в формулу $\neg Y$ вместо X, где Y – новая переменная. Легко проверить, что в полу-

чаемой формуле каждая переменная встречается один или два раза без отрицания и один раз с отрицанием. □

Иногда для доказательства NP-полноты некоторого частного случая NP-полной задачи достаточно заметить, что все индивидуальные задачи, которые строятся при доказательстве NP-полноты общей задачи, лежат внутри интересующего нас частного случая.

В заключение отметим ещё два подобных результата о проблеме ГАМИЛЬТОНОВ ЦИКЛ, второй из которых сильнее первого:

а) Проблема ГАМИЛЬТОНОВ ЦИКЛ NP-полна даже тогда, когда рассматриваются только те графы, в которых степени вершин не превосходят четырёх.

б) Проблема ГАМИЛЬТОНОВ ЦИКЛ для графов, в которых степени всех вершин равны 3, NP-полна.

NP-полные проблемы и труднорешаемые проблемы. В параграфе 8.2 мы провели классификацию массовых проблем в зависимости от трудностей (времени) решения их (обыкновенным, или детерминированным) алгоритмом. По этой классификации "хорошо" или эффективно решаемые проблемы – это те, которые могут быть алгоритмически решены за полиномиальное время они образуют класс **P**, труднорешаемые – те, которые могут быть решены за экспоненциальное время и не могут быть решены никаким полиномиальным алгоритмом. Таким образом, здесь сравнивается время работы, одинаковых по своим качествам алгоритмов.

В настоящем параграфе мы произвели другую классификацию массовых проблем по их сложности, положив в основу этой классификации качества алгоритмов, считая, что работать они должны в течение реального времени, измеримого полиномом, зависящим лишь от размера решаемой единичной задачи. При такой классификации "хорошо" или эффективно решаемые проблемы снова образовали тот же класс **P**. А вот труднорешаемые проблемы в этой классификации образовали класс, который мы назвали классом NP-полных проблем. В этот класс отнесены самые сложные задачи из класса **NP**. Они характеризуются тем, что если нам всё-таки удастся найти полиномиальный алгоритм решения какой-либо из них, то отсюда последует, что все задачи из класса **NP** допускают решения алгоритмами полиномиальной сложности.

Возникает вполне естественный вопрос о взаимоотношениях между этими двумя классами трудно решаемых проблем – классом труднорешаемых проблем в смысле первой классификации и классом NP-полных проблем в смысле второй классификации. Первые при-

меры труднорешаемых проблем были найдены в 60-ых – начале 70-ых годов прошлого века. Некоторые из них оказались труднорешаемы не только с помощью детерминированных (т.е. обычных) алгоритмов, но и с помощью недетерминированного вычислительного устройства. Однако большинство представляющихся труднорешаемыми практических задач в действительности могут быть решены за полиномиальное время с помощью недетерминированного вычислительного устройства.

Одним из основных открытых вопросов современной математики и теоретической кибернетики является вопрос о том, действительно ли NP-полные проблемы труднорешаемы. В настоящее время ни для одной NP-полной проблемы не известен полиномиальный алгоритм для её решения (и это несмотря на настойчивые усилия многих блестящих исследователей в течении ряда десятилетий). Вопреки готовности большинства специалистов считать, что все NP-полные проблемы труднорешаемы, прогресс как в доказательстве, так и в опровержении этого далеко идущего предположения весьма незначителен. Тем не менее, несмотря на отсутствие доказательства того, что из NP-полноты следует труднорешаемость, NP-полнота проблемы означает, что для её решения полиномиальным алгоритмом требуется по крайней мере крупное открытие.

Глава IX

АЛГОРИТМИЧЕСКИЕ ПРОБЛЕМЫ МАТЕМАТИЧЕСКОЙ ЛОГИКИ

В параграфе 9.1 будут рассмотрены знаменитые теоремы К.Гёделя и А.Тарского о формальной арифметике, доказанные ими в начале 30-ых годов XX века, – о неполноте формальной арифметики, о невозможности доказать её непротиворечивость её собственными средствами и о невыразимости в языке формальной арифметики "арифметических истин". Эти теоремы не касаются непосредственно алгоритмических проблем математической логики, но доказываются с существенным использованием методов и результатов теории алгоритмов.

Параграф 9.2 посвящается уже непосредственно алгоритмической проблеме математической логики, которая была решена первой из подобных проблем – проблеме разрешимости формализованного исчисления предикатов. Её отрицательное решение было получено американским логиком А.Чёрчем в 1936 г.

Здесь мы от термина "массовая проблема" снова возвращаемся к термину "алгоритмическая проблема", поскольку с исторической точки зрения он здесь более употребителен.

9.1. Теоремы К.Гёделя и А.Тарского о формальной арифметике

Начнём с первой теоремы К.Гёделя, доказанной им в 1931 г. – теореме о неполноте формальной арифметики, утверждающей, что любая непротиворечивая формальная аксиоматическая теория, формализующая арифметику натуральных чисел, не являются (абсолютно) полной. В настоящем параграфе даётся доказательство этой теоремы, опирающееся на идеи и методы теории алгоритмов. Тем самым будет ещё раз продемонстрирована на самом высоком уровне теснейшая связь математической логики и теории алгоритмов – двух математических дисциплин, образующих фундамент всей

современной математики. Наше изложение будет основываться на доказательстве, разработанном М.Арбибом [32].

После доказательства теоремы 6.3.4 о том, что существует перечислимое, но не разрешимое множество натуральных чисел, было заявлено, что она фактически включает в себя в неявном виде теорему Гёделя о неполноте формальной арифметики. Цель настоящего параграфа состоит в том, чтобы обосновать это заявление. Таким образом, в рамках общей теории алгоритмов, кроме тех теорем, которые были доказаны в двух предыдущих параграфах, будет продемонстрировано продвижение теории алгоритмов в направлении решения чисто логических проблем. Для этого сначала предстоит увязать терминологию логической проблемы о неполноте формальной арифметики с методологической терминологией общей теории алгоритмов, методами которой эта проблема будет решена. При этом, утверждение теоремы 6.3.4 о существовании перечислимого, но неразрешимого множества натуральных чисел будет основополагающей предпосылкой для доказательства теоремы Гёделя, которую мы докажем в следующей формулировке: "*Каждая адекватная ω-непротиворечивая формальная арифметика неполна.*"

Далее, мы поясним, что будем понимать под формальной арифметикой, а также определим и разъясним те понятия, которые участвуют в приведённой формулировке теоремы Гёделя. Начнём с формальных аксиоматических теорий.

Формальные аксиоматические теории и натуральные числа. В курсе математической логики определяется понятие *формальной аксиоматической теории* (см. первый пункт §28 в Учебнике МЛ). Чтобы задать такую теорию T, нужно задать *алфавит* (счётное множество символов); в множестве всех слов, составленных из букв данного алфавита, выделить подмножество, элементы которого будут называться *формулами* (или правильно построенными выражениями) данной теории; в множестве формул выделить те, которые будут называться *аксиомами* теории; наконец, должно быть заданно конечное число отношений между формулами, называемых *правилами вывода.* При этом, должны существовать эффективные процедуры (алгоритмы) для определения того, являются ли данные слова (выражения) формулами (т.е. правильно построенными выражениями), являются ли данные формулы аксиомами, и наконец получается ли одна данная формула из ряда других данных формул с помощью данного правила вывода. Это означает, что множество всех формул разрешимо и множество всех аксиом разрешимо. Следовательно, каждое из этих множеств перечислимо.

Понятия *вывода* и *теоремы* в формальной аксиоматической теории определены в определении 28.2 указанного учебника.

Все теоремы, приводимые в настоящем параграфе, в соответствии с нашей терминологией является фактически метатеоремами т.е. теоремами о свойствах (формальных) аксиоматических теорий. Но поскольку здесь никакой конкретной аксиоматической теории мы не рассматриваем, никаких теорем такой теории не доказываем, т.е. никаких теорем, кроме метатеорем, здесь не будет, то мы метатеоремы будем называть просто теоремами.

Теорема 9.1.1. *Множество $Th(T)$ всех теорем формальной аксиоматической теории T перечислимо.*

Доказательство. Мы уже отметили, что множество аксиом формальной теории перечислимо, т.е. мы можем их эффективно перенумеровать A_1, A_2, A_3, ... Поскольку все формулы состоят из конечного числа букв (символов), все выводы содержат конечное число формул и каждый вывод использует лишь конечное число аксиом, то ясно, что для каждого натурального n существует лишь конечное число выводов, имеющих не более, чем n формул (шагов) и использующих только аксиомы $\{A_1, A_2, ..., A_n\}$. Следовательно, двигаясь от $n = 1$ к $n = 2$, $n = 3$ и т.д., можно эффективно перенумеровать все теоремы данной теории. Это и означает, что множество теорем перечислимо. □

Теперь от произвольных формальных теорий будем переходить к таким, которые так или иначе имеют дело с натуральными числами. Если мы хотим в нашей теории говорить о подмножестве Q множества натуральных чисел, то мы должны иметь эффективный способ выписывания для каждого натурального n формулы H_n, означающей, что $n \in Q$. Более того, если мы сможем доказать, что формула H_n является теоремой теории T тогда и только тогда, когда $n \in Q$, то будем говорить, что теория T *полуполна для* Q (или что в T имеется *полуполное описание* Q). Точнее, это определение сформулируем так.

Определение 9.1.2. Теория T называется *полуполной для множества натуральных чисел* $Q \subseteq N$, если существует перечислимое множество формул H_0, H_1, ..., H_n, ... такое, что $Q = \{n : \vdash H_n\}$.

Определение 9.1.3. Далее, теория T называется *полной для* Q, если она полуполна для Q и мы также имеем формулу $\neg H_n$, которая интерпретируется как $n \notin Q$, и мы можем доказать, что $\neg H_n$

является теоремой теории T тогда и только тогда, когда $n \notin Q$. Другими словами, теория T полна для Q, если в T для каждого n мы можем установить принадлежит оно Q или нет. Точнее, это означает, что теория T называется полной для множества натуральных чисел Q, если она полуполна для Q и полуполна для его дополнения $\overline{Q}$.

ТЕОРЕМА 9.1.4. *Если теория T полуполна для множества Q, то Q перечислимо.*

ДОКАЗАТЕЛЬСТВО. По определению полуполноты T для Q, множество Q есть множество номеров тех формул из некоторого перечислимого множества $\{H_0, H_1, ..., H_n, ...\}$ формул, которые являются теоремами теории T, т.е. принадлежит и множеству $Th(T)$. Таким образом, Q есть множество номеров всех формул из множества $Th(T) \cap \{H_0, H_1, ..., H_n, ...\}$. Каждое из этих пересекаемых множеств перечислимо: первое по предыдущей теореме 9.1.1, второе – по сказанному в начале доказательства. Следовательно, и их пересечение, по теореме 6.2.2, перечислимо. Но тогда перечислимо и множество номеров тех формул, которые входят в это пересечение, т.е множество Q. □

СЛЕДСТВИЕ 9.1.5. *Если Q перечислимое, но не разрешимое множество натуральных чисел, то никакая формальная теория не может быть полной для Q.*

ДОКАЗАТЕЛЬСТВО. Если множество Q перечислимо, но не разрешимо, то в силу теоремы 6.3.2, его дополнение $\overline{Q}$ не перечислимо. Тогда по теореме 9.1.4, никакая теория T не является полуполной для $\overline{Q}$. Следовательно, никакая теория T не полна для Q. □

От этого следствия до теоремы Гёделя совсем недалеко. Для этого нужно средствами некоторой формальной теории T развить теорию натуральных чисел, причём так, чтобы принадлежность чисел данному множеству Q можно было трактовать адекватно (т.е. число n принадлежит Q тогда и только тогда, когда некоторая эффективно сопоставленная ему формула теории T является теоремой этой теории). Это возможно только тогда, когда Q по меньшей мере перечислимо.

Формальная арифметика и её свойства. Формальная арифметика как формальная аксиоматическая теория строится на базе формализованного исчисления предикатов, рассмотренного в §21 Учебника МЛ и §14 Задачника. Предметные переменные $x_1, x_2, x_3, ...$

здесь будем называть числовыми, потому что вместо них будем подставлять натуральные числа.

Предметная переменная называется *свободной* в формуле, если она не стоит под знаком квантора (общности или существования), и *связанной* в противном случае. Формула называется *замкнутой*, если все её предметные переменные связаны, и *открытой*, если в ней имеются свободные переменные. *Замыканием* формулы F называется формула $C(F)$, получающаяся из F дописыванием спереди кванторов общности по всем переменным, свободным в F. Ясно, что для любой формулы F формула $C(F)$ замкнута. Если F замкнута, то $C(F) = F$.

Функция $C(F)$ вычислима. Отсюда следует, что *класс замкнутых формул разрешим*, поскольку F ему принадлежит тогда и только тогда, когда $C(F) = F$ и для распознавания этого равенства существует вычислительная процедура.

С понятием подстановки в формулу мы уже знакомы (см. конец §3 Учебника МЛ. Если в формулу F вместо символа (слова) X везде, где он входит в F, вставить слово (формулу) H, то получим новое слово (формулу), обозначаемое $S_X^H F$ и называемое *результатом подстановки* в F слова H вместо слова X. Тогда ясно, что:

$$S_X^H(\neg F) \equiv \neg S_X^H F \quad ; \qquad S_X^H(F \to G) \equiv S_X^H F \to S_X^H G \quad ;$$

если $i \neq j$, то $S_{x_i}^H(\forall x_j)(F) \equiv (\forall x_j) S_{x_i}^H F$,

$$S_{x_i}^H(\exists x_j)(F) \equiv (\exists x_j) S_{x_i}^H F.$$

Имея дело с натуральными числами, мы хотели бы иметь возможность подставлять их в формулы нашей формальной теории (арифметики), т.е. иметь возможность говорить о числах на языке нашей формальной теории. Для этой цели в нашей формальной теории необходимо иметь слова, которые служили бы названиями натуральных чисел. Такие слова называются *нумералами*. Нумерал числа n обозначается n^*. Требование к этим названиям (именам) вполне естественное: различные числа должны называться различными именами, т.е. если $m \neq n$, то $m^* \neq n^*$. (Идея введения нумералов состоит в том, чтобы разделить вещи и имена этих вещей).

Теперь в формулы нашей теории мы будем подставлять вместо числовых переменных x_1, x_2, x_3, ... не сами натуральные числа $m, n, k, ...$, а их нумералы (имена) m^*, n^*, k^*, ... соответственно.

Наконец, мы можем сформулировать последнее требование (аксиому), которое мы предъявляем к формальной арифметике. Назо-

вём его ***аксиомой арифметики***: если предметная переменная x_i не связана в F, то

$$(\text{AA}): \quad S_{x_i}^{n^*} F \to (\exists x_i)(F) \quad .$$

Если ввести для $S_{x_i}^{n^*} F$ обозначение $F(n^*)$, то эта аксиома принимает вид:

$$(\text{AA}): \quad F(n^*) \to (\exists x)(F(x)) \quad .$$

Это исключительно естественное требование: если формула F превращается в истинное высказывание при подстановке в неё вместо переменной x_i какого-нибудь натурального числа n^*, то истинно и высказывание $(\exists x_i)(F)$.

Никаких других ограничений на формализацию арифметики не накладывается. Не важно, в частности, как определяется сложение и умножение натуральных чисел, как вводится отношение порядка, чем мы скрупулёзно занимались при построении теории натуральных чисел на основе системы аксиом Пеано. Даже при таких самых общих допущениях на формализацию арифметики эта формализация будет подчиняться теореме Гёделя: если она будет непротиворечива, то она будет неполной.

Итак, определившись с понятием формальной арифметики, посвятим оставшуюся часть этого пункта понятиям ω-непротиворечивости, адекватности и полноты этой формальной теории, участвующим в точной формулировке теоремы Гёделя.

Начнём с понятия непротиворечивости. Как и всякая аксиоматическая теория, формальная арифметика называется непротиворечивой, если в ней нельзя доказать какое-либо утверждение и его отрицание, т.е. если не существует такой формулы F, что одновременно $\vdash F$ и $\vdash \neg F$.

Предположим теперь, что для некоторой формулы $G(x)$, содержащей свободно единственную предметную переменную x, установлено, что $\vdash G(n^*)$ для всех натуральных чисел $n = 0, 1, 2, 3, \ldots$. Даже если в формальной арифметике невозможно доказать $\vdash (\forall x)(G(x))$, мы конечно же, можем считать это утверждение следствием приведённого списка теорем. Следовательно, если в теории удастся доказать теорему $\vdash \neg(\forall x)(G(x))$, то такую формальную арифметику следует считать противоречивой.

ОПРЕДЕЛЕНИЕ 9.1.6. Формальная арифметика называется *ω-непротиворечивой*, если в ней нет такой формулы $G(x)$ с единственной свободной предметной переменной x, что для всех натуральных чисел n справедливы теоремы $\vdash G(n^*)$ и $\vdash \neg(\forall x)(G(x))$.

Теорема 9.1.7. *Если формальная арифметика ω-непротиворечива, то она непротиворечива.*

Доказательство. В самом деле, если бы она была противоречива, то, как доказано в §27 Учебника МЛ (после определения 27.1), все её формулы были бы теоремами, в том числе и те, которые создают ω-противоречивость формальной арифметики, и последняя была бы ω-противоречива. □

Определение 9.1.8. Назовём n-местный предикат $P(x_1, ..., x_n)$ над множеством натуральных чисел N ***вполне представимым в формальной арифметике***, если существует такая формула $F(x_1, ..., x_n)$, свободными предметными переменными которой являются n переменных x_1, ... , x_n и только они, что:

а) для каждой n-ки натуральных чисел $(a_1, ..., a_n)$, для которой предикат P превращается в истинное высказывание $P(a_1, ..., a_n)$, имеет место теорема: $\vdash F(a_1^*, ..., a_n^*)$;

б) для каждой n-ки натуральных чисел $(a_1, ..., a_n)$, для которой предикат P превращается в ложное высказывание $P(a_1, ..., a_n)$, имеет место теорема: $\vdash \neg F(a_1^*, ..., a_n^*)$;

Таким образом, вполне представимость предиката в формальной арифметике означает, что мы средствами этой формальной теории всегда можем решить, превратится он в истинное или ложное высказывание при подстановке вместо всех его предметных переменных тех или иных натуральных чисел.

Разъясним теперь понятие адекватности формальной арифметики, участвующее в формулировке теоремы Гёделя. Мы хотели бы иметь возможность отвечать на вопросы о перечислимых множествах в такой арифметике. В теореме 9.1.4 мы показали, что лишь перечислимые множества чисел могут иметь полуполное описание в формальной теории, т.е. существует перечислимое множество формул H_0, H_1, H_2, ... такое, что $Q = \{n : \vdash H_n\}$. Адекватность нашей формальной теории (арифметики) могла бы означать, что она является полуполной для каждого перечислимого множества натуральных чисел, т.е. что в ней имеет полуполное описание всякое множество, которое вообще может иметь такое описание хотя бы в какой-нибудь теории.

В теореме 9.1.1 мы установили, что множество всех теорем формальной теории перечислимо, т.е. все теоремы и, значит, приводящие к ним выводы (доказательства) могут быть эффективно перенумерованы. Возьмём наше множество Q и соответствующее ему множество теорем $\{H_0, H_1, H_2, ...\}$. Рассмотрим следующий пре-

дикат $P(x,y)$: "y – номер доказательства теоремы H_x". Если высказывание $P(m,n)$ истинно, то это означает, что n есть номер вывода теоремы H_m, что, в свою очередь, означает, что $m \in Q$, т.е. n есть номер вывода о том, что $m \in Q$. Обратно, взяв конкретные числа m и n, мы можем эффективно построить теорему (формулу) H_m и эффективно построить n-ый вывод, после чего эффективно определить, является ли построенный вывод выводом теоремы H_m, т.е. эффективно узнать, истинно ли высказывание $P(m,n)$. Следовательно, $P(x,y)$ – такой вычислимый предикат, что $Q = \{x : (\exists y)(\lambda[P(x,y)] = 1)\}$.

Сформулируем теперь определение.

ОПРЕДЕЛЕНИЕ 9.1.9. Формальная арифметика называется *адекватной*, если для каждого перечислимого множества Q натуральных чисел существует вполне представимый в этой арифметике предикат $P(x,y)$ такой, что $Q = \{x : (\exists y)(\lambda[P(x,y)] = 1)\}$.

Под *полнотой* формальной арифметики будем понимать абсолютную полноту, т.е. если для каждой замкнутой формулы F этой теории либо она сама, либо её отрицание является теоремой этой теории: $\vdash F$ или $\vdash \neg F$.

Теперь мы можем проделать

Доказательство теоремы Гёделя о неполноте. Теорема утверждает следующее. ***Всякая ω-непротиворечивая и адекватная формальная арифметика не является полной.***

Докажем её. Согласно теореме 6.3.4, выберем такое множество Q натуральных чисел, которое перечислимо, но неразрешимо. Так как наша формальная арифметика адекватна, то существует вполне представимый в ней предикат $P(x,y)$ такой, что

$$Q = \{x : (\exists y)(\lambda[P(x,y)] = 1)\}. \qquad (*)$$

Вполне представимость предиката $P(x,y)$ в формальной арифметике означает, что найдётся такая формула $F(x,y)$ этой теории, содержащая лишь две свободных предметных переменных, что для каждой пары натуральных чисел (a,b), для которой $\lambda[P(a,b)] = 1$, имеет место теорема: $\vdash F(a^*,b^*)$, а для каждой пары натуральных чисел (a,b), для которой $\lambda[P(a,b)] = 0$, имеет место теорема: $\vdash \neg F(a^*,b^*)$.

Применим к формуле $F(x,y)$ квантор общности по переменной y. Получим формулу с единственной свободной предметной переменной x : $G(x) \equiv (\exists y)(F(x,y))$. Покажем, что

$$Q = \{x : \vdash G(x^*)\} \qquad (**)$$

Предположим, что $m \in Q$. Тогда (согласно (*)) найдётся такое натуральное n, что высказывание $P(m, n)$ истинно. Следовательно, имеет место теорема: $\vdash F(m^*, n^*)$. В силу аксиомы арифметики (AA) имеем теорему:

$$\vdash F(m^*, n^*) \to (\exists y)(F(m^*, y)).$$

Из двух последних теорем по правилу MP заключаем:

$$\vdash (\exists y)(F(m^*, y)), \text{ т.е. } \vdash G(m^*).$$

Это означает, что $m \in \{x : \vdash G(x^*)\}$. Таким образом, $Q \subseteq \{x : \vdash G(x^*)\}$.

Обратно, предположим, что $m \in \{x : \vdash G(x^*)\}$, т.е. $\vdash G(m^*)$, т.е. $\vdash (\exists y)(F(m^*, y))$. Отсюда в силу известного выражения (по закону де Моргана) квантора существования через квантор общности заключаем, что

$$\vdash \neg(\forall y)(\neg F(m^*, y)).$$

Поскольку наша формальная арифметика, кроме того, ω-непротиворечива то, ввиду наличия в ней последней теоремы, должно существовать такое натуральное число n_0, что формула $\neg F(m^*, n_0^*)$ не является теоремой этой арифметики. А раз так, то высказывание $P(m, n_0)$ истинно (если бы оно было ложно, то мы имели бы теорему $\vdash \neg F(m^*, n_0^*)$, что не так). По определению (*) множества Q, это означает, что $m \in Q$. Таким образом, $\{x : \vdash G(x^*)\} \subseteq Q$.

Итак, равенство (**) доказано.

Теперь выясним, в каком отношении находятся между собой множества $\overline{Q}$ (дополнение Q) и $\{x : \vdash \neg G(x^*)\}$. Пусть $m \in \{x : \vdash \neg G(x^*)\}$, т.е. $\vdash \neg G(m^*)$. Тогда $m \in \overline{Q}$, ибо если бы $m \in Q$, то, в силу (**), мы имели бы $\vdash G(m^*)$ и наша формальная арифметика была бы противоречивой, но это не так в силу её ω-непротиворечивости (по условию) и теоремы 9.1.7. Таким образом, $\{x : \vdash \neg G(x^*)\} \subseteq \overline{Q}$.

Покажем, что последнее включение является строгим. Напомним, что мы выбрали множество Q перечислимым, но не разрешимым. Тогда согласно следствию 9.1.5, никакая формальная теория не может быть полной для Q. Равенство (**) говорит, что наша формальная арифметика полуполна для Q. Если бы имело место равенство $\overline{Q} = \{x : \vdash \neg G(x^*)\}$, то это означало бы, что наша формальная арифметика полуполна и для $\overline{Q}$ и, значит, она полна для Q. Последнее невозможно в силу следствия 9.1.5. Следовательно, $\{x : \vdash \neg G(x^*)\} \neq \overline{Q}$.

Итак, $\{x : \vdash \neg G(x^*)\} \subset \overline{Q}$. Следовательно, существует такое число $m_0 \in \overline{Q}$, что $m_0 \notin \{x : \vdash \neg G(x^*)\}$, т.е. неверно, что $\vdash$

$\neg G(m_0^*)$. В то же время неверно также, что $\vdash G(m_0^*)$, поскольку это, в силу (**), означало бы, что $m_0 \in Q$, что не так. Следовательно, мы нашли формулу $G(m_0^*)$ такую, что ни она сама, ни её отрицание $\neg G(m_0^*)$ не являются теоремами нашей формальной арифметики. Это и означает, что данная формальная арифметика не полна.

Теорема Гёделя полностью доказана. □

Обратимся ещё раз к высказыванию $\neg G(m_0^*)$. Согласно равенству (**) его можно интерпретировать как $m_0 \in \overline{Q}$ и, следовательно, оно обязательно является "истинным" высказыванием. Но тем не менее, оно не является теоремой нашей формальной арифметики. Если добавить формулу $G(m_0^*)$ к списку аксиом и рассмотреть новую формальную арифметику, то положение не изменится: для вновь полученной формальной арифметики верны все те предпосылки, которые привели нас к теореме Гёделя. Значит, мы снова найдём такое число m_1, что высказывание $\neg G(m_1^*)$ истинно, но не является теоремой новой формальной арифметики, и т.д.

Ещё один взгляд на теорему Гёделя о неполноте. Обратимся ещё раз к началу доказательства теоремы Гёделя. Оно начинается с выбора множества Q натуральных чисел, которое перечислимо но не разрешимо, и предиката $P(x, y)$, который это множество определённым образом выделяет. Этим объектам можно придать более наглядное содержание и тем самым лучше понять подлинную причину неполноты формальной арифметики.

Проинтерпретируем предикат $P(x, y)$ следующим образом. В параграфе 7.1 мы установили эффективную нумерацию машин Тьюринга, в которой каждая машина Тьюринга M получила номер x, по которому она может быть однозначно восстановлена – M_x. Введём тогда следующий предикат:

$M_x(x, y)$: *"Машина Тьюринга M_x с номером x при подаче на её вход слова x остановится через y шагов"*.

Тогда рассматриваемое в теореме множество Q принимает вид: $Q = \{x : (\exists y)[M_x(x, y)]\}$ – это есть множество номеров x тех машин Тьюринга, для которых найдётся число y, представляющее собой число шагов, через которое машина Тьюринга с номером x остановится, если на её вход будет подан её собственный номер x. Другими словами, Q – множество номеров всех самоприменимых машин Тьюринга.

Вспомним теперь теорему 6.4.3 о перечислимости, но неразрешимости множества всех номеров вычислимых функций, номера которых входят в область их определения. Если под вычислимой функ-

цией понимать функцию, вычислимую по Тьюрингу, то мы приходим к утверждению о том, что множество номеров всех самоприменимых машин Тьюринга, т.е. множество Q, перечислимо но неразрешимо, а его дополнение $\overline{Q}$ неперечислимо и неразрешимо. Последнее означает, что элементы множества $\overline{Q}$ в каком-то смысле оказываются неуловимыми: у нас нет никакого метода, с помощью которого можно было бы распознать и различить элементы множества $\overline{Q}$. Мы не можем ни определить, какие числа принадлежат $\overline{Q}$, а какие – нет (это следует из неразрешимости множества $\overline{Q}$), ни вычислить по порядку все элементы множества $\overline{Q}$ с помощью какой-либо порождающей процедуры (это следует из неперечислимости множества $\overline{Q}$).

Но в множестве $\overline{Q}$ нам удаётся выделить некоторое собственное (т.е. не совпадающее со всем $\overline{Q}$) подмножество $\{x : \vdash \neg G(x^*)\}$, состоящее из таких чисел (номеров) x, для которых доказуема формула $\neg G(x^*)$, где $G(x)$ может быть интерпретирована как формула $(\exists y)[M_x(x,y)]$. Поскольку это подмножество не совпадает со всем $\overline{Q}$), поэтому в $\overline{Q}$) имеется такой элемент $m_0 \in \overline{Q}$), который этому подмножеству не принадлежит, т.е. для которого формула $\neg G(x^*)$ не доказуема. Но поскольку $m_0 \notin Q$, а Q охарактеризовано ранее (см. равенство (**)) как множество, состоящее из таких чисел (номеров) x, для которых доказуема формула $G(x^*)$, поэтому формула $G(m_0^*)$ не доказуема.

Итак, мы обнаруживаем такую формулу нашей формальной арифметики $G(m_0^*)$, что ни она сама, ни её отрицание $\neg G(m_0^*)$ не доказуемы в формальной теории. Это и означает, что формальная теория не является полной, а причина этой неполноты, как мы видели, кроется в существовании множеств, которые одновременно являются неперечислимыми и неразрешимыми.

Вторая теорема К.Гёделя утверждает, что непротиворечивость формальной арифметики как аксиоматической теории не может быть доказана средствами самой этой аксиоматической теории. Идея её доказательства состоит в следующем. Высказывание, выражающее непротиворечивость формальной арифметики, формулируется так: "*Не существуют две такие формулы, что одна является отрицанием другой и обе доказуемы*". Это высказывание можно закодировать в виде некоторой формулы языка формальной арифметики; обозначим эту формулу через H.

Далее, недоказуемую и неопровержимую формулу $\neg G(m_0^*)$, рассмотренную в предыдущей теореме о неполноте формальной арифметики, можно интерпретировать в терминах машин Тьюринга как

$\neg(\exists y)(M_{m_0^*}(m_0^*, y))$, где $m_0 \in \overline{Q}$ (но не входит в множество $\{x : \vdash \neg G(x^*)\}$). Формула
$\neg(\exists y)(M_{m_0^*}(m_0^*, y))$ утверждает: "*Не существует такого натурального числа y, что машина Тьюринга с номером m_0, будучи применённой к своему собственному номеру m_0, остановится через y шагов.*" Если допустить, что такое значение y тем не менее найдётся, т.е. будет истинным высказывание $(\exists y)(M_{m_0^*}(m_0^*, y))$, то это будет означать, что $m_0 \in Q$. Получаем противоречие с тем, что $m_0 \in \overline{Q}$. Значит, если указанное y действительно существует, то наша формальная теория противоречива. Таким образом, можно записать

$$(\exists y)(M_{m_0^*}(m_0^*, y)) \quad \rightarrow \quad (\text{Арифметика противоречива}) \ ,$$

или что равносильно,

$$(\text{Арифметика непротиворечива}) \quad \rightarrow \quad \neg(\exists y)(M_{m_0^*}(m_0^*, y)) \ .$$

Переведём это истинное высказывание на формализованный язык. Посылку этой импликации мы закодировали с помощью формулы H, а следствие кодируется формулой $\neg G(m_0^*)$. В итоге получится формула $H \rightarrow \neg G(m_0^*)$.

Можно доказать, что формула $H \rightarrow \neg G(m_0^*)$ доказуема в формальной арифметике. Если теперь допустить, что непротиворечивость арифметики можно доказать её собственными средствами (т.е. в ней самой), т.е. в формальной арифметике доказана теорема $\vdash H$, то из теорем H и $H \rightarrow \neg G(m_0^*)$ по правилу Modus Ponens выводится теорема $\vdash \neg G(m_0^*)$, т.е. доказывается, что формула $\neg G(m_0^*)$ доказуема. Но ведь, как установлено в доказательстве предыдущей теоремы о неполноте формальной арифметики, формула $\neg G(m_0^*)$ недоказуема (как и её отрицание $G(m_0^*)$). Следовательно, допущение неверно, и значит, непротиворечивость формальной арифметики нельзя доказать средствами самой этой формальной теории.

Заканчивая обсуждение теорем Гёделя, отметим, что они верны не только для формальной арифметики, но и для любой более богатой формальной аксиоматической теории, содержащей формальную арифметику. В такой форме их можно назвать обобщёнными теоремами Гёделя.

Теорема А.Тарского. Эта теорема, доказанная в 1936 г., добавляет новые штрихи к характеристике формальной арифметики как аксиоматической теории. Она утверждает, что если формальная арифметика непротиворечива, то утверждение "*быть истинной*

формулой арифметики" невыразимо в языке формальной арифметики. Другими словами, множество истинных формул арифметики неописуемо средствами этой же формальной теории.

Методологическая сущность этой теоремы вполне понятна, и она осознавалась ещё философами античности: высказывание, произнесённое на естественном языке, не несёт никакой информации об истинностном значении этого высказывания; более того, утверждение об истинности того или иного высказывания, произнесённого на естественном языке, не может быть выражено в самом этом естественном языке. В самом деле, пусть A_0 есть некоторое высказывание, например: "*В день распятия Иисуса в Иерусалиме шёл дождь, гремел гром и сверкали молнии*" и пусть это событие действительно имело место. Но поскольку из содержания высказывания A_0 не следует, что оно истинно, поэтому необходимо дополнительное высказывание A_1: "*Высказывание A_0 истинно.*" В свою очередь, истинность высказывания A_1 также ниоткуда не следует. Поэтому необходимо новое высказывание A_2: "*Высказывание A_1 истинно*" – и так далее, до бесконечности. Получается, что понятие истинности высказывания естественного языка невыразимо средствами самого этого естественного языка.

Аналогичная ситуация имеется и с формальным языком арифметики: истинность факта содержательной арифметики невыразима в языке формальной арифметики. Понятие истинности выходит за рамки исследуемого предметного языка и относится уже к языку исследователя, или, как говорят, к метаязыку.

Теорема Гёделя о неполноте формальной арифметики с позиций программиста. В этом пункте представлен взгляд на теорему Гёделя о неполноте формальной арифметики и её доказательство с точки зрения программиста, изложенные известным советским математиком и кибернетиком одним из основоположников кибернетической науки в Советском Союзе директором созданного им в Киеве Института кибернетики Академии наук СССР академиком Виктором Михайловичем Глушковым [47].

Знаменитая теорема Гёделя о неполноте формальной арифметики принадлежит к числу наиболее сложных по доказательству во всей истории математики. Широко распространено мнение, что это доказательство доступно только специалистам в области математической логики. Однако внимательный анализ показывает, что особую сложность ему придают главным образом многочисленные технические подробности, связанные с использованием аппарата частично-рекурсивных функций. Что же касается базовых идей, лежа-

щих в основе доказательства, то они достаточно просты для любого человека, владеющего элементами современной математической культуры.

Здесь предлагается новый подход к доказательству теорем о неполноте формальных теорий, основанный на современной теории программирования. Благодаря этому доказательство очищается от технических подробностей – они заменяются апелляцией к *программистскому чувству очевидности*. Такой приём для доказательства теорем в чистой математике применяется, по-видимому, впервые, хотя сама по себе апелляция к очевидности другой природы используется постоянно. Например, в анализе принято считать очевидным существование функций с теми или иными свойствами (скажем, с заданными на некотором отрезке точками максимума и минимума) без явного выписывания этих функций. Для случая, рассматриваемого в данной работе, будет считаться очевидным, что существует программа с некоторыми заданными свойствами, если более или менее опытный программист сразу видит возможность её фактического построения. Как и в случае апелляции к формульно-функциональной очевидности, доказательства, апеллирующие к программистской очевидности, можно считать вполне строгими в том смысле, что при необходимости любой читатель-программист легко восстановит недостающие детали, фактически выписав все требуемые программы.

Перейдём теперь к постановке задач о полноте или неполноте формальных теорий на программистской основе. Введём определения для новых или видоизмененных понятий, а также напомним отдельные известные факты из математической логики, не связанные непосредственно с теорией программирования.

Первым понятием, на котором зиждутся все последующие построения, является понятие алгоритмической системы. Под *алгоритмической системой*, или сокращенно *A-системой*, будем понимать совокупность входного (алгоритмического) языка L и интерпретатора C, в качестве которого может выступать либо человек, снабжённый неограниченным запасом бумаги, либо (что будем принимать в дальнейшем) *идеализированная вычислительная машина* (ИВМ) с соответствующим матобеспечением. Входной язык строится на базе конечного алфавита, конечные последовательности букв которого служат для представления *данных* и (входных) *программ*.

Идеализированная ВМ имеет бесконечное оперативное запоминающее устройство (ОЗУ), благодаря чему отпадает необходимость во внешней памяти. Из периферийного оборудования ИВМ снаб-

жена лишь устройством ввода, способным прочитывать записи на входном языке, помещая их в последовательные ячейки ОЗУ, а также выводным печатающим устройством с неограниченным запасом бумаги, позволяющим выводить на печать все символы входного алфавита и знак пробела. Входной язык системы, а также система команд ИВМ и её математическое обеспечение в принципе могут быть любыми. Важно только, чтобы ИВМ могла осуществить автоматическую загрузку входной программы (после помещения её в неограниченно большой входной бункер вводного устройства) и её исполнение (включая печать) путём интерпретации или предварительной трансляции. В любом случае исполняемая программа (находящаяся в ОЗУ) будет называться ***рабочей программой***.

Понятие ***программы*** (как входной, так и рабочей) в А-системах несколько отличается от программ, обычно рассматриваемых в теории программирования. Прежде всего, исполнение программы не обязано заканчиваться через конечное число шагов – оно может продолжаться бесконечно.

Более того, в дальнейшем для удобства будем предполагать, что исполнение *любой* программы может продолжаться бесконечно: после окончания естественного исполнения можно считать, что исполнение продолжается, но только с помощью ***пустых шагов***.

Понятие пустого шага даёт возможность рассматривать как входную программу любую конечную последовательность букв входного алфавита: до тех пор пока возможна её интерпретация как истинной программы, выполняются соответствующие шаги рабочей программы, после чего следуют пустые шаги. В частности, не исключено, что все исполнение ограничится только пустыми шагами.

А-система S превращается в ***А-исчисление*** $\overline{S}$ (***алгоритмическое исчисление***), если во входном языке L существует понятие ***булева выражения*** с соответствующими логическими константами И и Л ("истина" и "ложь"), логическими операциями $\neg$, $\wedge$, $\vee$ и $\rightarrow$, а при наличии в языке L содержательных предметных понятий для операндов, отличных от булевых переменных (например, понятия целого числа), – также кванторов $\forall$ и $\exists$.

Булево выражение $B(x_1, ..., x_n)$, не содержащее других свободных предметных переменных, кроме $x_1, ..., x_n$, представляет собой n-местный предикат. При связывании всех предметных переменных кванторами предикат превращается в высказывание.

Добавляя к А-исчислению входную программу P, при исполнении которой на печать выводится некоторое множество R высказываний, замыкаемых справа знаками пробела, превращаем его в ***фор-***

мальную теорию над данным исчислением S, или, что то же самое, над данной А-системой S. Программа P называется ***программой вывода*** (логического вывода!) данной теории T, а все высказывания множества R и только они – ***выводимыми*** (формально доказуемыми) в теории T высказываниями, или ***теоремами*** данной теории. Обычно эта программа состоит из некоторого конечного числа элементарных правил вывода, применяемых к уже доказанным теоремам в определённом порядке и сочетающихся с конечным числом теорем, которые принимаются выводимыми априори (так называемые ***аксиомы***) и с которых начинает свою работу программа вывода. Заметим, однако, что при таком абстрактном понимании программы вывода возможность подобной её интерпретации не обязательна.

Выводимые высказывания формальной теории T и только они считаются ***формально истинными*** высказываниями этой теории. Помимо такого формального понятия истинности высказываний, которое определяется конструктивно, т.е. путём исполнения некоторой программы, можно говорить об их содержательной истинности. ***Содержательная истинность*** высказываний устанавливается путём анализа их "смысла" средствами, выходящими за рамки данной формальной теории, причём не обязательно конструктивным путём. Например, убирая в высказывании – булевом выражении языка L – все кванторы и вычислив значения полученного в результате бескванторного булева выражения при всех возможных комбинациях значений предметных переменных, на основании смысла кванторов $\forall$ (для всех) и $\exists$ (существует) можем, абстрактно говоря, определить, истинно ли исходное высказывание или ложно. Но недостаток в том, что при бесконечной предметной области подобный вывод может быть сделан лишь *после бесконечного числа проверок*, что заведомо не является конструктивной процедурой.

Понятие содержательной истинности априори предполагает его ***непротиворечивость*** в том смысле, что не могут быть содержательно истинными два противоположных высказывания: B и $\neg B$. Это обстоятельство позволяет по определению считать ***содержательно ложными*** отрицания всех содержательно истинных высказываний.

В отличие от содержательной истинности понятие формальной истинности (выводимости) априори может оказаться противоречивым. Поэтому его непротиворечивость (в том смысле, что и выше), или, что то же самое, ***непротиворечивость формальной теории***, должна доказываться или заранее оговариваться.

Полнота формальной теории означает возможность вывода в ней (формального доказательства) любого содержательно истинно-

го высказывания. Легко видеть, что в непротиворечивой и полной формальной теории выводимыми будут все содержательно истинные высказывания и только они.

Приведём ещё одно замечание – оно касается связи между предикатами и множествами: каждый одноместный предикат $Q(x)$ однозначно определяет некоторое подмножество M_Q области значений переменной x, а именно, – множество всех тех и только тех значений x, для которых предикат $Q(x)$ содержательно истинен. Ясно, что имеет место и обратное: каждое подмножество M области значений переменной x однозначно определяет предикат $Q(x)$ такой, что $M = M_Q$.

Предположим теперь, что для каждой предметной области входного языка L (например, области всех целых неотрицательных чисел N) в матобеспечение интерпретатора системы (ИВМ) включен *редактор вывода*. Его роль состоит в том, чтобы оставлять при выводе на печать лишь законченные (ограниченные справа другими символами) слова алфавита, представляющие собой элементы данной предметной области, разделяя их знаком пробела □, а все остальное выбрасывать.

Например, при выводе строки $-001{+}a : 1, 2{\times}b31{\vee}25$ "неотрицательно целочисленный" редактор напечатает строку 001□1□2□31□. Первый знак пробела здесь заменяет слово $+a$: , второй – запятую (,), третий – слово $\times b$, четвёртый – знак $\vee$. Число 25 как "незаконченное" (не ограниченное справа) редактором выбрасывается.

Работая с подобным редактором, любая входная программа (т.е. любая конечная последовательность букв входного алфавита) породит некоторое (быть может, пустое) подмножество предметной области. Заметим, что элементы этого подмножества "перечисляются" программой (в процессе печати) в произвольном порядке и, вообще говоря, с повторениями. Условимся называть все получаемые таким образом множества ***конструктивно перечислимыми*** в данной формальной теории.

Будем предполагать в дальнейшем, что входной язык L и интерпретирующая ИВМ алгоритмически универсальны, в частности, что они являются соответственно универсальным алгоритмическим языком и универсальной ИВМ (с соответствующим матобеспечением и с дополнительным условием бесконечности ОЗУ), наиболее знакомыми читателю.

В этом случае, очевидно, для каждой входной программы R, генерирующей (с соответствующим редактором) некоторое множество

M целых неотрицательных чисел, может быть построена программа-предикат $Q_M(x)$, которая при подстановке вместо x любого элемента множества M через конечное число шагов печатает И ("истина") и не печатает ничего (работая, быть может, бесконечно долго) при подстановке вместо x любого элемента, не принадлежащего M.

Искомая программа может быть построена, например, следующим образом. Она вводит в ИВМ число x и программу R, у которой вместо печати перечисление элементов множества M ведётся в ОЗУ. После вычисления каждого очередного элемента из M он сравнивается с x. В случае, если числа совпали, печатается И, если не совпали – программа R генерирует очередной элемент и цикл повторяется. Заметим, что входная программа $Q_M(x)$ может рассматриваться как булево выражение; оно определено лишь частично, а именно: при тех значениях x, при которых оно истинно. Вообще говоря, его нельзя конструктивно доопределить в остальных точках (путём печати символа Л), поскольку ни на каком шаге работы программы $Q_M(x)$ нельзя гарантировать, что $\notin M$.

Дополнив множество предикатов подобными "частично конструктивными" предикатами, перейдём к основной цели – доказательству обобщённой теоремы о неполноте.

Построим программы P_1 и P_2, позволяющие обосновать первый принципиальный шаг доказательства теоремы о неполноте.

Поскольку входной программой называем любую последовательность символов входного языка, нетрудно (при сделанных выше предположениях об универсальности входного языка L и интерпретатора – ИВМ) построить входную программу P_1, которая, будучи введённой в ИВМ, генерирует в её памяти сначала все односимвольные программы, потом все двухсимвольные, трехсимвольные и т.д. до бесконечности. При этом программа P_1 одновременно (с помощью матобеспечения ИВМ) приводит эти программы к виду, пригодному для исполнения (т.е. превращает их в рабочие программы), и приписывает им (в соответствии с порядком их порождения) последовательные номера $0, 1, 2, \ldots$

Дальнейшее усовершенствование программы P_1 приводит к программе P_2, которая, породив очередную пару (p_n, n), т.е. рабочую программу p_n и её номер n, передаёт управление для выполнения первого шага программы p_n, затем второго шага программы p_{n-1}, третьего шага программы p_{n-2} и т.д., пока не будет произведён $(n+1)$-й шаг программы p_0.

При этом должны быть соблюдены следующие дополнительные

условия. Во-первых, если очередной шаг не может быть выполнен в силу неинтерпретируемости рабочей программы или её окончания на одном из предыдущих шагов, предполагается, что осуществляется ничего не меняющий "пустой" шаг. Во-вторых, все выводы на печать от построенных рабочих программ блокируются. Их заменяет выписывание соответствующих символов в специально отводимую для каждой программы область (бесконечного) ОЗУ. В-третьих, производится проверка полностью "выпечатанных" (т.е. записанных в соответствующую область ОЗУ) целых чисел с номером, присвоенным соответствующей программе, и в случае их совпадения производится подлинный вывод на печать соответствующего номера, завершаемый каким-либо знаком раздела. Термин "полностью выпечатанное" применяется при этом к числу, ограниченному справа знаком раздела.

После того как программа P_2 продвинет таким образом на очередной шаг все порождённые ею рабочие программы, она приступает к порождению новой программы и новому циклу продвижений всех порождённых программ.

Нумеруя все порождаемые ею рабочие программы p_n, программа P_2 тем самым нумерует и конструктивно-перечислимые множества M_n, определяемые этими программами. При этом программа P_2 выводит на печать лишь такие номера n, для которых имеет место принадлежность $n \in M_n$.

Далее, любая рабочая программа p_n с фиксированным номером n рано или поздно будет продвинута на любое, сколь угодно большое число шагов. Поэтому в процессе такого продвижения через конечное число шагов обязательно появится любой наперёд заданный элемент множества M_n. Отсюда непосредственно следует справедливость следующего предложения.

Лемма 9.1.10. *Программа P_2 конструктивно перечисляет множество M, состоящее из всех тех и только тех чисел $n = 0, 1, 2, \ldots$, для которых имеет место принадлежность $n \in M_n$.*

Возьмём теперь дополнение $\overline{M}$ множества M в множестве всех целых неотрицательных чисел $N = \{0, 1, 2, \ldots\}$ и докажем, что имеет место следующее предложение.

Лемма 9.1.11. *Множество $\overline{M}$ не является конструктивно перечислимым.*

Действительно, предполагая противное, следует допустить, что существует программа p_n, порождающая множество $\overline{M}$. Иными

словами, должен существовать номер n, для которого $M = M_n$. Каким бы ни было множество M_n (пустым или непустым), для него обязательно должно иметь место одно и только одно из двух предположений, а именно: $n \in M_n$ или $n \notin M_n$.

При первом предположении, в силу леммы 1, $n \in M$, а значит, $n \notin \overline{M} = M_n$, т.е. $n \notin M_n$ в противоречие с предположением.

При втором предположении, также в силу леммы 1, $n \notin M$. Следовательно, $n \in \overline{M} = M_n$, т.е. $n \in M_n$, что тоже противоречит высказанному предположению.

Лемма доказана. □

Построим теперь программу-предикат $P_3(x)$, которая будет выдавать через конечное число шагов символ И ("истина") при подстановке вместо x тех и только тех значений, которые принадлежат множеству M (о способе построения таких программ говорилось выше).

Предикат $\neg P_3(x)$ соответствует дополнению $\overline{M}$ множества M. Построим программу P_4, которая генерирует последовательно высказывания $\neg P_3(0)$, $\neg P_3(1)$, $\neg P_3(2)$, ... и одновременно расширяет множество R выводимых в этой теории высказываний B_1, B_2, ... с помощью соответствующего продвижения программы вывода P рассматриваемой формальной теории. При этом, разумеется, вывод на печать перечисляемых формально истинных высказываний B_1, B_2, ... должен быть заменен их размещением в свободной зоне ОЗУ.

Если в дополнение к этому программа P_4 будет сравнивать элементы (высказывания) двух строящихся множеств и выпечатывать со знаками пробела справа все номера n, для которых $\neg P_3(n) \in R$ (на некотором шаге построения множества R), то тем самым определяем некоторое конструктивно-перечислимое (программой P_4) множество K неотрицательных целых чисел. Оно состоит из всех тех и только тех номеров n, для которых высказывание $\neg P_3(n)$ выводимо в рассматриваемой формальной теории T.

Из высказываний $\neg P_3(n)$ содержательно истинными будут те и только те, для которых $n \in \overline{M}$. Все остальные высказывания этого вида будут, таким образом, ложными. Если теория T непротиворечива и полна, то в ней, как отмечалось выше, выводимы все содержательно истинные высказывания и только они.

Это означает, очевидно, совпадение множеств $\overline{M}$ и K, что невозможно, поскольку выше доказано, что одно из них (K) конструктивно перечислимо, а другое нет. Тем самым доказано следующее предложение, которое будем называть обобщенной теоремой о неполноте формальных теорий.

Теорема 9.1.12. *Если алгоритмическая система S (входной язык L и интерпретатор C) такова, что в ней могут быть построены и правильно исполнены программы P_1, P_2, P_3, P_4, то любая непротиворечивая формальная теория над системой S будет неполной, т.е. в ней обязательно будут содержательно истинные, но формально недоказуемые предложения (высказывания).*

Заметим, что над одной и той же алгоритмической системой может быть построено много формальных теорий, различающихся друг от друга программами вывода P, или – в терминах классической математической логики – системами аксиом и правилами вывода. При этом никакое расширение программы вывода, в частности никакое включение в состав аксиом новых содержательно истинных высказываний (разумеется, конечного их числа), не может сделать теорию полной. Легко понять, что то же будет и в случае дополнения системы аксиом любым конструктивно-перечислимым множеством содержательно истинных высказываний. Иначе можно было бы, объединив программу генерирования этого множества с программой вывода P, получить новую программу вывода, устраняющую неполноту в соответствующем расширении теории.

Таким образом, *множество содержательно истинных, но не выводимых высказываний в любой формальной теории над А-системой будет обязательно бесконечным и притом конструктивно-неперечислимым.*

Из доказанной теоремы следует, что причина неполноты формальной теории не в собственно самой теории (программе вывода), а в её алгоритмическом базисе – А-системе S. При достижении этим базисом алгоритмической полноты всякая строящаяся над ним непротиворечивая формальная теория будет обязательно неполной.

Интересно оценить минимальную сложность алгоритмического базиса, при котором верна теорема о полноте. Можно показать, что теорема о неполноте будет верна, если в качестве интерпретатора взять любую универсальную машину Тьюринга, а в качестве входного языка – язык команд этой машины. В любом случае для справедливости теоремы о полноте А-система должна обладать возможностью генерировать бесконечные множества, в том числе такие, дополнения которых не могут генерироваться системой.

Оставаясь в выбранном вначале классе интерпретаторов (человек или ИВМ), можно минимизировать различным образом входной язык L системы. Теорему о неполноте можно, в частности, доказать, если во входном языке оставить возможность (сохраняя алгоритмическую универсальность) оперировать лишь натуральными числа-

ми. Тогда эта теорема превратится в представленную в несколько необычной новой форме классическую теорему Гёделя о неполноте арифметики натуральных чисел.

О других размышлениях, связанных с теоремой Гёделя, в частности, философского характера, см. [15], [16], [18].

9.2. Неразрешимость формализованного исчисления предикатов

Построение формализованного исчисления предикатов было начато нами в § 25 Учебника МЛ и продолжено в § 11 Задачника. В § 29 Учебника МЛ были установлены некоторые свойства этого исчисления как аксиоматической теории. В частности, доказана полнота этой теории: замкнутая формула доказуема в этой теории тогда и только тогда, когда она является общезначимой формулой логики предикатов: $\vdash F \iff \models F$ (теорема 29.14). Из аналогичной теоремы для формализованного исчисления высказываний (теорема 16.6 из упомянутого учебника) вытекает разрешимость формализованного исчисления высказываний (теорема 16.11). Это объясняется тем, что теоремой о полноте проблема разрешимости из области формальной теории переводится в область алгебры высказываний. В последней имеется эффективная процедура для определения того, является ли данная формула тождественно истинной (тавтологией) – это процедура составления таблицы значений данной формулы. В отличие от алгебры высказываний, в логике предикатов подобная процедура для определения общезначимости формулы отсутствует. Сам по себе этот факт, конечно, ещё не служит доказательством того, что формализованное исчисление предикатов есть неразрешимая аксиоматическая теория, но является тревожным симптомом к этому.

Напомним, что разрешимость аксиоматической теории означает наличие для этой теории разрешающего алгоритма, т.е. алгоритма, позволяющего для каждого утверждения, сформулированного на языке данной теории, получить ответ, является это утверждение теоремой данной теории или нет. Так вот, оказывается, что для формализованного исчисления предикатов такого алгоритма не существует, т.е. формализованное исчисление предикатов – неразрешимая аксиоматическая теория. Это впервые доказал в 1936 г. американский математик А.Чёрч. Упомянутая в начале параграфа теорема о полноте формализованного исчисления предикатов сразу же переводит этот результат из формализованного исчисления предикатов в

содержательную логику предикатов: алгоритмически неразрешима проблема определения общезначимости (т.е. проблема *истинности*) формул логики предикатов. Д.Гильберт считал этот результат самым фундаментальным касающимся неразрешимости результатом во всей математике.

В настоящем параграфе теорема о неразрешимости формализованного исчисления предикатов будет доказана. Идея доказательства основывается на методе от противного: будет показано, что если бы соответствующий алгоритм существовал, то оказалась бы разрешимой проблема остановки для машин Тьюринга. Но в параграфе 7.1 мы доказали, что никакая машина Тьюринга не может реализовать процедуру, решающую проблему остановки для машин Тьюринга. В силу тезиса Тьюринга, это означает, что не существует вообще никакого алгоритма, решающего проблему остановки для машин Тьюринга. Это противоречие будет доказывать отсутствие алгоритма, решающего проблему разрешения для формализованного исчисления предикатов.

Идея доказательства. Доказательство от противного будет строиться так: мы покажем, как для данной машины Тьюринга, зная её программу, и для данного натурального числа n можно эффективно построить такое конечное множество формул логики предикатов $\Delta = \{F_1, F_2, ..., F_w\}$ и ещё одну формулу логики предикатов H, что $\Delta \vdash H$ (из Δ выводится H в ФИП) тогда и только тогда, когда рассматриваемая машина Тьюринга, будучи запущенной с числом r на входе (т.е. когда она начинает работу в состоянии q_1, считывая при этом самую левую единицу в сплошном массиве из r единиц на ленте с пустыми символами в остальных клетках), в конце концов останавливается.

Если теперь предположить, что мы можем эффективно решать проблему распознавания общезначимости формул логики предикатов, то взяв формулы $F_1, F_2, ..., F_w, H$, мы можем определить, является ли общезначимой следующая формула $(F_1 \wedge F_2 \wedge ... \wedge F_w) \rightarrow H$. А раз так, то мы можем эффективно распознать, будет ли выполняться в логике предикатов логическое следование $F_1 \wedge F_2 \wedge ... \wedge F_w \models H$, а значит, и следование $F_1, F_2, ..., F_w \models H$. А раз так, то в силу теоремы адекватности для формализованного исчисления предикатов (теорема 29.13 из Учебника МЛ) мы можем распознать, имеет ли место выводимость $F_1, F_2, ..., F_w \vdash H$, т.е. $\Delta \vdash H$. Наконец, переходя теперь на построенную модель, можем определить, остановится ли данная машина Тьюринга, если на её вход будет подано число n (разумеется, в закодированном виде). Таким образом, сделанное

предположение приводит нас к алгоритму, который способен решить проблему остановки для машин Тьюринга. Согласно теореме 7.1.5, такого алгоритма не существует. Следовательно, сделанное предположение неверно, т.е. алгоритма, распознающего общезначимость формул логики предикатов, не существует. Тем более, и подавно не существует алгоритма, распознающего доказуемые в ФИП формулы, ибо если бы такой алгоритм существовал, то с его помощью на основании теоремы о полноте ФИП (теорема 16.6 из учебника) мы могли бы распознавать общезначимые формулы логики предикатов: $\models F \iff \vdash F$, что невозможно по только что выше доказанному.

Построение для данной машины Тьюринга множества формул Δ и формулы H. Понятие машины Тьюринга введено в §2.1. Прежде чем приступить к построению множества формул Δ и формулы H, сделаем некоторые уточнения, касающиеся этого понятия. Во-первых, будем считать, что командами машины Тьюринга являются лишь выражения одного из следующих видов:

$$q_i a_j \to q_k a_l \text{С}, \quad q_i a_j \to q_k a_j \text{П}, \quad q_i a_j \to q_k a_j \text{Л} \ .$$

Это означает, что два машинных действия – смена содержимого обозреваемой ячейки и шаг вправо или влево по ленте машины не могут выполняться по одной команде, а для этого нужно выполнить две команды. Например, чтобы получить тот же результат, что и при выполнении прежней команды $q_i a_j \to q_k a_l \text{П}$, теперь нужно последовательно выполнить две команды: $q_i a_j \to q_k a_l \text{С}$, $q_k a_l \to q_k a_l \text{П}$. Напомним, что машина Тьюринга имеет внешний алфавит $A = \{a_0, a_1, ..., a_n\}$ – список символов, записываемых в ячейки бесконечной ленты и считываемых из них, и алфавит внутренних состояний $Q = \{q_0, q_1, ..., q_m\}$.

Во-вторых, будем считать, что ячейки бесконечной ленты занумерованы целыми числами следующим образом:

...		−4	−3	−2	−1	0	1	2	3	4			...

Наконец, в-третьих, будем предполагать, что время разбито на бесконечную последовательность моментов t, в каждом из которых машина выполняет точно одну команду, и что машина начинает работу в момент времени 0, считывая при этом содержимое ячейки 0. Моменты времени предполагаются неограниченно продолжаемыми как в будущее, так и в прошлое, а лента бесконечно простирается и влево, и вправо.

Теперь приступим к построению множества формул Δ и формулы H для заданной нам конкретной машины Тьюринга. Эти фор-

мулы, по существу, будут описывать на языке логики предикатов программу этой машины Тьюринга.

Каждому состоянию q_i, в котором может пребывать машина Тьюринга, ставится в соответствие некоторый двуместный предикатный символ Q_i; каждой букве a_j внешнего алфавита ставится в соответствие двуместный предикатный символ A_j. Интерпретировать эти символы применительно к данной машине Тьюринга будем следующим образом: $Q_i(t,x)$ истинно тогда и только тогда, когда машина в момент времени t находится в состоянии q_i и обозревает на ленте ячейку с номером x; $A_j(t,x)$ истинно тогда и только тогда, когда машина в момент времени t в ячейке с номером x записан символ a_j.

Кроме этого, в составляемых формулах будут задействованы следующие символы: 0 (обозначает число нуль), одноместный функциональный символ $'$ (обозначает функцию следования, заданную на множестве натуральных чисел, т.е. $x' = x+1$), двуместный предикатный символ $<$ (обозначает отношение порядка на множестве целых чисел). Наконец, будем использовать символ равенства $=$ и символы логических операций $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\forall$, $\exists$.

Для каждого из трёх типов команд машины Тьюринга составим соответствующие формулы логики предикатов, описывающие эти команды.

Для каждой команды вида $q_i a_j \rightarrow q_k a_l \text{С}$ мы включаем в множество Δ формулу:

$$(\forall t)(\forall x)(\forall y)\{(Q_i(t,x) \wedge A_j(t,x)) \rightarrow [Q_k(t',x) \wedge A_l(t',x) \wedge y \neq x \rightarrow$$
$$\rightarrow (A_0(t,y) \rightarrow A_0(t',y)) \wedge \ldots \wedge (A_n(t,y) \rightarrow A_n(t',y))]\} . \quad (1)$$

Эта формула для нашей машины Тьюринга говорит следующее. Если в момент времени t машина находится в состоянии q_i, обозревая при этом в ячейке с номером x символ a_j, то в следующий момент времени $t' = t+1$ машина перейдёт в состояние q_k, содержимое обозреваемой ячейки с номером x заменит на a_l, а во всех остальных ячейках с номерами, отличными от x, в момент $t' = t+1$ останутся те же самые символы, что и в предыдущий момент t (для всех t и x).

Для каждой команды вида $q_i a_j \rightarrow q_k a_j \Pi$ включаем в множество Δ формулу:

$$(\forall t)(\forall x)(\forall y)\{(Q_i(t,x) \wedge A_j(t,x)) \rightarrow$$
$$\rightarrow [Q_k(t',x') \wedge (A_0(t,y) \rightarrow A_0(t',y)) \wedge \ldots \wedge (A_n(t,y) \rightarrow A_n(t',y))]\}. \quad (2)$$

Для каждой команды вида $q_i a_j \to q_k a_j$Л включаем в множество Δ формулу:

$$(\forall t)(\forall x)(\forall y)\{(Q_i(t,x') \wedge A_j(t,x')) \to$$
$$\to [Q_k(t',x) \wedge (A_0(t,y) \to A_0(t',y)) \wedge ... \wedge (A_n(t,y) \to A_n(t',y))]\}. \quad (3)$$

Следующая формула описывает начальную конфигурацию машины Тьюринга: в момент времени 0 машина находится в состоянии q_1, обозревает ячейку с номером 0, в ячейках с номерами 0, 1, ... , $r-1$ записан символ a_1 (читай – символ 1), а во всех остальных ячейках записан символ a_0 (читай – символ 0, что означает пустоту этих ячеек), так что на ленту в закодированном виде записано число r:

$$Q_1(0,0) \wedge A_1(0,0) \wedge A_1(0,0') \wedge ... \wedge A_1(0,0^{(r-1)}) \wedge$$
$$\wedge (\forall y)[(y \neq 0 \wedge y \neq 0' \wedge ... \wedge y \neq 0^{(r-1)}) \to A_0(0,y)] . \quad (4)$$

Здесь $0^{(r-1)}$ обозначает результат применения $r-1$ символов $'$ следования к символу 0, т.е. код числа $r-1$. Если $r=0$, то формула (4) принимает вид:

$$Q_1(0,0) \ \wedge \ (\forall y)[A_0(0,y)]$$

(во всех ячейках ленты записан символ a_0).

Следующие две формулы, которые мы также включим в Δ, утверждают некоторые свойства функции следования $'$. Первая – что всякое целое число является следующим точно за одним целым:

$$(\forall z)(\exists x)(z = x') \ \wedge \ (\forall z)(\forall x)(\forall y)[(z = x' \wedge z = y') \ \to \ x = y] . \quad (5)$$

Вторая позволяет получить из неё формулу $(\forall x)(x^{(p)} \neq x^{(q)})$, если $p \neq q$, утверждающую, что никакие последующие разного порядка для одного и того же числа не могут быть равными. Все такие формулы являются следствиями формулы:

$$(\forall x)(\forall y)(\forall z)[(x < y \ \wedge \ y < z) \ \to \ x < z] \wedge$$
$$\wedge(\forall x)(\forall y)(x' = y \ \to \ x < y) \ \wedge \ (\forall x)(\forall y)(x < y \ \to \ x \neq y). \quad (6)$$

Например, из (6) выводятся неравенства $x'' < x'''$ (так как $(x'')' = x'''$) и $x''' < x''''$, а значит, $x'' < x''''$, откуда $x'' \neq x''''$, т.е. $x^{(2)} \neq x^{(4)}$.

Примем за Δ множество всех формул (1), (2), (3), соответствующих всем командам данной машины Тьюринга, вместе с тремя дополнительными формулами (4), (5), (6). Ясно, что рассмотренная интерпретация с помощью данной машины Тьюринга служит моделью для множества формул Δ.

Теперь построим формулу H. Для этого заметим, что всякая машина Тьюринга останавливается в момент времени t, если она в этот момент находится в таком состоянии q_i и обозревает ячейку с номером x, в которой записан такой символ a_j, что среди команд этой машины нет команды, начинающейся с комбинации $q_i a_j$. (В этом случае считаем, что машина, не зная, что делать дальше, останавливается). Тогда за формулу H мы принимаем дизъюнкцию всех таких формул вида

$$(\exists t)(\exists x)[Q_i(t,x) \wedge A_j(t,x)] , \tag{7}$$

для которых среди команд данной машины Тьюринга нет команд, начинающихся с комбинации $q_i a_j$. Если же для всякой такой комбинации имеются соответствующие команды, то машина никогда не остановится, и за H в этом случае мы принимаем какое-либо предложение, ложное в данной интерпретации, например, $0 \neq 0$, т.е. $\neg(0 = 0)$.

Подведём итог: мы показали, как по данной машине Тьюринга и входному значению r построить такие конечное множество Δ формул и отдельную формулу H, что ***выводимость*** $\Delta \vdash H$ ***имеет место тогда и только тогда, когда машина, получив*** r ***на входе, в конце концов останавливается.*** Последнее утверждение, разумеется, должно быть доказано. Необходимость проверяется тривиально. Пусть имеет место выводимость $\Delta \vdash H$. Поскольку, кроме того, по построению все формулы из Δ превращаются в нашей модели в истинные высказывания, поэтому и формула H превращается в истинное утверждение. Но истинность H означает, что машина, получив r на входе, в конце концов останавливается.

Доказательство достаточности сложнее, и оно потребует значительно больших усилий. Придётся применить результаты, полученные в математической логике о логическом следовании в логике предикатов, о выводимости формул в формализованном исчислении предикатов (ФИП), теорему адекватности ФИП, устанавливающую связь между этими понятиями (см. упомянутый учебник, §§ 25, 29).

Доказательство достаточности. Формула (4) из предыдущего пункта описывает на языке логики предикатов начальную конфигурацию машины Тьюринга, т.е. её конфигурацию в момент времени 0. Составим теперь формулу логики предикатов, представляющую собой обобщение формулы (4) и описывающую конфигурацию машины Тьюринга в произвольный момент времени s:

$$Q_i(0^{(s)}, 0^{(p)}) \wedge A_{j_1}(0^{(s)}, 0^{(p_1)}) \wedge \ldots \wedge A_j(0^{(s)}, 0^{(p)}) \wedge \ldots \wedge A_{j_v}(0^{(s)}, 0^{(p_v)}) \wedge$$

$$\wedge (\forall y)[(y \neq 0^{(p_1)} \wedge \ \dots \ \wedge y \neq 0^{(p)} \wedge \ \dots \ \wedge y \neq 0^{(p_v)}) \ \rightarrow \ A_0(0^{(s)}, y)] \ . \qquad (8)$$

Эта формула означает: в момент времени s машина находится в состоянии q_i, обозревает ячейку с номером p, в которой записан символ a_j; в ячейках с номерами $p_{j_1}, \dots, p_{j_v}$ записаны символы $a_{j_1}, \dots, a_{j_v}$ соответственно; все остальные ячейки ленты пусты (в них записан символ a_0). При этом, мы считаем, что номера ячеек идут в порядке возрастания $p_{j_1} < \dots < p < \dots < p_{j_v}$ и ячейка с номером p находится среди ячеек с номерами $p_{j_1}, \dots, p_{j_v}$.

Остаётся уточнить один нюанс в этом описании: как описывать содержимое ячеек с отрицательными номерами, т.е. ячеек, расположенных левее фиксированной начальной ячейки с номером 0. Если p – отрицательное целое число и $p = -q$ (т.е. q – целое положительное), то записи слева рассматриваются как сокращения для формул, записанных справа:

$Q_i(x, 0^{(p)})$ *означает* $(\exists z)[Q_i(x, z) \ \wedge \ z^{(q)} = 0]$,

$A_j(x, 0^{(p)})$ *означает* $(\exists z)[A_i(x, z) \ \wedge \ z^{(q)} = 0]$,

$y \neq 0^{(p)}$ *означает* $(\exists z)[y \neq z) \ \wedge \ z^{(q)} = 0]$.

(z – это номер такой ячейки, что , сделав от этой ячейки q шагов вправо, мы окажемся в начальной ячейке с номером 0).

Обратимся теперь к формуле (8) и покажем, что *если машина Тьюринга не останавливается до момента времени s ($s \geq 0$), то из множества формул Δ как из гипотез логически выводится в формализованном исчислении предикатов формула (8), дающая описание на языке логики предикатов конфигурации машины Тьюринга в момент времени s.*

Доказательство будем вести индукцией по s.

База индукции. Пусть $s = 0$. Множество Δ содержит по построению формулу (4), которая, как мы отмечали, представляет собой формулу (8) при $s = 0$. Тогда очевидно формула (4) выводима (доказуема) из Δ: $\Delta \vdash (4)$. (См. Учебник МЛ, пример 15.2 и следствие 15.5).

Предположение индукции. Предположим, что выделенное курсивом утверждение верно для момента времени s.

Шаг индукции. Покажем тогда, что рассматриваемое утверждение будет верно и для следующего момента времени $s + 1$. Для этого предположим, что наша машина Тьюринга не остановилась

до момента $s+1$. Это значит, что она не остановилась ни до момента s, ни в самый момент s. Тогда по предположению индукции, из Δ выводится формула (8), описывающая конфигурацию машины Тьюринга в момент времени s. Нужно доказать, что из Δ в формализованном исчислении предикатов будет выводиться формула, описывающая конфигурацию машины Тьюринга в момент времени $s+1$, т.е. формула вида (8), в которой вместо s стоит $s+1$.

Для этого мы должны знать, что будет делать машина в момент времени $s+1$, т.е. какую команду она будет выполнять. В момент s машина находится в состоянии q_i, считывая при этом ячейку с номером p, в которой записан символ a_j. Поскольку машина в момент s не остановилась, в её программе должна присутствовать команда одного из трёх следующих видов: $q_i a_j \to q_k a_l \text{С}$, $q_i a_j \to q_k a_j \text{П}$, $q_i a_j \to q_k a_j \text{Л}$.

Рассмотрим последовательно эти три возможности.

1) Если выполняется команда первого вида $q_i a_j \to q_k a_l \text{С}$, то в множестве Δ ей соответствует описывающая её формула (1). Покажем, как в формализованном исчислении предикатов (ФИП) из формул (1), (5), (6) и (8) вывести следующую формулу, описывающую конфигурацию машины Тьюринга в момент времени $s+1$:

$$Q_k(0^{(s+1)}, 0^{(p)}) \wedge A_{j_1}(0^{(s+1)}, 0^{(p_1)}) \wedge \ldots$$
$$\wedge A_l(0^{(s+1)}, 0^{(p)}) \wedge \ldots \wedge A_{j_v}(0^{(s+1)}, 0^{(p_v)}) \wedge$$
$$\wedge (\forall y)[(y \neq 0^{(p_1)} \wedge \ldots \wedge y \neq 0^{(p)} \wedge \ldots \wedge y \neq 0^{(p_v)}) \to$$
$$\to A_0(0^{(s+1)}, y)] \, . \qquad (8')$$

Вывод начнём с формулы (1), представив её в виде $(\forall t)(F(t))$, где $F(t)$ – вся часть формулы (1) без первого квантора $(\forall t)$. Добавим к ней формулу $(\forall t)(F(t)) \to F(0^s)$, являющуюся 1-ой аксиомой Барвайса для ФИП (см. Учебник МЛ, § 25). Из этих двух формул по правилу Modus Ponens (MP) выводится формула $F(0^s)$, которую, в свою очередь, можно представить в виде $(\forall x)(G(x))$, где $G(x)$ – вся часть формулы $F(0^s)$ без первого квантора $(\forall x)$. Аналогично, добавив к ней формулу $(\forall x)(G(x)) \to G(0^p)$, по правилу MP выведем формулу $G(0^p)$, в которой в префиксной части стоит единственный квантор $(\forall y)$, а бескванторная часть представляет собой импликацию, посылка которой не зависит от y. Поэтому ввиду равносильности в) из теоремы 21.12 учебника из логики предикатов (см. также Задачник, № 9.49 к), этот квантор может быть пронесён к заключению импликации. В свою очередь, заключение импликации

представляет собой конъюнкцию трёх формул, первые две из которых не зависят от y. Поэтому ввиду равносильности а) из теоремы 21.11 (Учебник МЛ) из логики предикатов (см. также Задачник, № 9.49 в), этот квантор может быть пронесён к третьему члену конъюнкции. В итоге оказывается выведенной следующая формула:

$$(Q_i(0^{(s)},0^{(p)}) \wedge A_j(0^{(s)},0^{(p)})) \to \{Q_k(0^{(s+1)},0^{(p)}) \wedge A_l(0^{(s+1)},0^{(p)}) \wedge$$
$$\wedge\ (\forall y)[y \neq 0^{(p)} \to ((A_0(0^{(s)},y) \to A_0(0^{(s+1)},y)) \wedge \ldots$$
$$\ldots \wedge (A_n(0^{(s)},y) \to A_n(0^{(s+1)},y)))]\}\,. \qquad (9)$$

Остаётся отметить следующий нюанс. Мы пользовались здесь равносильностями логики предикатов, а должны были доказывать соответствующие равносильности в ФИП. Необходимую связь здесь обеспечивает теорема адекватности ФИП: $\Phi \vdash F \iff \Phi \models F$ (см. теорему 29.13 в учебнике). Этой теоремой мы будем пользоваться и в дальнейших рассуждениях.

Обратимся теперь к формуле (8). Она представляет собой конъюнкцию конечного числа членов, каждый из которых из неё выводится, и значит, выводится из множества Δ (правило удаления конъюнкции: учебник, теорема 15.10 б). В частности, выводятся формулы:

$$(Q_i(0^{(s)},\ 0^{(p)}) \qquad \text{и} \qquad A_j(0^{(s)},\ 0^{(p)}))\ ,$$

а следовательно, выводится конъюнкция этих формул:

$$Q_i(0^{(s)},\ 0^{(p)}) \wedge A_j(0^{(s)},\ 0^{(p)})\ , \qquad (10)$$

(правило введения конъюнкции: учебник, теорема 15.9 б).

Из формул (9), (10) по правилу MP выводится следующая формула:

$$Q_k(0^{(s+1)},0^{(p)}) \wedge A_l(0^{(s+1)},0^{(p)}) \wedge$$
$$\wedge\ (\forall y)[y \neq 0^{(p)} \to ((A_0(0^{(s)},y) \to A_0(0^{(s+1)},y)) \wedge \ldots$$
$$\ldots \wedge (A_n(0^{(s)},y) \to A_n(0^{(s+1)},y)))]\,. \qquad (11)$$

Из конъюнкции (11) выводится каждый её член (правило удаления конъюнкции):

$$Q_k(0^{(s+1)},\ 0^{(p)}) \wedge A_l(0^{(s+1)},\ 0^{(p)}) \qquad (12)$$

и

$$(\forall y)[y \neq 0^{(p)} \to ((A_0(0^{(s)},y) \to A_0(0^{(s+1)},y)) \wedge \ldots$$
$$\ldots \wedge (A_n(0^{(s)},y) \to A_n(0^{(s+1)},y)))]\,. \qquad (13)$$

Представляя формулу (13) в виде $(\forall y)(F(y))$, запишем 1-ую аксиому Барвайса для ФИП (см. Учебник МЛ, § 25). $(\forall y)(F(y)) \to F(0^{p_1})$. Из этой формулы по правилу Modus Ponens (MP) выводится формула:

$$0^{(p_1)} \neq 0^{(p)} \to ((A_0(0^{(s)}, 0^{(p_1)}) \to A_0(0^{(s+1)}, 0^{(p_1)})) \wedge \ldots$$
$$\ldots \wedge (A_{j_1}(0^{(s)}, 0^{(p_1)}) \to A_{j_1}(0^{(s+1)}, 0^{(p_1)})) \wedge \ldots$$
$$\wedge (A_n(0^{(s)}, 0^{(p_1)}) \to A_n(0^{(s+1)}, 0^{(p_1)})))] . \quad (14)$$

Мы уже отмечали, что формулы (5) и (6), включённые в множество формул Δ, дают возможность выводить утверждение $(\forall x)(x^{(p)} \neq x^{(q)})$ для $p \neq q$. Поэтому выводима формула $0^{(p_1)} \neq 0^{(p)}$. Из неё и формулы (14) по правилу MP выводится конъюнкция импликаций, из которой, в свою очередь, выводится одна импликация:

$$A_{j_1}(0^{(s)}, 0^{(p_1)}) \to A_{j_1}(0^{(s+1)}, 0^{(p_1)}) .$$

Поскольку из формулы (8) выводится посылка этой импликации, поэтому по правилу MP выводится и следствие: $A_{j_1}(0^{(s+1)}, 0^{(p_1)})$.

Аналогичными рассуждениями устанавливается выводимость формул: $A_{j_2}(0^{(s+1)}, 0^{(p_2)}) , \ldots , A_{j_v}(0^{(s+1)}, 0^{(p_v)})$. Следовательно, выводима и конъюнкция этих формул:

$$A_{j_2}(0^{(s+1)}, 0^{(p_2)}) \wedge \ldots \wedge A_{j_v}(0^{(s+1)}, 0^{(p_v)}) . \quad (15)$$

Из выводимости формул (12) и (15) следует выводимость их конъюнкции (по правилу введения конъюнкции):

$$Q_k(0^{(s+1)}, 0^{(p)}) \wedge A_{j_1}(0^{(s+1)}, 0^{(p_1)}) \wedge \ldots \wedge A_l(0^{(s+1)}, 0^{(p)}) \wedge \ldots$$
$$\ldots \wedge A_{j_v}(0^{(s+1)}, 0^{(p_v)}) . \quad (16)$$

Нам остаётся обосновать выводимость последнего члена конъюнкции в формуле (8′). Подобно формуле (13) из Δ могут быть выведены следующие формулы:

$$(\forall y)[y \neq 0^{(p_1)} \to ((A_0(0^{(s)}, y) \to A_0(0^{(s+1)}, y)) \wedge \ldots$$
$$\ldots \wedge (A_n(0^{(s)}, y) \to A_n(0^{(s+1)}, y)))] ,$$

. . . .

$$(\forall y)[y \neq 0^{(p_v)} \to ((A_0(0^{(s)}, y) \to A_0(0^{(s+1)}, y)) \wedge \ldots$$
$$\ldots \wedge (A_n(0^{(s)}, y) \to A_n(0^{(s+1)}, y)))] .$$

Из каждой из этих формул соответственно выводятся следующие формулы:

$$(\forall y)[y \neq 0^{(p_1)} \rightarrow ((A_0(0^{(s)}, y) \rightarrow A_0(0^{(s+1)}, y))] \ ,$$

$$\cdots \qquad (17)$$

$$(\forall y)[y \neq 0^{(p_v)} \rightarrow ((A_0(0^{(s)}, y) \rightarrow A_0(0^{(s+1)}, y))] \ .$$

Это утверждение основано на следующей выводимости:

$$\frac{(\forall y)[F(y) \rightarrow (G_0(y) \wedge G_1(y))]}{(\forall y)[F(y) \rightarrow G_0(y)]} ,$$

которая легко проверяется в логике предикатов и по теореме адекватности ФИП переносится в ФИП.

Из формул (17) выводится следующая формула: (18)

$$(\forall y)[(y \neq 0^{(p_1)} \wedge \ldots \wedge y \neq 0^{(p_v)}) \rightarrow (A_0(0^{(s)}, y) \rightarrow A_0(0^{(s+1)}, y))].$$

Это утверждение основано на следующей выводимости:

$$\frac{\begin{array}{c}(\forall y)[P_1(y) \rightarrow Q(y)] \\ \cdots \\ (\forall y)[P_v(y) \rightarrow Q(y)]\end{array}}{(\forall y)[(P_1(y) \wedge \ldots \wedge P_v(y)) \rightarrow Q(y)]} ,$$

которая легко проверяется в логике предикатов и по теореме адекватности ФИП переносится в ФИП.

Наконец, из формул (16) и (18) по правилу введения конъюнкции выводится формула (8′).

2) Если выполняется команда второго вида $q_i a_j \rightarrow q_k a_j \Pi$, то в множестве Δ ей соответствует описывающая её формула (2). Покажем, как в формализованном исчислении предикатов (ФИП) из формул (2), (5), (6) и (8) вывести следующую формулу, описывающую конфигурацию машины Тьюринга в момент времени $s+1$ (после выполнения команды указанного вида):

$$Q_k(0^{(s+1)}, 0^{(p+1)}) \wedge A_{j_1}(0^{(s+1)}, 0^{(p_1)}) \wedge \ldots \wedge A_j(0^{(s+1)}, 0^{(p)}) \wedge \ldots$$
$$\ldots \wedge A_q(0^{(s+1)}, 0^{(p+1)}) \wedge \ldots \wedge A_{j_v}(0^{(s+1)}, 0^{(p_v)}) \wedge$$
$$\wedge (\forall y)[(y \neq 0^{(p_1)} \wedge \ldots \wedge y \neq 0^{(p)} \wedge y \neq 0^{(p+1)} \wedge \ldots \wedge y \neq 0^{(p_v)}) \rightarrow$$
$$\rightarrow A_0(0^{(s+1)}, y)] \ . \qquad (8'')$$

Эта формула говорит: машина Тьюринга перешла в состояние q_k, обозревая при этом ячейку с номером $p+1$ (конъюнктивный член $Q_k(0^{(s+1)}, 0^{(p+1)})$) и наблюдая в этой ячейке символ a_q (конъюнктивный член $A_q(0^{(s+1)}, 0^{(p+1)})$); содержимое ячеек с записанными

в них символами внешнего алфавита не изменилось (члены конъюнкции $A_{j_1}(0^{(s+1)},\ 0^{(p_1)})$, ... , $A_j(0^{(s+1)},\ 0^{(p)})$, ... , $A_q(0^{(s+1)},\ 0^{(p+1)})$, ... , $A_{j_v}(0^{(s+1)},\ 0^{(p_v)})$); во всех остальных ячейках как был записан в момент времени s пустой символ a_0, так он и остаётся записанным в них в следующий момент $s+1$ (последний член конъюнкции с квантором общности).

Вывод начнём с формулы (2). Как и в случае **1**, аналогичным образом выведем из неё следующую формулу, подобную формуле (9):

$$(Q_i(0^{(s)},0^{(p)}) \wedge A_j(0^{(s)},0^{(p)})) \rightarrow \{Q_k(0^{(s+1)},0^{(p+1)}) \wedge \quad (19)$$
$$\wedge (\forall y)[(A_0(0^{(s)},y) \rightarrow A_0(0^{(s+1)},y)) \wedge ... \wedge (A_n(0^{(s)},y) \rightarrow A_n(0^{(s+1)},y))]\}.$$

Как и в случае **1**, аналогичным образом выводятся формулы:

$$Q_k(0^{(s+1)},\ 0^{(p+1)}) \quad (20)$$

и (21)

$$(\forall y)[(A_0(0^{(s)},y) \rightarrow A_0(0^{(s+1)},y)) \wedge ... \wedge (A_n(0^{(s)},y) \rightarrow A_n(0^{(s+1)},y))]\},$$

подобные формулам (12) и (13) соответственно.

Из формулы (21) выводится формула:

$$A_{j_1}(0^{(s)},\ 0^{(p_1)}) \rightarrow A_{j_1}(0^{(s+1)},\ 0^{(p_1)}) .$$

Поскольку из формулы (8) выводится посылка этой импликации, поэтому по правилу МР выводится и следствие $A_{j_1}(0^{(s+1)},\ 0^{(p_1)})$.

Аналогичными рассуждениями устанавливается выводимость формул: $A_{j_2}(0^{(s+1)},\ 0^{(p_2)})$, . . . , $A_{j_v}(0^{(s+1)},\ 0^{(p_v)})$. Следовательно, выводима и конъюнкция этих формул (по правилу введения конъюнкции):

$$A_{j_1}(0^{(s+1)},\ 0^{(p_1)}) \wedge \ldots \wedge A_{j_v}(0^{(s+1)},\ 0^{(p_v)}) . \quad (22)$$

Из выводимости формул (20) и (22) следует выводимость их конъюнкции (по правилу введения конъюнкции): (23)

$$Q_k(0^{(s+1)},\ 0^{(p+1)}) \wedge A_{j_1}(0^{(s+1)},\ 0^{(p_1)}) \wedge \ldots \wedge A_{j_v}(0^{(s+1)},\ 0^{(p_v)}) .$$

Нам остаётся обосновать выводимость последнего члена конъюнкции в формуле (8″). Из формулы (8) выводим последний член конъюнкции:

$$(\forall y)[(y \neq 0^{(p_1)} \wedge \ldots \wedge y \neq 0^{(p_v)}) \rightarrow A_0(0^{(s)},\ y)] . \quad (24)$$

Из формулы (21) выводим формулу, в которой квантор общности $(\forall y)$ отнесён к каждому члену конъюнкции, а из неё выводим, в свою очередь, первый член этой конъюнкции:

$$(\forall y)[A_0(0^{(s)},\ y)\ \rightarrow\ A_0(0^{(s+1)},\ y)]\ . \qquad (25)$$

Из формулы (24), (25) выводится формула:

$$(\forall y)[(y \neq 0^{(p_1)}\ \wedge\ \ldots\ \wedge\ y \neq 0^{(p_v)})\ \rightarrow\ A_0(0^{(s+1)},\ y)]\ . \qquad (26)$$

Это утверждение основано на следующем логическом следовании:

$$\frac{\begin{array}{c}(\forall y)[P(y)\ \rightarrow\ Q(y)]\\ (\forall y)[Q(y)\ \rightarrow\ R(y)]\end{array}}{(\forall y)[P(y)\ \rightarrow\ R(y)]}\ ,$$

которое легко проверяется в логике предикатов, а затем в силу теоремы адекватности ФИП, переносится в ФИП, где становится соответствующей выводимостью.

Наконец, из формул (23) и (26) по правилу введения конъюнкции выводится формула (8″).

3) Если выполняется команда третьего вида $q_i a_j \rightarrow q_k a_j \text{Л}$, то в множестве Δ ей соответствует описывающая её формула (3). После её выполнения в момент времени $s+1$ в машине Тьюринга создастся конфигурация, которая будет описываться следующей формулой логики предикатов:

$$Q_k(0^{(s+1)},\ 0^{(p+1)})\ \wedge\ A_{j_1}(0^{(s+1)},\ 0^{(p_1)})\ \wedge\ \ldots\ \wedge\ A_q(0^{(s+1)},\ 0^{(p-1)})\ \wedge\ \ldots$$
$$\ldots\ \wedge\ A_j(0^{(s+1)},\ 0^{(p)})\ \wedge\ \ldots\ \wedge\ A_{j_v}(0^{(s+1)},\ 0^{(p_v)})\ \wedge\ (8''')$$
$$\wedge\ (\forall y)[(y \neq 0^{(p_1)}\ \wedge\ \ldots\ \wedge y \neq 0^{(p-1)} \wedge y \neq 0^{(p)}\ \wedge\ \ldots\ \wedge y \neq 0^{(p_v)})\ \rightarrow$$
$$\rightarrow\ A_0(0^{(s+1)}, y)]\ . \qquad (8''')$$

Покажите самостоятельно, что и в этом случае формула (8‴) выводится в формализованном исчислении предикатов (ФИП) из формул (3), (5), (6) и (8).

Итак, методом полной математической индукции по дискретному времени s мы доказали, что ***если машина Тьюринга не останавливается до момента времени s ($s \geq 0$), то из множества формул Δ как из гипотез выводится в ФИП формула (8), дающая описание на языке логики предикатов конфигурации машины Тьюринга в момент времени s: $\Delta \vdash (8)$.***

Вернёмся теперь к утверждению, которое мы должны доказать (доказательство достаточности), и используем для его доказательства только что доказанное утверждение. Нам дано: машина Тьюринга, получив r на входе, в конце концов останавливается в момент времени s. Требуется доказать: $\Delta \vdash H$. Поскольку до момента s машина не остановилась, то по доказанному утверждению,

имеет место выводимость: $\Delta \vdash (8)$. В свою очередь, из формулы (8) выводится следующий член конъюнкции в этой формуле: $Q_i(0^{(s)}, 0^{(p)}) \wedge A_j(0^{(s)}, 0^{(p)})$. Добавив к этой формуле 2-ую аксиому Барвайса для ФИП (см. Учебник МЛ, § 25):

$$Q_i(0^{(s)}, 0^{(p)}) \wedge A_j(0^{(s)}, 0^{(p)}) \rightarrow (\exists t)(\exists x)[Q_i(t, x) \wedge A_j(t, x)] ,$$

выводим из них формулу (по правилу MP):

$$(\exists t)(\exists x)[Q_i(t,\ x) \wedge A_j(t,\ x)] .$$

Эта формула входит одним из дизъюнктивных членов в формулу H. Поэтому H выводится из этой формулы и значит, H выводится из Δ: $\Delta \vdash H$ (правило введения дизъюнкции; Учебник МЛ, теорема 15.9 в).

Этим завершается доказательство неразрешимости формализованного исчисления предикатов, или, как говорят, логики первого порядка. □

Доказанная теорема и, в первую очередь, метод её доказательства красноречиво показывают, насколько тесно связаны между собой эти две формальные математические теории – логика предикатов как теория языка и мышления и машина Тьюринга как теория алгоритмической вычислимости: неразрешимость логики предикатов обусловлена неразрешимостью алгоритмических проблем, связанных с машинами Тьюринга. В свою очередь, связь математических теорий отражает объективно существующую связь между описываемыми ими объективными явлениями – мышлением логическим и мышлением алгоритмическим. Одним словом, доказанная теорема даёт просторное поле для глубоких философских размышлений и обобщений.

Глава X

АЛГОРИТМИЧЕСКИЕ ПРОБЛЕМЫ МАТЕМАТИКИ

В заключительной главе мы коснёмся ряда алгоритмических проблем из различных разделов математики. Первый параграф посвятим знаменитой 10-ой проблеме Гильберта о диофантовых уравнениях, решение которой растянулось на 70 лет в XX столетии. В параграфе 10.2 рассмотрим проблему тождества слов и ряд проблем из различных разделов алгебры и топологии. Наконец в заключительном параграфе 10.3 отметим направление в изучении алгоритмических проблем математики, связанное с анализом сложности этих проблем в том аспекте, в каком сложность обсуждалась нами в главе VIII.

10.1. Десятая проблема Д.Гильберта

История проблемы. На заре XX века, в 1900 г., в Париже состоялся Всемирный математический конгресс, на котором один из величайших математиков того времени Давид Гильберт (1862 – 1943) представил 23 проблемы из различных областей математики. В этих фундаментальных проблемах он попытался "проникнуть в предстоящие успехи нашей науки и тайны её развития в начинающемся столетии." Попытка эта оказалась необычайно успешной и ценной. Проблемы действительно были поставлены в узловых точках будущего развития различных математических теорий. Не исключением явилась и проблема, которая в перечне Гильберта получила номер 10. Её формулировка по Гильберту такова: "*Пусть дано произвольное диофантово (т.е. алгебраическое с целыми коэффициентами) уравнение с произвольным числом неизвестных...; требуется указать общий метод, следуя которому можно было бы в конечное число шагов узнать, имеет данное уравнение решение в целых... числах или нет.*" Эта проблема была, пожалуй, единственной из всех проблем Гильберта, наиболее ярко иллюстрировавшей обязательное требование, предъявлявшееся Гильбертом к действительно фундаментальной проблеме: "возможность изложить её содержание первому встречному."

Конечно, Гильберт сформулировал свою 10-ую проблему не в терминах теории алгоритмов, ибо в начале XX века такой теории просто не существовало. Более того, речь шла о положительном решении проблемы, т.е. не о том, существует или не существует "общий метод", а об указании "общего метода." Мысль о том, что такого "общего метода" может просто не существовать, в 1900 г. никому не приходила в голову.

Задача о решении уравнений в целых числах является древней и очень трудной задачей. Ею занимались сам Пифагор в VI в. до н.э. и Диофант (III в. н.э.), по имени которого и названы уравнения. Основная трудность состоит в том, чтобы выяснить, имеет ли данное уравнение хотя бы одно целочисленное решение. Решением таких уравнений различных видов собственно и занимались математики на протяжении многих веков. Например, до сих пор неизвестно, имеют ли решения в целых числах диофантовы уравнения $x^3 + y^3 + z^3 = 30$ и $x^4 + y^4 + z^4 = u^4$. Гильберт на заре XX в. высветил новую грань этой проблемы: он предложил сосредоточиться на поиске "общего метода" её решения. Так поставленная Гильбертом проблема несомненно явилась одним из стимулов и толчков к созданию теории алгоритмов, а последовавшее затем доказательство алгоритмической неразрешимости этой массовой проблемы явилось ещё одним косвенным подтверждением фактической трудности каждой её единичной задачи.

В связи с этой проблемой интересно отметить такой факт: если относительно какого-либо диофантова уравнения известно, что оно имеет целочисленное решение, то алгоритм отыскания этого решения существует. Действительно, вот такой алгоритм: надо просто подставлять в уравнение всевозможные наборы натуральных чисел; в конце концов будет найден набор, удовлетворяющий уравнению.

Было затрачено немало усилий, чтобы отыскать этот "общий метод." Задача о целых решениях произвольного диофантова уравнения легко сводится к задаче о натуральных (целых неотрицательных) решениях. Далее, было показано, что достаточно ограничиться диофантовыми уравнениями степени не выше четвёртой. Для диофантовых уравнений степени не выше второй искомый общий метод был найден. Но дальше этого дело не шло: уже уравнения третьей степени не поддавались никаким усилиям. В связи с этими неудачами возникло подозрение, что тот общий метод, об отыскании которого говорится в формулировке Гильберта, не существует. Сходная ситуация складывалась и в ряде других задач аналогичного характера.

Как мы знаем, лишь в 30-ых годах XX века оформилось математически точное понятие алгоритма, а к концу 40-ых годов вера в адекватность этого понятия укрепилась уже настолько, что можно было всерьёз ставить вопрос об отрицательном решении 10-ой проблемы Гильберта и ряда других математических проблем, касающихся существования алгоритмов. В начале 50-ых годах появились первые работы, направленные на доказательство несуществования такого алгоритма. В 1970 г. советский математик Ю.В.Матиясевич завершил доказательство алгоритмической неразрешимости 10-ой проблемы Гильберта. В настоящем параграфе излагаются основные идеи этого доказательства.

Диофантовы множества и их связь с перечислимыми множествами. Напомним (см. § 6.2), что перечислимые множества – это такие, которые могут быть порождены некоторым алгоритмом, или которые являются множествами значений некоторых вычислимых функций.

С перечислимыми множествами тесно связаны диофантовы множества. Пусть $M \subseteq N$. Множество M называется *диофантовым*, если существует многочлен $p(x, y_1, ..., y_n)$ с целыми коэффициентами, такой, что для всех $x \in N$ имеем:

$$x \in M \iff (\exists y_1 \in N)...(\exists y_n \in N)[p(x, y_1, ..., y_n) = 0] \ .$$

Например, взяв многочлен $p(x, y_1, y_2) = y_1^2 + y_2^2 - x^2$, мы получаем диофантово множество:

$$M = \{x \in N : \ (\exists y_1 \in N)(\exists y_2 \in N)[y_1^2 + y_2^2 - x^2] = 0\} \ ,$$

которому, в частности, принадлежат числа 5 (при этом $y_1 = 3$, $y_2 = 4$), 10, 20 и т.д.

Следующая теорема достаточно проста, но она выявляет важную взаимосвязь между перечислимыми множествами и диофантовыми множествами.

ТЕОРЕМА 10.1.1. *Всякое диофантово множество является перечислимым множеством.*

ДОКАЗАТЕЛЬСТВО. Пусть множество $M \subseteq N$ диофантово. Рассмотрим тогда квазихарактеристическую функцию множества M (см. § 6.4):

$$h_M(x) = \begin{cases} 1, & \text{если } x \in M, \\ \text{не определено}, & \text{если } x \notin M. \end{cases}$$

Покажем, что эта функция вычислима. Алгоритм для её вычисления действует следующим образом. Пусть x – элемент из N. Перебираем последовательно все строки вида $(y_1, ..., y_n) \in N^n$ и проверяем выполнимость условия: $p(x, y_1, ..., y_n) = 0$. (Все строки указанного вида можно, следуя канторовскому диагональному процессу, занумеровать натуральными числами (линейно упорядочить): сначала нумеруются все строки веса 0, затем – все строки веса 1, затем – веса 2 и т.д. Перебор и проверка осуществляются в полученной последовательности). Если при этом $x \in M$, то за конечное число шагов мы обнаружим этот факт, и на выходе алгоритма появится результат: 1. Если же $x \notin M$, то процесс перебора строк никогда не закончится и будет продолжаться бесконечно, что будет соответствовать тому, что для этого x значение $h_M(x)$ не определено.

Это означает, что рассматриваемый алгоритм действительно вычисляет функцию $h_M(x)$, т.е. эта функция вычислима. Тогда по теореме 6.4.5 отсюда следует, что множество M перечислимо.

Теорема доказана. □

Оказывается, справедливо и утверждение, обратное по отношению к доказанной теореме, но его доказательство потребовало от математиков значительно больших усилий.

Теорема 10.1.2. *Пусть* $M \subseteq N$ *и* M *– перечислимое множество. Тогда множество* M *является диофантовым множеством.*

Впервые гипотезу о диофантовости перечислимых множеств высказал Мартин Дэвис, когда в 50-ые годы XX в. группа американских математиков (Р.Робинсон, Х.Путнам, Дж.Робинсон, М.Дэвис) получила ряд значительных и обнадёживающих результатов на пути решения 10-ой проблемы Гильберта. Ценою значительных усилий им удалось свести задачу к доказательству диофантовости отношения $y = x^u$. Джулия Робинсон пошла даже несколько дальше, показав, что достаточно построить конкретное уравнение $R(u, v, x_1, \ldots, x_k) = 0$, не допускающее решение с $v > u^u$, но для каждого n имеющее решение с $v > u^n$. Точку в этих исследованиях поставил Ю.В.Матиясевич, которому именно такого рода уравнение и удалось построить. Он предложил вполне элементарную, но чрезвычайно остроумную и оригинальную конструкцию, связанную с последовательностью чисел Фибоначчи. При этом, ему пришлось решать задачу необычную для традиционной теории чисел. В теории чисел по заданному уравнению, как правило, исследуются свойства его решений; здесь же наоборот, задавшись определёнными свойства ре-

шений, нужно было искать требуемое уравнение. Ю.В.Матиясевич с ней блестяще справился, доказав тем самым теорему 10.1.2.

С подробностями доказательств можно познакомиться по работам [38], [81], [82], [83] .

Решение 10-ой проблемы Гильберта. Итак, доказанная Ю.В.Матиясевичем теорема 10.1.2 позволяет по каждому перечислимому множеству M строить такое диофантово уравнение $p(x, y_1, ..., y_n) = 0$, которое имеет натуральные решения y_1, ... , y_n для всех x, принадлежащих M, и только для таких x.

Отсюда легко получается отрицательное решение 10-ой проблемы Гильберта.

ТЕОРЕМА 10.1.3. *Не существует общего метода (алгоритма), позволяющего для любого заданного диофантова уравнения установить, имеет оно решение в целых числах или нет.*

ДОКАЗАТЕЛЬСТВО. Допустим противное, т.е. указанный алгоритм существует. Назовём его условно "алгоритмом Гильберта". Возьмём тогда в качестве множества M такое перечислимое множество, которое не является разрешимым. (Такие множества, как мы знаем, существуют – см. теоремы 6.3.4 и 6.4.3). Тогда, по теореме 10.1.2, множество M – диофантово, т.е. для него существует диофантово уравнение $p(x, y_1, ..., y_n) = 0$, обладающее указанным выше свойством. Покажем тогда, что множество M разрешимо. В самом деле, возьмём $x = 0$. Используя алгоритм Гильберта, мы за конечное число шагов можем узнать, имеет ли уравнение $p(0, y_1, ..., y_n) = 0$ целочисленные решения. В силу свойства многочлена p, это означает, что мы можем узнать, принадлежит ли число 0 множеству M. Взяв $x = 1$, мы с помощью алгоритма Гильберта за конечное число шагов выясняем, имеет ли уравнение $p(1, y_1, ..., y_n) = 0$ целочисленные решения, и значит, выясняем, принадлежит ли число 1 множеству M. И так далее. Таким образом, алгоритм Гильберта фактически становится разрешающим алгоритмом для множества M, и множество M оказывается разрешимым, что противоречит его выбору. Следовательно, гипотетического "алгоритма Гильберта", решающего 10-ую проблему Гильберта, не существует, и эта проблема алгоритмически неразрешима. □

Следствия из теоремы Ю.В.Матиясевича. Эта теорема 10.1.2 имеет ряд интересных следствий, из которых наиболее эффектным, по-видимому, является следствие, связанное с множеством простых чисел.

Теорема 10.1.4. *Пусть $M \subseteq N$. Множество M является перечислимым множеством тогда и только тогда, когда M является множеством неотрицательных значений некоторого многочлена $p(x_1, ..., x_n)$ с целыми коэффициентами, при условии, что переменные x_1, ... , x_n пробегают все значения из N.*

Доказательство. Достаточность. Пусть M – множество неотрицательных значений многочлена $p(x_1, ..., x_n)$ с целыми коэффициентами, при условии, что переменные x_1, ... , x_n пробегают все значения из N, т.е.

$$M = \{x \in N : (\exists y_1 \in N)(\exists y_2 \in N)[p(y_1, ..., y_n) = x]\} .$$

Тогда, как и в доказательстве теоремы 10.1.1, доказывается, что множество M перечислимо.

Необходимость. Пусть множество M перечислимо. Тогда по теореме 10.1.2 Ю.В.Матиясевича, множество M является диофантовым. Это означает, что существует такой целочисленный многочлен $p(x_1, ..., x_n)$, что

$$M = \{x \in N : (\exists y_1 \in N)(\exists y_2 \in N)[p(y_1, ..., y_n) = 0]\} .$$

Построим новый многочлен:

$$P(x, y_1, ..., y_n) = x - (x + 1) \cdot [p(x_1, ..., x_n)]^2 .$$

Ясно, что значение нового многочлена неотрицательно тогда и только тогда, когда $p(y_1, ..., y_n) = 0$, т.е. когда $x \in M$; причём, в этом случае его значение равно x. Таким образом, множество M представляет собой неотрицательных значений целочисленного многочлена $P(x, y_1, ..., y_n)$, при условии, что его переменные пробегают все значения из N.

Теорема доказана. □

Пример 10.1.5. Рассмотрим в качестве M множество всех простых чисел. Оно, как известно, разрешимо и, следовательно, тем более перечислимо. Тогда по теореме 10.1.4 оно исчерпывается положительными значениями некоторого целочисленного многочлена. Укажем явно такой многочлен $F(a, b, c, ..., z)$ (см.[1], стр. 43). Он имеет степень 25 и содержит 26 переменных – все буквы латинского алфавита:

$$F(a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z) =$$

---

[1] *Боро В., Цагир Д., Рольфс Ю., Крафт Х., Янцен Е.* Живые числа. Пять экскурсий. – М.: Мир, 1985.

$$= [k+2]\cdot[1-(wz+h+j-q)^2-(2n+p+q+z-e)^2-(a^2y^2-y^2+1-x^2)^2-$$
$$-((e^4+2e^3)(a+1)^2+1-o^2)^2-(16(k+1)^3(k+2)(n+1)^2+1-f^2)^2-$$
$$-(((a+u^4-u^2a)^2-1)(n+4dy)^2+1-(x+cu)^2)^2-(ai+k+1-l-i)^2-$$
$$-((gk+2g+k+1)(h+j)+h-z)^2-(16r^2y^4(a^2-1)+1-u^2)^2-$$
$$-(p-m+l(a-n-1)+b(2an+2a-n^2-2n-2))^2-$$
$$-(z-pm+pla-p^2l+t(2ap-p^2-1))^2-$$
$$-(q-x+y(a-p-1)+s(2ap+2a-p^2-2p-2))^2-$$
$$-(a^2l^2-l^2+1-m^2)^2-(n+l+v-y)^2]\ .$$

Десятая проблема Гильберта с точки зрения логики предикатов. Подобно понятию диофантова множества вводится понятие диофантова предиката. Собственно, это такие предикаты, которые задают диофантовы множества, т.е. множества истинности которых являются диофантовыми множествами.

n-местный предикат от переменных $x_1, \dots . x_n$ называется *диофантовым*, если он имеет следующий вид:

$$(\exists y_1)\dots(\exists y_m)[p(x_1,\dots,x_n,y_1,\dots,y_m) = 0]\ ,$$

где $p(x_1,\dots,x_n,y_1,\dots,y_m)$ – многочлен с целыми коэффициентами, или равносилен предикату, имеющему такой вид.

Например, предикат "*Число x есть полный квадрат*" диофантов, так как он равносилен предикату $(\exists y)(x-y^2-0)$.

Подобно тому, как диофантовы множества оказались тесно связанными с перечислимыми множествами, диофантовы предикаты оказываются тесно связанными с частично разрешимыми предикатами.

ТЕОРЕМА 10.1.6. *Всякий диофантов предикат является частично разрешимым.*

Доказательство. Ясно, что для всякого многочлена p предикат $p(x_1,\dots,x_n,y_1,\dots,y_m) = 0$ разрешим, поскольку для любых натуральных чисел $x_1,\dots,x_n,y_1,\dots,y_m$ легко эффективно проверить, превращают ли они многочлен p в 0. Но тогда этот предикат частично разрешим и по следствию 7.3.6 будет частично разрешим и диофантов предикат

$$(\exists y_1)\dots(\exists y_m)[p(x_1,\dots,x_n,y_1,\dots,y_m) = 0]\ . \quad \square$$

Несмотря на то, что диофантовы предикаты имеют весьма специфический вид, оказалось, что ими-то , с точностью до равносильности предикатов, исчерпываются все частично разрешимые предикаты. Этот результат был получен Ю.В.Матиясевичем в 1970 г.

Теорема 10.1.7. (Ю.В.Матиясевич, 1970) *Всякий частично разрешимый предикат является диофантовым.*

Основная трудность доказательства состоит в том, чтобы доказать, что класс диофантовых предикатов замкнут относительно операции взятия ограниченного квантора общности, т.е. если предикат $P(x_1, ..., x_n, y)$ диофантов, то диофантовым будет и предикат $(\forall z < y)P(x_1, ..., x_n, z)$. Отметим, что в теореме 7.3.8 мы без труда установили замкнутость класса частично разрешимых предикатов относительно операций взятия ограниченных кванторов общности и существования.

Эта теорема Ю.В.Матиясевича позволяет легко получить отрицательное решение 10-й проблемы Гильберта, т.е. теорему 10.1.3.

Второе доказательство теоремы 10.1.3. Сначала покажем, что к 10-ой проблеме Гильберта о решении диофантовых уравнений в целых числах может быть сведена проблема решения диофантовых уравнений в натуральных числах, т.е. если сможем решить первую проблему, то сможем решить и вторую. Предположим, что поставленная Гильбертом проблема разрешима и её решает некий алгоритм, который условно назовём "алгоритмом Гильберта". Нам требуется решить в натуральных числах диофантово уравнение $p(x_1, ..., x_n) = 0$. Из теории чисел известно, что всякое натуральное число можно представить в виде суммы квадратов четырёх натуральных чисел. Сделаем тогда в данном уравнении замену:

$$x_1 = s_1^2 + t_1^2 + u_1^2 + v_1^2 ,$$
$$\ldots\ldots\ldots\ldots\ldots$$
$$x_n = s_n^2 + t_n^2 + u_n^2 + v_n^2 .$$

Получим снова диофантово уравнение:

$$p(s_1^2 + t_1^2 + u_1^2 + v_1^2, \ldots, s_n^2 + t_n^2 + u_n^2 + v_n^2) = 0$$

(с числом неизвестных, в четыре раза большим, чем в исходном уравнении). С помощью "алгоритма Гильберта" находим его целочисленные решения s_i, t_i, u_i, v_i, $(i = 1, 2, ..., n)$. Подставляем их в формулы замены для x_1, ... x_n и находим натуральные решения исходного уравнения $p(x_1, ..., x_n) = 0$.

Обратимся теперь к доказательству теоремы. Сведём к проблеме разрешимости диофантовых уравнений в натуральных числах какую-нибудь заведомо неразрешимую проблему. Это будет означать, что из разрешимости проблемы разрешимости диофантовых уравнений в натуральных числах последовала бы разрешимость этой неразрешимой проблемы, что невозможно.

Если в первом доказательстве теоремы 10.1.3 мы, далее, выбирали перечислимое, но не разрешимое множество, то сейчас выберем предикат, являющийся частично разрешимым, но не являющийся разрешимым. Возьмём в качестве него предикат "$x \in W_x$" (характеризующий проблему самоприменимости алгоритмов). Так как он частично разрешим, то по теореме 10.1.7, он является диофантовым, т.е. существует многочлен $p(x, y_1, ..., y_m)$ с целыми коэффициентами, такой, что

$$x \in W_x \iff (\exists y_1)...(\exists y_m)[p(x, y_1, ..., y_m) = 0] \ .$$

Тогда если бы существовала разрешающая процедура для решения диофантовых уравнений в натуральных числах, то с её помощью мы бы построили следующую разрешающую процедуру для проблемы "$x \in W_x$": чтобы выяснить, верно ли, что $a \in W_a$, нужно с помощью указанной процедуры определить, разрешимо ли в *натуральных числах* диофантово уравнение $p(a, y_1, ..., y_m) = 0$. Если разрешимо, то утверждение "$a \in W_a$" истинно; если не разрешимо, то утверждение "$a \in W_a$" ложно.

Поскольку проблема "$x \in W_x$" не разрешима (теорема 7.2.1), поэтому не разрешима и проблема Гильберта в натуральных числах. А раз так, то на основании первого абзаца доказательства, не существует алгоритма, решающего проблему Гильберта и в целых числах. □

Две модификации 10-ой проблемы Гильберта. Эти модификации связаны с рассмотрением решений диофантовых уравнений не над множеством Z целых чисел (как это имело место в проблеме Гильберта), а над множеством R действительных чисел и множеством Q рациональных чисел. Проблемы также состоят в существовании алгоритмов, позволяющих для любого заданного алгебраического уравнения с целыми коэффициентами устанавливать, имеет оно или не имеет решение: 1) в множестве R действительных чисел; 2) в множестве Q рациональных чисел.

Первая из этих проблем была положительно решена польским логиком и математиком А.Тарским в 40-ые годы XX века. Вторая остаётся нерешённой до сих пор.

10.2. Проблема тождества слов и другие математические проблемы

Истоки проблемы тождества слов. Хорошо известна детская игра в слова, в которой нужно преобразовать одно слово в другое, последовательно заменяя в словах точно одну букву на другую букву так, чтобы каждый раз получалось осмысленное слово. Например, преобразовать слово ВОЛК в слово КОЗА. Решение: ВОЛК – ПОЛК – ПОЛА – ПОЗА – КОЗА.

Можно упростить задачу, отменив требование осмысленности промежуточных слов. Тогда исходная задача допускает более короткое решение: ВОЛК – КОЛК – КОЗК – КОЗА.

Сразу возникают две алгоритмические проблемы: существует ли алгоритм, позволяющий для любых двух слов в русском алфавите установить, получается ли одно из другого последовательностью замен буквы на букву (с осмысленными промежуточными словами или с бессмысленными). Очевидно, что во втором случае преобразование одного слова в другое возможно в том и только в том случае, когда слова имеют одинаковую длину (т.е. одинаковое число букв). Так что требуемый алгоритм существует: он сравнивает длины данных слов и, если они совпадают, выдаёт ответ ДА; если не совпадают, то ответ – НЕТ. Можно показать, что разрешима также и первая алгоритмическая проблема.

Из школьного курса алгебры хорошо известна проблема доказательства тождеств. Тождества строятся из букв (обозначающих переменные, пробегающие множество вещественных чисел) и вещественных чисел с помощью действий сложения, вычитания и умножения. Хорошо также известен и следующий общий приём (алгоритм) решения этой массовой проблемы. Используя распределительный закон для умножения, раскрывают скобки в левой и правой частях любого данного тождества и осуществляют приведение подобных членов в соответствии с хорошо известными правилами. После осуществления всех этих преобразований как левая, так и правая части исходного тождества превращаются в полиномы (многочлены). Тождество будет справедливым в том и только в том случае, когда эти многочлены тождественно совпадут друг с другом. Другими словами, справедливость тождества означает, что после перенесения всех членов преобразованного тождества в одну часть эти

члены взаимно уничтожаются, в результате чего тождество превращается в тривиальное тождество $0 = 0$.

Таким образом, проблема тождества в элементарной алгебре алгоритмически разрешима: существует единый конструктивный приём (алгоритм), позволяющий за конечное число шагов решить, представляет ли любое заданное соотношение тождественное соотношение или нет.

Тем не менее, можно построить примеры таких алгебраических систем, в которых проблема тождества является алгоритмически неразрешимой проблемой. В качестве таких алгебраических систем могут быть, например, выбраны полугруппы или группы, заданные системами образующих элементов и определяющих соотношений.

Проблема слов в теории полугрупп. В математике под *алфавитом* понимается любое конечное множество различных символов. Символы, составляющие алфавит, называются *буквами*. Например, $\{\beta, ъ, ?, 5, *\}$ – алфавит; β, ъ, ?, 5, $*$ – буквы. *Словом* в алфавите A называется любая конечная последовательность $a_1a_2...a_m$, составленная из букв данного алфавита. Для алфавита A символом A^* обозначается множество всех слов в этом алфавите. Например, если $A = \{0, 1\}$, то $A^* = \{0, 1, 00, 01, 10, 11, 000, 001, 010, ...\}$. Рассматривают также и пустое слово: под ним понимают слово, не содержащее ни одной буквы; его обозначают Λ.

Конкатенацией (от лат. *catena* – цепь) или *сшиванием* слов $L = a_1a_2...a_m$ и $M = b_1b_2...b_n$ называется новое слово $N = L \cdot M = a_1a_2...a_mb_1b_2...b_n$, в котором первые m букв образуют слово L, а следующие n букв – слово M. Ясно, что операция конкатенации обладает свойством ассоциативности:

$$(\forall X, Y, Z)\ [(XY)Z = X(YZ)]\ .$$

Всякая алгебра с одной бинарной операцией, обладающей свойством ассоциативности, называется *полугруппой*. Таким образом, $< A^*; \cdot >$ – полугруппа. Она называется *полугруппой слов* или *свободной полугруппой* над алфавитом A, а элементы из A называются её *образующими*. Эта полугруппа не коммутативная.

Пустое слово Λ относительно операции конкатенации обладает свойством: $(\forall X)(X\Lambda = \Lambda X = X)$, т.е. является единичным (нейтральным) элементом. Если пустое слово Λ добавить к A^*, то мы получаем свободную полугруппу с единицей $< A^* \cup \{\Lambda\}; \cdot >$.

Пусть L, M – слова в алфавите A. Если L является частью M, то говорят, что слово L *входит* в слово M, или что имеется вхождение L в M. Например, для слов $L = acb$ и $M = bbacbc$ в алфавите

$A = \{a, b, c\}$ слово L входит в слово M. Одно слово может входить в другое несколько раз.

Опишем теперь процесс преобразования слов, который будет обобщать процесс преобразования слов в той детской игре в слова, с которой мы начали этот параграф. Зададим в данном алфавите A систему допустимых подстановок: $P - Q$, $L - M$, ... , $S - T$, где $P, Q, L, M, ..., S, T$ – слова в алфавите A. Любую подстановку вида $L - M$ можно применять к некоторому слову X в алфавите A следующим способом: если в слове X имеется одно или несколько вхождений слова L, то любое из этих вхождений может быть заменено словом M, и наоборот, если имеется вхождение слова M, то его можно заменить словом L. Например, подстановка *ac* – *cbc* применима четырьмя способами к слову *acbcbcac*. Замена каждого из двух вхождений *ac* даёт слова *cbcbcbcac* и *acbcbccbc*, а замена каждого из двух вхождений *cbc* даёт слова *acbcac* и *acbacac*. Допустимые подстановки называют также ***определяющими соотношениями.***

К полученным с помощью допустимых подстановок словам можно снова применить допустимые подстановки, в результате чего получаются новые слова и т.д. Все получаемые таким образом слова называются ***эквивалентными*** между собой, или ***равносильными***, или ***равными***, или ***тождественными.***

Здесь возникает массовая проблема, называемая ***проблемой*** (тождества, равенства) ***слов***, или ***проблемой тождества***: являются ли равными (эквивалентными) два данных слова в данном алфавите при заданной системе допустимых подстановок (определяющих соотношений)? Существует ли алгоритм для решения этой массовой проблемы?

Согласно результатам, полученным независимо друг от друга А.А. Марковым в 1946 г. и Э.Постом в 1947 г., алгоритма для решения этой проблемы (с произвольным набором допустимых подстановок) не существует. Более того, они построили конкретные примеры систем допустимых подстановок, для которых не существует нормального алгоритма (см. главу IV), распознающего эквивалентность слов. Отсюда, в силу тезиса А.А.Маркова, не существует вообще никакого алгоритма, распознающего эквивалентность слов при данной системе подстановок. Следовательно, и подавно в своём общем виде проблема алгоритмически неразрешима.

Примеры систем подстановок, приведённые А.А.Марковым и Э.Постом, были весьма громоздкими и насчитывали сотни допустимых подстановок. Позже советский математик Г.С.Цейтин [20] при-

вёл сравнительно простой пример системы допустимых подстановок для слов в пятибуквенном алфавите $\{a, b, c, d, e\}$, содержащей лишь семь подстановок: $ac - ca$, $ad - da$, $bc - cb$, $bd - db$, $abac - abace$, $eca - ae$, $edb - be$, для которой проблема слов алгоритмически неразрешима. Отметим, что алгоритмическая неразрешимость этой проблемы доказана им методом сведения к этой проблеме некоторой вспомогательной проблемы, неразрешимость которой уже была установлена ранее. Эта вспомогательная проблема представляет собой частный случай общей проблемы тождества для ассоциативных исчислений.

Проблему слов в общем виде впервые поставил норвежский математик Аксель Туэ в 1914 г. задолго до появления точного определения понятия алгоритма (как и 10-ая проблема Гильберта). Сам он доказал существование алгоритма в ряде частных случаев. Он особо выделил случаи, когда система допустимых подстановок состоит лишь из таких подстановок, что один из её членов есть пустое слово $L - \Lambda$, и таким образом, всякая разрешённая замена состоит в том, что можно в любое место имеющегося слова вставить указанное в подстановке слово или вычеркнуть его. В 1966 г. советский математик С.И.Адян доказал, что для любого набора допустимых вставок-вычёркиваний алгоритм, устанавливающий, эквивалентны или нет два заданных слова, существует, т.е. этот частный случай общей проблемы слов оказывается алгоритмически разрешим (см. [28]).

Задавая систему подстановок на множестве A^* всех слов в алфавите A, мы задаём на этом множестве отношение эквивалентности, которое разбивает это множество на попарно непересекающиеся классы эквивалентных слов. Можно показать, что эти классы образуют полугруппу относительно некоей глобальной операции над этими классами; причём, в этой полугруппе будут выполняться тождества, задаваемые допустимыми подстановками (или, как их ещё называют, определяющими соотношениями). Поэтому говорят о проблеме тождества для полугрупп, задаваемых порождающим множеством A и системой определяющих соотношений.

Алгоритмические проблемы слов в теории групп. Проблема тождества слов была распространена с теории полугрупп на теорию групп. При этом, слова здесь рассматриваются не в языке с одной бинарной операцией $\cdot$ (как это было для полугрупп), а в языке с двумя операциями: бинарной операцией $\cdot$ (умножение) и унарной операцией $^{-1}$ (обращение). В 1950-ые годы советский математик П.С.Новиков доказал алгоритмическую неразрешимость

этой проблемы. Кроме того, он доказал также, в частности, что не существует алгоритма, который для любых двух предъявленных групп распознавал бы, изоморфны они или нет. (Обзор алгоритмических проблем для групп и полугрупп приведён в статье [28]).

Проблема нахождения нулей многочлена. Приведём классический пример массовой проблемы из алгебры, имеющей алгоритмическое решение. Проблема состоит в том, чтобы узнать, сколько действительных корней имеет данный многочлен с действительными коэффициентами, или сколько действительных корней имеет данный многочлен с действительными коэффициентами в данном интервале $[a, b]$.

Известна классическая теорема Штурма, дающая алгоритм решения этой проблемы.

ТЕОРЕМА 10.2.1. (Штурм). *Пусть $p(x)$ – многочлен с действительными коэффициентами, и пусть p_0, p_1, ..., p_r – последовательность многочленов с действительными коэффициентами, определяемая следующим образом (называемая последовательностью Штурма):*

(a) $p_0 = p$,
(b) $p_1 = p'$ *(производная от p),*
(c) *для любого $0 < i < r$ существует многочлен q_i, такой, что $p_{i-1} = p_i q_i - p_{i+1}$, где $p_{i+1} \neq 0$ и степень многочлена p_{i+1} меньше степени многочлена p_i (тем самым многочлены q_i и $-p_{i+1}$ являются соответственно частным и остатком от деления многочлена p_{i-1} на многочлен p_i),*
(d) $p_{r-1} = p_r q_r$.

Для любого действительного числа c обозначим через $\delta(c)$ число перемен знака в последовательности $p_0(c)$, ..., $p_r(c)$ (нули при этом игнорируются).

Пусть a и b – действительные числа, которые не являются корнями $p(x)$, и пусть $a < b$. Тогда количество корней многочлена $p(x)$ в интервале $[a, b]$ равно $\delta(a) - \delta(b)$ (при этом каждый корень учитывается только один раз).

Мы конечно же не будем приводить здесь доказательство теоремы Штурма. С ним читатель может познакомиться, например, по книге [22], с. 288 – 290. Для наших целей теорема Штурма представляет интерес в связи с тем, что в ней содержится алгоритм

решения рассматриваемой проблемы. Она даёт положительный результат о вычислимости количества корней многочлена и о разрешимости утверждений о корнях многочлена.

Для формулировки таких результатов нам ограничимся рассмотрением многочленов с рациональными коэффициентами. Тогда объекты, с которыми мы будем иметь дело, будут конечными. Множество рациональных чисел обозначим через Q, и, таким образом, мы будем вести все рассуждения в терминах вычислимости над Q (которую, впрочем, можно определить в терминах вычислимости над N с помощью обычного кодирующего устройства). Заметим, что любой многочлен $p(x)$ с коэффициентами из Q есть, по сути дела, последовательность рациональных чисел.

Из теоремы Штурма вытекают, в частности, следующие результаты.

ТЕОРЕМА 10.2.2. (a) *Существует эффективная процедура вычисления числа действительных корней многочлена с коэффициента из* Q;

(b) Предикат "*многочлен* p *имеет корень на* $[a,b]$" разрешим, где p есть многочлен с коэффициентами из Q и $a,b \in Q$.

Доказательство. Для любого заданного многочлена p многочлены p_0, p_1, ... , p_r, определяемые в теореме Штурма, могут быть найдены эффективно с помощью стандартных правил дифференцирования и алгоритма деления многочленов.

(a). Эта процедура состоит в следующем. Для любого многочлена легко строится рациональное число $L > 0$, такое, что все корни многочлена p принадлежат интервалу $]-L, L[$. В самом деле, пусть $p(x) = a_0 + a_1 x + ... + a_n x^n$. Тогда достаточно взять число:

$$L = 1 + \frac{1}{|a_n|}(|a_0| + |a_1| + ... + |a_n|) .$$

Тогда по теореме Штурма количество корней многочлена $p(x)$ равно числу $\delta(-L) - \delta(L)$, которое вычисляется эффективно.

(b) Допустим, что нам дан многочлен $p(x)$ и рациональные числа a, b. Чтобы решить вопрос о том, имеет ли $p(x)$ корень на $[a, b]$, вычислим сначала значения $p(a)$ и $p(b)$. Если ни одно из этих чисел не равно нулю, то вычислим число $\delta(a) - \delta(b)$ и применим теорему Штурма. □

Разумеется, теорему Штурма можно использовать и для того, чтобы показать, что многие другие вопросы, касающиеся многочленов с коэффициентами из Q, являются вычислимыми или разрешимыми.

Покажите, например, что существует эффективная процедура нахождения для заданного многочлена $p(x)$ и заданных рациональных чисел a, b количества корней этого многочлена $p(x)$ на отрезке $[a, b]$. (Не забудьте, что числа a и b тоже могут быть корнями $p(x)$).

Алгоритмические проблемы в других разделах математики. М.Пресбургер в 1929 г. доказал разрешимость арифметики сложения натуральных чисел, а А.Чёрч в 1930-ые годы наряду с неразрешимостью исчисления предикатов установил также неразрешимость арифметики сложения и умножения натуральных чисел. Эти результаты открыли перспективу изучения проблемы разрешимости и вызвали массу исследований о разрешимости и неразрешимости различных математических теорий.

Начнём с элементарной геометрии. Пусть L – язык первого порядка, подходящий для выражения утверждений евклидовой геометрии на плоскости. Это означает, что в L имеется список переменных, областью значения которых являются точки плоскости; два трёхместных предиката:

$L(x, y, z)$: "x, y, z *лежат на одной прямой*" и

$M(x, y, z)$: "y *лежит между* x *и* z" ;

два шестиместных предиката:

$T(x, y, z, u, v, w)$: "$\Delta xyz \cong \Delta uvw$" (конгруэнтность треугольников)

и

$C(x, y, z, u, v, w)$: "$\angle xyz = \angle uvw$" (равенство величин углов) ;

четырёхместный предикат:

$E(x, y, u, v)$: "$[xy] = [uv]$" (равенство длин отрезков).

Например, формула

$$(\forall x, y, z, u, v)\{[C(x, y, v, z, y, v) \wedge C(x, z, u, y, z, u) \wedge M(x, y, u) \wedge \\ \wedge M(x, z, v) \wedge E(u, z, v, y)] \rightarrow E(x, y, x, z)\}$$

выражает в L известный из школьной геометрии факт, что если в треугольнике биссектрисы двух углов равны, то данный треугольник равнобедренный.

Польский математик Альфред Тарский в 1951 г. доказал, что теория EG, содержащая все предложения из L, истинные в евклидовой геометрии на плоскости, – так называемая элементарная геометрия – разрешима. В действительности, А.Тарский доказал более сильный результат: теория первого порядка поля действительных чисел разрешима. Отсюда выводится разрешимость элементарной геометрии путём введения декартовых координат и сведения геометрических утверждений к эквивалентным алгебраическим утверждениям.

Была также доказана разрешимость теорий рациональных чисел, алгебраически замкнутых полей, булевых алгебр, линейно упорядоченных множеств, вещественно замкнутых полей, поля p-адических чисел, класса всех конечных полей, коммутативных групп.

Из массовых проблем из области топологии выделяется классическая проблема распознавания гомеоморфизма для n-мерных многообразий. А.А.Марков построил пример четырёхмерного многообразия, для которого невозможен алгоритм, распознающий гомеоморфность ему любого другого четырёхмерного многообразия. Известно, что для $n = 2$ эта проблема алгоритмически разрешима. Для $n = 3$ вопрос остаётся открытым.

С обзором результатов по неразрешимым алгоритмическим проблемам и разрешимым математическим теориям можно познакомиться по статьям [50], [55], [94].

10.3. О дальнейших направлениях изучения алгоритмических проблем логики и математики

Разрешимые математические теории и программа Гильберта. На результаты, полученные при решении проблемы разрешимости различных математических теорий, можно посмотреть с точки зрения программы Гильберта обоснования математики. Идея Гильберта состояла в том, чтобы записать (закодировать) на формальном логико-математическом языке все математические утверждения, выделить в различных разделах математики системы аксиом, которые также записать на формальном логико-математическом языке, а затем, используя правила вывода формальной логики, выводить из этих формул-аксиом формулы-теоремы. Гильберт надеялся, что такая формализация превратит доказательство математических результатов в механическую игру с цепочками символов.

Кроме того, этот метод позволил бы дать исчерпывающий список всех формальных теорем любой математической теории, а это помогло бы доказать, что в этой теории не существует никакого формального утверждения и его отрицания, которые могут быть доказаны вместе, что в свою очередь, демонстрировало бы непротиворечивость математики. В программу Гильберта была включена и вера в то, что процесс вывода теорем может быть механизирован, или, говоря по-современному, что математические теории разрешимы.

Рассмотренные в параграфах 8.1 и 8.2 знаменитые теоремы К.Гёделя о неполноте формальной арифметики и А.Чёрча о неразрешимости логики предикатов, полученные в начале и в середине 30-ых годов XX века, разбили все надежды на реализацию программы Гильберта в её первоначальной форме. А именно, К.Гёдель продемонстрировал невозможность доказательства непротиворечивости любой значительной части математики посредством финитных методов, которые отстаивал Гильберт, а А.Чёрч доказал, что исчисление предикатов, как и арифметика сложения и умножения натуральных чисел, неразрешимы. Эти результаты открыли перспективу изучения проблемы разрешимости и вызвали массу исследований о разрешимости и неразрешимости различных математических теорий, о некоторых из которых говорилось в настоящей главе.

Алгоритмическая сложность разрешимых математических теорий. В 60-70-ые годы XX в. проблема разрешимости аксиоматических теорий приобрела дополнительную окраску: математики обратились к вопросу о вычислительной сложности алгоритмически разрешимых проблем разрешимости. И здесь выяснилось, что многие аксиоматические теории, хотя они теоретически и разрешимы, с практической точки зрения неразрешимы, потому что любой разрешающий алгоритм требовал бы практически невозможного числа вычислительных шагов. Так, в частности, для арифметики сложения натуральных чисел, разрешимость которой в 1929 г. была доказана М.Пресбургером, в 1974 г. было доказано, что для каждого разрешающего алгоритма $\mathbf{A}$ существует предложение P размера (т.е. с числом символов) n такое, что для $\mathbf{A}$ потребуется 2^{2^n} вычислительных шагов для ответа на вопрос, является ли P теоремой этой теории. Аналогично было показано, что элементарная геометрия, разрешимость которой доказана А.Тарским в 1951 г., с алгоритмической точки зрения является экспоненциально сложной. Таковыми же являются теория линейно упорядоченных множеств и ряд других аксиоматических теорий первого порядка.

Эти и подобные им результаты подвергают сомнению утверж-

дение о том, что якобы любая теория, о которой доказано, что она разрешима, в определённом смысле тривиальна, так как её теоремы можно было бы выявить с помощью вычислительной программы. Вычисления, требующие, скажем $2^{2^{30}}$ шагов, нельзя рассматривать как практический метод подтверждения доказуемости математического утверждения.

В связи с этим возникает естественный вопрос, существуют ли аксиоматические теории с *практически* разрешимыми проблемами разрешимости. Ответ на этот фундаментальный вопрос пока неизвестен. Ясно только, что проблема разрешимости любой формализованной теории по меньшей мере так же сложна, как и проблема разрешимости (выполнимости) для формул алгебры высказываний, о которой говорилось в § 8.3. Там же мы отметили, что, как доказал С.Кук, многие алгоритмические проблемы разрешимости, которые не поддались попыткам создания эффективной, а не экспоненциально сложной разрешающей процедуры, сводятся к проблеме выполнимости для формул алгебры высказываний. Эти факты заставляют верить, что проблема выполнимости для формул алгебры высказываний экспоненциально сложна, а вместе с ней экспоненциально сложны и проблемы разрешимости любых формализованных теорий.

Список литературы

Математическая логика с теорией алгоритмов

1. *Гладкий А.В.* Математическая логика. - М., 1998.

2. *Глухов М.М., Козлитин О.А., Шапошников В.А., Шишков А.Б.* Задачи и упражнения по математической логике, дискретным функциям и теории алгоритмов. - СПб и др.: Лань, 2008. - 112 с.

3. *Гуц А.К.* Математическая логика и теория алгоритмов. - Омск: ОГУ, 2003.

4. *Ершов Ю.Л., Палютин Е.А.* Математическая логика. - М.: Наука, 1979.

5. *Игошин В.И.* Математическая логика и теория алгоритмов. - Саратов: Изд-во СГУ, 1991. - 256 с.

6. *Игошин В.И.* Математическая логика и теория алгоритмов. - М.: Издательский центр "Академия", 2004, 2008, 2010. - 448 с.

7. *Игошин В.И.* Задачник-практикум по математической логике. - М.: Просвещение, 1986. - 160 с.

8. *Игошин В.И.* Задачи и упражнения по математической логике и теории алгоритмов. - М.: Издательский центр "Академия", 2005, 2006, 2007, 2008. - 304 с.

9. *Клини С.* Введение в метаматематику / Пер. с англ. - М., 1957.

10. *Клини С.* Математическая логика / Пер. с англ. - М., 1973.

11. *Колмогоров А.Н., Драгалин А.Г.* Математическая логика. - М.: Едиториал УРСС, 2004.

12. *Лавров И.А., Максимова Л.Л.* Задачи по теории множеств, математической логике и теории алгоритмов. - М.: Наука, 1975; 1984. Физматлит, 1995.

13. *Мендельсон Э.* Введение в математическую логику / Пер. с англ. - М.: Наука, 1971; 1976. - 320 с.

14. *Набедин А.А., Кораблёв Ю.П.* Математическая логика и теория алгоритмов. - М.: Научный мир, 2008. - 344 с.

15. *Нагель Э., Ньюмен Дж.Р.* Теорема Гёделя / Пер. с англ. - М.: Знание, 1970. - 64 с.

16. *Паршин А.Н.* Размышления над теоремой Гёделя // Вопросы философии, 2000, № 6, с. 92 - 109.

17. *Судоплатов С.В., Овчинникова Е.В.* Математическая логика и теория алгоритмов. - М.: ИНФРА-М; Новосибирск: Изд-во НГТУ, 2004. - 224с.

18. *Успенский В.А.* Теорема Гёделя о неполноте. - М.: Наука, 1982.

19. *Успенский В.А., Верещагин Н.К., Плиско В.Е.* Вводный курс математической логики. - М.: Физматлит, 2004. - 128 с.

20. *Цейтин Г.С.* Ассоциативное исчисление с неразрешимой проблемой эквивалентности // Тр. Матем. ин-та АН СССР, 1958, 52, с. 172 - 189.

21. *Шапорев С.Д.* Математическая логика и теория алгоритмов. - СПб, 2004.

Дискретная математика с теорией алгоритмов

22. *Ван дер Варден Б.Л.* Алгебра. - М.: Наука, 1976. - 648 с.

23. *Горбатов В.А., Горбатова А.В., Горбатова М.В.* Дискретная математика: Учебник для студентов втузов. - М.: АСТ, 2003. - 447 с.

24. *Ерусалимский Я.М.* Дискретная математика. - М.: Вузовская книга, 2002. - 266 с.

25. *Кузнецов О.П., Адельсон-Вельский Г.М.* Дискретная математика для инженера. 2-е изд., перераб. и доп. - М.: Энергия, 1988. - 480 с. [1-е изд. - 1980. - 344 с.]

26. *Плотников А.Д.* Дискретная математика. - М.: Новое знание, 2005. - 288 с.

27. *Салий В.Н.* Математические основы гуманитарных знаний. - М.: Высшая школа, 2009. - 304 с.

Теория алгоритмов и её приложения

28. *Адян С.И., Дурнев В.Г.* Алгоритмические проблемы для групп и полугрупп // Успехи матем. наук, 2000, т. 55, вып. 2 (332), с. 3 - 94.

29. *Агафонов В.Н.* Сложность алгоритмов и вычислений. - Новосибирск: Изд-во НГУ, 1975. - 146 с.

30. *Алфёрова З.В.* Теория алгоритмов. - М.: Статистика, 1973. - 164 с.

31. *Андерсон Р.* Доказательство правильности программ / Пер. с англ. - М.: Мир, 1982. - 287 с.

32. *Арбиб М.* Мозг, машина и математика / Пер. с англ. - М., 1968.

33. *Ахо А., Хопкрофт Дж., Ульман Дж.* Построение и анализ вычислительных алгоритмов / Пер. с англ. - М.: Мир, 1979. - 536 с.

34. *Белов А., Тихомиров В.* Сложность алгоритмов // Квант, 1999, № 2, с. 8 - 11.

35. *Булос Дж., Джеффри Р.* Вычислимость и логика / Пер. с англ. - М.: Мир, 1994. - 396 с.

36. *Баженин Ю.М.* Алгоритмы и алгебра. - Свердловск: Изд-во Урал. гос. ун-та, 1988. - 76 с.

37. *Варпаховский Ф.Л.* Элементы теории алгоритмов. - М., 1979. - 24 с.

38. *Варпаховский Ф.Л., Колмогоров А.Н.* О решении 10-ой проблемы Гильберта // Квант, 1970, № 7, с. 38 - 44.

39. *Верещагин Н.К., Шень А.* Лекции по математической логике и теории алгоритмов. Ч.3: Вычислимые функции. - М.: МЦНМО, 1999. - 173 с.; 2002. - 192 с.

40. *Вирт Н.* Алгоритмы и структура данных. - М., 1989.

41. *Воротников С.М.* Основы теории алгоритмов и рекурсивных функций. - Комсомольск-на Амуре: Комс.-на Амуре гос. техн. ун-т, 2007. - 121 с.

42. *Вялый М.Н.* Сложность вычислительных задач // Математическое просвещение. Серия 3, вып. 4. - М.: МЦНМО, 2000, с. 81 - 114.

43. *Габович И.Г.* Алгоритмический подход к решению геометрических задач. - М.: Просвещение, 1995.

44. *Гашков С.Б., Чубариков В.Н.* Арифметика. Алгоритмы. Сложность вычислений. - М.: Высшая школа, 2000. - 320 с.

45. *Гинзбург С.* Математическая теория контекстно-свободных языков. - М.: Мир, 1970.

46. *Глушков В.М.* Теория алгоритмов. - Киев: КВИРТУ, 1961. - 168 с.

47. *Глушков В.М.* Теорема о неполноте формальных теорий с позиций программиста // Кибернетика, 1979, № 2, с. 1 - 5.

48. *Грин Д., Кнут Д.* Математические методы анализа алгоритмов / Пер. с англ. - М.: Мир, 1987. - 119 с.

49. *Гэри М., Джонсон Д.* Вычислительные машины и труднорешаемые задачи / Пер. с англ. - М.: Мир, 1982. - 416 с.

50. *Девис М.* Неразрешимые проблемы / Справочная книга по математической логике: В 4-х частях / Под ред. Дж. Барвайса. - Ч.III. Теория рекурсии: Пер. с англ. - М.: Наука, 1982. С. 51 - 76.

51. *Ершов А.П.* Введение в теоретическое программирование. - М., 1977.

52. *Ершов А.П.* На родине великого учёного // Квант, 1984, № 8, с. 29 - 31.

53. *Ершов А.П.* Избранные труды. - Новосибирск: Наука, 1994. - 416 с.

54. *Ершов Ю.Л.* Теория нумераций. - М.: Наука, 1977.

55. *Ершов Ю.Л., Лавров И.А., Тайманов А.Д., Тайцлин М.А.* Элементарные теории // Успехи матем. наук, 1965, т. 20, № 4, с. 37 - 108.

56. *Ершов Ю.Л.* Проблемы разрешимости и конструктивные модели. - М.: Наука, 1980. - 416 с.

57. *Игошин В.И.* Основы теории алгоритмов. - Саратов: Издательский центр "Наука", 2008. - 96 с.

58. *Ильиных А.П.* Теория алгоритмов. - Екатеринбург: Уральский гос. пед. университет, 2006. - 149 с.

59. *Карп Р.М.* Сводимость комбинаторных проблем // Кибернетический сборник, вып. 12. - М.: Мир, 1975. С. 16 -38.

60. *Катленд Н.* Вычислимость. Введение в теорию рекурсивных функций. - М.: Мир, 1983.

61. *Кожевникова Г.П.* Теория алгоритмов. - Львов: Вища шк., 1978. - 98 с.

62. *Колмогоров А.Н.* Алгоритм, информация, сложность. - М.: Знание, 1991. - 43 с.

63. *Колмогоров А.Н., Успенский В.А.* К определению алгоритма // Успехи матем. наук, 1958, № 4 (13), с. 3 - 28.

64. *Кормен Т., Лейзерсон Ч., Ривест Р.* Алгоритмы: построение и анализ. - М.: МЦНМО, 2001.

65. *Королёв Л.Н., Миков А.И.* Информатика. Введение в компьютерные науки. - М.: Высшая школа, 2003. - 342 с.

66. *Косовский Н.К.* Основы теории элементарных алгоритмов. - Л.: Изд-во ЛГУ, 1987. - 152 с.

67. *Котова А.* Машина Тьюринга // Квант, 1992, № 7, с. 60 - 62.

68. *Криницкий Н.А.* Алгоритмы вокруг нас. - М.: Наука, 1984.

69. *Крупский В.Н.* Введение в сложность вычислений. - М.: Факториал-Пресс, 2006. - 128 с.

70. *Крупский В.Н., Плиско В.Е.* Теория алгоритмов. – М.: Издательский центр "Академия", 2009.

71. *Кудрявская Н., Ломакина И., Приз С.* Машина Поста // Квант, 1972, № 5.

72. *Ланда Л.Н.* Алгоритмизация в обучении. – М.: Просвещение, 1966. – 524 с.

73. *Лапшин В.А.* Лекции по математической лингвистике. - М.: Научный мир, 2010. - 248 с.

74. *Лисовенко Н.Н., Володина И.В.* Алгоритмизация и алгоритмические языки. Ч.1. - Днепропетровск, 1974. - 47 с.

75. *Макаренков Ю.А., Столяр А.А.* Что такое алгоритм? – Минск, 1989.

76. *Макконелл Дж.* Основы современных алгоритмов: учеб. пособие; пер. с англ. - 2-ое изд., доп. - М.: Техносфера, 2006. - 366 с.

77. *Мальцев А.И.* Алгоритмы и рекурсивные функции. - М.: Наука, 1965; 1986.

78. *Манин Ю.И.* Вычислимое и невычислимое. - М.: Советское радио, 1980.

79. *Марков А.А., Нагорный Н.М.* Теория алгорифмов. - М.: Наука, 1984. - 432 с.

80. *Марченков С.С.* Элементарные рекурсивные функции. - М.: МЦНМО, 2003. - 111 с.

81. *Матиясевич Ю.В.* Диофантовость перечислимых множеств // ДАН СССР, 1970, 191, № 2, с. 279 - 282.

82. *Матиясевич Ю.В.* Диофантовы множества // Успехи матем. наук, 1972, 27: 5 (167), 185 - 222.

83. *Матиясевич Ю.В.* Десятая проблема Гильберта. - М.: Физматлит, 1993.

84. *Матрос Д.Ш., Поднебесова Г.Б.* Теория алгоритмов. - М.: БИНОМ. Лаборатория знаний, 2008. - 202 с.

85. *Матросов В.Л.* Теория алгоритмов. - М.: Прометей (МГПИ им. В.И.Ленина), 1989. - 188 с.

86. Машины Тьюринга и рекурсивные функции / Пер. с нем. - М.: Мир, 1972. - 264 с.

87. *Минский М.* Вычисления и автоматы. - М.: Мир, 1971. - 364 с.

88. *Нагель Э., Ньюмен Дж.Р.* Теорема Гёделя / Пер. с англ. - М.: Знание, 1970. - 64 с.

89. *Непомнящий В.А., Рякин О.М.* Прикладные методы верификации программ. - М.: Радио и связь, 1988. - 256 с.

90. *Носов В.А.* Основы теории алгоритмов и анализа их сложности. - М.: Изд-во МГУ, 1992.

91. *Одинец В.П., Поспелов М.В.* Введение в теорию алгоритмов. – Сыктывкар: Изд-во Коми пед. ин-та, 2006. – 141 с.

92. *Пападимитриу Х., Стайглиц К.* Комбинаторная оптимизация. Алгоритмы и сложность / Пер. с англ. – М.: Мир, 1985. - 512 с.

93. *Петер Р.* Рекурсивные функции / Пер. с нем. - М.: ИЛ, 1954. – 264 с.

94. *Поляков Е.А.* Теория алгоритмов. - Иваново: Изд-во ИВГУ, 1976. - 88 с.

95. *Рабин М.О.* Разрешимые теории / Справочная книга по математической логике: В 4-х частях / Под ред. Дж. Барвайса. - Ч.III. Теория рекурсии: Пер. с англ. - М.: Наука, 1982. С. 77 - 111.

96. *Разборов А.А.* О сложности вычислений // Математическое просвещение. Серия 3, вып. 3. - М.: МЦНМО, 1999, с. 127 - 141.

97. *Рейуорд-Смит В.Дж.* Теория формальных языков: Вводный курс / Пер. с англ. - М.: Радио и связь, 1988. - 127 с.

98. *Роджерс Х.* Теория рекурсивных функций и эффективная вычислимость / Пер. с англ. - М.: Мир, 1972.

99. *Рублёв В.С.* Основы теории алгоритмов. - М.: Издательство Научный Мир, 2008. - 136 с.

100. *Семёнов А.Л., Успенский В.А.* Математическая логика в вычислительных науках и вычислительной практике // Вестник АН СССР, 1986, № 7, с. 93 - 103.

101. Сложность вычислений и алгоритмов. Сб. переводов. - М.: Мир, 1974. - 389 с.

102. *Смейл С.* О проблемах вычислительной сложности // Математическое просвещение. Серия 3, вып. 4. - М.: МЦНМО, 2000, с. 115 - 119.

103. *Трахтенброт Б.А.* Сложность алгоритмов и вычислений. - Новосибирск: Изд-во НГУ, 1967. - 258 с.

104. *Трахтенброт Б.А.* Алгоритмы и вычислительные автоматы. - М.: Советское радио, 1974. - 200 с.

105. *Тьюринг А.* Может ли машина мыслить ? - М.: Физматлит, 1960.

106. *Успенский В.А.* Лекции о вычислимых функциях. - М.: Физматгиз, 1960. - 492 с.

107. *Успенский В.А.* Машина Поста. - М.: Наука, 1979, 1988.

108. *Успенский В.А., Семёнов А.Л.* Решимые и нерешимые алгоритмические проблемы // Квант, 1985, № 7, с. 9 - 15.

109. *Успенский В.А., Семёнов А.Л.* Теория алгоритмов: основные открытия и приложения. - М.: Наука, 1987. - 288 с.

110. *Фалевич Б.Я.* Теория алгоритмов. - М.: Машиностроение, 2004. - 160 с.

Оглавление

Предисловие 3

Глава I. НЕФОРМАЛЬНОЕ (ИНТУИТИВНОЕ) ПРЕДСТАВЛЕНИЕ ОБ АЛГОРИТМАХ 6

1.1. Алгоритмы в жизни и в математике 6

Алгоритмы в жизни. Алгоритмы в математике. Алгоритм Евклида.

1.2. Неформальное понятие алгоритма и необходимость его уточнения 9

Неформальное понятие алгоритма. Необходимость уточнения понятия алгоритма.

Глава II. МАШИНЫ ТЬЮРИНГА И ВЫЧИСЛИМЫЕ ПО ТЬЮРИНГУ ФУНКЦИИ 14

2.1. Понятие машины Тьюринга и применение машин Тьюринга к словам 14

Определение машины Тьюринга. Применение машин Тьюринга к словам. Конструирование машин Тьюринга.

2.2. Примеры машин Тьюринга 22

Прибавление единицы. Перенос нуля. Левый сдвиг. Правый сдвиг. Транспозиция. Удвоение.

2.3. Композиция машин Тьюринга 26

Понятие композиции машин Тьюринга. Применение композиции машин Тьюринга для их конструирования. Циклический сдвиг. Копирование.

2.4. Вычислимые по Тьюрингу функции 28

Алгоритмы, функции и машины Тьюринга. Вычислимость функций на машине Тьюринга. Правильная вычислимость функций на

машине Тьюринга. Нуль-функция. Функции-проекторы. Вычисление сложных функций на машинах Тьюринга. Тезис Тьюринга (основная гипотеза теории алгоритмов). Машины Тьюринга и современные электронно-вычислительные машины.

Глава III. РЕКУРСИВНЫЕ ФУНКЦИИ 41

3.1. Начало теории рекурсивных функций 41

Происхождение рекурсивных функций. Простейшие функции. Оператор суперпозиции. Оператор примитивной рекурсии. Примеры других схем рекурсии (не примитивных). Понятие примитивно рекурсивной функции. Оператор минимизации. Понятие частично рекурсивной и общерекурсивной функции. Тезис Чёрча (основная гипотеза теории рекурсивных функций).

3.2. Примитивно рекурсивные функции 48

Примеры примитивно рекурсивных функций. Примитивная рекурсивность булевых функций. Операторы суммирования и мультиплицирования (умножения).

3.3. Примитивно рекурсивные предикаты 56

Понятие примитивно рекурсивного предиката. Примеры примитивно рекурсивных предикатов. Логические операции с примитивно рекурсивными предикатами. Ограниченные кванторы общности и существования. Примеры примитивно рекурсивных предикатов. Оператор условного перехода. (Кусочное задание функций). Оператор ограниченной минимизации. (Ограниченный μ-оператор). Дальнейшие примеры примитивно рекурсивных функций.

3.4. Примитивно рекурсивные функции и функции, вычислимые по Тьюрингу 67

Вычислимость по Тьюрингу примитивно рекурсивных функций. Функции Аккермана.

3.5. Частично рекурсивные функции и функции, вычислимые по Тьюрингу 73

Оператор минимизации. Общерекурсивные и частично рекурсивные функции. Вычислимость по Тьюрингу частично рекурсивных функций. Частичная рекурсивность функций вычислимых по Тьюрингу.

Глава IV. НОРМАЛЬНЫЕ АЛГОРИТМЫ МАРКОВА 84

4.1. Марковские подстановки, нормальные алгоритмы и нормально вычислимые функции 84

Марковские подстановки. Нормальные алгоритмы и их применение к словам. Нормально вычислимые функции. Принцип нормализации Маркова.

4.2. Рекурсивные функции и нормально вычислимые функции 97

Совпадение класса всех частично рекурсивных функций с классом всех нормально вычислимых функций. Совпадение класса всех нормально вычислимых функций с классом всех функций, вычислимых по Тьюрингу. Эквивалентность различных теорий алгоритмов.

Глава V. ОБЩИЙ ПОДХОД К ТЕОРИИ АЛГОРИТМОВ 102

5.1. Нумерации алгоритмов и вычислимых функций 102

Предварительные соображения. Вычислимые функции. Разрешимые предикаты и разрешимые алгоритмические проблемы. Нумерация (перечисление) алгоритмов. Нумерация машин Тьюринга. Нумерация вычислимых функций. Диагональный метод в теории алгоритмов.

5.2. Теорема о параметризации и универсальные функции и алгоритмы 114

Теорема о параметризации (s-m-n-теорема Клини). Универсальные функции и алгоритмы. Главные универсальные функции (главные нумерации). Эффективные операции над вычислимыми функциями.

5.3. Теорема о неподвижной точке и её применения 123

Теорема о неподвижной точке. Следствия из теоремы о неподвижной точке. Неформальные интерпретации теоремы о неподвижной точке. Пример применения теоремы о неподвижной точке: существование алгоритма, печатающего свой собственный текст.

Глава VI. РАЗРЕШИМОСТЬ И ПЕРЕЧИСЛИМОСТЬ МНОЖЕСТВ 128

Происхождение проблемы.

6.1. Разрешимые множества и их свойства 129

Понятие разрешимого множества и примеры. Свойства разрешимых множеств.

6.2. Перечислимые множества и их свойства 131

Понятие перечислимого множества и примеры. Свойства перечислимых множеств.

6.3. Взаимоотношения между разрешимыми и перечислимыми множествами 132

Разрешимые и перечислимые множества. Канторовская нумерация упорядоченных пар натуральных чисел. Существование перечислимого, но не разрешимого множества. Разрешимость и перечислимость: итог.

6.4. Связь разрешимых и перечислимых множеств с вычислимыми функциями 139

Перечислимые множества и вычислимые функции. Ещё один пример перечислимого, но не разрешимого множества. Примеры множеств, не являющихся перечислимыми. Ещё одна характеристика перечислимых множеств. Перечислимые множества как проекции разрешимых бинарных отношений. График вычислимой функции.

Глава VII. АЛГОРИТМИЧЕСКИ НЕРАЗРЕШИМЫЕ МАССОВЫЕ ПРОБЛЕМЫ 150

7.1. Алгоритмически неразрешимые проблемы, связанные с машинами Тьюринга 151

Существование не вычислимых по Тьюрингу функций. Пример не вычислимой по Тьюрингу функции. Проблема распознавания самоприменимости. Проблема распознавания применимости.

7.2. Алгоритмически неразрешимые проблемы в общей теории алгоритмов 154

Проблема распознавания самоприменимости алгоритмов. Проблема распознавания применимости (или остановки) алгоритмов. Проблема распознавания нулевых функций. Проблема распознавания равенства двух вычислимых функций. Проблема распознавания общерекурсивных (т.е. всюду определённых вычислимых) функций. Не всякая частичная вычислимая функция может быть доопределена до всюду определённой вычислимой функции. Существование

перечислимого, но не разрешимого множества. Проблема распознавания нетривиальных свойств вычислимых функций. Дальнейшие примеры алгоритмически неразрешимых массовых проблем. О значении алгоритмически неразрешимых массовых проблем для теории алгоритмов и практики программирования.

7.3. Частично разрешимые предикаты и частично разрешимые алгоритмические проблемы 170

Частично разрешимые предикаты. Связь частично разрешимых предикатов с разрешимыми предикатами. Признак вычислимости частичной функции.

Глава VIII. СЛОЖНОСТЬ ВЫЧИСЛЕНИЙ И МАССОВЫХ ПРОБЛЕМ 180

8.1. Как измерять сложность вычислительных задач и массовых проблем 181

Единичные вычислительные задачи и массовые проблемы. Вычислительная сложность единичной задачи. Сложность массовых проблем.

8.2. Сравнение и классификация массовых проблем и алгоритмов по их сложности 194

Концепция сравнения массовых проблем и алгоритмов по их сложности. Сложностные классы массовых проблем.

8.3. Основы теории NP-полных массовых проблем 201

Алгоритмическая сводимость массовых проблем. Проблемы распознавания. Формальные языки и грамматики. Проблемы распознавания и формальные языки. Детерминированные алгоритмы и классы **P**. Недетерминированные алгоритмы и класс **NP**. Взаимоотношения между классами **P** и **NP**. Полиномиальная сводимость формальных языков массовых проблем распознавания. Класс NP-полных языков и NP-полных массовых проблем распознавания. NP-полнота проблемы выполнимости для формул логики высказываний. Примеры NP-полных массовых проблем. NP-полнота проблемы 3-ВЫПОЛНИМОСТЬ. NP-полные проблемы и труднорешаемые проблемы.

Глава IX. АЛГОРИТМИЧЕСКИЕ ПРОБЛЕМЫ МАТЕМАТИЧЕСКОЙ ЛОГИКИ 252

9.1. Теоремы К.Гёделя и А.Тарского о формальной арифметике 252

Формальные аксиоматические теории и натуральные числа. Формальная арифметика и её свойства. Доказательство теоремы Гёделя о неполноте. Ещё один взгляд на теорему Гёделя о неполноте. Вторая теорема К.Гёделя. Теорема А.Тарского. Теорема Гёделя о неполноте формальной арифметики с позиций программиста.

9.2. Неразрешимость формализованного исчисления предикатов 273

Идея доказательства. Построение для данной машины Тьюринга множества формул Δ и формулы H.

Глава X. АЛГОРИТМИЧЕСКИЕ ПРОБЛЕМЫ МАТЕМАТИКИ 287

10.1. Десятая проблема Д.Гильберта 287

История проблемы. Диофантовы множества и их связь с перечислимыми множествами. Решение 10-ой проблемы Гильберта. Следствия из теоремы Ю.В.Матиясевича. 10-ая проблема Гильберта с точки зрения теории предикатов. Две модификации 10-ой проблемы Гильберта.

10.2. Проблема тождества слов и другие математические проблемы 296

Истоки проблемы тождества слов. Проблема слов в теории полугрупп. Алгоритмические проблемы слов в теории групп. Проблема нахождения нулей многочлена. Алгоритмические проблемы в других разделах математики.

10.3. О дальнейших направлениях изучения алгоритмических проблем логики и математики 303

Разрешимые математические теории и программа Гильберта. Алгоритмическая сложность разрешимых математических теорий.

СПИСОК ЛИТЕРАТУРЫ 306

По вопросам приобретения книг обращайтесь:
Отдел продаж «ИНФРА-М» (оптовая продажа):
127282, Москва, ул. Полярная, д. 31в, стр. 1
Тел. (495) 380-4260; факс (495) 363-9212
E-mail: books@infra-m.ru

•

Отдел «Книга—почтой»:
тел. (495) 363-4260 (доб. 232, 246)

Учебное издание

Владимир Иванович Игошин

ТЕОРИЯ АЛГОРИТМОВ

УЧЕБНОЕ ПОСОБИЕ

Подписано в печать 25.08.2011.
Формат 60×90/16. Бумага офсетная. Гарнитура Times.
Усл. печ. л. 20,0. Уч.-изд. л. 18,82.
Тираж 1000 экз. Заказ № 6718.
ТК 163100-10108-250811

Издательский Дом «ИНФРА-М»
127282, Москва, ул. Полярная, д. 31в
Тел.: (495) 380-05-40, 380-05-43. Факс: (495) 363-92-12.
E-mail: books@infra-m.ru
http://www.infra-m.ru

Отпечатано с электронных носителей издательства.
ОАО «Тверской полиграфический комбинат». 170024, г. Тверь, пр-т Ленина, 5.
Телефон: (4822) 44-52-03, 44-50-34, Телефон/факс: (4822) 44-42-15.
Home page – www.tverpk.ru Электронная почта (E-mail) sales@tverpk.ru

ПЕЧАТЬ
ПО ТРЕБОВАНИЮ (Book-on-demand)
ДЛЯ АВТОРОВ

Хотите издать и продавать научную монографию, роман, детскую книгу или книгу любого другого жанра?

Добро пожаловать в компанию, которая поможет Вам опубликовать свои произведения!

Мы поможем осуществить Вашу мечту и сделать книгу доступной покупателям книжных магазинов и любому пользователю Интернета!

РИОР предоставляет Вам, уважаемые авторы, возможность издания книги с учетом Ваших потребностей любым тиражом, начиная с одного экземпляра!

Наши специалисты помогут Вам выпустить в свет и начать продавать Вашу книгу и готовы провести Вас по всему книгоиздательскому процессу.

Назовем лишь некоторые преимущества издания книги по технологии «печать по требованию»:

- больше не нужно печатать огромные тиражи, печатайте столько, сколько нужно;
- возможность самостоятельно установить цену книги;
- Вашей книге присваивается международный книжный номер ISBN, что дает возможность свободно продавать книгу;
- мы обеспечиваем розничную и оптовую продажу;
- продажа Вашей книги через Интернет (это огромная аудитория!);
- Ваш гонорар – до 30 процентов с продажи каждого экземпляра.

С чего начать?

Мы предлагаем весь спектр издательских услуг – от простого размещения электронной версии Вашей книги на сайте нашего интернет-магазина (с возможностью ее напечатать при поступлении заказа в любом количестве) вплоть до редактирования Вашего текстового оригинала, полной подготовки оригинал-макета и маркетинговых услуг.

Мы готовы помочь определить, что Вам нужно и что из этого следует выбрать, подробно рассказать о каналах распределения Вашей книги и ответить на все «денежные» вопросы: о Вашем гонораре, о скидках и т.д.

Ознакомьтесь с более подробным обзором всех наших услуг, и, быть может, уже сегодня Вы захотите напечатать Вашу книгу!

www.rior.ru
Телефон: (495) 363•92•15

В.И. Игошин

Теория алгоритмов